Week 2: Monday, Jan 31

Problem du jour

What will the result of the following MATLAB code be?

```
x = 2;

% Loop 1
for k = 1:60
  x = sqrt(x);
end

% Loop 2
for k = 1:60
  x = x^2;
end

disp(x);
```

What is the condition number of the computation performed by loop 1? Is it backward stable? What about loop 2?

Logistics

1. HW 1 is due at the start of class Wednesday (2/2).
2. HW 2 is posted, and is due at the start of class next Wednesday (2/9).
3. Ivo's office hours are Monday 11-12.

Miscellaneous comments

1. There are a few themes that we will see repeatedly in this class. Approximation by Taylor series is one of them. If I ask you a question about approximation and you're not sure where to start, a Taylor expansion is usually not a bad guess.

2. A few of you may have gotten confused by my use of order notation for small quantities. But it's basically the same as order notation for large quantities. We say $f(n)$ is $O(g(n))$ when $f(n)/g(n)$ is bounded as $n \to \infty$; we say $f(x)$ is $O(g(x))$ when $f(x)/g(x)$ is bounded as $x \to 0$. Similarly, $f(n)$ is $o(g(n))$ when $f(n)/g(n) \to 0$ as $n \to \infty$, and $f(x)$ is $o(g(x))$ when $f(x)/g(x) \to 0$ as $x \to 0$. Whether we're interested in asymptotics as the argument approaches infinity or zero is usually clear from context.

3. In HW 1, problem 2, there are really three functions:

$$f(x) = \log \sqrt{1+x} - \log \sqrt{x}$$
$$\hat{f}(x) = \text{fl}(f(x)) \text{ (computed by the MATLAB code)}$$
$$\bar{f}(x) = \frac{1}{2x}$$

   For very large values of x, $\bar{f}(x)$ approximates $f(x)$ to high relative accuracy. In contrast, the value $\hat{f(x)}$ returned by MATLAB loses a great deal of accuracy, which is part of the point of the problem. When I asked you to approximate the error in $\hat{f}(x)$ as an approximation to $f(x)$, what I really had in mind is to compute the error in $\hat{f}(x)$ as an approximation to $\bar{f}(x)$; this latter approximation has a relative error of $O(x^{-1})$, which is pretty tiny when $x = 10^{17}$. Get comfortable with the idea of estimating error in an approximation by comparing to a more accurate approximation — it will come up again. There is an art to error analysis, and much of that art boils down to knowing what approximations are useful in what settings.

IEEE floating point arithmetic

The IEEE 754 floating point standard defines a set of normalized double-precision floating point numbers of the form:

$$(-1)^s \times 1.b_1 b_2 \ldots b_p \times 2^E, \quad E = e - \text{bias}$$

In double precision, we have $p = 52$ bits, 11 bits for the exponent, and a bias of 1023. There is also a single precision format (with 23 fraction bits and 8 bits for the exponent), but MATLAB uses double precision by default, and we will henceforth assume double precision unless otherwise stated.

The exponents $e = 0$ and $e = 2047$ ($E = -1023$ and $E = 1024$) have special meanings:

1. For $e = 0$, we have *subnormal numbers* [1]:

$$(-1)^s \times 0.b_1 b_2 \ldots b_p \times 2^{-1023}$$

   The number zero is a special case.

2. For $e = 2047$, we have representations of `inf`, `-inf`, and `NaN` (short for Not a Number). For example, $1/0$ returns `inf`, while $0/0$ returns `NaN`.

The rule for floating point arithmetic is to return *the exact result, correctly rounded*. Usually, "correctly rounded" means "rounded to the nearest floating point number." [2]

We say there is an *exception* when the floating point result is not the same as the exact result. The most standard exception is *inexact* (i.e. some rounding was needed). Other exceptions occur when we fail to produce a normalized floating point number with 53 bits of accuracy. These exceptions are:

Underflow: An expression is too small to be represented as a normalized floating point value. The default behavior is to return a subnormal.

Overflow: An expression is too large to be represented as a floating point number. The default behavior is to return `inf`.

Invalid: An expression evaluates to Not-a-Number (such as $0/0$)

Divide by zero: An expression evaluates "exactly" to an infinite value (such as $1/0$ or $\log(0)$).

Question: What does `1-3*(4/3-1)` yield in double precision?

[1] In *Knowing Machines: Essays on Technical Change*, sociologist Donald MacKenzie tells the short version of the story of how subnormal numbers were accepted as part of the IEEE floating point standard. I've heard the longer version of the story from W. Kahan, and MacKenzie's summary is accurate as I understand it. And I think it's terrific that a sociologist should end up writing about floating point arithmetic! When you find yourself with some spare time, I recommend the essay in particular and the book in general.

[2] There are other rounding modes (round toward 0, round toward infinity) that are sometimes useful, particularly for techniques like *interval arithmetic*.

