

Physics 305: Ch. 1

Numerical Integration

In this chapter, we will examine the evaluation of definite integrals using the computer. Integration is basic to a large portion of what we will do in this class. Not only is it interesting to be able to evaluate definite integrals themselves, but because ordinary differential equations can be expressed in terms of integrals, integration is basic to the solution of differential equations. Much of the discussion of errors that follows will be used later in this course when we deal with differential equations.

Numerical integration, historically called quadrature, has been around since the advent of calculus. The search for quadrature formulas that minimized the number of times a function had to be analyzed with the maximum accuracy, was an active branch of inquiry: until the advent of the electronic computer. The most accurate quadrature formulas are quite complex, but since the computer can now analyze functions quickly, simple formulations of the problem are preferred.

The most basic strategy in numerical integration computer codes is to divide and conquer. Given a function you wish to integrate, say $f(x)$, first pick a set of $N + 1$ abscissæ, called a “grid”, $x_i, i = 0, 1, 2 \dots N - 1, N$. Then, approximate the integral in some simple way on each of the intervals $[x_i, x_{i+1}]$. Finally, the value of the full integral is then the sum of the values on each interval.

Initially we will consider well behaved integrals over a finite range. The basic idea is to evaluate the function at specific locations, and to use those function evaluations to approximate the integral. The idea should be familiar to you from elementary calculus. There you learned to think about the integral as the area under the curve, and you learned that this area could be found by a limiting process. In particular, you draw a lot of rectangles whose height is the function, sum the areas of the rectangles, and take the limit as the width of the rectangles, dx , goes to zero (and the number of rectangles goes to infinity). This is illustrated in the picture below. We use a particular example function, $y = 0.5 + x^2$, which has the nice feature that we can check our answers by integrating it exactly. More importantly, it is a smooth function over the integration interval.

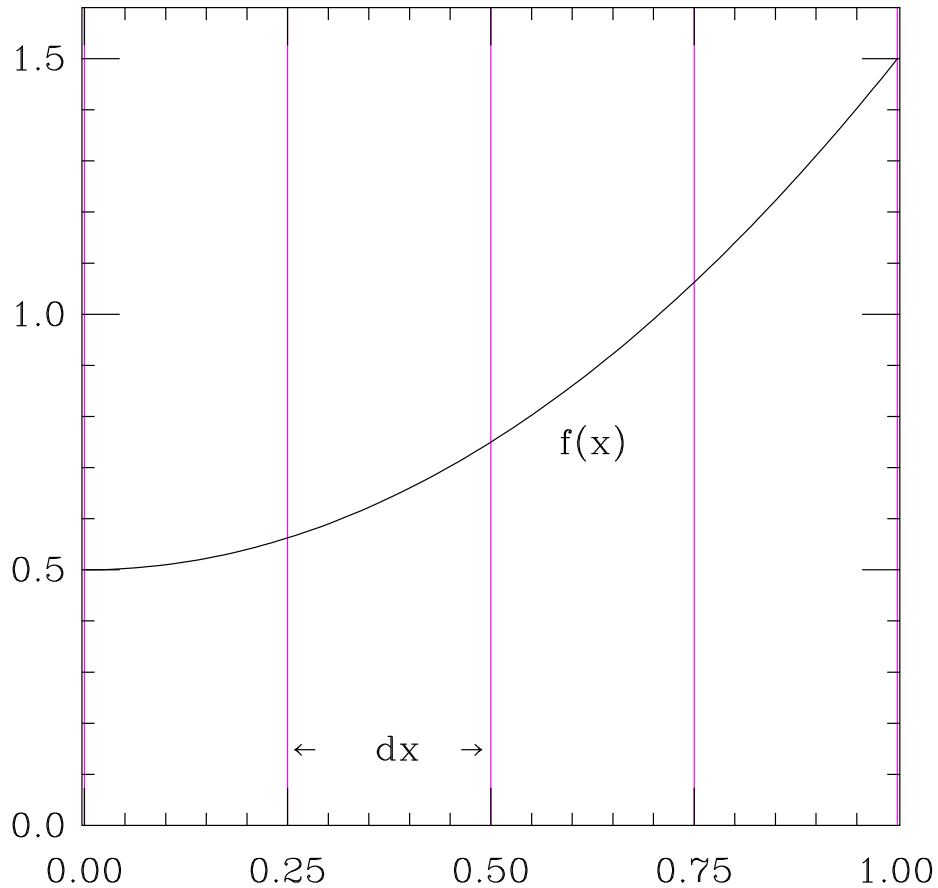


Figure 1.1: We will break our integral into many small pieces, each of width dx .

We thus seek solutions to the general formula

$$I = \int_{x_0}^{x_N} f(x) dx \sim \sum_{i=0}^{N-1} w_i f(x_i) \Delta x \quad (1.1)$$

where x_i are the evaluation points, and w_i is the weight to be assigned to the function evaluated $f(x_i)$. For the present we assume that the step size is constant, thus Δx is constant.

1.1 Piecewise Constant Integration

The simplest version of this formula is to draw rectangles whose height is equal to the function at the left hand side of each bin.

We call this “piecewise constant integration”, since it amounts to approximating the function by a constant in each little piece (dx). We can also call it the “left hand rule”, since the function in each bin is approximated by its value at the left side of

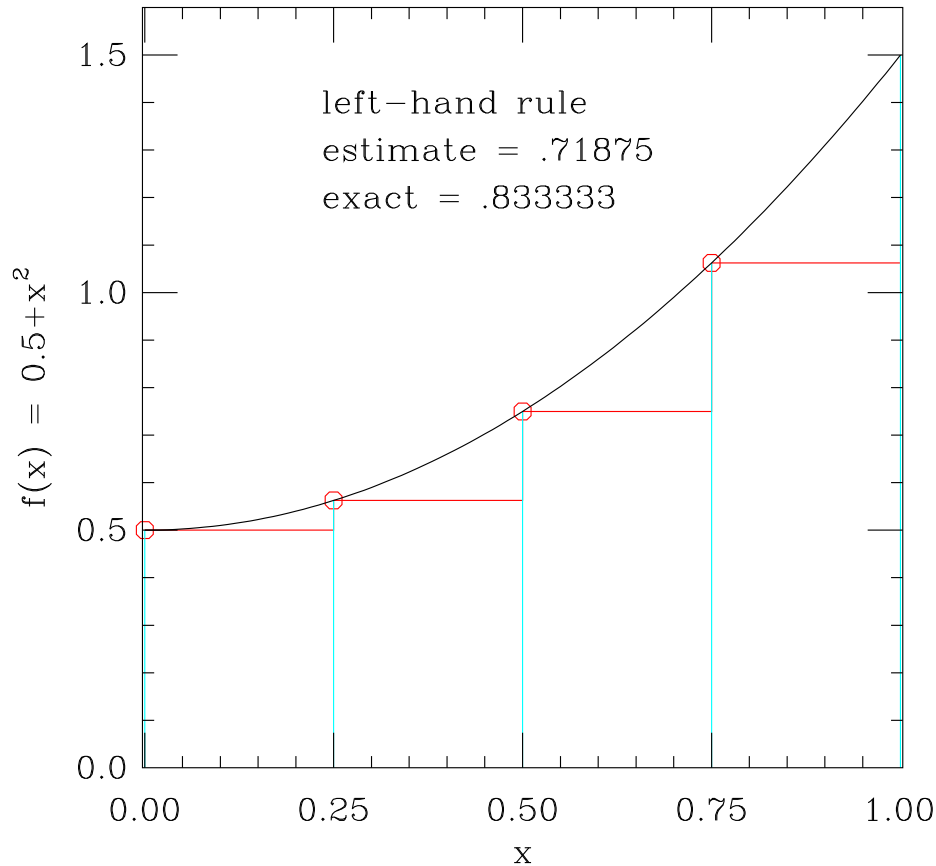


Figure 1.2: Here we try estimating the area in each piece using the height at the left-hand side.

the bin. We determine the integral by summing up the contributions of all intervals, written as

$$I = \int_{x_0}^{x_N} f(x) dx = \sum_{i=0}^{N-1} \int_{x_{i-1}}^{x_i} f(x) dx \sim \sum_{i=0}^{N-1} (x_{i+1} - x_i) f(x_i). \quad (1.2)$$

Thus, for the piecewise constant integration the weights, w_i , in equation (1.1) are all equal to 1.

