Fortran is Dead! Long Live Forth!

J.V. Noble

Department of Physics
University of Virginia
Charlottesville, Virginia 22901

Abstract

Many scientists, engineers and others using computers for numerical analysis and various forms of scientific computation have found Forth to be a strong contender as the scientific programming language of the future. One reason is that its extensibility permits Forth to handle floating-point and complex-number arithmetic as expeditiously and transparently as FORTRAN. And Forth has advantages, described below, over FORTRAN's antiquated ways.

Although previous reports of FORTRAN's demise have been exaggerated, avant garde number crunchers feel that Forth is poised to displace FORTRAN as the pre-eminent number crunching language. For this to happen, the Forth community will have to accept some standards for floating-point and other "scientific" data types.

Introduction

The community of number-crunchers is large and diverse, including physical scientists, mathematicians, statisticians, life scientists, engineers, operations analysts, architects, students, and many more. Since its introduction, FORTRAN has been the primary computer language of this community. That is generally recognized in the Forth community, where it has led to a regretable apathy concerning, e.g., such matters as Forth standards for floating-point arithmetic. The major themes of what I call "fear of floating" seem to be: "If we add floating-point, it won't be Forth," [1] and "If the people want to crunch numbers, let them use FORTRAN." [2] The latter Marie Antoinette approach is inappropriate for the grass-roots, people's language that Forth purports to be. Obviously someone wants to crunch numbers in Forth, or Forth vendors would not supply floating-point and complex-arithmetic extensions. [3]

There has to be a reason why the number-crunching community is turning from FORTRAN. The dear old lady still shows a fair turn of speed and style even with all those miles on her clock. Why abandon her now? and for Forth of all things? Sad to say, FORTRAN's age is showing. Her wrinkles peek through and she sags in embarassing places. The more adventurous number crunchers, realizing this, have flirted with C or Pascal, seeking renewal in a younger language. Their quest generally unsuccessful, most have returned — disillusioned —to their old FORTRAN. She may be tired, her structure may be nothing to brag of, she may be out of trim from too much spaghetti (code), but she gives them what they want and need. What does FORTRAN give them? In a nutshell, floating-point arithmetic, complex arithmetic, and double precision versions of both — all without having to jump through hoops or write unnatural and contorted code. And it doesn't hurt that FORTRAN also provides conveniences such as the formula translator that it's named for; rigid I/O conventions; and a natural array notation.

Why Crunch Numbers in Forth?

Many scientists and engineers have learned about Forth because it is fast and compact and offers a simple way to interface microprocessors with laboratory equipment and other machines. [4] Forth has not found great favor in number-crunching applications, however. But a few number crunchers (e.g. me) have discovered that Forth is a very good, perhaps ideal, candidate to replace FORTRAN. In Forth we have found a high-level language that is easy to program in, debug and maintain, that embodies the post-FORTRAN programming precepts of portability, structure, modularity and information-hiding, that permits recursion, that enables dynamic storage allocation (and re-allocation), that allows extension of types and operators as necessary, and that can be speed-tuned easily with in-line machine-code for selected, time-critical functions.

We Forth-wise number crunchers have found that, especially in conjunction with a co-processor equipped workstation, Forth can be extended to handle floating-point and complex-number arithmetic as transparently as FORTRAN. This is no trivial matter; I have seen articles [5] in which complex arithmetic is implemented in Pascal — essentially by re-inventing parts of Forth.

