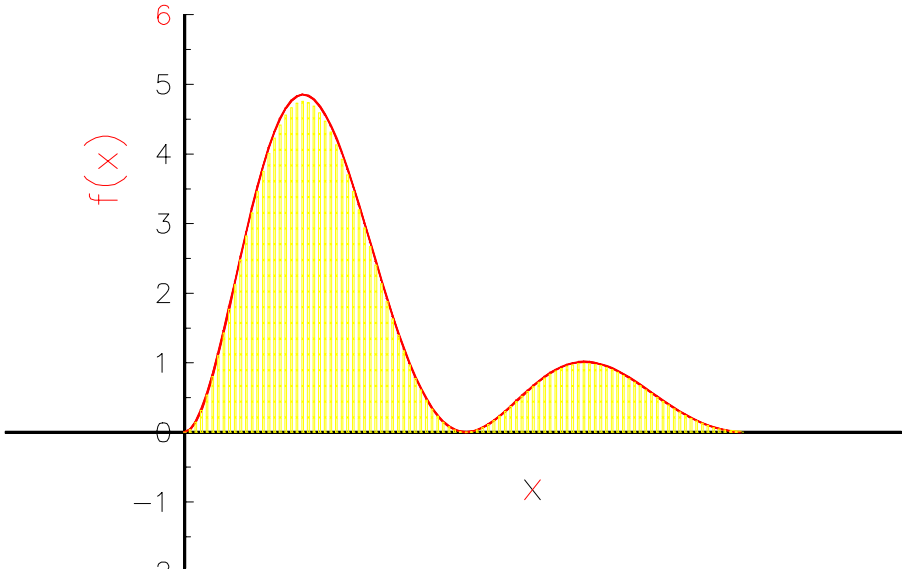
Numerical quadrature

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 *quadrature*. Finally, we discuss several 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:



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.

$$w \sum_{n=0}^{(b-a)/w} \min\Big[f(a+nw), f(a+nw+w)\Big] \le \int_a^b dx\, f(x) \le w \sum_{n=0}^{(b-a)/w} \max\Big[f(a+nw), f(a+nw+w)\Big].$$

It is easy to see that each rectangle in the upper bound is about $w\,|f'(x)|$ too high[2] on 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

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

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 $\int_a^b dx\, f(x)$ 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 $\int_a^b dx\, f(x)$ 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) is

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

so that

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

This 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.

Thus we could calculate a definite integral using a differential equation solving program (to be developed in a subsequent chapter). We can express the problem in the following form:

---

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

Solve the differential equation

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

on the interval $x \in [a, b]$ with the initial condition $F(a) = 0$.

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 following is a "brute force" Monte Carlo integration scheme: if a curve is entirely bounded within a box of corners (xa, ya), (xb, ya), (xa, yb), (xb, yb) then if we pick points at random in the box, the fraction that fall inside the curve is, in the limit of many points, proportional to the ratio of the area under the curve to that of the box. It is rather like throwing darts at random, at a target on a wall. The darts must go *somewhere* on the wall, and the fraction that hit the target is, within statistical errors, the ratio of the area of the target to that of the wall.

The following routine implements this idea:

```
\ Brute force Monte Carlo integration in 1 dimension
\
\     (c) Copyright 1998  Julian V. Noble.
\         Permission is granted by the author to
\         use this software for any application pro-
\         vided this copyright notice is preserved.
\
\ Usage:  use( fn.name xa xb ya yb )bfmc
\ Examples:
\ use( fsqrt  10000 0e 2e 0e 2e fsqrt )bfmc fs. 1.88006E0  ok
\ use( fsqrt  10000 0e 2e 0e 2e fsqrt )bfmc fs. 1.90806e0  ok
\ use( fsqrt  10000 0e 2e 0e 2e fsqrt )bfmc fs. 1.88486e0  ok
\ use( fsqrt  10000 0e 2e 0e 2e fsqrt )bfmc fs. 1.89363E0  ok

\ Environmental dependencies:
\  FLOAT wordset, separate floating point stack

MARKER -bfmc

\ Conditional definition of non-Standard words
: undefined    BL WORD  FIND  NIP  0=  ;
undefined  prng  [IF]  include  prng.f  [THEN]
undefined  s>f   [IF] : sf   S>D  D>F  ;    [THEN]
```

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

```
undefined  f^2   [IF] : f^2   FDUP  F*  ;    [THEN]
undefined  ftuck [IF] : ftuck  FSWAP  FOVER  ;   [THEN]

undefined  use(  [IF]
\ Vectoring: for using function names as arguments
: use(      '       \ state-smart ' for syntactic sugar
    STATE @  IF  POSTPONE LITERAL  THEN  ;  IMMEDIATE
' NOOP  CONSTANT  'noop
: v:   CREATE  'noop  ,  DOES> PERFORM  ;   \ create dummy def'n
: 'dfa   ' BODY  ;                          ( — data field address)
: defines    'dfa   STATE @
     IF   POSTPONE  LITERAL    POSTPONE  !
     ELSE   !   THEN  ;  IMMEDIATE
\ end vectoring
[THEN]

\ Data structures
    v: fdummy

    1000 VALUE  Nmax
    0    VALUE Npoints
    0    VALUE Nhits

    FVARIABLE xa    FVARIABLE xb-xa
    FVARIABLE ya    FVARIABLE yb-ya

\ Program begins here
    0.1 seed 2!     \ initialize prng

: x     ( f: — x = xa + xi*[xb-xa])    \ guess a new point
    prng   xb-xa F@  F*   xa F@   F+  ;

: y     ( f: — y = ya + xi*[yb-ya])    \ guess a new point
    prng   yb-ya F@  F*   ya F@   F+  ;

: new_point
    Npoints  1+  TO Npoints
    x fdummy    y F>
    IF  Nhits  1+  TO Nhits  THEN  ;

: initialize       ( xt n —)   ( f:  xa xb ya yb —)
    TO Nmax     0 TO Npoints    0 TO Nhits    defines fdummy
    FOVER  F-  yb-ya F!     ya F!    FOVER  F-  xb-xa F!     xa F!   ;

: )bfmc   ( xt —)   ( f:  xa xb ya yb — integral)
    initialize
    BEGIN   Npoints Nmax
    WHILE   new_point
    REPEAT  Nhits  s>f   Npoints  s>f   F/
    xb-xa F@   yb-ya F@  F*  F*  ;
```

This random-sampling method is called the *Monte Carlo* method because of the element of chance. The disadvantages are twofold: first, at every step, one more random number must be computed than the dimensionality of the volume of integration. Thus for a function of one variable we need two

random numbers per step, *etc.* The second disadvantage is that a bounding rectangular hyper-solid must be known in advance. In some applications this may be difficult to arrange.