In the picture above there are only four bins, and the sum of the areas of the four rectangles is 0.71875, while the actual integral of the function is 0.833333. Of course, all we have to do to get a better answer is to use more bins, and if you have a computer that isn't too hard to do.

Now let's try to understand the approximation we are making a little better. This is important, because to generate useful results we need to have some idea of how accurate they are. Alternatively, we may require an answer with a given accuracy, and we would need to decide how many bins to use. In the picture above, it is clear

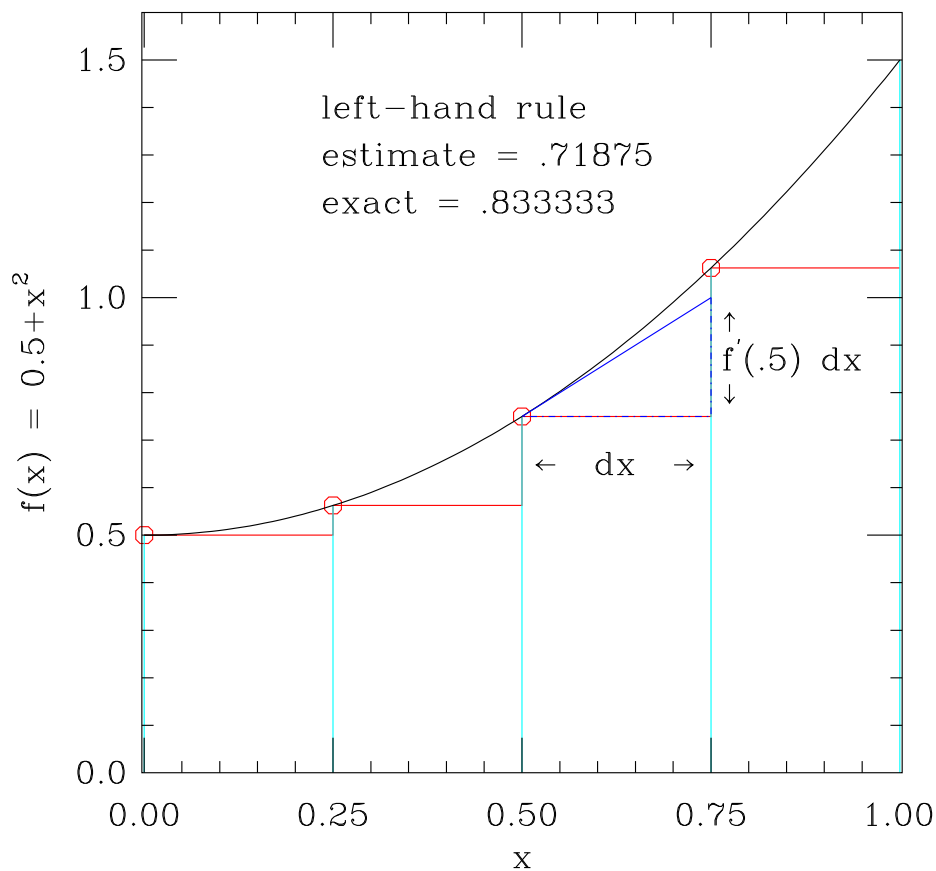


Figure 1.3: You can see that the error in our left-hand rule is scaling as $O(dx^2)$.

that the error we make is the area between the horizontal lines at the top of the rectangles and the original function. In general, we can't get these areas exactly — that would require knowing the integral of the function, and the reason we are doing the integral numerically is we don't know that. To make progress, observe that these areas are almost triangles — the graph of the function is almost a straight line within each bin. To make this quantitative, draw a triangle that is tangent to the function at the left hand side. That is, it has the same derivative as the function at the left hand side of the bin. Here is a picture of this for one of the bins:

The slope of the line is $\left. \frac{df(x)}{dx} \right|_{x_i}$, and the area of the right triangle is $\frac{1}{2} dx \frac{df}{dx} dx$. The crucial point is that this error is proportional to dx^2 , plus some higher powers of dx . For example, if we decrease dx by a factor of two, the area of the triangle will decrease by a factor of four. But we really want to know the error on the entire integral, and there is one more complication. When dx is decreased by a factor of two, we now need twice as many bins to cover the interval. So, even though the error in each bin is about 1/4 of what it was before, there are twice as many bins contributing to the total error, the the error on the entire integral decreased by a factor of $\frac{1}{4} \times 2 = \frac{1}{2}$. So,

for example, if you wanted to decrease your error by a factor of ten, you would need to decrease the bin size by a factor of ten, and sum over ten times as many bins. We call an algorithm where the error scales like this a “**first order algorithm**”, because the error scales as one over the first power of the amount of work you do.

Very shortly we will learn how to make the error scale better. But first, let’s make the graphical analysis of the error that we just did quantitative. This will be essential when we try to understand more complicated methods.

Our tool for analyzing this, and many other algorithms, is Taylor’s theorem. Hopefully you remember this from your calculus courses; if not you should go back and review it. Briefly, to approximate a smooth function $f(x)$ near some point x_i ,

$$f(x) = f(x_i) + (x - x_i)f'(x_i) + \frac{(x - x_i)^2}{2!}f''(x_i) + \frac{(x - x_i)^3}{3!}f'''(x_i) \dots \quad (1.3)$$

Note that in this equation the function and all of the derivatives are evaluated at the fixed value x_i , so they are all constants, and the whole thing is a polynomial in x .

Now look back at the last figure. The first term in this Taylor series, $f(x_i)$, is just the constant at the top of each rectangle, so this is the approximation for the function that we actually used in the left-hand, or piecewise constant, integration. Keeping one more term, $f(x_i) + (x - x_i)f'(x_i)$, is just the straight line at the top of the triangle.

To find the correct answer for the integral of $f(x)$ in this one bin, we can just integrate the Taylor series:

$$\begin{aligned} \int_{x_i}^{x_{i+1}} f(x) &= \int_{x_i}^{x_{i+1}} \left\{ f(x_i) + (x - x_i)f'(x_i) + \frac{(x - x_i)^2}{2!}f''(x_i) \dots \right\} \\ &= (x_{i+1} - x_i)f(x_i) + \frac{1}{2}(x_{i+1} - x_i)^2 f'(x_i) + \dots \end{aligned} \quad (1.4)$$

This is reproducing our graphical analysis. The first term is what we kept, and the next term is the largest part of our error. For brevity, define the interval width as $h = (x_{i+1} - x_i)$.

$$\int_{x_i}^{x_{i+1}} f(x) = h f(x_i) + \frac{h^2}{2} f'(x_i) + \dots \quad (1.5)$$

Again, the leading term in the error **from this one bin** is proportional to h^2 times the first derivative evaluated at x_i . When you integrate the function from a to b , you need $\frac{b-a}{h}$ bins, so the total error is proportional to $\frac{h^2}{h} = h$, and we say it is $O(h)$ (“order h ”).

