1982

# Understanding and Documenting Programs

V. R. Basili

Harlan D. Mills

# Understanding and Documenting Programs

VICTOR R. BASILI AND HARLAN D. MILLS

*Abstract*—This paper reports on an experiment in trying to understand an unfamiliar program of some complexity and to record the authors' understanding of it. The goal was to simulate a practicing programmer in a program maintenance environment using the techniques of program design adapted to program understanding and documentation; that is, given a program, a specification and correctness proof were developed for the program. The approach points out the value of correctness proof ideas in guiding the discovery process. Toward this end, a variety of techniques were used: direct cognition for smaller parts, discovering and verifying loop invariants for larger program parts, and functions determined by additional analysis for larger program parts. An indeterminate bounded variable was introduced into the program documentation to summarize the effect of several program variables and simplify the proof of correctness.

*Index Terms*—Program analysis, program correctness, program documentation, proof techniques, software maintenance.

## I. Introduction

### *Understanding Programs*

We REPORT here on an experiment in trying to understand an unfamilar program of some complexity and to record our understanding of it. We are as much concerned with recording our understanding as with understanding. Every day programmers are figuring out what existing programs do more or less accurately. But most of this effort is lost, and repeated over and over, because of the difficulty of capturing this understanding on paper. We want to demonstrate that the very techniques of good program design can be adapted to problems of recording hard-won understandings about existing programs.

In program design we advocate the joint development of design and correctness proof, as shown in [2], [4], [6], rather than *a posteriori* proof development. Nevertheless, we believe that the idea of program correctness provides a comprehensive *a posteriori* strategy for developing and recording an understanding of an existing program. In fact, we advocate another kind of joint development, this time, of specification and correctness proof. In this way, we have a consistent approach dealing always with three objects, namely, 1) a specification, 2) a program, and 3) a correctness proof. In writing a program, we are given 1) and develop 2) and 3) jointly; in reading a program, we are given 2) and develop 1) and 3) jointly. In either case, we end up with the same harmonious arrangement of 1) and 2) connected by 3) which contains our understanding of the program.

Manuscript received June 10, 1980; revised September 3, 1981 and November 6, 1981. This work was supported in part by the U.S. Air Force Office of Scientific Research under Grant AFOSR-77-3181B.

V. R. Basili is with the Department of Computer Science, University of Maryland, College Park, MD 20742.

H. D. Mills is with the Department of Computer Science, University of Maryland, College Park, MD 20742 and the Federal Systems Division, IBM Corporation, Bethesda, MD 20034.

In the experiment at hand, our final understanding exceeded our most optimistic initial expectations, even though we have seen these ideas succeed before. One new insight from this experiment was how little we really had to know about the program to develop a complete understanding and proof of what it does (in contrast to how it does it). Without the correctness proof ideas to guide us, we simply would not have discovered how little we had to know. In fact, we know a great deal more than we have recorded here about how the program works, which we chalk up to the usual dead ends of a difficult discovery process. But the point is, without the focus of a correctness proof, we would still be trying to understand and record a much larger set of logical facts about the program than is necessary to understand precisely what it does.

In retrospect, we used a variety of discovery techniques. For simpler parts of the program, we used direct cognition. In small complex looping parts, we discovered and verified loop invariants. In the large, we organized the effect of major program parts as functions to be determined by additional analysis. We also discovered a new way to express the effect of a complex program part by introducing a bounded indeterminate variable which radically simplified the proof of correctness of the program part.

### *The Program*

We were interested in a short but complex program. Our goal was to simulate a practicing programmer in a program maintenance environment. The program was chosen by Prof. J. Vandergraft of the University of Maryland as a difficult program to understand. It was a Fortran program called ZEROIN which claimed to find a zero of a function given by a Fortran subroutine. We were given the program and told its general function. The problem then was to understand it, verify its correctness, and possibly modify it, to make it more efficient or extend its applicability. We were not given any more about the program than the program itself. The program given to us is shown in Fig. 1, the original Fortran ZEROIN. Prof. Vandergraft played the role of a user of the program and posed four questions regarding the program.

1) I have a lot of equations, some of which might be linear. Should I test for linearity and then solve the equation directly, or just call ZEROIN? That is, how much work does ZEROIN do to find a root of a linear function?

2) What will happen if I call ZEROIN with F(AX) and F(BX) both positive? How should the code be changed to test for this condition?

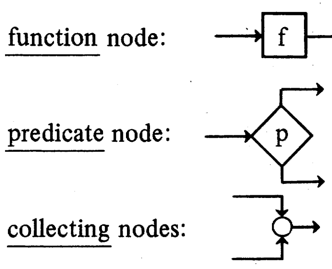
3) It is claimed that the inverse quadratic interpolation saves only 0.5 function evaluations on the average. To get a shorter program, I would like to remove the inverse quadratic interpolation part of the code. Can this be done easily? How?

4) Will ZEROIN find a triple root?

## II. Techniques for Understanding Programs

### *Flowcharts*

Any flowchartable program can be analyzed in a way we describe next for better understandability and documentation. For a fuller discussion, see [6]. We consider flowcharts as directed graphs with nodes and lines. The lines denote flow of control and the nodes denote tests and operations on data. Without loss of generality, we consider flowcharts with just three types of nodes, namely,



where f is any function mapping the data known to the program to new data, e.g., a simple Fortran assignment statement, and p is any predicate on the data known to the program, e.g., a simple Fortran test. An *entry line* of a flowchart program is a line adjacent to only one node, its head; an *exit line* is adjacent to only one node, its tail.

### *Functions and Data Assignments*

Any function mapping the data known to a program to new data can be defined in a convenient way by generalized forms of data assignment statements. For example, an *assignment*, denoted

x := e (e.g., x := x + y)

where x is a variable known to the program and e is an expression in variables known to the program, means that the value of e is assigned to x. Such an assignment also means that no variable except x is to be altered. The *concurrent assignment*, denoted

x1, x2, · · · , xn := e1, e2, · · · , en

means that expressions e1, e2, · · · , en are evaluated independently, and their values assigned simultaneously to x1, x2, · · · , xn, respectively. As before, the absence of a variable on the left side means that it is unchanged by the assignment.

The *conditional assignment*, denoted

(p1 → A1 | p2 → A2 | · · · | pn → An)

where p1, p2, · · · , pn are predicates and A1, A2, · · · , An are assignments (simple, concurrent, or conditional) means that particular assignment Ai associated with the first pi, if any, which evaluates true; otherwise, if no pi evaluates true, then the conditional assignment is undefined.

An expression in an assignment may contain a function value, e.g.,

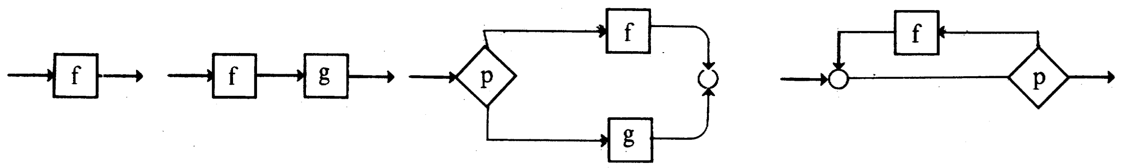
x := max (x, abs(y))

where max and abs are functions. But the function defined by the assignment statement is different, of course, from max or abs.

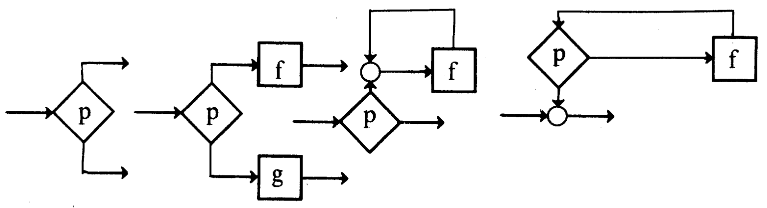
We note that many programming languages permit the possibility of so-called side effects, which alter data not mentioned in assignment statements or in tests. Side effects are specifically prohibited in our definition of assignments and tests.

### *Proper Programs*

We define a *proper program* to be a program whose flowchart has exactly one entry line, one exit line, and, further, for every node a path from the entry through that node to the exit. For example,



