More Programming Examples

Contents

§1 Numerical integration 157
§§1 The integral of a function 158
§§2 The fundamental theorem of calculus 159
§§3 Monte-Carlo method 160
§§4 Adaptive methods 164
§§5 Adaptive integration on the real line 165
§§6 Adaptive integration in the Argand plane 179
§2 Fitting functions to data 181
§§1 Fast Fourier transform 184
§§2 Gram polynomials 193
§§3 Simplex algorithm 201
§3 Appendices 204
§§1 Gaussian quadrature 204
§§2 The trapezoidal rule 205
§§3 Richardson extrapolation 206
§§4 Linear least-squares: Gram polynomials 207
§§5 Non-linear least squares: simplex method 208

In this chapter we apply some of the FORTH tools we have been developing (complex arithmetic, typed data) to two standard problems in numerical analysis: numerical integration of a function over a definite interval; determining the function of a given form that most closely fits a set of data.

§1 Numerical Integration

We begin by defining the definite integral of a function f(x). Then we discuss some methods for (numerically) approximating the integral. This process is called **numerical integration** or **numerical quadrature**. Finally, we write some FORTH programs based on the various methods we describe.

§§1 The integral of a function

The definite integral $\int_a^b f(x)\,dx$ is the area between the graph of the function and the x-axis as shown below in Fig. 8-1:

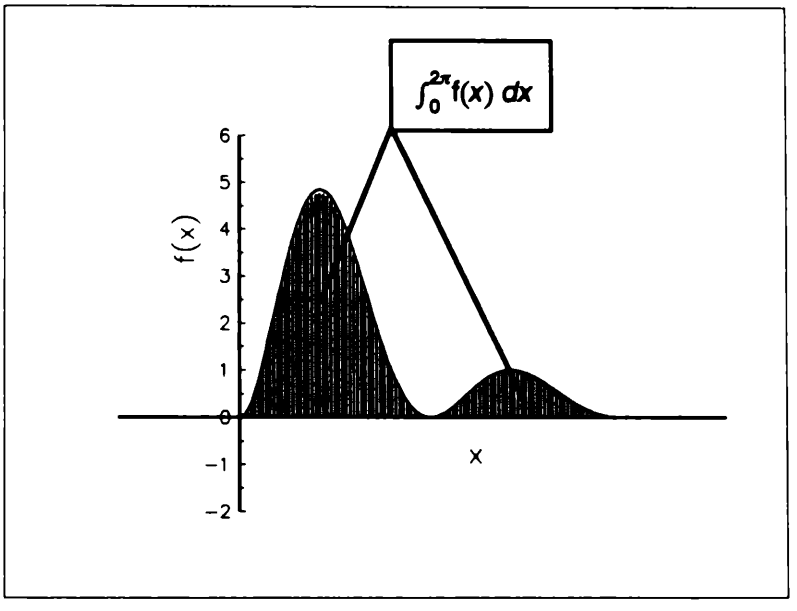


Fig. 8-1 *The integral of a function is the area under the curve.*

We estimate the integral by breaking up the area into narrow rectangles of width *w* that approximate the height of the curve at that point and then adding the areas of the rectangles[1]. For rectangles of non-zero width the method gives an approximation. If we calculate with rectangles that consistently protrude above the curve (assume for simplicity the curve lies above the x-axis), and with rectangles that consistently lie below the curve, we capture the exact area between two approximations. We say that we have **bounded** the integral above and below. In mathematical language,

---

1. If a rectangle lies below the horizontal axis, its area is considered to be negative.

$$w \sum_{n=0}^{(b-a)/w} \min\left[f(a+nw), f(a+nw+w)\right]$$

$$\leq \int_a^b f(x)\,dx \qquad (1)$$

$$\leq w \sum_{n=0}^{(b-a)/w} \max\left[f(a+nw), f(a+nw+w)\right]$$

It is easy to see that each rectangle in the upper bound is about $w|f'(x)|$ too high[2] on the average, hence overestimates the area by about $\frac{1}{2}w^2|f'(x)|$. There are $(b-a)/w$ such rectangles, so if $|f'(x)|$ remains finite over the interval $[a, b]$ the total discrepancy will be smaller than

$$\frac{1}{2}w(b-a) \max_{a\leq x\leq b} |f'(x)|.$$

Similarly, the lower bound will be low by about the same amount. This means that if we halve w (by taking twice as many points), the accuracy of the approximation will double. The mathematical definition of $\int_a^b f(x)\,dx$ is the number we get by taking the limit as the width w of the rectangles becomes arbitrarily small. We know that such a limit exists because the actual area has been captured between lower and upper bounds that shrink together as we take more points.

§§2 The fundamental theorem of calculus

Suppose we think of $\int_a^b f(x)\,dx$ as a function —call it $F(b)$— of the upper limit, b. What would happen if we compared the area $F(b)$ with the area $F(b + \Delta b)$: We see that the difference between the two is (for small Δb)

$$\Delta F(b) = F(b+\Delta b) - F(b) \approx f(b)\Delta b + O((\Delta b)^2) \qquad (2)$$

---

2. $f'(x)$ is the slope of the line tangent to the curve at the point x. It is called the first derivative of $f(x)$.

so that

$$F'(b) = \lim_{\Delta b \to 0} \frac{1}{\Delta b}\left(\int_a^{b+\Delta b} dx - \int_a^b f(x)\,dx\right) \qquad (3)$$

Equation 3 is a fancy way to say that integration and differentiation are **inverse operations** in the same sense as multiplication and division, or addition and subtraction.

This fact lets us calculate a definite integral using the differential equation routine developed in Chapter 6. We can express the problem in the following form:

Solve the differential equation

$$\frac{dF}{dx} = f(x) \qquad (4)$$

from $x = a$ to $x = b$, subject to the initial condition

$$F(a) = 0.$$

The desired integral is $F(b)$.

The chief disadvantage of using a differential equation solver to evaluate a definite integral is that it gives us no **error criterion**. We would have to solve the problem at least twice, with two different step sizes, to be sure the result is sufficiently precise[3].

§§3 Monte-Carlo method

The area under $f(x)$ is exactly equal to the average height $\bar{f}$ of $f(x)$ on the interval $[a, b]$, times the length, $b-a$, of the interval[4]. How can we estimate $\bar{f}$? One method is to sample $f(x)$ at random,

---

3. This is not strictly correct: one could use a differential equation solver of the "predictor/corrector" variety, with variable step-size, to integrate Eq. 4. See, *e.g.*, Press, *et al.*, *Numerical Recipes* (Cambridge University Press, Cambridge, 1986), pp. 102 ff.

4. That is, this statement **defines** $\bar{f}$.

choosing N points in [a,b] with a random number generator. Then

$$\bar{f} \approx \frac{1}{N} \sum_{n=1}^{N} f(x_n) \tag{5}$$

and

$$\int_a^b f(x)\,dx \approx (b-a)\bar{f} \tag{6}$$

This random-sampling method is called the **Monte-Carlo** method (because of the element of chance).

§§3–1 Uncertainty of the Monte-Carlo method

The statistical notion of **variance** lets us estimate the accuracy of the Monte-Carlo method: The variance in $f(x)$ is

$$\begin{aligned} \mathrm{Var}(f) &= \int_{-\infty}^{+\infty} \rho(f)\left(f-\bar{f}\right)^2 df \\ &\approx \frac{1}{N}\sum_{n=1}^{N}\left(f(x_n)-\bar{f}\right)^2 \end{aligned} \tag{7}$$

(here $\rho(f)df$ is the probability of measuring a value of f between f and $f+df$).

Statistical theory says the variance in estimating $\bar{f}$ by random sampling is

$$\mathrm{Var}(\bar{f}) = \frac{1}{N}\mathrm{Var}(f) \tag{8}$$

i.e., the more points we take, the better estimate of $\bar{f}$ we obtain. Hence the uncertainty in the integral will be of order

$$\Delta\left(\int_a^b f(x)\,dx\right) \approx \frac{(b-a)\sqrt{\mathrm{Var}(f)}}{\sqrt{N}} \tag{9}$$

and is therefore guaranteed to decrease as $\frac{1}{\sqrt{N}}$.

It is easy to see that the Monte-Carlo method converges slowly. Since the error decreases only as $\frac{1}{\sqrt{N}}$, whereas even so crude a rule as adding up rectangles (as in §1§§1) has an error term that decreases as $1/N$, what is Monte-Carlo good for?

Monte-Carlo methods come into their own for multidimensional integrals, where they are much faster than multiple one-dimensional integration subroutines based on deterministic rules.

§§3–2 A simple Monte-Carlo program

Following the function protocol and naming convention developed in Ch. 6 §1§§3.2, we invoke the integration routine *via*

```
USE( F.name  % L.lim  % U.lim  % err )MONTE
```

We pass **)MONTE** the name **F.name** of the function $f(x)$, the limits of integration, and the absolute precision of the answer. The answer should be left on the ifstack. **L.lim**, **U.lim** and **err** stand for explicit floating point numbers that are placed on the 87stack by **%** [5]. The word **%** appears explicitly because in a larger program – of which **)MONTE** could be but a portion – we might want to specify the parameters as numbers already on the 87stack. Since this is intended to be an illustrative program we keep the fstack simple by defining **SCALAR**s to put the limits and precision into.

```
3 REAL*4 SCALARS  A B-A  E
```

The word **INITIALIZE** will be responsible for storing these numbers.

The program uses one of the pseudo-random number generators (**prng**'s) from Ch. 3 §5. We need a word to transform prn's –uniformly distributed on the interval (0,1)– to prn's on the interval (A, B):

---

5. The word **%** pushes what follows in the input stream onto the 87stack, assuming it can be interpreted as a floating point number.

```
: NEW.X  RANDOM  B-A  G@  F*  A G@  F+ ;
```

The program is described by the simple flow diagram of Fig. 8-2 below:

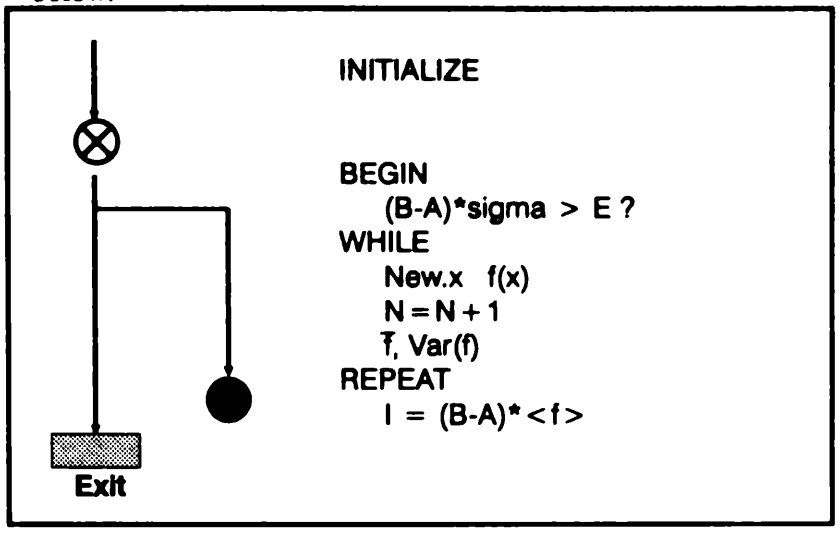


Fig. 8-2 *Flow diagram of Monte Carlo Integration*

From the flow diagram we see we have to recompute $\bar{f}$ and Var($\bar{f}$) at each step. From Eq. 5 we see that

$$\bar{f}_{N+1} = \bar{f}_N + \frac{f(x_{N+1}) - \bar{f}_N}{N+1}$$

and

$$\mathrm{Var}_{N+1} = \mathrm{Var}_N + \frac{(f_{N+1} - \bar{f}_N)(f_{N+1} - \bar{f}_{N+1}) - \mathrm{Var}_N}{N+1}$$

Writing the program is almost automatic:

```
: USE(   [COMPILE] ' CFA LITERAL ; IMMEDIATE
3 REAL*4 SCALARS  Av.F  old.Av.F  Var.F

: Do.Average     ( n -- n+1  87: f -- f )
   Av.F  G@  old.Av.F  G!   \ save old.Av
   FDUP   1+      ( -- n+1  87: -- ff )
   old.Av.F  G@   ( 87: -- ff old.Av.F)
   FUNDER  F-
   DUP  S->F  F/  F+   ( 87: -- f Av.F)
   Av.F  G!  ;          \ put away   \ cont'd below
```

```
: Do.Variance      ( n - - n 87: f - - )
  FDUP  old.Av.F G@  ( 87: f f old.Av )
  FUNDER F-  FSWAP
  Av.F G@  F- F*
  ( 87: [f-old.Av]*[f-Av] )
  Var.F G@  FUNDER  F-
  DUP  S->F  F/  F+ ( 87: - - Var' )
Var.F G!  ;

: INITIALIZE      ( : adr - - 87: a b e - - )
  IS  adr.f
  E  G!
  FOVER F-  B-A  G!     A  G!
  FINIT
  F=0   Var.F  G!
  F=0   Av.F  G!
  F=0   old.Av.F  G!
  0  5  0 DO     \ exercise 5 times
        NEW.X     adr.f  EXECUTE
        Do.Average Do.Variance
  LOOP  ;

: Not.Converged?  Var.F G@  FSQRT
  B-A  G@  F*    E  G@  F>  ;
: DEBUG  DUP
        10 MOD          \ every 10 steps
  0=  IF  CR  DUP .
            Av.F  G@ F.
            Var.F G@ F. THEN  ;
: )MONTE
  INITIALIZE
  BEGIN    DEBUG  Not.Converged?
  WHILE    NEW.X  adr.f  EXECUTE
       Do.Average      Do.Variance
  REPEAT
  DROP  Av.F  G@  B-A  G@  F* ;
```

The word **DEBUG** is included to produce useful output every 10 points as an aid to testing. The final version of the program need not include **DEBUG**, of course. Also it would presumably be prudent to **BEHEAD** all the internal variables.

The chief virtue of the program we have just written is that it is easily generalized to an arbitrary number of dimensions. The generalization is left as an exercise.

§§4 Adaptive methods

Obviously, to minimize the execution time of an integration subroutine requires that we minimize the number of times the function $f(x)$ has to be evaluated. There are two aspects to this:

- First, we must evaluate $f(x)$ only once at each point x in the interval.
- Second, we evaluate $f(x)$ more densely where it varies rapidly than where it varies slowly. Algorithms that can do this are called **adaptive**.

To apply adaptive methods to Monte Carlo integration, we need an algorithm that biases the sampling method so more points are

chosen where the function varies rapidly. Techniques for doing this are known generically as **stratified sampling**[6]. The difficulty of automating stratified sampling for general functions puts adaptive Monte Carlo techniques beyond the scope of this book.

However, adaptive methods can be applied quite easily to deterministic quadrature formulae such as the **trapezoidal rule** or **Simpson's rule**. Adaptive quadrature is both interesting in its own right and illustrates a new class of programming techniques, so we pursue it in some detail.

§§5 Adaptive integration on the real line

We are now going to write an adaptive program to integrate an arbitrary function $f(x)$, specified at run-time, over an arbitrary interval of the x-axis, with an absolute precision specified in advance. We write the integral as a function of several arguments, once again to be invoked following Ch. 6 §1.3.2:

```
USE( F.name  % L.lim  % U.lim  % err )INTEGRAL
```

Now, how do we ensure that the routine takes a lot of points when the function $f(x)$ is rapidly varying, but few when f(x) is smooth? The simplest method uses **recursion**[7].

