Floating Point Arithmetic

Contents

This chapter discusses various aspects of floating point arithmetic in FORTH. Our approach assumes the central processor (CPU) has a dedicated floating point co-processor (FPU) available to it, such as the 80x87 for the Intel 80x86 family; the built-in FPU on the 80486; the 68881/2 for Motorola 680x0 machines; various add-ins and clones like the Weitek, Cyrix, IIT and AMD chips; or digital signal processing and array-processing co-processor boards.

If no floating point co-processor were available, one could employ co-processor emulation routines. This is the approach taken in commercial software written in FORTH, such as the (unfortunately now-deceased) VP-Planner spreadsheet. Since this text is not a *vade mecum* for writing commercial software, but rather a handbook for using FORTH to solve computational

problems in science and engineering, we consider a co-processor essential.

§1   Organization of floating point arithmetic

FORTH was originally invented as a language for controlling machinery. It is still used extensively for this purpose, with the machines in question being as varied as industrial robots, laboratory instruments, the Hubble Space Telescope, special effects motion picture cameras, and other computers. The floating point co-processor in a typical computer is a *machine*, and any numerical calculation with floating point or complex numbers, *e.g.* can be organized in terms of loading operands into the coprocessor, and transferring results from it to memory. That is, FORTH can *control* the FPU through the calculation, as indicated in Fig. 3-1 below:
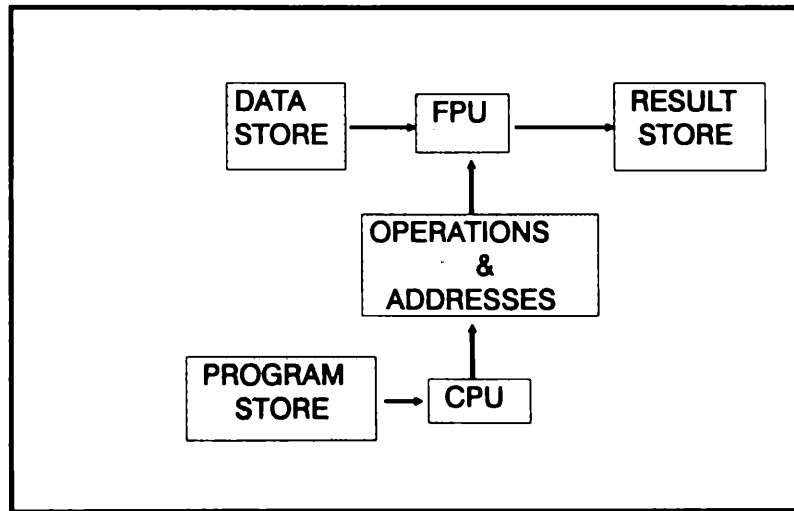

Fig. 3-1 *Data flow diagram of a CPU controlling an FPU through a calculation.*

We assume a stack for floating point numbers separate from the parameter or return stacks. We call this the **fstack**, and assume i has arbitrary depth. (We denote it by :: in stack comments.)

> Whether the fstack should be distinct from the parameter stack is currently a subject of lively debate within the FORTH community. One faction wishes to combine the two. The other faction, including the author and most other FORTH number-crunchers, believes that to organize a floating point- intensive calculation as data flow through a dedicated coprocessor, the parameter stack must be reserved for addresses, loop indices and flags. The data fed to the coprocessor therefore has to stay elsewhere, *i.e.* in the data store and the fstack.

The words we shall need fall into the categories of fstack manipulation, special constants, arithmetic, tests, and mathematical functions. Their names are nearly standard[1].

§§1 Fstack manipulation

The fstack words are[2]

F@	(addr - - :: - - x)
F!	(addr - - :: x - -)
FDUP	(:: x - - x x)
FSWAP	(:: y x - - x y)
FDROP	(:: x - -)
FROT	(:: zyx - - yxz)
FOVER	(:: yx - - yxy)
S->F	(n - - :: - - n)
D->F	(d - - :: - - d)
F->S	(:: x - - : - - int[x])

---

1. That is, the names in MMS FORTH, HS/FORTH, UniFORTH and PCFORTH are nearly the same and close to the proposed FORTH Vendors' Group (FVG) standard — see, *e.g.*, Ray Duncan's and Martin Tracy's article in Dr. Dobb's Journal, September 1984, p. 110. The new ANSII standard does not address floating point, hence it is likely the FVG standard will become pre-eminent by default.