are proper programs, but



are not proper programs.

```
***** ZEROIN.PROGRAM *****

      REAL FUNCTION ZEROIN (AX, BX, F, TOL, IP)
      REAL AX, BX, F, TOL
C
C
      REAL A, B, C, D, E, EPS, FA, FB, FC, TOL1, XM, P, Q, R, S
C
C  COMPUTE EPS, THE RELATIVE MACHINE PRECISION
C
      EPS = 1.0
   10 EPS = EPS/2.0
      TOL1 = 1.0 + EPS
      IF (TOL1 .GT. 1.0) GO TO 10
C
C  INITIALIZATION
C
        IF (IP .EQ. 1) WRITE (6, 11)
 11     FORMAT(' THE INTERVALS DETERMINED BY ZEROIN ARE')
      A = AX
      B = BX
      FA = F(A)
      FB = F(B)
C
C  BEGIN STEP
C
   20 C = A
      FC = FA
      D = B - A
      E = D
 30     IF (IP .EQ. 1) WRITE(6,31) B,C
 31     FORMAT (2E15.8)
      IF (ABS(FC) .GE. ABS(FB) ) GO TO 40
      A = B
      B = C
      C = A
      FA = FB
      FB = FC
      FC = FA
C
C  CONVERGENCE TEST
C
   40 TOL1 = 2.0*EPS*ABS(B) + 0.5*TOL
      XM =  .5*(C - B)
      IF (ABS(XM) .LE. TOL1) GO TO 90
      IF (FB .EQ. 0.0) GO TO 90
C
C  IS BISECTION NECESSARY
C
      IF (ABS(E) .LT. TOL1) GO TO 70
      IF (ABS(FA) .LE. ABS(FB)) GO TO 70
C
C  IS QUADRATIC INTERPOLATION POSSIBLE
C
      IF (A .NE. C) GO TO 50
C
C  LINEAR INTERPOLATION
C
      S = FB/FA
      P = 2.0*XM*S
      Q = 1.0 - S
      GO TO 60
C
C  INVERSE QUADRATIC INTERPOLATION
C
   50 Q = FA/FC
      R = FB/FC
      S = FB/FA
      P = S*(2.0*XM*Q*(Q - R) - (B - A) * (R - 1.0))
      Q = (Q - 1.0)*(R - 1.0)*(S - 1.0)
C
C  ADJUST SIGNS
C
   60 IF (P .GT. 0.0) Q = -Q
      P = ABS(P)
C
C  IS INTERPOLATION ACCEPTABLE
C
      IF ((2.0*P) .GE. (3.0*XM*Q - ABS(TOL1*Q))) GO TO 70
      IF (P .GE. ABS(0.5*E*Q)) GO TO 70
      E = D
      D = P/Q
      GO TO 80
C
C  BISECTION
C
   70 D = XM
      E = D
C
C  COMPLETE STEP
C
   80 A = B
      FA = FB
      IF (ABS(D) .GT. TOL1) B = B + D
      IF (ABS(D) .LE. TOL1) B = B + SIGN(TOL1, XM)
      FB = F(B)
      IF ((FB*(FC/ABS (FC))) .GT. 0.0) GO TO 20
      GO TO 30
C
C  DONE
C
   90 ZEROIN = B
      RETURN
      END

***** ZEROIN.INFO *****

   ZEROIN IS A FUNCTION SUBPROGRAM WHICH FINDS
   A ZERO OF THE FUNCTION F(X) IN THE INTERVAL AX, BX .
   THE CALLING STATEMENT SHOULD HAVE THE FORM

       X* = ZEROIN(AX, BX, F, TOL, IP)
 WHERE THE PARAMETERS ARE DEFINED AS FOLLOWS.

  INPUT

  AX    LEFT ENDPOINT OF INITIAL INTERVAL
  BX    RIGHT ENDPOINT OF INITIAL INTERVAL
  F     FUNCTION SUBPROGRAM WHICH EVALUATES F(X) FOR ANY X IN
        THE INTERVAL AX, BX
  TOL   DESIRED LENGTH OF THE INTERVAL OF UNCERTAINTY OF THE
        FINAL RESULT ( .GE. 0.0)
 IP     AN INTEGER PRINT FLAG.  WHEN SET TO 0, NO PRINTING
        WILL BE DONE BY ZEROIN.  IF SET TO 1, THEN
        ALL OF THE INTERVALS COMPUTED BY ZEROIN WILL
        BE PRINTED OUT.


  OUTPUT

   ZEROIN ABCISSA APPROXIMATING A ZERO OF F IN THE INTERVAL AX, BX


     IT IS ASSUMED THAT F(AX) AND F(BX) HAVE OPPOSITE SIGNS
   WITHOUT A CHECK.  ZEROIN RETURNS A ZERO X IN THE GIVEN INTERVAL
   AX, BX TO WITHIN A TOLERANCE 4*MACHEPS*ABS(X) + TOL, WHERE MACHEPS
   IS THE RELATIVE MACHINE PRECISION.
     THIS FUNCTION SUBPROGRAM IS A SLIGHTLY MODIFIED TRANSLATION OF
   THE ALGOL 60 PROCEDURE ZERO GIVEN IN RICHARD BRENT, ALGORITHMS FOR
   MINIMIZATION WITHOUT DERIVATIVES, PRENTICE - HALL, INC. (1973).
 THIS VERSION IS COPIED FROM "COMPUTER METHODS FOR MATHEMATICAL
 COMPUTATIONS" BY FORSYTHE, MALCOLM, AND MOLED.  THE ONLY CHANGE
 IS THE INCLUSION OF THE PRINT FLAG IP.
```

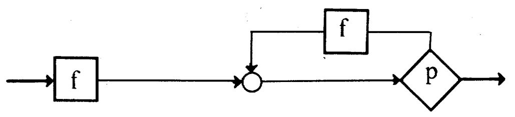
Fig. 1. Original Fortran ZEROIN.

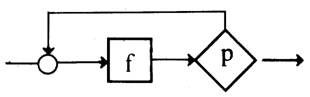
### Program Functions

We define a *program function* of a proper program P, denoted [P], to be the function computed by all possible executions of P which start at its entry and terminate at its exit. That is, a program function [P] is a set of ordered pairs, the first member being a state of the data on entry to P and the second being the resulting state on exit. Note that the state of data includes input and output files, which may be read from or written to intermittently during execution. Also note that if a program does not terminate by reaching its exit line from some initial data at its entry, say by looping indefinitely or by aborting, no such pair will be determined and no mention of this abnormal execution will be found in its program function.

Proper programs are convenient units of documentation. Their program functions *abstract* their entire effect on the data known to the program. Within a program, any subprogram that is proper can be also abstracted by its program function, that is, the effect of the subprogram can be described by a single function node whose function is the program function of the subprogram.

We say two programs are *function equivalent* if their program functions are identical. For example, the programs

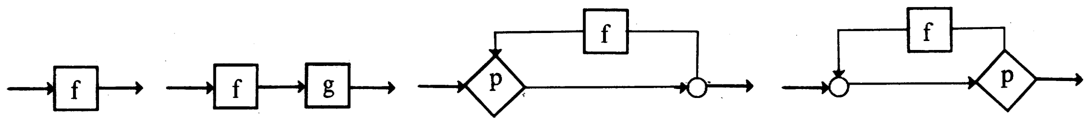




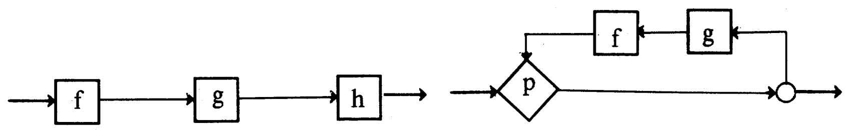
have different flowcharts but are function equivalent.

### Prime Programs

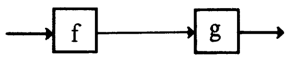
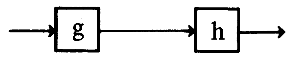
We define a *prime program* to be a proper program that contains no subprogram that is proper, except for itself and function nodes. For example,



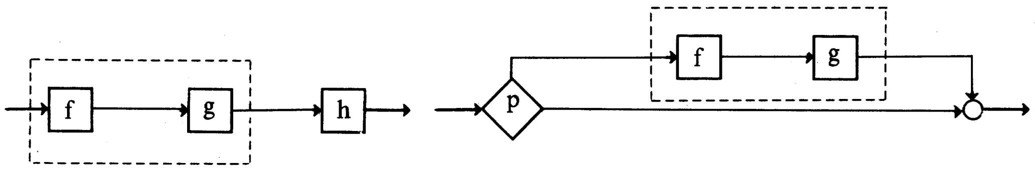
are primes, while



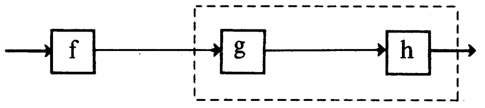
are not prime (*composite programs*), the first (of the composites) having subprograms

 and 

Any composite program can be decomposed into a hierarchy of primes, a prime at one level serving as a function node at the next higher level. For example, the composite programs above can be decomposed as shown next:



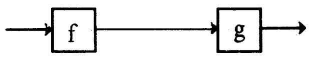
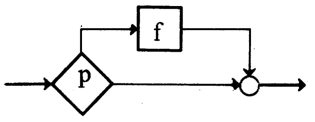
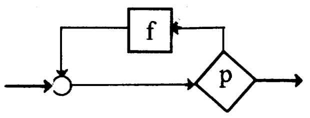
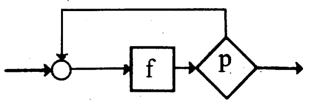
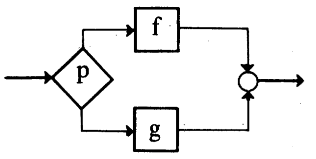
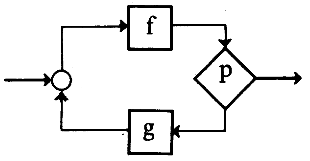
In each case, a prime is identified to serve as a function node in another prime at the next level. Note also that the first composite can also be decomposed as



so that the prime decomposition of proper programs is not necessarily unique.

*Prime Programs in Text Form*

There is a striking resemblance between prime programs and prime numbers, with function nodes playing the role of unity, and subprograms the role of divisibility. Just as for numbers, we can enumerate the control graphs of prime programs and give a text description of small primes in PDL (Process Design Language) [6] as follows:

| Flowchart | PDL |
|---|---|
|  | f; g |
|  | if p then f fi |
|  | while p do f od |
|  | do f until p od |
|  | if p then f else g fi |
|  | do1 f while p do2 y od |

Larger primes will go unnamed here, although the case statement of Pascal is a sample of a useful larger prime. All the primes above, except the last (dowhiledo), are common to many programming languages. Prime programs in text form can be displayed with standard indentation to make the subprogram structure and control logic easily read, which we will illustrate for ZEROIN.