Straight sampling Monte-Carlo integration

Another way to integrate by random sampling uses the following (obvious) fact: the area under a curve $f(x)$ is exactly equal to the average height $\langle f \rangle$ of $f(x)$ on the interval $[a, b]$, times the length, $b - a$, of the interval[4]. How can we estimate $\langle f \rangle$? Suppose we sample $f(x)$ at random, choosing N points in $[a, b]$ with a random number generator. Then

$$\langle f \rangle \approx \frac{1}{N} \sum_{n=1}^{N} f(x_n)$$

and

$$\int_a^b f(x)\, dx \approx (b - a) \langle f \rangle .$$

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

$$\mathrm{Var}(f) \overset{df}{=} \int_{-\infty}^{\infty} df\, \rho(f) \left(f - \langle f \rangle \right)^2$$

where $\rho(f)\, df$ is the probability of measuring a value of f between f and $f + df$. Statistical theory says the variance in estimating $\langle f \rangle$ by random sampling is

$$\mathrm{Var}\left(\langle f \rangle \right) = \frac{1}{N} \mathrm{Var}\,(f)$$

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

$$\Delta \left(\int_a^b f(x)\, dx \right) \approx (b - a) \left[\frac{\mathrm{Var}\,(f)}{N} \right]^{1/2}$$

and is therefore guaranteed to decrease as $N^{-1/2}$.

4. That is, this statement *defines* $\langle f \rangle$.

Here is a simple Forth program that implements this scheme:

```
\ One-dimensional Monte-Carlo integration

\     (c) Copyright 1998  Julian V. Noble.          \
\       Permission is granted by the author to      \
\       use this software for any application pro-  \
\       vided this copyright notice is preserved.   \

\ Usage:  use( fn.name xa xb )monte
\ Examples:
\       use( FSQRT 10000 0e  1e  )monte FS. 6.67675E-1  ok
\       use( FSQRT 10000 0e  2e  )monte FS. 1.88408E0  ok
\       : F1     FDUP FSQRT F*  ;  ok
\       use( f1  0e  1e  1e-3 )monte FS. 3.97621E-1  ok
\       use( f1  0e  2e  1e-4 )monte FS. 2.27428E0  ok

MARKER -mcint
\ Conditional definition of non-Standard words
: undefined    BL WORD  FIND  NIP  0=  ;
undefined  prng  [IF]  include  prng.f  [THEN]
undefined sf   [IF] : sf   SD  DF  ;    [THEN]
undefined f^2   [IF] : f^2   FDUP  F*  ;    [THEN]
undefined ftuck [IF] : ftuck  FSWAP  FOVER  ;   [THEN]
undefined  use( [IF]

\ Vectoring: for using function names as arguments
: use(      '       \ state-smart ' for syntactic sugar
    STATE @  IF  POSTPONE LITERAL  THEN  ;  IMMEDIATE
' NOOP  CONSTANT  'noop
: v:   CREATE  'noop  ,  DOES PERFORM  ;   \ create dummy def'n
: 'dfa   ' BODY  ;                         ( — data field address)
: defines    'dfa   STATE @
             IF   POSTPONE  LITERAL    POSTPONE  !
             ELSE   !   THEN  ;  IMMEDIATE
\ end vectoring [THEN]

\ Program starts here
\ Data structures
    v: fdummy
    1000 VALUE  Nmax
    0 VALUE Npoints
    0.1 seed 2!
    FVARIABLE xa    FVARIABLE xb-xa
    FVARIABLE  Var
    FVARIABLE
\ Actions
    : x     ( f: — x = xa + xi*[xb-xa])    \ guess a new point
        prng   xb-xa F@  F*   xa F@   F+  ;

    : initialize    ( xt n —)    ( f: xa xb error — integral)
        TO Nmax
        defines  fdummy
        5 TO Npoints
        FOVER  F-  xb-xa F!     xa F!
```

```
    f0.0  <f> F!   f0.0 Var F!
    5 0  DO  x  fdummy  FDUP
             <f> F@ F+  <f> F!
             f^2   Var  F@   F+   Var  F!
    LOOP
     <f>  F@ Npoints  s>f  F/   <f>  F!
    Var  F@   F@  f^2  Npoints s>f F*  F-   Var F!  ;

: New_point
    x fdummy
     <f> F@  ftuck  F-  ftuck              ( f: f-<f>  <f>  f-<f>)
    Npoints 1+  DUP TO  Npoints     \ n=n+1
    s>f  F/   F+   <f>   F!         \ <f'> = <f> + (f-<f>)/(n+1)
    Npoints DUP  1-  s>f  F*
    s>f   F/                        ( f: n*[f-]^2/[n+1])
    Var F@   F+   Var F!            \ Var' = Var + n*(f-)^2/(n+1)
;

: )monte    ( xt - - )    ( f: xa xb error - - integral)
    initialize
    BEGIN     Npoints Nmax <
    WHILE     New_point
    REPEAT    F@ xb-xa F@ F* ;

: %error   ( f: – error )
    Var F@  FSQRT  Npoints s>f  F/
     <f> F@ FABS FDUP F0>
    IF    F/  1.e2  F*
    ELSE  ."  too close to 0" THEN ;
```

It is easy to see that the Monte-Carlo method converges slowly. Since the error decreases only as $N^{-1/2}$, whereas even so crude a rule as adding up rectangles has an error term that decreases as N^{-1}, 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. The reason they are so much faster is that, say, using Simpson's rule with n points in each of ten dimensions we must take roughly $N = n^{10}$ points, but the accuracy is of order $n^{-4} \equiv N^{-2/5}$. That is, the uncertainty using a compounded quadrature formula in many-dimensional integration can be more slowly decreasing than that from simple Monte Carlo integration.

The preceding programs are easily generalized to an arbitrary number of dimensions. (The generalization is left as an exercise.)

