Math 515
Fall, 2012

Brief Notes on
Conditioning, Stability and Finite Precision Arithmetic

Most books on numerical analysis, numerical linear algebra, and matrix computations have a lot of material covering conditioning, stability and finite precision arithmetic. Here I summarize some of the important material that you should know.

1 Backward and Forward Error

Consider computing the quantity $y = f(x)$. If we can only compute an approximation of y, which we denote as $\hat{y}$, then we usually discuss two ways to measure the error associated with this computation.

Forward Error. The forward error is a measure of the difference between the approximation $\hat{y}$ and the true value y. That is,

$$\text{(absolute) forward error:} \qquad |\hat{y} - y|$$

$$\text{(relative) forward error:} \qquad \frac{|\hat{y} - y|}{|y|}$$

The forward error is a natural quantity to measure, but usually (since we don't know the true y) we can only get an upper bound on this error. Moreover, it can be very difficult to get tight upper bounds on the forward error.

Backward Error. Here we ask the question: *For what set of data have we actually solved the problem?* Specifically, we would like to find the smallest Δx such that

$$\hat{y} = f(x + \Delta x).$$

Here we mean that $\hat{y}$ is the *exact* value of $f(x+\Delta x)$. The value $|\Delta x|$, or $\dfrac{|\Delta x|}{|x|}$ is called the *backward error*.

Figure 1 illustrates the difference between forward and backward error in approximating $f(x)$.

Example. Suppose we want to compute $y = \sqrt{2}$, and we obtain the approximation $\hat{y} = 1.4$. Then

- Forward error: $|\Delta y| = |\hat{y} - y| = |1.4 - 1.41421\cdots| \approx 0.0142$.
- Backward error: Note that $\sqrt{1.96} = 1.4$, so $|\Delta x| = |2 - 1.96| = 0.04$.

2 Conditioning (or sensitivity of a problem)

A mathematical problem with input x and output $y = f(x)$ is:

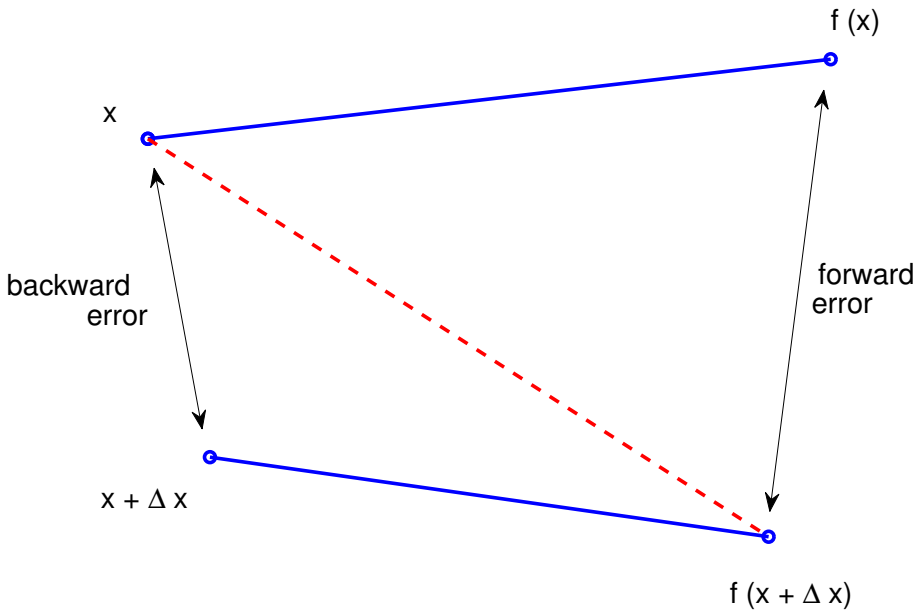

Figure 1: This figure shows illustrates the difference between forward and backward error.

- Well conditioned if small changes in x lead to small changes in y.
- Ill-conditioned if small changes in x can lead to large changes in y.

