10.001: Numerical Integration

R. Sureshkumar

1 Introduction

The topic for today's discussion is numerical integration of functions. In particular, we would like to obtain a numerical approximation to the integral $I(f)$, of a sufficiently smooth, integrable function $f(x)$, defined in the closed interval $a \leq x \leq b$. i.e.,

$$I(f) = \int_a^b f(x)dx. \tag{1}$$

This problem arises in many engineering applications. In some cases, the function may be too complicated to allow for an analytical evaluation of the integral. In some other cases, the 'function' itself may be available at a discrete set of points, either from experiment or from simulation.

2 Numerical Methods

Numerical methods are developed based on the results of mathematical analyses. For today's lecture, our understanding of elementary calculus suffices. The class of numerical integration techniques we discuss today can be compactly expressed as:

$$I(f) \approx \sum_{i=0}^{i=n} \alpha_i f(x_i). \tag{2}$$

i.e., we express the numerical approximation to the integral $I(f)$ as a sum of a series of finite number of terms, each term containing the value of the function at a given point x_i weighted by a coefficient α_i. One way to determine the coefficients α_i is to replace the original function $f(x)$ by an *interpolating* function which is simple enough, or in the terminology of numerical analysts, to use a *quadrature* technique.

Let us explore this idea a little further:

2.1 Simple Quadrature Rules

Consider the function $f(x)$ whose integral we wish to approximate over the interval $a \leq x \leq b$. The simplest of the interpolating functions we can think of is a constant function, i.e., we assume that for $a \leq x \leq b$, the function $f(x)$ is approximated by the interpolant function $C(x)$ which is simply given by $C(x) = f(a)$. We now proceed to compute the integral as the area enclosed by the rectangle formed by $y = 0$, $y = f(a)$, $x = a$, and $x = b$ in the $x - y$ Cartesian plane. This gives

$$I(f) \approx (b - a)f(a), \tag{3}$$

The interpolation rule given by Eq.3 is known as the *Rectangle Rule*.

Alternatively, we can use the constant interpolant $C(x)$, given by the value of $f(x)$ at the midpoint of the interval, i.e., $C(x)$ is now given by $C(x) = f(\frac{a+b}{2})$. This results in the *Midpoint Rule* given by

$$I(f) \approx (b - a)f(\frac{a + b}{2}). \tag{4}$$

Evidently, Eq.3 is exact for constant functions and Eq.4 is exact for constant or linear functions of the type $f(x) = \alpha x + \beta$ (Verify this, why so?). However, for the integrals of functions of higher order and complexity, Eqs.3 and 4 in general offer poor approximations. For instance, $\int_0^1 x^2 dx = 1/3$ when approximated with the midpoint rule gives a value of $1/4$. This error would amplify as the polynomial order is increased further.

This motivates us to look for higher order interpolant functions. We could use a linear interpolant, i.e., we approximate the function $f(x)$ as a straight line, say $y_1(x)$, which passes passes through the points $(a, f(a))$ and $(b, f(b))$ (What is the equation representing this line?). We now evaluate the integral as the area enclosed by the trapezoid enclosed by $y = 0$, $y = y_1(x)$, $x = a$ and $x = b$. This results in the *Trapezoidal Rule*, given by

$$I(f) \approx \frac{b - a}{2}[f(a) + f(b)]. \tag{5}$$

A more accurate quadrature rule can be obtained by using a quadratic interpolant function, which passes through the points $(a, f(a))$, $(b, f(b))$ and $((a+b)/2, f((a+b)/2))$. If we now evaluate the integral as the area enclosed by this parabola, the x axis, and the lines $x = a$ and $x = b$, we arrive at the Simpson's rule, given by

$$I(f) \approx \frac{b - a}{6}[f(a) + 4f(\frac{a + b}{2}) + f(b)]. \tag{6}$$

In a similar fashion, we can derive higher order quadrature rules through the use of polynomials of arbitrary degree as interpolants. The general class of quadrature rules which result from such exercises are called *Newton-Cotes* rules. However, this approach has many drawbacks from mathematical as well as computational points of view. Mathematically, the convergence of the right hand side of Eq.2 is not guaranteed as $n \to \infty$ even for infinitely differentiable functions. Computationally, for $n \geq 8$, we run into negative coefficients (α_i in Eq.2) resulting in unacceptably high *roundoff errors* (We will discuss roundoff errors in \$4.2). Hence, we seek an alternative approach, the one of using composite formula as explained below.