The speed issue is also crucial. Consider having to solve a lot of linear equations — say, 300 to 1000. Such problems sometimes arise in the kind of physics I do. Without access to an array processor or supercomputer, how can I handle a system of such magnitude? One obvious way is to use a library routine on the CDC mainframe at my university's Academic Computing Center. That routine will be written in FORTRAN, will use standard, well-tested techniques, and I can quickly write the program to use it, confident that it will work. So what's wrong? Why did I take so much trouble to roll my own in Forth for a 10MHz PC *cum* 8087? The problem is memory. Solving linear equations is practical only if the matrix and inhomogeneous term fit into RAM. The 640K of PC RAM easily accomodates a 350×350 REAL*4 matrix and its inhomogeneous term, or a 247×247 set of REAL*8 data. Isn't the CDC memory big enough? Of course it is. But to get access to such a large chunk of CDC memory I need a very high priority (it is a timeshare system, of course). In practical terms, this means batch submission, with turnaround time of several hours, perhaps overnight. But my PC, and it's not such a great one, does 350×350 in about sixteen minutes using Forth with the innermost loop hand-optimized in assembler. [6] With a Definicon [7] 68020/68881 board, I expect to reduce the time for 350×350 to one or two minutes. How would I cope with 999×999? The CDC's virtual memory will allow me the room, but cannot know which part of the matrix should go in RAM and which on disk. Thrashing will slow it considerably. Whereas, working in Forth on my PC using LIM extended memory to hold the data, and using moderate care in organizing the calculation, should realize the (theoretical) factor of 37x from a 3-fold partition of the matrix. The time would be about 8 hours on my PC, perhaps 0.5 to 1 hour on the Definicon board, probably only 5 minutes with an INMOS T800. [8]

Figure 1 contains another example, a FORTRAN subroutine for minimizing a function using the simplex algorithm, reproduced from the book Numerical Recipes. [9] The figure illustrates another reason, beyond sheer speed, that I love Forth, and prefer it to FORTRAN. Without comments it would be tedious to decipher either what the FORTRAN program does or its logical structure in terms of data or decision flow. The variable names are too cryptic to be mnemonic and there is no obvious decomposition by operation. The code is well-structured, but so what? It's still hard to understand. Contrast this with the main word in my Forth variant of this algorithm (Fig. 2). The structure of the Forth version of the simplex algorithm is so clear and readable that one hardly needs the flow diagram or comments (Fig. 3). Learning Forth gave me a strange sensation — I could understand my own programs! Figures 1 and 2 should show why.

```
SUBROUTINE AMOEBA(P,Y,MP,NP,NDIM,FTOL,FUNK,ITER)
      PARAMETER  (NMAX=2Ø,ALPHA=1.Ø,BETA=Ø.5,GAMMA=2.Ø,ITMAX=5ØØ)
      DIMENSION  P(MP,NP),Y(MP),PR(NMAX),PRR(NMAX),PBAR(NMAX)
      MPTS=NDIM+1
      ITER=Ø

1     ILO=1
      IF(Y(1).GT.Y(2))THEN
        IHI=1
        INHI=2
      ELSE
        IHI=2
        INHI=1
      ENDIF
      DO 11 I=1,MPTS
       IF(Y(I).LT.Y(ILO))  ILO=I
       IF(Y(I).GT.Y(IHI))THEN
          INHI=IHI
          IHI=I
       ELSE IF(Y(I).GT.Y(INHI))THEN
          IF(I.NE.IHI)  INHI=I
       ENDIF
11    CONTINUE
      RTOL=2.*ABS(Y(IHI)-Y(ILO))/
      (ABS(Y(IHI))+ABS(Y(ILO)))
      IF(RTOL.LT.FTOL)RETURN
      IF(ITER.EQ.ITMAX)  PAUSE
      'too many iterations.'
      ITER=ITER+1
      DO 12 J=1,NDIM
        PBAR(J)=Ø.
12    CONTINUE
      DO 14 I=1,MPTS
        IF(I.NE.IHI)THEN
          DO 13  J=1,NDIM
            PBAR(J)=PBAR(J)+P(I,J)
13        CONTINUE
        ENDIF
14    CONTINUE
      DO 15 J=1,NDIM
        PBAR(J)=PBAR(J)/NDIM
        PR(J)=(1.+ALPHA)*PBAR(J)
        -ALPHA*P(IHI,J)
15    CONTINUE
      YPR=FUNK(PR)
      IF(YPR.LE.Y(ILO))THEN
        DO 16 J=1,NDIM
          PRR(J)=GAMMA*PR(J)
          +(1.-GAMMA)*PBAR(J)
          DO 17  J=1,NDIM
            P(IHI,J)=PRR(J)
17        CONTINUE
          Y(IHI)=YPRR
        ELSE
          DO 18  J=1,NDIM
            P(IHI,J)=PR(J)
18        CONTINUE
          Y(IHI)=YPR
        ENDIF
      ELSE IF(YPR.GE.Y(INHI))THEN
        IF(YPR.LT.Y(IHI))THEN
          DO 19  J=1,NDIM
            P(IHI,J)=PR(J)
19        CONTINUE
          Y(IHI)=YPR
        ENDIF
        DO 21  J=1,NDIM
          PRR(J)=BETA*P(IHI,J)
          +(1.-BETA)*PBAR(J)
21      CONTINUE
        YPRR=FUNK(PRR)
        IF(YPRR.LT.Y(IHI))THEN
          DO 22  J=1,NDIM
            P(IHI,J)=PRR(J)
22        CONTINUE
          Y(IHI)=YPRR
        ELSE
          DO 24  I=1,MPTS
            IF(I.NE.ILO)THEN
               DO 23 J=1,NDIM
                 PR(J)=Ø.5*(P(I,J)
                 +P(ILO,J))
                 P(I,J)=PR(J)
23             CONTINUE
               Y(I)=FUNK(PR)
            ENDIF
24        CONTINUE
        ENDIF
      ELSE
        DO 25  J=1,NDIM
          P(IHI,J)=PR(J)
25      CONTINUE
        Y(IHI)=YPR
```

