Chapter

4
Numerical integration

Contents

Many scientific computation problems involve evaluation of definite integrals. This subject requires some sophisticated mathematics, but it's easy to describe. The required programming techniques are fairly straightforward. Thus, numerical integration is an appropriate topic to occur early in a book on scientific applications programming.

Considerable space is devoted to discussion of the fundamental mathematical concepts involved. The author believes this is appropriate because the major programming problems arise while implementing that part of the mathematics. When the analysis of integrals and numerical integration techniques requires deeper mathematics, references are cited. Only the most familiar numerical integration techniques are employed here, because more advanced ones involve no new programming methods appropriate to introduce in this chapter. For those integration techniques you should consult a numerical analysis text like Reference [3] or [10].

The chapter begins with a section on the definite integral concept, and introduces the properties of integrals required to understand the integration techniques discussed later. Section 4.2, on Riemann sums, continues this foundation, but forms a bridge to the practical: it introduces the first routine for approximating definite integrals, and gives a complete analysis of the resulting error. The Riemann sum integration routine is the first in this book to use function pointers as parameters to other functions. Section 4.2 also introduces the Richardson extrapolation technique. Calculating integrals with Riemann sums, even enhanced by Richardson extrapolation, is too inaccurate for much practical use, so the more sophisticated trapezoid and Simpson's rule approximations are introduced in Section 4.3. The formulas for these methods are derived here, but for the error analyses, you're referred to the literature. A single example is used to compare all three methods, and show the effect of Richardson extrapolation.

4.1 Definite integrals

Concepts

Definite integrals and area
Theorems about integrals
Fundamental theorem of calculus

The definite integral

$$J = \int_a^b f$$

of a function f over an interval $[a, b]$ is the area between the horizontal axis and the graph of f. Areas P above and N below the axis are considered positive and negative, so that $J = P - N$, as in Figure 4.1.1. For example, the following integral represents the semicircular area in Figure 4.1.2:

$$f(x) = \sqrt{1 - x^2}$$

$$\frac{\pi}{2} = \int_{-1}^{1} f = \int_{-1}^{1} \sqrt{1 - x^2}\, d$$

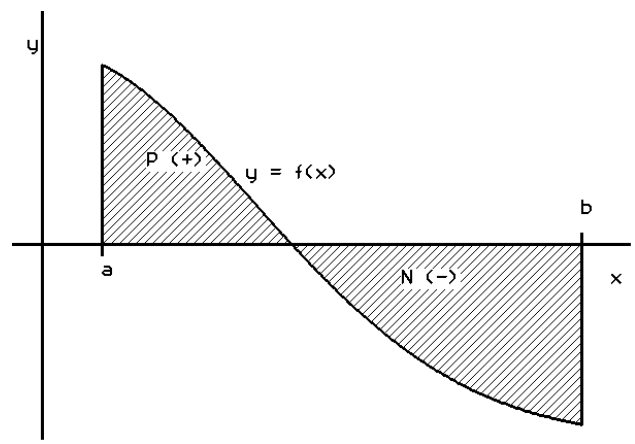


Figure 4.1.1 A definite integral

The dx symbol signifies that x is the argument of the function defined by the formula.

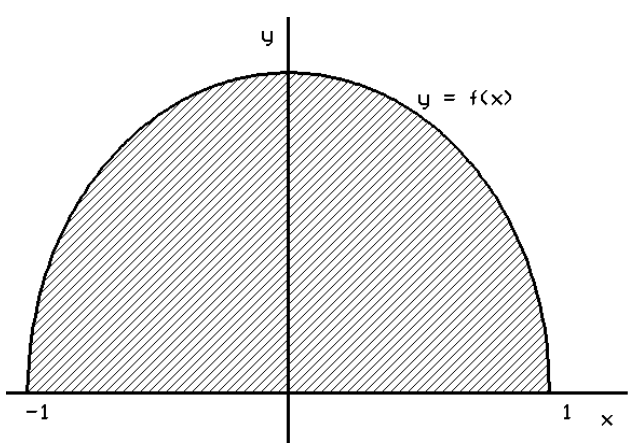


Figure 4.1.2 Area of a semicircle

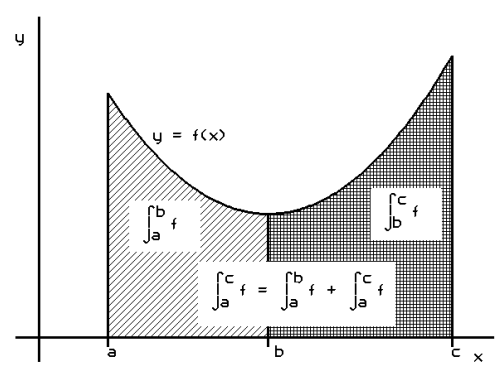


Figure 4.1.3 Integrals add like areas.

