Lecture Notes for

AMSC/CMSC 660, Scientific Computing I

By

Jian-Guo Liu and Simon P. Schurr

jliu@math.umd.edu, spschurr@math.umd.edu

Applied Mathematics and Scientific Computation Program

University of Maryland at College Park

September 2001

Chapter One - Scientific Computing Fundamentals

1 Introduction

Scientific Computing, traditionally known as Numerical Analysis, deals with the design and analysis of numerical algorithms that are implemented on a computer. We attempt to understand the theoretical fundamentals of numerical algorithms in addition to investigating how they might be best implemented computationally. This chapter describes the basic ideas of Scientific Computing, and how they are applied towards understanding and developing complex computational algorithms.

A general strategy for examining an algorithm is to replace a complex problem with a simple but non-trivial problem, and to apply the algorithm to this simplified problem. For example, consider two well-known algorithms for numerical integration, the Midpoint and Simpson's rules. These algorithms are applied to a relatively simple problem of evaluating the integral

$$I = \int_0^\pi f(x)dx \quad \text{where} \quad f(x) = x(2 + \sin\, kx) \quad \text{for positive integers } k \tag{1}$$

The integrand is a linear function superimposed with k wave structures. The exact solution is

$$I = \pi^2 + \frac{\sin(\pi k)}{k^2} - \frac{\pi\cos(\pi k)}{k}\,.$$

In practice the integrand $f(x)$ may be more complicated, and determining an exact value for I may not be possible. However we can attempt to estimate the value of I numerically (given a value of k) by using the two algorithms mentioned above. The interval of integration $[0,\pi]$ is discretized into N subintervals $[x_i\ , x_{i+1}]$ for $i = 0, 1, 2, \cdots, N-1$, where $x_i = ih$ and the grid spacing is $h = \frac{\pi}{N}$. For the Midpoint rule, we will require the midpoints $x_{i+\frac{1}{2}}$ of each interval $[x_i\ , x_{i+1}]$ where $x_{i+\frac{1}{2}} = (i+\frac{1}{2})h = x_i + \frac{h}{2}$. The integral is now estimated using the

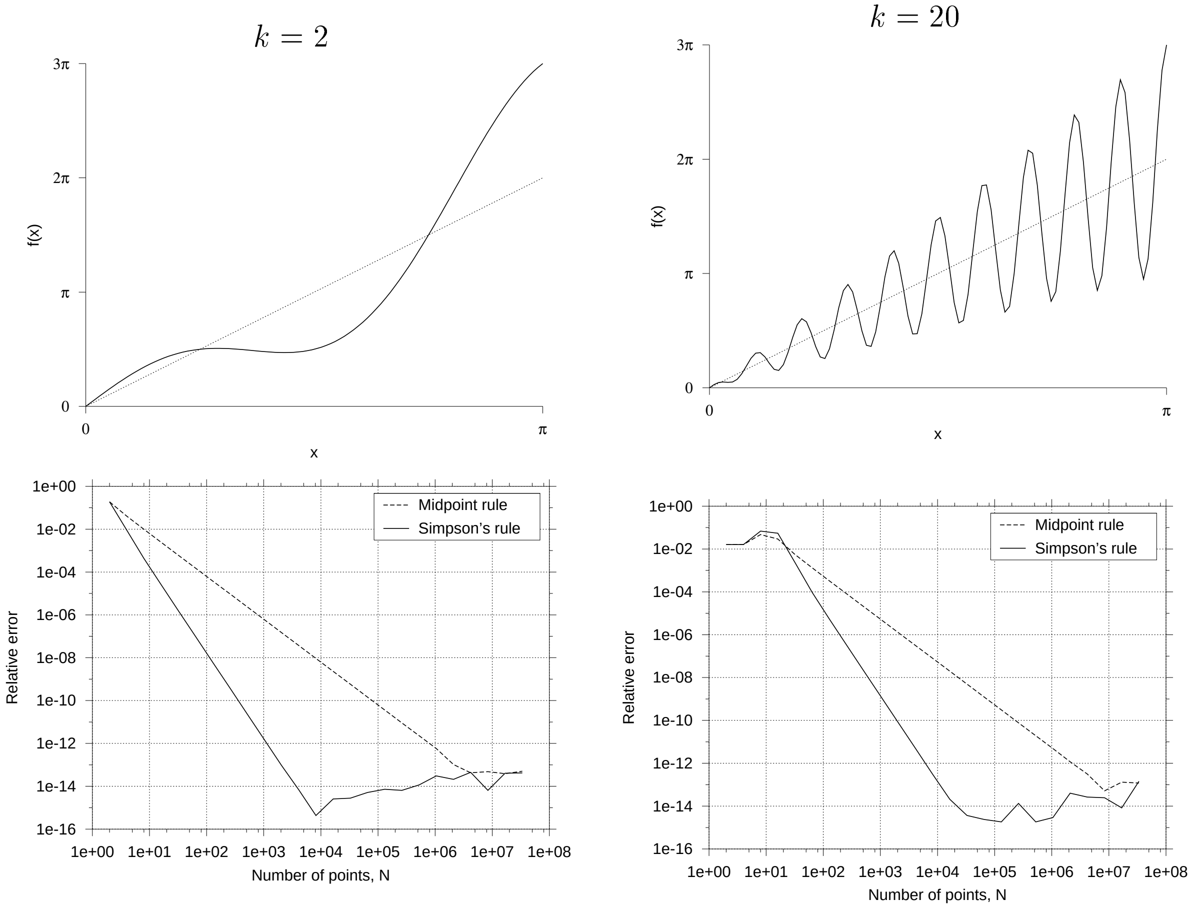


Figure 1: Errors involved in numerical integration for $k = 2$ and $k = 20$ (see Eq. 1)

two algorithms mentioned above:

Midpoint rule:

$$\int_{x_0}^{x_N} f(x)dx \approx h\left[f(x_{\frac{1}{2}}) + f(x_{\frac{3}{2}}) + f(x_{\frac{5}{2}}) + \cdots + f(x_{N-\frac{1}{2}})\right] \tag{2}$$

Simpson's rule:

$$\int_{x_0}^{x_N} f(x)dx \approx h\left[\frac{1}{3}f(x_0) + \frac{4}{3}f(x_1) + \frac{2}{3}f(x_2) + \cdots + \frac{4}{3}f(x_{N-1}) + \frac{1}{3}f(x_N)\right] \tag{3}$$

In Simpson's rule we require N to be even. For later reference we also give the Trapezoidal rule:

$$\int_{x_0}^{x_N} f(x)dx \approx h\left[\frac{1}{2}f(x_0) + f(x_1) + f(x_2) + \cdots + f(x_{N-1}) + \frac{1}{2}f(x_N)\right] \tag{4}$$

In order to understand the first two algorithms, the integral in (1) is evaluated using a series of grids with $N = 2, 4, 8, 16, \cdots$. Denote the relative error in the numerical solution by $E(h)$. (Recall that the absolute error is the magnitude of the difference between the exact answer and its approximation, and the relative error is the ratio of the absolute error to the exact answer). This quantity is shown in Fig. 1 for two values of k. The corresponding integrand $f(x)$ is also shown to better understand the behavior of the errors. It can be observed that in both cases the relative errors decreased with decreasing grid size (i.e. increasing values of N), and the slope of the error curves is constant over some range of grid sizes, say $h_0 > h > h_1$ (or $\frac{\pi}{h_0} < N < \frac{\pi}{h_1}$). This interval of values is said to be the interval of *clean accuracy*.