2. F@ and F! stand for a suite of words for fetching and storing 16-, 32-, 64-, and 80-bit numbers from/to the fstack. In HS/FORTH they have names like I16@ I16! I32@ I32! R32@ R32! R64@ R64! R80@ R80! and equivalents with suffices E or L, that transfer data from segments other than the data segment.

```
F->D            ( :: x - - : - - dint[x])
%               ( place a FP# from input stream on fstack)
```

To these we sometimes add[3]

```
: FUNDER   FSWAP   FOVER  ;
: FPLUCK   FSWAP   FDROP ;
FnX         ( n = 2 - 6 | defined in code for speed)
FnR         ( n = 3 - 7 | defined in code for speed)
```

The Intel mathematics co-processors 80x87 (8087/80287/80387) and their clones incorporate a stack of limited depth (in fact 8 deep), the **87stack**. It is far faster to get a number from the 87stack than from memory. Thus, as Palmer and Morse[4] emphasize, optimizing for speed demands maximum use of the 87fstack to store intermediate results, frequently used constants, *etc.*

In Ch. 4 §7 we show how to extend the 87fstack into memory. The cost of unlimited fstack-depth is reduced speed when the 87stack spills over to memory.

§§2   Special constants

In defining various floating point operations it is convenient to be able to place certain constants on the fstack directly, by invoking their names. Here are some words that have proven useful:

```
F = 0                    ( :: - - 0)
F = 1                    ( :: - - 1)
F = PI                   ( :: - - pi = 3.14159...)
F = L2(10)               ( :: - - log₂ 10)
F = L2(E)                ( :: - - log₂ e)
F = L10(2)               ( :: - - log₁₀2)
F = LN( 2)               ( :: - - logₑ2)
```

---

3.   **FnX** (exchange ST(n) and ST(0) on fstack) and **FnR** (roll ST(n) to ST(0) on fstack) are part of HS/FORTH's floating point lexicon.

4.   J.F. Palmer and S.P. Morse, *The 8087 Primer* (John Wiley and Sons, Inc., New York, 1984). Hereinafter called **8087P**. Basically the same principle holds for other coprocessors such as the Weitek 1167 or Transputer 800 that incorporate intrinsic fstacks. The Motorola 68881/2 have registers, hence we must synthesize an fstack in software for these chips.

§3   Arithmetic operators

As noted in Chapters 1 and 2, FORTH arithmetic operators are words —*dumb* words. FORTH uses a distinct set of operators for each kind (16-bit integer, 32-bit integer, REAL, COMPLEX) of arithmetic, so the compiler has nothing to decide.

Languages like FORTRAN, BASIC and APL **overload** arithmetic operators — their meanings are context-dependent. This makes it possible to write –say– a FORTRAN expression using REAL*4, REAL*8, COMPLEX*8 or COMPLEX*16 literals and variables without worrying about how to fetch, store or convert them. Operator overloading increases the complexity of compilers and limits the speed and efficiency of compilation.

As we shall see in Chapter 5§1, FORTH enables an alternative solution in which "smart" data "know" their own types and "smart" operators "know" what kinds of data are being combined. The slightly reduced execution speed is offset by improved flexibility: *one* canned routine can work with all data types. Even better, adding this kind of "intelligence" makes no extra demands on the FORTH compiler.

The standard, dumb FORTH floating-point arithmetic operations and their actions are

F+	(:: y x - - y + x)
F–	(:: y x - - y–x)
FR–	(:: y x - - x–y)
F*	(:: y x - - y*x)
F/	(:: y x - - y/x)
FR/	(:: y x - - x/y)
FNEGATE	(:: x - - –x)
FABS	(:: x - - |x|)
1/F	(:: x - - 1/x)

To these it is sometimes useful to add words that do not consume all their arguments, such as **F*NP** (floating multiply, no pop)

F*NP   (:: x y - - x y*x) ,

that are faster, more convenient, and less demanding of the 87stack than the phrase **FOVER F***.

§§4   Example: evaluating a polynomial

L et us now write a little program to calculate something using the floating point lexicon, say a program to evaluate a general polynomial $P_N(x)$. The formula to evaluate is