§§5-1 Digression on recursive algorithms

We have so far not discussed recursion, wherein a program calls itself directly or indirectly (by calling a second routine that then calls the first).

Since there is no way to know *a priori* how many times a program will call itself, memory allocation for the arguments must be dynamic. That is, a recursive routine places its arguments on a

---

6. J.M. Hammersley and D.C. Hanscomb, *Monte Carlo Methods* (Methuen, London, 1964).
7. See, *e.g.*, R. Sedgewick, *Algorithms* (Addison-Wesley Publishing Company, Reading, MA, 1983), p. 85.

stack so each invocation of the program can find them. This is the method employed in recursive compiled languages such as Pascal, C or modern BASIC. Recursion is of course natural in FORTH since stacks are intrinsic to the language.

We illustrate with the problem of finding the greatest common divisor (gcd) of two integers. Euclid[8] devised a rapid algorithm for finding the gcd[9] which can be expressed symbolically as

$$\gcd(u,v) = \begin{cases} u, & v=0 \\ \gcd(v, u \bmod v) & else \end{cases} \qquad (10)$$

That is, the problem of finding the gcd of u and v can be replaced by the problem of finding the gcd of two much smaller numbers. A FORTH word that does this is[10]

```
: GCD                    ( u v - - gcd)
   ?DUP 0>               \ stopping criterion
   IF   UNDER   MOD   RECURSE   THEN  ;
```

Here is a sample of **GCD** in action, using **TRACE**[11] to exhibit the rstack (in hex) and stack (in decimal):

---

8. An ancient Greek mathematician known for one or two other things!
9. See, *e.g.* Sedgewick, *op. cit.*, p. 11.
10. Most FORTHs do not permit a word to call itself by name; the reason is that when the compiler tries to compile the self-reference, the definition has not yet been completed and so cannot be looked up in the dictionary. Instead, we use **RECURSE** to stand for the name of the self-calling word. See Note 14 below.
11. **TRACE** is specific to HS/FORTH, but most dialects will support a similar operation. **SSTRACE** is a modification that single-steps through a program.

```
784 48 TRACE GCD
                                rstack          stack
: GCD                                           784 48
  DUP                                           784 48 48
  0=                                            784 48 0
  0BRANCH<8>0                                   784 48
  UNDER                                         48 784 48
  MOD                                           48 16
  :GCD                                          48 16
    DUP                         4B76            48 16 16
    0=                          4B76            48 16 0
    0BRANCH<8>0                 4B76            48 16
    UNDER                       4B76            16 48 16
      MOD                       4B76            16 0
      :GCD                      4B76            16 0
        DUP                     4B76 4B76       16 0 0
        0=                      4B76 4B76       16 0 65535
        0BRANCH<8>-1            4B76 4B76       16 0
        DROP                    4B76 4B76       16
      EXIT                      4B76            16
  EXIT                                          16
```

Note how **GCD** successively calls itself, placing the same address (displayed in hexadecimal notation) on the rstack, until the stopping criterion is satisfied.

Recursion can get into difficulties by exhausting the stack or rstack. Since the stack in **GCD** never contains more than three numbers, only the rstack must be worried about in this example.

Recursive programming possesses an undeserved reputation for slow execution, compared with nonrecursive equivalent programs[12]. Compiled languages that permit recursion —*e.g.*, BASIC, C, Pascal— generally waste time passing arguments to subroutines, *i.e.* recursive routines in these languages are slowed by parasitic calling overhead. FORTH does not suffer from this speed penalty, since it uses the stack directly.

---

12. For example, it is often claimed that removing recursion almost always produces a faster algorithm. See, *e.g.* Sedgewick, *op. cit.*, p. 12.

Nevertheless, not all algorithms should be formulated recursively. A disastrous example is the Fibonacci sequence

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2} \tag{11}$$

expressed recursively in FORTH as

```
: FIB                                ( : n - - F[n] )
   DUP  0>  NOT
   IF   DROP  0   EXIT  THEN
   DUP  1  =
   IF   DROP  1   EXIT  THEN         \ n > 1
   1-  DUP  1-                       ( - - n-1  n-2 )
   RECURSE   SWAP                    ( - - F[n-2]  n-1 )
   RECURSE   +   ;
```

This program is vastly slower than the nonrecursive version below, that uses an explicit **DO** loop:

```
: FIB                  ( : n - - F[n] )
   0 1  ROT            ( : 0 1 n )
   DUP  0>  NOT
   IF  DDROP  EXIT  THEN
   DUP   1  =
   IF  DROP PLUCK  EXIT  THEN
   1  DO   UNDER  +   LOOP   PLUCK  ;
```

Why was recursion so bad for Fibonacci numbers? Suppose the running time for F_n is T_n ; then we have

$$T_n \approx T_{n-1} + T_{n-2} + \tau \tag{12}$$

where τ is the integer addition time. The solution of Eq. 12 is

$$T_n = \tau\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - 1\right] \tag{13}$$

That is, the execution time increases **exponentially** with the size of the problem. The reason for this is simple: recursion managed to replace the original problem by two of nearly the same size, *i.e.* recursion nearly **doubled** the work at each step!

The preceding analysis of why recursion was bad suggests how recursion can be helpful: we should apply it whenever a given

problem can be replaced by –say– two problems of **half** the original size, that can be recombined in n or fewer operations. An example is **mergesort**, where we divide the list to be sorted into two roughly equal lists, sort each and then merge them:

```
subroutine sort(list[0,n])
    partition(list, list1, list2)
    sort(list1)
    sort(list2)
    merge(list1, list2, list)
end
```

In such cases the running time is

$$T_n \approx T_{n/2} + T_{n/2} + n = 2T_{n/2} + n \qquad (14)$$

for which the solution is

$$T_n \approx n \log_2 (n) \qquad (15)$$

(In fact, the running time for *mergesort* is comparable with the fastest sorting algorithms.) Algorithms that subdivide problems in this way are said to be of **divide and conquer** type.

Adaptive integration can be expressed as a divide and conquer algorithm, hence recursion can simplify the program. In pseudocode (actually QuickBasic®) we have the program shown below:

```
function simpson(f, a, b)
    c = (a + b)/2
    simpson = (f(a) + f(b) + 4*f(c)) * (b - a) /6
end function

function integral(f, a, b, error)
    c = (a + b)/2
    old.int = simpson(f, a, b)
    new.int = simpson(f, a, c) + simpson(f, c, b)
    if abs( old.int - new.int) < error then
        integral = (16*new.int - old.int) /15
    else
        integral = integral(f, a, c, error/2) +
                   integral(f, c, b, error/2)
    end if
end function
```

Clearly, there is no obligation to use Simpson's rule on the sub-intervals: any favorite algorithm will do.

To translate the above into FORTH, we decompose into smaller parts. The name of the function representing the integrand (actually its execution address or **cfa**) is placed on the stack by **USE(**, as in Ch. 8 §1.3.2 above. Thence it can be saved in a local variable –either on the rstack or in a **VAR** or **VARIABLE** that can be **BEHEAD**ed– so the corresponding phrases

```
R@   EXECUTE          \ rstack
name EXECUTE          \ VAR
name EXECUTE@         \ VARIABLE
```

evaluate the integrand. Clearly the limits and error (or **tolerance**) must be placed on a stack of some sort, so the function can call itself. One simple possibility is to put the arguments on the 87stack itself. (Of course we then need a software fstack manager to extend the limited 87stack into memory, as discussed in Ch. 4 §7.) Alternatively, we could use the intelligent fstack (**ifstack**) discussed in Ch. 5 §2.5. We thus imagine the fstack to be as deep as necessary.

The program then takes the form[13] shown on p. 171 below.

Note that in going from the pseudocode to FORTH we replaced **)INTEGRAL** by **RECURSE** inside the word **)INTEGRAL**. The need for this arises from an ideosyncrasy of FORTH: normally words do not refer to themselves, hence a word being defined is hidden from the dictionary search mechanism (compiler) until the final **;** is reached. The word **RECURSE** unhides the current name, and compiles its **cfa** in the proper spot[14].

---

13. For generality we do not specify the integration rule for sub-intervals, but factor it into its own word. If we want to change the rule, we then need redefine but one component (actually two, since the Richardson extrapolation –see Appendix 8.C– needs to be changed also).

14. We may define **RECURSE** (in reverse order) as

```
: RECURSE  ?COMP  LAST-CFA , ;
: ?COMP    STATE @ 0= ABORT" Compile only!" ;
: LAST-CFA  LATEST PFA CFA ; IMMEDIATE
\ These defintions are appropriate for HS/FORTH
```

Note that **RECURSE** is called **MYSELF** in some dialects.

```
: USE(  [COMPILE] ' CFA LITERAL ;
IMMEDIATE

: f(x)  ( : cfa -- cfa )  DUP EXECUTE ;

: )Integral  ( f: a b -- I )
        \ uses trapezoidal rule
    XDUP  FR-  F2/    ( 87: -- a b [b-a]/2 )
    F-ROT  f(x)  FSWAP  f(x)  F+  F* ;
: )Richardson        \ R-extrap. for trap. rule
    3 S->F  F/  F+  ;   ( 87: I' I -- I" )

DVARIABLE ERR          \ place to store err
CREATE  OLD.I  10 ALLOT
                       \ place to store I[a,b]
```

```
: )INTEGRAL  ( : adr -- 87: a b err -- I )
  ERR R32!
  XDUP )Integral    ( 87: -- a b I0 )
  OLD.I R80!
  XDUP  F+  F2/ ( 87: -- a b c=[a+b]/2 )
  FUNDER FSWAP ( 87: -- a c c b )
  XDUP )Integral ( 87: -- a c c b I1)
  F4P  F4P   ( 87: -- a c c b I1 a c)
  )Integral F+  ( 87: -- a c c b I1+I2)
  FDUP  OLD.I R80@ F-
  FDUP FABS  ERR R32@  F<
  IF  )Richardson
     FPLUCK FPLUCK FPLUCK FPLUCK
  ELSE  FDROP FDROP  ( 87: -- a c c b)
       ERR R32@  F2/
     F-ROT  F2P ( 87: -- a c err/2  c b err/2 )
       RECURSE  ( 87: -- a c err/2 I[c,b] )
       F3R F3R F3R  RECURSE  F+
  THEN   DROP  ;
```

§§5–2 Disadvantages of recursion in adaptive integration

The main advantage of the recursive adaptive integration algorithm is its ease of programming. As we shall see, the recursive program is much shorter than the non-recursive one. For any reasonable integrand, the fstack (or ifstack) depth grows only as the square of the logarithm of the finest subdivision, hence never gets too large.

However, recursion has several disadvantages when applied to numerical quadrature:

- The recursive program evaluates the function more times than necessary.
- It would be hard to nest the function **)INTEGRAL** for multi-dimensional integrals.

Several solutions to these problems suggest themselves:

- The best, as we shall see, is to eliminate recursion from the algorithm.
- We can reduce the number of function evaluations with a more precise quadrature formula on the sub-intervals.

- We can use "open" formulas like Gauss-Legendre, that omit the endpoints (see Appendix 8.1).

§§5–3 Adaptive integration without recursion

The chief reason to write a non-recursive program is to avoid any repeated evaluation of the integrand. That is, the optimum is not only the smallest number of points x_n in $[A, B]$ consistent with the desired precision, but to evaluate $f(x)$ once only at each x_n. This will be worthwhile when the integrand $f(x)$ is costly to evaluate.

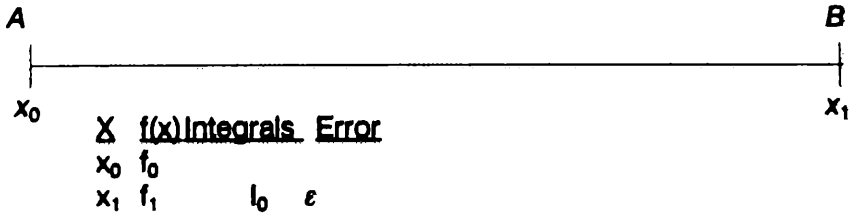
To minimize evaluations of $f(x)$, we shall have to save values $f(x_n)$ that can be re-used as we subdivide the intervals.

The best place to store the $f(x_n)$'s is some kind of stack or array. Moreover, to make sure that a value of $f(x)$ computed at one mesh size is usable at all smaller meshes, we must subdivide into two equal sub-intervals; and the points x_n must be equally spaced and include the end-points. Gaussian quadrature is thus out of the question since it invariably (because of the non-uniform spacings of the points) demands that previously computed $f(x_n)$'s are thrown away because they cannot be re-used.

The simplest quadrature formula that satisfies these criteria is the **trapezoidal rule** (see Appendix 8.2). This is the formula used in the following program.

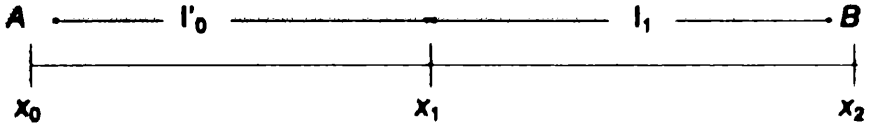
To clarify what we are going to do, let us visualize the interval of integration, and mark the mesh points (where we evaluate $f(x)$ with + :

Step 1: N = 1



We now save (temporarily) I_0 and divide the interval in two, computing I'_0 and I_1 on the halves, as shown. This will be one fundamental operation in the algorithm.

Step 2: N = N + 1 = 2



X	f(x)	Integrals	Error
x_0	f_0		
x_1	f_1	I'_0	$\varepsilon/2$
x_2	f_2	I_1	$\varepsilon/2$

We next compare $I'_0 + I_1$ with I_0. The results can be expressed as a branch in a flow diagram, shown below.