The slopes of the relative error curves show that the errors are proportional to h^2 for the Midpoint rule, and h^4 for Simpson's rule. (This behavior can be verified theoretically via Taylor expansions.)

A commonly used notation to denote order of magnitude of a quantity is the "big O" notation, i.e. the relative error can be written as $E(h) = O\big(h^p\big)$ if there exists a positive constant C such that

$$|E(h)| \leq Ch^p \text{ for all } h \text{ sufficiently small.} \tag{5}$$

In this case $E(h)$ is said to be of order p. Therefore we can write $E(h) = O\big(h^2\big)$ for the Midpoint rule and $E(h) = O\big(h^4\big)$ for Simpson's rule. The two rules are said to possess second and fourth order accuracy respectively in the interval of clean accuracy. (Strictly speaking, this interval is not an interval of 'sufficiently small h', as used in the above "big O" notation, since it does not include values of h that lie below h_1. So in this sense we have abused the notation slightly.)

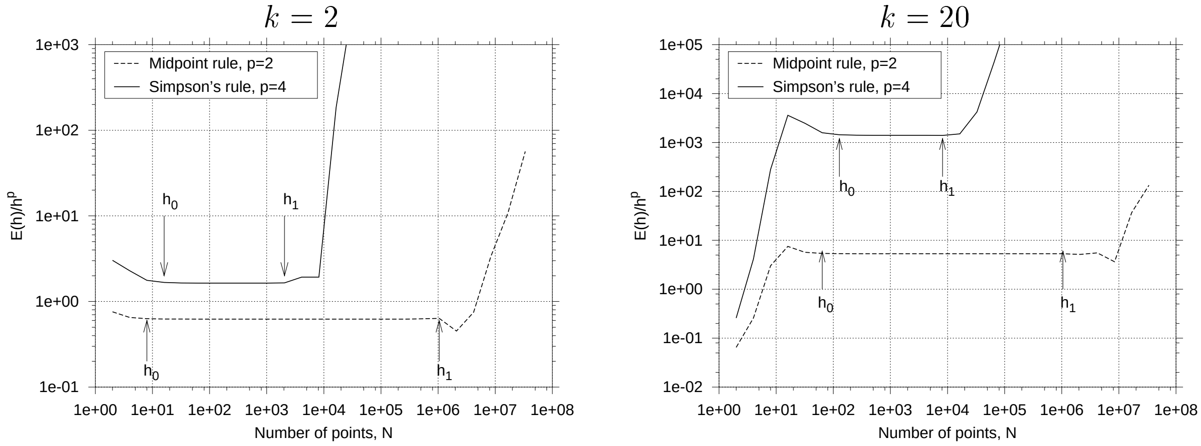


Figure 2: Range of grid sizes limited by resolution and round-off errors

Figure 2 shows the errors plotted in terms of $E(h)/h^2$ for the Midpoint rule and $E(h)/h^4$ for Simpson's rule. These quantities can be seen to remain constant (i.e. exhibit clean accuracy) over a continuous range of N values. The endpoints of this range that correspond to the values h_0 and h_1 are also noted in the Figure. For very fine grids, both methods do not exhibit the expected (theoretical) behavior. This is because for $h < h_1$ the comparatively large number of calculations performed causes round-off errors to become more dominant. This leads to a loss of accuracy. On the other hand, if the grid is too coarse, the relative error is also large due to a lack of resolution in the grid. In other words, the grid points x_i are spaced too far apart for the numerical method to be able to accurately take into account the behavior of the integrand $f(x)$. So the interval of clean accuracy can be thought of as set of h values where both the rounding and truncation errors are not too large.

In this example, when $k = 20$ the increased number of wave structures in the integrand means that a larger number of grid points is needed to 'resolve' the integrand. As a general rule of thumb, at least five points per wave structure in the function are necessary to obtain a reasonable interval of clean accuracy. So as k becomes larger, the grid size h must become smaller. It can be seen that the Midpoint rule, although only second order accurate, produces a much larger interval of clean accuracy than that for Simpson's rule. On the other hand, in some problems Simpson's rule may be more *efficient* because the Midpoint rule requires a finer grid (and therefore more function evaluations) compared to Simpson's rule to achieve a desired level of accuracy. This is the basic of idea of efficiency: an algorithm may be more efficient for a particular problem to be solved, although it may not be so in general.

We can also see that the grid size (which is a measure of the size of the problem here) is very important in numerical calculations. For a problem of estimating an integral in one dimension, the number of function evaluations required by the above methods is roughly $1/h$. However to perform a similar integration over 3 dimensions we will need roughly $(1/h)^3$ function evaluations. For example, if $h = 0.01$, we will be performing about a million function evaluations. It then becomes necessary to compromise between solution accuracy and speed of computation. In any case, as we have seen above, excessively small values of h lead to roundoff errors.

For a further example of how round-off errors can cause a loss of accuracy, consider the problem of approximating the derivative of a smooth function f at a point x by some function values in the vicinity of x. This technique is used in many areas of Applied Mathematics, and in particular when we are attempting to find the numerical solution to a Differential Equation. One possible approximation is a centered difference scheme:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h},$$

where h is relatively small. Essentially we are approximating the derivative of f at x by the slope of the line between the points $(x-h, f(x-h))$ and $(x+h, f(x+h))$. However the computer may not be able to compute the exact values of $f(x-h)$ and $f(x+h)$ (as real numbers) since it only has a finite storage capacity. Given an input $x-h$, the output of the machine will be a number $\tilde{f}(x-h)$, where $\tilde{f}(x-h) = f(x-h) + \epsilon_1$. ϵ_1 is the roundoff error, and $f(x-h)$ is the exact value which is unknown to the machine. Similarly we will obtain $\tilde{f}(x+h) = f(x+h) + \epsilon_2$. The roundoff errors will generally be bounded above by a number ϵ which is a measure of the accuracy of the machine. The magnitude $E(h)$ of the error that results from approximating $f'(x)$ by $\frac{f(x+h)-f(x-h)}{2h}$ can be examined by expressing $f(x \pm h)$ in terms of Taylor expansions. We have

$$
\begin{aligned}
f(x \pm h) &= f(x) \pm hf'(x) + \frac{1}{2!}h^2 f^{(2)}(x) \pm \frac{1}{3!}h^3 f^{(3)}(x) + \cdots \\
\implies E(h) &= \left| \frac{\tilde{f}(x+h) - \tilde{f}(x-h)}{2h} - f'(x) \right| \\
&= \left| \frac{(f(x+h)+\epsilon_2) - (f(x-h)+\epsilon_1)}{2h} - f'(x) \right| \\
&= \left| \frac{\epsilon_2 - \epsilon_1}{2h} + \frac{1}{6}h^2 f^{(3)}(x) + \cdots \right| \\
&\leq \frac{|\epsilon_2 - \epsilon_1|}{2h} + \frac{1}{6}h^2 |f^{(3)}(x)| + \cdots
\end{aligned}
$$

where higher order terms in the Taylor expansions are neglected, since higher order derivatives of f are assumed to be bounded, and h is assumed to be sufficiently small. In fact these assumptions allow us to bound the sum of all of the error terms (including $\frac{1}{6}h^2|f^{(3)}(x)|$) by Ah^2 (where A is a positive constant independent of h) in the neighborhood $(x-h, x+h)$. In the above expression for $E(h)$, the worst case occurs when ϵ_1 and ϵ_2 achieve their maximum magnitude ϵ, and are of opposite sign. We then have $E(h) = \frac{\epsilon}{h} + Ah^2$. The minimum value of this function occurs when $E'(h) = -\frac{\epsilon}{h^2} + 2Ah = 0$, so that the approximate value of h that minimizes $E(h)$ is $h^* = (\frac{\epsilon}{2A})^{1/3}$. For example in MATLAB, where $\epsilon \approx 10^{-16}$, we have $h^* \approx 10^{-5}$. To confirm that this value of h^* indeed gives a minimum $E(h)$, we observe that $E''(h^*) = 6A > 0$. A plot of $E(h)$ is shown in Fig. 3 for some sample values of A and ϵ. Also shown is a plot of $E(h)/h^2$ so that we can verify that there is a interval of clean accuracy.