From this description, some fundamental properties of the integral are apparent. First, if $a < b < c$, then

$$\int_a^c f = \int_a^b f + \int_b^c f .$$

(See Figure 4.1.3.) Second, if $f(x) \le g(x)$ whenever $a \le x \le b$, then

$$\int_a^b f \le \int_a^b g .$$

(See Figure 4.1.4.)

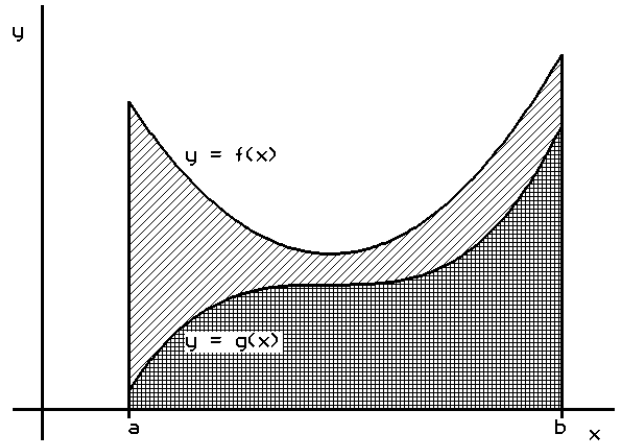


Figure 4.1.4 Order relation for integrals

The *linearity* properties of integrals stem from *Cavalieri's area principle*. Consider cross sections of two areas A and B cut by a pencil of parallel lines. Suppose the lengths of corresponding sections of A and B are proportional, with ratio c. Then c is also the ratio of areas A and B. (See Figure 4.1.5.) This shows that

$$\int_a^b (cf) = c\int_a^b f .$$

In particular, for $c = -1$, you get

$$\int_a^b (-f) = -\int_a^b f .$$

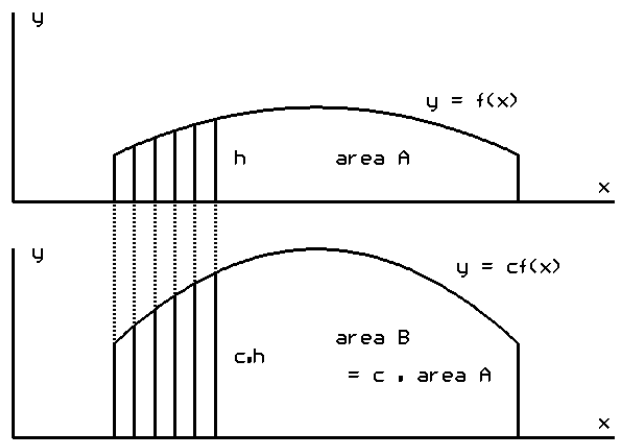


Figure 4.1.5 Cavalieri's principle

Cavalieri's principle also underlies the equation

$$\int_a^b f + \int_a^b g = \int_a^b (f+g) ,$$

as follows. In Figure 4.1.6, each section from x to P, representing the integral of f, is equal to the corresponding section from Q to R. By Cavalieri's principle, the area under the graph of f equals that between the graphs of g and $f + g$. That is, the integral of f is the difference of the integrals of $f + g$ and g.

You can draw some more graphs to verify that the properties just illustrated hold even when f and g have some negative values.

These properties have a consequence that plays an important role in the next section:

$$\left| \int_a^b f \right| \le \int_b^a |f| \quad .$$

To verify this, note that for each x, $-|f(x)| \le f(x) \le |f(x)|$, hence

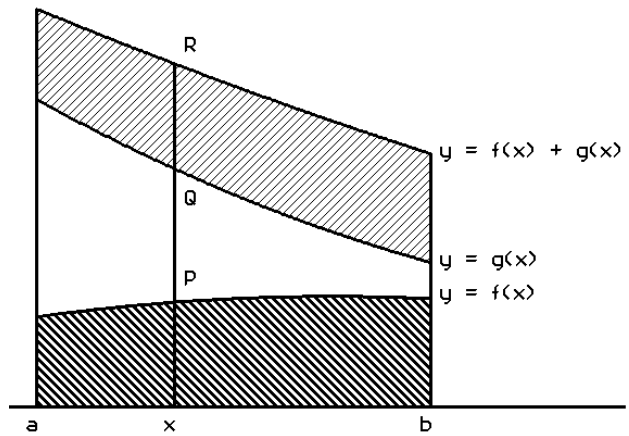


Figure 4.1.6 Integral of the sum of two functions

$$-\int_a^b |f(x)|\,dx \le \int_a^b f(x)\,dx \le \int_a^b |f(x)|\,dx$$

$$-\int_a^b |f| \le \int_a^b f \le \int_a^b |f| \quad .$$

The two major calculus operations—differentiation and integration—are in a sense inverses. This is stated by the two equations of the *Fundamental Theorem of Calculus*:

$$\frac{d}{dx}\int_a^x f = f(x) \qquad\qquad \int_a^b g' = g(b) - g(a) \quad .$$

The second equation provides a powerful technique for evaluating integrals: to determine

$$J = \int_a^b f \ ,$$

find a function g such that $f = g'$, then compute $J = g(b) - g(a)$. Since so many problems from mathematics and science can be formulated as questions about definite integrals like J, the Fundamental Theorem provides a powerful tool, as long as you can express the corresponding functions f as derivatives g'.