$$P_N(x) = a_0 x^0 + a_1 x^1 + \ldots + a_N x^N$$

$$= a_0 + x\left(a_1 + x(\ldots + x a_N)\right)$$

The algorithm can be represented by the (pseudo) FORTH flow diagram[5], where ▨ indicates the end of the program.


```
    ⊗    0 N 1-  DO   \ from I = N-1 to 0

         sum = sum * x + a[I]

    ●    -1   +LOOP

         sum = a[N]

   ▨     EXIT
```
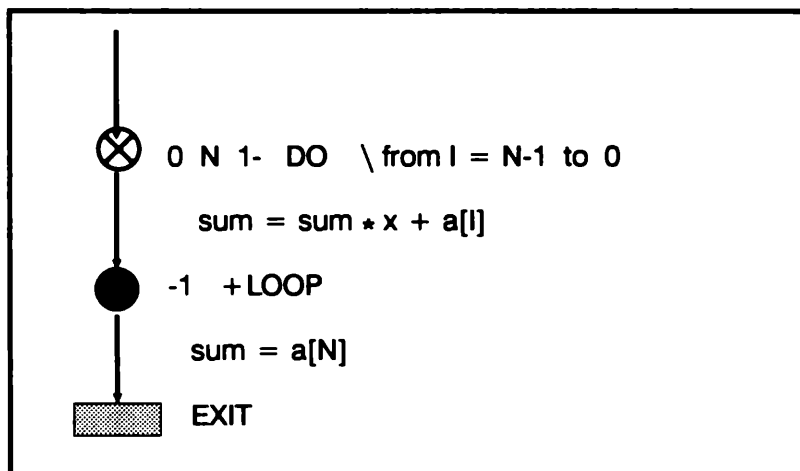
Fig. 3-2 *Pseudo FORTH flow diagram of polynomial evaluation*

Now we translate Fig. 3-2 above into FORTH[6,7] :

---

5.   See, *e.g.*, SF. Lines trace program flow; a branch indicates a decision; at a loop ⊗ marks the beginning and • the branch back to the beginning, as in BEGIN...REPEAT or DO...LOOP.

6.   We assume arrays have been defined using the intrinsically typed data structures and generic operators of Ch. 5.

7.   The notation [x] means "the address of x".

```
: }POLY   ( [a{] [x] - - )       \ evaluate p(x,N)
    FINIT     G@               \ x on fstack
    DUP    type@  G = 0       ( : : - - x  sum = 0 )
    LEN@                      ( - - [a{]  N )
    1  SWAP  DO               \ begin loop
      DUP  I }  G@            ( : : - - x  sum  a[i] )
      G +   GOVER  G*         ( : : - - x  sum' )
    -1  + LOOP                \ end loop
    GPLUCK                    ( : :  - - sum )
    0 }  G@  G +   ;          ( : : - - p[x,n] )

\ Say:    A{ X }POLY
```

Note that the function }**POLY** expects the addresses of its arguments on the stack, consumes them and leaves its result on the fstack. User-defined FORTH functions will in general have an interface of this sort. This will be especially true of the functions built into numerical co-processors.

Actually, such behavior is typical of subroutine linkage in most high level languages, as anyone knows who has written assembler subroutines that can be linked to compiled FORTRAN, C or BASIC. So FORTH really isn't different, only more explicit and efficient.

§§5   Optimizing: FORTH vs. FORTRAN

A simple-minded compiler will translate an expression such as

$$y = \left(sin(x)\right)^2$$

into a form requiring two function calls:

    Y = SIN(X)*SIN(X) .

Obviously this is silly. One of the claims often made for "optimizing" FORTRAN or C compilers is the ability to recognize an expression requiring unnecessary function calls, and to re-express it as, say,

    TEMP = SIN(X)
    Y = TEMP * TEMP .

A globally optimizing compiler has a more extensive repertoire; usually it can recognize static expressions ("invariant code") within a loop and move them outside; and it can find and eliminate code that is never evaluated ("dead" code).

FORTH assumes a good programmer *never* overlooks trivial optimizations like this. Thus nothing in the FORTH incremental compiler or optimizer is inherently capable of recognizing silly code and eliminating it.

Optimization in FORTH takes one of several forms, that can be combined for best results. The simplest is the use of stacks and registers to avoid extra memory shuffling. Referring to the preceding bad example, we note that a simple floating-point function $f(x)$ finds its argument x on the top of the fstack, consumes it, and leaves the result in its place. A simple F**2, defined as

   : F**2   FDUP F* ;

will then evaluate $[f(x)]^2$, with no fetch/store penalty from defining a temporary variable.

Some FORTHs can optimize by substituting inline code for jumps and returns to subroutines. In other words, by making the compiled code longer, some advantage in speed can be gained. HS/FORTH offers a recursive-descent optimizer of just this sort that –within its limitations– can optimize as well as good C or FORTRAN compilers. An optimizer-improved word consists of all the code bodies of the words in its definition, jammed end to end and with redundant pushes and pops deleted.

Virtually all FORTH implementations have a built-in assembler that permits defining a word in machine language. Judiciously machine-coding selected words can dramatically reduce execution time, since careful hand coding offers the ultimate performance the machine is capable of. Some versions of Pascal and C also have this ability; and of course most compiled and linked languages can link to functions and subroutines defined in machine code.