Fig. 1 A FORTRAN subroutine for minimizing by the simplex algorithm.

The Unique Benefits of Forth

There are some interesting things that Forth can do for number crunching that FORTRAN cannot encompass. For example, I have developed typed generic variables (defined by SCALAR), accessed and manipulated by generic operations that decide, by vectoring, what to do at run-time. **[10]** I pay a small speed penalty for the extra overhead (although nothing like that of object-oriented programming), but the virtue is that I write (and name and load) only one version of a word, such as a linear equation solver or an adaptive numerical quadrature routine, that works as easily with REAL*8, COMPLEX and COMPLEX*16 data types as with REAL. Now programming a contour integral in the complex plane is no harder than an integral on the real line, once I have defined and tested the generic adaptive quadrature routine at its core.

Forth permits another style that I have found useful — Forth makes it easy to encapsulate run-time code with the data structure that it operates on. For example, in a complex Monte-Carlo simulation of a process in high energy physics, one must choose random numbers from several different kinds of distributions, avoiding statistical correlation with preceding or succeeding choices. I manage this by tabulating the cumulative distributions that I need to choose from, **[11]** then looking up in the table using an address given by a uniformly distributed pseudo-random number generator (PRNG) with a very long period. **[12]** I handle correlations **[13]** by planting the seeds in the table itself, and designing the PRNG to get current seeds from the current table. The whole process of defining the table, filling it and initializing its seeds takes one step using **CREATE ... DOES>**, which has been justly called **[14]** the "pearl of Forth". All the tedium is neatly hidden from my eyes: I merely invoke the name of the appropriate table at the appropriate point in the simulation, and an appropriate random number is placed on the floating-point stack. I know of no such simple mechanism for encapsulations in any procedural language. Object-oriented languages permit it, of course, but they are usually too slow for number crunching.

```
\ SIMPLEX MINIMIZATION ALGORITHM
\ Usage:  USE{ Fn.Name  ERROR %  E }MINIMIZE


     : Not.Better?      New.Point   Worst.Point   Better.Test?
          IF  New.Point  INSERT   Ø  ELSE  Clean.Up   -1   THEN ;


     : }MINIMIZE     INITIALIZE
          BEGIN      Not.Close.Enuf? Iterations   Max. Iter  <  AND
          WHILE      REFLECT  Not.Better?    ( INSERT)
                     IF    DOUBLE Not.Better?  ( INSERT)
                          IF  HALVE    Not.Better? ( INSERT)
                               IF  SHRINK   SORT  THEN
                          THEN
                     THEN
          REPEAT   ;
```

Fig. 2 A FORTH variant of the simplex minimization algorithm.

