Error in Numerical Methods

By Brian D. Storey

This section will describe two types of error that are common in numerical calculations: roundoff and truncation error. Roundoff error is due to the fact that floating point numbers are represented by finite precision. Truncation error occurs when we make a discrete approximation to a continuous functio. This document will provide some examples of each of these errors and discuss their importance.

1. Roundoff Error

A computer cannot store floating point numbers with perfect precision, instead the number is represented by a finite number of bytes. Most programming languages allow the coder to decide if a floating point numbers should be represented with 8 bytes (often called single precision) or 16 bytes (double precision). On some platforms designed for scientific computing single precision means 16 bytes and double precision is 32. MATLAB defaults to define all floating point numbers with 16 bytes. To see how many digits of precision you get with MATLAB you can try the following code.

```
x = 1;
for i = 1:10000
   x = x - 1e-17;
end
format long
x
```

The value for x created by this code will be one, even though the computer has enough precision to perform the operation `1 - 1e-13` (which is what this program is really doing). Change the line inside the `for` loop to be `x = x-1e-16` and you should see that the computer can perform this operation. In MATLAB, the roundoff error of floating point operations is about $\epsilon = 10^{-16}$. Precisely MATLAB has an implicit variable `eps` which is the 'distance from 1.0 to the next largest floating point number', according to the MATLAB help. Try the following:

```
format long
eps
```

As the somewhat contrived example above shows, there is a limit to the difference in magnitude between numbers that should be added. Addition between numbers that differ by 16 orders of magnitude is simply not possible (if you care about the differentiating the result - in many cases $1 + 10^{-16} = 1$ is acceptable). Adding numbers that vary by several orders of magnitude is possible, but some error will be introduced. When adding numbers of like magnitude, the roundoff error

is limited to the sixteenth decimal place, surely acceptable for most applications. As another example you could try the following MATLAB program:

```
x = pi;
x = x - 5/3*1e10;
x = x + 5/3*1e10;
x-pi
```

You will find that as you increase the exponent from 10^{10} the error will increase. Clearly a perfect computer would return $x - \pi = 0$.

Roundoff error can accumulate as your perform more and more calculations, for example in an iteration loop. Statistics tells us that the total roundoff error of the calculation would grow as $\sqrt{N}\epsilon$ if the roundoff error was completely random; ϵ is the last digit of precision, and N is the number of iterations. Some algorithms cause the roundoff error to preferentially accumulate in one direction causing the error to grow much faster. The precision given by MATLAB is high enough that roundoff is not a serious issue in most calculations.

We provide a very simple program that performs many iterations to look at the accumulatation of roundoff error. Inspection of this program simply shows that we start with $x = \pi$ and we compute a random number which we add and subsequently subtract from x. We repeat this iteration N times to see how the error changes with number of iterations.

```
x = pi;
for i = 1:N
   a = randn*100;
   x = x + a;
   x = x - a;
end
error = abs(pi-x);
```

You can easily see that on a perfect computer after each iteration the computer should return to $x = \pi$. However, roundoff error accumulates and the error grows as the number of iterations is increased. The result of the program for different values of N is shown in Figure 1. However, after many time steps $1p^7$ we see that the accumulated error is still quite small.

The result is only somewhat more dramatic if we write the program and use only single precision representation. The result for a comparable code written is shown in Figure 2. Here we see that roundoff error starts to influence the calculation in the fifth decimal place, which could become a problem for some calculations. It is clear that with MATLAB and double precision that roundoff error is not a serious problem.

Besides accumulating over time, roundoff error can influence a calculation when the algorithm is unstable. You can think of algorithmic stability just like mechanical stability. Theoretically, it is possible to balance a sharpened pencil on its point on a hard table. In practice we know this is impossible because the configuration is not stable. If you were able to balance the pencil for an instant any small perturbation to the pencil, including molecular fluctuations, will topple the pencil. Likewise algorithms can be unstable such that the roundoff error will act like small random fluctuations. Obviously it is best to avoid unstable algorithms. An interesting