1.2 The Midpoint Method

The piecewise constant method above suffered because the constant was chosen at one side of the grid zone, leading to a large error. Intuitively, it would be more sensible

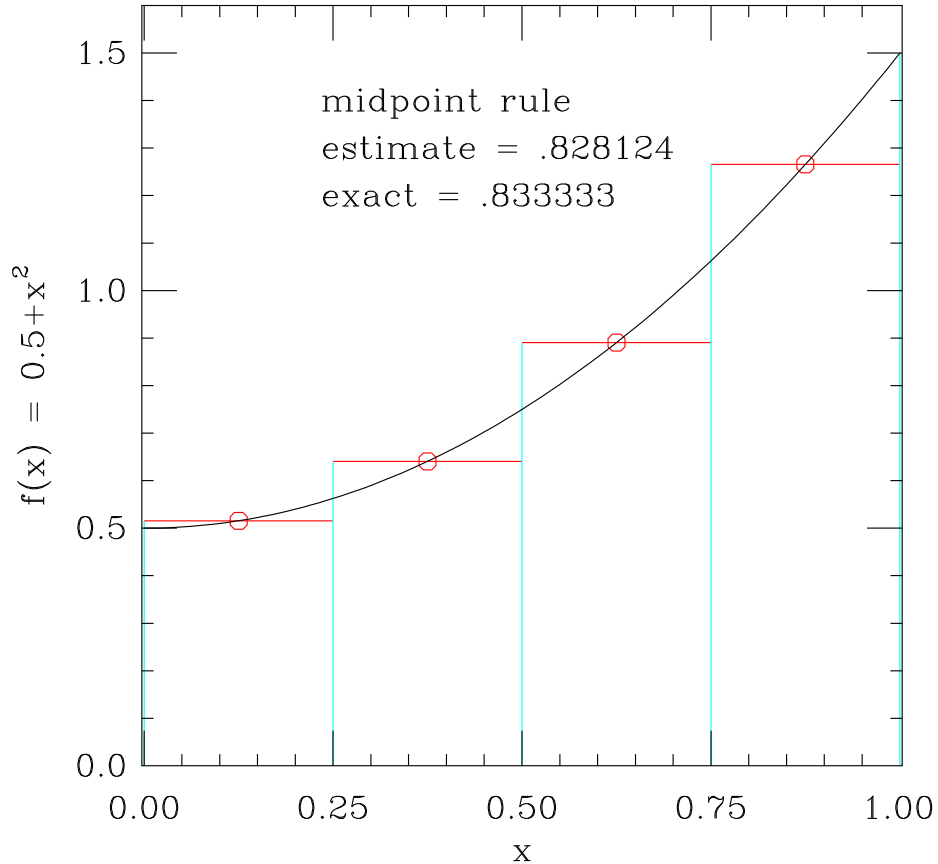


Figure 1.4: Now we'll estimate the height by using the value at the center of the zone.

to evaluate the function at the center of the zone, That is,

$$\int_{x_i}^{x_{i+1}} f(x)dx \approx (x_{i+1} - x_i)f\left(\frac{x_i + x_{i+1}}{2}\right). \quad (1.6)$$

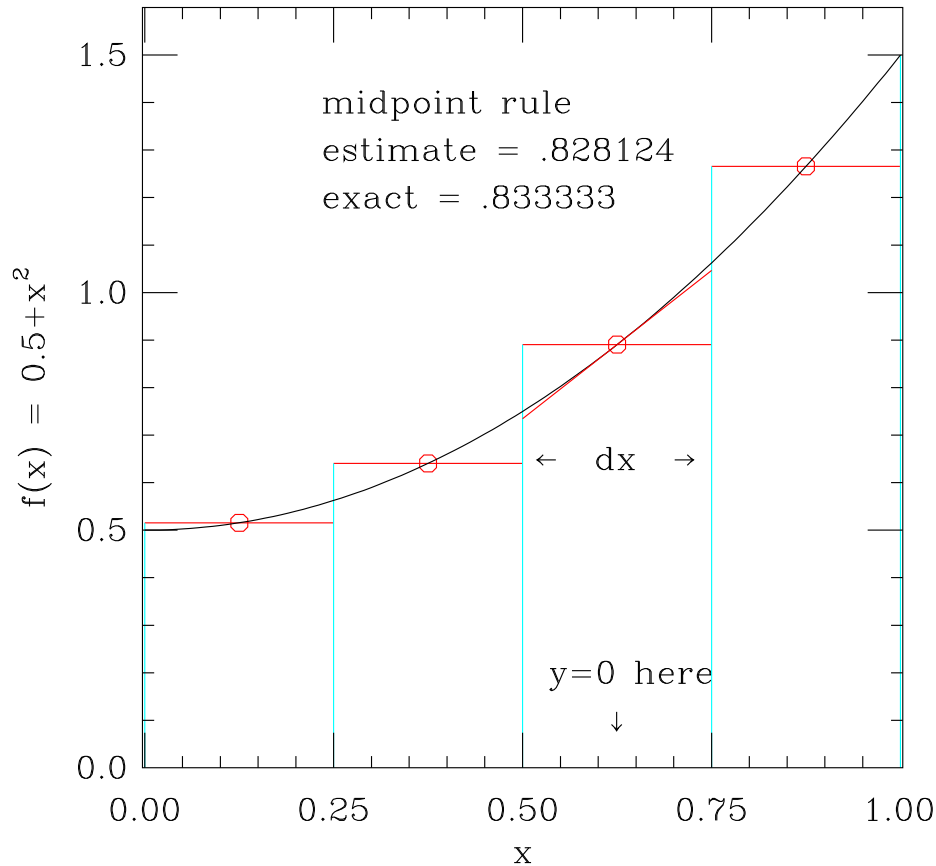
This is particularly easy to program: evaluate the function at the midpoint of all of the grid zones, sum the evaluations, and multiply by the grid spacing. Denoting the centers of the bins as $x_{1/2}$, $x_{3/2}$, etc., we have

$$\int_{x_0}^{x_N} f(x)dx = h [f_{1/2} + f_{3/2} + \dots + f_{(N-1/2)}]. \quad (1.7)$$

This is often called the “midpoint method”.

Here is a graphical illustration of the midpoint method for the same example as above. Again, our approximation is to sum the areas of the rectangles.

How accurate is this? Again, let's start with a picture. Looking at the third bin, imagine replacing the horizontal line through the function at the midpoint by a straight line tangent to the function at this point. The area under the trapezoid with this sloping line for its top is exactly equal to the area of the rectangle – the

Figure 1.5: The errors no longer scale as $O(dx^2)$.

two small right triangles whose hypotenuses are the straight line have the same area. Now the remaining error is just the small regions between the tangent line and the function itself.