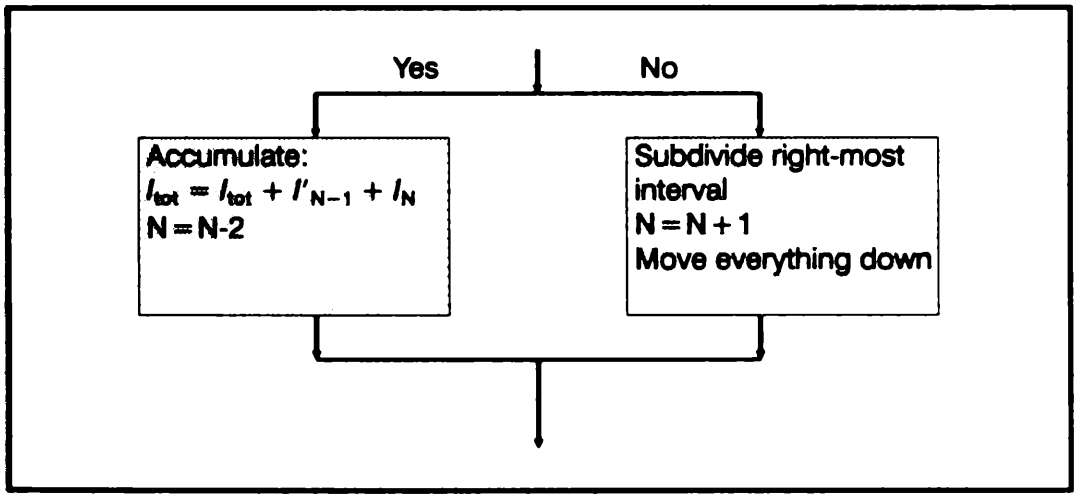
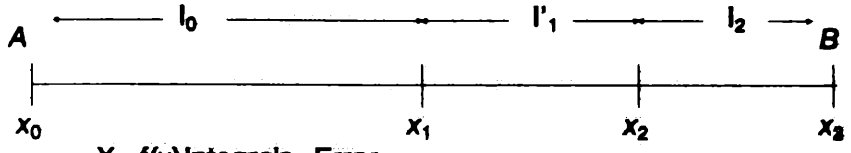


Fig. 8-3 **SUBDIVIDE** *branch in adaptive integration*

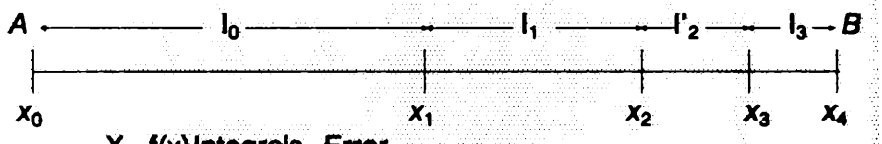
If the two integrals disagree, we subdivide again, as in Step 3 and Step 4 below:

Step 3: N = N + 1 = 3



X	f(x)	Integrals	Error
x_0	f0		
x_1	f_1	I_0	$\varepsilon/2$
x_2	f_2	I'_1	$\varepsilon/4$
x_3	f_3	I_2	$\varepsilon/4$

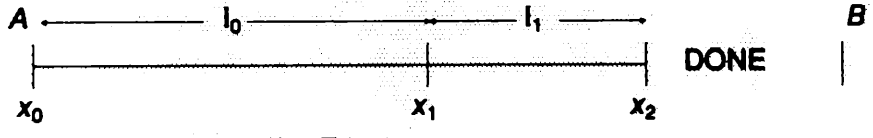
Step 4: N = N + 1 = 4



X	f(x)	Integrals	Error
x_0	f0		
x_1	f_1	I_0	$\varepsilon/2$
x_2	f_2	I_1	$\varepsilon/4$
x_3	f_3	I'_2	$\varepsilon/8$
x_4	f_4	I_3	$\varepsilon/8$

Now suppose the last two sub-integrals ($I_3 + I'_2$) in step 4 agreed with their predecessor (I_2); we then accumulate the part computed so far, and begin again with the (leftward) remainder of the interval, as in Step 5:

Step 5: N = N-2 = 2 $\quad I = I + (I'_2 + I_3) + (I'_2 + I_3 - I_2)/3$



X	f(x)	Integrals	Error
x_0	f0		
x_1	f_1	I_0	$\varepsilon/2$

The flow diagram of the algorithm now looks like Fig. 8-4 below:

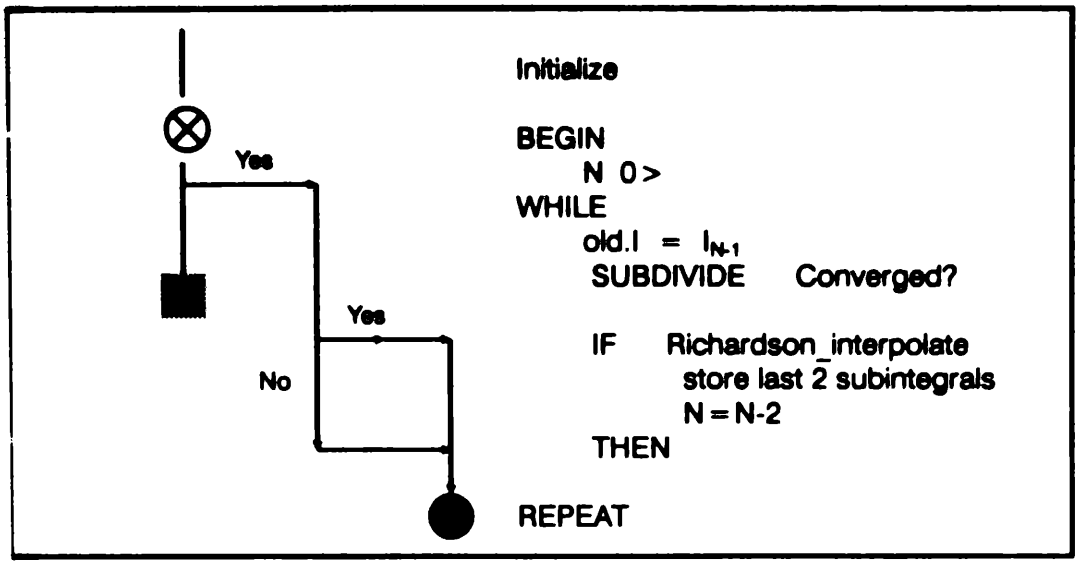


Fig. 8-4 *Non-recursive adaptive quadrature*

and the resulting FORTH program is [15]:

15. We use the generalized arrays of Ch. 5 §3.4; A, B and E are fp #'s on the fstack, TYPE is the data-type of f(x) and INTEGRAL.

```
\ COPYRIGHT 1991 JULIAN V. NOBLE
TASK INTEGRAL
FIND CP@L 0= ?( FLOAD COMPLEX )
\ define data-type tokens if not already
FIND REAL*4 0= ?(((
        0 CONSTANT REAL*4
        1 CONSTANT REAL*8
        2 CONSTANT COMPLEX
        3 CONSTANT DCOMPLEX )))

FIND 1ARRAY 0= ?( FLOAD MATRIX.HSF )
\ function usage
: USE( [COMPILE] ' CFA ; IMMEDIATE

\ BEHEADing starts here
0 VAR N

: inc.N N 1+ IS N ;
: dec.N N 2- IS N ;

0 VAR type

\ define "stack"
        20 LONG REAL*8 1ARRAY X{
        20 LONG REAL*4 1ARRAY E{
        20 LONG DCOMPLEX 1ARRAY F{
        20 LONG DCOMPLEX 1ARRAY I{

2 DCOMPLEX SCALARS old.I final.I
: )integral ( n - - )\ trapezoidal rule
        X{ OVER } G@L
        X{ OVER 1- } G@L
        F- F2/
        F{ OVER } G@L
        F{ OVER 1- } G@L
        type 2 AND
        IF X+ FROT X*F
        ELSE F+ F* THEN
        I{ SWAP 1- } G!L ;
0 VAR f.name
: f(x) f.name EXECUTE ;

: INITIALIZE
        IS type \ store type
        type F{ !
        type I{ !\ set types for function
        type ' old.I !
        type ' final.I !    \ and integral(s)
        type 1 AND X{ !
                \ set type for indep. var.
        E{ 0 } G!L      \ store error
        X{ 1 } G!L      \ store B
        X{ 0 } G!L      \ store A
        IS f.name       \ ! cfa of f(x)
        X{ 0 } G@L f(x) F{ 0 } G!L
        X{ 1 } G@L f(x) F{ 1 } G!L
        1 IS N
        N )integral
        type 2 AND IF F=0 THEN
        F=0 final.I G!L
        FINIT ;

: E/2 E{ N 1- } G@L F2/ E{ N 1- } G!L ;

: }move.down ( adr n - - )
        } #BYTES >R         ( - - seg off)
        DDUP R@ +
                ( - - s.seg s.off d.seg d.off )
        R> CMOVEL ;

: MOVE.DOWN
        E{ N 1- }move.down
        X{ N }move.down
        F{ N }move.down ;

: new.X         ( 87: - - x' )
        X{ N } G@L X{ N 1- } G@L
        F+ F2/ FDUP X{ N } G!L ;

\ cont'd. ...
```

```
\ INTEGRAL cont'd
\ debugging code
: GF.   1 >  IF FSWAP  E. THEN E. ;
: F@.   DUP >R  G@L  R>  GF. ;
: .STACKS  CR ." N"
        8 CTAB ." X"
        19 CTAB ." Re[F(X)]"
        31 CTAB ." Im[F(X)]"
        45 CTAB ." Re[I]"
        57 CTAB ." Im[I]"
        71 CTAB ." E"
        N 2+ 0 DO  CR I .
                    3 CTAB X{ I } F@.
                   16 CTAB F{ I } F@.
                   42 CTAB I{ I }  F@.
                   65 CTAB E{ I } F@.
        LOOP
        CR  5 SPACES ." old.I ="  old.I F@.
        5 SPACES  ." final.I ="  final.I F@.  CR  ;

CASE: <DEBUG>  NEXT .STACKS ;CASE
0 VAR (DEBUG)
: DEBUG-ON   1 IS (DEBUG)  5 #PLACES !  ;
: DEBUG-OFF  0 IS (DEBUG)  7 #PLACES !  ;
: DEBUG (DEBUG)  <DEBUG> ;

: SUBDIVIDE
  N 19 > ABORT" Too many subdivisions!"
  E/2  MOVE.DOWN
  I{ N 1- }  DROP  old.I #BYTES CMOVEL
    new.X  f(x)  F{ N } G!L
  N )integral  N 1+ )integral ;

: CONVERGED?  ( 87: - - I[N] + I'[N-1] - I[N-1] - - f)
        I{ N } G@L  I{ N 1- } G@L  old.I G@L
        type 2 AND
        IF      CP- CP+ CPDUP CPABS
        ELSE    F-  F+  FDUP  FABS
        THEN
        E{ N 1- } G@L  F2*  F< ;

CASE:  g*f  CP*F  F*  ;CASE
4 S->F  3 S->F  F/   FCONSTANT  F=4/3

: INTERPOLATE  ( 87: I[N] + I'[N-1] - I[N-1] - - )
        F=4/3   type 2/  g*f
        old.I  G@L   final.I G@L
        type 2 AND
        IF      CP+  CP+
        ELSE    F+   F+    THEN
        final.I G!L ;
\ BEHEADing ends here


: )INTEGRAL  ( 87: A B ERR - - I[A,B] )
        INITIALIZE
        BEGIN  N 0>
        WHILE     SUBDIVIDE  DEBUG
             CONVERGED?  Inc.N
             IF  INTERPOLATE  dec.N
             ELSE  type 2 AND IF  FDROP
             THEN        FDROP
             THEN
        REPEAT  final.I  G@L  ;
BEHEAD" N INTERPOLATE      \ optional
\ USE( F.name % A % B % E  type )INTEGRAL
```

The nonrecursive program obviously requires *much* more code than the recursive version. This is the chief disadvantage of a nonrecursive method[16].

16. The memory usage is about the same: the recursive method pushes limits, *etc.* onto the fstack.

§§5–4 Example of)INTEGRAL IN USE

The debugging code ("**DEBUG-ON**") lets us track the execution of the program by exhibiting the simulated stacks. Here is an example, $\int_1^2 dx\ \sqrt{x}$:

```
USE( FSQRT  % 1. % 2. % 1.E-3  REAL*4  )INTEGRAL E
```

```
N     X          F          I          E
0 1.0000E+00 1.0000E+00 5.5618E-01 5.0000E-04
1 1.5000E+00 1.2247E+00 6.5973E-01 5.0000E-04
2 2.0000E+00 1.4142E+00 1.4983E-01 1.2500E-04
old.I = 1.2071E+00   final.I = 0.0000E+00

0 1.0000E+00 1.0000E+00 5.5618E-01 5.0000E-04
1 1.5000E+00 1.2247E+00 3.1845E-01 2.5000E-04
2 1.7500E+00 1.3228E+00 3.4213E-01 2.5000E-04
3 2.0000E+00 1.4142E+00 1.7396E-01 1.2500E-04
old.I = 6.5973E-01   final.I = 0.0000E+00

0 1.0000E+00 1.0000E+00 5.5618E-01 5.0000E-04
1 1.5000E+00 1.2247E+00 3.1845E-01 2.5000E-04
2 1.7500E+00 1.3228E+00 1.6826E-01 1.2500E-04
3 1.8750E+00 1.3693E+00 1.7396E-01 1.2500E-04
4 2.0000E+00 1.4142E+00 0.0000E+00 0.0000E+00
old.I = 3.4213E-01   final.I = 0.0000E+00

0 1.0000E+00 1.0000E+00 5.5618E-01 5.0000E-04
1 1.5000E+00 1.2247E+00 1.5621E-01 1.2500E-04
2 1.6250E+00 1.2747E+00 1.6235E-01 1.2500E-04
3 1.7500E+00 1.3228E+00 1.7396E-01 1.2500E-04
old.I = 3.1845E-01   final.I = 3.4226E-01
```

```
N     X          F          I          E
0 1.0000E+00 1.0000E+00 2.6475E-01 2.5000E-04
1 1.2500E+00 1.1180E+00 2.9284E-01 2.5000E-04
2 1.5000E+00 1.2247E+00 1.6235E-01 1.2500E-04
old.I = 5.5618E-01   final.I = 6.6087E-01

0 1.0000E+00 1.0000E+00 2.6475E-01 2.5000E-04
1 1.2500E+00 1.1180E+00 1.4316E-01 1.2500E-04
2 1.3750E+00 1.1726E+00 1.4983E-01 1.2500E-04
3 1.5000E+00 1.2247E+00 1.7396E-01 1.2500E-04
old.I = 2.9284E-01   final.I = 6.6087E-01

0 1.0000E+00 1.0000E+00 1.2879E-01 1.2500E-04
1 1.1250E+00 1.0606E+00 1.3616E-01 1.2500E-04
2 1.2500E+00 1.1180E+00 1.4983E-01 1.2500E-04
old.I = 2.6475E-01   final.I = 9.5392E-01
1.2189E+00 ok
```

$$\int_1^2 dx\ \sqrt{x} = \frac{2}{3}\left(2^{3/2} - 1\right)$$

Notice that, although $\sqrt{x}$ is perfectly finite at $x = 0$, its first derivative is not. This is not a problem in the above case, because the lower limit is 1.0.

It is an instructive exercise to run the above example with the limits (0.0, 1.0). The adaptive routine spends many iterations approaching $x = 0$ (25 in the range [0., 0.0625] *vs.* 25 in the range [0.0625, 1.0]). This is a concrete example of how an adaptive routine will unerringly locate the (integrable) singularities of a function by spending lots of time near them. The best answer to this problem is to separate out the bad parts of a function by hand, if possible, and integrate them by some other algorithm that takes the singularities into account. By the same token, one should always integrate *up to*, but not *through*, a discontinuity in $f(x)$.

§§6 Adaptive integration in the Argand plane

We often want to evaluate the complex integral

$$I = \oint_{\Gamma} f(z)\,dz \qquad (16)$$

where Γ is a **contour** (simple, closed, piecewise-continuous curve) in the complex z-plane, and $f(z)$ is an **analytic**[17] function of z.

The easiest way to evaluate 16 is to parameterize z as a function of a real variable t ; as t runs from A to B, $z(t)$ traces out the contour. For example, the parameterization

$$z(t) = z_0 + R\cos(t) + iR\sin(t)\,,\quad 0 \le t \le 2\pi \qquad (17)$$

traces out a (closed) circle of radius R centered at $z = z_0$.

We assume that the derivative $\dot{z}(t) \equiv \frac{dz}{dt}$ can be defined; then the integral 16 can be re-written as one over a real interval, with a complex integrand:

$$I = \int_A^B \dot{z}(t) f\big(z(t)\big)\,dt \qquad (18)$$

Now our previously defined adaptive function **)INTEGRAL** can be applied directly, with **F.name** the name of a *complex* function

$$g(t) = \dot{z}(t) f\big(z(t)\big)\,, \qquad (19)$$

of the *real* variable t.

Here is an example of complex integration: we integrate the function $f(z) = e^{1/z}$ around the unit circle in the counter-clockwise (positive) direction.

---

7. "Analytic" means the ordinary derivative df(z)/dz exists. Consult any good text on the theory of functions of a complex variable.

