Round-off error

Algorithms like the Gaussian elimination algorithm do a lot of arithmetic. Performing Gaussian elimination on an n by n matrix typically requires on the order of $O(n^3)$ arithmetic operations. One obvious problem with this is that as the size of the matrix grows the amount of time needed to complete Gaussian elimination grows as the cube of the number of rows. That is a very serious issue that we will eventually get around to dealing with. For now, we focus on another issue, *round-off error*.

Round-off error is a side-effect of the way that numbers are stored and manipulated on a computer. Most commonly, when we use a computer to do arithmetic with real numbers we will want to make use of the computer's CPU's native hardware facilities for doing arithmetic. Doing arithmetic in hardware is by far the fastest method available. The trade-off for using the CPU's hardware for doing arithmetic with real numbers is that we have to use the CPU's native format for representing real numbers. On most modern CPUs, the native data type for doing real valued arithmetic is the IEEE 754 double precision floating point data type. This data type is a sequence of 64 bits divided into a sign bit, an 11 bit exponent, and a 52 bit mantissa. For our purposes, the most relevant fact about this structure is that it only allows us to represent a maximum of 16 decimal digits in a number. Any digits beyond 16 are simply dropped.

The main problem with this digit limit is that it is very easy to exceed that limit. For example, consider the following multiplication problem:

$$12.003438125 * 14.459303453 = 173.561354328684345625$$

The problem with this result is that although both of the operands fit comforably in the 16 decimal digit limit, the result has 21 decimal digits. To fit the result into a standard double we will have to simply *round off* or discard the last 5 digits. This introduces a round-off error to our calculation.

Round-off errors get more severe when we do division. For example, if we want to do

$$57.983/102.54 = 0.565467134776672518041739808855...$$

we see that doing a calculation with two five digit numbers produces a result with essentially an infinite number of decimal digits. Once again, that result will have to be rounded off to 16 decimal digits.

One of the most severe problems with round-off error occurs when we do simple addition involving two quantities with widely differing magnitudes. Here is an example:

$$1762.0345 + 0.0023993825 = 1762.0368993825$$

Note in this case that both of the operands have 8 decimal digits of precision, yet to fully represent the result requires 14 digits. If we were to arbitrarily round off the result to only 8 digits we would be in effect losing all but 2 of the original 8 digits from the second operand. This is a fairly severe round-off error.

Propagating round-off errors

All of the examples I showed above show that round-off error is a problem when we do just one arithmetic operation. This gets compounded when we have to do calculations requiring many arithmetic operations. Here is a rather dramatic example to demonstrate how this can become a real problem. The following Python program is designed to compute the summation

$$\sum_{n=1}^{10000000} \frac{1}{n} = 16.695311365859855$$

The program does this twice, once adding the terms from left to right and a second time adding the terms from right to left.

```
def forward_sum():
    sum = 0
    for n in range(1,10000001):
        sum = sum + 1/n
    return sum

def backward_sum():
    sum = 0
    for n in range(10000000,0,-1):
        sum = sum + 1/n
    return sum

print("Forward sum:  "+str(forward_sum()))
print("Backward sum: " + str(backward_sum()))
```

The results it returns are

```
Forward sum:  16.695311365857272
Backward sum: 16.695311365859965
```

Neither of these are right, and the forward sum is clearly worse. The reason for the problems with the forward sum has to do with the nature of the additions being done. When we do the sum in the forward direction we construct a cummulative total and store it in the variable sum. After the first few hundred terms the sum has magnitude of about 10.0 or so. At the same time, the terms we are trying to add to the running total keep shrinking. This attempt to add ever smaller terms to a growing running total is precisely the scenario that makes round-off error worse: adding two numbers of very different magnitude.

The situation gets better with the reverse sum. In that case we start by adding together the tiniest terms, so the running total starts out quite small. If we add small terms to a small running total we won't see as much round-off error. As the sum progresses the total grows, but so do the terms as we work our way from right to left. That keeps things more closely in balance and lessens the severity of round-off error.

Round-off error is a pervasive problem in numerical analysis. Almost any algorithm that requires that we conduct a large number of simple arithmetic steps is subject to this problem. Going forward from here we will need to be

aware constantly of this issue and will need to strive constantly to mitigate it. Fortunately, there are a few basic mitigation strategies that we can apply. Most of these involve restructuring the computation to avoid the most obvious sources of round-off error.

Round-off error in Gaussian elimination

The Gaussian elimination algorithm is a very simple algorithm that is easy to analyze for potential round-off error issues. The algorithm consists of the following steps repeated many (typically $O(n^3)$) times:

1. Construct a multipler $m_{j,i} = a_{j,i}/a_{i,i}$
2. Replace row A_j with A_j - $m_{j,i}\, A_i$

Note in particular the subtraction in step 2. When doing that subtraction we have to take care to ensure that the numbers being subtracted do not have greatly differing magnitudes. That can happen if the multiplier gets too big or small relative the numbers in the two rows. One way to cause trouble is for the *pivot value* $a_{i,i}$ to be a number close to 0. Dividing by a small number produces a multiplier which is too large, which in turn can cause problems in step 2.

Avoiding Round-off by selecting pivots carefully

We have seen that using a pivot value $a_{i,i}$ which is too small introduces a heightened danger of round-off error. The obvious fix for this is to avoid using pivots that are too small. We can manage to avoid small pivots by taking advantage of the fact that interchanging any two rows in an augmented matrix does not change the result. This leads to the following strategy, called *partial pivoting*:

Before using $a_{i,i}$ as a pivot, find the smallest p such that

$$a_{p,i} = max_{i \le k \le n} |a_{k,i}| \text{ and swap row } p \text{ with row } i.$$

This guarantees that each pivot we use is a large as possible to minimize the potential for round-off error.

A somewhat more sophisticated strategy is called *scaled partial pivoting*. In regular partial pivoting, we simply try to find the largest $a_{k,i}$ with $i \le k \le n$. On closer examination, what may matter more ultimately is how large each candidate $a_{k,i}$ is *relative to the other elements of its row*. To determine this, we can start by finding a *scale factor* s_i for each row in the original augmented matrix.

$$s_i = max_{1 \le j \le n} |a_{i,j}|$$

The scale factor is simply the largest element in the row. Whenever we swap rows the scale factors for the rows involved have to be swapped along with the rows. We use the scale factors to select candidate pivots by doing

Before using $a_{i,i}$ as a pivot, find the smallest p such that

$$\frac{a_{p,i}}{s_p} = max_{i \le k \le n} \left| \frac{a_{k,i}}{s_k} \right| \text{ and swap row } p \text{ with row } i.$$

In other words, we try to select the pivot that is largest *relative to the scale factor for its row.*

An even more aggressive strategy, called complete pivoting, searches the entire sub-matrix below and to the right of the potential pivot to find the largest number. Row and column interchanges are used to swap that number into the pivot location before proceeding. As the text points out, however, the added cost imposed by the extra searching and swapping rarely balances the benefit.