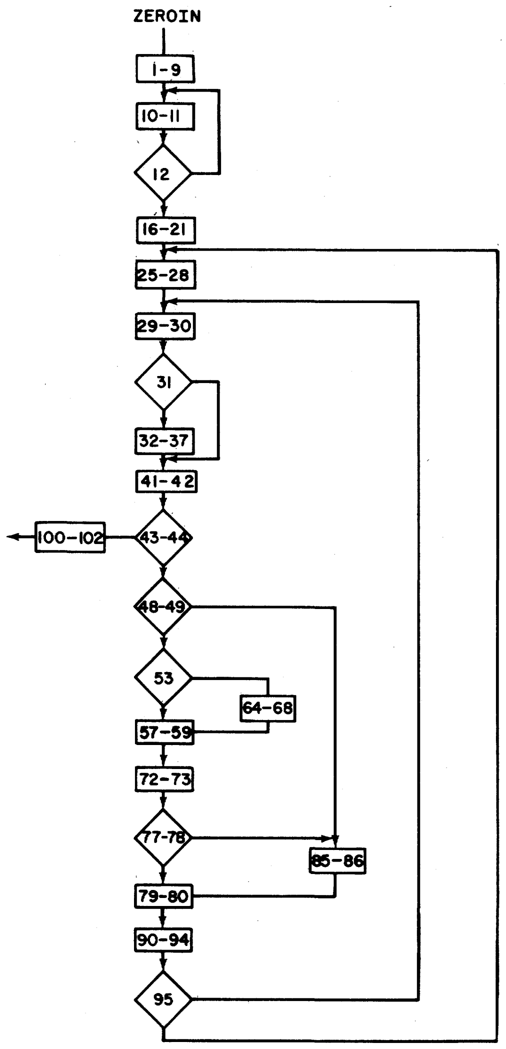


Fig. 2. Flowchart of Fortran ZEROIN.

## III. Understanding ZEROIN

Our overall approach in understanding ZEROIN is carried out in the following steps.

1) Perform a prime program decomposition which involves a restructuring of the program into a set of simple constituents which are represented by the single predicate prime programs discussed in the last section.

2) Develop a data reference table and analyze the data references from the point of view of where variables have been set and referenced. This provides insights into the inputs and outputs of the various prime program segments.

3) Perform a function decomposition of the program associating functions with each of the prime program segments. In this way, step by step, the whole program function can be determined by whatever correctness techniques are available. In what follows, the authors have used axiomatic correctness techniques, creating loop invariants along the way, and functional correctness techniques.

### *The Prime Program Decomposition of ZEROIN*

Our first step in understanding ZEROIN was to develop a prime program decomposition of its flowchart. After a little experimentation, the flowchart for ZEROIN was diagrammed as shown in Fig. 2. The numbers in the nodes of the flowchart represent contiguous segments of the Fortran program of Fig. 1, so all lowest level sequence primes are already identified and abstracted.

The flowchart program of Fig. 2 was then reduced, a step at a time, by identifying primes therein and replacing each such prime by a newly numbered function node, e.g., R.2.3 names prime 3 in reduction 2 of the process. This prime decomposition of the Fortran ZEROIN is shown in Fig. 3, leading to a hierarchy of six levels. Of all primes shown in Fig. 3, we note only two that contain more than one predicate, namely R.3.1 and R.5.1, and each of these is easily transformed into a composite made up of primes with no more than one predicate. These transformations are shown in Fig. 4. We continue the reduction of these new composite programs to their prime decompositions in Fig. 5. In each of these two cases, a small segment of programs is duplicated to provide a new composite that clearly executes identically to the prime. Such a modification, which permits a decomposition into one predicate primes, is always possible provided an extra counter is used. In this case, it was fortunate that no such counter was required. It was also fortunate that the duplicated segments were small; otherwise, a program call in two places to the duplicated segment might be a better strategy.

### *A Structured Design of ZEROIN*

Since a prime program decomposition of a program equivalent to ZEROIN has been found with no primes of more than one predicate, we can reconstruct this program in text form in the following way. The final reduced program of ZEROIN is given in Reduction 6 of Fig. 3, namely, that R.6.1 is a sequence, repeated here,

R.6.1 =

$$\begin{tikzcd} {} \arrow[d] \\ \boxed{\text{R.2.1}} \arrow[d] \\ \boxed{\text{R.5.1}} \arrow[d] \\ \boxed{\text{100-102}} \arrow[d] \\ {} \end{tikzcd}$$

Now R.2.1 can be looked up, in turn, as

R.2.1 =

$$\begin{tikzcd} {} \arrow[d] \\ \boxed{\text{1-9}} \arrow[d] \\ \boxed{\text{R.1.1}} \arrow[d] \\ \boxed{\text{16-21}} \arrow[d] \\ {} \end{tikzcd}$$

etc., until all intermediate reductions have been eliminated. Recall that R.5.1 and R.3.1 was further reduced in Fig. 5. When these intermediate reductions have all been eliminated,

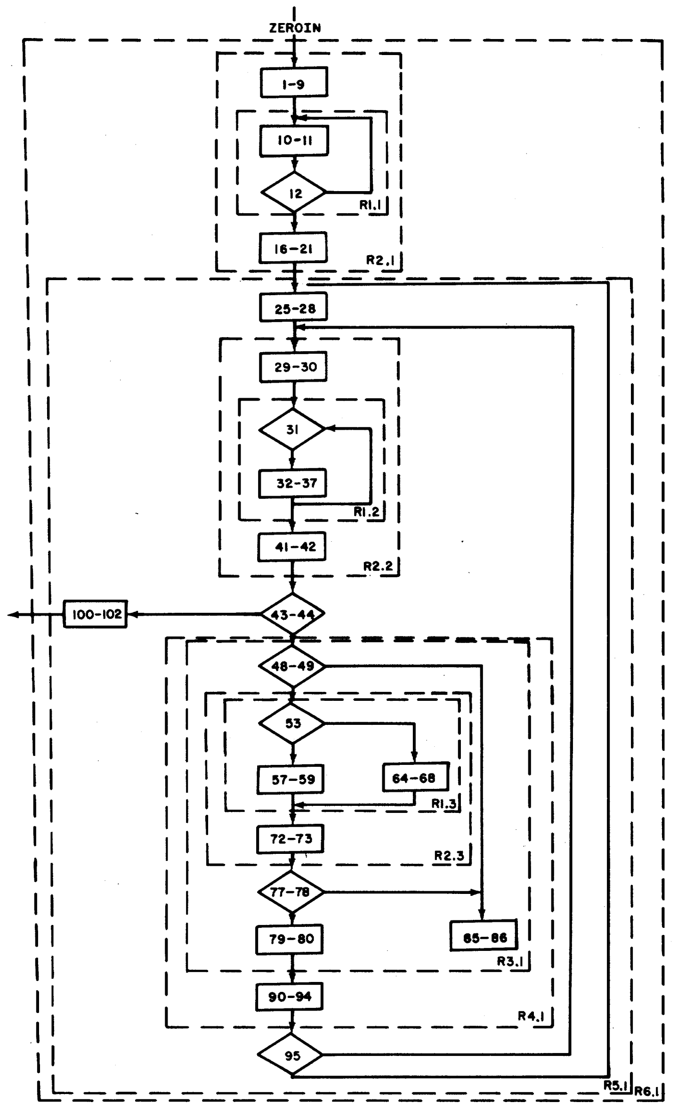


Fig. 3. Prime decomposition of Fortran ZEROIN.

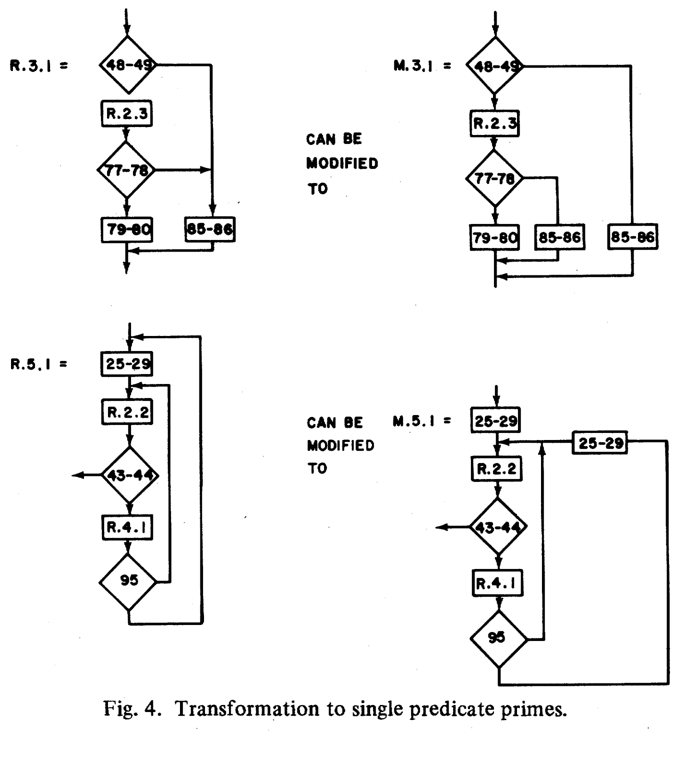


Fig. 4. Transformation to single predicate primes.

we obtain a structured program [2], [6], in PDL for ZEROIN shown in Fig. 6. Note there are three columns of statement numberings. The first column holds the PDL statement number; the second holds the Fortran line numbering of Fig. 1; the third holds the Fortran statement numbering of Fig. 1. The Fortran comments have been kept intact in the newly structured program and appear within square brackets [ , ]. From here on, statement numbers refer to the PDL statements of Fig. 6.

The duplication of code introduced in Fig. 4 can be seen in PDL 72, 73, and PDL 96-99. It should be noted, however, that in PDL 87-91 the second IF STATEMENT in Fortran 93 can be eliminated by use of the if-then-else. This permits an execution time improvement to the code. A second improvement can be seen in PDL 62-66. The use of the absolute value function can be eliminated by using the else part of an if-then-else to change the sign of a negative p.

By construction, the PDL program of Fig. 6 is function equivalent to the Fortran program of Fig. 1. But the structured PDL program will be simpler to study and understand.

## *Data References in ZEROIN*

Our next step in understanding ZEROIN was to develop a data reference table for all data identifiers. While straightforward and mechanical, there is still much learning value in carrying out this step, in becoming familiar with the program in the new structured form. The results are given in Fig. 7. This familiarization led to the following observations about the data references in ZEROIN (in no particular order of significance, but as part of a chronological, intuitive, discovery process).

1) ax, bx, f, ip, tol are never set, as might be expected, since they are all input parameters (but this check would discover initialized data if they existed, and the presence of side effects by the program on its parameters if passed by reference).