Why FORTRAN is Slow to Die

FORTRAN programs are usually clumsy, but the formula translator is excellent. FORTRAN arithmetic is performed by "smart" operators working on typed variables and literals. This permits the convenience of mixed-mode arithmetic expressions, such as

$$A = B1*3 + B2*1.2E\text{-}5 - H(3)/3.14159265358979D\text{-}14 + K$$

FORTRAN provides a limited suite of data types: INTEGER, LONG-INTEGER, REAL, DREAL, COMPLEX, DCOMPLEX, LOGICAL and CHARACTER. It provides no facilities for defining new types (other than arrays of the above). But these are sufficient for most number crunching problems. Types can be either implicit, or declared explicitly. Arrays must be declared according to a strict format, but up to 3 indices are permitted. FORTRAN's array notation is simple, logical and follows the conventions of algebra, replacing subscripts with parentheses, e.g. A_{ij} becomes A(I,J).

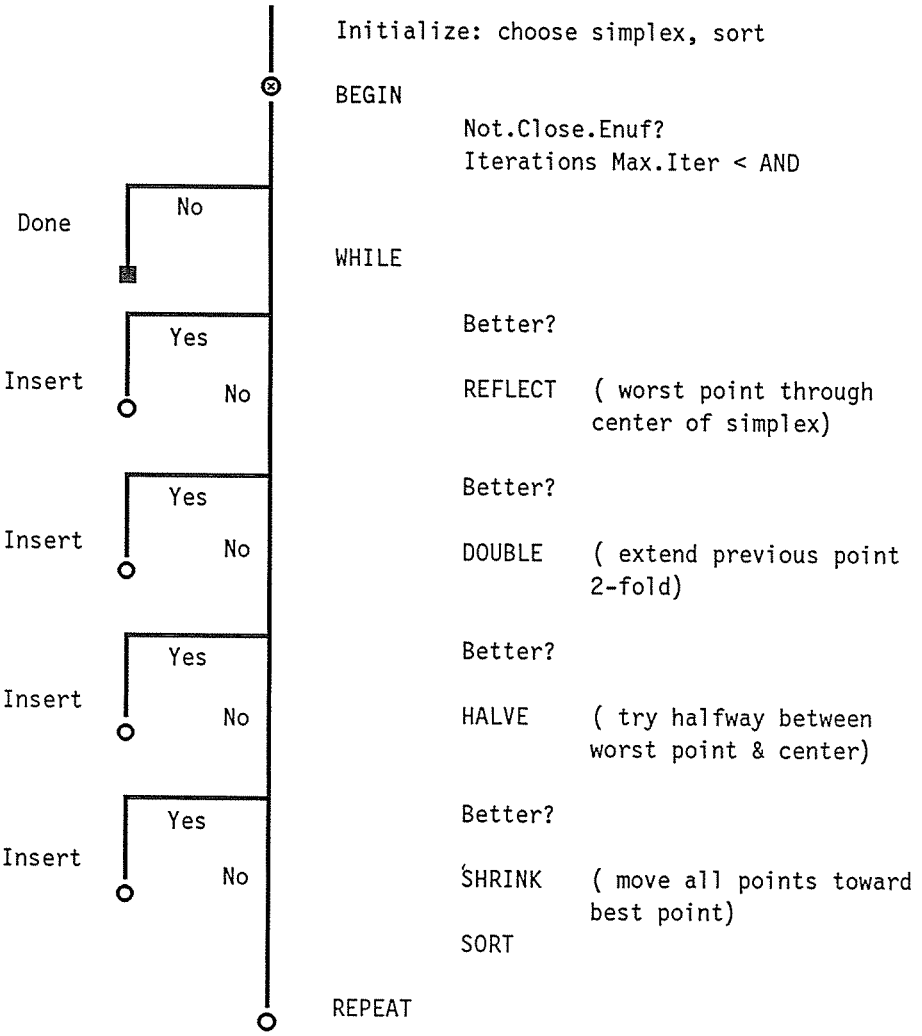

Fig. 3 Flow diagram of Simplex algorithm.

Crucial to FORTRAN's utility in scientific programming is the mathematical function library, including REAL, DREAL and COMPLEX versions of trigonometric functions, exponentials, logarithms, inverse trigonometric functions, sometimes hyperbolic functions and their inverses, and often a random number generator of uncertain quality. Very importantly, FORTRAN supports modularity through separate compilation of functions and subroutines. This enables the construction and exchange of standard subroutine libraries such as Numerical Recipes, CERNLIB, MATLAB, LINPACK, and so on. But this is not so much an attribute of the language as it is the historical accumulation of useful subroutines and functions. The same could exist for Forth.

All of these attributes give FORTRAN the essential functionality to solve scientific computational problems. It works, and many are reluctant to junk the horse and buggy if they have to learn to drive. The horse gets there too, eventually.

Why FORTRAN Should Die

To achieve the simplicity of mixed-mode expressions, the FORTRAN compiler must be prepared for any eventuality. The operators "+", "–", "*", "/", "**" and "=" must be "smart" — they must "know" (or at least be able to figure out) what kinds of numbers are going to be used and what kinds of arithmetic will be used to combine them. The smartness is built in at compile-time: Consider the actions performed by the FORTRAN compiler in parsing the mixed-mode expression

