Week 2: Friday, Sep 4

Conditioning

At the end of the last lecture, we introduced the *condition number* $\kappa(A) = \|A\|\|A^{-1}\|$ that characterizes the relationship between a small relative perturbation to A and a small relative perturbation to the matrix-vector product Ax. That is, if $\hat{y} = \hat{A}x$ is a perturbation of $y = Ax$, where A is invertible, then we found

$$\begin{aligned}\frac{\|\hat{y} - y\|}{\|y\|} = \frac{\|(\hat{A} - A)x\|}{\|y\|} &= \frac{\|(\hat{A} - A)A^{-1}y\|}{\|y\|} \\ &\leq \|\hat{A} - A\|\|A^{-1}\| = \kappa(A)\frac{\|\hat{A} - A\|}{\|A\|}.\end{aligned}$$

This inequality is tight, in a sense that we will see more clearly in a future lecture.

There are condition numbers for other problems, too. In general, the condition number relates the size of a relative change to a problem to the size of a relative change in the solution. We say a problem is *ill-conditioned* when the condition number is large, where "large" depends on the setting.

Floating point

Why do we study conditioning of problems? One reason is that we may have input data contaminated by noise, resulting in a bad solution even if the intermediate computations are done perfectly accurately. But our concern in this class is less with measurement errors and more with numerics, and so we will study condition numbers with an eye to floating point error analysis.

In the IEEE floating point standard implemented on most machines, there are three types of numbers:

1. *Normalized* numbers have the form

   $$(-1)^s \times (1.b_1\, b_2\, \ldots\, b_p)_2 \times 2^e,$$

   where s is the sign bit, $b_1\, b_2\, \ldots\, b_p$ are the fraction bits in the significand, and $e_{\min} \leq e \leq e_{\max}$ is the exponent. For single precision, $p = 23$,

$e_{\min} = -126$, and $e_{\max} = 127$. For double precision, $p = 52$, $e_{\min} = -1022$, and $e_{\max} = 1023$.

Think of normalized numbers as the binary equivalent of ordinary scientific notation.

2. If we only had the normalized numbers, we would be missing an important number from our system: zero! But we would lose other desirable properties, too. For example, suppose I compute $(1.01)_2 \times 2^{e_{\min}} - (1.00)_2 \times 2^{e_{\min}}$ in floating point. The true answer, $2^{e_{\min}-2}$, is closer to zero than to any normalized floating point number. If we want always to return a floating point result which is close as possible to the true result, then either we must return zero (in which case $x - y = 0$ and $x = y$ would no longer be equivalent in floating point), or we need something different.

   The somethings different that solves these problems are *denormalized* numbers, which have the form

$$(-1)^s \times (0.b_1\, b_2\, \ldots b_p)_2 \times 2^{e_{\min}}.$$

   Unlike the normalized numbers, the denormalized numbers don't get ever denser as they go to zero; rather, they evenly fill in the gap between the smallest postive and negative normalized numbers. When an arithmetic operation produces a denormalized number, we call it an *underflow.*

3. *Infinity* (positive or negative) can be produced either when the "true" answer is infinite (e.g. 1/0) or when the result of an operation is finite, but so large that it cannot be represented in the ordinary exponent range (an *overflow*).

4. *NaN* (Not-A-Number) is produced on invalid operations (e.g. 0/0).

The default rule for basic operations like addition, subtraction, multiplication, division, and square root is that they should produce the *correct result, correctly rounded.* That is, we want to produce the closest number in the floating point system to the correct result, with a rule for tie breaking. Assuming the magnitude of a true result x is between the smallest and largest normalized floating point numbers, there is a floating point number $\hat{x}$ such that $\hat{x} = x(1 + \delta)$ with $|\delta| \leq \epsilon = 2^{-(p+1)}$, where p is the number of bits in the fraction. This leads to a useful *model* for floating point error analysis: if a, b, and $a \odot b$ are all in the normalized floating point range, then

$$\mathrm{fl}(a \odot b) = (a \odot b)(1+\delta), \quad |\delta| < \epsilon, \tag{1}$$

where $\odot$ could be addition, subtraction, multiplication, or division.