2) Zeroin is never used, but is returned as the purported zero found for f (since Zeroin is set to b just before the return of the program, it appears that b may be a candidate for this zero during execution).

3) eps is set by the dountil loop 6-11 at the start of program execution, then used as a constant at statement 36 from then on.

4) tol 1 is used for two different unrelated purposes, namely, as a temporary in the dountil loop 6-11 which sets eps, then reset at statement 36 as part of a convergence consideration in 36-88.

5) Function f is called only three times, at 16, 17 to initailize fa, fb, and at 92 to reset fb to f(b) (more evidence that b is the candidate zero to be returned).

6) Identifiers a, c are set to and from b, and the triple a, b, c seems to be a candidate for bracketing the zero that b (and zeroin) purports to approach.

7) Identifiers fa, fb, fc are evidently stand-ins for f(a), f(b), f(c), and serve to keep calls on function f to a minimum.

8) Identifiers p, q, r, s are initialized and used only in the section of the program that the comments indicate is concerned with interpolation.

9) Focusing on b, aside from initialization at statement 15 and as part of a general exchange among a, b, c at statement

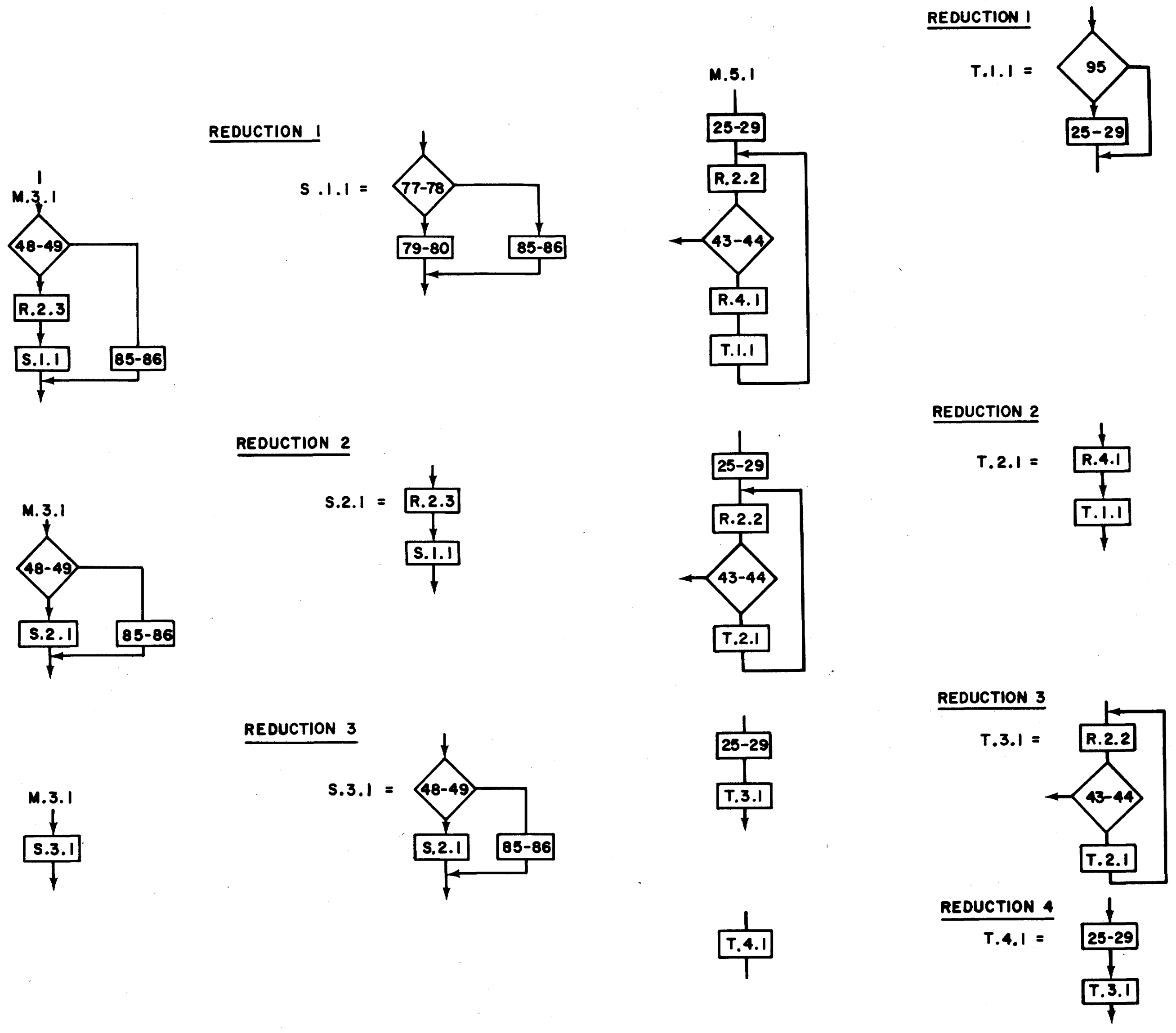


Fig. 5. Prime decomposition of the transformed ZEROIN.

28-29, b is updated only in the ifthenelse 83-90, incremented by either d or tol 1.

10) d is set to xm or p/q (as a result of a more complex bisection and interpolation process); xm is set only at statement 37 to the half interval of (b, c) and appears to give a bisection value for b.

## *A Function Decomposition of ZEROIN*

The prime program decomposition and the familiarity developed by the data reference tabulation and observations suggest the identification of various intermediate prime or composite programs in playing important roles in summing up a functional structure for ZEROIN. Each such intermediate prime or composite program computes values of a function. The inputs (function arguments) of this function are defined by the initial value of all identifiers that are inputs (function arguments) for statements that make up the intermediate program. The outputs (function values) of this function are defined by the final values of all identifiers that are outputs (function values) for statements that make up the intermediate program. Of course, further analysis may disclose that such a function is independent of some inputs, if, in fact, such an identifier is always initialized in the intermediate program before its use.

On the basis of this prime decomposition and data analysis, we reformulated ZEROIN of Fig. 6 as zeroin 1, a sequence of four intermediate programs, as shown in Fig. 8, with function statements using the form f. n-m where n, m are the boundary statements of the intermediate programs of ZEROIN from Fig. 6. Identifier *outfile in the output lists refers to the fact that data are being transferred to an outfile by an intermediate program. The phrase (x,z,v) projection of some function x,y,z,u,v,w := p,q,r,s,t,u means the new function x,z,v := p,r,t.

In the following program descriptions, all arithmetic operations are assumed to represent machine arithmetic. However, we will occasionally apply normal arithmetic axioms in order to simplify expressions. We next look at the intermediate programs.

*f.5-11:* The intermediate program that computes the values of f.5-11 is a sequence, namely, an initialized dountil, i.e.,

```
FORTRAN

Line      Stmt
Refer-     #
ence      ref.
               ZEROIN. PROGRAM

  1  1-2       func zeroin (real ax, bx, f, tol, integer ip)
  2    5         real a, b, c, d. e, eps, fa, fb, fc.
  3              tol 1. xm. p, q, r. s
  4    7        [COMPUTE EPS, THE RELATIVE MACHINE PRECISION]
  5    9         eps := 1.0
  6              do
  7   10   10      eps := eps/2.0
  8   11           tol 1 := 1.0 + eps
  9              until
 10   12           tol 1 ≤ 1
 11              od
 12   14        [INITIALIZATION]
 13   16         if ip = 1 then write ('THE INTERVALS DETERMINED BY ZEROIN ARE') fi
 14   18         a := ax
 15   19         b := bx
 16   20         fa := f(a)
 17   21         fb := f(b)
 18   23        [BEGIN STEP]
 19   25   20    c := a
 20   26         fc := fa
 21   27         d := b-a
 22   28         e := d
 23              do1
 24   29   30      if ip = 1 then write (b, c) fi
 25                if
 26   31              abs (fc) < abs (fb)
 27                then
 28   32              a := b
 29   33              b := c
 30   34              c := a
 31   35              fa := fb
 32   36              fb := fc
 33   37              fc := fa
 34                fi
 35   39        [CONVERGENCE TEST]
 36   41   40      tol 1 := 2.0 * eps * abs (b) + 0.5 * tol
 37   42           xm := .5 * (c-b)
 38   43        while
 39   44           abs (xm) >  tol 1 and fb ≠ 0
 40             do2
 41                [IS BISECTION NECESSARY]
 42                if
 43                  abs (e) < tol 1 or abs (fa) ≤ abs (fb)
 44   83           then [BISECTION]
 45   85   70         d := xm
 46   86              e := d
 47   46           else [IS QUADRATIC INTERPOLATION POSSIBLE]
 48                   if
 49   48                a ≠ c
 50   62              then [INVERSE QUADRATIC INTERPOLATION]
 51   64   50           q := fa/fc
 52   65                r := fb/fc
 53   66                s := fb/fa
 54   67                p := s * (2.0 * xm * q * (q-r) - (b-a) * (r-1.0))
 55   68                q := (q-1.0) * (r-1.0) * (s-1.0)
 56   55              else [LINEAR INTERPOLATION]
 57   57                s := fb/fa
 58   58                p := 2.0 * xm * s
 59   59                q := 1.0 - s
 60                   fi
 61   70              [ADJUST SIGNS]
 62                   if                                  /* note can be          */
 63   72   60           p > o                           /* if p > o then q := -q */
 64                   then                                /*          else p := -p */
 65   72                q := -q                         /* in PDL               */
 66                   fi                                  /*                      */
 67   73              p := abs(p)
 68   75              [IS INTERPOLATION ACCEPTABLE]
 69                   if
 70   77               (2.0 * p) ≥ (3.0 * xm * q - abs (tol 1 * q))
 71   83              then [BISECTION]
 72   85   70           d := xm                         /* note 85-86 repeated */
 73   86                e := d                          /*       in PDL        */
 74                   else
 75   79                e :=d
 76   80                d := p/q
 77                   fi
 78                fi
 79                [COMPLETE STEP]
 80   90   80      a := b
 81   91           fa := fb
 82                if
 83   92             abs(d) > tol 1                     /* note test done twice */
 84                then                                  /*      in FORTRAN      */
 85   92             b := b + d
 86                fi                                    /*   here and           */
 87                if                                    /*   in  PDL line 88    */
 88   93             abs(d) ≤ tol 1
 89                then
 90   93             b := b + sign (tol 1, xm)
 91                fi

 92   94           fb := f(b)
 93                if
 94                  fb * (fc/abs (fc)) > 0.0
 95                then [BEGIN STEP]
 96   25   20        c := a                             /* note 25-28 */
 97   26             fc := fa                           /*  repeated  */
 98   27             d := b - a                         /*  in PDL    */
 99   28             e := d
100                fi
101             od
102   98        [DONE]
103  100        zeroin := b
104  101        return
105  102     cnuf
```