How can we measure conditioning?

$$\begin{aligned}
\text{condition number} &= \frac{\text{change in solution}}{\text{change in input}} \\
&= \frac{\left|\dfrac{\hat{y}-y}{y}\right|}{\left|\dfrac{\hat{x}-x}{x}\right|} \\
&= \frac{\left|\dfrac{\Delta y}{y}\right|}{\left|\dfrac{\Delta x}{x}\right|}
\end{aligned}$$

Note that the condition number can be approximated as

$$\begin{aligned}
\text{condition number} &= \frac{\text{change in solution}}{\text{change in input}} \\
&= \frac{|\Delta y|}{|\Delta x|}\frac{|x|}{|y|} \\
&= \left|\frac{f(x+\Delta x)-f(x)}{\Delta x}\right|\frac{|x|}{|f(x)|} \\
&\approx \left|\frac{xf'(x)}{f(x)}\right|
\end{aligned}$$

Example. The condition number for computing values of the function $f(x) = \ln x$ can be approximated as:

$$\text{condition number} \approx \left|\frac{xf'(x)}{f(x)}\right| = \left|\frac{x(1/x)}{\ln x}\right| = \left|\frac{1}{\ln x}\right|$$

Therefore, if $x \approx 1$, then the condition number is large. That is, evaluating $\ln x$ for $x \approx 1$ is an ill-conditioned problem; small changes in x can lead to large changes in $\ln x$.

Remark. Conditioning is a property of the *problem* and has nothing to do with the computer.

3 Stability (or sensitivity of an algorithm)

An algorithm is *stable* if the result it produces is relatively insensitive to errors made during the computations. That is, the result it produces is the exact solution of a nearby problem.

Remark. Stability of an algorithm does not guarantee an accurate computed solution. Accuracy depends on conditioning of the problem *and* stability of an algorithm. We will see this in our study of linear systems.

4 Computer Arithmetic

If you have never seen anything on computer (most importantly, floating point) arithmetic, machine epsilon, and discussions of roundoff error, then you should read chapter 2 in my undergraduate book. If you need a copy, please let me know and I will send you a pdf version.

One important source of error that can arise from using floating point arithmetic, can occur from *catastrophic cancellation*. This occurs when two nearly equal numbers are subtracted. For example, suppose we want to compute $x - y$, but because the computer cannot store numbers with infinite precision, and because possibly previous computations (which introduce roundoff errors) were needed to first compute x and y, we only have approximations $\hat{x} \approx x$ and $\hat{y} \approx y$. But we will suppose that $\hat{x}$ and $\hat{y}$ are good approximations of x and y. But we will assume that the

approximations are good; that is, assume

$$\begin{aligned} \hat{x} &\approx x\,, \quad \text{with (small) error} \quad \frac{|\hat{x}-x|}{|x|} \\ \hat{y} &\approx y\,, \quad \text{with (small) error} \quad \frac{|\hat{y}-y|}{|y|} \end{aligned}$$

Then the relative error in computing $x-y$ is:

$$\begin{aligned} \left|\frac{(\hat{x}-\hat{y})-(x-y)}{(x-y)}\right| &= \left|\frac{(\hat{x}-x)-(\hat{y}-y)}{x-y}\right| \\ &\leq \frac{|\hat{x}-x|}{|x|}\,\frac{|x|}{|x-y|} + \frac{|\hat{y}-y|}{|y|}\,\frac{|y|}{|x-y|} \end{aligned}$$

From this we can see that if $x \approx y$ then $|x-y|$ can be small compared to $|x|$ and/or $|y|$, and thus the error in computing $x-y$ can be large.