If necessary, you can consult a real analysis text—for example, Bartle (Reference [1]) —to find out under what conditions the Fundamental Theorem holds, as well as various properties of integrals like those just discussed. Not every function has an integral; characterizing those that do is a task for advanced mathematics.

Using the Fundamental Theorem to calculate integrals is often not easy, and sometimes impossible. For example, unless you've studied integration techniques rather deeply, you'll have trouble finding a function g whose derivative is the function f used earlier in representing $\pi/2$ as an integral:

$$g'(x) = f(x) = \sqrt{1 - x^2} .$$

An integral table will lead you to select

$$g(x) = \sin^{-1}x + x\sqrt{1 - x^2} .$$

This formula's complexity should suggest the true situation: given a function f you often cannot find *any* function g, defined by a formula involving functions familiar from elementary mathematics, whose derivative is f. Therefore, it's necessary to develop other methods for calculating integrals. The next two sections are devoted to methods called *numerical* integration techniques, that yield approximations as accurate as you wish, provided you have sufficient computing resources.

4.2 Riemann sums

Concepts

Using Riemann sums to approximate integrals
Using Riemann sums to define integrals
Function `Riemann`; example
Error analysis for Riemann sum approximation
Richardson extrapolation; example

One way to approximate a definite integral

$$J = \int_a^b f$$

is to subdivide the interval $[a, b]$ into n subintervals by inserting subdivision points $x_1 \ldots x_{n-1}$ with $a = x_0 < x_1 < \ldots < x_{n-1} < x_n = b$, then consider n rectangles with bases $h_k = x_{k+1} - x_k$ and altitudes $f(x_k)$ for $k = 0 \ldots n - 1$. Normally the step size is constant: $h_k = h = (b - a)/n$ for each k. The rectangle with base $[x_k, x_{k+1}]$ has area $hf(x_k)$

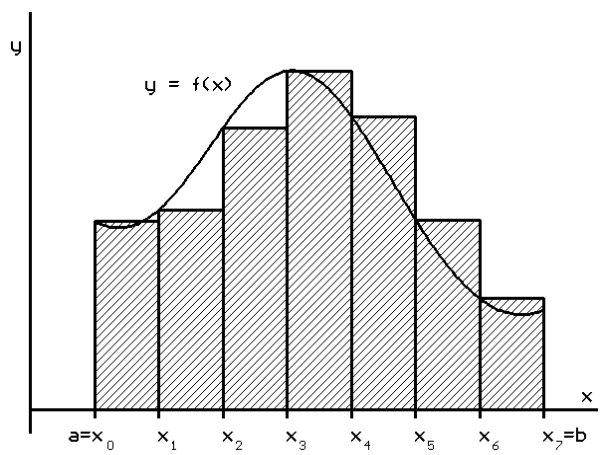


Figure 4.2.1 Riemann sum approximation

(positive or negative depending on its position above or below the axis). The sum of the rectangular areas is approximately equal to the integral J:

$$\int_a^b f \approx \sum_{k=0}^{n-1} hf(x_k) \quad .$$

As you can see from Figure 4.2.1, the approximation should become more accurate as the step size decreases.

A sum of values of a function, multiplied by corresponding interval lengths h, is called a *Riemann* sum. Later in this section, a C++ function is introduced that uses Riemann sums to approximate definite integrals. Before proceeding to that practical matter, briefly consider applying a slightly more general technique to a theoretical question. Recall that the definite integral was introduced in Section 4.1 as the area between a curved graph and the horizontal axis. Nineteenth Century mathematical research into some delicate integration problems yielded some perplexing, sometimes apparently contradictory results. Mathematicians discovered the problem: they didn't properly understand the concepts *curve* and *area*. To clarify the situation G.F.B. Riemann reformulated the notion of definite integral, making it independent of the concept of area. Then you can analyze the area of a curved region by using the integral—whose theory is validated independently—as a tool. Riemann considered the integral of a function f as a limit of sums of values $f(x_k)$ multiplied by corresponding step sizes h_k. That is, he *defined* the integral as a limit of Riemann sums. (For this purpose, you mustn't restrict the step sizes h_k to a constant value h, and you must consider $f(x_k)$ values with x_k selected anywhere in its subinterval, not just at the left end. That extra generality isn't necessary for the approximation methods discussed here.) Riemann's technique is usually mentioned in calculus courses and considered in complete detail in analysis courses. The Riemann sum theory doesn't settle *all* area questions. Mathematicians are still developing new approaches, and some of these are yielding new computational methods.