The calculus of residues (Cauchy's theorem) gives

$$\oint_{|z|=1} dz\, e^{1/z} = 2\pi i \tag{20}$$

We parameterize the unit circle as $z(t) = \cos(2\pi t) + i \sin(2\pi t)$, hence $\dot{z}(t) = 2\pi i z(t)$, and we might as well evaluate

$$\int_0^1 dt\, z(t)\, e^{1/z(t)} \equiv 1. \tag{21}$$

For reasons of space, we exhibit only the first and last few iterations:

```
FIND FSINCOS  0= ?( FLOAD TRIG )
: Z(T)  F=PI  F*  F2*  FSINCOS ;          ( 87: t - - z )
: XEXP  FSINCOS  FROT  FEXP  X*F ;
   ( 87: x y - - e^x cos[y]  e^x sin[y] )
: G(T)  Z(T)  XDUP  1/X  XEXP  X* ;
DEBUG-ON
USE( G(T) % 0 % 1 % 1.E-2 COMPLEX )INTEGRAL  X.
```

```
N    X        F                I        E
0 0.0000 2.7182   0.0000   .58760   0.0000 .0049999
1 .50000 -.36787  -0.0000  .58760   0.0000 .0049999
2 1.0000 2.7182   0.0000   .00000   .00000 .0000000
old.I = 2.7182 0.0000   final.I = 0.0000 0.0000

N    X        F          I        E
0 0.0000 2.7182   0.0000   .58760   0.0000   .0049999
1 .50000 -.36787  -0.0000  .059198  -.067537 .0024999
2 .75000 .84147   -.54030  .44496   -.067537 .0024999
3 1.0000 2.7182   0.0000   .00000   .00000   .0000000
old.I = .58760 0.0000   final.I = 0.0000 0.0000

N    X        F          I        E
0 0.0000 2.7182 0.0000 .58760 0.0000  .0049999
1 .50000 -.36787 -0.0000 .059198 -.067537 .0024999
2 .75000 .84147 -.54030 .17896 -.043682 .0012499
3 .87500 2.0219 -.15862 .29626 -.009914 .0012499
4 1.0000 2.7182 0.0000 .00000 .000000 .0000000
old.I = .44496 -.067537   final.I = 0.0000 0.0000

N    X        F          I        E
0 0.0000 2.7182 0.0000 .58760 0.0000  .0049999
1 .50000 -.36787 -0.0000 .059198 -.067537 .0024999
2 .75000 .84147 -.54030 .17896 -.043682 .0012499
3 .87500 2.0219 -.15862 .14190 -.005745 .00062499
4 .93750 2.5189 -.025229 .16366 -.000788 .00062499
5 1.0000 2.7182 0.0000 .000000 .000000 .00000000
old.I = .29626 -.0099138   final.I = 0.0000 0.0000

N    X        F          I        E
0 0.0000 2.7182 0.0000 .58760 0.0000        .0049999
1 .50000 -.36787 -0.0000 .059198 -.067537   .0024999
2 .75000 .84147 -.54030 .17896 -.043682     .0012499
3 .87500 2.0219 -.15862 .14190 -.005745     .00062499
4 .93750 2.5189 -.025229 .081022 -.0004467  .00031249
5 .96875 2.6665 -.0033577 .08414 -.0000525  .00031249
6 1.0000 2.7182 0.0000 .000000 .0000000     .00000000
old.I = .16366 -.00078842   final.I = 0.0000 0.0000
N    X        F          I        E
0 0.0000 2.7182 0.0000 .58760 0.0000  .0049999
1 .50000 -.36787 -0.0000 .059198 -.067537 .0024999
2 .75000 .84147 -.54030 .17896 -.043682 .0012499
3 .87500 2.0219 -.15862 .14190 -.0057453 .00062499
4 .93750 2.5189 -.025229 .08102 -.0004467 .00031249
5 .96875 2.6665 -.0033577 .04197 -.0000296 .00015624
6 .98437 2.7052 -.000426 .042371 -.0000033 .00015624
7 1.0000 2.7182 0.0000 0.0000 0.0000  0.0000
old.I = .084137 -.000052465   final.I = 0.0000 0.0000

N    X        F          I        E
0 0.0000 2.7182 0.0000 .58760 0.0000  .0049999
1 .50000 -.36787 -0.0000 .059198 -.067537 .0024999
2 .75000 .84147 -.54030 .17896 -.043682 .0012499
3 .87500 2.0219 -.15862 .14190 -.005745 .00062499
4 .93750 2.5189 -.025229 .040020 -.0002833 .00015624
5 .95312 2.6036 -.011036 .041173 -.0001125 .00015624
6 .96875 2.6665 -.003358 .042371 -.0000033 .00015624
old.I = .061022 -.00044657   final.I = .084404 -.0000265
```

```
N    X        F        I        E
0 0.0000 2.7182 0.0000 .29626 .0099136 .0012499
1 .12500 2.0219 .15862 .10753 .016479 .00062499
2 .18750 1.4190 .36873 .039711 .013048 .00031249
3 .21875 1.1224 .46620 .030886 .015726 .00031249
4 .25000 .84147 .54030 .000053670 .0079676 .00015624

old.I = .070842 .028407   final.I = .51343 -.050786N

N    X        F        I        E
0 0.0000 2.7182 0.0000 .29626 .0099136 .0012499
1 .12500 2.0219 .15862 .058518 .0065556 .00031249
2 .15625 1.7232 .26094 .049099 .0098386 .00031249
3 .18750 1.4190 .36873 .030886 .015726 .00031249
old.I = .10753 .016479   final.I = .58374 -.021892

N    X        F        I        E
0 0.0000 2.7182 0.0000 .16366 .00078842 .00062499
1 .062500 2.5189 .025229 .14190 .0057453 .00062499
2 .12500 2.0219 .15862 .049099 .0098386 .00031249
old.I = .29626 .0099136   final.I = .69139 -.0055268

N    X        F        I        E
0 0.0000 2.7182 0.0000 .16366 .00078842 .00062499
1 .062500 2.5189 .025229 .075223 .0015953 .00031249
2 .093749 2.2954 .076875 .067457 .0036796 .00031249
3 .12500 2.0219 .15862 .030886 .015726 .00031249
old.I = .14190 .0057453   final.I = .69139 -.0055268

N    X        F        I        E
0 0.0000  2.7182 0.0000  .16366 .00078842 .00062499
1 .062500 2.5189  .025229 .075223 .0015953 .00031249
2 .093749 2.2954  .076875 .034833 .0014942 .00015624
3 .10937  2.1632  .11439  .032696 .0021329 .00015624
4 .12500  2.0219  .15862  .0000537 .0079676 .00015624
old.I = .067457 .0036796   final.I = .69139 -.0055268

N    X        F        I        E
0 0.0000 2.7182 0.0000 .16366 .00078842 .00062499
1 .062500 2.5189 .025229 .038546 .00068464 .00015624
2 .078125 2.4190 .047044 .036800 .00098812 .00015624
3 .093749 2.2954 .076875 .032696 .0021329 .00015624
old.I = .075223 .0015953   final.I = .75894 -.0019171

N    X        F        I        E
0 0.0000 2.7182 0.0000 .084137 .000052465 .00031249
1 .031250 2.6665 .0033577 .081022 .00044667 .00031249
2 .062500 2.5189 .025229 .036800 .00098812 .00015624
old.I = .16366 .00078842   final.I = .83433 -.00040523

N    X        F        I        E
0 0.0000 2.7182 0.0000 .084137 .000052465 .00031249
1 .031250 2.6665 .0033577 .041173 .00011247 .00015624
2 .046875 2.6036 .011038 .040020 .00028334 .00015624
3 .062500 2.5189 .025229 .032696 .0021329 .00015624
old.I = .081022 .00044667   final.I = .83433 -.00040523

N    X        F        I        E
0 0.0000 2.7182 0.0000 .042371 .000003331 .00015624
1 .015625 2.7052 .00042642 .041966 .0000296 .000 5624
2 .031250 2.6665 .0033577 .040020 .00028334 .00015624
old.I = .084137 .000052465   final.I = .91558 -.000026373
.99999 .00000001 ok
```

Note:
answer = 1

§2 Fitting functions to data

One of the most important applications of numerical analysis is the representation of numerical data in functional form. This includes fitting, smoothing, filtering, interpolating, *etc.*

A typical example is the problem of table lookup: a program requires values of some mathematical function $-\sin(x)$, say – for arbitrary values of x. The function is moderately or extremely time-consuming to compute directly. According to the Intel tim-

ings for the 80x87 chip, this operation should take about 8 times longer than a floating point multiply. In some real-time applications this may be too slow.

There are several ways to speed up the computation of a function. They are all based on compact representations of the function — either in tabular form or as coefficients of functions that are faster to evaluate. For example, we might represent sin(x) by a simple polynomial[18]

$$\sin(x) \approx x\left(0.994108 - 0.147202x\right), \qquad (22)$$

accurate to better than 1% over the range $-\frac{\pi}{2} \le x \le \frac{\pi}{2}$, this requires but 3 multiplications and an addition to evaluate. This would be twice as fast as calculating sin(x) on the 80x87 chip[19].

To achieve substantially greater speed requires table look-up. To locate data in an ordered table, we might employ binary search: that is, look at the x-value halfway down the table and see if the desired value is greater or less than that. On the average, $\log_2(N)$ comparisons are required, where N is the length of the table. For a table with 1% precision, we might need 128 entries, *i.e.* seven comparisons.

Binary search is unacceptably slow — is there a faster method? In fact, assuming an ordered table of equally-spaced abscissae the fastest way to locate the desired x-value is **hashing**, a method for *computing* the address rather than finding it using comparisons. Suppose, as before, we need 1% accuracy, *i.e.* a 128-point table with x in the range $[0,\pi/2]$. To look up a value, we multiply x by $256/\pi \cong 81.5$, truncate to an integer and quadruple it to get a (4-byte) floating point address. These operations — including fetch to the 87stack — take about 1.5-2 fp multiply times, hence the speedup is 4-fold.

The speedup factor does not seem like much, especially for a function such as $\sin(x)$ that is built into the fast co-processor. However, if we were speaking of a function that is considerably slower to evaluate (for example one requiring evaluation of an integral or solution of a differential equation) hashed table lookup with interpolation can be several orders of magnitude faster than direct evaluation.

We now consider how to represent data by mathematical functions. This can be useful in several contexts:

- The theoretical form of the function, but with unknown parameters, may be known. One might like to **determine** the parameters from the data. For example, one might have a lot of data on pendulums: their periods, masses, dimensions, *etc.* The period of a pendulum is given, theoretically, by

$$\tau = \left(\frac{2\pi L}{g}\right)^{1/2} f\left(\frac{L}{r}, \frac{m_{bob}}{m_{string}}, \ldots\right) \qquad (23)$$

  where L is the length of the string, g the acceleration of gravity, and f is some function of ratios of typical lengths, masses and other factors in the problem. In order to determine g accurately, one generally fits a function of all the measured factors, and tries to minimize its deviation from the measured periods. That is, one might try

$$\tau_n = \left(\frac{2\pi L_n}{g}\right)^{1/2} \left[1 + \alpha\frac{r_n}{L_n} + \beta\left(\frac{m_{bob}}{m_{string}}\right)_n + \ldots\right] \qquad (24)$$

  for the n'th set of observations, with g, α, β, ... the unknown parameters to be determined.

- Sometimes one knows that a phenomenon is basically smoothly varying; so that the wiggles and deviations in observations are noise or otherwise uninteresting. How can we filter out the noise without losing the significant part of the data? Several methods have been developed for this purpose, based on the same principle: the data are represented as a sum of functions from a **complete** set of functions, with unknown coefficients. That is, if $\varphi_m(x)$ are the functions, we say (y_n are the data)

$$y_n = \sum_{m=0}^{\infty} c_m \varphi_m(x_n) \qquad (25)$$

Such representations are theoretically possible under general conditions. Then to filter we keep only a finite sum, retaining the first N (usually simplest and smoothest) functions from the set. An example of a complete set is **monomials**, $\varphi_m(x) = x^m$. Another is **sinusoidal (trigonometric) functions**,

$$\sin(2\pi mx),\ \cos(2\pi mx),\ 0 \le x \le 1,$$

used in Fourier-series representation. **Gram polynomials**, discussed below, comprise a third useful complete set.

The representation in Eq. 25 is called **linear** because the unknown coefficients c_m appear to their first power. Thus, if all the data were to double, we see immediately that the c_m's would have to be multiplied by the same factor, 2. Sometimes, as in the example of the measurement of g above, the unknown parameters appear in more complicated fashion. The problem of fitting with these more general functional forms is called **nonlinear** for obvious reasons. The **simplex algorithm** of Ch. 8 §2.3 below is an example of a nonlinear fitting procedure.

We are now going to write programs to fit both linear and nonlinear functions to data. The first and conceptually simplest of these is the **Fourier transform**, namely representing a function as a sum of sines and cosines.

§§1 Fast Fourier transform

What is a Fourier transform? Suppose we have a function that is **periodic** on the interval $0 \le x \le 2\pi$:

$$f(x + 2\pi) = f(x)\ ;$$

Then under fairly general conditions the function can be expressed in the form

$$f(x) = a_0 + \sum_{n=1}^{\infty} \Big(a_n \cos(nx) + b_n \sin(nx) \Big) \qquad (26)$$

Another way to write Eq. 26 is

$$f(x) = \sum_{-\infty}^{+\infty} c_n e^{inx}. \tag{27}$$

In either way of writing, the c_n are called **Fourier coefficients** of the function $f(x)$. Looking, *e.g.*, at Eq. 27, we see that the **orthogonality** of the sinusoidal functions leads to the expression

$$c_n = \frac{1}{2\pi} \int_0^{2\pi} f(x)\, e^{-inx}\, dx\,. \tag{28}$$

Evaluating Eq. 28 numerically requires – for given n – at least 2n points[20]. Naively, for each n = 0 to N-1 we have to do a sum

$$c_n \approx \sum_{k=1}^{2N} f_k\, e^{-2\pi ink/N}$$

which means carrying out $2N^2$ complex multiplications.

The **fast Fourier transform** (**FFT**) was discovered by Runge and König, rediscovered by Danielson and Lanczos and re-rediscovered by Cooley and Tukey[21]. The FFT algorithm can be expressed as three steps:

- Discretize the interval, *i.e.* evaluate $f(x)$ only for

  $$x_k = 2\pi \frac{k}{N}\,,\ 0 \le x \le N-1.$$

  Call $f(x_k) \equiv f_k$.

- Express the Fourier coefficients as

  $$c_n = \sum_{k=0}^{N-1} f_k\, e^{-2\pi ink/N}\,. \tag{29}$$

- With $w_n = e^{-2\pi in/N}$, Eq. 29 is an N-1'st degree polynomial in w_n. We evaluate the polynomial using a fast algorithm.

---

20. to prevent **aliasing**.
21. See, *e.g.*, D.E. Knuth, *The Art of Computer Programming* v. 2 (Addison-Wesley Publishing Co., Reading, MA, 1981) p. 642.

To evaluate rapidly the polynomial

$$c_n = P_N(w_n) \equiv \sum_{k=0}^{N-1} f_k (w_n)^k$$

we divide it into two polynomials of order N/2, dividing each of those in two, *etc*. This procedure is efficient only for $N = 2^\nu$, with ν an integer, so this is the case we attack.