Fig. 6. Transformed PDL ZEROIN.

| | Assigned | Used |
|---|---|---|
| a | 14,28,80 | 16,19,21,30,49,54,96,98 |
| ax | | 14 |
| b | 15,29,85,90 | 17,21,24,28,36,37,54,80,85,90,92,98,103 |
| bx | | 15 |
| c | 19,30,96 | 29,37,49 |
| d | 21,45,72,76,98 | 22,46,73,75,83,85,88,99 |
| e | 22,46,73,75,99 | 43 |
| eps | 5,7 | 7,8,36 |
| f | | 16,17,92 |
| fa | 16,31,81 | 20,33,43,51,53,57,97 |
| fb | 17,32,92 | 26,31,39,43,52,53,57,81,94 |
| fc | 20,33,97 | 26,32,51,52,94 |
| ip | | 13,24 |
| p | 54,58,67 | 63,67,70,76 |
| q | 51,55,59,65 | 54,55,65,70,76 |
| r | 52 | 54,55 |
| s | 53,57 | 54,55,58,59 |
| tol | | 36 |
| tol 1 | 8,36 | 10,39,43,70,83,88 |
| xm | 37 | 39,45,54,58,70,72,90 |
| zeroin | 101 | |

Fig. 7. Data references of PDL ZEROIN.

```
1   func zeroin 1 (real ax, bx, f, tol, integer ip)
2     real a, b, c, d, e, eps, fa, fb, fc, p, q, r, s, tol 1, xm
3     integer ip
4     [compute eps, the relative machine precision]
5       eps, tol 1 := f. 5-11
6     [initialize data]
7       a, b, c, d, e, fa, fb, fc, *outfile := f. 13-22 (ip, ax, bx, f)
8     [estimate b as a zero of f]
9       a, b, c, d, e, fa, fb, fc, p, q, r, s, tol 1, xm, *outfile :=
        f. 23-101 (a, b, c, d, e, f, fa, fb, fc, ip, p, q, r, s, tol 1, xm)
10    [set zeroin for return, zeroinl := b]
11      zeroinl := f. 103-103(b)
12  cnuf
```

Fig. 8. Top level function/data partition of PDL ZEROIN.

```
 5  eps := 1.0
 6  do
 7     eps := eps/2.0
 8     tol 1 := 1.0 + eps
 9  until
10     tol 1 ≤ 1
11  od
```

After some thinking, we determine that at PDL 6, an invariant of the form

$$I6 = (\exists k \geq 0 \, (eps = 2^{-k})) \wedge 1 + eps > 1$$

must hold, since entry to PDL 6 must come from PDL 5 or PDL 10 (and in the latter case tol 1 > 1, having just been set to 1.0 + eps, so 1.0 + eps > 1). Furthermore, at PDL 9 the invariant

$$I9 = (\exists k \geq 1 \, (eps = 2^{-k})) \wedge \text{tol } 1 = 1 + eps$$

must hold, by observing the effect of PDL 7, 8 on the invariant I6 at PDL 6. Therefore, at exit (if ever) from the segment PDL 5-11, we must have the condition I9 ∧ PDL 10, namely,

$$(\exists k \geqslant 1 \ (\text{eps} = 2^{-k})) \wedge 1 + 2 \ \text{eps} > 1 \wedge \text{tol } 1 = 1 + \text{eps} \leqslant 1.$$

Thus we have the following.

*Lemma 5-11:* The program function of f.5-11 is the constant function:

$$\{(\phi, (\text{eps}, \text{tol } 1)) \mid (\exists k \geqslant 1 \ (\text{eps} = 2^{-k})) \wedge 1 + 2 \ \text{eps} > 1 \wedge \text{tol } 1 = 1 + \text{eps} \leqslant 1\}.$$

Since tol 1 is reassigned (in PDL 36) before it is used again, f.5-11 can be thought of as computing only eps.

*f.13-22:* The intermediate program that computes the value of f.13-22 can be written directly as a multiple assignment. It is convenient to retain the single output statement PDL 13, and write

f.13-22 = f.13-13; f.14-22

yielding the following.

*Lemma 13-22:* The (a,b,c,d,e,*outfile) projection of f.13-22 is function equivalent to the sequence

f.13-13; f.14-22

where f.13-13 = <u>if</u> ip = 1 <u>then</u> write ('THE INTERVALS DETERMINED BY ZEROIN ARE') and

f.14-22 = a,b,c,d,e, fa, fb, fc

:= ax,bx,ax,bx - ax,bx - ax, f(a),f(b),f(a).

*f.23-101:* The intermediate program that computes the value of f.23-101 is a bit more complicated than the previous program segments and will be broken down into several subsegments. We begin by noticing that several of the input and output parameters may be eliminated from the list. Specifically, as noted earlier, p, q, r, and s are local variables to f.23-101 since they are always recalculated before they are used in f.23-101 and they are not used outside of f.23-101. The same is true for xm and tol 1. fa, fb, and fc can be eliminated since they are only used to hold the values of f(a), f(b) and f(c).

After considerable analysis and a number of false starts leading into a great deal of detail, we discovered an amazing simplification, first as a conjecture, then as a more precise hypothesis, and finally as a verified result. This simplification concerned the main body of the iteration of zeroin, namely, PDL 41-92, and obviated the need to know or check what kind of interpolation strategy was used, step by step. This discovery was that the new estimate of b always lay strictly within the interval bracketed by the previous b and c. That is, PDL 41-92, among other effects, has the (b) projection

$$b := b + \alpha(c - b), \qquad \text{for some } \alpha, 0 < \alpha < 1$$

so that the new b was a fraction $\alpha$ of the distance from the previous b to c. With a little more thought, it became clear that the precise values of d, e could be ignored, their effects being captured in the proper (but precisely unknown) value of $\alpha$. Furthermore, this new indeterminate (but bounded) variable $\alpha$ could be used to summarize the effect of d, e in the larger program part PDL 23-101, because d, e are never referred to subsequently. Thus, we may rewrite f.23-101 at this level as

a, b, c *outfile := f.23-101 (a, b, c, f, ip)

and we define it as an initialized while loop.

*Lemma 23-101:* The (a, b, c, *outfile) projection of f.23-101 is function equivalent to

```
(ip = 1 → write (b, c) | true → I);                [Lemma 24]
(abs(f(c)) < abs(f(b)) → a,b,c := b,c,b | true → I);
                                                   [Lemma 25-34]
while
    f(b) ≠ 0 ∧ (abs(c-b)/2) > 2 eps abs(b) + tol/2
do
    a, b, c := b, b + α(c-b), c    where 0 < α < 1;
                                                   [Lemma 41-92]
    (f(b) * f(c) > 0 → a, b, c := a, b, a | true → I);
                                                   [Lemma 93-100]
    (ip = 1 → write (b, c) | true → I);            [Lemma 24]
    (abs(f(c)) < abs(f(b)) → a,b,c := b,c,b | true → I)
                                                   [Lemma 25-34]
od
```

where I is the identity mapping.

The structure of f.23-101 corresponds directly to the structure of PDL 23-101 except for a duplication of segment PDL 23-34 in order to convert the dowhiledo into a whiledo. The proof of the correctness of the assignments of f.23-101 is given in separate lemmas as noted in the comments attached to the functions in Lemma 23-101. The while test is obtained by direct substitution of values for tol 1 and xm defined in PDL 36-37 into the test in PDL 39 using eps as defined in Lemma 5-11.

*Lemma 24:* PDL 24 is equivalent to

(ip = 1 → write (b, c) | <u>true</u> → I).

*Proof:* By direct inspection.

*Lemma 25-34:* The (a, b, c) projection of the program function of PDL 25-34 is function equivalent to

(abs(f(c)) < abs(f(b)) → a,b,c := b,c,b | <u>true</u> → I).

*Proof:* By direct inspection of PDL 25-34.

*Lemma 41-92:* The (a, b, c) projection of the program function of PDL 41-92 is function equivalent to

$$a, b, c := b, b + \alpha(c - b), c \qquad \text{where } 0 < \alpha < 1.$$

The proof will be done by examining the set of relationships that must hold among the variables in PDL 41-92 and analyzing the values of p and q only. That is, it is not necessary to have any knowledge of which interpolation was performed to be able to show that the new b can be defined by

$$b := b + \alpha(c - b), \qquad 0 < \alpha < 1.$$

We will ignore the test on PDL 48 since it will be immaterial to the lemma whether linear or quadratic interpolation is performed. We will examine only the key tests and assignments and do the proof in two basic cases—interpolation and bisection—to show that the (d) projection of the program function of PDL 41-78 is

$$d = (c-b)(\alpha) \quad \text{where } 0 < \alpha < 1.$$

*Case 1–Interpolation:* If interpolation is done, an examination of Fig. 6 shows that the following set of relations holds at PDL 78:

| | |
|---|---|
| • $I1 \equiv \text{tol 1} = 2 * \text{eps} * \text{abs}(b) + .5 * \text{tol}$ | (PDL 36) |
| • $I2 \equiv xm = (c-b)/2$ | (PDL 37) |
| • $I3 \equiv \text{abs}(xm) > \text{tol 1}$ | (PDL 39) |
| • $I4 \equiv p \geqslant 0$ | (PDL 67) |
| • $I5 \equiv 2 * p < 3 * xm * q - \text{abs}(\text{tol 1} * q)$ | (PDL 70) |
| • $I6 \equiv d = p / q$ | (PDL 76) |
| • $I7 \equiv \text{abs}(d) > \text{tol 1}$ | (PDL 83) |

Now let us examine the set of cases on p and q.

