More Programming Examples

Contents

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:

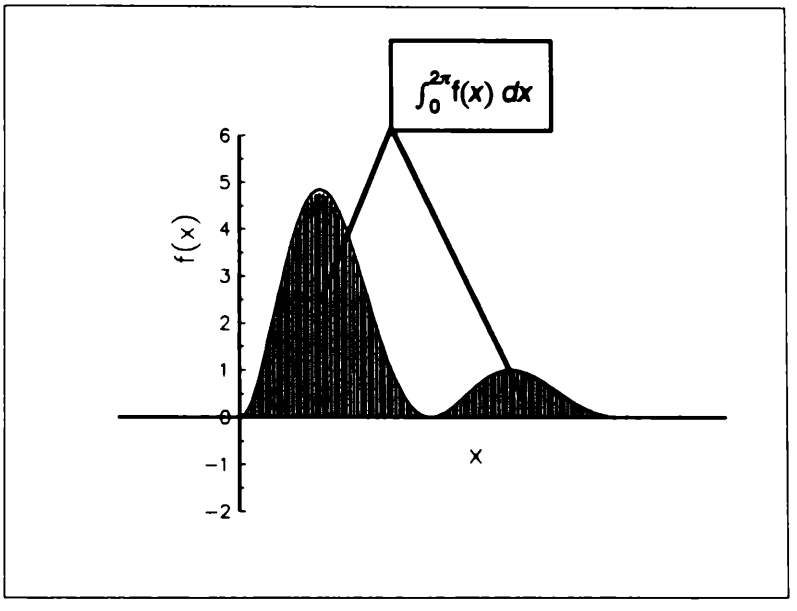


Fig. 8-1 *The integral of a function is the area under the curve.*

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:

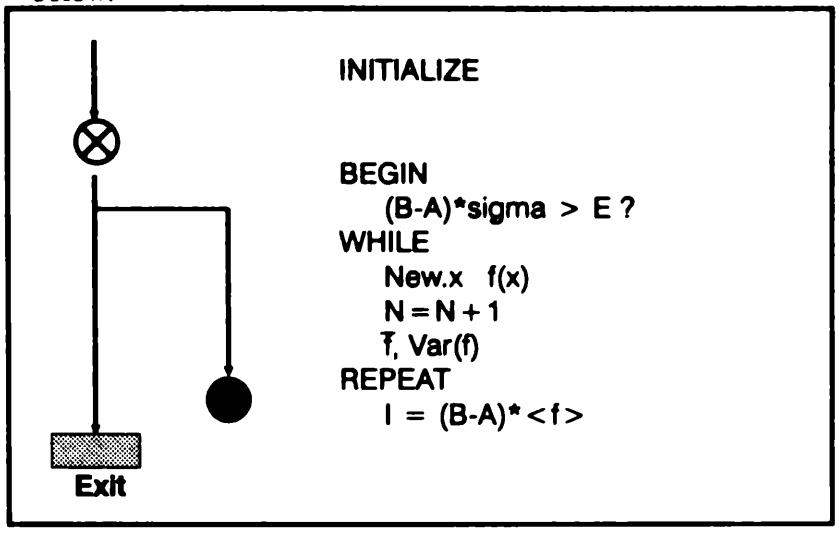


Fig. 8-2 *Flow diagram of Monte Carlo Integration*

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.

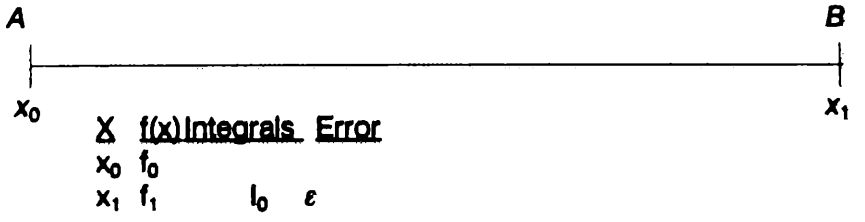
To minimize evaluations of $f(x)$, we shall have to save values $f(x_n)$ that can be re-used as we subdivide the intervals.

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.

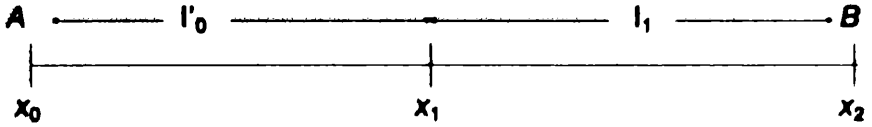
To clarify what we are going to do, let us visualize the interval of integration, and mark the mesh points (where we evaluate $f(x)$ with + :

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.