How does dividing a polynomial in two help us? If we segregate the odd from the even powers, we have, symbolically,

$$P_N(w) = E_{N/2}(w^2) + w\, O_{N/2}(w^2)\,. \qquad (30)$$

Suppose the time to evaluate $P_N(w)$ is T_N. Then, clearly,

$$T_N = \lambda + 2T_{N/2} \qquad (31)$$

where λ is the time to segregate the coefficients into odd and even, plus the time for 2 multiplications and a division. The solution of Eq. 31is λ(N-1). That is, it takes O(N) time to evaluate a polynomial.

However, the discreteness of the Fourier transform helps us here. The reason is this: to evaluate the transform, we have to evaluate $P_N(w_n)$ for N values of w_n. But w_n^2 takes on only N/2 values as n takes on N values. Thus to evaluate the Fourier transform for all N values of n, we can evaluate the two polynomials of order N/2 for half as many points.

Suppose we evaluated the polynomials the old-fashioned way: it would take 2(N/2)≡N multiplications to do both, but we need do this only N/2 times, and N more (to combine them) so we have $N^2/2 + N$ rather than N^2. We have gained a factor 2. Obviously it pays to repeat the procedure, dividing each of the sub-polynomials in two again, until only monomials are left.

Symbolically, the number of multiplications needed to evaluate a polynomial for N (discrete) values of w is

$$\tau_N = N\lambda + 2\tau_{N/2} \qquad (32)$$

whose solution is

$$\tau_N = \lambda N \log_2(N) . \qquad (33)$$

Although the FFT algorithm can be programmed recursively, it almost never is. To see why, imagine how the coefficients would be re-shuffled by Eq. 30: we work out the case for 16 coefficients, exhibiting them in Table 8-1 below, writing only the indices:

Table 8-1 Bit-reversal for re-ordering discrete data

Start	Step 1	Step 2	Step 3	Bin_0	Bin_3
0	0	0	0	0000	0000
1	2	4	8	0001	1000
2	4	8	4	0010	0100
3	6	12	12	0011	1100
4	8	2	2	0100	0010
5	10	6	10	0101	1010
6	12	10	6	0110	0110
7	14	14	14	0111	1110
8	1	1	1	1000	0001
9	3	5	9	1001	1001
10	5	9	5	1010	0101
11	7	13	13	1011	1101
12	9	3	3	1100	0011
13	11	7	11	1101	1011
14	13	11	7	1110	0111
15	15	15	15	1111	1111

The crucial columns are "Start" and "Step 3". Unfortunately, they are written in decimal notation, which conceals a fact that becomes glaringly obvious in binary notation. So we re-write them in binary in the columns Bin_0 and Bin_3 —and see that the final order can be obtained from the initial order simply by reversing the order of the bits, from left to right!

A standard FORTRAN program for complex FFT is shown below. We shall simply translate the FORTRAN into FORTH as expeditiously as possible, using some of FORTH's simplifications.

One such improvement is a word to reverse the bits in a given integer. Note how clumsily this was done in the FORTRAN

```
      SUBROUTINE FOUR1(DATA, NN, ISIGN)
C
C     from Press, et al., Numerical Recipes, Ibid., p. 394.
C
C     ISIGN DETERMINES WHETHER THE FFT
C     IS FORWARD OR BACKWARD
C
C     DATA IS THE (COMPLEX) ARRAY OF DISCRETE INPUT
      COMPLEX W, WP, TEMP, DATA(N)
      REAL*8 THETA
      J=0
      DO 11 I=0,N-1              \ begin bit.reversal
       IF (J.GT.I) THEN
         TEMP=DATA(J)
         DATA(J)=DATA(I)
         DATA(I)=TEMP
       ENDIF
       M=N/2
1      IF ((M.GE.1).AND.(J.GT.M)) THEN
         J=J-M
         M=M/2
         GO TO 1
       ENDIF
       J=J+M
11    CONTINUE             \ end bit.reversal

         MMAX=1             \ begin Danielson-Lanczos section
2        IF (N.GT.MMAX) THEN
           ISTEP=2*MMAX     \ executed lg(N) times
                            \ init trig recurrence

         THETA=3.141592653589793D0/(ISIGN*MMAX)
         WP=CEXP(THETA)
         W=DCMPLX(1.D0,0.D0)
         DO 13 M=1,MMAX,2           \ outer loop
           DO 12 I=M,N,ISTEP        \ inner loop
             J=I+MMAX               \ total=N times
             TEMP=DATA(J)*W
             DATA(J)=DATA(I)-TEMP
             DATA(I)=DATA(I)+TEMP
12         CONTINUE                 \ end inner loop
C
           W=W*WP           \ trig recurrence
C
13       CONTINUE           \ end outer loop
         MMAX=ISTEP
         GO TO 2
       ENDIF                \ end Danielson-Lanczos section
       RETURN
```

program. Since practically every microprocessor permits right-shifting a register one bit at a time and feeding the overflow into another register from the right, **B.R** can be programmed easily in machine code for speed. Our fast bit-reversal procedure **B.R** may be represented pictorially as in Fig. 8-5 below.

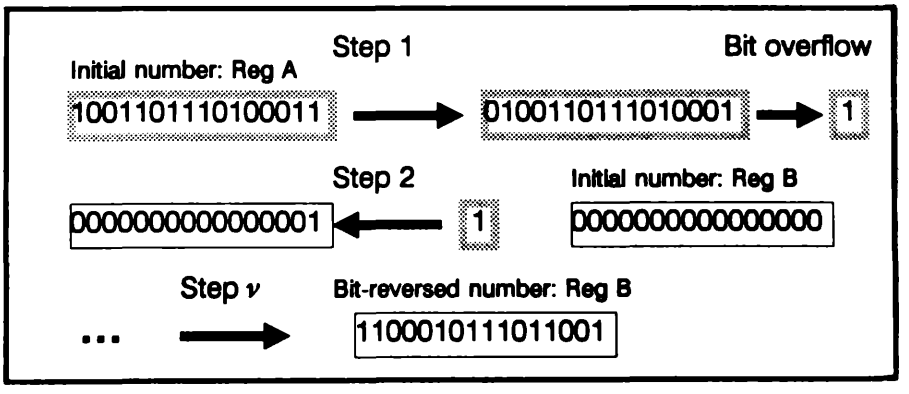


Fig. 8-5 *Pictorial representation of bit-reversal*

Bit-reversal can be accomplished in high-level FORTH *via*

```
: B.R           ( n - - n' )      \ reverse order of bits
  0 SWAP        ( - - 0 n )       \ set up stack
  N.BITS 0 DO
       DUP  1 AND                 \ pick out 1's bit
       ROT  2* +                  \ left shift 1, add 1's bit
       SWAP 2/                    \ right-shift n
  LOOP DROP ;
```

Note: **N.BITS** is a **VAR**, previously set to $\nu = \log_2(N)$

We will use **B.R** to re-order the actual data array (even though this is slightly more time-consuming than setting up a list of scrambled pointers, leaving the data alone). We forego indirection for two reasons: first, we have to divide by N (N steps) when inverse-transforming, so we might as well combine this with bit-reversal; second, there are N steps in rearranging and dividing by N the input vector, whereas the FFT itself takes $N\log_2(N)$ steps, *i.e.* the execution time for the preliminary N steps is unimportant.

Now, how do we go about evaluating the sub-polynomials to get the answer? First, let us write the polynomials (for our case N = 16) corresponding to taking the (bit-reversed) addresses off the stack in succession, as in Fig. 8-6 below.

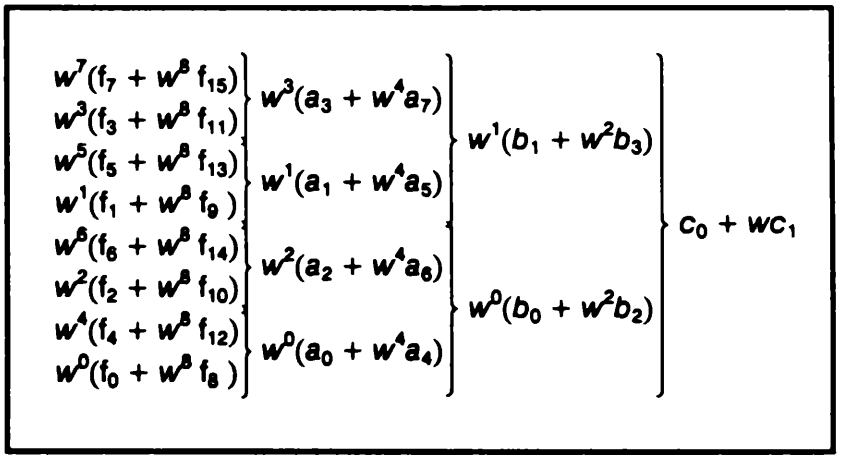


Fig. 8-6 *The order of evaluating a 16 pt. FFT*

We see that w_n^8 (for N = 16) has only two possible values, ±1. Thus we must evaluate not 16×8 terms like $f_i + w^8 f_{i+8}$, but only 2×8. Similarly, we do not need to evaluate 16×4 terms of form $f_i + w^4 f_{i+4}$, but only 4×4, since there are only 4 possible values of w_n^4. Thus the total number of multiplications is

$$2\times8 + 4\times4 + 8\times2 + 16\times1 = 64 \equiv 16 \log_2 16,$$

as advertised. This is far fewer than 16×16 = 256, and the ratio improves with N – for example a 1024 point FFT is 100 times faster than a slow FT.

We list the FFT program on page 191 below. Since **}FFT** transforms a one-dimensional array we retain the curly braces notation introduced in Ch. 5. We want to say something like

```
V{ n.pts  FORWARD  }FFT
```

where **V{** is the name of the (complex) array to be transformed, **n.pts** (a power of 2) is the size of the array, and the flag-setting words **FORWARD** or **INVERSE** determine whether we are taking a FFT or inverting one.

Now we test the program. Table 8-2 on page 192 contains the weekly stock prices of IBM stock, for the year 1983 (the 52 values have been made complex numbers by adding 0*i*, and the table padded out to 64 entries (the nearest power of 2) with complex zeros)[22]. The first two entries (2, 64) are the type and length of the file. (The file reads from left to right.)

We FFT Table 8-2 using the phrase **IBM{ 64 DIRECT }FFT**. The power spectrum of the resulting FFT (Table 8-3) is shown in Fig. 8-7 on page 192 below.

22. This example is taken from the article "FORTH and the Fast Fourier Transform" by Joe Barnhart, *Dr. Dobb's Journal*, September 1984, p. 34.

```
\ Complex Fast Fourier Transform
\ Usage: Vector.name{ N FORWARD ( INVERSE ) }FFT
TASK FFT
\ check for presence of these extensions and load
FIND C+          0= ?( FLOAD COMPLEX )
FIND 1ARRAY      0= ?( FLOAD MATRIX.HSF )
FIND FILL        0= ?( FLOAD FILBIO.FTH )
FIND TRIG        0= ?( FLOAD TRIG )
\ if not there
DECIMAL
\ ---------------------------------
\ auxiliary words
  CODE SHR BX 1 SHR END-CODE
  : LG2 ( n -- lg2[n] ) 0 SWAP  ( -- 0 n) SHR
     BEGIN ?DUP 0>
     WHILE  SHR SWAP 1+ SWAP  REPEAT ;

0 VAR DIRECTION?
FORWARD   0 IS DIRECTION? ;
INVERSE   -1 IS DIRECTION? ;

     0 VAR N.BITS                  \ some VARs
     0 VAR N
     0 VAR MMAX
     0 VAR f{

: C/N  N S->F C/F ;
: NORMALIZE  DIRECTION?
       IF C/N CPSWAP C/N CPSWAP THEN ;
\ end auxiliary words
\ ---------------------------------
\ lazy bit-reversal routines!
0 VAR I.R
: B.R (n -- n')               \ reverses order of bits
   0 SWAP  ( -- 0 n )         \ set up stack
   N.BITS 0 DO  DUP 1 AND  \ pick out 1's bit
        ROT 2* +          \ double sum and add 1's bit
        SWAP 2/           \ n - n/2
   LOOP DROP ;

: BIT.REVERSE 0 IS I.R
     N 0 DO  I B.R  IS I.R
          I.R I < NOT        ( I.R > = I ? )
          IF f{ I.R } G@L f{ I } G@L  NORMALIZE
             f{ I.R } G!L f{ I } G!L THEN
     LOOP ;
\ end bit-reversal (N times)
```

```
\ ---------------------------------
                                   \ main algorithm
CODE C-+  2 FLD.  1 FXCH.  3 FSUBP.
     1 FADDP.  1 FXCH.  3 FLD.
     1 FXCH.  4 FSUBP.  1 FADDP.  1 FXCH.  END-CODE
  ( 87: w z -- w-z  w+z )

: THETA  F= PI MMAX S->F F/  DIRECTION? FSIGN ;

CREATE WP 16 ALLOT OK I.W
: INIT.TRIG  FINIT THETA EXP(i*PI*t) WP DCP! C= 1 ;

: NEW.W  ( 87: w -- w' ) WP DCP@  C* ;

0 VAR ISTEP

: DO.INNER.LOOP
     DO  MMAX I + IS I.R
         CPDUP  f{ I.R } G@L  C*  f{ I } G@L
         CPSWAP C-+  f{ I } G!L  f{ I.R }  G!L
     ISTEP +LOOP ;

: }FFT  ( adr n -- )  IS N  IS f{
     1 IS MMAX
     N LG2 IS N.BITS
     FINIT
     BIT.REVERSE
     BEGIN
       N MMAX >
     WHILE
       INIT.TRIG  MMAX 2* IS ISTEP
       MMAX 0 DO
         N I DO.INNER.LOOP  NEW.W
       LOOP
       ISTEP IS MMAX
     REPEAT CPDROP ;

: POWER 0 DO f{ I } G@L CABS CR I . F. LOOP ;
\ power spectrum of FFT                \ end of fft code
\ ---------------------------------
                                   \ an example
64 LONG COMPLEX 1ARRAY A{
: INIT.A  A{ $" IBM.EX" OPEN-INPUT FILL  CLOSE-INPUT ;
INIT.A
A{ 64 DIRECT }FFT
                                   \ end of example
\ ---------------------------------
```

How do we know the FFT program actually worked? The simplest method is to inverse-transform the transform, and compare with the input file. The FFT and inverse FFT are given, respectively, in Tables 8-3 and 8-4 on page 193 below. Within roundoff error, Table 8-4 agrees with Table 8-2 on page 192.