*$p > 0 \wedge q < 0$:* We have $d = p/q < 0$ (by hypotheses), $p/q > 3/2\ xm + \text{tol } 1/2$ (by I5), and $\text{tol } 1 > 0$ (by I1). Since $\text{abs}(xm) > \text{tol 1}$ (by I3) and $3/2\ xm + \text{tol } 1/2 < 0$ (since $p/q < 0$) we have $xm < 0$ implying $0 > d > p/q > 3/2\ xm > 3/4\ (c-b) > (c-b)$. Thus $0 > d > (c-b)$ yielding $d = \alpha(c-b)$ where $0 < \alpha < 1$.

*$p > 0 \wedge q > 0$:* We have $d = p/q > 0$ (by hypotheses), $p/q < 3/2\ xm - \text{tol } 1/2 < 3/2\ xm = 3/4\ (c-b) < (c-b)$ (by I5, I1, I2) implying $0 < d < (c-b)$. Thus $d = \alpha(c-b)$ where $0 < \alpha < 1$.

*$p > 0 \wedge q = 0$:* $q = 0$ implies $0 > 2 * p$ (by I5) and we know $p > 0$ (by hypotheses), implying a contradiction.

*$p = 0 \wedge q = \text{anything}$:* $\text{abs}(p/q) > \text{tol 1}$ (by I6, I7) and $\text{tol 1} \geqslant 0$ (by I1) implies p cannot be 0.

*$p < 0 \wedge q = \text{anything}$:* $p \geqslant 0$ (by I4) implies a contradiction.

*Case 2–Bisection:* If bisection is done, an examination of Fig. 6 shows that the following set of relations holds at PDL 78:

| | |
|---|---|
| $B1 \equiv xm = (c-b)/2$ | (PDL 37) |
| $B2 \equiv \text{abs}(xm) > \text{tol 1}$ | (PDL 39) |
| $B3 \equiv d = xm$ | (PDL 45 or PDL 72). |

Here $d = xm$ (by B3) implies $\alpha = 1/2$ (by B1) and thus $d = (c-b)(\alpha)$ where $0<\alpha<1$.

PDL 82-91 implies if $|d| \leqslant \text{tol 1}$ (i.e., if d is too small) then increment b by tol 1 with the sign adjusted appropriately, i.e., set

$$\alpha = \begin{Bmatrix} d & \text{abs}(d) > \text{tol 1} \\ \text{sign}(\text{tol 1}, sm) & \text{otherwise} \end{Bmatrix}.$$

But $\text{tol } 1 < \text{abs}(xm)$ (by I3 and B2) $= \text{abs}((c-b)/2)$ and the sign (tol 1) is set to the sign (xm) implying

$$\text{tol } 1 = \alpha(c-b) \quad \text{where } 0 < \alpha < 1.$$

Thus, in PDL 82-91 b is incremented by d or tol 1, both of which are of the form $\alpha(c-b)$ where $0 < \alpha < 1$. Thus we have

$$b := b + \alpha(c-b), \quad 0 < \alpha < 1$$

and since in PDL 80-81 we have a, fa := b, fb we get the statement of the lemma.

Once again, the reader is reminded that the proof of Lemma 41-92 was done by examining cases on p and q only. No knowledge of the actual interpolations was necessary. Only tests and key assignments were examined. Also, the program function was abstracted to only the key variables a, b, c and $\alpha$ represented the effect of all other significant variables.

*Lemma 93-100:* The (a,b,c) projection of PDL 93-100 is function equivalent to

$$(f(b) * f(c) > 0 \rightarrow a, b, c := a, b, a \mid \underline{\text{true}} \rightarrow I).$$

*Proof:* By direct inspection, PDL 93-100 is an ifthen statement with if test equivalent to the condition shown above and assignments that include the assignments above.

The last function in zeroin 1 (from Fig. 8) is the single statement PDL 103, which can be easily seen as Lemma 103.

*Lemma 103:* f.103 is function equivalent to zeroin := b.

Now that each of the pieces of zeroin 1 have been defined, the program function of ZEROIN will be given. First, let us rewrite zeroin1, all in one place, using the appropriate functions (Fig. 9).

The program ZEROIN has the required effect of finding and returning a root if there is one between the endpoints provided to it. The conditions under which this works are when either of the endpoints are roots or there is one root or an odd number of roots between the two endpoints (i.e., the functional values of the endpoints are of opposite signs). However, if the two endpoints provided to the program are identical, their value will be returned as the root. If there are no roots or a multiple of two roots between the two endpoints, the program will return a value as a root. This value may be one of the actual roots or it may be some point lying between the two points which is arrived at by continually halving the interval and eventually choosing one of the endpoints of a halved interval when the interval gets small enough.

The behavior of the program is more formally defined in the following theorem.

*Theorem 1-105:*

```
func zeroin has program function [zeroin] =
    (ax = bx → root := bx |
     f(bx) = 0 → root := bx |
     f(ax) = 0 → root := ax |
     f(ax) * f(bx) < 0 → root := approx (f, ax, bx, tol) |
     true → (∀ k = 1,2, · · · ,f($b_k$) * f($c_k$) > 0 → root
                 := unpredictable|
             ∃ k > 0 (f($b_k$) * f($c_k$) ⩽ 0 ∧ ∀ j = 1,2, · · · k - 1,
                 f($b_j$) * f($c_j$) > 0) → root
                 := approx (f, $b_k$, $c_k$, tol)
```

where approx (f, ax, bx, tol) is some value, x, in the interval (ax, bx) within $4 * \text{eps} * |x| + \text{tol}$ of some zero, x of the function f and the sequence $(b_1, c_1), (b_2, c_2), \cdots$ is defined so that each succeeding interval is a subinterval of the preceding interval; $(b_1, c_1) = (ax, bx)$, $(b_{k+1}, c_{k+1})$ defines the half interval of $(b_k, c_k)$ such that the endpoint kept is the one that minimizes the absolute value of f.

*Proof:* The proof will be carried out in cases, corresponding to the conditions in the rule given in the theorem. The first three cases follow directly by inspection of zeroin1, as special cases for input values, which bypass the while loop. That is, if ax = bx, then the values of a, b, c and root can be traced in zeroin 1 as follows:

| | a | b | c | root |
|---|---|---|---|---|
| zeroin 1.8 | bx | bx | bx | |
| 0.11 | bx | bx | bx | |
| [condition 13 fails since c-b = 0] | | | | |
| 0.21 | bx | bx | bx | bx. |

Cases 2 and 3 proceed in a similar fashion.

Case 4, $f(ax) * f(bx) < 0$, will be handled by an analysis of the whiledo loop and its results will apply to the last subcase of the last case as well. The first subcase of the last case arises when no zero of f is even bracketed and zeroin1 runs a predictable course, as will be shown.

*Case 4:* It will be shown that the entry condition $f(ax) * f(bx) < 0$ leads to the following condition at the whiletest of zeroin1:

$$I = (a = c \neq b \vee a < b < c \vee c < b < a) \\ \wedge f(b) * f(c) \leqslant 0 \wedge \text{abs}(f(b) \leqslant \text{abs}(f(c)).$$

The proof is by induction. First, I holds on entry to the whiledo loop because by direct calculation

after zeroin1.8 $\quad a = c \wedge f(b) * f(c) < 0 \wedge c \neq b$

after zeroin1.11 $\quad a = c \wedge f(b) * f(c) < 0 \wedge \text{abs}(f(b)) \leqslant \text{abs}(f(c)) \wedge c \neq b.$

Next, suppose the invariant I holds at any iteration of the whiledo at the whiletest, and the whiletest evaluates true, it can be shown that I is preserved by the three-part sequence of the do part. In fact, the first part, in seeking a better estimate of a zero of f, may destroy this invariant, and the last two parts restore the invariant. It will be shown in Lemma 15-18 that

after zeroin1.15 $\quad (a < b < c \vee c < b < a) \wedge f(a) * f(c) < 0$

after zeroin1.16 $\quad (a=c\neq b \vee a < b < c \vee c < b < a) \wedge f(b) * f(c) \leqslant 0$

after zeroin1.18 $\quad (a=c\neq b \vee a < b < c \vee c < b < a) \wedge f(b) * f(c) \leqslant 0 \wedge \text{abs}(f(b)) \leqslant \text{abs}(f(c))$

which is I. Thus, I is indeed an invariant at the whiletest.

Consider the question of termination of the whiledo. In Lemma 15-18T it will be shown using $c_0$ and $b_0$ as entry values to the do part, that for some $\alpha$, $0<\alpha<1$, after zeroin1.18 $\text{abs}(c-b) < \text{abs}(c_0 - b_0) \max(\alpha, 1-\alpha)$. Therefore, the whiledo must finally terminate because the condition

$$f(b) \neq 0 \wedge \text{abs}((c-b)/2) > 2 * eps * \text{abs}(b) + tol/2$$

must finally fail, because by the finiteness of machine precision abs(c-b) will go to zero if not terminated sooner.