4. Numerical quadrature rules

In the preceding sections we examined two schemes for computing the integral of an arbitrary function: the first, a deterministic method, involved adding up the areas of rectangles that lay entirely below or entirely above the given curve, in order to derive lower and upper bounds for the area. This method is more of an existence proof than a useful algorithm. But it leads us to consider possible improvements. The main idea is to approximate the given function by a polynomial over some integral, then to integrate that polynomial exactly[5].

Trapezoidal rule

The simplest polynomial is the straight line joining the points $\left(x_0\,,\ f_0\right)$ and $\left(x_1\,,\ f_1\right)$:

$$y(x) \;=\; f_0\,\frac{x-x_1}{x_0-x_1} \;+\; f_1\,\frac{x-x_0}{x_1-x_0}\,,$$

whose integral is

$$\int_{x_0}^{x_1} y(x)\,dx \;=\; \frac{1}{2}\left(x_1-x_0\right)\left(f_0+f_1\right) \approx \int_{x_0}^{x_1} f(x)\,dx\ .$$

Since this is the area of the trapezoid bounded by the straight line $y(x)$, the x-axis and the vertical lines at x_0 and x_1, this quadrature formula is called the *trapezoidal rule*. Normally we take the points close together; to employ the trapezoidal rule over a finite interval we simply add the areas from the sub-intervals. In practice this means we take equally spaced intervals and write

$$\int_a^b f(x)\,dx \;\approx\; h\left(\frac{1}{2}f_0+f_1+f_2+\ldots+\frac{1}{2}f_n\right) - \frac{nh^3}{12}f^{(2)}(\xi)$$

where h is the spacing between successive points.

A similar approach can be taken using a quadratic approximation to equally-spaced function values:

$$y(x) \;=\; f_0\left(\frac{x-x_1}{x_0-x_1}\right)\left(\frac{x-x_2}{x_0-x_2}\right) \;+\; f_1\left(\frac{x-x_0}{x_1-x_0}\right)\left(\frac{x-x_2}{x_1-x_2}\right) \;+\; f_2\left(\frac{x-x_0}{x_2-x_0}\right)\left(\frac{x-x_1}{x_2-x_1}\right)$$

whose integral is

$$\int_{x_0}^{x_2} y(x)\,dx \;=\; \frac{h}{3}f_0 \;+\; \frac{4h}{3}f_1 \;+\; \frac{h}{3}f_2 \;\approx\; \int_{x_0}^{x_2} f(x)\,dx\ ,$$

giving the *extended Simpson's rule*

$$\int_a^b f(x)\,dx \;\approx\; h\left(\frac{1}{3}f_0+\frac{4}{3}f_1+\frac{2}{3}f_2+\ldots+\frac{4}{3}f_{2n-1}+\frac{1}{3}f_{2n}\right) - \frac{nh^5}{90}f^{(4)}(\xi)\ .$$

---

5. Of course we need not be limited to polynomials—trigonometric functions are also an acceptable way

Numerical quadrature rules employing equally-spaced abscissas, including the endpoints of the interval, are generically known as *closed* Newton-Cotes rules[6]. Of course it would be perfectly feasible to devise rules with equally-spaced abscissas that omit the endpoints—these are the *open* Newton-Cotes formulas. Recipes that include one or the other endpoint but not both are *mixed* Newton-Cotes rules. Open, and mixed Newton-Cotes rules are only of minor interest since the applications for which they are optimal occur but rarely—in fact I have never seen one.

Gaussian quadrature

Open, closed or mixed Newton-Cotes formulae represent the integral of a function as

$$\int_a^b f(x)\,dx \approx \sum_{k=k_0}^{k_0+n} f\Big(a+kh\Big)w_k$$

where the points are equally spaced, and the weights w_k are chosen so that polynomials of a given degree are integrated exactly (that is, so that the function is represented by a truncated sum of such polynomials). Since there are n free parameters w_k , we can in general fit a polynomial of order $n-1$ to the function. What if we relaxed the condition that the points be equally spaced? Then we would be moved to represent an integral by

$$\int_a^b dx\, f(x) \approx \sum_{k=1}^{n} f\Big(\xi_k\Big)w_k\ ,$$

where the n points ξ_k lie in the interval $[a, b]$. Now we have twice as many free parameters to work with, hence can fit a polynomial of order $2n-1$. It turns out to be convenient to transform the interval to $[-1, 1]$ *via*

$$x = \frac{a+b}{2} + \frac{b-a}{2}\,t\ .$$

Then to determine the parameters ξ_n and w_n for this interval we might think of solving a rather horrible set of polynomial equations. Fortunately there is a simpler method: consider integrals of the form

$$I = \int_{-1}^{1} dt\ \varphi(t)\Big(t-\xi_1\Big)\Big(t-\xi_2\Big)\cdots\Big(t-\xi_n\Big).$$

Manifestly, $I=0$ when $\varphi(t)$ is any polynomial of order $n-1$ (recall the formula is supposed to be exact for all polynomials of degree up to and including $2n-1$).

However, the polynomial of n'th degree that is orthogonal to all polynomials of lesser degree on the interval $[-1, 1]$ is the Legendre polynomial $P_n(t)$; that is, we may immediately identify the points

6. See, *e.g.*, R. Hamming, *Numerical Analysis for Scientists and Engineers* ()

ξ_k as the roots of $P_n(\xi) = 0$. Next, how do we calculate the weights w_k ? We note that for any function the approximate integral is

$$\int_{-1}^{1} dx\, f(x) \approx \sum_{k=1}^{n} f\left(\xi_k\right) w_k \overset{df}{=} \sum_{k=1}^{n} f_k\ w_k\ ;$$

now suppose we make the Lagrangian polynomial approximation to $f(x)$:

$$f(x) \approx \sum_{k=1}^{n} f_k\ \frac{P_n(x)}{(x-\xi_k)\,P'_n(\xi_k)}\ .$$

This is a polynomial of $n-1$'st degree that passes through all n ordinates f_k , hence the integral of it must be exact. That is, we can identify the weights as

$$w_k = \int_{-1}^{1} dx\, \frac{P_n(x)}{(x-\xi_k)\,P'_n(\xi_k)}$$

For, say, $n = 3$ we find the roots and weights (recall $P_3(x) = \frac{5}{2}x^3 - \frac{3}{2}x$)

$$x_{\pm 1} = \pm\sqrt{3/5} \qquad w_{\pm 1} = 5/9$$

$$x_0 = 0 \qquad w_0 = 8/9\ .$$

Singular integrals

Occasionally we must evaluate singular integrals numerically. One example is the Cauchy principal value integral defined by

$$\mathscr{P}\int_a^b dx\, \frac{f(x)}{x-x_0} \overset{df}{=} \lim_{\varepsilon \to 0^+} \left(\int_a^{x_0-\varepsilon} dx\, \frac{f(x)}{x-x_0} + \int_{x_0-\varepsilon}^{b} dx\, \frac{f(x)}{x-x_0} \right).$$