2.2 Composite Rules

I will illustrate this concept using the midpoint rule defined by Eq.4. The idea is to divide the interval $[a, b]$ into n subintervals, the ith subinterval being $[x_{i-1}, x_i]$. The length of the the ith subinterval, h_i, is then given by $h_i = x_i - x_{i-1}$. Notice that $x_0 = a$ and $x_n = b$. Now the contribution to the integral from the ith subinterval, according to the midpoint rule is given by $h_i f((x_{i-1} + x_i)/2)$. Hence, adding up the elemental contributions from each one of the n subintervals, we arrive at the *Composite Midpoint Rule*, given by

$$I_{CM} = \sum_{i=1}^{n} h_i f(\frac{x_{i-1} + x_i}{2}), \tag{7}$$

where the subscript CM denotes Composite Midpoint.

A similar procedure using Eq.5 and 6 results in the *Composite Trapezoidal* and the *Composite Simpson's* rules respectively, given by

$$I_{CT} = \sum_{i=1}^{n} \frac{h_i}{2}[f(x_{i-1}) + f(x_i)] \text{ for the composite trapezoidal rule} \tag{8}$$

and

$$I_{CS} = \sum_{i=1}^{n} \frac{h_i}{6}[f(x_{i-1}) + 4f(\frac{x_{i-1} + x_i}{2}) + f(x_i)] \text{ for the Composite Simpson's rule.} \tag{9}$$

3 Writing C Programs to Use the Composite Rules

We would now like to develop a C program which would compute the integral of a user defined function using the composite trapezoidal rule using equally spaced points, i.e.,

$h_i = h = (b-a)/n$ for each subinterval. The program I wrote for this purpose is given in /mit/10.001/Examples/ctrule.c. Let's analyze this little C program briefly. First of all, we would need an *algorithm* based on the composite trapezoidal rule given by Eq.8. A simple algorithm would be to compute the n elemental contributions (with h factored out) to the integral and add them up. This would result in $2n$ function evaluations and $2n$ additions. However, let's expand Eq.8 to examine the CT rule more closely. When we expand the series, we get

$$I_{CT} = \frac{h}{2}\left[f(a) + 2f(x_1) + 2f(x_2) + \cdots + 2f(x_{n-1}) + f(b)\right]. \tag{10}$$

The following algorithm can now be developed based on Eq. 10:

- Initialize Integral = $(f(a) + f(b))/2$.
- Within a *for* loop which runs from $i = 2$ to $n-1$, let integral = Integral + $f(x_i)$.
- Multiply the final answer with h.

This method requires only n function evaluations and n additions and hence would yield answers in half the time as compared to the original algorithm which required $2n$ function evaluations and $2n$ additions. This is the algorithm that has been implemented in the C program ctrule.c in the /mit/10.001/Examples directory.

Further analysis of program ctrule.c shows that the function to be integrated is defined separately in a subprogram: double f(double x). This allows for the *modularity* of the program, i.e., if we need to use this program to integrate another function, we only need to change the function definition in this subprogram.

We could write similar programs to implement the composite midpoint and the composite Simpson's rules. In fact we could incorporate in one program, all the three methods discussed above so that a given user can choose from any one of the given options. It is evident that this flexibility can be obtained through an *if .. then* construct discussed in the previous lectures.

4 Testing the C program

Once we have written and successfully compiled the program, we need to test its performance. The common practice is to solve a problem whose solution is known to us using

the new program. For instance, in the program ctrule.c mentioned above, we could put in various functions, such as $f(x) = 1, x, x^2, ..$ etc. and check whether the program yields an answer close to the known analytical result. I have used an exponential function which may be thought of as a polynomial of infinite order.

Well, thinking of good test cases for your programs is an art in itself. For instance, to test the program, we could try to evaluate the integral $\int_{-1}^{1} xexp(x^2)dx$ using ctrule.c since we know the analytical result is 0 (Why?).