Let's analyze this the same way we did for the left-hand rule. Now, however, we expand the function in a Taylor series around the midpoint of the interval. For the piece between x_0 and x_1 , we expand around $x_{1/2} = (x_0 + x_1)/2$.

$$f(x) = f(x_{1/2}) + (x - x_{1/2})f'(x_{1/2}) + \frac{(x - x_{1/2})^2}{2!}f''(x_{1/2}) + \dots \quad (1.8)$$

Now we would find the full answer for the integral of $f(x)$ over this bin by integrating this:

$$\int_{x_0}^{x_1} f(x)dx = \int_{x_0}^{x_1} \left\{ f(x_{1/2}) + (x - x_{1/2})f'(x_{1/2}) + \frac{(x - x_{1/2})^2}{2!}f''(x_{1/2}) + \frac{(x - x_{1/2})^3}{3!}f'''(x_{1/2}) \dots \right\} dx \quad (1.9)$$

This is easier to picture if we shift our variables so the midpoint of the interval is at zero: let $y = x - x_{1/2}$ (see figure above). Also, let's use the bin width $h = x_1 - x_0$.

$$\begin{aligned}
\int_{x_0}^{x_1} f(x)dx &= \int_{-h/2}^{h/2} f(y)dy \\
&= \int_{-h/2}^{h/2} \left\{ f(y=0) + y f'(0) + \frac{y^2}{2!} f''(0) + \frac{y^3}{3!} f'''(0) \dots \right\} \\
&= hf(0) + 0 + \left(\frac{1}{2}\right) 2 \left(\frac{1}{3}\right) \left(\frac{h}{2}\right)^3 f''(0) + 0 \dots \tag{1.10}
\end{aligned}$$

We wrote the the third term, $\left(\frac{1}{2}\right) 2 \left(\frac{1}{3}\right) \left(\frac{h}{2}\right)^3$ in a funny way to remind you of how we got it: $\left(\frac{1}{2}\right)$ from the coefficient in the term we are integrating, two limits of the integral — left and right, $1/3$ from $\int_0^a y^2 dy = (1/3)a^3$, and then the limit $h/2$ cubed. The whole term is just $\frac{1}{24} f''(x_{1/2}) h^3$. Remember, if you are getting confused, that this series is what we would get from integrating the original function over this interval. Now again you see that the first term is just what we used in the midpoint rule. The second term vanishes, so unlike the analysis of the left-hand rule, we have to go to the third term to find the largest part of our error. With the sign convention that the error is our approximate answer minus the correct answer, we would say the the error from each bin is $\frac{-1}{24} f''(x_{1/2}) h^3$ (plus higher order terms in h).

Actually, the second, fourth, sixth etc. terms all vanish, and we will use this later. You can see this in the equation above — we integrate an odd function of y ($f(-y) = -f(y)$) over an interval, $-h/2$ to $h/2$, that is symmetric around $y = 0$.

Now the important thing here is not that the coefficient $1/24$ is small, it is that the error from each bin is proportional to the cube of h , instead of the square of h as it was in the left-hand rule. Again you must remember that the total error is the sum of the errors in all the bins, and the number of bins is proportional to $1/h$. Thus the total error scales like $h^3/h = h^2$.

For the left-hand rule, if you increased the number of bins by a factor of ten, your error decreased by about a factor of ten. For the midpoint rule, it would decrease by about a factor of one hundred. That's a big improvement!

The midpoint rule is really no more difficult to program than the left hand rule. You might find yourself writing a code fragment like this: (note there is no problem with floating point comparisons at the upper limit of this loop.)

```

h = (b-a)/nbins;
sum=0.0;
for( x=a+h/2.0; x<b; x=x+h ){
    sum = sum + XXXX    // you figure this out
}

```

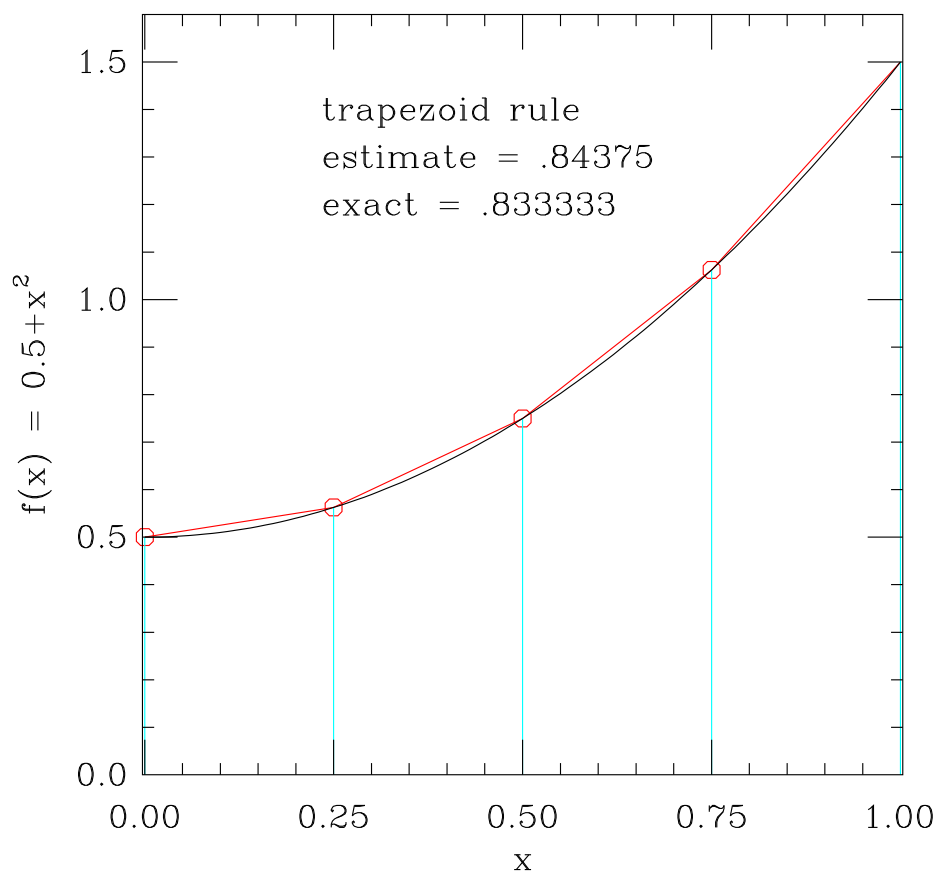


Figure 1.6: In the trapezoidal rule, we use both ends of the zone to estimate the area.

1.3 Trapezoidal Rule

Another way to improve on the piecewise-constant (left-hand) method would be to approximate the function $f(x)$ on each interval by the average of its values at the left and right sides of the interval. This improved approximation is shown here — we just sum up the areas of all the trapezoids:

Our approximation to the integral is then

$$\int_a^b f(x) dx \sim \sum_i (x_{i+1} - x_i) \frac{[f(x_{i+1}) + f(x_i)]}{2} . \quad (1.11)$$

This integration method is known as the “trapezoidal rule”. Notice that you don’t really have to evaluate the function twice as many times as in the midpoint rule. Except for the first and last points, each point is used twice, each time with weight $(x_{i+1} - x_i)/2$, for a total weight of one. Therefore, the integration weights, w_i , in equation 1.1 will be $1/2, 1, 1, \dots, 1, 1/2$.