Table 8-2 *Weekly IBM common stock prices, 1983*

```
2 64
 98.63 0.0   99.13 0.0   94.63 0.0   97.38 0.0
 97.38 0.0   96.38 0.0   98.63 0.0  100.38 0.0
102.25 0.0  100.75 0.0   99.88 0.0  102.13 0.0
101.63 0.0  103.88 0.0  110.13 0.0  117.25 0.0
117.00 0.0  117.63 0.0  116.50 0.0  110.63 0.0
113.00 0.0  114.00 0.0  114.25 0.0  121.13 0.0
123.00 0.0  121.00 0.0  121.50 0.0  120.13 0.0
124.38 0.0  120.38 0.0  119.75 0.0  118.50 0.0
122.50 0.0  117.83 0.0  119.75 0.0  122.25 0.0
123.13 0.0  126.63 0.0  126.88 0.0  132.25 0.0
131.75 0.0  127.00 0.0  126.00 0.0  122.25 0.0
126.88 0.0  123.50 0.0  121.00 0.0  117.88 0.0
122.25 0.0  120.88 0.0  123.63 0.0  122.00 0.0
  0.0  0.0    0.0  0.0    0.0  0.0    0.0  0.0
  0.0  0.0    0.0  0.0    0.0  0.0    0.0  0.0
  0.0  0.0    0.0  0.0    0.0  0.0    0.0  0.0
```

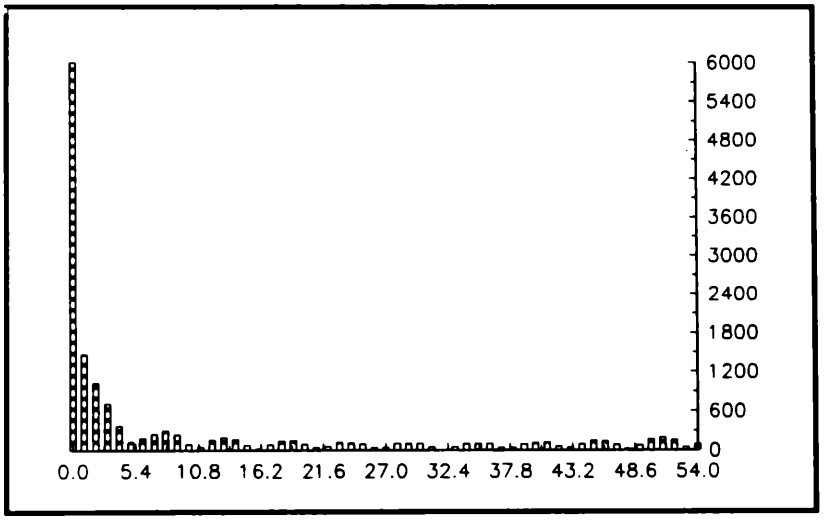


Fig. 8-7 *Power spectrum of FFT of 1983 IBM prices (from Table 8-3)*

Table 8-3 *FFT of IBM weekly stock prices, 1983*

0	5889.4689	0.0000000	32	3.1601882	0.0000000	16	7.2500000	-15.170043	48	7.2500000	15.170043
1	-1356.9239	562.94860	33	13.450067	44.866729	17	-27.548482	81.061904	49	69.775169	29.838697
2	-374.28417	956.64282	34	69.175987	32.346035	18	68.084251	117.80709	50	146.10401	-64.187637
3	305.66314	634.45819	35	99.756884	-25.042837	19	119.02930	71.422492	51	84.502433	-165.21282
4	327.40542	177.60118	36	39.756885	-83.410886	20	87.868354	-24.258874	52	-49.510869	-147.45129
5	125.21042	-8.0053304	37	-30.828173	-14.841442	21	1.8497861	-37.864341	53	-49.475231	5.4212093
6	-178.28804	39.630813	38	19.293210	29.322504	22	-11.327768	52.313049	54	80.279541	33.341926
7	-90.525482	226.27159	39	66.786362	22.908525	23	44.172229	110.77391	55	191.18481	-128.15400
8	150.24284	280.91506	40	107.71731	-34.415077	24	107.71731	34.415077	56	150.24284	-280.91506
9	191.18481	128.15400	41	44.172229	-110.77391	25	66.786362	-22.908525	57	-90.525482	-226.27159
10	80.279541	-33.341926	42	-11.327768	-52.313049	26	19.293210	-29.322504	58	-178.28804	-39.630813
11	-49.475231	-5.4212093	43	1.8497861	37.864341	27	-30.828173	14.841442	59	125.21042	8.0053304
12	-49.510869	147.45129	44	87.868354	24.258874	28	39.756885	83.410886	60	327.40542	-177.60118
13	84.502433	165.21282	45	119.02930	-71.422492	29	99.756584	25.042837	61	305.66314	-634.45819
14	146.10401	64.187637	46	68.084251	-117.80709	30	69.175987	-32.346035	62	-374.28417	-956.64282
15	69.775169	-29.838697	47	-27.548482	-81.061904	31	13.450067	-44.866729	63	-1356.9239	-562.94860

Table 8-4 *Reconstructed IBM prices (inverse FFT)*

0	96.630	0.0000	32	122.50	0.0000	16	117.00	-.000000000	48	122.25	.000000000
1	99.129	.000000705	33	117.82	-.000000245	17	117.62	-.000000518	49	120.87	.000000132
2	94.629	.000000023	34	119.75	.000000148	18	116.50	-.000000119	50	123.63	-.000000052
3	97.379	.000000227	35	122.24	-.000000379	19	110.62	.000001375	51	121.99	-.000000944
4	97.380	.000000385	36	123.13	-.000000385	20	113.00	.000001076	52	.000012864	-.000001076
5	96.379	-.000000362	37	126.62	.000000796	21	114.00	.000000975	53	-.000001277	-.000001434
6	98.630	.000001697	38	126.88	-.000000851	22	114.25	.000000378	54	-.000002880	-.000001224
7	100.38	.000002158	39	132.25	-.000001661	23	121.12	.000000317	55	-.000005683	-.000000315
8	102.25	-.000000000	40	131.75	-.000000000	24	123.00	.000000000	56	-.000001602	-.000000000
9	100.75	.000000799	41	127.00	-.000000373	25	121.00	.000001130	57	-.000002995	-.000001630
10	99.880	-.000000271	42	128.00	.000000401	26	121.50	-.000001469	58	-.000010348	.000001339
11	102.12	-.000000078	43	122.24	-.000001011	27	120.12	.000002522	59	-.000005928	-.000001711
12	101.62	.000000316	44	126.87	-.000000316	28	124.37	.000000027	60	-.000003016	-.000000027
13	103.88	-.000000043	45	123.50	.000000390	29	120.38	.000001261	61	-.000002253	-.000001583
14	110.13	-.000001615	46	121.00	.000001937	30	119.75	-.000000618	62	-.000003121	.000000295
15	117.25	-.000002237	47	117.87	.000001164	31	118.49	.000000573	63	-.000003713	.000000500

§2 Gram polynomials

Gram polynomials are useful in fitting data by the linear least-squares method. The usual method is based on the following question: What is the "best" polynomial,

$$P_N(x) = \sum_{n=0}^{N} \gamma_n x^n, \tag{34}$$

(of order N) that I can use to fit some set of M pairs of data points,

$$\begin{Bmatrix} x_k \\ f_k \end{Bmatrix}, \quad k=0, 1, \ldots, M-1$$

(with M > N) where f(x) is measured at M distinct values of the independent variable x ?

The usual answer, found by Gauss, is to minimize the **squares** of the **deviations** (at the points x_k) of the fitting function $P_N(x)$ from the data – possibly weighted by the uncertainties of the data. That is, we want to minimize the **statistic**

$$\chi^2 = \sum_{k=0}^{M-1} \left(f_k - \sum_{n=0}^{N} \gamma_n x_k^n \right)^2 \frac{1}{\sigma_k^2} \tag{35}$$

with repect to the N + 1 parameters γ_n .

From the differential calculus we know that a function's first derivative vanishes at a minimum, hence we differentiate χ^2 with respect to each γ_n independently, and set the results equal to zero. This yields N + 1 linear equations in N + 1 unknowns:

$$\sum_m A_{nm} \gamma_m = \beta_n, \quad n=0, 1, \ldots, N \tag{36}$$

where (the symbol $\hat{=}$ means "is defined by")

$$A_{nm} \hat{=} \sum_{k=0}^{M-1} (x_k)^{n+m} \frac{1}{\sigma_k^2} \tag{37a}$$

and

$$\beta_n \hat{=} \sum_{k=0}^{M-1} x_k^n f_k \frac{1}{\sigma_k^2} \tag{37b}$$

In Chapter 9 we develop methods for solving linear equations. Unfortunately, they cannot be applied to Eq. 36 for $N \gtrsim 9$ because the matrix A_{nm} approximates a Hilbert matrix,

$$H_{nm} = \frac{const.}{n+m+1},$$

a particularly virulent example of an exponentially **ill-conditioned matrix**. That is, the roundoff error in solving 36 grows exponentially with N, and is generally unacceptable. We can avoid roundoff problems by expanding in polynomials rather than monomials:

$$\chi^2 = \sum_{k=0}^{M-1} \left(f_k - \sum_{n=0}^{N} \gamma_n p_n(x_k) \right)^2 \frac{1}{\sigma_k^2} . \tag{38}$$

The matrix then becomes

$$A_{nm} = \sum_{k=0}^{M-1} p_n(x_k) p_m(x_k) \frac{1}{\sigma_k^2} \tag{39a}$$

and the inhomogeneous term is now

$$\beta_n = \sum_{k=0}^{M-1} p_n(x_k) f_k \frac{1}{\sigma_k^2} \tag{39b}$$

Is there any choice of the polynomials $p_n(x)$ that will eliminate roundoff? The best kinds of linear equations are those with nearly diagonal matrices. We note the sum in Eq. 39a is nearly an integral, if M is large. If we choose the polynomials so they are **orthogonal** with respect to the weight function

$$w(x) = \frac{1}{\sigma_k^2} \theta(x_k - x)\, \theta(x - x_{k-1}) ,$$

where

$$\theta(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases}$$

then A_{nm} will be nearly diagonal, and well-conditioned.

Orthogonal polynomials play an important role in numerical ana ysis and applied mathematics. They satisfy **orthogonality rel tions**[23] of the form

$$\int_A^B dx\, w(x) p_n(x)\, p_m(x) = \delta_{nm} \equiv \begin{cases} 1, & m=n \\ 0, & m \neq n \end{cases} \qquad (40)$$

where the **weight function** $w(x)$ is positive.

For a given $w(x)$ and interval [A,B], we can construct orthogon polynomials using the **Gram-Schmidt orthogonalization** proces

Denote the integral in Eq. 40 by $(p_n\,, p_m\,)$ to save having to writ it many times. We start with

$$p_{-1} = 0\,,$$
$$p_0(x) = \left(\int_A^B dx\, w(x) \right)^{-1/2} = \text{const.}\,,$$

and assume the polynomials satisfy the 2-term upward recursio relation

$$p_{n+1}(x) = \left(a_n + x b_n \right) p_n(x) \; + \; c_n\, p_{n-1}(x) \qquad (41)$$

Now apply Eq. 41: assume we have calculated p_n and p_{n-1} an want to calculate p_{n+1}. Clearly, the orthogonality property give

$$(p_{n+1}\,,\, p_n\,) = (p_{n+1}\,,\, p_{n-1}\,) = (p_n\,,\, p_{n-1}\,) = 0\,,$$

and the assumed normalization gives

$$(p_n\,,\, p_n\,) = 1\,.$$

These relations yields two equations for the three unknowns, a b_n and c_n:

---

23. Polynomials can be thought of as vectors in a space of infinitely many dimensions ("Hilbert" space). Certain polynomials are like the vectors that point in the (mutually orthogonal) direc- tions in ordinary 3-dimensional space, and so are called **orthogonal** by analogy.

$$a_n + b_n (p_n , x p_n) = 0$$

$$c_n + b_n (p_n , x p_{n-1}) = 0$$

We express a_n and c_n in terms of b_n to get

$$p_{n+1}(x) = b_n \Big[\Big(x - (p_n , x p_n) \Big) p_n(x) - (p_n , x p_{n-1}) p_{n-1}(x) \Big] \tag{42}$$

We determine the remaining parameter b_n by again using the normalization condition:

$$(p_{n+1} , p_{n+1}) = 1.$$

In practice, we pretend $b_n = 1$ and evaluate Eq. 42; then we calculate

$$b_n = (\bar{p}_{n+1} , \bar{p}_{n+1})^{-1/2}, \tag{43}$$

multiply the (un-normalized) $\bar{p}_{n+1}$ by b_n , and continue.

The process of successive orthogonalization guarantees that p_n is orthogonal to all polynomials of lesser degree in the set. Why is this so? By construction, $p_{n+1} \perp p_n$ and p_{n+1} . Is it $\perp p_{n-2}$? We need to ask whether

$$\Big(p_n , (x - \alpha_n) p_{n-2} \Big) = 0 .$$

But we know that any polynomial of degree N-1 can be expressed as a linear combination of independent polynomials of degrees 0, 1, ... , N-1. Thus

$$(x - \alpha_n) p_{n-2} \equiv \sum_{k=0}^{n-1} \mu_k p_k(x) \tag{44}$$

and (by hypothesis) $p_n \perp$ every term of the rhs of Eq. 44, hence it follows (by mathematical induction) that

$$p_{n+1} \perp \{ p_{n-2} , p_{n-3} , \cdots \} .$$

Let us illustrate the process for Legendre polynomials, defined by weight $w(x) = 1$, interval [-1,1]:

$$p_0 = \left(\frac{1}{2}\right)^{1/2},$$

$$p_1 = \left(\frac{3}{2}\right)^{1/2} x,$$

$$p_2 = \left(\frac{5}{2}\right)^{1/2} \left(\frac{3}{2}x^2 - \frac{1}{2}\right),$$

.....

These are in fact the first three (normalized) Legendre polynomials, as any standard reference will confirm.

Now we can discuss Gram polynomials. While orthogonal polynomials are usually defined with respect to an **integral** as in Eq. 40, we might also define orthogonality in terms of a **sum**, as in Eq. 39a. That is, suppose we define the polynomials such that

$$\sum_{k=0}^{M-1} p_n(x_k)\, p_m(x_k)\, \frac{1}{\sigma_k^2} \equiv \delta_{nm} = \begin{cases} 1, & m=n \\ 0, & m \neq n \end{cases} \qquad (45)$$

Then we can construct the Gram polynomials, calculating the coefficients by the algebraic steps of the Gram-Schmidt process, except now we evaluate sums rather than integrals. Since $p_n(x)$ satisfies 45 by construction, the coefficients γ_n in our fitting polynomial are simply

$$\gamma_n = \sum_{k=0}^{M-1} p_n(x_k)\, f_k\, \frac{1}{\sigma_k^2}; \qquad (46)$$

they can be evaluated without solving any coupled linear equations, ill-conditioned or otherwise. Roundoff error thus becomes irrelevant.

The algorithm for fitting data with Gram polynomials may be expressed in flow-diagram form:

Read in points f_k, x_k and $w_k \equiv 1/\sigma_k^2$.

DO n = 1 to N-1 (outer loop)
Construct a_n , c_n :

DO k = 0 to M-1 (inner loop)
$p_{n+1}(x_k) = (x_k - a_n)p_n(x_k) - c_n p_{n-1}(x_k)$

$sum = sum + \left(p_{n+1}(x_k)\right)^2 w_k$

$c_{n+1} = c_{n+1} + f_k\, p_{n+1}(x_k)\, w_k$
LOOP (end inner loop)
$c_{n+1} = c_{n+1}\ /sum$

DO k = 0 to M-1 (normalize)
$p_{n+1}(x_k) = p_{n+1}(x_k) / \sqrt{sum}$
LOOP

LOOP

Fig. 8-5 *Construction of Gram polynomials*

The required storage is 5 vectors of length M to hold x_k, $p_n(x_k)$, $p_{n-1}(x_k)$, f_k and $w_k \equiv 1/\sigma_k^2$. We also need to store the coefficients a_n, c_n and the normalizations b_n —that is, 3 vectors of length $N<<M$— in case they should be needed to interpolate. The time involved is approximately 7M multiplications and additions for each n, giving 7NM. Since N can be no greater than M-1 (M data determine at most a polynomial of degree M-1), the maximum possible running time is $7M^2$, which is much less than the time to solve M linear equations.

In practice, we would never wish to fit a polynomial of order comparable to the number of data, since this would include the noise as well as the significant information.

