Computer Numbers and their Precision II
Errors and Uncertainties in Calculations

To err is human, to forgive divine.

- Alexander Pope

Learning goal: To understand how the ways computers store numbers lead to limited precision and how that introduces errors into calculations.

Learning objective

Computational and mathematical objectives:

- To understand that truncation or roundoff occurs when the mantissa of a floating point number becomes too long.
- To understand some of the consequences of the roundoff of floating point numbers.

Science model/computation objectives:

- To understand that just as laboratory experiments always have limits on the precision of their measurements, so too do computer simulations have limits on the precision of their numbers.
- To understand the difference between "precision" and "accuracy".
- Students will practice the following scientific skills:
    - Doing numerical experiments on the computer.

Activities

In this lesson, students will:

- Perform calculations on the computer as experiments to determine the computer's limits in regard to the storage of numbers.
- Perform numerical calculations to see how the rules of mathematics are implemented.
- Sum the series for the sine function to see the effect of error accumulation.

Where's Computational Scientific Thinking and Intuition Development

- Understanding that computers are finite and therefore have limits that affect calculations.
- Being cognizant that uncertainties in numerical calculations are unavoidable.
- Understanding how to obtain meaningful results even with a computer's limited precision.
- Understanding the range of numbers that may be necessary to describe a natural phenomenon.

Background

In the previous module, *Precision I,* we saw how the limited amount of memory used to store numbers on a computer leads to less-than-perfect storage of numbers. In particular, we saw that floating point numbers (those with exponents) are stored with limited precision and range for the exponent, while integers have an even more restricted range of values.

In this module we look at uncertainties that enter into calculations due to the limited memory that computers use to store numbers. Although the computer really has done nothing "wrong" in the calculations, these uncertainties are often referred to as "errors", which is what we shall call them from now on. Although the error in any one step of a computation may not be large, when a calculation involves thousands or millions of steps, then the individual small errors from each steps may accumulate and lead to rather large overall error. Accordingly, a computational thinker is careful when drawing conclusions from a calculation.

Types of Computer Errors

Whether you are careful or not, errors and uncertainties are a part of computation. Some real errors are the ones that humans inevitably make, while some are introduced by the computer. In general we can say that there are four classes of errors that may plague computations:

1. *Blunders or bad theory* Typographical errors entered with a program or data, running the wrong program or having a fault in reasoning (theory), using the wrong data, and so on. You can try to minimize these by being more careful, more thoughtful or smarter, but it seems that these things do not come easy to humans

2*. Random errors* Imprecision caused by events such as fluctuations in electronics or cosmic rays passing through the computer. These may be rare, but you have no control over them and their likelihood increases with running time. This means that while you may have confidence in a 20 second calculation, a week-long calculation may have to be run several times to ensure reproducibility.

3. *Approximation or algorithm errors* An *algorithm* is just a set of rules or a recipe that is followed to solve a problem on a computer, and usually involves some kind of an approximation so that the calculation can be done in finite time and with finite memory. Approximation or algorithm errors include the replacement of infinite series by finite sums, infinitesimal intervals by finite ones, and variable functions by constants.

For example, in mathematics, the sine function with argument in radians is equal to the infinite series:

$$\sin(x) = \mathrm{x} - \frac{\mathrm{x}^3}{3!} + \frac{\mathrm{x}^5}{5!} - \frac{\mathrm{x}^7}{7!} + \ \dots \tag{1}$$

Mathematically, this series converges for all values of x to the exact answer. An algorithm to calculate sin(x) might then be just the first N terms in the series,

(2) $$\sin(x) \simeq \sum_{n=1}^{N} \frac{(-1)^{n-1}\, x^{2n-1}}{(2n-1)!}.$$

For example, let's assume here that N = 2. We then have

(3) $$\sin(x) \simeq x - \frac{x^3}{3!}.$$

Because equations (2) and (3) do not include all of the terms present in (1), the algorithms provide only an approximate value for sin(x). Yet if the angle in radians x is much smaller than 1, then the higher terms in the series get progressively smaller and smaller, and we expect (2) with N > 2 to be a better algorithm than (3).