W. J. Thompson has suggested[7] numerically evaluating the Cauchy principal value integral by rewriting it in the form

$$\mathscr{P}\int_a^b = \int_a^{x_0-\Delta} + \int_{x_0+\Delta}^{b} + \mathscr{P}\int_{x_0-\Delta}^{x_0+\Delta}$$

where Δ is a convenient finite number. The principal value integral

$$\mathscr{P}\int_{x_0-\Delta}^{x_0+\Delta} dx\, \frac{f(x)}{x-x_0} \equiv \mathscr{P}\int_{-\Delta}^{+\Delta} dx\, \frac{f(x+x_0)}{x} = \mathscr{P}\int_{-\Delta}^{+\Delta} dx\, \frac{g(x)}{x}$$

only involves the odd part of $g(x)$ and hence may be evaluated *via* Taylor's series employing only odd derivatives.

---

7. *Computers in Physics* **12** (1998) 94.

As it is usually inconvenient to compute high-order derivatives by interpolation or explicitly, as Thompson advocates, we employ a simpler scheme based on even-order Gauss-Legendre quadrature. First scale out the parameter Δ :

$$I(\Delta) = \mathscr{P}\int_{-\Delta}^{+\Delta} dx\,\frac{g(x)}{x} = \mathscr{P}\int_{-1}^{+1} dx\,\frac{g(x\Delta)}{x}.$$

Then subtract $g(0)$ from the integrand to get

$$I(\Delta) = \int_{-1}^{1} dx\,\frac{g(x\Delta) - g(0)}{x} \approx \sum_{n=1}^{2N} \frac{w_n}{\xi_n}\, g\left(\xi_n \Delta\right)$$

where the ξ_n and w_n are respectively the abscissas and weights of the $\mathscr{O}(2N)$ Gauss-Legendre quadrature formula[8]. An even-order formula is preferable to an odd-order one because the abscissas do not include the point $x = 0$. For example, the results for the integral

$$\mathscr{P}\int_{-1}^{+1} dx\,\frac{e^x}{x} = 2.11450175075.. \approx \sum_{n=1}^{2N} \frac{w_n}{\xi_n}\,\exp\left(\xi_n\right) \overset{df}{=} S_{2N}$$

are given in the Table below:

Numerical Evaluation of Principal Value Integral

$2N$	S_{2N}
2	2.11297772844928
4	2.11450171810538
6	2.11450175075134

Similar ideas can be applied to other types of singular integral. For example, the integral

$$I = \int_{-\Delta}^{\Delta} dx\,\frac{f(x)}{|x|^{\alpha}} \equiv \frac{1}{2}\int_{-\Delta}^{\Delta} dx\,\frac{f(x) + f(-x)}{|x|^{\alpha}}$$

is integrable if $f(x) \neq 0$ and Re $\alpha < 1$.

Gaussian integration with weight functions

A final remark on Gaussian quadrature: sometimes we need to evaluate integrals of the form

$$I = \int_a^b dx\,\sigma(x)\; f(x)$$

where the function $\sigma(x)$ is a known "weight function". For example, integrals such as

$$\int_0^{\infty} dx\, e^{-x} f(x) \text{ and } \int_{-\infty}^{\infty} dx\, e^{-x^2} f(x)$$

8. *Handbook of Mathematical Functions*, ed. by M. Abramowitz and I. Stegun (National Bureau of Standards, Washington, DC, 1964) p. 916.

often arise in evaluating the amplitudes for certain quantum mechanical transitions. The same technique we employed to derive the sample points in Gauss-Legendre integration (that is, the zeros of Legendre polynomials) we see that a formula such as

$$\int_0^\infty dx\, e^{-x} f(x) \approx \sum_{k=1}^{n} w_k f\left(\xi_k\right)$$

dictates that the sample points be the zeros of the n'th–order Laguerre polynomial; whereas the appropriate abscissas for

$$\int_{-\infty}^{\infty} dx\, e^{-x^2} f(x) \approx \sum_{k=1}^{n} w_k f\left(\xi_k\right)$$

are the zeros of the n'th order Hermite polynomial. Other weight functions and other intervals lead to various forms of Gauss-Chebyshev integration.

A change of variable lets us relate one case to another. For example, suppose we want abscissas and weights for the weight function $\ln(1/x)$ on the interval $[0,1]$. Letting $x = e^{-u}$ we get

$$-\int_0^1 dx \ln x\ f(x) \equiv \int_0^\infty du\, e^{-u}\, u\, f\left(e^{-u}\right) \approx \sum_{k=1}^{n} w_k \xi_k\ f\left(e^{-\xi_k}\right) \equiv \sum_{k=1}^{n} s_k\ f\left(\lambda_k\right)$$

where ξ_k and w_k are the points and weights of Gauss-Laguerre integration; then we immediately identify the points and weights appropriate to logarithmic integration as

$$\lambda_k = e^{-\xi_k}$$
$$s_k = \xi_k\, w_k \ .$$

5. 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 $f(x)$ more densely where it varies rapidly than where it varies slowly. Algorithms that can do this are called ***adaptive***.
- Second, we evaluate $f(x)$ as few times as possible—the ideal would be never to discard a value of $f(x)$ once it has been evaluated[9].

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*[10] or *importance sampling*[11]. The difficulty of automating

9. However this may not be the most efficient way to proceed, as we shall see below.
10. J.M. Hammersley and D.C. Hanscomb, *Monte Carlo Methods* (Methuen, London, 1964).
11. J. W. Negele and H. Orland, *Quantum Many-Particle Systems* (Addison-Wesley Publishing Company,

advanced sampling techniques for general functions puts adaptive Monte Carlo techniques beyond the scope of these lectures.