Figure 4.2.2 shows a function `Riemann` from module `Integral` that will return a Riemann sum approximation to a definite integral

$$\int_a^b f$$

given the lower and upper limits `a` and `b`, a pointer to function `f`, and the number `n` of subdivisions. Like several other integration functions in this module, its parameter `f` has type `double(double):` pointer to a function `f` with prototype `double f(double x)`.

Program `Riemann` in Figure 4.2.3 demonstrates the Riemann sum technique, and tests function `Riemann`, by computing approximations to

$$\int_0^1 \cos x \, dx$$

for various step sizes. The displayed output suggests that the error is proportional to the step size h: each tenfold cut in h decreases the error about tenfold. The rest of this section is devoted to explaining why that is so.

```
double Riemann(double   a,              //                 ⎮b
               double   b,              // Approximate     ⎮  f  by a
               double   f(double),      //                 ⎮a
               unsigned n) {            // Riemann sum with  n
if (a == b) return 0;                   // subdivisions.
if (n <= 0) n = 1;                      // Use at least  1  step.
double h = (b - a)/n;                   // Step size  h .
double Sum = 0;
for (unsigned k = 0; k < n; ++k) {      //       n-1
  double xk = a + k*h;                  // Sum =  Σ  f(xk)
  Sum += f(xk); }                       //       k=0       .
return h*Sum; }
```

Figure 4.2.2 Function **Riemann**

Error analysis

You can use calculus to derive a close relationship between the Riemann sum error and the step size. In practice, when you approximate an integral, you must refer to such a relationship to determine the step size necessary to attain the accuracy appropriate to your application.

The theoretical tool for the error analysis is the *Mean Value Theorem* of differential calculus: if a function f satisfies certain smoothness conditions, then between any two arguments x_1 and x_2 there's a point ξ such that

$$f(x_1) - f(x_2) = (x_1 - x_2)\, f'(\xi) \ .$$

If you know an upper bound M_1 for $|f'(\xi)|$, then you can conclude that

$$|f(x_1) - f(x_2)| \le |x_1 - x_2|\, M_1 \ .$$

```
//*************************************************************
// Riemann.CPP     Testing function  Riemann

#include   "Scalar.H"
#include "Integral.H"
#include     <Math.H>

void main() {
  clrscr();
  cout << "                  ∫1                              \n"
          "Approximating     ∫ cos x dx  by Riemann sums  \n"
          "                  0                          \n\n"
          "No. of sub-     Approxi-     True           \n"
          "divisions       mation       value     Error\n";
  for (unsigned n = 1; n <= 10000; n *= 10) {
    double  Approximation = Riemann(0,1,cos,n);
    double  TrueValue     = sin(1);
    double  Error         = Approximation - TrueValue;
    cout << setw(6)  << n << setprecision(5)
         << setw(17) << Approximation
         << setw(12) << TrueValue
         << "   ";      Show(Error);          cout << "\n"; }
  Inspect; }
```

Output

```
                 ∫1
Approximating    ∫ cos x dx  by Riemann sums
                 0

No. of sub-     Approxi-     True
divisions       mation       value     Error
     1                1     0.84147    0.16
    10          0.86375     0.84147    0.022
   100          0.84376     0.84147    0.0023
  1000          0.84170     0.84147    0.00023
 10000          0.84149     0.84147    2.3e-05
```

Figure 4.2.3 Riemann sum approximations

This inequality is used to estimate the error in Riemann sum approximation of the integral

$$J = \int_a^b f \ .$$

Introduce subdivision points $x_1 \ldots x_{n-1}$ with $a = x_0 < x_1 < \ldots < x_{n-1} < x_n = b$ and $x_{k+1} - x_k = h = (b-a)/n$ for $k = 0 \ldots n-1$. Then for each k,

$$\left| \int_{x_k}^{x_{k+1}} f - hf(x_k) \right| = \left| \int_{x_k}^{x_{k+1}} f(x)\,dx - \int_{x_k}^{x_{k+1}} f(x_k)\,dx \right|$$

$$= \left| \int_{x_k}^{x_{k+1}} \left[f(x) - f(x_k) \right] dx \right| \le \int_{x_k}^{x_{k+1}} \left| f(x) - f(x_k) \right| dx$$

$$\le \int_{x_k}^{x_{k+1}} M_1 \left[x - x_k \right] dx = M_1 \int_{x_k}^{x_{k+1}} \left[x - x_k \right] dx$$

$$= \frac{M_1}{2} \left[x - x_k \right]^2 \Big|_{x_k}^{x_{k+1}} = \frac{M_1 h^2}{2} .$$