4. ***Round-off errors.*** Imprecision arising from the finite number of digits used to store floating-point numbers, and familiar from the preceding module. These ``errors'' are analogous to the uncertainty in the measurement of a physical quantity encountered in an elementary physics laboratory. The overall round-off error accumulates as the computer handles more numbers, that is, as the number of steps in a computation increases. For example, if your computer kept four decimal places, then it would store 1./3 as 0.3333 and 2./3 as 0.6667, where the computer has ``rounded off'' the last digit in 2./3. (We write 1./3 and 2./3 to ensure that floating point numbers are used in these operations, if integers were used, both operations would yield 0.)

For example, if we ask the computer to do as simple a calculation as 2(1/3)-2/3, it produces

(4) $$2 \times \frac{1.}{3} - \frac{2.}{3} = 0.6666 \quad - 0.6667 = -0.0001 \neq 0.$$

So even though the error is small, it is not 0, and if we repeat this type of calculation millions of times, the final answer might not even be small (garbage begets garbage).

Exercise: Round Off Error

Go to your calculator or computer and calculate $2 \times \frac{1.}{3} - \frac{2.}{3}$. Explain why the answer is not zero.

Relative Error & Subtractive Cancellation An error of an inch would be considered unacceptably large when building a kitchen cabinet, but rather small when measuring the distance to Mars. Clearly, it often is not the *absolute* error that matters but rather the *relative* error, and this is particularly true in computational science. To be more precise, we define relative error as

(5) $$\text{Relative Error} = \left|\frac{\textit{Correct Answer} - \textit{Computed Answer}}{\textit{Correct Answer}}\right|,$$

where the absolute value is included so that we do not have to worry about signs. An alternate, and maybe more practical, definition would use the computed answer in the denominator:

(6) $$\text{Relative Error} = \left|\frac{\textit{Correct Answer} - \textit{Computed Answer}}{\textit{Computed Answer}}\right|.$$

Of course we often do not know the correct answer, in which case we might have to make a guess as to what the numerator might be. (It often is pointless if not impossible to calculate an exact value for the error, since if you knew the error exactly, then there would not be any error!).

As an example, consider the calculation in Equation (3). If we used Equation (4) with the correct answer in the denominator, we would obtain an answer of infinity, which is both discouraging and not too useful – other than to tell us that it is hard to calculate 0 very precisely. If we instead use Equation (5), we would obtain a relative error

(7) Relative Error = $\left|\frac{0 - 0.0001}{0.0001}\right| = 1 = 100\%.$

This clearly tells us that our computed answer has an error as large as the answer, which is consistent with having 0 as an answer, even though that is not what we got as an answer.

The very large relative error occurring in this calculation occurs because we are subtracting two numbers that are nearly equal to each other and ending up with a result that is much smaller than the original numbers. This means that we are subtracting off the most significant figures of each number (0.66) and so are being left with the least significant figures (0.0067), or maybe with just some garbage left after truncation and round off. The process is known as *subtractive cancellation* and usually leads to large errors. This can be stated as the axiom, *subtracting two numbers to get a result that is much smaller is error prone.* Note that this cancellation occurs in the mantissa of floating-point numbers and not in the exponents.

As another example (which you should work out on your calculator or computer as you read), imagine that the correct answer to a calculation is x =0.12345678, while the computer returned an answer of x ≅ 0.12345679. This is not bad in that the two numbers differ by only one digit in the eight decimal place, and so we would say that the

(8) Relative Error = $\left|\frac{0.12345678 - 0.12345679}{0.12345678}\right| \cong 8 \times 10^{-8},$

which is a small number indeed. However, image now that the next step in the calculation requires us to compute x-y, where y = 0.12345670. The exact answer is

(9) x - y = 0.12345678 - 0.12345670 = 0. 00000008.

However, if we used the approximate value of x, we would obtain

(9) x - y = 0.12345679 - 0.12345670 ≅ 0. 00000009.