FORTH's advantage over other languages lies in making the process of designing, testing and linking hand- coded components nearly painless.

Another advantage of FORTH over other compiled languages is that one can specify which parts to optimize and which to leave as high-level definitions. This is both faster to compile and much more compact, than optimizing all of the program uniformly. The rationale of partial (sometimes called "peephole") optimization is that most programs spend 90% of their execution time in 10% of the code. This 10% is the only part of the program worth optimizing[8].

§2   Testing floating point numbers

Analogous to the test words for integer arithmetic, we require the words **F0> F0= F0< F> F= F<** .

Test words leave a flag on the parameter stack depending on the relationship they discover. Moreover, these words consume one or two arguments on the fstack, following the standard FORTH practice. As a simple first example of test words, let us define **FMAX** and **FMIN** analogous to **MAX** and **MIN** :[9]

```
: XDUP  FOVER  FOVER  ;
: FMAX  XDUP  F<  IF  FSWAP  THEN  FDROP ;
: FMIN  XDUP  F>  IF  FSWAP  THEN  FDROP ;
```

§3   Mathematical functions — the essential function library

Scientific programming in FORTH requires a suite of exponential, logarithmic and trigonometric functions (included with all FORTRAN systems, most BASICs, C's, APL, LISP, *etc.*) The minimal function library is

---

8.   An example is discussed in Chapter 9, where the innermost loop of 3 nested loops is optimized (even hand-coded), and it is seen from the timings that little is to be gained by optimizing the next outer loop.

9.   XDUP is called CPDUP in HS/FORTH's complex arithmetic lexicon.

FSQRT	(:: x - - √x)
FLN	(:: x - - ln[x])
FLOG	(:: x - - $\log_{10}$ [x])
F2**	(:: x - - 2^x)
F**	(:: x y - - y^x)
FEXP	(:: x - - e^x)
FSIN	(:: x - - sin[x])
FCOS	(:: x - - cos[x])
FTAN	(:: x - - tan[x])
DEG->RAD	(:: x - - x*pi/180)
RAD->DEG	(:: x - - x*180/pi)
FATAN	(:: x - - atan[x])
FASIN	(:: x - - asin[x])
FACOS	(:: x - - acos[x])

Machine code definitions of the above functions for the 80x87 chip will be given in Chapter 4.

§4 Library extensions

The minimal function library is easily extended. We illustrate below with the **FSGN** function and with hyperbolic and inverse hyperbolic functions. Complex extensions of the function library is deferred to Chapter 6.

§§1 The FSGN function

The most useful form of **FSGN** finds one argument *n* (from which to take an algebraic sign) on the parameter stack, and the floating point argument *x* on the fstack. We may define it using logic, as

```
: FSGN  ( n - - :: x - - sgn[n]*abs[x])
    FABS  0<  IF  FNEGATE   THEN ;
```

§§2 Cosh, Sinh and their inverses

We now code the hyperbolic sine and cosine. The formulae are

$$sinh(x) = \frac{1}{2}\left(e^{x} - e^{-x}\right)$$

$$cosh(x) = \frac{1}{2}\left(e^{x} + e^{-x}\right)$$

and their definitions are

```
: F2/    F = 1    ( :: x - - x/2)
    FNEGATE    FSWAP  FSCALE   FPLUCK  ;
: HYPER    FEXP  FDUP  1/F ;       ( :: - - e**x  e**-x )
: SINH   HYPER   F-  F2/ ;
: COSH   HYPER   F+  F2/ ;
```

The hyperbolic tangent is then

```
FIND XDUP  0 = ?( :  XDUP    FOVER    FOVER   ; )
                \ conditionally compile XDUP
: TANH     HYPER      ( :: - - e x  e-x )
    XDUP  F-     F-ROT   F+    F/   ;
```

Finally, the inverse hyperbolic sine and cosine can be defined in terms of logarithms:

$$arcsinh(x) = \ln\left(x + (x^2 + 1)^{1/2}\right) \quad , \quad -\infty < x < \infty$$

$$arccosh(x) = \ln\left(x + (x^2 - 1)^{1/2}\right) \quad , \quad -\infty < x < \infty$$

The corresponding definitions are[10]

```
FIND F**2  0 = ?( :  F**2   FDUP   F* ; )
: ARCSINH   FDUP  F**2  F = 1  F+
    FSQRT   F+   FLN ;
: ARCCOSH  FDUP  F**2  F = 1  F-
    FDUP   F0 <
    ABORT" x < 1 in ARCCOSH"
    FSQRT  F+   FLN  ;
```

───

10.   Note the test for $x^2 \geq 1$ in ARCCOSH, to prevent an error in FSQRT.

§5   Pseudo-random number generators (PRNG's)

The subject of computer-generated (pseudo) random numbers has been discussed extensively in the literature of computation[11,12]. We shall confine ourselves here to translating two useful algorithms into FORTH, and discussing tests for pseudo-random number generators (PRNG's).