The analysis of this method is just a little more difficult than the analysis for the midpoint rule. Surprisingly, it is still easiest to do the Taylor expansions around the

middle of the interval. We can begin from Eq. 1.10,

$$\begin{aligned}
 \int_{x_0}^{x_1} f(x) dx &= \int_{-h/2}^{h/2} f(y) dy \\
 &= \int_{-h/2}^{h/2} \left\{ f(y=0) + y f'(0) + \frac{y^2}{2!} f''(0) + \frac{y^3}{3!} f'''(0) \dots \right\} \\
 &= h f(0) + 0 + \frac{h^3}{24} f''(0) + 0 \dots
 \end{aligned} \tag{1.12}$$

where as before we use the shifted variable $y = x - x_{1/2}$.

OK – that is still the correct answer for integrating the function. But what did we do? We took $\frac{h}{2} \left(f(y = \frac{-h}{2}) + f(y = \frac{+h}{2}) \right)$. So let's expand the functions in this expression in Taylor series as well:

$$\begin{aligned}
 &\frac{h}{2} \left\{ f \left(y = \frac{-h}{2} \right) + f \left(y = \frac{+h}{2} \right) \right\} \\
 &= \frac{h}{2} \left\{ f(y=0) + \frac{-h}{2} f'(0) + \frac{1}{2} \left(\frac{-h}{2} \right)^2 F''(0) + \frac{1}{6} \left(\frac{-h}{2} \right)^3 F'''(0) + \dots \right\} \\
 &= \frac{h}{2} \left\{ f(y=0) + \frac{h}{2} f'(0) + \frac{1}{2} \left(\frac{h}{2} \right)^2 F''(0) + \frac{1}{6} \left(\frac{h}{2} \right)^3 F'''(0) + \dots \right\} \\
 &= h f(0) + 0 + \frac{h^3}{8} f''(0) + 0 + \dots
 \end{aligned} \tag{1.13}$$

$$\tag{1.14}$$

Again, because of the reflection symmetry, all the even powers of h cancel.

Now to find the error from this one little interval, take the approximation we used, in Eq. 1.13 and subtract the correct answer, Eq. 1.12, to see that the trapezoidal rule give an error from each bin of $\frac{+1}{12} f''(x_{1/2}) h^3$ (plus higher order terms in h).

Like the midpoint rule, for the same small grid spacing, h , the trapezoidal rule integration will have an error which is about a factor of h smaller than a piecewise constant integration. This also tells us that if we make the interval h smaller by a factor of two our error from each bin will decrease by a factor of 8. Of course, we need twice as many bins, so the error in the final answer decreases by a factor of 4. (plus, as always, higher order terms in h)

1.4 Simpson's Rule

For the midpoint rule, we found that the error from one bin was $\frac{-1}{24} f''(x_{1/2}) h^3 + \mathcal{O}h^5$ (the h^4 term cancelled). For the trapezoid rule, we found $\frac{1}{12} f''(x_{1/2}) h^3 + \mathcal{O}h^5$ (again, the h^4 term cancelled).

These errors have opposite signs, so why not average the two methods? In particular, take $2/3$ times the midpoint rule answer plus $1/3$ times the trapezoid rule answer. This is called Simpson's rule. To find the error in this method, we need only take $2/3$ times the midpoint rule error plus $1/3$ times the trapezoidal rule error. But this is zero all the way out to order h^5 ; the h^3 terms cancel. Notice that it is important that the order h^4 terms cancelled in both the midpoint and trapezoid rules. This was for the same reason in both cases, namely that the way we treated each bin was symmetric under reflecting the bin, or interchanging the left and right sides. Because of this, the error for the one bin was an odd function of h in each case.

Again, for the total error we remember that there are order $1/h$ bins all contributing to the error, so the total error in integrating a smooth function by Simpson's rule is order h^4 ; doubling the number of gridpoints (halving the size of h) will give us an answer which is 16 times more accurate. For Simpson's Rule, the integration weights, w_i , in equation 1.1 are $1/3, 4/3, 2/3, 4/3, \dots, 2/3, 4/3, 1/3$. In applying this equation with these weights, notice that Simpson's rule involves evaluating the function at intervals of $h/2$ with the definition of h used above. In Eq. 1.1 the grid spacing is therefore $1/2$ of this.

1.5 Higher Order Formulæ

One can carry this on *ad infinitum*, fitting the function on each interval with ever higher-order polynomials to take into account ever more derivatives of the function. By so doing, we can cancel more terms in the expression of the error to achieve ever more accurate (in the sense of scaling with higher powers of h) expressions for the integral.

Note, however, that the formulæ get more complicated just as fast as they get more accurate! In practice, we can get as accurate an answer as we like from the midpoint or trapezoidal rule just by increasing the number of gridpoints. In the "old days" before computers, it may have been a savings in time to minimize the number of function evaluations (each of which you had to do by hand!), in which case employing a "higher-order" integration rule would be more efficient – one gets the same accuracy from fewer function evaluations.

With a computer, you only have to program the function evaluation once. That done, it is no more effort (for you) to compute that function very many times. Computers are fast, and thus increasing the number of gridpoints rather than increasing the complexity is usually a more efficient use of *your* (not the computer's) time to get an accurate solution. Round-off error will alter this conclusion in some cases, but that is a topic for a later lecture.

As a first try, then, our recommendation is first to use the midpoint or the trapezoid rule with as many points as you need to get a reasonable answer. Once you have obtained a numerical approximation to an integral, you can ask: how accurate *is* the answer? *Proving* that an answer is of a given accuracy is difficult, but as long as your

function is not too ill-behaved, a first try is to double the number of gridpoints and recalculate the integral. If the answer is not too different from the first try, it is a good bet that your error is at least approximately bounded by the difference between the two results.

1.6 Algorithms

You are probably well aware of the idea that when you sit down to write a lengthy paper, you should begin by writing an outline. Similarly, when beginning a programming project of any complexity, it is important to make an outline of all the steps that will be involved. This outline is called an “algorithm”. A well-written algorithm should make all of the important logical steps clear to the reader. Like most outlines, more details should be given in the more difficult sections of the program. Examples of algorithms for the methods in this section follow on the next pages; take a look at them now. Note that by studying the algorithms, you should be able not only to understand what the method does but also to convince yourself that it will solve the problem!

Once the algorithm is written, converting each step to actual C code should be relatively straight-forward. If you find that implementing a step is tricky, it probably means that it should have been described in the algorithm! Occasionally you may discover while writing a step that the problem is more complicated than you originally thought or that the method you are implementing has some flaw. In this case, you need to back up and revise your algorithm.

It is especially important for beginning programmers to write out their algorithms before starting to write code. One of the key benefits of this approach is that one can recognize how the problem can be broken into a series of functions. It is generally easier and less error-prone to write a set of simple functions than it is to write a complicated program in one go! Remember that the your total cost of a manner of writing code is *not* the amount of time required to get a “first draft” but rather the amount of time required to get a program that works correctly and reliably. As the complexity of the problems increase, you will find that the time saved debugging well-planned, modular code outweighs the time spent in planning.

A related piece of advice that is more specific to scientific computing is that you should *always* derive any non-trivial equation on paper before trying to implement it in code. In other words, when coding equations, always work from paper (and keep that paper so that you can refer to it when you’re trying to debug!).

1.7 Implementation

The implementations of numerical integration generally boil down to needing to sum up a list of numbers. You will write this as a loop, inside of which you compute one

functional value at a time and add it to a variable that is accumulating the values. Before the loop, you should set the summation variable to zero. Of course, you need to decide the number of grid zones before you start the loop, so that the loop can execute the correct number of times. When the loop is done, you can multiply by the grid zone width.