This gives the error bound

$$\left| \int_{x_k}^{x_{k+1}} f - hf(x_k) \right| \le \frac{M_1 h^2}{2}$$

for approximating the integral over a single subinterval by a single term of the Riemann sum—the area of one rectangle. Now you can determine an error bound for the integral J over the whole interval:

$$\left| J - \sum_{k=0}^{n-1} hf(x_k) \right| = \left| \sum_{k=0}^{n-1} \int_{x_k}^{x_{k+1}} f - \sum_{k=0}^{n-1} hf(x_k) \right|$$

$$= \left| \sum_{k=0}^{n-1} \left[\int_{x_k}^{x_{k+1}} f - hf(x_k) \right] \right| \le \sum_{k=0}^{n-1} \left| \int_{x_k}^{x_{k+1}} f - hf(x_k) \right|$$

$$\le \sum_{k=0}^{n-1} \frac{M_1 h^2}{2} = \frac{n M_1 h^2}{2} = \frac{(b-a) M_1 h}{2} .$$

(The first of the two inequalities here is an instance of the theorem that the absolute value of the sum of some terms never exceeds the sum of their individual absolute values.) Using the equation $h = (b-a)/n$, you can rewrite the above result as follows: the error in approximating the integral J by a Riemann sum is

$$\left| J - \sum_{k=0}^{n-1} hf(x_k) \right| \le \frac{(b-a) M_1 h}{2} = \frac{(b-a)^2 M_1}{2n} .$$

The middle term explains the behavior of the errors tabulated for the example in Figure 4.2.3: the error is *bounded* by a linear function of h.

The right hand term of the last inequality shows how to select the number n of subdivisions for a required accuracy. To insure that the error doesn't exceed the largest error ε tolerable by your application, you should have

$$\frac{(b-a)^2 M_1}{2n} \le \varepsilon \qquad\qquad n \ge \frac{(b-a)^2 M_1}{2\varepsilon} .$$

Richardson extrapolation

Often in practice you don't know the bound M_1 needed to use the previous inequality for determining the step size. It's common in such situations to compute approximations $A(h)$ to the integral J for a decreasing sequence of step sizes h, and terminate when two successive approximations differ by less than your tolerance ε. Of course, you only know then that your approximations are close to each other—not necessarily to the true value of J.

Suppose you have good evidence that the error is approximately proportional to h^p for some power p:

$$A(h) - J \approx ch^p$$

for some constant c. (For Riemann sum approximation, $p = 1$.) In that case, you can make better use of the successive approximations $A(h)$, as follows. The approximation for the next step size h/d satisfies a similar equation:

$$A\left(\frac{h}{d}\right) - J \approx c\left(\frac{h}{d}\right)^p .$$

Solve these approximate two equations for c, then for J, to get

$$J \approx \frac{d^p A\left(\frac{h}{d}\right) - A(h)}{d^p - 1} .$$

That should be a better approximation. It's called the *Richardson extrapolation* from $A(h)$ and $A(h/d)$.

The Figure 4.2.3 program `Riemann.CPP` has been modified to demonstrate this technique. You'll find the resulting program `Richards.CPP` on the accompanying diskette. Alongside the Riemann sum approximations $A(h)$ and their errors, it displays the Richardson extrapolations

$$\frac{10A\left(\frac{h}{10}\right) - A(h)}{9}$$

and their errors. The output is shown in Figure 4.2.4. You can see that the extrapolations are far better approximations than the original Riemann sums. In fact the error with Richardson extrapolation seems proportional to h^2: a tenfold cut in step size produces a hundredfold decrease in the error. That's predictable: Richardson extrapolation generally increases by at least one the power of h to which the errors in a sequence of successive approximations are approximately proportional. (See Reference [10, Section 7.1D].)

Approximating $\int_0^1 \cos x \, dx$ by Riemann sums

Subdivisions	True value	Riemann approximation	Riemann error	Richardson extrapolation	Richardson error
1	0.841470985	1	0.16		
10	0.841470985	0.863754527	0.022	0.848616141	0.0071
100	0.841470985	0.843762461	0.0023	0.841541120	7.0e-05
1000	0.841470985	0.841700764	0.00023	0.841471686	7.0e-07
10000	0.841470985	0.841493969	2.3e-05	0.841470992	7.0e-09

Figure 4.2.4 Richardson extrapolation

4.3 Trapezoid and Simpson rules

Concepts

Trapezoid rule
Function `Trapezoid`; example
Simpson's rule
Function `Simpson`; example
Relationship between the trapezoid and Simpson rules and Richardson extrapolation

Section 4.2 showed that, with the aid of Richardson extrapolation, Riemann sums provide rather poor approximations to definite integrals. Even with the simple integral considered there, an attempt to attain more than five digit accuracy would require more than ten thousand steps. The time required would increase, and the error accumulated from rounding off each term in the sums would begin to be significant. New ideas are required.