The first is a method called GGUBS[13] based on the recursion

$$r_{n+1} = 16807 \times r_n \ \text{MOD} \ (2^{31} - 1).$$

Since 32-bit modulo arithmetic is inefficient on a 16-bit processor, the program uses the 80x87 chip, and uses synthetic division to get N MOD $(2^{31} - 1)$. A version that uses the 32-bit registers of the 80386/80486 would not be hard to program.

Two specialized words are needed[14], that fetch/store 32-bit integers to/from the fstack from/to memory:

```
CODE I32@   < % 9B DB 07 5B 9B % >  END-CODE
CODE I32!   < % 9B DB 1F 5B 9B % >  END-CODE
```

The program data are stored in variables rather than registers so they can be moved directly to the co-processor[15].

```
DVARIABLE     BIGDIV
21474.83647   BIGDIV   D!  \ 2**31-1
DVARIABLE     DIVIS
1277.73       DIVIS   D!
DVARIABLE     SEED
VARIABLE      M1    16807 M1  !
VARIABLE      M2    2836  M2  !
```

11. D.E. Knuth, *The Art of Computer Programming, v. 2: Seminumerical Algorithms*, 2nd ed. (Addison-Wesley Publishing Company, Reading, MA, 1981), Ch. 3.

12. R. Sedgewick, *Algorithms* (Addison-Wesley Publishing Company, Reading, MA, 1983).

13. P. Bratley, B.L. Fox and L.E. Schrage, *A Guide to Simulation* (Springer-Verlag, Berlin, 1983).

14. We anticipate using the FORTH assembler – see Ch. 4 §1§§1.1.

15. Of course, it would have been feasible to define, *via* **CREATE ... DOES >**, data types that place the data on the 87stack, *e.g.* : **FICONSTANT   CREATE , DOES > I16@** ; But such words can't be optimized; to optimize time-critical word(s) in assembler requires **VARIABLEs**.

The high-level FORTH program itself is[16,17]

```
: RAND                         ( :: - - seed)
    FINIT      SEED  DUP I32@
    DIVIS      I32@
    XDUP       F/
    FTRUNC  FRNDINT
    FUNDER       F*  FROT  FR-
    M1    I16@  F*
    FSWAP    M2  I16@
    F*  F-  FDUP  I32!  ;
: RANDOM                       ( :: - - random#)
    RAND  BIGDIV  I32@      ( :: - - seed 2**31-1)
    FSWAP  FDUP  F0<       ( - - f  :: - - 2**31-1 seed)
    IF  FOVER    F +   THEN  FR/   ;
BEHEAD" BIGDIV  RAND
\ make BIGDIV, RAND local
```

To test the algorithm start with the seed 1, and generate 1000
prn's. The result should then be 522329230.

```
: GGUBS.TST  0.1  SEED   D!
    1000  0 DO   RANDOM   LOOP
    SEED  D@   D.   ;
```

GGUBS.TST 522329230 ok

---

16. The new words are **FINIT** (initialize 80x87 chip), **FTRUNC** (set roundoff mode to truncate float-
     ing point numbers toward zero); and **FRNDINT** (round flotaing point number to integer). The
     80x87 idiosyncratically possesses several roundoff policies: A policy is selected by setting bits 10
     and 11 (numbering from 0) of the 16-bit control word using **HEX 0C00 OR** . See **8087P** for
     details.

17. **BEHEAD"** is one of several HS/FORTH words that remove dictionary entries and reclaim lost
     space. **BEHEAD'** beheads one word, **BEHEAD"** beheads all words, inclusive, in a range: in this
     case, **BIGDIV, DIVIS, SEED, M1, M2** and **RAND.**

§§1   Testing random number generators

When defining PRNGs it is always important to include a test for randomness. The simplest is called the χ^2 test: use the PRNG to generate N integers[18] in the range $[0,n\text{-}1]$ and record the number of occurrences, f_s, of each integer $s = 0, 1, \ldots n\text{-}1$. If the PRNG is really random, then the probability that an integer should have any of the n values is $1/n$, hence the expected frequencies are $<f_s> \equiv \lambda = \dfrac{N}{n}$. The χ^2 statistic for f_s is defined to be

$$\chi^2 = \sum_{s=0}^{\infty} \frac{1}{\lambda}\left(f_s - \lambda\right)^2 \;, \; \lambda = \frac{N}{n} \qquad (2)$$