We therefore calculate a statistic called $\chi^2/(degree\ of\ freedom)$[24]: With M data points and an N'th order polynomial, there are M-N-1 degrees of freedom. That is, we evaluate Eq. 38 for fixed N, and divide by M-N-1. We then increase N by 1 and do it again. The value of N to stop at is the one where

$$\sigma^2_{M,N} = \frac{\chi^2_{M,N}}{M-N-1}$$

stops decreasing (with N) and begins to increase.

The best thing about the $\chi^2_{M,N}$ statistic is we can increase N without having to do any extra work:

$$\chi^2_{M,N} = \sum_{k=0}^{M-1} \left(f_k - \sum_n \gamma_n p_n(x_k)\right)^2 w_k$$

$$\equiv \sum_{k=0}^{M-1} (f_k)^2 - \sum_{n=0}^{N} (\gamma_n)^2 \qquad (47)$$

The first term after ≡ in Eq. 47 is independent of N, and the second term is computed as we go. Thus we could turn the outer loop (over N) into a **BEGIN ... WHILE ... REPEAT** loop, in which N is incremented as long as $\sigma^2_{M,N}$ is larger than $\sigma^2_{M,N+1}$. (Incidentally, Eq. 47 guarantees that as we increase N the fitted curve deviates less and less, on the average, from the measured points. When N = M-1, in fact, the curve goes through the points. But as explained above, this is a meaningless fit, since all data contain measurement errors. A fitted curve that passes closer than σ_k to more than about ⅓ of the points is suspect.)

The code for Gram polynomials is relatively easy to write using the techniques developed in Ch. 5. The program is displayed in full in Appendix 8.4.

---

24. That is, "chi-squared per degree of freedom".

§§3 Simplex algorithm

Sometimes we must fit data by a function that depends on parameters in a **nonlinear** manner. An example is

$$f_k \approx \frac{F}{1 + e^{\alpha (x_k - X)}} \qquad (48)$$

Although the dependence on the parameter F is linear, that on the parameters α and X is decidedly nonlinear.

One way to handle a problem like fitting Eq. 48 might be to transform the data, to make the dependence on the parameters linear. In some cases this is possible, but in 48 no transformation will render linear the dependence on all three parameters at once.

Thus we are frequently confronted with having to minimize numerically a complicated function of several parameters. Let us denote these by θ_0, θ_1, ... , θ_{N-1}, and denote their possible range of variation by **R**. Then we want to find those values of $\{\theta\} \subset \mathbf{R}$ that minimize a positive function:

$$\chi^2 \left(\bar{\theta}_0, \ldots, \bar{\theta}_{N-1}\right) = \min_{\{\theta\} \subset \mathbf{R}} \chi^2 \left(\theta_0, \ldots \theta_{N-1}\right) \qquad (49)$$

One way to accomplish the minimization is *via* calculus, using a method known as **steepest descents**. The idea is to differentiate the function χ^2 with respect to each θ_k, and to set the resulting N equations equal to zero, solving for the N θ's. This is generally a pretty tall order, hence various approximate, iterative techniques have been developed. The simplest just steps along in θ-space, along the direction of the local downhill gradient $-\nabla\chi^2$, until a minimum is found. Then a new gradient is computed, and a new minimum sought[25].

Aside from the labor of computing $-\nabla\chi^2$, steepest descents has two main drawbacks: first, it only guarantees to find *a* minimum, not necessarily *the* minimum – if a function has several local

---

25. This is not by itself very useful. Useful modifications can be found in Press, *et al*., *Numerical Recipes*, *ibid*., p. 301ff.

minima, steepest descents will not necessarily find the smallest. Worse, consider a function that has a minimum in the form of a steep-sided gulley that winds slowly downhill to a declivity – somewhat like a meandering river's channel. Steepest descents will then spend all its time bouncing up and down the banks of the gulley, rather than proceeding along its bottom, since the steepest gradient is always nearly perpendicular to the line of the channel.

Sometimes the function χ^2 is so complex that its gradient is too expensive to compute. Can we find a minimum *without* evaluating partial derivatives? A standard way to do this is called the **simplex method**. The idea is to construct a **simplex** – a set of N + 1 distinct and **non-degenerate** vertices in the N-dimensional θ-space ("non-degenerate" means the geometrical object, formed by connecting the N + 1 vertices with straight lines, has non-zero N-dimensional volume; for example, if N = 2, the simplex is a triangle.)

We evaluate the function to be minimized at each of the vertices, and sort the table of vertices by the size of χ^2 at each vertex, the best (smallest χ^2) on top, the worst at the bottom. The simplex algorithm then chooses a new point in θ-space by the a strategy, expressed as the flow diagram, Fig. 8-8 on page 203 below, that in action somewhat resembles the behavior of an amoeba seeking its food. The key word **)MINIMIZE** that implements the complex decision tree in Fig. 8-8 (given here in pseudocode) is

```
: )MINIMIZE  ( n.iter - - 87: rel.error - - )
    INITIALIZE
    BEGIN  done? NOT  N N.max < AND  .
    WHILE
      REFLECT  r> =best?
      IF  r> =2worst?
        IF  r<worst?  IF STORE.X THEN
            HALVE  r<worst?
            IF STORE.X ELSE SHRINK THEN
        ELSE  STORE.X  THEN
      ELSE DOUBLE    r> =best?
       IF STORE.XP ELSE STORE.X THEN
      THEN
      N 1+  IS N  SORT
    REPEAT  ;
```

used in the format

```
USE( f.name 20 % 1.E-4 )MINIMIZE
```

Fleshing out the details is a —by now— familiar process, so we leave the program *per se* to Appendix 8.5. We also include there a FORTRAN subroutine for the simplex algorithm, taken from

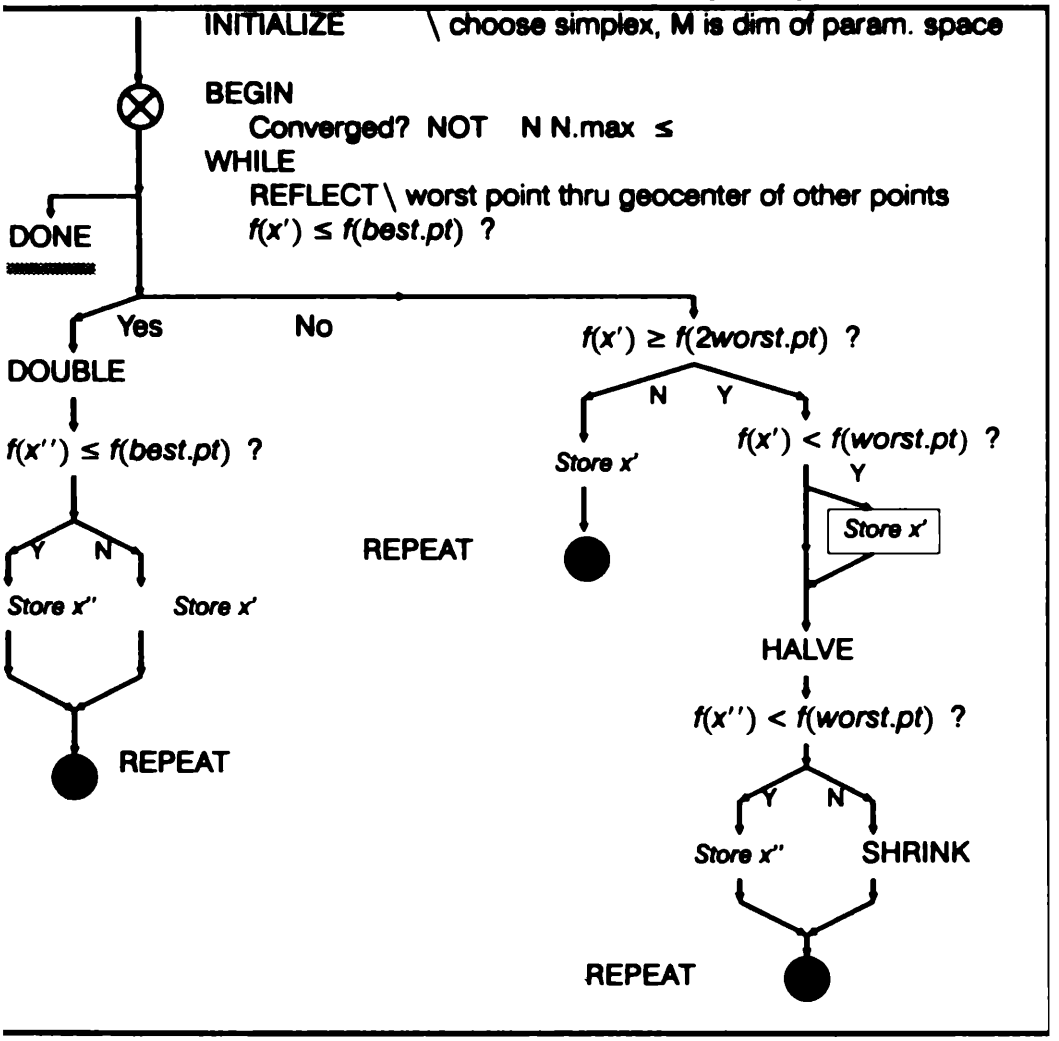


Fig. 8-8 *Flow diagram of the simplex algorithm*

Numerical Recipes [26], as an example of just how indecipherable traditional languages can be.

§3 Appendices

§§1 Gaussian quadrature

Gaussian quadrature formulae are based on the following idea: if we let the points ξ_n and weights w_n be 2N free parameters (n runs from 1 to N), what values of them most accurately represent an integral by the formula

$$I = \int_A^B dx\, \sigma(x) f(x) \approx \sum_{n=1}^{N} w_n f(\xi_n) \quad ? \qquad (50)$$

In Eq. 50 $\sigma(x)$ is a (known) positive function and $f(x)$ is the function we want to integrate. This problem can actually be solved, and leads to tables of points ξ_n and weight coefficients w_n specific to a particular interval [A,B] and weight function $\sigma(x)$. **Gauss-Legendre** integration pertains to $[-1,+1]$ and $\sigma(x)=1$. (Note any interval can be transformed into $[-1,+1]$.)

The interval $[0,\infty)$ and $\sigma(x) = e^{-x}$ leads to **Gauss-Laguerre** formulae, whereas the interval $(-\infty,+\infty)$ and $\sigma(x) = e^{-x^2}$ leads to **Gauss-Hermite** formulae.

Finally, we note that the more common integration formulae such as Simpson's rule or the trapezoidal rule can be derived on the same basis as the Gauss methods, except that the points are specified in advance to be equally spaced and to include the endpoints of the interval. Only the weights w_n can be determined as free fitting parameters that give the best approximation to the integral.

For given N Gaussian formulae can be more accurate than equally-spaced rules, as they have twice as many parameters to play with.

---

26. Press, *et al.*, *Numerical Recipes*, *ibid.*, p. 289ff.

Some FORTH words for 5-point Gauss-Legendre integration:

```
% 0.906179845938664   FCONSTANT  x2
% 0.538469310105683   FCONSTANT  x1
% 0.568888888888889   FCONSTANT  w0
% 0.478628670499366   FCONSTANT  w1
% 0.236926885056189   FCONSTANT  w2

: scale  ( 87: A B - - [A+B]/2 [B-A]/2 )
        FOVER F- F2/ FUNDER F+ ;
: rescale        ( 87: a b x - - a+b*x) F* F+ ;
: F3R+ ( 87: a b c x - - a+x b c) F3R F+ F-ROT ;
```

```
: }Integral       ( 87: A B - - I)
        scale  ( 87: A B - - [A+B]/2 [B-A]/2 )
        FOVER F(X) w0 F* F-ROT  ( 87: - - I a b)
        XDUP  x1 rescale  F(X)  w1 F*  F3R+
        XDUP  x1 FNEGATE rescale
        F(X)  w1 F*  F3R+
        XDUP  x2 rescale  F(X)  w2 F*  F3R+
        XDUP  x2 FNEGATE rescale
        F(X)  w2 F*  F3R+
```

§§2 The trapezoidal rule

Recall (Ch. 8, §1.1) how we approximated the area under a curve by capturing it between rectangles consistently higher- and lower than the curve; and calculating the areas of the two sets of rectangles. In practice we use a better approximation: we average the rectangular upper and lower bounds. The errors tend to cancel, resulting in[27]

$$\frac{w}{2}\sum_{n=0}^{(B-A)/w}\Big(f(A+nw)+f(A+nw+w)\Big) \tag{51}$$

$$\approx \int_A^B dx\, f(x) + \frac{1}{12}\left(\frac{B-A}{w}\right) w^3 \max_{A<x<B}\left|f''(x)\right|$$

Now the error is much smaller[28] – of order w^2 – so if we double the number of points, we decrease the error four-fold. Yet this so-called **trapezoidal rule**[29] requires no more effort (in terms of the number of function evaluations) than the rectangle rule.

---

27. See, *e.g.*, Abramowitz and Stegun, *HMF*, p. 885.
28. $f''(x)$ is the second derivative of $f(x)$, *i.e.* the first derivative of $f'(x)$.
29. Called so because the curve $f(x)$ is approximated by straight line segments between successive points $x, x+w$. Thus we evaluate the areas of trapezoids rather than rectangles.

§§3 Richardson extrapolation

In the program)**INTEGRAL** in Ch.8 §1§§5.3 the word **INTERPOLATE** performs Richardson extrapolation for the trapezoidal rule. The idea is this: If we use a given rule, accurate to order w^n, to calculate the integral on an interval [a,b], then presumably the error of using the formula on each half of the interval and adding the results, will be smaller by 2^{-n}. For the trapezoidal rule, n = 2, hence we expect the error from summing two half-intervals to be 4× smaller than that from the whole interval.

Thus, we can write $\left(I_0 = \int_a^b \ , \ I'_0 = \int_a^{(a+b)/2} \ , \ I_1 = \int_{(a+b)/2}^b \right)$

$$I_0 = I_{exact} + R \tag{52a}$$

$$I'_0 + I_1 = I_{exact} + \frac{R}{4} \tag{52b}$$

Equation 52b is only an approximation because the R that appears in it is not exactly the same as R in Eq. 52a. We will pretend the two R's are equal, however, and eliminate R from the two equations (8.52a,b) ending with an expression for I_{exact}:

$$I_{exact} \approx \frac{4}{3}\left(I'_0 + I_1 - I_0\right) \tag{53}$$

Equation 53 is exactly what appears in **INTERPOLATE**.