Trapezoid rule

The *trapezoid rule* is a simple elaboration of the Riemann sum method that is more accurate. As before, introduce equally spaced subdivision points $x_1 \ldots x_{n-1}$ with

$$a = x_0 < x_1 < \ldots < x_{n-1} < x_n = b$$

and $x_{k+1} - x_k = h = (b - a)/n$ for each $k < n$. Instead of approximating an integral

$$\int_{x_k}^{x_{k+1}} f$$

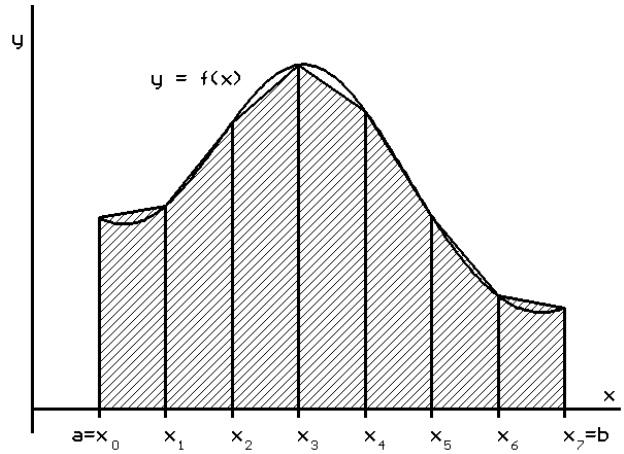


Figure 4.3.1 Trapezoid rule approximation

over a single subinterval by the area of a rectangle, use a trapezoid with bases $f(x_k)$ and $f(x_{k+1})$ and altitude h. Sum these trapezoids, as in Figure 4.3.1, to approximate the integral over the entire interval. The area of a single trapezoid is

$$\frac{h}{2} f(x_k) + \frac{h}{2} f(x_{k+1}) \ ,$$

hence

$$\int_a^b f \approx \left[\frac{h}{2} f(x_0) + \frac{h}{2} f(x_1) \right] + \left[\frac{h}{2} f(x_1) + \frac{h}{2} f(x_2) \right] + \cdots$$
$$+ \left[\frac{h}{2} f(x_{n-2}) + \frac{h}{2} f(x_{n-1}) \right] + \left[\frac{h}{2} f(x_{n-1}) + \frac{h}{2} f(x_n) \right] .$$

For each bracketed term except the last, the second part matches the first part of the next, so you can simplify the expression for the sum:

$$\int_a^b f \approx \frac{h}{2} f(x_0) + h \sum_{k=0}^{n-1} f(x_k) + \frac{h}{2} f(x_n) \ .$$

The **`Integral`** module includes a straightforward implementation of the trapezoid rule:

```
double Trapezoid(double   a,            //                 ⎰b
                 double   b,            // Approximate      ⎱ f  by
                 double   f(double),    //                 a
                 unsigned n) {          // the trapezoid rule with
                                        // n  subdivisions.
```

Since it's so similar to function **Riemann** in Figure 4.2.2, the code isn't displayed here. You'll find it in file **Integral.CPP** on the accompanying diskette. **Trapzoid.CPP**, a test program analogous to the Figure 4.2.3 program **Riemann.CPP**, is also on the diskette. Its output, shown in Figure 4.3.2, demonstrates the greater accuracy of the trapezoid rule. The error is approximately proportional to the square of the step size h: each tenfold cut in h decreases the error about a hundredfold. This reflects the trapezoid rule error formula

$$\left| \begin{array}{c} \text{Trapezoid} \\ \text{Rule} \\ \text{Error} \end{array} \right| \le \frac{(b-a)M_2 h^2}{12} = \frac{(b-a)^3 M_2}{12n^2}$$

$$M_2 = \max_{a \le x \le b} |f''(x)| .$$

For a derivation of this inequality, consult Reference [3, Section 4.3].

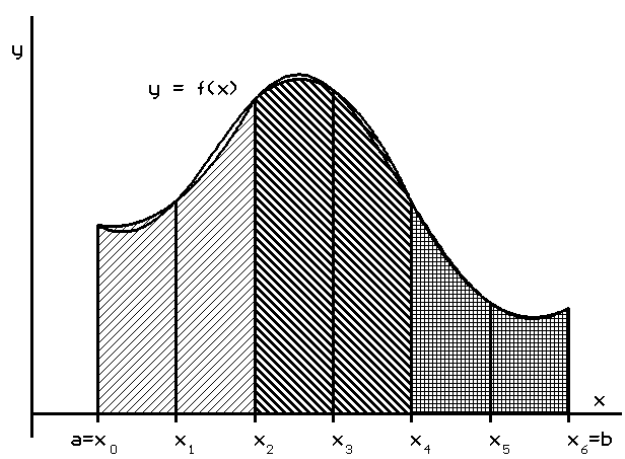
Approximating $\int_0^1 \cos x \, dx$ by the trapezoid rule