As a practical matter, when using a variable to accumulate a very large list of numbers (e.g., millions), it is better to use double precision at least for this one variable. Otherwise, the round-off error of the accumulating list can lead to undesirable loss of precision.

1.7.1 Algorithm for the Midpoint method

1. Declare variables.
2. Set the number of grid zones N .
3. Compute the width of the grid zones h .
4. Set the summation variable to be zero.
5. Loop from 0 to $N - 1$, inclusive.
 1. Compute the lower boundary of this particular grid zone, e.g, as the lower bound plus the step number i times the zone width h .
 2. Compute the functional value at this x .
 3. Add this value to the summation variable.
6. Multiply the summation variable by the width of the grid zones.
7. Write to the terminal the value of the summation variable and the number of grid zones used.

Function `fnc1`: Given 1 argument (type `float`):

1. Declare any additional variables.
2. Calculate the value of the function at the given point.
3. Return the value as a `float`.

1.7.2 Algorithm for the Midpoint method

1. Declare variables
2. Set the number of grid zones N
3. Compute the width of the grid zones h
4. Set the summation variable to be zero.
5. Loop from 0 to $N - 1$, inclusive.
 1. Compute the midpoint of this particular grid zone, e.g, as the lower bound plus the step number $i + 1/2$ times the zone width h .
 2. Compute the functional value at this x .
 3. Add this value to the summation variable.
6. Multiply the summation variable by the width of the grid zones.
7. Write to the terminal the value of the summation variable and the number of grid zones used.

Function fnc1: Given 1 argument (type float):

1. Declare any additional variables.
2. Calculate the value of the function at the given point.
3. Return the value as a float.

1.7.3 Algorithm for the Trapezoidal method

1. Declare variables
2. Set the number of grid zones N
3. Compute the width of the grid zones h
4. Set the summation variable to be half the sum of the functional values at the lower limit and the functional value at the upper limit.
5. Loop from 1 to $N - 1$, inclusive.
 1. Compute the x_i boundary of the grid zone, e.g, as the lower bound plus the step number i times the zone width h .
 2. Compute the functional value at this x_i .
 3. Add this value to the summation variable.
6. Multiply the summation variable by the width of the grid zones.
7. Write to the terminal the value of the summation variable and the number of grid zones used.

Function fnc2: Given 1 argument (type float):

1. Declare any additional variables.
2. Calculate the value of the function at the given point.
3. Return the value as a float.

1.8 Improper Integrals: Handling an infinite domain

What happens if the function $f(x)$ is singular somewhere in the desired domain? Or if one of the limits of integration is at infinity? These are known as “improper integrals”. As you know, these situations can sometimes yield convergent answers. But for the computer, it is a big problem: we can’t use a grid that extends to infinity (or at least the program will never finish!), and the result will be infinite if one of the grid points falls at a singular point. Fortunately, with a simple change of variables and the use of the midpoint method, we can convert these problems into our conventional form.

Let’s consider the infinite domain first, e.g., a problem of the form

$$\int_a^\infty f(x)dx. \quad (1.15)$$

Of course, this only gives a finite answer if $f(x)$ goes to zero faster than $1/x$, i.e., we need the limit of $xf(x)$ as $x \rightarrow \infty$ to be zero. If the convergence is very fast, e.g., like $\exp(-x)$ or $\exp(-x^2)$, then the simplest approach may be to pick the upper limit of the integral to a number that is comfortably larger than where the integrand is large enough to matter, but still small enough that a reasonable number of grid zones still give good sampling where the integrand is big. A plot can help decide this, and one can try a few upper limits to check that the answer is well converged.

However, many integrands do not converge to zero that quickly. We can make the problem much more efficient with a change of variables. For example, consider the transform $y = a/x$ or $x = a/y$. Since this implies $dx = (a/y^2)dy$, the integral then becomes

$$\int_a^\infty f(x)dx = \int_0^1 af(a/y)y^{-2}dy = \int_0^1 g(y)dy \quad (1.16)$$

for $g(y) = af(a/y)y^{-2}$. This has converted our domain to a finite range, so we can use a finite grid.

We have two problems remaining. First, if we try to evaluate $g(y)$ at $y = 0$, our code will crash with a divide by zero error. There are two solutions to this. One is to note that for some functions $f(a/y)$, the convergence as y goes to zero is fast enough that the product f/y^2 is zero. If so, one can insert a condition in the code that checks if $y = 0$ and returns zero while skipping the division by y . The other solution is to use the midpoint method, because that only evaluates the function at the midpoints of the grid cells, which means we will never evaluate at $y = 0$!

The other problem is that the function $g(y)$ may have a singularity at $y = 0$. This will occur if the function $f(x)$ goes to zero as a power-law between $1/x$ and $1/x^2$. The answer is still finite, but the problem is that the integrand is increasing so rapidly near $y = 0$ that we may have to use a very large number of grid cells to get an accurate answer! The error formulae we derived all had nice dependences on h ,

but they also involved a higher derivative of the integrand itself, and those derivatives can be very big near a singularity! This problem with singularities is just an example of the generic problem with improper integrals (next section).

Of course, if the function $f(x)$ was converging faster than $1/x^2$, then $g(0) = 0$, and there is no singularity, so a normal solution will work for the transformed problem.

1.9 Improper Integrals: Handling a singularity (Supplemental topic)

The remaining parts of this chapter won't be on your homework assignments. Since this is an introductory course, we are covering only the basic parts of most of the topics. The next few pages would be covered in a more advanced course, and give you an idea of what you might learn in the future.

In the singular case, again we want to change variables. Here, it is best if one can evaluate the power-law dependence of the singularity. For example, let's imagine that we want to solve an integral with a singularity that is diverging as $x^{-1/2}$, e.g.,

$$\int_0^a f(x)dx = \int_0^a \frac{h(x)}{\sqrt{x}}dx \quad (1.17)$$

where $h(x)$ is a non-singular function (which we may still prefer to compute as $\sqrt{x}f(x)$). Then let's consider a change of variables to $y = \sqrt{x}$ or $x = y^2$; this implies $dy = dx/2\sqrt{x}$. The integral becomes

$$\int_0^{\sqrt{a}} 2h(y^2)dy. \quad (1.18)$$

The problem has gone away! We still have a finite range of integration and the function h was non-singular so the integrand $2h(y^2)$ is finite everywhere. We can use our normal trapezoidal or midpoint method.

What has happened? In a sense, the problem with the singularity is that our uniform grid in x was having the areas of the grid cells getting bigger and bigger as we approached the singularity. The sum was finite, but the total value was very sensitive to the grid choice. With the change of variables, we have created a new grid that is uniform in y but non-uniform in x ; note that $x = y^2$ means that we have more zones closer to $x = 0$. The new grid is carefully chosen so that the areas of the cells are closer to constant near the singularity. By making the cells more equally important, we avoid the sensitivity to the grid choice.

This is an example of decomposing an integrand into a product of two functions: one of which we can integrate analytically and the other of which is slowly varying. By using this information to change variables, we can focus our work only on the smoothly varying part, which can yield an accurate answer with only a modest number of function evaluations. Even our $y = a/x$ transformation of the last section can be

boosted in performance by using the right choice of power-law to match the limiting behavior of the integrand.

Having said this, computers are very fast, and using a thousand or even a million grid zones will finish quickly. It is often a better use of one's time to use a simple method with very small spacing than to manipulate the integral into a more optimal form. Working out a fancier transformation is best left for when you need a lot of accuracy or a lot of speed.