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

Adaptive integration in one dimension

We now construct 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. To make the user interface as FORTRAN-like as possible, we invoke the integration function with several arguments:

```
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*[12].

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 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. The ancient Greek mathematician Euclid devised a rapid algorithm for finding the gcd[13] which can be expressed symbolically as

$$\gcd(m,n) = \begin{cases} m & \text{if } n = 0 \\ \gcd(n,\ m \bmod n) & \text{otherwise.} \end{cases}$$

That is, the problem of finding the gcd of m and n can be replaced by the problem of finding the gcd of two much smaller numbers. A Forth subroutine that does this is[14]

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

Reading, MA, 1988) p. 406.

12. See, *e.g.*, R. Sedgewick, *Algorithms* (Addison-Wesley Publishing Company, Reading, MA, 1983), p. 85.
13. R. Sedgewick, *op. cit.*, p. 11.
14. Forth does not permit a subroutine to call itself by name: 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.

Here is an example of **GCD** in action, using the debugger from Win32Forth to display the steps:

```
784 48 dbg gcd [2] 784 48
code  ?DUP           –> [3] 784 48 48
code  0>             –> [3] 784 48 -1
code  IF             –> [2] 784 48
code  TUCK           –> [3] 48 784 48
   :  MOD            –> [2] 48 16
   :  GCD            –> [2] 48 16
   code  ?DUP           –> [3] 48 16 16
   code  0>             –> [3] 48 16 -1
   code  IF             –> [2] 48 16
   code  TUCK           –> [3] 16 48 16
      :  MOD            –> [2] 16 0
      :  GCD            –> [2] 16 0
      code  ?DUP           –> [2] 16 0
      code  0>             –> [2] 16 0
      code  IF             –> [1] 16
      code  ;              –>  ok
```

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

Recursive programming possesses an undeserved reputation for slow execution, compared with nonrecursive equivalent programs[15]. 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.

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

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

expressed recursively in Forth as

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

```
: FIB              ( : n - - F[n] )
   DUP   0>  NOT
   IF    DROP   0   EXIT   THEN
   DUP   1  =
   IF             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   2DROP   EXIT   THEN
   DUP    1  =
   IF   DROP  NIP   EXIT   THEN
   1   DO     TUCK   +    LOOP    PLUCK   ;
```

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

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

where τ is the integer addition time. The solution is

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

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. In such cases the running time is given by

$$T_n \approx T_{n/2} + T_{n/2} + n = 2T_{n/2} + n$$

for which the solution is[16]

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

16. To see this, let $n=2^k$ and write $T_n/n=U_k$. Then $U_k=U_{k-1} + 1$.

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.
End of Digression

Adaptive integration can be expressed as a divide and conquer algorithm, hence recursion can simplify the program. In QuickBasic®, a language that permits recursion, we have the program:

```
DECLARE FUNCTION dummy! (x!)
DECLARE FUNCTION simpson! (a!, b!)
DECLARE FUNCTION integral! (a!, b!, e!)
' Main program PRINT integral(0, 1, .001)

FUNCTION dummy (x)
    dummy = x * SQR(x)
 END FUNCTION

FUNCTION integral (a, b, e)
    c = (a + b) / 2
    old.int = simpson(a, b)
    new.int = simpson(a, c) + simpson(c, b)
    IF ABS(old.int - new.int) < e THEN
        integral = (16 * new.int - old.int) / 15
    ELSE        integral = integral(a, c, e / 2) + integral(c, b, e / 2)
    END IF
END FUNCTION
```

One is not obliged to use Simpson's rule on the sub-intervals—any favorite algorithm will do.

A Forth version of this program, using 3 point Gauss-Legendre quadrature, is given at the end of this chapter. In the Forth version we replaced `)INTEGRAL` by `RECURSE` inside the word `)INTEGRAL`. As noted (in a footnote) above, Forth normally does not permit words to refer to themselves thus avoiding unintentional recursion. Hence a word being defined remains hidden from the dictionary search mechanism (compiler) until the definition is terminated by a concluding semicolon (`;`). The word `RECURSE` unhides the current name, and compiles its execution token[17] in the proper spot.

Disadvantages of recursion in adaptive integration

The main advantage of a recursive adaptive integration algorithm is its ease of programming. The recursive program is shorter than the non-recursive one. For any reasonable integrand, the stack depth grows only as the square of the logarithm of the finest subdivision, hence never gets too large.

17. An "execution token" can be an index into a table or an address pointing to actual code. In Forth the execution token is what the word `EXECUTE` uses to execute the corresponding subroutine.

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

- The recursive program discards values of the function, hence may make more function calls than a nonrecursive method.
- It would be hard to nest the function `)INTEGRAL` for multidimensional integrals.
- The recursive program can run out of stack space, with unpredictable results[18].

Several solutions to these problems suggest themselves:

- We can eliminate recursion from the algorithm, thereby maintaining control of memory usage.
- 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.

Adaptive integration without recursion

The chief reason to write a non-recursive program is to maintain control of memory. We should also like to minimize the number of points x_n in $[A, B]$ consistent with the desired precision, evaluating $f(x)$ once only at each x_n. This will be especially worthwhile when $f(x)$ is costly to evaluate.

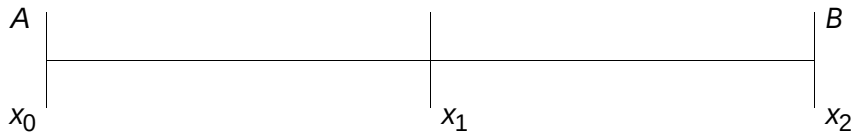
To minimize evaluations of $f(x)$, we save values $f_n = f(x_n)$ that can be re-used as we subdivide. The best place to save the f_n's is some kind of stack or array. Moreover, to make sure that a value of $f(x)$ computed at one mesh size is usable at all smaller meshes, we must subdivide into equal sub-intervals; that is, 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_n's be discarded because they cannot be re-used[19].

The simplest quadrature formula that satisfies these criteria is the *trapezoidal rule*. 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 | :

18. Actually the results are all too predictable: a machine crash, or in a protected-mode system such as MS Windows® or Linux, freezeup of a running task.
19. However, a Gaussian rule may actually lead to fewer total evaluations of $f(x)$ despite throwing points away, as we shall see below.

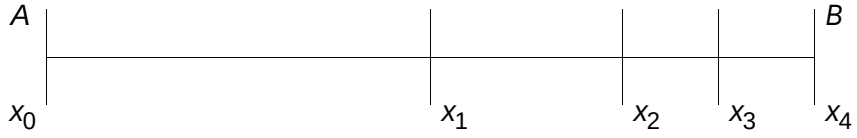
We now save (temporarily) I_0 and divide the interval in two, computing I'_0 and I_1 on the halves.



x	$f(x)$	sub-integral	error
x_0	f_0		
x_1	f_1	I'_0	$\varepsilon / 2$
x_2	f_2	I_1	$\varepsilon / 2$

This will be one fundamental operation in the algorithm.

We next compare $I'_0 + I_1$ with I_0. If the two integrals disagree, we subdivide again, as shown below (we imagine two steps have taken place):



x	$f(x)$	sub-integral	error
x_0	f_0		
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	$\varepsilon / 8$

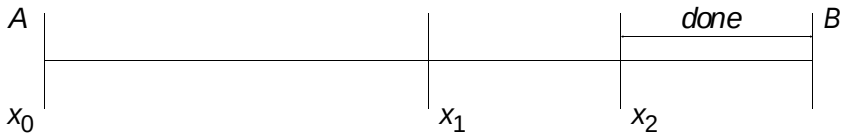
Now suppose the last two sub-integrals ($I_3 + I_2'$) in the last step agreed with their predecessor (I_2); we then accumulate the part computed so far, and begin again with the (leftward) remainder of the interval, as shown on the next page.

The program for non-recursive integration using the trapezoidal rule is given at the end of the chapter. The nonrecursive program requires substantially more code than the corresponding recursive version. This is the chief disadvantage of a nonrecursive method[20].

For completeness I have included a nonrecursive version of the adaptive 3-point Gaussian routine as the third program at the end of the chapter. It requires less memory than the trapezoidal rule because it does not save values of the function.

---

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



x	$f(x)$	sub-integral	error
x_0	f_0		
x_1	f_1	I_0	$\varepsilon / 2$
x_2	f_2	I_1	$\varepsilon / 4$

Comparison of trapezoidal and 3-point Gaussian methods

Although the trapezoidal rule was employed because, *ab initio*, it would seem to require fewer function evaluations (because it never discards any values) than—say—a Gaussian quadrature formula, it is useful to check whether this assumption is actually borne out in practice.

Here are the results of high-precision integration of (integrable) functions with end-point singularities:

```
\ Results of adaptive 3-point Gaussian:
   12 set-precision  ok
   use( fsqrt 0e 1e 1e-8 )integral 609  function calls ok
   fs. 6.66666666670E-1  ok

   : f1   FDUP  FSQRT  F*  ;  ( f: x – x^1.5)  ok
   use( f1 0e 1e 1e-8 )integral 165  function calls ok
   fs. 3.99999999994E-1  ok

\ Results with adaptive trapezoidal rule:
   use( fsqrt 0e 1e 1e-8 )integral 14377  function calls ok
   fs. 6.66666666667E-1  ok

   : f1   FDUP  FSQRT  F*  ;  ( f: x – x^1.5)  ok
   use( f1 0e 1e 1e-8 )integral 8925  function calls ok
   fs. 4.00000000000E-1  ok
```

In other words, the 3-point Gaussian adaptive algorithm uses 23.6 times fewer function calls (hence for a complicated function, is faster by that factor) to evaluate the integral of $x^{1/2}$ at its singularity than the trapezoidal rule, despite discarding points! The factor is even more severe—54.1—for the less singular function $x^{3/2}$.

Now let's really stress the algorithms by integrating $x^{-1/2}$ near the singularity:

```
\ Adaptive 3-point Gaussian
   : f2  fsqrt  1e fswap f/ ;  ok
   use( f2 1e-14 1e 1.e-6 )integral 5889  function calls ok
   fs. 1.99999979913E0  ok
```

```
\ Adaptive trapezoidal rule
   use( f2 1e-2 1e 1e-6 )integral 5329  function calls ok
   fs. 1.80000E0  ok
   use( f2 1e-4 1e 1e-6 )integral 22575  function calls ok
   fs. 1.98000E0  ok
   use( f2 1e-6 1e 1e-6 )integral 76683  function calls ok
   fs. 1.99800E0  ok
```

The first attempt to duplicate the 3-point Gaussian result (using the trapezoidal program with a very small lower limit) froze the program into what seemed like an endless loop (on a very fast machine). The best I was able to do was the three cases shown above, which are not terribly satisfactory.

The preceding tests offer several morals:

- Assumptions that seem too obvious to be questioned may turn out incorrect. This holds true in all aspects of existence, but especially when designing algorithms. It is therefore worthwhile to propose alternate hypotheses and test them honestly.
- Every aspect of a program must be tested. Design stringent tests that will make the program fail. We often learn more from failure than success.
- In these particular examples, it would be worthwhile to factor out the singular behavior and integrate it explicitly—one method uses Gaussian quadrature with singular weight functions.[21]

21. See, e.g., Abramowitz and Stegun, p. 888ff, for a discussion of Gauss-Chebyshev formulas.