When the whiledo terminates, the invariant I must still hold. In particular $f(b) * f(c) \leqslant 0$, which combined with the negation of the whiletest gives

$$IT = f(b) * f(c) \leqslant 0 \wedge (f(b)) = 0 \vee \text{abs}((c-b)/2) \\ \leqslant 2 * eps * \text{abs}(b) + tol/2.$$

IT states that

1) a zero of f is bracketed by the interval (b, c);

2) either the zero is at b or the zero is at most $|c-b|$ from b, i.e., the zero is within $4 * eps * |b| + tol$ of b.

```
1  func zeroinl  (real ax, bx, f, tol, integer ip)
2    real a, b, c, d, e, eps, fa, fb, fc, α
3    file *outfile
4    [compute eps, the relative machine precision]
5       eps := {x | (∃ k ⩾ 1 (x = 2^-k)) ∧ 1 + 2 x > 1 ∧ 1 + eps ⩽ 1} ;
6    [initialize data]
7       (ip = 1 → *outfile := 'THE INTERVALS DETERMINED BY ZEROIN
                              ARE'  | true → I) ;
8       a,b,c,d,e := ax,bx,ax,bx-ax,bx-ax
9    [estimate b as a zero of f]
10      (ip = 1 → *outfile (b, c) | true → I) ;
11     (abs(f(c)) < abs(f(b)) a, b, c := b, c, b | true → I)
12      while
13         f(b)  ≠ 0 ∧ abs((c-b)/2) > 2 eps abs(b) + tol/2
14      do
15         a, b, c := b, b + α (c-b), c  where  0 < α < 1;*
16         (f(b) * f(c) > 0 → a, b, c := a, b, a | true  → I) ;
17         (ip = 1 → *outfile(b, c) | true → I) ;
18         (abs(f(c)) < abs(f(b)) → a, b, c := b, c, b | true → I)
19      od;
20      [set zeroinlfor return, zeroinl := b]
21      zeroinl:= b
22    return
23  cnuf
*  α is an indeterminate based on the current values of a, b, c, d, e, f,
   fa, fb, fc, tol and eps
```

Fig. 9. Function abstraction of PDL ZEROIN.

This is the definition of approx (f, b, c, tol).

Now, beginning with the interval (ax, bx), every estimate of b created at zeroin1.15 remains within the interval (b,c) current at the time.[1] Since c and b are initialized as ax and bx at zeroin1.8, the final estimate of b is given by approx (f, ax, bx, tol). The assignment zeroin := b at zeroin1.21 provides the value required by case 4.

*Case5–Part 1:* We first show that in this case the condition $a = c$ will hold at zeroin1.15 if $f(b) * f(c) > 0$. By the hypothesis of case 5, part 1, $f((b+c)/2)$ is of the same sign as f(b) and f(c). Therefore, the first case of zeroin1.16 will hold and the assignment $c := a$ will be executed implying $a = c$ when we arrive at zeroin1.15 from within the loop. Also, if we reach zeroin1.15 from outside the loop (zeroin1.8-11) we also get $a = c$.

We now apply Lemma 15L, which states that under the above condition the (a, b, c) projection of zeroin1.15 is

$$(f(b) * f(c) > 0 \rightarrow a, b, c \\ := b, \begin{Bmatrix} b + (c-b)/2, & \text{if abs}(c-b)/2 > \text{tol } 1 \\ b + \text{tol } 1, & \text{otherwise} \end{Bmatrix}, c \\ \underline{\text{true}} \rightarrow a, b, c := b, b + \alpha(c-b), c)$$

which is a refinement of zeroin1.15.

Note that zeroin1.18 may exchange b,c depending on abs(f(b)) and abs(f(c)). Thus, the (b,c) projection of the function computed by zeroin1.15-18 in this case is

$$b, c := \begin{Bmatrix} b + (c-b)/2 \\ b + \text{tol } 1 \end{Bmatrix}, b \text{ or } b, c := b, \begin{Bmatrix} b + (c-b)/2 \\ b + \text{tol } 1 \end{Bmatrix},$$

i.e., the new interval (b, c) is the half interval of the initial $(b_0, c_0)$ which includes $b_0$ (for increments greater than tol 1),

[1]This is because $f(b) * f(c) \leqslant 0$ is part of I.

and the new b is chosen to minimize the value abs(f(b)). The result of iterating this dopart is unpredictable unless more is known about the values of f. For example, if the values of f in (ax, bx) are of one sign and monotone increasing or decreasing, then the iteration will go to the endpoint ax or bx for which abs(f) is minimum. In general, the iteration will tend toward a minimum for abs(f), but due to the bisecting behavior, no guarantees are possible.

*Case 5–Part 2:* This covers the happy accident of some intermediate pair b,c bracketing an odd number of zeros of f by happening into values $b_k$, $c_k$, such that $f(b_k) * f(c_k) \leqslant 0$. The tendency to move towards a minimum for abs(f(b)) may increase the chances for such a happening, but provides no guarantee. Once such a pair $b_k$, $c_k$ is found, case 4 applies and some zero will be approximated.

This completes the proof of the theorem except for the proofs of the three lemmas used in the proofs which are given in the Appendix.

## IV. Conclusion

### *Answering the Questions*

We can now answer the questions originally posed by Prof. Vandergraft.

*Question 1:* If the equation is linear and the size of the interval (a,b) is greater than or equal to tol 1, and there is no roundoff problem, the program will do a linear interpolation and find the root on one pass through the loop. If the size of the interval (a,b) is smaller than tol 1, the program will perform a bisection (based upon the test at PDL 43). If abs(fa) = abs(fb) at PDL 43, then bisection will also be performed. However, in this case bisection is an exact solution. The case that the size of the interval is smaller than tol 1 is unlikely, but possible.

*Question 2:* The theorem states that if f(a) and f(b) are both of the same sign, we will get an answer that is some point between a and b even though there is no root in the interval (a,b) (case 5a of the Theorem). If there are an even number of roots in the interval (a, b) then it is possible the program will happen upon one of the roots and return that root as an answer (case 5b of the Theorem). To check for this condition, we should put a test right at entry to the program between PDL 3 and PDL 4 of the form

```
if
   f(ax) * f(bx) > 0
then
   write ('F(AX) and F(BX) ARE BOTH OF THE SAME
           SIGN, RETURN BX')
   B := BX
else
   PDL 4-102
fi
```

Unfortunately, this does not indicate an error to the calling program. One approach in handling an error indication would be to add an extra parameter to the parameter list which would be set to indicate an error. Another approach would be to return a special value for the root, e.g., the largest negative number on the machine, as an error signal.

*Question 3:* It would be easy to remove the inverse quadratic interpolation part of the code. We can do this simply by removing several PDL statements, i.e., PDL 47-55. However, this would not leave us with the best solution since much of the code surrounding the inverse quadratic interpolation could be better written. For example,

1) there would be no need to keep a, b, and c;

2) the test in PDL 70 could be removed if we checked in the loop that f(a) * f(b) was always greater than zero, since bisection and linear interpolation would never take us out of the interval.

Cleaning up the algorithm would probably require a substantial transformation.

*Question 4:* Zeroin will find a triple root, assuming it is the only root in the interval. It will not inform the user that it is a triple root, but will return it as a root because once it has a root surrounded by two points such that f(a) and f(b) are of opposite signs, it will find that root (case 4 of the theorem).

It is also worth noting that ax and bx do not have to be the left and right endpoints of the interval; they could be interchanged. Also, any value of IP other than 1 will be equivalent to zero.

### *Program History*

Since most programs seem by practicing programmers do not have a history in the literature, we did not research the history of ZEROIN until we had completed our experiment. The plexity of the program is partially due to the fact that it was modified over a period of time by different authors, each modification making it more efficient, effective or robust. The code is based on the secant method [7]. The idea of combining it with bisection had been suggested by several people. The first careful analysis seems to have been by Dekker [3]. Brent [1] added to Dekker's algorithm the inverse quadratic interpolation option, and changed some of the convergence tests. The Brent book contains an Algol 60 program. The Fortran program of Fig. 1 is found in [5] and is a direct translation of Brent's algorithm, with the addition of a few lines that compute the machine-rounding error. We understand that ZEROIN is a significant and actively used program for calculating the roots of a function in a specific interval to a given tolerance.

### *Understanding and Documenting*

As it turns out, we were able to answer the questions posed and discover the program function of ZEROIN. The techniques used included function specification, the discovery of loop invariants, case analysis, and the use of a bounded indeterminate auxiliary variable. The discovery process used by the authors was not as direct as it appears in the paper. There were several side trips which included proving the correctness of the inverse quadratic interpolation (an interesting result but not relevant to the final abstraction or the questions posed).

There are some implications that the algorithm of the program was robust in that it was overdesigned to be correct and that the tests may be more limiting than necessary. This made the program easier to prove correct, however.

In documenting this program, we learned all the details first and, in that sense, worked bottom up. The method provided a systematic way to accumulate the detailed knowledge and package it in small pieces which consisted of theorems and lemmas. Learning of the details first was necessary for the higher level understanding. This bottom-up process is typical in maintaining programs; the form of recording that understanding is not.

Unfortunately, we kept no record of time because the work was done over a rather long period of time in bits and pieces. The authors would guess that it would take several weeks for a maintenance programmer versed in these concepts to develop and document an understanding of this program, as was done here. The implication is that maintenance without good documentation is a highly expensive proposition and clearly an extremely creative process. Unfortunately, in many environments only novice programmers are put on the maintenance task. Probably it would be better for programmers to work in senior/junior pairs, devoting part-time to the problem.

The role of good maintenance should be to keep the requirements, specifications, design and code documents up to date during development so they will be available and can be updated during maintenance. This study supplies some evidence that the payoff in not having to recreate the specification and design structure during maintenance is considerable. Although this approach of formalizing the understanding and documentation process of maintenance may appear to be overdone, it is unfortunately a necessity for many environments. To maintain a program in an embedded system, it is necessary to understand it to modify it. If there is no documentation on the requirements of the current system (which has been modified over time), there is no choice but to take the approach that was taken by the authors. There do exist systems which no one really knows what they do. The only way to be able to understand them and document them so that they can be changed or updated is by going through processes similar to processes performed by the authors.

To reiterate, the process consists of reducing the program to be understood to small prime programs and then creating in a step-by-step process the functions produced by those primes, combining them at higher and higher levels until a full specification is achieved. It is the price we pay for maintenance when only the code exists as the final documentation of a system.