No. of sub-divisions	Approxi-mation	True value	Error
1	0.7701511529	0.8414709848	-0.071
10	0.8407696421	0.8414709848	-0.00070
100	0.8414639725	0.8414709848	-7.0e-06
1000	0.8414709147	0.8414709848	-7.0e-08
10000	0.8414709841	0.8414709848	-7.0e-10

Figure 4.3.2 Trapezoid rule approximations

Simpson's rule

Simpson's rule provides a still more accurate approximation. To derive it, reconsider the Riemann sum and trapezoid methods. The Riemann sum approximation $hf(x_k)$ to the integral over a single subdivision is the integral of the *constant* polynomial $c(x) = f(x_k)$ that agrees with $f(x)$ at the single point $x = x_k$:



$$\int_{x_k}^{x_{k+1}} f \approx hf(x_k) = \int_{x_k}^{x_{k+1}} f(x_k)\,dx = \int_{x_k}^{x_{k+1}} c.$$

Figure 4.3.3 Simpson's rule approximation

Similarly, the trapezoid rule approximation is the integral of the linear polynomial $l(x)$ that agrees with $f(x)$ at two points $x = x_k$ and x_{k+1}:

$$\int_{x_k}^{x_{k+1}} f \approx \frac{h}{2} f(x_k) + \frac{h}{2} f(x_{k+1}) = \int_{x_k}^{x_{k+1}} l \ .$$

(The trapezoid is the region between the x axis and the graph of l.) Simpson's idea is to approximate the integral over *two* subdivisions by the integral of the *quadratic* polynomial $q(x)$ that agrees with $f(x)$ at three points $x = x_k$, x_{k+1}, and x_{k+2}:

$$\int_{x_k}^{x_{k+1}} f \approx \int_{x_k}^{x_{k+1}} q \ .$$

Sum these double-interval approximations, as in Figure 4.3.3, to approximate the integral over the entire interval.

The first step in deriving Simpson's rule is to find the quadratic polynomial that agrees with $f(x_k)$ at three points $x = x_k$, x_{k+1}, and x_{k+2}. There's a systematic way to do that; once you see the trick, you should be able to write down, for any m, an mth degree polynomial—called the *Lagrange interpolating polynomial*—that agrees with any specified $m + 1$ values of f. Here's the quadratic; you can verify that it agrees with f at the specified points.

$$\begin{aligned} q(x) &= f(x_k) \frac{(x - x_{k+1})(x - x_{k+2})}{(x_k - x_{k+1})(x_k - x_{k+2})} + f(x_{k+1}) \frac{(x - x_k)(x - x_{k+2})}{(x_{k+1} - x_k)(x_{k+1} - x_{k+2})} \\ &\quad + f(x_{k+2}) \frac{(x - x_k)(x - x_{k+1})}{(x_{k+2} - x_k)(x_{k+2} - x_{k+1})} \\ &= f(x_k) L_k(x) + f(x_{k+1}) L_{k+1}(x) + f(x_{k+2}) L_{k+2}(x) \ . \end{aligned}$$

To integrate q over the interval $[x_k, x_{k+2}]$, compute

$$\int_{x_k}^{x_{k+2}} L_k = \int_{x_k}^{x_{k+2}} \frac{(x - x_{k+1})(x - x_{k+2})}{(x_k - x_{k+1})(x_k - x_{k+2})} dx = \frac{1}{2h^2} \int_{x_k}^{x_{k+2}} (x - x_{k+1})(x - x_{k+1} - h) dx$$

Substituting $u = x - x_{k+1}$, you get

$$\int_{x_k}^{x_{k+2}} L_k = \int_{-h}^{h} u(u - h) du = \frac{h}{3} \ .$$

Similarly, you can compute

$$\int_{x_k}^{x_{k+2}} L_{k+1} = \frac{4h}{3} \qquad \int_{x_k}^{x_{k+2}} L_{k+2} = \frac{h}{3}$$

$$\int_{x_k}^{x_{k+2}} q = f(x_k)\int_{x_k}^{x_{k+2}} L_k + f(x_{k+1})\int_{x_k}^{x_{k+2}} L_{k+1} + f(x_{k+2})\int_{x_k}^{x_{k+2}} L_{k+2}$$

$$= \frac{h}{3}f(x_k) + \frac{4h}{3}f(x_{k+1}) + \frac{h}{3}f(x_{k+2}) \quad .$$

Thus, the Simpson rule approximation of the integral of f over two subdivisions is

$$\int_{x_k}^{x_{k+2}} f = \int_{x_k}^{x_{k+2}} q = \frac{h}{3}f(x_k) + \frac{4h}{3}f(x_{k+1}) + \frac{h}{3}f(x_{k+2}) \quad .$$