One always has the option to separate the original integrand into multiple pieces and use different transforms on each. For example, if we have an integral

$$\int_0^{\infty} f(x) \quad (1.19)$$

then the transform $y = 1/x$ will map the $x \rightarrow \infty$ limit to a finite value, but will make a mess of $x = 0$ limit, which it will map to $y = \infty$. Rather than trying to figure out a different transform, just break the integral:

$$\int_0^{\infty} f(x) = \int_0^1 f(x) + \int_1^{\infty} f(x) \quad (1.20)$$

and solve the two parts separately, using the transform on the second one.

Of course, if the singularity is diverging too rapidly, the integral has an infinite value, and there's no way to get a solution, computer or not!

1.10 Extrapolation, or dirty tricks (Supplemental topic)

It is useful to note that for the case of the trapezoidal rule if we double the number of gridpoints, the new grid can contain the same points as the old grid, plus an equal number of points in between. Thus, once we have calculated a numerical integral on a given grid, doubling the number of grid points only requires doing the *same* amount of work if we re-use the information obtained in doing the first integral.

If we perform this doubling successively, we generate a sequence of results, I_N , one from each halving of the stepsize. Let's look at how we calculate these summations while reusing the old gridpoints. When we implement the trapezoid rule, say on the interval 0 to 1, we should begin by calculating

$$I_1 = \frac{1}{2} (f(0) + f(1)) \quad (1.21)$$

and then proceed with

$$I_2 = \frac{I_1}{2} + \frac{1}{2} [f(0.5)], \quad (1.22)$$

$$I_4 = \frac{I_2}{2} + \frac{1}{4} [f(0.25) + f(0.75)], \quad (1.23)$$

$$I_8 = \frac{I_4}{2} + \frac{1}{8} [f(0.125) + f(0.375) + f(0.625) + f(0.875)], \quad (1.24)$$

and so forth. You should check for yourself that the I_N given by this recursion exactly implement the trapezoidal rule for N points. We decide to stop with I_{2N} and I_N differ by less than some amount.

Theoretically, the successive I_{2N} should converge to the true value of the integral. The question which naturally arises is whether or not there is some sensible way of extrapolating to the limit from the few values already at hand. One simple form of this extrapolation is based on the assumption that the derivative of the principal error term remains constant (C) as we progress along our sequence of integral estimates. If we are using the trapezoidal rule, we have

$$I = \frac{H}{2}(f_0 + f_2) + CH^3 = I_1 + 8Ch^3 \quad (1.25)$$

where $H=2h$, because we are evaluating the integral using the next stepsize. We could also evaluate this as

$$I = h\left(\frac{1}{2}f_0 + f_1 + \frac{1}{2}f_2\right) + 2Ch^3 = I_2 + 2Ch^3 \quad (1.26)$$

Utilizing the assumption that C is constant (not always a great assumption), we can combine the above two equations, and we derive

$$I = \frac{4}{3}I_2 + \frac{1}{3}I_1 \quad (1.27)$$

as our extrapolation formula. The above equation is sometimes called the *Richardson extrapolation* formula. This is, in fact, the result we would obtain if we used Simpson's Rule (check this out for yourself). Therefore, we find that we can use a combination of the results from calculations using the trapezoid rule to improve our estimate.

1.11 Extrapolation: An example (Supplemental topic)

As a simple example of the power of using the convergence of the sequence of I_N , we turn back to the integral $\int_0^\pi \sin(x)dx$. We can generate the following series of I_N for

various numbers of gridpoints N using the trapezoidal rule:

N	h_N	I_N	error/ h_N^2
2	3.1415927E+00	1.9236707E-16	2.0264237E-01
4	1.0471976E+00	1.8137994E+00	1.6979462E-01
8	4.4879895E-01	1.9663167E+00	1.6722886E-01
16	2.0943951E-01	1.9926838E+00	1.6678864E-01
32	1.0134170E-01	1.9982880E+00	1.6669520E-01
64	4.9866550E-02	1.9995855E+00	1.6667357E-01
128	2.4736950E-02	1.9998980E+00	1.6666837E-01
256	1.2319971E-02	1.9999747E+00	1.6666709E-01
512	6.1479308E-03	1.9999937E+00	1.6666677E-01
1024	3.0709606E-03	1.9999984E+00	1.6666669E-01
2048	1.5347302E-03	1.9999996E+00	1.6666667E-01
4096	7.6717769E-04	1.9999999E+00	1.6666666E-01

Where we have taken the error as the exact value minus the numerical approximation ($2 - I_N$). This table demonstrates two important facts. First, the integral gets more accurate (approaches the exact answer of 2) as the number of intervals increases. Second, as advertised, the error is roughly proportional to h^2 (the error on each interval is proportional to h^3 and there are $N \propto 1/h$ intervals). Note that every time we double the number of points, we only have to evaluate the function at half the gridpoints as we have already computed half when we determined the previous solution.

Since the error is approximately proportional to h^2 , we will fit the error as a function of stepsize with a polynomial in h^2 . We can then extrapolate the result to the limit of zero stepsize. Because we haven't yet considered fitting polynomials in this course, as a simple example, let us fit a polynomial of order 1 (a line) in h^2 to the first two points in the table. We have

$$I_N = m * (h_N^2) + b. \quad (1.28)$$

The slope of the line m between the first two lines in the table is

$$m = \frac{I_4 - I_2}{h_4^2 - h_2^2} \quad (1.29)$$

The answer extrapolated to zero stepsize $h = 0$ is then the value of the y-intercept b . Putting in the numbers from the table, the extrapolated answer is $1.974E + 00$, a 2% error (see Figure 1.7). This is considerably better than the answer using 4 points, which makes a 9% error.

By fitting a quadratic polynomial to the first three points, the extrapolated answer is $2.001E + 00$, which is better than one part in 10^3 and is as accurate as the direct trapezoidal rule result for 64 points while using only 8. Carrying on, fitting a cubic

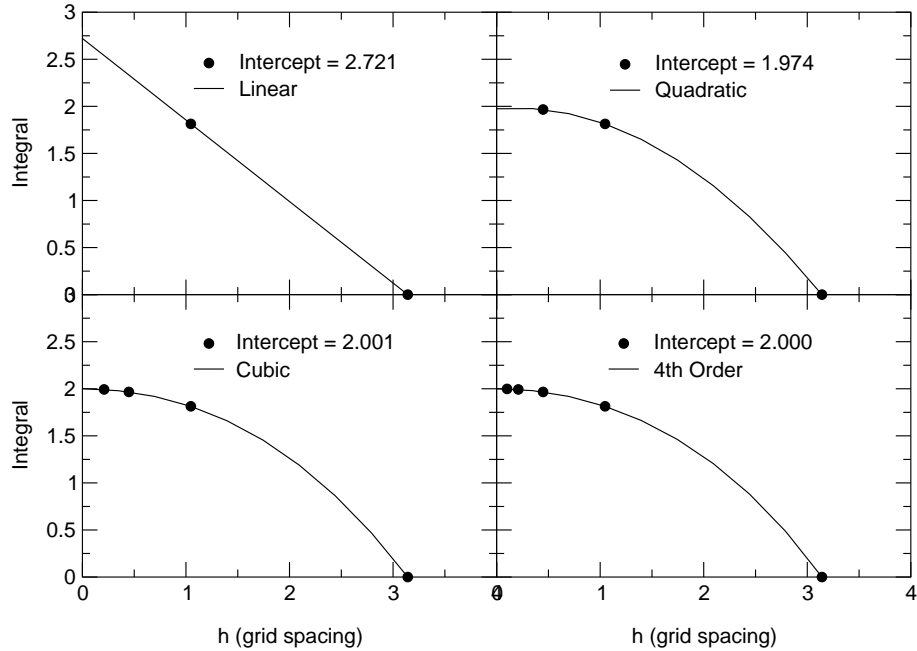


Figure 1.7: Extrapolating the results from the output of previous table using fits of h^2 to increasingly higher order polynomials.

polynomial to the first four points gets the answer to machine accuracy with 16 points, better than using 2048 points with the direct trapezoidal rule.