A = B1*3 + B2*1.2E-5 -H(3)/3.14159265358979D-14 + K

1) Space is defined and reserved for a floating-point single-precision variable A (implicit type REAL) if A does not already exist.
2) The literal integer constant 3 is converted to floating-point and multiplied by the (implicit-REAL) variable B1's current value (fetched from memory).
3) The product is placed in temporary storage (TEMP). The implicit-REAL variable B2 is fetched and multiplied by the REAL literal 1.2E-5.
4) The second product is added to the contents of TEMP.
5) The third element of the (implicit-REAL) array H is fetched and divided by the DREAL (double-precision) literal 3.14159265358979D-14, converting to and from DREAL format as necessary.
6) The dividend is converted to REAL and subtracted from TEMP.
7) The implicit INTEGER variable K is converted to REAL and added to TEMP. Finally,
8) The result is moved from TEMP to the memory reserved for A. It's clear why FORTRAN is a formula translator.

FORTRAN assumes implicit data types when names begin with certain letters of the alphabet. It also permits explicit type declarations that override the implicit compiler actions described above. Thus, had the program contained these declarations in its first few lines

```
INTEGER A, H(15), B1, B2
REAL K
```

the conversions and assignments would have been floating-point to integer, rather than vice-versa.

Imagine the complexity of a compiler that must be able to decide the type of each variable and then select the appropriate routine to combine them, perhaps optimizing at various levels. All this huffing and puffing is a mixed blessing. The compiler that can do it will be both complex and slow. My experience using Microsoft's two-pass FORTRAN compiler to perform a fairly

simple calculation in 120 lines of code, on a portable PC with only floppy drives, nearly curdled forever my basic forgiving kindness. The result of this complex compilation is that lengthy FORTRAN routines for micros are usually developed on a mainframe and ported to the smaller machine.

Modularity and separate compilation is another mixed blessing; many a subtle bug has been introduced in a FORTRAN program by omitting an argument from a long calling sequence, or by inverting arguments in a list (thereby, for example, telling a subroutine to interpret a REAL as an INTEGER — of order 10^9). I can vouch for these problems from long, sad experience at debugging FORTRAN.