Error analysis and a standard model

The rule for floating point arithmetic is to return *the exact result, correctly rounded*. Usually, "correctly rounded" means "rounded to the nearest floating point number." It is hard to analyze errors using the "exact result, correctly rounded" characterization of floating point. Instead, we usually analyze floating point computations using a standard *model* for the behavior of normalized floating point numbers in terms of a bound on relative error. The model is that for $\otimes \in \{+, -, \times, /\}$, the floating point value $\mathrm{fl}(x \otimes y)$ generated by computing $x \otimes y$ in floating point is

$$\mathrm{fl}(x \otimes y) = (x \otimes y)(1 + \delta), \quad |\delta| \leq \epsilon_{\mathrm{mach}}.$$

In double precision, the *machine epsilon*[3] is $\epsilon_{\mathrm{mach}} = 2^{-53} \approx 10^{-16}$. This model does not capture the behavior of subnormals or exceptional values, and it is too pessimistic to describe some useful floating point tricks[4]. Nonetheless, it is a simple way to understand the error behavior of most floating point computations, and so is deservedly popular.[5]

As a simple example, consider the phenomenon of cancellation that we described last time. Suppose that $\mathrm{fl}(x) = x(1 + \delta_1)$ and $\mathrm{fl}(y) = y(1 + \delta_2)$ are two positive floating point numbers; then

$$\mathrm{fl}(x - y) = (x(1 + \delta_1) - y(1 + \delta_2))(1 + \delta_3),$$

and

$$\frac{|\,\mathrm{fl}(x - y) - (x - y)|}{|x - y|} = \frac{|x\tilde{\delta}_1 - y\tilde{\delta}_2|}{|x - y|}$$

where

$$|\tilde{\delta}_1| = |\delta_1 + \delta_3 + \delta_1\delta_3| \lesssim 2\epsilon_{\mathrm{mach}}$$

[3]For the purposes of this class, *machine epsilon* is the largest value δ such that $1 + \delta$ gets rounded to 1. According to some other authors, the machine epsilon is the distance between 1 and the next largest floating point number.

[4]For example, if x and y are floating point numbers within a factor of two of each other, then $x - y$ is *exactly* representable in floating point. This actually simplifies life so enormously that I will take advantage of it even though it uses a little more information than "small relative error at each step."

[5]You may decide for yourself if anything having to do with floating point should be described properly as "popular."

and similarly with $\tilde{\delta_2}$. Thus, we have the error bound

$$\frac{|\,\mathrm{fl}(x-y)-(x-y)|}{|x-y|} \lesssim 2\epsilon_{\mathrm{mach}} \frac{|x|+|y|}{|x-y|}.$$

This shows us what can go wrong with cancellation: if x and y are individually much larger than $x-y$, then the relative error in the computed difference may be much larger than ϵ_{mach}.

Notice that we dropped a term in the above analysis: $\delta_1\delta_3 = O(\epsilon_{\mathrm{mach}}^2)$. It is common to drop higher-order terms in this way, since it makes the analysis more tractable, particularly when nonlinear functions are involved. For example, for an exact floating point value x, I would usually write

$$\mathrm{fl}(\sqrt{1+x}) \approx (\sqrt{1+x})(1+\delta/2),$$

implicitly using the fact that $\sqrt{1+\delta} = 1+\delta/2+O(\epsilon_{\mathrm{mach}}^2)$.

Question: Using this sort of linearization, derive an approximate relative error bound for $\mathrm{fl}((1-\sqrt{1-z^2})/2)$.

Rounding in finite differences

Question: Suppose $f(x)$ is computed to within a relative error of ϵ_{mach}. Assuming x_2 and x_1 are distinct and exactly representable in floating point, estimate the relative error due to rounding in the evaluation of

$$f[x_1,x_2] \equiv \frac{f(x_2)-f(x_1)}{x_2-x_1}.$$

For simplicity, assume x_1 and x_0 are close (certainly within a factor of two), and similarly that $f(x_1)$ and $f(x_0)$ are within a factor of two of each other[6].

Answer: Let's work term by term. Let's start with the evaluations of f:

$$\mathrm{fl}(f(x_1)) = f(x_1)(1+\delta_1)$$
$$\mathrm{fl}(f(x_2)) = f(x_2)(1+\delta_2)$$

[6]If we didn't make the assumptions, the analysis would not change substantially. But these assumptions save a couple algebraic steps, and this sort of error analysis is tedious enough without any more symbols flying about.