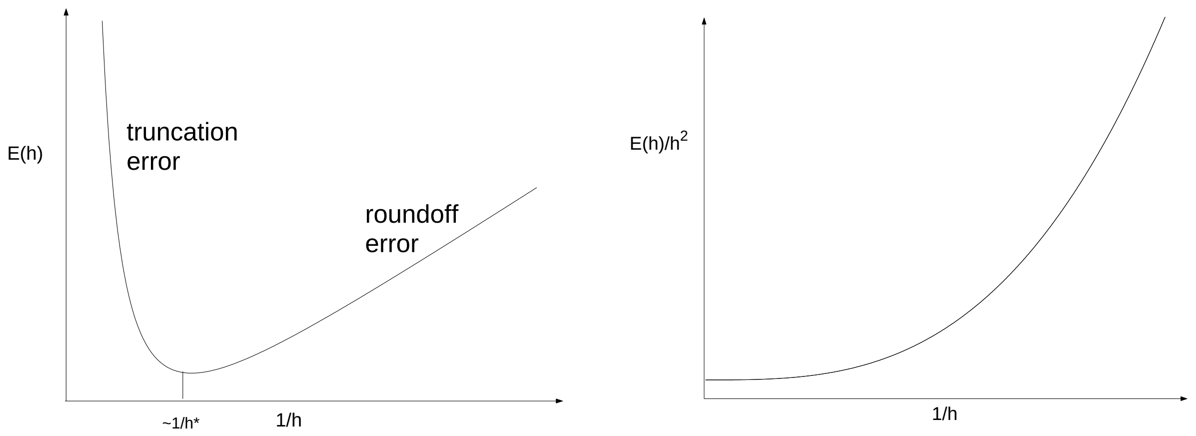


Figure 3: Plots of the error in approximating $f'(x)$ by $\frac{f(x+h)-f(x-h)}{2h}$

There are many desirable qualities of a numerical algorithm, e.g. Stability, Accuracy, and Efficiency. The stability of an algorithm refers to its ability to tackle a broad range of problems. However, a problem itself may be sensitive or ill conditioned, in which case even a stable algorithm may not give the desired results. The concept of ill-conditioning will be discussed in detail later. Accuracy and Efficiency are related and depend on the errors involved in numerically solving the problem. These errors may originate from several sources and cannot always be eliminated. A few sources of errors in numerical solutions are:

Modeling errors: e.g. in fluid dynamics, many problems are simplified by assuming the fluid to be incompressible and irrotational.
Empirical errors: e.g. post processing of experimental data, and empirical constants and coefficients used in a model, etc.
Statistical errors: e.g. confidence levels in Monte-Carlo type methods, which are statistical methods involving the use of random numbers.
Round-off errors: because real numbers must be represented in truncated form in a computer as floating point numbers (since a computer can only store a finite number of digits).
Truncation errors: e.g. errors introduced by finite difference approximations to derivatives, finite truncation of an infinite series expansion, etc.

In Scientific Computing, we are mainly concerned with the last three types of errors, which are numerical in origin. The statistical errors will be addressed Chapter 2 in the context of Monte-Carlo methods. The round-off errors will also be discussed later with a brief introduction to floating point numbers. The truncation errors are the most important factor governing the accuracy and efficiency of an algorithm. We will now learn how to determine and analyze some types of truncation error.

2 Truncation errors

Truncation errors are inherent characteristics of an algorithm, and therefore are an important consideration in the design or choice of improved algorithms. In this section the concept of an asymptotic error expansion, commonly used to analyze the nature of truncation errors, is explained using the previous example of numerical integration. Using the Midpoint rule (Eq. 2), we have

$$\int_{x_i}^{x_{i+1}} f(x)dx \approx hf(x_{i+\frac{1}{2}}), \quad h = x_{i+1} - x_i.$$

To examine the errors introduced in this approximation, we assume that the function $f(x)$ has sufficiently many high order derivatives in the interval $[x_i \ , x_{i+1}]$, so that it can be expanded using a Taylor series about the midpoint $x_{i+\frac{1}{2}}$ of the interval to give

$$\begin{aligned} f(x) \quad &= \quad f(x_{i+\frac{1}{2}}) + (x - x_{i+\frac{1}{2}})f'(x_{i+\frac{1}{2}}) + \frac{1}{2!}(x - x_{i+\frac{1}{2}})^2 f^{(2)}(x_{i+\frac{1}{2}}) \\ &\quad + \frac{1}{3!}(x - x_{i+\frac{1}{2}})^3 f^{(3)}(x_{i+\frac{1}{2}}) + \cdots \end{aligned}$$

We have used the notation $f^{(n)}(x) \equiv \frac{d^n f}{dx^n}(x)$. By integrating the above expansion term by term, we obtain

$$\int_{x_i}^{x_{i+1}} f(x)dx = hf(x_{i+\frac{1}{2}}) + \frac{h^3}{12} f^{(3)}(x_{i+\frac{1}{2}}) + \cdots .$$

For a small grid size h, the higher order terms in h are negligible, and the most dominant term in the *absolute* error is $\frac{h^3}{12} f^{(3)}(x_{i+\frac{1}{2}})$, and therefore the *relative* error is $O(h^2)$. So for sufficiently smooth functions, the error would decrease quadratically with grid size, and the algorithm is said to be second order accurate. As an exercise, verify that for Simpson's rule, the relative error is $O(h^4)$.

2.1 Asymptotic error expansion

The Taylor series expansion shown in the above example is a commonly occurring instance of an asymptotic error expansion, which is now described more formally. Suppose that the exact solution to a computational problem is given by the value A. Suppose that we attempt to estimate the value of A by some numerical method, which requires the discretization of an interval or region into small pieces, the size of which is measured by h. Let $A(h)$ denote the approximation to the answer A when the numerical method is applied to a grid with grid size h. Denote the error in the discretized, grid dependent solution by

$$E(h) = A(h) - A \,.$$

In theory, this approximation should become increasingly accurate as h becomes small, i.e.

$$E(h) \longrightarrow 0 \quad \text{as} \quad h \longrightarrow 0 \,.$$

and the error can be represented by an asymptotic expansion

$$E(h) = A_1 h^{p_1} + A_2 h^{p_2} + \cdots \qquad \text{for integers } p_i \text{ with } 0 < p_1 < p_2 < \ldots . \tag{6}$$

Equivalently, the approximation can be expressed in the form

$$A(h) = A + A_1 h^{p_1} + A_2 h^{p_2} + \cdots \,. \tag{7}$$

Note that Eq. 7 not only implies that $A(h)$ *converges* to the exact solution A with order p_1, but also that $A + A_1 h^{p_1}$ is a good approximation to $A(h)$, because the error in this approximation is $O(h^{p_2})$ for sufficiently small h. Similarly, the error in approximating $A(h)$ by $A + A_1 h^{p_1} + A_2 h^{p_2}$ is of order h^{p_3}, and is therefore an even better approximation, and so on. In many cases, as in the previous section, the powers and coefficients can be found using a Taylor series expansion. In practice, as noted in the previous section, the above expansions are only valid in the region of clean accuracy.