4.1 Testing for Convergence

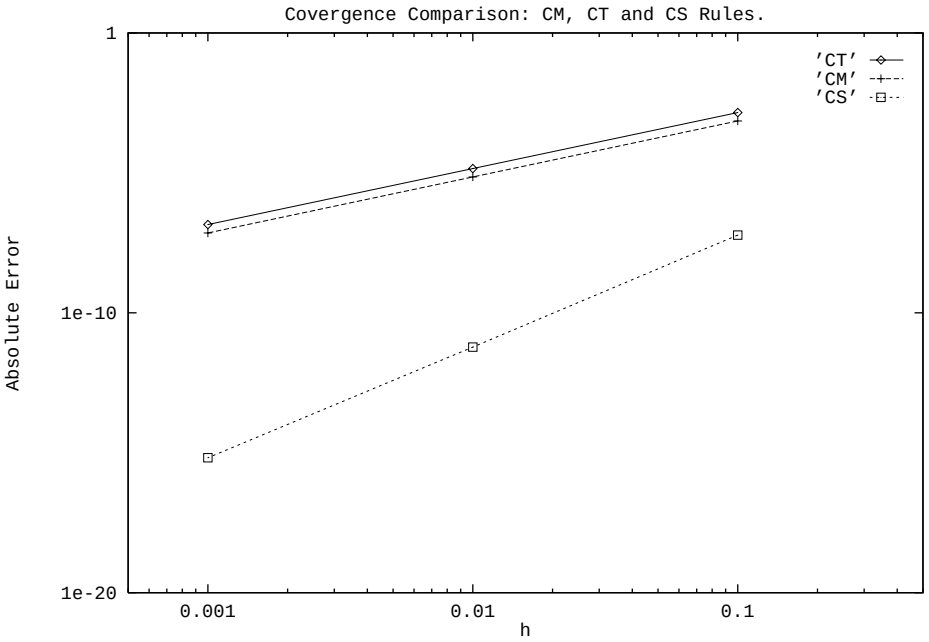
The concept of *convergence* is of cardinal importance in numerical analysis. This can be explained as follows with our current example. If we use any of the equations 7 - 9 to evaluate the integral, how fast does the numerically computed answer, say I_h, converge to the exact answer I ? To quantify this, we could define different measures of convergence. One of them is the absolute error, E_A, defined by $E_A = |I - I_h|$ and we will use this measure for the purpose of our discussion.

Let's evaluate from the program ctrule.c E_A for different values of h. If we do this and plot E_A vs. h in a log-log plot (see figure below), we see that the error decreases as h is decreased and on a log-log plot we get a straight line of slope 2. This error behavior is typical of *quadratically convergent* algorithms. It can be shown that both the CM and the CT rules are quadratically convergent, whereas the CS rule converges as the fourth-power (quartically, if you like) of the subinterval size h.

Once we have made such observations, we should try to reconcile them with relevant theoretical analysis. For instance, why does the trapezoidal rule converge as the second-power of h? A theoretical explanation of this requires the following theorem which arises primarily from the concept of the Taylor series expansion of a smooth function, and we simply state the theorem without proof and use it to see how we can explain the convergence characteristics of the trapezoidal rule.

Polynomial Interpolation Theorem: Let $f(x)$ be a function with $n+1$ continuous derivatives on an interval containing distinct points $x_0 < x_1 < ... < x_n$. If $p(x)$ is the unique polynomial of degree n or less such that $p(x_i) = f(x_i)$, then for any value of x in the interval $[x_0, x_n]$, we have

$$f(x) - p(x) = \frac{(x - x_0)(x - x_1)...(x - x_n)}{(n+1)!} f^{(n+1)}(z), \tag{11}$$



where $f^{(n+1)}$ denotes the $(n+1)$th derivative of f and z is some point in the interval $[x_0, x_n]$.

Let's apply this theorem for the trapezoidal rule. Here, for each one of the n subintervals, we use linear interpolation functions, so that the error in this approximation in the ith subinterval is given by $\frac{(x-x_{i-1})(x-x_i)}{2} f^{(2)}(z)$. So the absolute value of the error in the integral evaluation from this subinterval is given by (verify this result)

$$\begin{aligned} E_{A,i} &= \left| \int_{x_{i+1}}^{x_i} \frac{(x-x_{i-1})(x-x_i)}{2} f^{(2)}(z)dx \right| \\ &= \frac{(x_i - x_{i-1})^3}{12} f^{(2)}(z) \\ &= \frac{h^3}{12} f^{(2)}(z). \end{aligned} \tag{12}$$