Further, modern FORTRAN has been defined by accretion, with additions designed not to obsolesce older methods of doing things. Thus FORTRAN has several ways to define functions, such as through external function subprograms and through inline definitions like BASIC. And it has several ways to allocate memory for arrays. Data types can be changed explicitly via functions and implicitly via replacement statements, leading to such redundancies as

```
A = FLOAT(K)
```

being the same as

```
A = K
```

or

```
K = IFIX(A)
```

meaning

```
K = A
```

FORTRAN has several overlapping control structures: GOTO, computed GOTO, assigned GOTO, and ASSIGN are horrible relics of a an ancient past, leading to exceptionally unstructured, unmaintainable, and unreadable code. And in addition to the semi-modern logical IF THEN ELSE, FORTRAN retains the antediluvian numerical IF. The "WHILE" structures of more modern languages: BEGIN WHILE REPEAT, DO WHILE, WHILE WEND, etc. are a closed book to FORTRAN. They have to be simulated by IFs and GOTOs.

Finally, and perhaps worst, FORTRAN imposes a high overhead on subroutine and function calls, thereby discouraging decomposition of problems into small, single-purpose subroutines. Conversely, because the subroutines tend to be long their argument lists are also long (this is part of the overhead in CALL), providing fertile soil for the germination, transposition, and propagation of bugs.

The Eventual Triumph of Forth

Every operation that FORTRAN is capable of can be programmed easily in Forth. But Forth can not only do anything FORTRAN can, and using less memory, compiling much faster and perhaps executing faster as well, but Forth can do things that FORTRAN can not. I have given some examples above, and my book **[15]** contains many more.

But there are several hindrances to the general acceptance of Forth by the FORTRAN-using, number-crunching community that converts like me are zealously working to overcome. First is the lack of a generally accepted standard for floating-point arithmetic. This makes it virtually impossible to define standard libraries of useful routines, such as exist in FORTRAN. It further hinders large programming projects because Forth programmers tend to develop idiosyncratic, and largely incompatible, styles and notations. Second many FORTRAN users find RPN an-

tithetic, despite its familiarity from Hewlett-Packard calculators. I find especially that the most common Forth matrix notation for placing the address of a matrix element on the stack,

```
m n Mat.Name
```

is unacceptable to most number-crunchers. Third, the formula translator in FORTRAN is extremely convenient, and many find it hard to give it up for an entirely new style.

What may be done to remove these stumbling-blocks? While aware of a variety of innovations, I can speak most confidently of those I have been concerned with personally. First needed is a standard set of floating-point operators, based on a separate floating-point stack, or fstack. These must include standard words for formatting, inputing and displaying floating-point numbers, including standard exponential notation as in FORTRAN. I fortunately did not have to work out most of these myself, as they were supplied with my Forth system. [16] However, I have reworked and customized some of the code to more closely resemble the functions in a standard FORTRAN mathematical library. But since the words are not standardized, I cannot share code with those using other systems. I cannot overemphasize that a standard floating-point lexicon must become part of Forth. I do not imply that floating-point must be part of the kernel, but provision must be made for a standard extension. Particularly useful would be extension of NUMBER (perhaps by vectoring it) to recognize floating-point input and place it on the fstack. Second, a standard for complex arithmetic is also badly needed. We all tend to implement our own versions, [17] and this ruins portability. Third, names are important: I suggest names for complex operators beginning with X, rather than C (which is widely used for byte operators).

As mentioned above, I have worked out a system for using typed data with generic operators. At present I include more types than are strictly necessary, but certainly `REAL`, `COMPLEX`, `REAL*8` and `COMPLEX*16` must be included. My operators have forms like `G@` and `G!` , `G*` , `G/` , `G+` , `G-` , `1/G` , and so on, and use fast vectoring to decide which operation to use at run-time. My original aim was to reduce the burden of remembering which of the many possible `@` and `!` operators to use in a program. The unexpected benefit was that a word defined using generic operations would work fine with any type of data.