χ^2 should have a value roughly n[19]. GGUBS passes the χ^2 test: A program to calculate this statistic (with $N = 1000$ and $n = 100$) is

```
CREATE   FREQS   200   ALLOT   OKLW
: IRAND   RANDOM   100   S->F
    F*   FROUND-   F->S ;
: INIT-FREQS    \ initialize freqs array
    FREQS   200   0   FILL ;

: GET-FREQS    \ make frequency table
    1000 0 DO   IRAND   2*
        DUP   199   > ABORT" IRAND TOO LARGE"
        FREQS   +   1+!
    LOOP ;
: START.CHISQ   INIT-FREQS   GET-FREQS   ;
: CHISQ   0   \ sum on stack
    200 0 DO  I  FREQS   +   @
              10 -   DUP   *   +
    2    +LOOP ;

: .FREQS          \display distribution
    200 0 DO  I  FREQS   +   @   I  CR   .   .
    2      +LOOP ;
```

---

18. *N* is assumed $>> n$

19. For those interested in details, the probability a PRNG will produce the integer "s" f times is $p_f = \lambda_f e^{-\lambda}/(f!)$. Thus the expected value of $(f-\lambda)^2/\lambda$ is 1, and therefore the expected value of χ^2 is *n*. The variance in the χ^2 statistic should then be $n(2+\lambda^{-1})$.

The results are displayed below in Table 3-1 on page 59. The mean of these χ^2 values is 99.1, and their variance σ^2 is 210. This agrees remarkably well with the theoretical formula

$$\sigma^2 = n\left(2 + \frac{1}{\lambda}\right),$$

when $n = 100$ and $\lambda = 10$.

Table 3-1 *Values of χ^2 for GGUBS*

START.CHISQ CHISQ . 1114 ok	START.CHISQ CHISQ . 1130 ok
START.CHISQ CHISQ . 840 ok	START.CHISQ CHISQ . 1064 ok
START.CHISQ CHISQ . 1294 ok	START.CHISQ CHISQ . 966 ok
START.CHISQ CHISQ . 650 ok	START.CHISQ CHISQ . 1052 ok
START.CHISQ CHISQ . 990 ok	START.CHISQ CHISQ . 760 ok
START.CHISQ CHISQ . 994 ok	START.CHISQ CHISQ . 842 ok
START.CHISQ CHISQ . 1072 ok	START.CHISQ CHISQ . 976 ok
START.CHISQ CHISQ . 1110 ok	START.CHISQ CHISQ . 1064 ok
START.CHISQ CHISQ . 1180 ok	START.CHISQ CHISQ . 950 ok
START.CHISQ CHISQ . 860 ok	START.CHISQ CHISQ . 956 ok
START.CHISQ CHISQ . 1080 ok	START.CHISQ CHISQ . 892 ok

In one application GGUBS was unsatisfactory because it contained correlations not revealed by the above tests. This led me to seek another PRNG with — perhaps — better properties, a longer cycle, *etc.* I offer it as an alternative, since — at the very least — it will enable the reader to test his applications with more than one PRNG[20]. Here is the second PRNG:

```
\ PRNG -- B.A. WICHMAN & I.D. HILL, BYTE 3/87
VARIABLE X  VARIABLE Y  VARIABLE Z
: RAND-INIT    1 X !  10000 Y !  3000 Z ! ;
: GEN    ( a b [n] -- n*a mod b )     \ hi-level version
      DUP >R  @  -ROT  DUP >R
         ( -- n a b  :R: -- [n] b )
      */MOD  DROP  DUP  0<
      IF  R>  +  ELSE  RDROP  THEN
      DUP  R>  ! ;
\ CODE  GEN  CX POP.  AX POP.
\    [BX] WORD-PTR  IMUL.
\    CX  IDIV.  DX  0 #V  CMP.    JGE.  POSITIVE.
\    DX  CX  ADD.
\    > > > POSITIVE.  [BX]  DX  MOV.  END-CODE
( Ex.: 171  30269  X  GEN )
```

```
: RANDOM  FINIT  FTRUNC
      171  30269  DUP  S->F  X  GEN  I16@  FR/
      172  30307  DUP  S->F  Y  GEN  I16@  FR/
      F+
      170  30323  DUP  S->F  Z  GEN  I16@  FR/
      F+  FRAC ;
\ FRAC takes the fractional part of a number
\ FTRUNC specifies rounding toward 0
```

The corresponding χ^2 results are given below in Table 3-2.