Typically, when h is small the leading error term is the most dominant term and the order of accuracy of the approximation is given by the leading power p_1. How small a h is small enough often depends on individual problem, in particular on the coefficients $A_1, A_2, \cdots$. Two methods may have the same order of accuracy but a different asymptotic error expansion. It should be noted that a method may be p_1-order accurate without having an asymptotic error expansion of the form Eq. 7. This is typical of Monte-Carlo type methods, which will be discussed later in Chapter 2. However, it is important to know whether a given method has an asymptotic error expansion. Two important applications of this are convergence analysis and Richardson extrapolation.

2.2 Convergence Analysis: Estimating Order of Accuracy

Convergence analysis is an useful test, based on asymptotic error expansions described above, to examine whether a given code, or the algorithm it is based on, is correct. It verifies not only that the error approaches zero with increasing grid discretization level, but also that it does so in a manner consistent with the preceding analysis. This is particularly useful in uncovering programming errors.

In the simple example shown in the beginning of this chapter, the exact solution to the problem was known. In practice one seldom writes a computer program to solve problems whose exact solution is known. However, it is useful to apply a given algorithm to a simple, yet nontrivial problem, whose solution is known. A trivial problem is one which is too simple to test and understand a given algorithm. If the integrand in $\int_{x_i}^{x_{i+1}} f(x)dx$ is a linear function of x, both the Midpoint and Simpson's rules would yield the exact solution. Likewise, for a quadratic or cubic polynomial, Simpson's rule would also yield the exact solution. (Why?) In these cases, there would be no truncation errors to be analyzed using a convergence analysis.

For simple problems with a known solution, the order of accuracy of the algorithm can be estimated using a *logarithmic* plot of the error, $E(h)$, as a function of h. Fig. 1 provides an example of this. The slope of the graphs give an indication of the order of accuracy of the algorithm. Also the coefficients in the asymptotic expansion can also be estimated from Fig. 2. For larger values of k, these coefficients have a higher value because of the increased number of wave structures.

If the truncation errors have an asymptotic error expansion, they can be expressed in the form

$$\begin{aligned} E(h_1) &= A_1 h_1^{p_1} + A_2 h_1^{p_2} + \cdots \\ E(h_2) &= A_1 h_2^{p_1} + A_2 h_2^{p_2} + \cdots . \end{aligned}$$

For small h_1, h_2, only the first term in these expansions is significant, so the ratio of these errors is approximately given by

$$\frac{E(h_1)}{E(h_2)} \approx \left(\frac{h_1}{h_2}\right)^{p_1} .$$

So the order of accuracy p_1 is given by

$$p_1 \approx \frac{\log(\frac{E(h_1)}{E(h_2)})}{\log(\frac{h_1}{h_2})} .$$

Some results are shown in Table 1 for the two integration algorithms given earlier. For large numbers of grid points, a loss of accuracy occurs because of round-off errors.

In practice there are three approaches to estimate the order of accuracy:

(1) We could use the solution on the finest grid (that is, the grid with the smallest value of h) as the best approximation to the 'exact' solution. Denoting this solution by A_∞, we have $|A_h - A_\infty|/h^{p_1} \approx C$ from Eq. 7, where C is a positive constant. In practice we must ensure that the finest grid is not too fine (i.e. outside the interval of clean accuracy.) Of course this interval is not normally known a priori, so some experimentation may be required.

Midpoint rule	
Grid points, N	Order
256	2.00004
512	2.00001
1024	2
2048	2
4096	2
8192	2
16384	2
32768	2
65536	2.00006
524288	1.98866

Simpson's rule	
Grid points, N	Order
32	4.02004
64	4.00498
128	4.00124
256	4.00031
512	4.00008
1024	3.99933
2048	3.99153
4096	3.77479
8192	4
16384	2.58496

Table 1: Order of accuracy estimated from errors

(2) Under some circumstances, the exact solution A to the problem may be knwon. We then have $|A_h - A|/h^{p_1} \approx C$ from Eq. 7, where C is a positive constant. As mentioned above though, in practice this will not be true for extremely small values of h because of the effects of round-off error. The plots in Fig. 2 illustrate this. This method is useful if we wish to investigate the accuracy of an algorithm without having a specific problem that we are required to solve. We can 'choose' several different exact solutions, and use the corresponding approximate numerical solutions to get some idea of the order of the algorithm.
For example, consider the following partial differential equation (PDE) in two variables x and y:

$$\Delta u \equiv \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y), \quad (x, y) \in \Omega = [0, 1] \times [0, 1].$$

The classical numerical method for solving this PDE would be to divide the region Ω into a grid of small squares, each of length h.

Now by *choosing* a smooth non-trivial function $u_e(x, y)$ that satisfies the PDE as well as some boundary conditions, we can explicitly calculate the corresponding RHS $f_e(x, y) = \Delta u_e(x, y)$, and the boundary values $g_e := u_e\big|_{\partial\Omega}$ (where $\partial\Omega$ denotes the boundary of Ω.) We can now apply a numerical scheme (whose exact nature will depend on how we discretize the Δ operator) to the problem

$$\Delta u_h = f_e \text{ on } \Omega - \partial\Omega, \qquad u_h = g_e \text{ on } \partial\Omega$$

to obtain the approximate solution u_h (that depends on the grid size h). After computing the exact solution u_e and approximate solution u_h at each grid point, we can check the accuracy of the numerical method. If it is of order p_1, we should have $\|u - u_h\|/h^{p_1} \approx C = \mathit{const}$.