§§4 Linear least-squares: Gram polynomials

Here is a FORTH program for implementing the algorithm derived in §2§§2 above.

```
ASK GRAM1

      CODEed words to simplify fstack management
CODE G(N+1) 4 FMUL. FCHS. 2 FLD. 6 FSUB.
    2 FMUL. 1 FADDP.
    DX DS MOV. DS POP.  R64 DS: [BX] FST.
    DS DX MOV. BX POP.  END-CODE
( sag off -- 87: s a b w x g[n] g[n-1] -- s a b w x g[n] g[n+1] )

CODE B(N+1) 2 FXCH. 2 FMUL. 3 FMUL.
    1 FXCH. 1 FMUL. DX DS MOV. DS POP.
    R64 DS: [BX] FADD. DS: [BX] FSTP.
    DS DX MOV. BX POP. END-CODE
( sag off -)
 87: s a b w x g[n] g[n+1] -- s a b w g[n+1] w x g[n+1] )

CODE A(N+1) 1 FMUL. DX DS MOV. DS POP.
    R64 DS: [BX] FADD.
    DS: [BX] FSTP.  DS DX MOV. BX POP. END-CODE
( sag off -- 87: s a b w g[n+1] wxg[n+1] -- s a b w g[n+1] )

CODE C(N+1) 1 FXCH. 2 FMUL*. 2 FMUL.
    2 FXCH. 1 FMULP.
    DX DS MOV. DS POP.
    R64 DS: [BX] FADD. DS: [BX] FSTP.
    DS DX MOV. BX POP. 3 FADDP. END-CODE
( sag off -- 87: s a b w g[n+1] f -- s=s+wg[n+1]**2 a b )

-------------------------Gram polynomial coding
REAL SCALAR DELTA
VAR Nmax
usage: A{ B{ C{ G{{ Nmax }FIT

: FIRST.AB's  F=0 a{ 0 } G!  F=0 b{ 0 } G! ;

: INIT.DELTA  (87: -- g{{ I I }} )  F=0 F=0
    M 0 DO w{ I } G@ y{ I } G@  F**2 FOVER F*
    (87: s s' -- s s' w w*I^2)
    FROT F+  F-ROT F+  FSWAP
    (87: -- s=s+w s'=s'+w*I^2)
  LOOP  DELTA G! FSQRT 1/F ;

: FIRST.G's (87: g{{ 1 I }} -- g{{ 1 I }} )
  M 0 DO F=0 g{{ 0 I }} G! FDUP g{{ 1 I }} G! LOOP ;

: SECOND.AB's  (87: g{{ 1 I }} -- g{{ 1 I }} )
  F=0 b{ 1 0 } G!  FDUP F**2
  F=0 M 0 DO w{ I 0 } G@ x{ I 0 } G@ F* F+ LOOP
  F* a{ 1 0 } G! ;

: FIRST.& SECOND.C's (87: g{{ 1 I }} -- )
  F=0 c{ 0 0 } G! F=0
  M 0 DO w{ I 0 } G@ y{ I 0 } G@ F* F+ LOOP
  F* c{ 1 0 } G! ;

: INITIALIZE  FINIT  IS Nmax  IS g{{  IS c{  IS b{  IS a{
  FIRST.AB's  INIT.DELTA  FIRST.G's  SECOND.AB's
  FIRST.& SECOND.C's ;

0 VAR N  0 VAR N+1
: inc.N  N+1 DUP IS N  1+  IS N+1 ;
: DISPOSE  R DDUP R@ G@  R ;
: inc.OFF  #BYTES DROP + ;
: ZERO.L  DDUP F=0 R64!L ;

: START.Next.G
( -- [c{n+1}] [a{n+1}] [b{n+1}]  87: -- s=0 a{n} b{n} )
    FINIT  F=0  c{ N+1 0 } DROP  ZERO.L
    a{ N 0 } DISPOSE  inc.OFF  ZERO.L
    b{ N 0 } DISPOSE  inc.OFF  ZERO.L ;


: SET.FSTACK    w{ I' 0 } G@  x{ I' 0 } G@  g{{ N I' }} G@
     g{{ N 1- I' }} G@ ;

: }@*I  (adr n -- 87: x -- x)  FDUP 0} DISPOSE F* G! ;

: NORMALIZE  (87: sum -- )
     1/F a{ N+1 }@*I  FSQRT
     b{ N+1 }@*I  c{ N+1 }@*I
     M 0 DO FDUP g{{ N+1 I }} DISPOSE F* G! LOOP
     FDROP ;
CODE 6DUP  OPT" 6 PICK 6 PICK 6 PICK
          6 PICK 6 PICK 6 PICK " END-CODE
CODE 6DROP OPT" DDROP DDROP DDROP " END-CODE
```

```
\ GRAM POLYNOMIAL LEAST-QUARES (CONT'D)

Next.G  START.Next.G
   M 0 DO  6DUP  SET.FSTACK
       g{{ N+1 I }} DROP  G(N+1)  B(N+1)  A(N+1)
       y{ I 0} G@  C(N+1)
   LOOP  6DROP  FDROP FDROP  NORMALIZE ;

: New.DELTA  (:: – old.delta new.delta)
     DELTA G@  FDUP  c{ N 0} G@  F**2  F-  FDUP
      DELTA G! ;

: NOT.ENUF.G's?  New.DELTA  M N+1 -  S-F
     FDUP  F=1 F-
( CR  .FS  ." NEXT ITERATION?" ?YN  0= IF ABORT THEN )
     (:: – d d' m-n-1  m-n-2 )
     FROT  F\  F-ROT  F/  (:: – d'/[m-n-2]  d/[m-n-1] )
     FOVER  F0  IF  F  ELSE  FDROP FDROP 0 THEN ;

: }FIT  ( X{ Y{ S{ Nmax A{ B{ C{ G{{ – )
    INITIALIZE   1 IS N  2 IS N+1
    BEGIN    NOT.ENUF.G's?   N Nmax  AND
    WHILE    Next.G  inc.N
    REPEAT  FINIT ;
\---------------------------------- end of code

: RECONSTRUCT  M 0  DO  CR  x{ I 0} G@ F.  y{ I 0} G@ F.
         F=0  N+1 1+  1 DO
            c{ I 0} G@  g{{ I J }} G@  F* F+
           LOOP  F.
        LOOP  ;
```

§§5 Non-linear least squares: simplex method

A FORTRAN program for the simplex method is given bel
on page 209. The FORTH version, as discussed in §2§§3
given on pages 210 and 211.

```
      SUBROUTINE AMOEBA(P,Y,MP,NP,NDIM,FTOL,FUNK,ITER)
      PARAMETER (NMAX=20,ALPHA=1.0,
     ;     BETA=0.5,GAMMA=2.0,ITMAX=500)
      DIMENSION P(MP,NP),Y(MP),PR(NMAX),PRR(NMAX),PBAR(NMAX)
      MPTS=NDIM+1
      ITER=0
      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
1     CONTINUE
      RTOL=2.*ABS(Y(IHI)-Y(ILO))/(ABS(Y(IHI))+ABS(Y(ILO)))
      IF(RTOL.LT.FTOL)RETURN
      IF(ITER.EQ.ITMAX) PAUSE 'Amoeba exceeding maximum iterations.'
      ITER=ITER+1
      DO 12 J=1,NDIM
        PBAR(J)=0.
2     CONTINUE
      DO 14 I=1,MPTS
        IF(I.NE.IHI)THEN
          DO 13 J=1,NDIM
            PBAR(J)=PBAR(J)+P(I,J)
3         CONTINUE
        ENDIF
4     CONTINUE
      DO 15 J=1,NDIM
        PBAR(J)=PBAR(J)/NDIM
        PR(J)=(1.+ALPHA)*PBAR(J)-ALPHA*P(IHI,J)
5     CONTINUE
      YPR=FUNK(PR)
      IF(YPR.LE.Y(ILO))THEN
        DO 16 J=1,NDIM
          PRR(J)=GAMMA*PR(J)+(1.-GAMMA)*PBAR(J)
6       CONTINUE
        YPRR=FUNK(PRR)
        IF(YPRR.LT.Y(ILO))THEN
          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)=0.5*(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
      ENDIF
      GO TO 1
      END
```

```
\ MINIMIZATION BY THE SIMPLEX METHOD
\ VERSION OF 20:51:19 4/30/1991

TASK AMOEBA

\ ---------------------- FUNCTION NOTATION
  VARIABLE <F>
  : USE( [COMPILE] ' CFA <F> ! ;
  : F(X) EXECUTE@ ;
  BEHEAD' <F>
\ ------------------- END FUNCTION NOTATION

\ ------------------------ DATA STRUCTURES
3 VAR Ndim
0 VAR N
0 VAR N.max

CREATE SIMPLEX{{ Ndim 4 (bytes) * Ndim 1+ (# points)
    * ALLOT
CREATE F{ Ndim 1+ 4 (bytes) * ALLOT \ residuals
CREATE index Ndim 1+ 2* ALLOT \ array for scrambled indices

: >index (I - I') 2* index + @ ;
DVARIABLE Residual
DVARIABLE Residual'
DVARIABLE Epsilon
CREATE X{ Ndim 4 (bytes) * ALLOT \ trial point
CREATE XP{ Ndim 4 (bytes) * ALLOT \ 2nd trial point
CREATE Y{ Ndim 4 (bytes) * ALLOT \ geocenter

: } (adr n - adr + 4n) 4* + ; \ part of array notation
: }} (adr m n - adr + [m*Ndim + n]*4) SWAP Ndim * + } ;
\ ---------------- ---- END DATA STRUCTURES
\ ---------------------------- ACTION WORDS
: RESIDUALS
    Ndim 1+ 0 DO SIMPLEX{{ I 0 }} F(X) F{ I } R32!
    LOOP ;
: <index> Ndim 1+ 0 DO I index I 2* + ! LOOP ; \ fill index

: ORDER <index>
    Ndim 1+ 0 DO F{ I >index } R32@
    I 1+ BEGIN Ndim 1+ OVER
    WHILE F{ OVER >index } R32@ FOVER FOVER F>
        IF FSWAP
            I >index OVER >index
            index I 2* + ! index 3 PICK 2* + !
        THEN FDROP 1+
    REPEAT FDROP DROP
    LOOP ;
CENTER ( - ) FINIT Ndim S-F (87: - Ndim)
    Ndim 0 DO F=0 \ loop over components
        Ndim 0 DO \ average over vectors

            SIMPLEX{{ I index J }} R32@ F+
        LOOP FOVER F/
        Y{ I } R32! \ put away
    LOOP FDROP ;
\ note: Worst.Point is SIMPLEX{{ Ndim index J }}
\ -- excluded!

: DONE! CR ." We're finished." ;
: TOO.MANY CR ." Too many iterations." ;

: V.MOVE (src.adr dest.adr - ) \ move vector
    Ndim 0 DO OVER I } OVER I } 2 MOVE
  LOOP DDROP ;
: STORE (adr1 adr2 -- ) 0 }
    SIMPLEX{{ Ndim index 0 }} V.MOVE
    F{ Ndim index } 2 MOVE ;
: STORE.X Residual X{ STORE ;
: STORE.XP Residual' XP{ STORE ;

: New.F X{ F(X) Residual R32! ;
: EXTRUDE (87: scale.factor -- )
\ extend pseudopod
    Ndim 0 DO DUP I } R32@ (87: -- s.f x)
    Y{ I } R32@ FUNDER F- (87: -- s.f y x-y)
    FROT FUNDER F* (87: -- y s.f [x-y]*s.f)
    FROT F+ (87: -- s.f y+[x-y]*s.f)
    X{ I } R32!
    LOOP DROP FDROP New.F ;

: F=1/2 F=1 FNEGATE F=1 FSCALE FPLUCK ;
: F=2 F=1 FDUP FSCALE FPLUCK ;

\ ---------------------- DEBUGGING CODE
0 VAR DBG
: DEBUG-ON -1 IS DBG ;
: DEBUG-OFF 0 IS DBG ;
: .V 3 SPACES Ndim 0 DO DUP I } R32@ F.
    LOOP DROP ;
: .M (--)
    Ndim 1+ 0 DO DBG CR
    IF I . 2 SPACES THEN
    SIMPLEX{{ I index 0 }} .V
    DBG IF F{ I index } R32@ F. THEN
    LOOP CR
  DBG IF CR X{ .V Residual R32@ F. THEN ;

: .F Ndim 1+ 0 DO CR F{ I index } R32@ F.
    LOOP ;
\ ----------------- END DEBUGGING CODE
: REFLECT CENTER
    F=1 FNEGATE \ scale.factor = -1
    SIMPLEX{{ Ndim index 0 }} \ worst pt.
    EXTRUDE \ calculate x, f(x)
    DBG IF CR ." REFLECTING" THEN .M ;
```

```
AMOEBA CONT'D

DOUBLE   Residual Residual' 2 MOVE
   \ save residual
     X{ XP{ V.MOVE              \ save point
     FINIT F=2 FNEGATE (?)   \ scale factor = -2
     SIMPLEX{{ Ndim index 0 }}     \ worst pt.
     EXTRUDE                  \ calculate x, f(x)
     DBG IF CR ." DOUBLING" THEN .M ;

HALVE    FINIT F=1/2              \ scale factor = 0.5
     SIMPLEX{{ Ndim index 0 }}     \ worst pt.
     EXTRUDE                  \ calculate x, f(x)
     DBG IF CR ." HALVING" THEN .M ;
SHRINK   SIMPLEX{{ 0 >index 0 }}       \ best pt.
     Y{ V.MOVE                \ save it
     Ndim 1+ 1 DO             \ by vector
       Ndim 0 DO              \ by component
         SIMPLEX{{ J >index I }} DUP
         R32@ Y{ I } R32@  FUNDER F-
         F=1/2 F* F+   R32!
       LOOP
     LOOP  RESIDUALS  ORDER
     DBG IF CR ." SHRINKING" THEN .M ;
-------------------------- END ACTION WORDS
------------------------------- TEST WORDS
test) FINIT Residual R32@
   F{ SWAP >index } R32@  F> NOT ;
>=best?  0    (test) ;

<worst?  Ndim  (test) NOT ;
>=2worst? Ndim 1- (test)  ;

done?  F{ Ndim index } R32@
   F{ 0  index } R32@
   FOVER FOVER   F- F2*
   F-ROT F+ F/ FABS  Epsilon R32@  F> ;
--------------------------- END TEST WORDS
```

```
: )MINIMIZE  ( n.iter -- 87: error -- )
   IS N.max  Epsilon R32!  0 IS N        \ initialize
   RESIDUALS                    \ compute residuals
   ORDER                        \ locate best, 2worst, worst
   BEGIN                        \ start iteration
     done?  NOT   N N.max  <  AND
   WHILE
     REFLECT  r> =best?
     IF  r> =2worst?
       IF  r<worst?  IF STORE.X THEN
         HALVE  r<worst?
         IF STORE.X ELSE SHRINK THEN
       ELSE  STORE.X  THEN
     ELSE DOUBLE
       r> =best?
       IF STORE.XP ELSE STORE.X THEN
     THEN
     N 1+ IS N   ORDER
   REPEAT  DONE! ;

\------------------------ EXAMPLE FUNCTIONS
: F1     ( adr -- 87: F ) DUP 0 } R32@ FDUP F**2
                DUP 1 } R32@ F**2  F2* F+
                  2 } R32@ F**2  F2* F2* F+
      36 S-F F- F**2 F2/ FSWAP 6 S-F F* F- ;
\ f1 = .5 * (x * x + 2 * y * y + 4 * z * z - 36) ^ 2 - 6 * x

: F2     ( adr -- 87: F ) DUP DUP 0 }
     R32@ FDUP F**2 1 } R32@ F**2
          F2* F+  36 S-F F- F**2 F2/ FSWAP 6 S-F F* F-
F**2 F* ;  DUP 1 } R32@ 0 } R32@ F/ FATAN F2* FCOS
\ f2 = [(x ^ 2 2*y ^ 2 - 36) ^ 2 - 6*x] * cos(2*atan(y/x)) ^ 2
\ Usage:  USE( MYFUNC 10 1.E-3 )MINIMIZE
FLOATS
5. SIMPLEX{{ 0 0 }} R32!      5. SIMPLEX{{ 1 0 }} R32!
-3. SIMPLEX{{ 0 1 }} R32!     3. SIMPLEX{{ 1 1 }} R32!
7. SIMPLEX{{ 0 2 }} R32!      -1.5 SIMPLEX{{ 1 2 }} R32!

-10. SIMPLEX{{ 2 0 }} R32!    5. SIMPLEX{{ 3 0 }} R32!
1. SIMPLEX{{ 2 1 }} R32!      3. SIMPLEX{{ 3 1 }} R32!
3. SIMPLEX{{ 2 2 }} R32!      3. SIMPLEX{{ 3 2 }} R32!
```