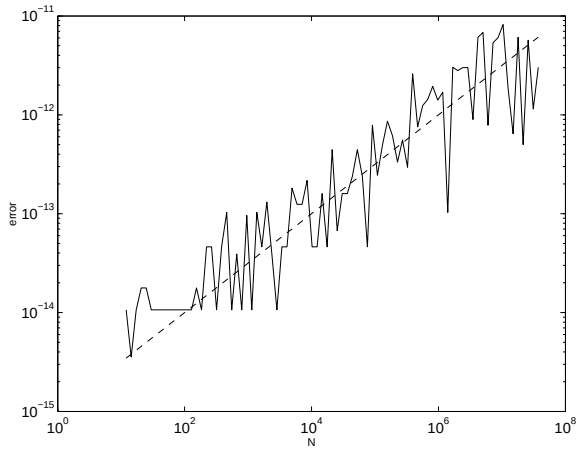

Figure 1. Roundoff error of a simple mathematical operation as a function of iteration number. The dashed line shows exact $\sqrt{N}$ behavior. The randomness is related to the randomness of roundoff error. This plot was generated using double precision arithmetic.

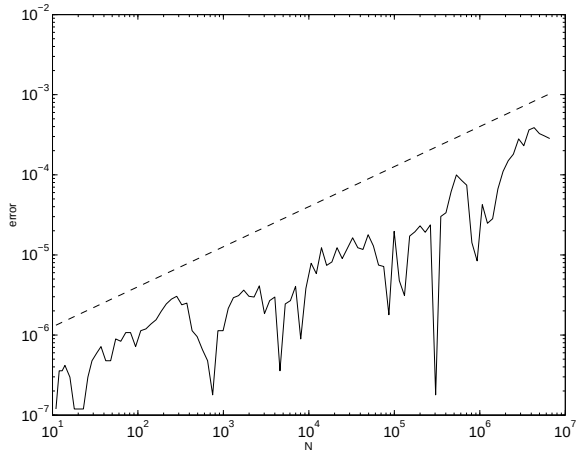

Figure 2. Roundoff error of a simple mathematical operation as a function of iteration number. The dashed line shows exact $\sqrt{N}$ behavior. The randomness is related to the randomness of roundoff error. This plot used single precision which truncates at $\epsilon = 10^{-8}$.

example is given in Numerical Recipes (1992). Define ϕ as

$$\phi = \frac{\sqrt{5}-1}{2}. \tag{1.1}$$

It is fairly easy to prove that to compute ϕ^n you can use an iteration equation

$$\phi^{n+1} = \phi^{n-1} - \phi^n \tag{1.2}$$

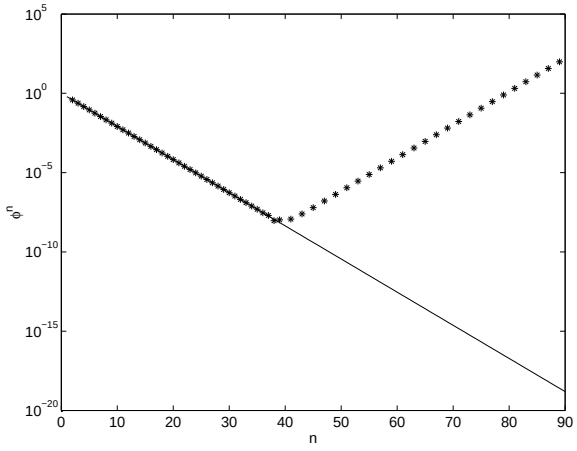

Figure 3. Stability properties of the iteration equation 1.2. The figure shows ϕ^n computed exactly (solid line) and from the iteration equation (data points). We see that the two agree for small number of iterations until the algorithm destabilizes.

knowing that $\phi^0 = 1$ and $\phi^1 = (\sqrt{5} - 1)/2$. The result of the iteration equation and the correct answer are shown in Figure 3. We find that the algorithm works for a small number of iterations, but that eventually the instability sets in and the iteration equation diverges from the true solution. We will prove later in the course why some algorithms are stable and others are not.

Roundoff error is not as serious as it once was. It is not a computational burden to perform all double precision arithmetic; in the 'dark ages' of computing double precision arithmetic was expensive and best avoided. In general roundoff error is much smaller than truncation error which is where we now turn.