Now, suppose that n is even, and sum up these approximations:

$$\int_a^b f \approx \left[\frac{h}{3}f(x_0) + \frac{4h}{3}f(x_1) + \frac{h}{3}f(x_2)\right] + \left[\frac{h}{3}f(x_2) + \frac{4h}{3}f(x_3) + \frac{h}{3}f(x_4)\right] + \cdots$$

$$+ \left[\frac{h}{3}f(x_{n-4}) + \frac{4h}{3}f(x_{n-3}) + \frac{h}{3}f(x_{n-2})\right] + \left[\frac{h}{3}f(x_{n-2}) + \frac{4h}{3}f(x_{n-1}) + \frac{h}{3}f(x\right.$$

For each bracketed term except the last, the third part matches the first part of the next, so you can simplify the expression for the sum:

$$\int_a^b f \approx \frac{h}{3}f(x_0) + \frac{4h}{3}f(x_1) + \frac{2h}{3}f(x_2) + \frac{4h}{3}f(x_3) + \frac{2h}{3}f(x_4) + \cdots$$

$$+ \frac{2h}{3}f(x_{n-4}) + \frac{4h}{3}f(x_{n-3}) + \frac{2h}{3}f(x_{n-2}) + \frac{4h}{3}f(x_{n-1}) + \frac{h}{3}f(x_n) \quad .$$

Figure 4.3.4 shows a function **`Simpson`** that returns a Simpson rule approximation to a definite integral

$$\int_a^b f$$

given the lower and upper limits **`a`** and **`b`**, a pointer to function **`f`**, and the number **`n`** of subdivisions. **`Simpson`** belongs to the module **`Integral`**; its source code is in file **`Integral.CPP`** on the accompanying diskette.

```
double Simpson  (double   a,            //                 ⎰b
                 double   b,            // Approximate      ⎱a f  by
                 double   f(double),    //
                 unsigned n) {          // Simpson's rule with  n
if (a == b) return 0;                   // subdivisions.
n += n % 2;                             // Ensure  n  is even.
if (n <= 0) n = 2;                      // At least  2  steps.
double h = (b - a)/n;                   // Step size  h .
double Sum = (f(a) + 4*f(a+h) + f(b)); // h/3 + 4h/3 + ... + h/3
for (unsigned k = 1; k < n/2; ++k) {
  double xk = a + 2*k*h;                // 2h/3 + 4h/3 + ...
  Sum += 2*f(xk) + 4*f(xk+h); }         //   ... + 2h/3 + 4h/3
return h*Sum/3; }                       // Here's the  h/3 .
```

Figure 4.3.4 Function **Simpson**

Test program **Simpson.CPP** is also on the diskette. It's analogous to the Figure 4.2.3 program **Riemann.CPP**. Its output, displayed in Figure 4.3.5, shows that Simpson's rule is far more accurate than the trapezoid rule. The error is approximately proportional to the fourth power of the step size h: a tenfold cut in h can decrease the error by about a factor of ten thousand. This reflects the Simpson rule error formula

$$\left| \begin{array}{c} \text{Simpson's} \\ \text{Rule} \\ \text{Error} \end{array} \right| \le \frac{(b-a)M_4 h^4}{180} = \frac{(b-a)^5 M_4}{180 n^4}$$

$$M_4 = \max_{a \le x \le b} | f^{iv}(x) | .$$

For a derivation of this inequality, consult Reference [3, Section 4.3]. The output in Figure 4.3.5 has some interesting features. First, the line for $n = 1$ subdivision is really for 2 subdivisions: function **Simpson** adjusts n minimally to ensure that it's even. The error for $n = 2$ is so small because the graph of $\cos x$ is nearly a parabola on the interval $0 \le x \le 1$. Second, there's little improvement from $n = 1000$ to 10000 because **Simpson** has reached the limit of attainable accuracy: fifteen significant digits. With more subdivisions, accuracy will deteriorate as round-off error becomes more significant.

```
                 ⎰1
Approximating    ⎱0 cos x dx  by Simpson's rule

No. of sub-      Approxi-            True
divisions        mation              value          Error
     1     0.841772092238272  0.841470984807897  0.00030
    10     0.841471452848890  0.841470984807897  4.7e-07
   100     0.841470984854646  0.841470984807897  4.7e-11
  1000     0.841470984807901  0.841470984807897  4.7e-15
 10000     0.841470984807895  0.841470984807897 -1.6e-15
```

Figure 4.3.5 Simpson rule approximations

You should experiment with Richardson extrapolation applied to trapezoid or Simpson rule approximations. Modify the program **Richards.CPP** mentioned in Section 4.2 to integrate with function **Trapezoid** instead of **Riemann**, and use $p = 2$ in the Richardson formula. You'll find that with the trapezoid rule and Richardson extrapolation the error is nearly proportional to h^4, just as with Simpson's rule. That's predictable, because of an alternative algebraic way of deriving Simpson's rule: the Simpson approximation for n subdivisions is just the Richardson extrapolation from the trapezoid approximations with $\frac{1}{2}n$ and n subdivisions. Try it!