Example. Suppose we want to evaluate the function

$$f(x) = \frac{1-\cos x}{x^2}\,.$$

First note that for $x \neq 0$, $0 \leq f(x) < 1/2$. Suppose $x = 1.2 * 10^{-5}$. Then,

- The value of $\cos x$ rounded to 10 significant digits is

$$c = \cos x = 0.9999999999$$

- Now compute the numerator,

$$1 - c = 0.0000000001 = 10^{-10}.$$

- We then get our computed approximation of $f(x)$ by dividing:

$$\frac{1-c}{x^2} = \frac{10^{-10}}{1.44 * 10^{-10}} = 0.6944\cdots$$

Note that our computed approximation is not in the range $0 \leq f(x) < 1/2$. That means our computed approximation has no correct digits! We started with a 10 significant digit approximation of $\cos x$, and ended up with a result that has no correct significant digits! The problem here:

- $1-c$ is a subtraction of numbers of approximately the same value.
- When subtracted, the leading significant digits are "canceled".
- Only the very least significant digit is left.
- This results in a number $1-c$ whose error is the same size as c. Thus, the importance of the error in c is elevated.

It is not always possible to avoid catastrophic cancellation, but it in some cases, careful implementation can minimize the possibility of it occurring. In this example, we can use the trigonometric identity $\cos x = 1 - 2\sin^2(x/2)$ to rewrite $f(x)$ as

$$f(x) = \frac{1}{2}\left(\frac{\sin(x/2)}{x/2}\right)^2$$

Since this form for $f(x)$ does not contain any subtractions (or additions), we do not risk catastrophic cancellation.

As an exercise, you might try using MATLAB to evaluate both forms of $f(x)$ in the above example, for small values of x, and see if you can observe the fact that catastrophic cancellation produces poor results with the first form of $f(x)$, but that the second form computes accurate results for even very small values of x.

5 FLOPS

A FLOP is a *floating point operation*; that is, a multiplication, division, addition or subtraction. To assess cost of an algorithm, we often count the number of FLOPS it requires. Usually it is easier to separate the count of multiplications and divisions with the count of additions and subtractions. For example, consider the following piece of MATLAB code, which corresponds to the major part of work to compute the reduced QR factorization using the modified Gram-Schmidt method.

```
for k = 1:n
  R(k,k) = norm(A(:,k));
  Q(:,k) = A(:,k)/R(k,k);
  for j = k+1:n
    R(k,j) = Q(:,k)'*A(:,j);
    A(:,j) = A(:,j) - Q(:,k)*R(k,j);
  end
end
```

Suppose we want to count the number of multiplications required by this code. Then observe that:

- Computing `norm(A(:,k))`, the norm of a column vector with m entries, requires m multiplications.
- Computing `A(:,k)/R(k,k)` requires m divisions, which is equivalent to m multiplications.
- Computing the inner product `Q(:,k)'*A(:,j)` requires m multiplications.
- Computing `Q(:,k)*R(k,j)` requires m multiplications.

We now need to add up all of the multiplications each time we go through the loops. Specifically, the total number of multiplications is:

$$\text{multiplications} = \sum_{k=1}^{n}\left(m + m + \sum_{j=k+1}^{n}(m+m)\right) = \sum_{k=1}^{n}\left(2m + \sum_{j=k+1}^{n}2m\right)$$

We can use summation formulas to get the final count, but a simple way to approximate the total, which gives the correct highest order term, is to use integration. That is, replace the summations with integration:

$$\text{multiplications} \approx \int_1^n \left(2m + \int_{k+1}^n 2m \, dj \right) dk = \cdots = mn^2$$

Similarly, we can count additions:

- Computing `norm(A(:,k))`, the norm of a column vector with m entries, requires $(m-1)$ additions.
- Computing `A(:,k)/R(k,k)` requires 0 additions.
- Computing the inner product `Q(:,k)'*A(:,j)` requires $(m-1)$ additions.
- Computing `A(:,j)-Q(:,k)*R(k,j)` requires m additions.

That is, the total number of additions is:

$$\begin{aligned} \text{additions} &= \sum_{k=1}^{n} \left(m - 1 + \sum_{j=k+1}^{n} (2m-1) \right) \\ &\approx \int_1^n \left(m - 1 + \int_{k+1}^n (2m-1) dj \right) dk \\ &\approx mn^2 + \text{lower order terms} \end{aligned}$$

Thus the total number of FLOPS for the code is $2mn^2+$ lower order terms.