```
\ Integration by recursion - - an illustration of the concept
\   Uses 3 point Gaussian integration

FALSE [IF]
  Examples:
        use( fsqrt 0e 1e 1e-4 )integral fs. 6.66666744641E-1  ok
        use( fsqrt 0e 1e 1e-6 )integral fs. 6.66666670150E-1  ok
        : f1     FDUP FSQRT F*  ;  ok
        use( f1 0e 1e 1e-4 )integral fs. 3.99994557189E-1  ok
        use( f1 0e 2e 1e-6 )integral fs. 2.26274169632E0  ok
[THEN]

MARKER  -rint
: undefined   BL WORD  FIND  NIP  0=  ;
\ vectoring: for using function names as arguments, or fwd recursion
undefined use(  [IF]
    : use(      '       \ state-smart ' for syntactic sugar
        STATE @  IF  POSTPONE LITERAL  THEN  ;  IMMEDIATE
    ' NOOP  CONSTANT  'noop
    : v:   CREATE  'noop  ,  DOES> PERFORM  ;   \ create dummy def'n
    : 'dfa   ' BODY  ;                          ( – data field address)
    : defines    'dfa   STATE @
             IF   POSTPONE  LITERAL    POSTPONE  !
             ELSE   !   THEN  ;  IMMEDIATE
[THEN]
\ end vectoring
\ define FVALUEs
undefined FVALUE    [IF]
    : FVALUE    CREATE   0e   F,  DOES>  F@  ;
    : FTO       'dfa  STATE @
            IF      POSTPONE  LITERAL  POSTPONE  F!
            ELSE    F!   THEN  ;  IMMEDIATE
[THEN]
\ points and weights for 3 point Gauss-Legendre integration
    8e  9e  F/          FCONSTANT  w0
    5e  9e  F/          FCONSTANT  w1
    9e  15e  F/ FSQRT   FCONSTANT  x1
    x1  FNEGATE         FCONSTANT -x1

v: fdummy
FVALUE  dx FVALUE  <x>

: scale    ( f: xa xb - - )
    FOVER    F-    F2/      ( f: [xb-xa]/2)
    FSWAP  FOVER  F+        ( f: [xb-xa]/2  [xa+xb]/2 )
    FTO  <x>    FTO  dx  ;

VARIABLE Ntimes             \ count of function evaluations
: )int  ( f: xa xb – gauss_int)    \ 3 point Gauss-Legendre
    scale
                 <x>      fdummy      w0  F*    \ f(0) * w0
    x1   dx  F*  <x>  F+  fdummy                \ f(x1)
    -x1  dx  F*  <x>  F+  fdummy  F+  w1  F*    \ [f(-x1)+f(x1)] * w1
    F+   dx  F*    3 Ntimes +! ;
```

```
FVALUE  oldI     FVALUE  newI     FVALUE  finI
FVALUE  xa       FVALUE  xb       FVALUE  xc       FVALUE  err

: juggle   ( f: – xa xc err/2  xc xb err/2 )
       xa xc err F2/   xc FOVER  xb  FSWAP  ;

: storeI   finI  F+  FTO  finI  ;

: adaptive      ( f: xa xb err – )
    FTO err   FTO xb   FTO xa
    xa  xb  F+  F2/    FTO xc
    xa xb )int                      FTO  oldI
    xa xc )int   xc xb )int  F+     FTO  newI
    newI  oldI  F-  FDUP  FABS  err  F<
    IF   63e F/  newI  F+   storeI
    ELSE  FDROP   juggle  RECURSE RECURSE  THEN  ;

: )integral     ( xt - - )  ( f: xa xb err - - integral)
    defines fdummy          \ pass function xt to fdummy
    0e  FTO  finI           \ initialize integral
    0 Ntimes !              \ initialize count
    adaptive    finI
    Ntimes @  .  ."  function calls" ;
```

```
\ Adaptive integration using trapezoidal rule
\   with Richardson extrapolation
\ Integrate a real function from xa to xb
\
\     (c) Copyright 1998  Julian V. Noble.          \
\       Permission is granted by the author to      \
\       use this software for any application pro-  \
\       vided this copyright notice is preserved.   \
\
\ Usage:  use( fn.name xa xb err )integral
\ Examples:
\       use( fsqrt  0e  1e  1e-3 )integral fs. 6.666659e-1 ok
\       use( fsqrt  0e  2e  1e-4 )integral fs. 1.885618e0 ok
\       : f1     FDUP FSQRT F*  ;  ok
\       use( f1  0e  1e  1e-3 )integral fs. 4.00000017830E-1  ok
\       use( f1  0e  2e  1e-4 )integral fs. 2.26274170358E0  ok
\ Programmed by J.V. Noble (from "Scientific FORTH" by JVN)
\ ANS Standard Program  — version of  October 15th, 1998
\ This is an ANS Forth program requiring:
\      The FLOAT and FLOAT EXT word sets
\ Environmental dependencies:
\       Assumes independent floating point stack
\       Uses a FORmula TRANslator for clarity
MARKER  -int
\ Non STANDARD words:
: undefined   BL WORD  FIND  NIP  0=  ;
undefined f"        [IF]  include ftran110.f        [THEN]
undefined sf       [IF]  : sf     SD  DF  ;     [THEN]
undefined f0.0      [IF]  0.0E0  FCONSTANT  f0.0    [THEN]
undefined float_len [IF]  1 FLOATS  CONSTANT  float_len [THEN]
undefined 1array [IF]
    : long   ;
    : 1array    ( len data_size —)     CREATE  2DUP  ,  ,  *  ALLOT   ;
    : _len      ( base_addr — len)  \ determine length of an array
            CELL+  @  ;    : }         ( base_adr indx — adr[indx] )
            OVER  _len  OVER  <=  OVER  0<  OR  ABORT" Index out of range"
            OVER  @   *  +  CELL+  CELL+  ;
[THEN]
\      1array  create a a one-dimensional array
\           as in 20  1 FLOATS  1array A{
\      }    dereference a one-dimensional array
\           as in  A{ I }  ( base.adr — base.adr + offset )
\
\      v:   define a function vector
\      defines  (IMMEDIATE)  set a vector, as in
\            v: dummy  ;
\            : test    ( xt — ) defines dummy
\                   dummy  ;
\            3 5 ' * test .  15 ok
\      use(     (IMMEDIATE) get the xt of a word
\
\      the vectoring words are included in ftran110.f
```

```
\ Data structures
FVARIABLE  c43 4 SF  3 SF  F/   c43  F!
20 CONSTANT Nmax
Nmax long float_len    1array x{
Nmax long float_len    1array E{
Nmax long float_len    1array f{
Nmax long float_len    1array I{
0 VALUE  N
0 VALUE  NN
FVARIABLE  oldI
FVARIABLE  finI
FVARIABLE  deltaI