Table 3-2 χ^2 for Wichman-Hill PRNG

START.CHISQ CHISQ . 846	START-CHISQ CHISQ . 1172
START.CHISQ CHISQ . 1036	START-CHISQ CHISQ . 954
START.CHISQ CHISQ . 852	START-CHISQ CHISQ . 908
START-CHISQ CHISQ . 858	START-CHISQ CHISQ . 856
START-CHISQ CHISQ . 770	START-CHISQ CHISQ . 930
START-CHISQ CHISQ . 882	START-CHISQ CHISQ . 868
START-CHISQ CHISQ . 918	START-CHISQ CHISQ . 858
START-CHISQ CHISQ . 956	START-CHISQ CHISQ . 912
START-CHISQ CHISQ . 1202	START-CHISQ CHISQ . 952
START-CHISQ CHISQ . 1112	START-CHISQ CHISQ . 1016
START-CHISQ CHISQ . 778	

Interestingly, the mean of the χ^2 statistic for 21 tests is 93.5, perhaps a bit low, but not outrageously so; however, the variance in χ^2 is suspiciously small — only 135 *vs.* the expected 210. This may mean the distribution is excessively even!

One very useful test for randomness involves constructing a random walk — that is, a sequence of integers generated by the rule (r_n is the n'th PRN)

$$x_{n+1} = x_n + \begin{cases} 1 \text{ if } r_{n+1} > 0.5 \\ -1 \text{ if } r_{n+1} \leq 0.5 \end{cases} \tag{3}$$

and taking the discrete Fourier transform (DFT) of the se-
quence[21]. Any serial correlations will show up as periodicities,
with periods smaller than N, in the DFT of x_n .

§§2 Random data structures

The prng's we have discussed so far produce prn's uniformly dis-
tributed on the interval [0,1]. What if we want prn's that are
distributed according to the normal distribution on $(-\infty,\infty)$, or
according to some other standard distribution function of mathe-
matical statistics?

There are algorithms for generating prn's whose distribution
function is one of a few standard ones; however, in general one
must resort to brute force. We now engage in a brief mathematical
digression, before showing how prn's with distribution function[22]
$dp(\xi) = \theta(1 - \xi)d\,\xi$ can be converted to prn's with an arbitrary
distribution function $dp(x)=f(x)dx$.

We suppose there is a function $X(\xi)$ that converts uniform prn's
to prn's distributed according to $f(x)$. But if any of this is to make
sense, the **inverse function** $\xi(x)$ must also exist, since there is
nothing special about one distribution relative to another[23]. The
condition that both functions exist is $\dfrac{d\xi}{dx} \neq 0$.

Then[24]

$$f(x) = \int_0^1 d\,\xi\,\delta\left(x - X(\xi)\right)$$

$$= \int_0^1 d\,\xi\,\delta\left(x - X\left(\xi(x) + \xi - \xi(x)\right)\right)$$

(4)

which, using standard manipulations, we can evaluate as

---

21. Using, *e.g.*, the Fast Fourier Transform program from Chapter 8§2.
22. That is, random numbers uniformly distributes from 0 to 1.
23. Except in ease of computation, of course.
24. Here $\delta(x)$ is the so-called Dirac δ-function. See any standard text on distributions.

$$f(x) = \int_0^1 d\,\xi\,\delta\left[\left(\xi - \xi(x)\right)\frac{dX}{d\xi}\right] = \frac{d\xi}{dx} \qquad (5)$$

The ordinary differential equation 5 has the formal solution

$$\xi = \int_{X(0)}^{X(\xi)} dx\,f(x) \qquad (6)$$

that defines the new prn's distributed according to $f(x)dx$, if ξ is a prn uniformly distributed on [0,1]. In other words, we have to solve a (usually) transcendental equation to calculate $X(\xi)$.

Since most simulation problems demand a *lot* of prn's, it is no use solving Eq. 6 in real time. The better solution is to define a large enough table of the X's, and look them up according to ξ. In this case we actually want an integer PRNG uniformly distributed on [0,N-1], where the table has N entries.

```
\ © HARVARD SOFTWORKS 1986, ALL RIGHTS RESERVED.
HEX
1 VAR A   2 VAR B   03FF VAR MX

: RANDOM    A B +   MX   OVER   U<
       IF  MX  1+  -  THEN
       2*   MX   OVER   U<
       IF  MX  -       THEN
       B  IS  A   DUP  IS  B  ;
: RANDOMIZE  ( seed1 seed2 #bits - - )
       2 MAX  0F  MIN        \ #bits truncated to range 2-16
       1  SWAP  SAL  1-  IS  MX
       MX  MOD  IS  A
       MX  MOD  IS  B
       5 0  DO  RANDOM   DROP   LOOP  ;
```

Fig. 3-3 *PRNG supplied with HS/FORTH*

In Fig. 3-3 above we exhibit a PRNG that produces 16-bit integers uniformly distributed on $[0, 2^N - 1]$. I have no idea of its *modus operandi* or its origin, but it passes the χ^2 tests described above.