We believe this experience shows that the areas of program specification and program correctness have advanced enough to make them useful in understanding and documenting existing programs, and extremely important application today. In our case, we are convinced that without the focus of searching for a correctness proof relating the specification to the program, we would have learned a great deal, but would have been unable to record very much of what we learned for others.

Hamming pointed out that mathematicians and scientists stand on each other's shoulders but programmers stand on each other's toes. We believe that will continue to be true until programmers deal with programs as mathematical objects, as unlikely as they may seem to be in real life, as we have tried to do here.

## Appendix

*Lemma 15-18:* The invariant I defined as

$$I \equiv (a = c \neq b \vee a < b < c \vee c < b < a) \wedge f(b) * f(c) \leqslant 0 \wedge \mathrm{abs}(f(b)) \leqslant \mathrm{abs}(f(c))$$

is preserved by the execution of the loop body ZEROIN1.15-18.

*Proof:* We use the following abbreviations:

$$P \equiv \mathrm{abs}(f(b)) \neq 0 \wedge \mathrm{abs}((c-b)/2) > 2 * \mathrm{eps} * \mathrm{abs}(b) + \mathrm{tol}/2$$
$$I_0 \equiv ((c < b) \vee (c > b)) \wedge f(b) * f(c) < 0$$
$$I_1 \equiv (a < b < c \vee c < b < a) \wedge f(a) * f(c) < 0$$
$$I_2 \equiv (a = c \neq b \vee a < b < c \vee c < b < a) \wedge f(b) * f(c) \leqslant 0.$$

Note that P is the loop predicate. The validity of the lemma is an immediate consequence of the following conditions:

C1: $I \wedge P \Rightarrow I_0$
C2: $I_0$ {ZEROIN1.15} $I_1$
C3: $I_1$ {ZEROIN1.16} $I_2$
C4: $I_2$ {ZEROIN1.18} I.

Condition C1 is straightforward. C2 can be seen by considering $c < b$ and $c > b$ as different input cases. Condition C3 follows from

$I_1 \wedge f(b) * f(c) > 0 \{c := a\} I_2$ (note that setting $c = a$ changes the sign of $f(c)$)
$I_1 \wedge f(b) * f(c) \leqslant 0 \Rightarrow I_2$.

Similarly, C4 can be inferred from

$I_2 \wedge \mathrm{abs}(f(c)) < \mathrm{abs}(f(b)) \{a, b, c := b, c, b\} I$
$I_2 \wedge \mathrm{abs}(f(c)) \geqslant \mathrm{abs}(f(b)) \Rightarrow I$.

*Lemma 15-18T:* Given $b_0$, $c_0$ on entry to zeroin1.15-18 then for some $\alpha$, $0<\alpha<1$

after zeroin1.15 $\mathrm{abs}(c-b) = (1-\alpha)\,\mathrm{abs}(c_0-b_0)$
after zeroin1.16 $\mathrm{abs}(c-b) \leqslant \mathrm{abs}(c_0-b_0)\max(\alpha, 1-\alpha)$
after zeroin1.18 $\mathrm{abs}(c-b) \leqslant \mathrm{abs}(c_0-b_0)\max(\alpha, 1-\alpha)$.

*Proof:* After zeroin1.15

$$\mathrm{abs}(c-b) = \mathrm{abs}(c_0-b_0-\alpha(c_0-b_0)) = \mathrm{abs}(c_0-b_0)(1-\alpha) \qquad 0<\alpha<1$$
$$\mathrm{abs}(b-a) = \mathrm{abs}(b_0+\alpha(c_0-b_0) - b_0) = \mathrm{abs}\,\alpha(c_0-b_0) \qquad 0<\alpha<1.$$

After zeroin1.16

$$\mathrm{abs}(c-b) \leqslant \max(\mathrm{abs}(c_0-b_0)(1-\alpha), \mathrm{abs}(c_0-b_0)\alpha) \leqslant \mathrm{abs}(c_0-b_0)\max(\alpha, 1-\alpha).$$

After zeroin1.18

$\mathrm{abs}(c-b) \leqslant \mathrm{abs}(c_0-b_0)\max(\alpha,1-\alpha)$ since b and c are unchanged or exchanged.

It should be noted that in the above discussion, zeroin1.17 was ignored because its effect on the calculation of the root and termination of the loop is irrelevant.

We have one last lemma to prove.

*Lemma 15L:* Given $a = c$ and $f(a) * f(b) > 0$ then zeroin1.15 calculates the new b using the bisection method, i.e.,

$$b := b + \begin{cases} (b-c)/2 & \text{if abs}(c-b) > \text{tol 1} \\ \text{tol 1} & \text{otherwise} \end{cases}$$

*Proof:* From PDL 43, either abs(f(b)) < abs(f(a)) or bisection is done (PDL 45) with d = xm = (c-b)/2. Then PDL 82-91 implies

$$b := \begin{cases} b + d = b + (c-b)/2 & \text{if abs}(c-b)/2 > \text{tol 1} \\ b + \text{tol 1} & \text{otherwise} \end{cases}.$$

Since by hypothesis a = c, PDL 49 implies inverse quadratic interpolation is not done and linear interpolation (PDL 56) is attempted. Thus

$s = fb/fa$ and $0 < s < 1$ since $fb * fa > 0$ and $\text{abs}(fb) < \text{abs}(fa)$

$p = (c-b) * s$, using $xm + (c-b)/2$

$q = 1-s$, implying $q > 0$ in PDL 59.

The proof will be done by cases on the relationship between b and c.

*c > b:* c > b implies p > 0 in PDL 58. Since p > 0 before PDL 62, PDL 65 sets q to -q, so q < 0. Then the test at PDL 70 is true since

$2 * p = a * s$ is positive,

$3.0 * xm * q = \frac{3}{2}(c-b) * q$ is negative, and

abs(tol 1 * q) is positive

implying PDL 70 evaluates to true and bisection is performed in PDL 72-73.

*c < b:* c < b implies p < 0 in PDL 58. Since p < 0 before PDL 62, PDL 65 leaves q alone and PDL 67 sets p > 0 implying p = (b-c) * x. Then the test at PDL 70 is true since

$2 * p = 2 * (b-c) * s$ is positive,

$3.0 * xm * q = \frac{3}{2}(c-b) * q$ is negative, and

abs(tol 1 * q) is positive

implying PDL 70 evaluates to true and bisection is performed in PDL 72-73.

## Acknowledgment

The authors are grateful to D. Dunlop for his insightful review of this report and to C. Bacigaluppi for patiently typing numerous drafts.

## References

[1] R. P. Brent, *Algorithms for Minimization Without Derivatives.* Englewood Cliffs, NJ: Prentice-Hall, 1973.

[2] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming.* New York: Academic, 1972.

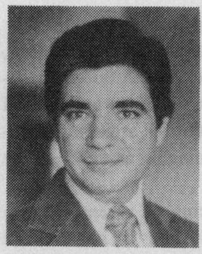
[3] T. J. Dekker, "Finding a zero by means of successive linear interpolation," in *Constructive Aspects of the Fundamental Theorem of Algebra*, B. Dejou and P. Henrici, Eds. Interscience, 1969.

[4] E. W. Dijkstra, *A Discipline of Programming.* Englewood Cliffs, NJ: Prentice-Hall, 1976.

[5] G. Forsythe, M. Malcolm, and M. Moler, *Computer Methods for Mathematical Computations.* Englewood Cliffs, NJ: Prentice-Hall, 1977.

[6] R. C. Linger, H. D. Mills, and B. I. Witt, *Structured Programming Theory and Practice.* Reading, MA: Addison-Wesley, 1979.

[7] J. Ortega and W. C. Rheinboldt, *Iterative Solution of Nonlinear Equations in Several Variables.* New York: Academic, 1970.



**Victor R. Basili** received the Ph.D. degree in computer science from the University of Texas, Austin.

He is an Associate Professor in Computer Science at the University of Maryland where he has been since 1970. He has been involved in the design and development of several software projects, including the SIMPL family of structured programming languages and is currently involved in the measurement and evaluation of software development at NASA/Goddard Space Flight Center. His interests lie in software development methodology and the quantitative analysis and evaluation of the software development process and product. This includes such specialized areas as cost modeling, error analysis, and complexity. He has authored about 50 journal papers, conference papers, and books in these areas. He has lectured on these topics around the world. He has consulted for several government agencies and industrial organizations, including IBM, GE, CSC, NRL, NSWC, and NASA. He has been Program Chairman for several conferences, including the Second Software Life Cycle and Management Workshop, the First ACM SIGSOFT Sponsored Engineering Symposium on Tools and Methodology Evaluation, and the Sixth International Conference on Software Engineering. He has served on several Editorial Boards, including the *Journal of Systems and Software* and the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING.

Dr. Basili is a member of the IEEE Computer Society and the Association for Computing Machinery.

**Harlan D. Mills** received the Ph.D. degree in mathematics from Iowa State University, Ames, in 1952.

He is currently an IBM Fellow in the Federal Systems Division, IBM Corporation, Bethesda, MD, and Professor of Computer Science at the University of Maryland, College Park. He has been employed with the IBM Corporation since 1964. He received an IBM Outstanding Contribution Award for new programming methodologies including top-down design, techniques used for structured programming, and the Chief Programmer Team concept. He has served on the Corporate Technical Committee and as FSD Director of Software Engineering and Technology. Before 1964, he served on the technical staffs at GE and RCA, as President of Mathematica, and as consultant to various government and industrial organizations. He was General Chairman of the First National Conference on Software Engineering 1975, and IEEE Fall COMPCON '81. He has served on the Program Committee for IFIP Congress '77 (as Software Area Chairman), and as an Editor of the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING. He was Keynote Speaker at the 2nd International Conference on Software Engineering in 1976 and Invited Speaker at IFIP Congress '80. He has served on faculties of Iowa State, Princeton, New York, and The Johns Hopkins Universities.

Dr. Mills was named an Honorary Fellow by Wesleyan University in 1962 and a Fellow of ACPA in 1975.