(3) We could combine the results of three approximate solutions $A(h)$, $A(2h)$ and $A(4h)$ obtained using three different grid sizes h, $2h$ and $4h$ for some reasonably small value of h: By writing the approximate solutions for these three different grids in the form of Eq. 7:

$$\begin{aligned} A(h) &= A + A_1 h^{p_1} + A_2 h^{p_2} + \cdots \\ A(2h) &= A + A_1 (2h)^{p_1} + A_2 (2h)^{p_2} + \cdots \\ A(4h) &= A + A_1 (4h)^{p_1} + A_2 (4h)^{p_2} + \cdots, \end{aligned}$$

it can be shown by eliminating A and the leading error terms $A_1(\cdot)^{p_1}$ that

$$\frac{A(4h)-A(2h)}{A(2h)-A(h)} \approx 2^{p_1}. \qquad \text{Now solve for } p_1.$$

2.3 Richardson extrapolation

Richardson extrapolation is a method of increasing the order of accuracy of any given approximation, provided that such an approximation has an asymptotic error expansion of the form of Eq. 7. In its simplest form, two approximate solutions are computed for two grid sizes, e.g. $A(h_1)$ and $A(h_2)$, and then the leading error term $A_1(\cdot)^{p_1}$ is eliminated from each solution using a linear combination of these two solutions.

$$\begin{aligned}
A(h_1) &= A + A_1 h_1^{p_1} + A_2 h_1^{p_2} + \cdots \\
A(h_2) &= A + A_1 h_2^{p_1} + A_2 h_2^{p_2} + \cdots \\
&= A + \left(\frac{h_2}{h_1}\right)^{p_1} A_1 h_1^{p_1} + \left(\frac{h_2}{h_1}\right)^{p_2} A_2 h_1^{p_2} + \cdots .
\end{aligned}$$

By taking a linear combination of these two solutions, we obtain another approximation to the value of A, denoted by $A^{(1)}(h)$. For simplicity, define $c \equiv \frac{h_2}{h_1}$:

$$\begin{aligned}
A^{(1)}(h_1) &:= \frac{c^{p_1} A(h_1) - A(h_2)}{c^{p_1}-1} \\
&= A + \frac{c^{p_1}-c^{p_2}}{c^{p_1}-1} A_2 h_1^{p_2} + \frac{c^{p_1}-c^{p_3}}{c^{p_1}-1} A_3 h_1^{p_3} + \cdots .
\end{aligned}$$

The terms of order p_1 are cancelled, and the resulting extrapolated approximation has a higher order of accuracy, p_2, than the original approximation. To illustrate this, consider the Trapezoidal method (Eq. 4) applied to two grids with grid sizes $h_1 = h$ and $h_2 = 2h$ on the interval $[x_i\ , x_{i+2}]$:

$$\begin{aligned}
A(h) &= h\left[\frac{1}{2} f(x_i) + f(x_{i+1}) + \frac{1}{2} f(x_{i+2})\right] , \\
A(2h) &= 2h\left[\frac{1}{2} f(x_i) + \frac{1}{2} f(x_{i+2})\right] .
\end{aligned}$$

Given that the Trapezoidal rule is of second order, viz., $p_1 = 2$, then we have:

$$\begin{aligned}
A^{(1)}(h) &:= \frac{4A(h) - A(2h)}{4-1} \\
&= h\left[\frac{1}{3} f(x_i) + \frac{4}{3} f(x_{i+1}) + \frac{1}{3} f(x_{i+2})\right] .
\end{aligned}$$

This is indeed Simpson's rule (compare with Eq. 3) using the three grid points x_i, x_{i+1}, and x_{i+2}, and is fourth order accurate as mentioned previously.

3 Floating Point Numbers and Round-off Errors

Since computers are now used for the vast majority of numerical computations, we need to understand how they store and manipulate numbers. Of particular interest is how they handle real numbers, some of which can not be stored in their entirety in a computer since the storage capacity of a computer is only finite. Some real numbers (e.g. irrational numbers) need to be truncated before they are stored, and other numbers may be too large or too small to be represented by the machine. In other words, some real numbers can only be represented by a 'nearby' number (known as a *floating point number* or a *machine number*) in a computer, giving rise to round-off errors. While there are an infinite number of real numbers, the number of floating point numbers is finite, and is limited by the storage capabilities of the machine we are using.

Consider the expression -2.491×10^{-6}, which is one way a calculator can display the number -0.000002491. This notation consists of four parts: a negative sign, a mantissa ($m = 2.491$), an exponent ($e = -6$), and a base ($b = 10$).

The IEEE format for floating point numbers uses the binary system, i.e., base $b = 2$. If a floating point number is stored as a 32-bit 'word' (also called *single precision* representation), the first bit is the sign bit representing whether the number is negative or positive. The next 8 bits form the exponent, e, one of which is used for the sign of the exponent, and the remaining 23 bits form the mantissa m. In general, a floating point number is said to be *normalized* if it is written in such a way that the mantissa can be expressed as $0.d_1 d_2 \cdots d_n$ with $d_1 \neq 0$ and $1 \leq d_i < b$ for $i = 2, 3, \cdots, n$. So in base $b = 2$ we represent non-zero numbers in the following form:

$$\pm 0.1 d_2 \cdots d_{24} \;.\; 2^{\pm e_1 e_2 \cdots e_7} \;.$$

The values of $e_1, e_2, \cdots, e_7$ are required to satisfy $1 \leq e_i < b$, so in the case of $b = 2$, each e_i is 0 or 1. By convention, we use a 0-bit to represent a negative sign, and a 1-bit to represent a positive sign. An alternative representation which is often used for the exponent, is $e_1 e_2 \cdots e_8 - 127$, where each e_i is 0 or 1, and no sign bit is required.

The advantages of using normalized numbers are that each number is uniquely represented, and that there is no wastage in storing leading zeros. In binary systems, there is a further advantage of not having to store the leading digit d_1 since it will always be 1.

For example, the number -2752.5 in base 10 can be written in the form

$$\begin{aligned}
-2752.5 &= -(2^{11} + 2^9 + 2^7 + 2^6 + 2^{-1}) \\
&= -2^{12}\left(2^{-1} + 2^{-3} + 2^{-5} + 2^{-6} + 2^{-13}\right) \\
&= -2^{12}\left((.1)_2 + (.001)_2 + (.00001)_2 + (.000001)_2 + (.0000000000001)_2\right) \\
&= -2^{12}(0.1010110000001)_2 \\
&= -2^{10001100}(0.1010110000001) \quad \text{in base 2} \;.
\end{aligned}$$

The first 1-bit in the exponent 10001100 represents the positive sign in the exponent. The number is stored as a 32-bit word which looks like

$$\underbrace{\boxed{0}}_{\textit{sign}}\;\underbrace{\boxed{1\,|\,0\,|\,0\,|\,0\,|\,1\,|\,1\,|\,0\,|\,0}}_{\textit{exponent}}\underbrace{\boxed{0\,|\,1\,|\,0\,|\,1\,|\,1\,|\,0\,|\,0\,|\,0\,|\,0\,|\,0\,|\,0\,|\,1\,|\,0\,|\,0\,|\,0\,|\,0\,|\,0\,|\,0\,|\,0\,|\,0\,|\,0\,|\,0\,|\,0}}_{\textit{mantissa}}$$

Since each of the 32 bits can take on the value 0 or 1, there are 2^{32} possible floating point numbers which can be represented in the above manner. The distinctive features of the IEEE standard are the interpretations of the extreme cases:

The smallest positive number occurs when the exponent is $e = 01111111$ and the mantissa is $m = (0.100\cdots0)_2$. In base 10 this number is 2^{-128}, and is known as the Underflow Level (UFL). The largest number, corresponding to $e = 11111111$ and $m = (0.111\cdots1)_2$, is slightly smaller than 2^{127}. This is known as the Overflow Level (OFL).

There are some extreme cases where some real numbers are not able to be adequately described by floating point numbers. For example, the machine will output `inf` or infinity, to represent results such as $1/0$, or will output `NaN` or 'not-a-number', to represent indeterminate results such as $0/0$ or ∞/∞. The computer also produces `inf` when the result of an calculation is too large to be represented in the machine by the largest floating point number. Other invalid operations, like square-root or logarithm of a negative number produce `NaN`. If a calculation in a computer yielded an extremely small number whose absolute value was even smaller than the smallest possible machine number, the output would be 0 (we have underflow), though it should be noted that 0 can not be represented in the above floating point form (since d_1 must always be nonzero).