Assuming $\mathrm{fl}(f(x_1))$ and $\mathrm{fl}(f(x_2))$ are within a factor of two of each other, the subtraction in the numerator incurs no additional error. The only error in the computed difference is due to the function evaluations:

$$\begin{aligned}\mathrm{fl}(f(x_2) - f(x_1)) &= \mathrm{fl}(f(x_2)) - \mathrm{fl}(f(x_1)) \\ &= f(x_2) - f(x_1) + (f(x_2)\delta_2 - f(x_1)\delta_2) \\ &= (f(x_2) - f(x_1))\,(1 + \delta_{\mathrm{sub}})\end{aligned}$$

where

$$|\delta_{\mathrm{sub}}| = \left|\frac{f(x_2)\delta_2 - f(x_1)\delta_2}{f(x_2) - f(x_1)}\right| \lesssim \frac{2|f(x_1)|\epsilon_{\mathrm{mach}}}{|f'(x_1)||x_2 - x_1|}$$

If x_2 and x_1 are within a factor of two, then there is also no error from this subtraction:

$$\mathrm{fl}(x_2 - x_1) = x_2 - x_1.$$

Therefore, the only additional error is due to the division:

$$\begin{aligned}\mathrm{fl}\left(f[x_1, x_2]\right) &= f[x_1, x_2](1 + \delta_{\mathrm{sub}})(1 + \delta_{\mathrm{div}}) \\ &= f[x_1, x_2](1 + \delta_{\mathrm{total}})\end{aligned}$$

where

$$|\delta_{\mathrm{total}}| \approx |\delta_{\mathrm{sub}} + \delta_{\mathrm{div}}| \lesssim \left(1 + \frac{2|f(x_1)|}{|f'(x_1)||x_2 - x_1|}\right)\epsilon_{\mathrm{mach}}.$$

Cumulative error in finite differences

The finite difference expression $f[x_1, x_2]$ is useful as an approximation to $f'(x_1)$. Using a first-order Taylor expansion with remainder, we have

$$f[x_1, x_2] \equiv \frac{f(x_2) - f(x_1)}{x_2 - x_1} = f'(x) + \frac{f''(\xi)}{2}(x_2 - x_1)$$

As an approximation to $f'(x_1)$, the exact value of $f[x_1, x_2]$ has an error of $O(|x_2 - x_1|)$ — the smaller the difference, the better. This is an example of *truncation error*. But $\mathrm{fl}(x[f_1, x_2])$ approximates the true $f[x_1, x_2]$ with an error of $O(\epsilon_{\mathrm{mach}}/|x_2 - x_1|)$ — the smaller the difference, the worse it gets! The error is minimal when truncation and rounding error are the same size.

Figure 1 shows the tension between rounding error and truncation error for this example. We use a log-log plot so that we can easily see the scaling

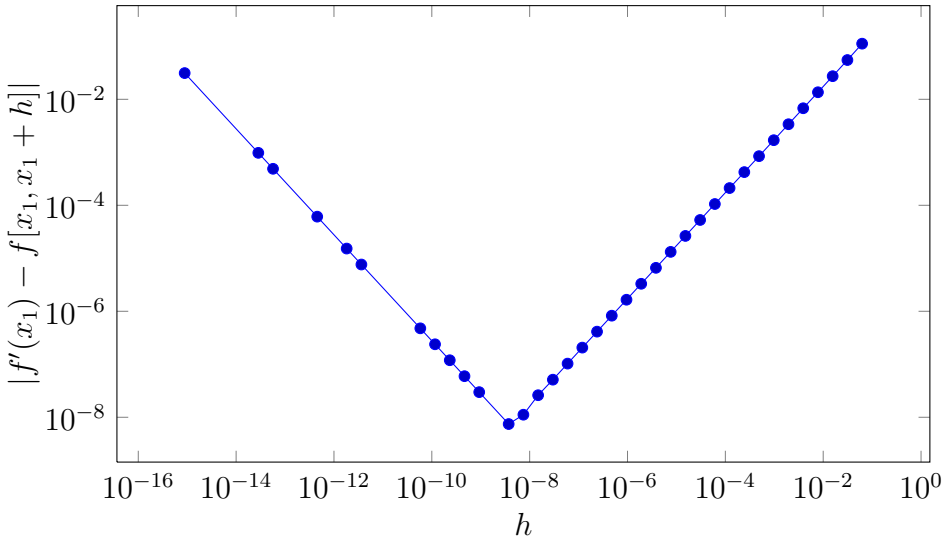

Figure 1: Relation between $f'(x_1)$ and $f[x_1, x_1 + h]$ for different steps h ($f(x) = x^3$ and $x_1 = \sqrt{1/3}$). Missing points correspond to where the finite difference computation returns exactly one.

behavior in terms of slopes. For step sizes larger than around $\sqrt{\epsilon_{\text{mach}}} \approx 10^{-8}$, the total error is dominated by truncation error ($O(h)$, and so slope 1 on a log-log plot). For step sizes smaller than $\sqrt{\epsilon_{\text{mach}}}$, the total error is dominated by rounding ($O(h^{-1})$, and so slope -1 on a log-log plot), except at a few points where the finite difference approximation is accidentally exact.