We also need a way to invoke user-defined functions: the method we have found is shown in Fig. 3-4 below.

```
VARIABLE  <F>
: USE(   [COMPILE]  '  CFA  <F>  !  ;
: F(X)     <F>   EXECUTE@  ;
BEHEAD'  <F>          \ make <F> local
```

Fig. 3-4 Protocol for function usage in FORTH

We also require a word analogous to , ("comma") that stores 32-bit floating point numbers in the parameter field of a word:

```
: F32,  HERE-L  4 ALLOT  R32! ;
\ store a 32-bit # from the fstack in the first
\ available place in the dictionary
```

With these auxiliary definitions we are in a position to define a word that creates tables of random variates and traverses them in a random order:

```
\ Defining word for tables of prn's distributed according to a given distribution:
\ P(x < X(xi) ) = xi defines X(xi).
\ Note: the first entry is automatically 0.
\ fn.name converts xi to X(xi)
\ dist.name is the name of the random N-long table created by )DISTRIBUTION

: SHAKE.UP  ( #bits -- )  TIME@ XOR  -ROT XOR  ROT  RANDOMIZE ;
                                \ randomize seeds
: )DISTRIBUTION:   DUP  LG2     ( -- N  #bits = lg[N] )
      SHAKE.UP                  \ initialize prng
      CREATE  F = 0  F32,       \ make first entry
      ( N -- ) 1 DO             \ N entry table
            I S->F  I' S->F  F/  \ xi on fstack
            F(X)  F32,           \ evaluate X(xi) and !
      LOOP
      DOES>  RANDOM  4*  +  R32@ ;
BEHEAD"  A  RANDOM              \ make these words local
\ Usage:  USE( fn.name  N  )DISTRIBUTION:  dist.name
```

In Fig. 3-5 below we exhibit a frequency plot of 10,000 random variates drawn from a (rather coarse) table of 64 entries, according to the distribution $p(x)dx = xe^{-x} dx$, together with $p(x)$.
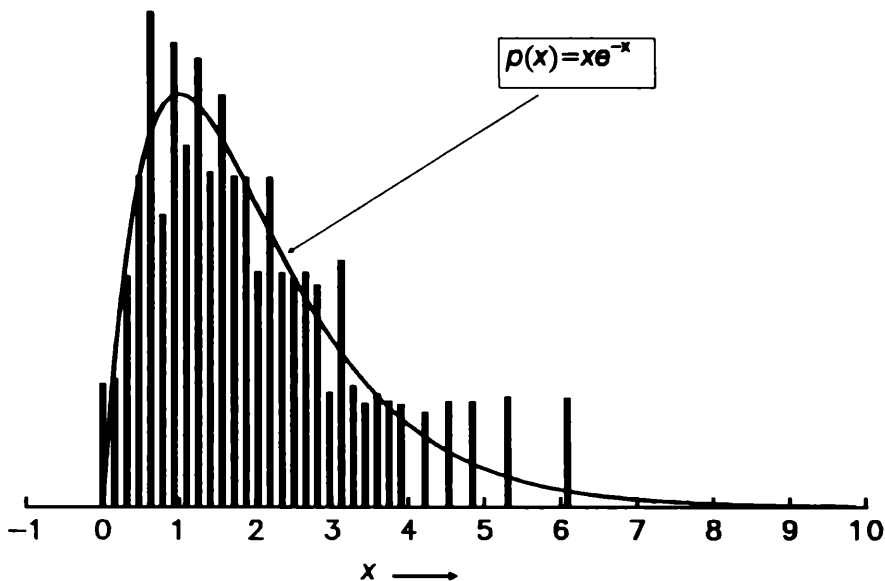
Fig. 3-5 *Random variates from table of 64 from distribution* $p(x) = xe^{-x}$

W̶ith slightly more elaboration we can arrange for each table to
have a unique prng, thus minimizing correlations between
tables. Because the resulting data structure is nearly an "object",
it is worthwhile to see how this may be done.

To make the prng unique, all that is necessary is to make the seed
VARs unique to a table, and to redefine **RANDOMIZE** and
RANDOM to know how to get a given table's seeds. We see that
RANDOM is invoked only in the runtime code for the structure
This means it has access to the base address, hence the **VAR**s can
be replaced with cells in the table, and the seeds planted there.