2. Truncation error

Truncation error is caused by the fact that in scientific computing we are often making a discrete approximation to something continuous. The truncation error is a discretization error and is introduced by the approximations that we are making, not the computer architecture that we are using. Truncation error would exist on a computer that could perform perfect arithmetic.

Consider the following example. Suppose that we have a function that is sampled at discrete points in time. We would like to perform some calculations that need the derivative of this function. The easiest way to compute the derivative of a discrete function is the two-point slope. By this we mean you take two points close together and take the difference in the function divided by the difference in time between the two samples. This should recall the fundamental definition of the derivative that you have learned in calculus. Precisely, we will make the following numerical approximation to the derivative.

$$\left.\frac{df}{dt}\right|_{t_n/2+t_{n+1}/2} = \frac{f(t_{n+1}) - f(t_n)}{t_{n+1} - t_n} \tag{2.1}$$

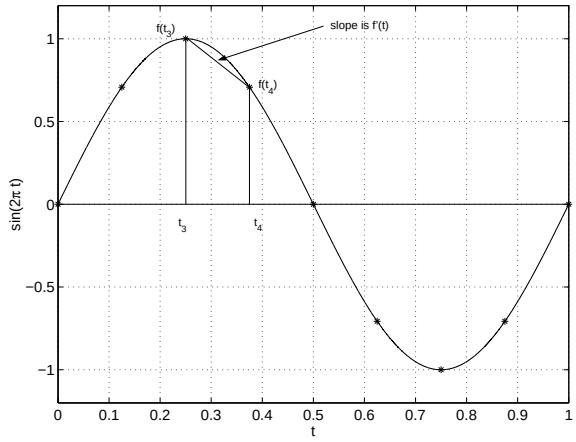

Figure 4. Graphical representation of the numerical derivative. The true function is $f(t) = sin(2\pi t)$, but we only have taken 8 samples at equally spaced intervals during one period of this sine wave. To compute the derivative, we compute the slope between adjacent sample points.

where t_n are the times that we have sampled the function. The left hand side of the above equation means - the derivative of the function f with respect to time, evaluated at the midpoint of the interval. The graphical representation of the numerical derivative approximation is shown in Figure 4.

We can test this accuracy of the approximation with a simple function where the derivative is easily calculated, in our example we will use $f(t) = sin(2\pi t)$. The error between the analytical derivative and the discrete approximation is plotted in Figure 5 and shows typical behavior. Usually the truncation error gets better as some power of the number of points, in this case N^2. Error that scales as N^3, N^4, etc is also possible and is caused by using different approximations to derivatives.

We will show later how you can derive the truncation error and create different discrete approximations that reduce its magnitude. Much of the research in numerical analysis is devoted to reducing the truncation error. The benefit of reducing truncation error is that you can get 'good' results for fewer grid points. Fewer grid points mean that bigger problems can be tackled via numerical simulation.

References

Press, Tuekolsky, Vetterling, & Flannery, 1992 *Numerical Recipes in C*, Cambridge University Press. See www.nr.com for free pdf version.

MATLAB product documentation. www.matlab.com

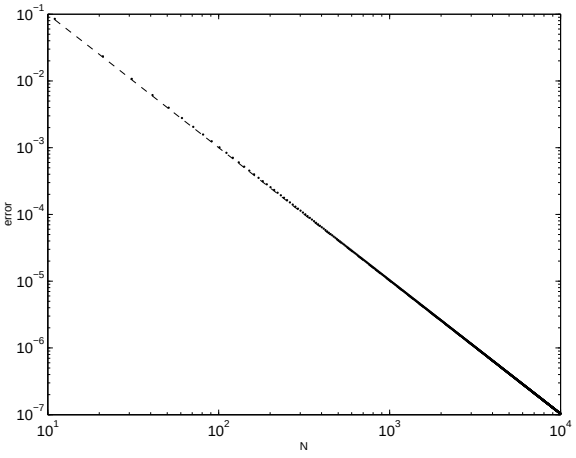

Figure 5. The error in computing the two point derivative of the function $sin(2\pi x)$ as the number of discrete points is increased. The error is plotted as data points and fit with the dashed line $error = 10/N^2$. Theoretical derivation of this error is quite easy and will be shown later in the course.