\ Begin program
: )int  ( n –)  TO NN              \ trapezoidal rule
   f" ( f{nn} + f{nn_1-} ) * ( x{nn} - x{nn_1-} )  “  F2/
   I{ nn 1- } F!  ;

v: dummy                                  \ dummy function name
VARIABLE Ntimes
: initialize  ( xt –)  ( f: xa xb eps – integral)
     defines  dummy
     1 TO N     E{ 0 } F!   X{ 1 } F!   X{ 0 } F!
     f" f{0} = dummy( x{0} ) “
     f” f{1} = dummy( x{1} ) “
     1 )INT     f0.0  finI F!
     2 Ntimes !  ;

: check.n
       N  [ Nmax 1- ] LITERAL         ABORT" Too many subdivisions!"  ;

: E/2   E{ N  1- }  DUP   F@   F2/    F! ;

: }down    ( adr n –)        OVER @  R   }   DUP   R@ +   R   MOVE  ;

: move.down    E{ N  1- }down
               x{ N     }down
               f{ N     }down  ;

: x’   f" x{N} + x{N_1-}  “  F2/  x{ N } F!
       f" f{N} = dummy( x{N} )  “
       1 Ntimes +!  ;

: N+1   N 1+   TO N  ; : N-2   N 2 -  TO N  ;

: subdivide
     check.n     E/2   move.down
     f" oldI = I{N_1-} “
     x’   N )int   N 1+ )int    ;

: converged?   ( f: –)  ( – f)
        f" I{N} + I{N_1-} - oldI “
        FDUP   deltaI  F!   FABS
        E{ N 1- } F@   F2*    F  ;
```

```
: interpolate    f" finI = deltaI * c43 + oldI + finI “  ;

: )integral    ( f: xa xb err — I[xa,xb]) ( xt —)
     initialize
     BEGIN   N 0>   WHILE
        subdivide
        converged?    N+1        IF    interpolate  N-2    THEN
     REPEAT   finI  F@     Ntimes @  .  ."  function calls" ;
```

```
\ Adaptive integration using 3 point Gauss-Legendre rule
\   with Richardson extrapolation
\
\     (c) Copyright 1998  Julian V. Noble.          \
\       Permission is granted by the author to      \
\       use this software for any application pro-  \
\       vided this copyright notice is preserved.   \
\
\ This is an ANS Forth program requiring:
\      The FLOAT and FLOAT EXT word sets
\ Environmental dependencies:
\       Assumes independent floating point stack
\       Uses a FORmula TRANslator for clarity
\ Usage:  use( fn.name xa xb err )integral
\ Examples:
\   12 set-precision  ok
\   use( fsqrt 0e 1e 1e-8 )integral cr fs. 609  function calls
\   6.66666666670E-1  ok
\
\   : x^1.5   fdup  fsqrt  f*  ;  ok
\   use( x^1.5  0e 1e 1e-8 )integral cr fs. 165  function calls
\   3.99999999994E-1  ok
MARKER  -int
needs ftran110.f
needs arrays.f

FALSE [IF]
    Non STANDARD words:
       1array       create a a one-dimensional array
                    Ex: 20  1 FLOATS  1array A{
       }            dereference a one-dimensional array
                    Ex: A{ I }  ( base_adr — base_adr + offset )

       Vectoring words included in ftran110.f
       v:           define a function vector
                    Ex:   v: dummy
       defines      set a vector
                    Ex:   ' *  defines  dummy  7 2 dummy .  14 ok
                          : test    ( xt — ) defines dummy  dummy  ;
                          3 5 ' * test .  15 ok
       use(         get the xt of a word
[THEN]

\ points and weights for 3 point Gauss-Legendre integration
    8e  9e   F/         FVARIABLE  w0   w0 F!
    5e  9e   F/         FVARIABLE  w1   w1 F!
    3e  5e   F/ FSQRT   FCONSTANT  x1

\ Data structures
64e  63e  F/  FVARIABLE Cinterp   Cinterp F!
20 CONSTANT Nmax
1 FLOATS  CONSTANT  float_len
```

```
Nmax long float_len    1array x{
Nmax long float_len    1array E{
Nmax long float_len    1array I{

0 VALUE  N
0 VALUE  nn
FVARIABLE  oldI
FVARIABLE  finI
FVARIABLE  deltaI
FVARIABLE dx
FVARIABLE dxi
FVARIABLE xi

\ Begin program
: scale    ( f: xa xb – )
    FOVER    F-    F2/      ( f: [xb-xa]/2)
    FSWAP  FOVER  F+        ( f: [xb-xa]/2  [xa+xb]/2 )
    xi F!    FDUP  dx F!
    x1 F*  dxi F!  ;

VARIABLE Ntimes             \ count of function evaluations

v: fdummy
: )int  ( n – )     \ 3 point Gauss-Legendre
    TO nn    x{ nn 1- }  F@  x{ nn }  F@   scale
    f" I{nn_1-}=dx*(w0*fdummy(xi)+w1*(fdummy(xi+dxi)+fdummy(xi-dxi))) “
    3 Ntimes +! ;

: initialize  ( xt –)  ( f: xa xb eps – integral)
     defines  fdummy
     1 TO N
     E{ 0 } F!   X{ 1 } F!   X{ 0 } F!
     0 Ntimes !
     1 )int
     0e  finI F! ;

: check.n
       N  [ Nmax 1- ] LITERAL  >
       ABORT" Too many subdivisions!"  ;

: E/2   E{ N  1- }  DUP   F@   F2/    F! ;

: }down    ( adr n –)
        OVER @  R   }   DUP   R@ +   R   MOVE  ;

: move.down    E{ N  1- }down
               x{ N     }down ;

: N+1   N 1+   TO N  ;
: N-2   N 2 -  TO N  ;
```

```
: subdivide
        check.n     E/2   move.down
        f" oldI = I{N_1-} “
        f” x{N} + x{N_1-} “  F2/  x{ N } F!
        N )int   N 1+ )int    ;

: converged?   ( f: –)  ( – f)
        f" I{N} + I{N_1-} - oldI “
        FDUP   deltaI  F!   FABS
        E{ N 1- } F@   F2*    F  ;

: interpolate    f" finI = deltaI * Cinterp + oldI + finI “  ;

: )integral    ( f: xa xb err – I[xa,xb]) ( xt –)
     initialize
     BEGIN   N 0>   WHILE
        subdivide
        converged?    N+1
        IF    interpolate  N-2    THEN
     REPEAT   finI  F@
     Ntimes @  .  ."  function calls" ;
```