Now, the if we add up the error contributions from all the n subintervals and replace $f^{(2(z))}$ by its maximum value in [a,b], say M_2, and utilize the fact that $nh = (b-a)$, we arrive at

$$E_A(h) \leq \frac{M_2}{12}(b-a)h^2, \tag{13}$$

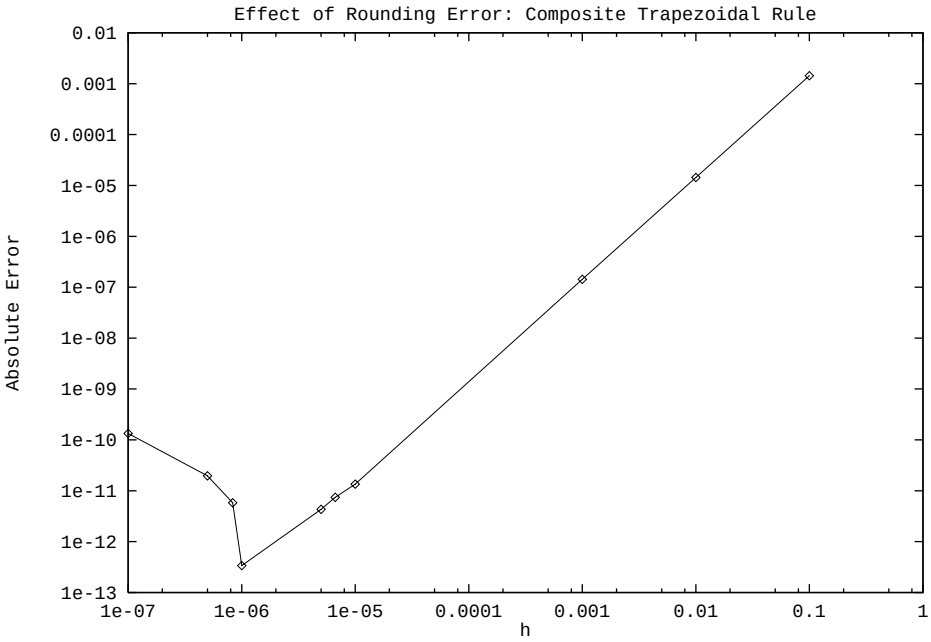
which correctly describes the error behavior we saw in the computations, see Figure 6. Similar analyses show that for the CM rule, $E_A(h) \leq \frac{M_2}{24}(b-a)h^2$ and for the CS rule,

$E_A(h) \leq \frac{M_4}{2880}(b-a)h^4$, where M_4 is the maximum value of the fourth derivative of the integrand function. Try to reconcile the numerical results of Figure 6 with the theoretically obtained error bounds for the three different integration rules.

4.2 Roundoff Error

The analysis in the previous subsection addressed the *discretization error* resulting from the numerical approximation. Another type of error which we have to guard our programs against is the *Roundoff Error*, arising purely due to the finite precision arithmetic of the computer. For instance, the number 0.0 can be represented in the computer as a decimal which has a finite number of digits after the decimal point as 0. In single precision, this could be for instance 0.00000000*** where *** denotes an integer over which we generally have little control. So if the first non-zero digit is greater than or equal to 5, the representation of 0 as a real number, correct to eight decimal digits on the computer is 0.00000001, or 10^{-8}. In double precision calculations, this representation is more accurate, up to 10^{-16}, and in more accurate quadruple precision, up to 10^{-32}.

We see from Eq.13 that the absolute error arising from the numerical discretization should go to 0 as $h \to 0$. However, the smaller h is, the larger the number of additions and function evaluations the computer has to perform and the rounding error incurred in each evaluation and addition adds up to a significant value. For the double precision calculations we employ, the rounding error in each operation is of the order 10^{-16}. Then if we do 10^8 evaluations/ additions, the total rounding error could be of the order 10^{-8}. This reasoning is substantiated by the numerical results presented in a figure above. We can see from there that for $h = 10^{-7}$, the error is greater than 10^{-10}.


Effect of Rounding Error: Composite Trapezoidal Rule
Absolute Error
h
0.01
0.001
0.0001
1e-05
1e-06
1e-07
1e-08
1e-09
1e-10
1e-11
1e-12
1e-13
1e-07
1e-06
1e-05
0.0001
0.001
0.01
0.1
1