Let's consider an example analysis that illustrates both the usefulness and the limitations of the model (1). Suppose I have points A, B, and C in the box $[1,2]^2$, and that the coordinates are given as exact floating point numbers. I want to determine whether C is above, below, or on the oriented line going through A and B. The usual way to do this is by looking at the sign of

$$\det \begin{bmatrix} B_x - A_x & C_x - A_x \\ B_y - A_y & C_y - A_y \end{bmatrix}.$$

We might compute this determinant as something like this:

$$\begin{aligned}
t_1 &= B_x - A_x \\
t_2 &= B_y - A_y \\
t_3 &= C_x - A_x \\
t_4 &= C_y - A_y \\
t_5 &= t_1 \times t_4 \\
t_6 &= t_2 \times t_3 \\
t_7 &= t_5 - t_6.
\end{aligned}$$

Now, suppose we do this computation in floating point. We will call the floating point results $\hat{t}_j$, to distinguish them from the exact results. According to the floating point model (1), we have $|\delta_i| \leq \epsilon$ so that

$$\begin{aligned}
\hat{t}_1 &= (B_x - A_x)(1+\delta_1) \\
\hat{t}_2 &= (B_y - A_y)(1+\delta_2) \\
\hat{t}_3 &= (C_x - A_x)(1+\delta_3) \\
\hat{t}_4 &= (C_y - A_y)(1+\delta_4) \\
\hat{t}_5 &= (\hat{t}_1 \times \hat{t}_4)(1+\delta_5) = (t_1 \times t_4)(1+\delta_1)(1+\delta_4)(1+\delta_5) = t_5(1+\gamma_5) \\
\hat{t}_6 &= (\hat{t}_2 \times \hat{t}_3)(1+\delta_6) = (t_2 \times t_3)(1+\delta_2)(1+\delta_3)(1+\delta_6) = t_6(1+\gamma_6) \\
\hat{t}_7 &= (\hat{t}_5 - \hat{t}_6)(1+\delta_7) \\
&= (t_5 - t_6)\left(1 + \frac{t_5\gamma_5 - t_6\gamma_6}{t_5 - t_6}\right)(1+\delta_7).
\end{aligned}$$

Here, $1 + \gamma_5 = (1 + \delta_1)(1 + \delta_2)(1 + \delta_3) = 1 + \delta_1 + \delta_2 + \delta_3 + O(\epsilon^2)$; that is, $|\gamma_5| \lesssim 3\epsilon$. Similarly, $|\gamma_6| \lesssim 3_\epsilon$.

Now, how large can the relative error in $\hat{t}_7$ be? Ignoring the final rounding error δ_7 – which is in this case insignficant – we have that the relative error is bounded by

$$\left| \frac{t_5\gamma_5 - t_6\gamma_6}{t_5 - t_6} \right| \leq 3\epsilon \frac{|t_5| + |t_6|}{|t_5 - t_6|}.$$

If t_5 and t_6 are not small but $t_5 - t_6$ is small, the relative error in t_7 could be quite large – even though the absolute error remains small. This effect of a large relative error due to a small result in a subtraction is called *cancellation*. In this case, if the relative error is one or larger, then we don't even necessarily have the right sign! That's not a good thing for our test.

Is this error analysis pessimistic? Yes and no. Some things in floating point are *exact*, such as multiplication by a power of two or subtraction of two numbers within a factor of two of each other. Thus, there will be no rounding error in the first four steps ($\delta_i = 0$ for $i = 1, 2, 3, 4$), since we have constrained each of the coordinates to lie in the interval $[1, 2]$. Note that this means that if t_5 and t_6 are close to each other, there will be no rounding error in the last step – δ_7 will be zero! But the overall outlook remains grim: the rounding errors in the multiplications are generally really there; and even if the subtraction is done exactly, it is the propogated error from the multiplication steps that destroys the relative accuracy of the final results.

For this computation, then, the critical rounding error is really in the multiplications. If we could do those multiplications exactly, then we would be able to compute the desired determinant to high relative accuracy (and thus reliably test the determinant's sign). As it turns out, we *can* perform the multiplication exactly if we use a higher precision for intermediate computations than for the input data. For example, suppose the input data are in single precision, but the intermediate computations are all performed in double precision. Then t_1 through t_4 can be represented with significands that are at most 24 bits, and the products t_5 and t_6 require at most 48 bits – and can thus be exactly represented as double precision numbers with 53 bits in the significand.

For this computation, we see a payoff by doing a more detailed analysis than is possible with only the model (1). But the details were tedious, and analysis with the model (1) will usually be good enough.