Even doing linear extrapolation on the most recent two approximations yields machine accuracy in 128 points. For more of the theory behind this method, and for information on how to do polynomial extrapolation, see *Numerical Recipes*, Chapter 4.

What happens if the function $f(x)$ is singular at one of the grid points? This is a slightly more advanced topic, but the case is easily handled (for some kinds of singularity, at least). Again, see *Numerical Recipes*, section 4.4, on “Improper Integrals.”

1.12 Gaussian Quadrature (Supplemental topic)

Without going into the details here, there is another approach to numerical integration which is worth mentioning. In the above discussion, we have kept the abscissæ fixed, with a constant stepsize between them. We then defined “weights” of various complexity to give us ever-increasing accuracy. We thus wrote the numerical integral of $f(x)$ as

$$\int_a^b f(x)dx \sim \sum_{i=0}^{N-1} w_i f(x_i). \quad (1.30)$$

There is no reason, except for simplicity, to determine the weights in such a simple manner. If we allow ourselves to choose the x_i in some other way, we will have twice the number of degrees of freedom to achieve greater accuracy from the same number of points. One common application of this idea is known as *Gaussian Quadrature* (quadrature is an archaic term for integration). Here, one chooses the abscissæ *and* weights to make the integral exact for integrals that are a polynomial times some function $W(x)$. The “weighting function” $W(x)$ can also be chosen to remove integrable singularities from the integral. Thus, we can find a set of abscissæ and weights such that

$$\int_a^b W(x)g(x)dx \sim \sum_{i=0}^{N-1} w_i g(x_i) \quad (1.31)$$

is exact if $g(x)$ is a polynomial. We can then use this procedure for integrals of a function $f(x)$ where $g(x) = f(x)/W(x)$ is “close” to a polynomial. The resulting accuracy is almost magical. One can get fabulous accuracy for some functions with *very* many fewer function evaluations than for the trapezoidal rule or its derivatives.

The theory of how to find these weights, though beyond the scope of this course, is clever (going back to Gauss) and very useful. You should have a look at *Numerical Recipes*, section 4.5 for a description. There are many routines you can use to find these “Gaussian Quadratures” for a given function $W(x)$. In fact, the weights and abscissæ are tabulated in many places for “popular” $W(x)$ s.

1.13 Homework 2

Due Monday, September 7, at 10 pm

This assignment is due before the end of the day on Monday, September 7. Those of you planning a wild Labor Day weekend should consider doing this assignment before the weekend.

This week we will learn how to do something that is actually useful with the computer, namely integrate a function numerically.

One of the things that we want you to learn about is the effects of the limits on precision of computer arithmetic on your results. Therefore, even if you already know enough about C to know about “double” variables, please continue to use “float” variables this week. Next week we will switch over to using double.

Once again, remember that your homework submission is a report which should contain your programs, but must also explain what each program does, what the numerical methods are, how you tested it, some sample input and output, and your conclusions. (Think about your laboratory reports in other classes — you wouldn’t just turn in a table of your measurements!) We do want you to turn in your (properly commented) programs for all three methods.

- 1) Write a program to calculate

$$\int_a^b x e^{-x} dx \quad (1.32)$$

using the “left hand” rule. The input should be the two endpoints, `a` and `b` and the number of intervals to be used in the integration. Note that the C function for producing e^x is `exp()`.

Once you have this program working, copy it and modify it to use the midpoint rule. Then, copy it again and modify it to use trapezoid rule.

Using `a=0` and `b=1.5`, make a table of the values your programs give (all three methods) using the number of bins 2, 4, 6, 10, 20, 50, 100, 200, 1000, 10000, 100000 and 1000000.

Compare your results to the correct value of the integral — you should be able to do this integral analytically. Then study the dependence of the error on the bin size. See the table in section 2.11 of the class notes for an example of how to go about this. In particular, note the last column where the error is rescaled to see if it follows the expected power law. Then make a log-log plot of the absolute value of the difference of your results from the correct value versus the number of intervals. You should turn in a postscript, PNG, or PDF version of the plot. Discuss whether or not your results are what should be expected.

(Assignment continues on next page)

Note: It may not matter for the way you choose to do the plotting, but beware that the C function for taking the absolute value of a floating point number is `fabs()`. The function `abs()` only works for integers.

2) Use your program to calculate

$$\int_0^{\infty} x e^{-x} dx \quad (1.33)$$

Use any method you want, but you may need to think a little about how to deal with the infinite upper bound. Be sure to explain how you are treating this issue.

Optional, Not for credit: If you want to do something only slightly harder, modify your program again to use Simpson's rule. Then find the error scaling in this method and add it to your analysis, including the log-log plot.

1.14 Homework 3

Due Monday, September 14, 10 pm

This week we will use the same basic problem as last week – numerical integration — to practice writing and using functions, to explore the effects of using double precision and to see some of the problems you can encounter in “simple” integrations. You will also learn something about programming — using functions.

1. Modify the programs you wrote last week to do numerical integrations using the left-hand rule, the midpoint method, and trapezoid rule so that the function to be integrated is evaluated by code in a separate file from the driving routine. Use this opportunity to fix up any remaining problems in your codes from last week, and fix up your table and plot of errors versus bin size if necessary.
2. Now modify the programs to use double precision variables. Repeat your evaluation of

$$\int_0^{1.5} x e^{-x} dx \quad (1.34)$$

using double precision, using the number of bins 2, 4, 10, 20, 50, 100, 1000, 10000, 100000 and 1000000. Make a new table and log-log plot of the error versus number of bins, and compare your results to last week’s results.

3. Now let’s try an integral that you can’t do with your pencil! After all, that’s the point of this course. Use the midpoint or trapezoid rule to calculate

$$\int_a^b e^{-x^2} dx \quad . \quad (1.35)$$

Estimate your error. (You have to think about how to do this, since you don’t know the correct value. Using the integrate function on your graphing calculator is not legal. After all, that is just a computer program written by someone else that does the integral numerically.)

(Assignment continues on next page)

4. Now write a new function for this integral.

$$\int_0^2 \sqrt{x} dx \tag{1.36}$$

Compile it together with the same main procedures you used in the previous part. Use numbers of bins 10, 20, 40, 80 and 160 (doubling each time!). Use other numbers of bins if you think that will be useful. To cut down on the work, you can use just the left-hand rule and midpoint rule for this integral. Again, study the dependence of the error on the number of bins. This integral is not quite as nice as the one we have been practicing on, and you should expect to have to think a little bit to understand your results, or even to get good results. Make a log-log plot to explore the dependence of error on step size. What is the slope? (*Hint: it is not two*). In your report, compare your answers to the exact answer.

(*Extra credit:* Can you explain why this integral doesn't converge as well as the first problem?)