So even though we have a good approximation to x, the relative error in the calculation of x-y is large:

(10) Relative Error = $\left|\frac{0.00000009 - 0.00000008}{0.00000008}\right| = \frac{1}{8} = 0.125 = 12.5\ \%.$

As we have said, this is an example of the axiom, *subtracting two numbers to get a result that is much smaller is error prone.*

Error Accumulation in Series Summation

Let us now try to get an idea as to how round-off errors may accumulate when we perform a full calculation. We take Equation (2) as our algorithm

$$\sin(x) \simeq \sum_{n=1}^{N} \frac{(-1)^{n-1}\, x^{2n-1}}{(2n-1)!} \quad . \tag{2}$$

There are two sources of error when using this algorithm. Since we sum only *N* terms, rather than out to infinity, the terms not included would be the approximation error. We expect the approximation to get better as we use more terms in the series, and consequently for *the approximation error to decrease as N is made larger.* The second source of error, while not obvious from looking at (2), is round-off error. However if we write out (2) more explicitly as

$$\sin(x) \simeq x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040} + \ \dots , \tag{11}$$

We see that eventually the terms become very small (the factorial in the denominator grows more rapidly than the power in the numerator) and that the series should converge. However, we also see that for positive *x* the terms in the series alternate in sign and accordingly tend to cancel each other off. Indeed, when *x* is large and positive the cancellation can be so large that we are left only with round off error as the algorithm fails.

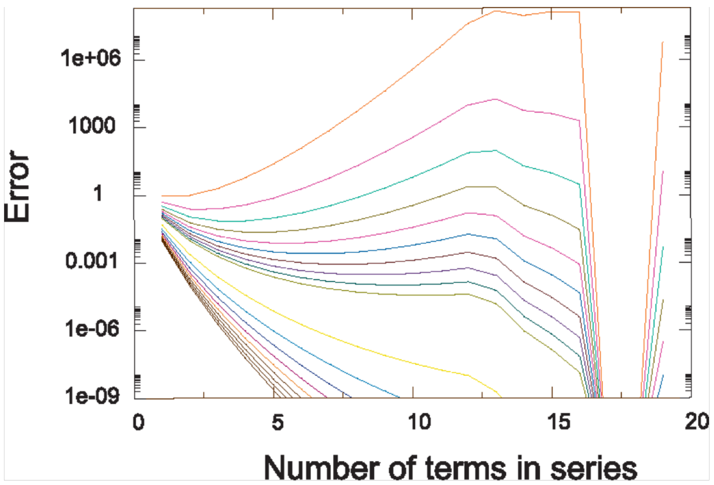

Figure 1 The relative error in the summation for sin(x) *versus* the number of terms in the series N. The values of x increase vertically for each curve. Note that the negative initial slope corresponds to decreasing error with N, that the dip corresponds to rapid convergence, and that the positive slopes for large N correspond to increasing error.

to

1. Write a program or use Excel or Vensim that calculates the series in Equation (2).
2. Evaluate the series for the maximum number of terms N = 5, 10, 12, 15, 17, 20.
3. For each value of N, evaluate the series for $x = \frac{\pi}{2}, \pi, \frac{5\pi}{2}$.
4. In each calculation, compare your computed value with the exact value (the one built into the program) and computer the relative error:

$$\frac{|series - \sin(x)|}{|\sin(x)|}$$

5. For each value of x, make a plot of the relative error versus the number of terms N. You should obtain a plot like Figure 1.

```
# Sine.py        power series for sin(x)
from numpy import *

x = math.pi/2.                    # initialization
N = 10
term = x
sum = 0.0
for i in range(2, (N + 1)/2):
sum  = sum + term
term = -term*x*x/(2*i-1.)/(2*i-2.)
print('N,sum = ', N,sum)
print("Enter and return a character to finish")
s = raw_input()
```

Summary and Conclusions

Round off error can accumulate and lead to inaccurate calculations. Once we understand how this occurs, it is possible to modify the computation to obtain accurate results.

Where's Computational Scientific Thinking

- Understanding that computers are finite and therefore have limits.
- Being cognizant that uncertainties in numerical calculations are unavoidable.
- Understanding how it is possible to work within the limits of a computer to obtain meaningful results.
- Understanding the range of numbers that may be necessary to describe a natural phenomenon.

References

[CP] Landau, R.H., M.J. Paez and C.C. Bordeianu, (2008), *A Survey of Computational Physics,* Chapter 5, Princeton Univ. Press, Princeton.

[UNChem] *UNC-Chapel Hill Chemistry Fundamentals Program, Mathematics Review,* http://www.shodor.org/unchem/math/; *Using Your Calculator,* http://www.shodor.org/unchem/math/calc/.