Now denote the floating point representation of a real number x by $\mathtt{fl}(x)$. An important measure of accuracy associated with a machine is the smallest positive floating point number ϵ such that $\mathtt{fl}(1+\epsilon) > 1$. This number is called the *machine epsilon* or *computer accuracy.* Of course in real arithmetic, there are no positive numbers ϵ satisfying $1+\epsilon = 1$, but since we are using floating point numbers, it is possible to find many sufficiently small $\epsilon > 0$ such that $\mathtt{fl}(1+\epsilon) = 1$. However at some point there will exist a positive value of ϵ such that $\mathtt{fl}(1+\epsilon) > 1$. To find the value of machine epsilon, we represent the real number 1.0 as $\mathtt{fl}(1) = 0.100\cdots00\ .\ 2^{+0000001}$ where the mantissa has 24 digits after the decimal point. Now the smallest floating point number that exceeds 1 is $\mathtt{fl}(1+\epsilon) = 0.100\cdots01\ .\ 2^{+0000001}$. An important detail we now have to consider, is the matter of whether a real number should be converted into a floating point number via *rounding* or *truncation.* If we choose rounding, then in base 2, given the above value of $\mathtt{fl}(1+\epsilon)$, the smallest positive value of $1+\epsilon$ is $0.100\cdots001\ .\ 2^{+0000001}$ where the mantissa now contains 25 digits after the decinmal point (hence 23 0-bits). So we can now use real arithmetic (rather than floating point arithmetic) to conclude that $\epsilon = (1+\epsilon) - 1 = 0.000\cdots001\ .\ 2^{+0000001} = 2^{-24}$ in base 10. (The mantissa still contains 25 digits, and therefore 24 0-bits.) It can be shown that if we use truncation instead of rounding, we obtain $\epsilon = 2^{-23}$ in base 10.

The value of machine epsilon is useful to know because it can be shown that it bounds the relative error when x is approximated by $\mathtt{fl}(x)$:

$$\left|\frac{\mathtt{fl}(x) - x}{x}\right| \leq \epsilon$$

Caution: it should be emphasized that **the machine epsilon ϵ is not the same as the smallest positive number (UFL).** For example, UFL does not depend on whether we employ rounding or truncation, but the value of ϵ does.

We now give without proof, two other useful results that give indications of the worst-case behavior when performing floating point arithmetic:

Theorem 1: $\mathtt{fl}(a*b) = (1+\delta)\, a*b$ for some operation $*$ such as $+, -, \times,$ or $/$, and $|\delta| \leq \epsilon$.

Theorem 2: If x and y are real numbers with $2^{-q} \leq (x-y)/x \leq 2^{-p}$, where p and q are integers with $0 \leq p \leq q$, then we will lose between p and q bits of accuracy in calculating the relative difference $(x-y)/x$.

The second result above introduces the idea of *cancellation errors.* Cancellation error is more severe than round-off because it results in the loss of more significant leading digits. An example of this occurring is when we wish to compute the difference of two close-by numbers. The most significant leading digits cancel each other, causing a loss of accuracy. For example, if the exact values of x and y are 103 and 100 respectively, then the exact value of $z = x - y$ is $z = 3$. However, if there are some errors present in the values of x and y - say for example they are calculated as $x = 102$ and $y = 101$ - then we obtain $z = 1$. Therefore, a relative error of less than 1% in x and y has caused a 67% error in z. This is a result of cancellation because of the fact that the leading digits in x and y (in this case the first two digits) cancel in the process of subtraction. The third, and the least significant digit in x and y determined the first and the most significant digit in z. Floating point arithmetic approximates numbers using a certain number of (binary) digits. If x and y are so close that they have many digits in common, then $z = x - y$ is determined only by the remaining digits that are different. Thus the accuracy of z is less than the accuracy of x or y.

Some remedies for cancellation: If possible ...
(1) Use Taylor expansions:
Example 1: When x is small, $\sin(x) \approx x$, so that the calculation of $\sin(x) - x$ can lead to cancellation errors. In this case it is possible to compute the Taylor expansion of $\sin(x)$:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} + \cdots ,$$

$$\text{so } \sin(x) - x = \frac{x^3}{3!} - \frac{x^5}{5!} + \cdots .$$

For small x, the terms in this series decrease rapidly, so by taking a small number of terms of this series, we should be able to obtain an accurate value of $\sin(x) - x$.

Example 2: Given a fixed x, evaluate

$$f(x) = \int_0^1 \cos\left(xt^2\right) dt$$

using a numerical integration algorithm such as the Midpoint or Simpson's rule. Verify the order of accuracy of the algorithm either using the exact solution or using different grids.

We can numerically calculate the 'exact' solution by integrating the Taylor expansion for $\cos(xt^2)$ term by term and including a sufficiently large number of terms. The Taylor series solution is given by

$$f(x) = \sum_{k=1}^{\infty} (-1)^k \frac{x^{2k}}{(2k)!(4k+1)} \tag{8}$$

A logarithmic plot of the relative error $E(h)$ versus the grid size for $x = 1$ and $x = 20$ shown in Fig. 4, reveals that the Midpoint rule is second-order accurate ($p = 2$), whereas Simpson's rule is fourth-order accurate ($p = 4$). (To verify this, calculate the slopes of the two plots in their respective regions of clean accuracy.) In this example, the finest grid was used as the 'exact' solution when calculating the relative errors.

In Simpson's rule with $x = 1$, the round-off errors dominate for grid sizes smaller than $h \approx 0.001$. However the Midpoint rule performs well over a wide range of grid sizes. For grid sizes larger than $h \approx 0.1$, Simpson's rule does not adequately resolve the integrand and shows slightly different behavior. When $x = 20$ roundoff errors are not a problem. To further illustrate this concept of resolution, the values of $E(h)/h^p$ is shown in Fig. 5 for the two numerical integration algorithms with $x = 1$ and $x = 20$. The corresponding integrands are also shown. Because the function $\cos\left(20t^2\right)$ has many oscillations in the range $0 \le t \le 1$, a finer grid is required to resolve the integrand, and $h > 0.1$ is unacceptable.

However if $x \gg 1$, the Taylor series expansion will not give correct results when we use it to estimate the exact solution. This is because the intermediate partial sums of the integrated series are outside the range of floating point representation, and a `NaN` result is obtained. However if x is large, the integral can be shown to possess the following *asymptotic expansion*:

$$f(x) \approx \sqrt{\frac{\pi}{8x}} + \frac{1}{2x}\sin(x) - \frac{1}{4x^2}\cos(x) + \cdots \tag{9}$$

The errors in Eq. 8 with a hundred terms and Eq. 9 using the first three terms are shown in Fig. 6 for the range $0.1 \le x \le 100$. For small x, the Taylor series approximation is highly accurate while the asymptotic approximation shows large errors. On the other hand, for larger x the asymptotic approximation shows relatively small errors, while the Taylor series approximation is less and less accurate because of the errors resulting from cancellation of large terms in the series. After $x \approx 40$, the Taylor series solution 'blows up' and we eventually get `NaN`. If we use intermediate values of x such as $1 \le x \le 20$, then the method of using the finest grid as the 'exact' solution seems to work well.