Elsewhere [18] I have proposed a notation reminiscent of FORTRAN:

```
Vec.Name{ n }

Mat.Name{{ m n }}
```

where the `{` in `Vec.Name{` and the `{{` in `Mat.Name{{` are mnemonic, but `}`and `}}` are operators that compute the address. The need is obviously felt by others, as I have seen similar addressing conventions proposed in other articles on Forth.

Finally, the programming style encouraged by Forth tends to obviate long expressions in algebraic or semi-algebraic notation. Nevertheless, a FORTRAN to Forth translator, beginning with formulas, could be extremely useful, [19] especially to beginning Forth programmers.

Conclusion

Forth can be extended to be an excellent language for number-crunchers. It can do everything that FORTRAN can, and much more. But most Forth designers are unaware of the many potential users in science and engineering. That has led to a neglect of essential functions, such as floating-point, complex arithmetic, and easily accessed arrays and matrices. And generic arithmetic operators along with the ability to translate (compile) algebraic notation would be of great help. Most FORTRAN users recognize its age and inconvenience, but don't go to Forth because, most systems lack what they need. When Forth does give what's needed it is the best and could become the most widely used language for number-crunching.

Footnotes and References

1. As all Forth mavens know, its inventor, Charles Moore, for reasons of speed and elegance eschewed software floating-point in favor of rescaled integers. Thus bare Forth has no floating-point. The ready availability of numeric coprocessors for all sorts of computers voids the argument against floating-point.
2. Translation: "Go away and leave us to serious programming concepts."
3. HS/FORTH, LMI's FORTH and MMSFORTH, to my knowledge, have floating-point extensions.
4. The ASYST system (ASYST Software Technologies Inc., 100 Corporate Woods, Rochester, NY 14623) is a very extended Forth system specifically tailored for the IBM PC family, with graphing and data analysis functions built in.
5. Cf. D. Gedeon, "Complex Math in Pascal", *Byte Magazine*, July 1987, p. 121.
6. The Microsoft FORTRAN version takes almost twice as long, because the innermost loop cannot be optimized this way without a great deal of trouble.
7. Definicon Systems Inc., 1100 Business Center Circle, Newbury Park, CA 90320.
8. Based on the quoted time of 3ms for a 64 point FFT on the 20 MHz T800 (Pete Wilson, "Floating-Point Survival Kit", *Byte Magazine*, March 1988, p. 220.)
9. W. H. Press, B. P. Flannery, S. A. Teukolsky and W. T. Vetterling, *Numerical Recipes: The Art of Scientific Computing*, Cambridge University Press, Cambridge, 1986, p. 292ff.
10. These are described by me in an article "Data Structures for Scientific Forth Programming" submitted to JFAR and in more detail in my forthcoming book *Scientific Forth: A Modern Language for Scientific Computing.*
11. Obtained by solving a transcendental equation, usually.
12. Thus the table is traversed many times, but in many different orders.
13. i.e., by avoiding them!
14. Michael Ham, *Dr. Dobb's Journal*, October, 1986.
15. *Scientific Forth: A Modern Language for Scientific Computing.*
16. HS/FORTH by Harvard Softworks, P.O. Box 69, Springboro, OH 45066
17. See, e.g., G.-E. April, *Journal of Forth Applications and Research*, Vol.5, No.1, pp. 79-82. Rochester, NY : Institute for Applied Forth Research, Inc., 1988.
18. "Data Structures for Scientific Forth Programming", submitted to JFAR.
19. I am writing one, using my techniques of typed data and generic operators. Formulas are much harder than loops or IF ... THEN. GOTOs and other antiquated control structures are the hardest, as Forth has no analogs of them.