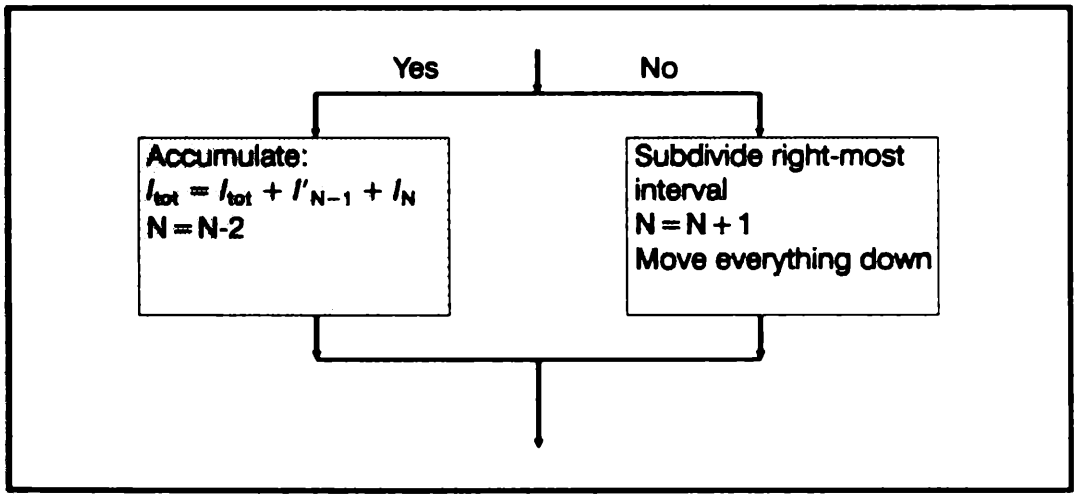
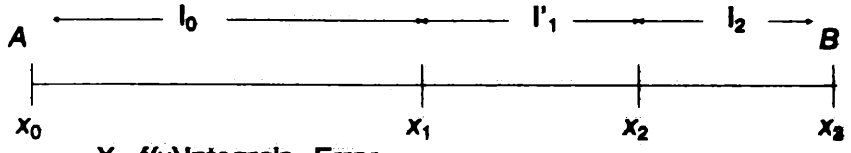


Fig. 8-3 **SUBDIVIDE** *branch in adaptive integration*

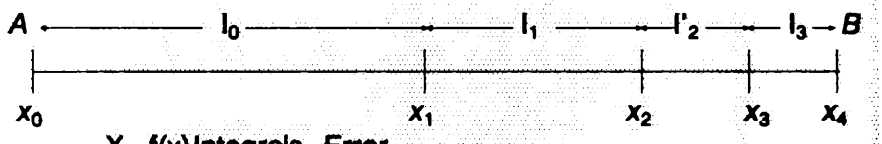
If the two integrals disagree, we subdivide again, as in Step 3 and Step 4 below:

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	$\varepsilon/4$

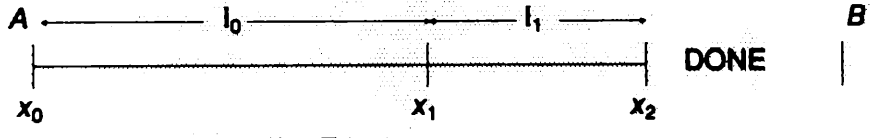
Step 4: N = N + 1 = 4



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:

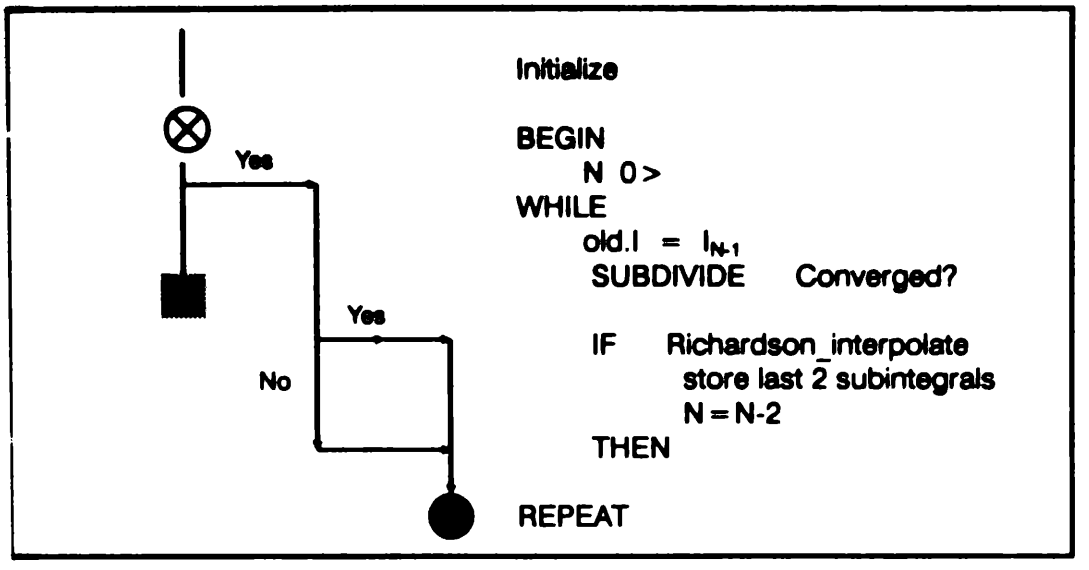


Fig. 8-4 *Non-recursive adaptive quadrature*

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.