(2) Use Horner's rule (for evaluating polynomials - for example, partial sums of Taylor series):

$$\begin{aligned} p(x) &= a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots a_n x^n \qquad (10) \\ &= a_0 + x(a_1 + a_2 x + a_3 x^2 + \cdots + a_n x^{n-1}) \\ &= a_0 + x(a_1 + x(a_2 + a_3 x + \cdots + a_n x^{n-2})) \\ &\vdots \\ &= a_0 + x(a_1 + x(a_2 + x(\cdots + x a_n)\cdots)) \end{aligned}$$

Horner's rule is useful because it avoids $\infty.0$ type calculations which can occur when we are multiplying small coefficients a_i by large values of x^i in Eq. 10. In addition, Horner's rule

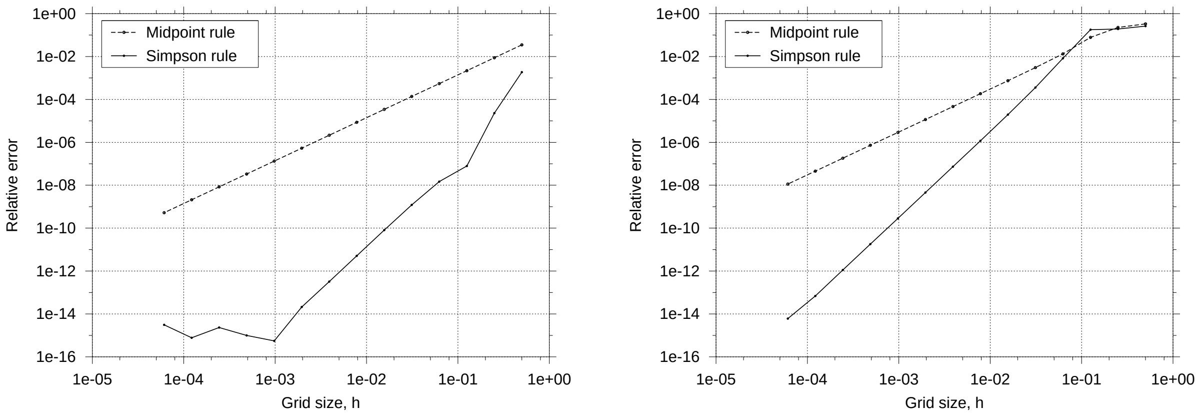


Figure 4: Verifying the order of accuracy of two numerical integration rules for $x = 1, 20$.

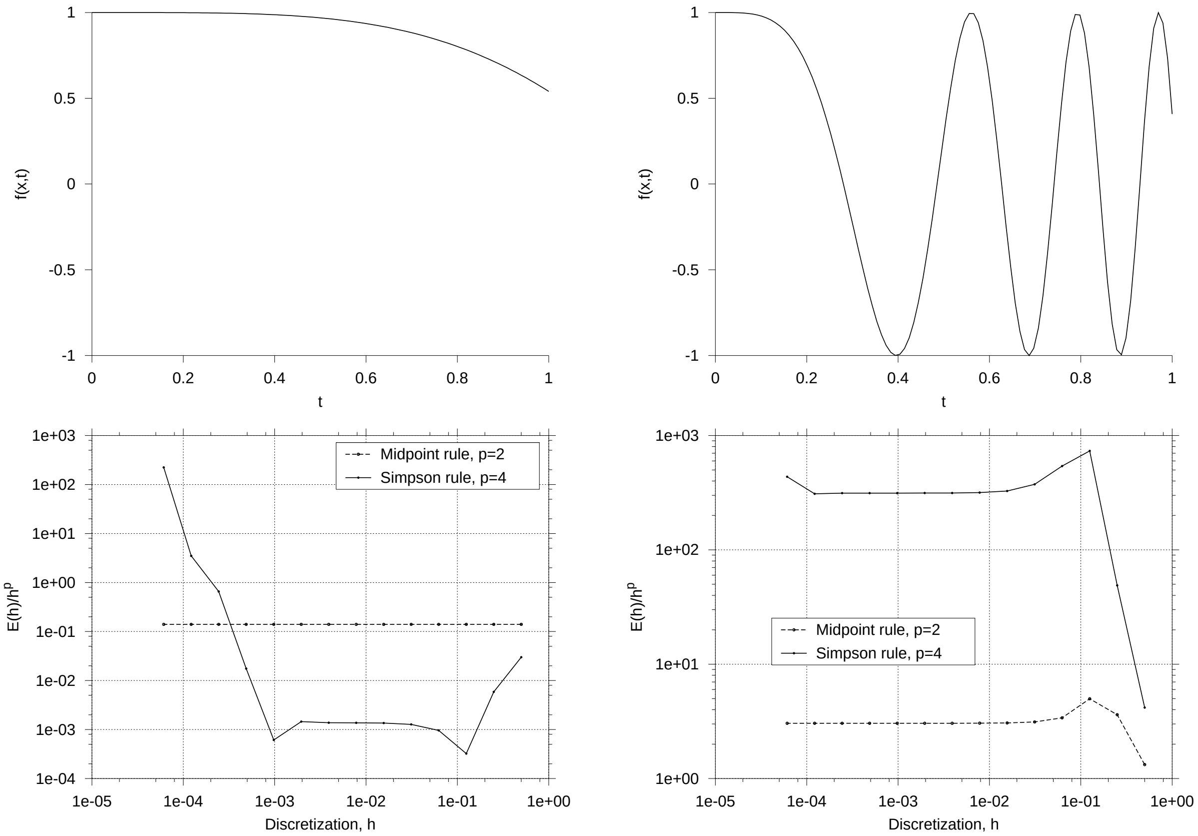
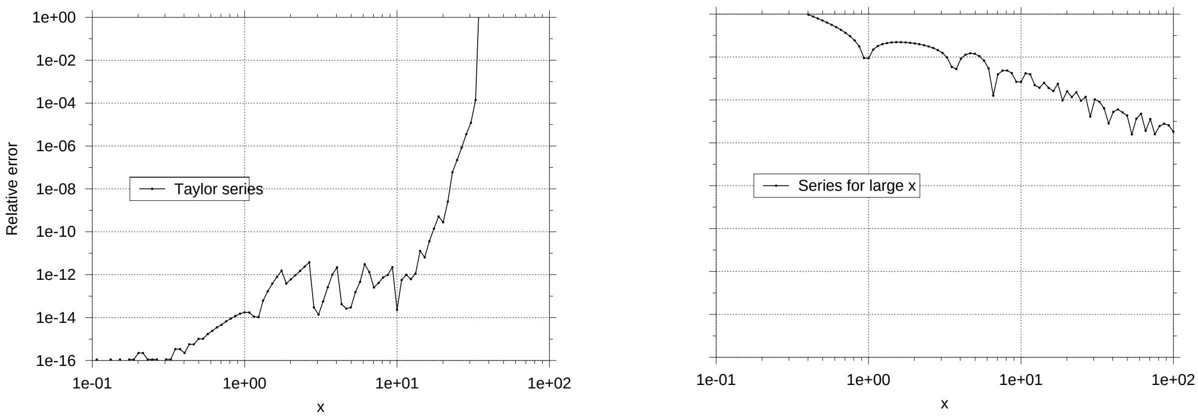


Figure 5: Two forms of the error expansion in Example Two for $x = 1$ and $x = 20$.



Valid for small x Valid for large x

Figure 6: Errors involved in the series approximations.

avoids the exponentiations that result from evaluating $p(x)$ term by term as in Eq. 10.

4 Stability and Conditioning

In many cases, inaccuracies in computed results are much larger than the round-off or truncation errors introduced during computation. This is because the errors may be 'amplified' by the algorithm, i.e. the errors from previous steps in the computation may accumulate and result in a total error which is much larger. Loosely speaking, we say that problem is *stable* if the solution depends continuously on the input parameters. This means that if 'small' changes are made to the input values, then the resulting changes in the output values will also be 'small'. If a problem is not stable it is said to be unstable. As we would expect, it is far more difficult to solve an unstable problem than a stable problem. However it is more accurate to classify stability on a continuous scale rather than to simply say that a problem is stable or unstable. This gives rise to the concept of a *condition number.*

The condition number of a problem is a measure of the sensitivity of its solution to small perturbations in the input parameters. It can be thought of as the ratio of the relative change in the solution to the relative change in the input. The condition number is significant in many problems because round-off errors in the input to a problem may lead to large changes in the solution. Such a problem is said to be *ill conditioned,* and cannot be correctly solved numerically because the round-off errors in the input parameters are inevitable. It is important to understand that **the condition number is a property of the problem to be solved itself, and not of the numerical algorithm employed to solve it.**

As an example, consider a square invertible system of linear equations $Ax = b$ with $A \in \mathbf{R}^{n\times n}$ and $b, x \in \mathbf{R}^n$. The unique solution is $x = A^{-1}b$. Now consider a perturbation Δb in the input vector b, and suppose that this perturbation causes a perturbation Δx in the solution, viz., $A(x + \Delta x) = b + \Delta b$, or $A\Delta x = \Delta b$. If we associate with each vector and matrix, a real numbered *norm* $\|\cdot\|$ that in some sense measures the 'size' of that vector or matrix, then we can use properties of vector and matrix norms to estimate the condition number $\mathrm{cond}(A)$ of the matrix A that we have used to produce the output x (via $x = A^{-1}b$).

The relative change in the solution is

$$\frac{||\Delta x||}{||x||} = \frac{||A^{-1}\Delta b||}{||x||} \leq \frac{||A^{-1}||||\Delta b||}{||x||} = ||A^{-1}||||A||\frac{||\Delta b||}{||A||||x||} \leq ||A^{-1}||||A||\frac{||\Delta b||}{||b||}$$

So the largest ratio $\frac{||\Delta x||}{||x||}/\frac{||\Delta b||}{||b||}$ of the relative change in the output to that of the input is $||A^{-1}||||A||$. We define this to be our condition number cond(A). Clearly this condition number is norm-dependent, but we will use the L_2 norm (also known as the 2-norm or Euclidean norm) since it is convenient for the following discussion. For a vector $x \in \mathbf{R}^n$ and a matrix $A \in \mathbf{R}^{n\times n}$, we define the vector 2-norm to be

$$||x||_{L_2} = \sqrt{{x_1}^2 + {x_2}^2 + \cdots + {x_n}^2}\ ,$$

where x_i is the ith component of x. The matrix 2-norm that is induced by this vector 2-norm is

$$||A||_{L_2} = \sup_{0\neq x \in \mathbf{R}^n} \frac{||Ax||_{L_2}}{||x||_{L_2}}\ .$$

In practice, calculating A^{-1} in the above formula for cond(A) can be computationally expensive, even if the size of A is only moderately large. Below we discuss a better method for estimating cond(A).

The following example illustrates the effect that the condition number can have on numerical calculations:

Example: For a given n, define a matrix $A = (a_{i,j}) \in \mathbf{R}^{n\times n}$ such that all elements of A are zero except for the diagonal elements $a_{i,i} = 1$ and the upper diagonal elements $a_{i,i+1} = 2$.
(a) Plot cond(A) for as many values of n as is possible.
(b) Let $x \in \mathbf{R}^n$ be such that

$$x_j = \begin{cases} 1 & \text{for even } j \\ 1/3 & \text{for odd } j \end{cases}$$

Compute $b = Ax$ by matrix multiplication. Now compute the solution y to the linear system $Ay = b$. Compare this numerical solution, y, with the exact solution, which is obviously x. Does the error in the numerical solution depend on n?

One way of computing cond(A) is to determine the eigenvalues, $\{\lambda_i\}_{i=1}^n$, of the matrix A^TA. It can be shown that A^TA has n positive eigenvalues, and there exists a full set of orthogonal eigenvectors that span $\mathbf{R}^n$. The condition number of A in the 2-norm is then given by $\sqrt{\lambda_{\max}/\lambda_{\min}}$. It is plotted against n in Fig. 7, and is seen to increase exponentially with n. For $n \geq 73$, the condition number cannot be represented in a machine that performs double precision (which is what we are using here), and therefore becomes `Inf`. This suggests that matrix inversion may not be possible on a computer for $n \geq 73$.

If the system of linear equations $Ay = b$ is solved for y using a simple back substitution algorithm using exact (real) arithmetic, we could obtain the exact answer. Therefore, any errors present must originate from the round-off errors. The 2-norm $||x - y||_{L_2}$ of the error $x - y$ is also shown in Fig. 7 for $n \leq 150$. The error also displays exponential growth with increasing matrix dimension. The condition number is plotted again for comparison. It is readily apparent that the error is directly proportional to the condition number: *error* =

$cond(A) \times 10^{-16}$. This shows that the round-off errors involved in the computations are $\approx 10^{-16}$ (which is very close to the value of machine epsilon). Clearly even an 'exact' solution algorithm like back substitution, is not suitable for large sparse matrices because the round-off errors can lead to significant errors in the final results.

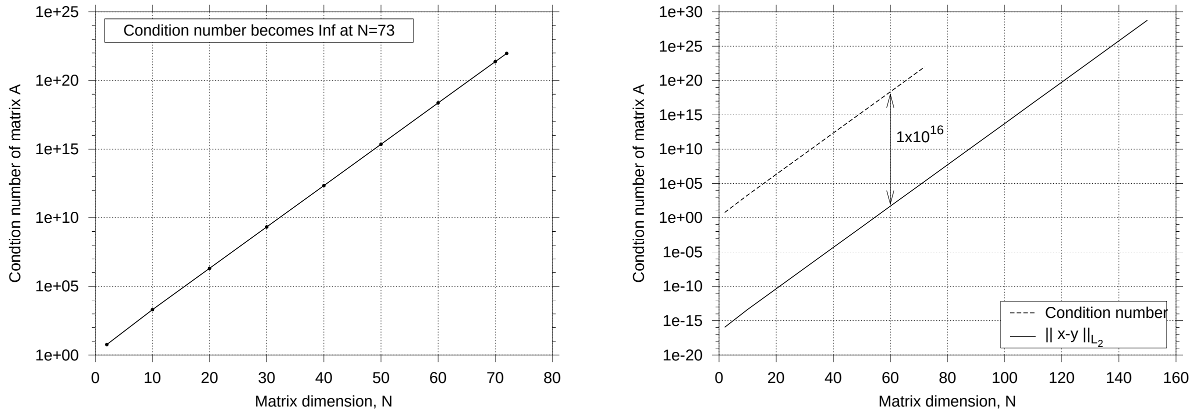


Figure 7: Condition number of a matrix and the round-off errors