Week 1: Monday, Jan 24

Logistics

We will go over the syllabus in the first class, but in general you should look at the class web page at

http://www.cs.cornell.edu/~bindel/class/cs3220-s11

The web page is your source for the syllabus, homework, lecture notes and slides, and course announcements. For materials that are private to the class (e.g. grades and solutions), we will use the Course Management System:

http://cms.csuglab.cornell.edu/web/guest

Course Overview

Scientific computing is about using computers to solve problems of science and engineering (and sometimes other fields). These problems involve *continuous* mathematics: real numbers, continuous functions, integrals, differential equations, and so on. We will focus on *numerical methods*[1] to solve these problems.

To solve real-world scientific problems, we need to know about the application (to ask questions that make sense), mathematical analysis (to formulate a useful model and numerical methods to analyze it), and computer science (to write fast, correct, robust solvers). In class, I gave three examples from my own research:

- Computer-aided design tools for micro-electro-mechanical systems,
- Inference of the properties of computer networks,
- Discovery of overlapping communities in social networks.

One semester is not a long time – too short to cover background for most real world problems. So we will focus on the computational aspects and the mathematical analysis that goes with it. Our goal is to learn how to use numerical methods wisely, and to craft fast, accurate, and robust soutions using on standard techniques and libraries.

[1] There are also *symbolic* methods for analyzing continuous mathematics problems, but we will not discuss these in our course.

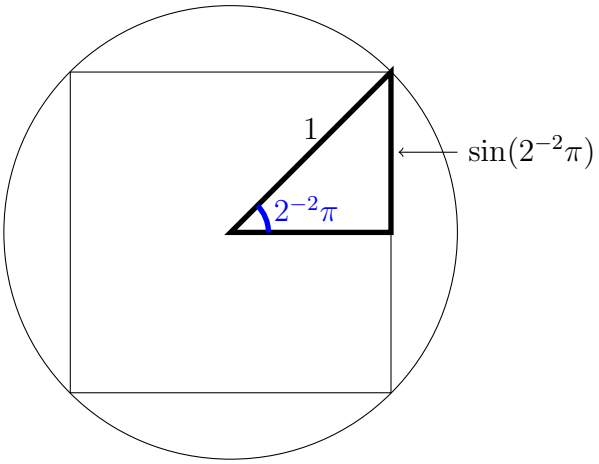

Figure 1: Computing the half side length of an inscribed 2^2-gon.

Tricky Teasers

One running theme in our class will be *approximation*. Even basic arithmetic on the computer is approximate: the computer represents real numbers using *floating point*, a sort of (binary) scientific notation with some bells and whistles. We will talk about this in more detail in the next lecture. For now, I want to mention the bare minimum needed to discuss a couple examples:

- Suppose $\hat{x}$ approximates x. The *absolute* error is $e = \hat{x} - x$. Absolute error is not always helpful; a centimeter is a tiny error in a measurement of the radius of the earth, but a huge error in a measurement of the radius of a hair. The *relative error* $\delta = (\hat{x} - x)/x$ is often a better measure.

- Double precision floating point numbers (the kind used by default in MATLAB) have a significand 53 bits long — that's about 16 decimal digits. Basic arithmetic operations in floating point follow the rule *compute the exact result, correctly rounded*.

An Approximation Algorithm of Archimedes

Archimedes estimated the value of π by computing the semiperimeter of regular N-sided polygons (N-gons) inscribed in a unit circle (Figure 1):

$$\pi \approx \frac{1}{2} \text{ side length of } N\text{-gon} \times N = \sin\left(\frac{\pi}{N}\right) \times N.$$

We will look at the semiperimeters of 2^k-gons:

$$s_k = 2^k \sin(2^{-k}\pi).$$

If we define $x_k = \sin^2(2^{-k}\pi)$, we can relate x_k to x_{k-1}:

$$x_k^2 - x_k + \frac{1}{4}x_{k-1} = 0.$$

We know that $x_2 = 1/2$ (the side length of an inscribed square is 1), so we can write a little MATLAB program that recursively computes values of x_k to get ever better estimates of π:

```
% s = lec01pi(kmax)
%
% Compute semiperimeters s(k) of 2^k-gons for k = 2:kmax.

function s = lec01pi(kmax)

  x = zeros(1,kmax);
  x(2) = 0.5;
  for k = 3:kmax
    x(k) = ( 1 - sqrt(1-x(k-1)) )/2;
  end
  s = 2.^(1:kmax) .* sqrt(x);
```

The results of the computation are shown in Figure 2. Clearly, something has gone awry; the semiperimeter for a 2^{30}-gon (with about a billion sides) is computed to be zero! The question is: what should have happened in exact arithmetic, what went wrong in floating point, and how might we fix it?

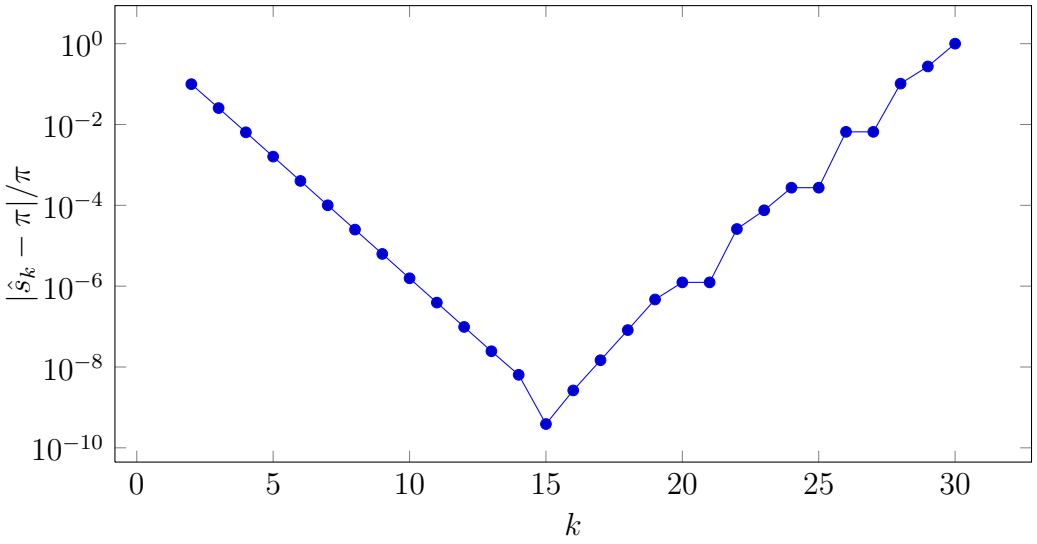

Figure 2: Error in estimating π by 2^k-gon semiperimeters from `lec01pi`.

Analyzing Archimedes

Using Taylor series, we can write

$$\begin{aligned} s_k &= 2^k \left((2^{-k}\pi) - \frac{1}{6}(2^{-k}\pi)^3 + O(2^{-5k}) \right) \\ &= \pi \left(1 - \frac{\pi^2}{6} 2^{-2k} + O(2^{-4k}) \right). \end{aligned}$$

Thus in exact arithmetic, the relative error in approximating π by s_k is

$$\frac{|s_k - \pi|}{\pi} \approx \frac{\pi^2}{6} 2^{-2k}.$$

This is exactly the behavior we see in the MATLAB calculation until $k \approx 14$, corresponding to a relative error of about 6×10^{-9}. After that, the floating point values $\hat{s}_k$ become successively *worse* approximations to π, until we reach a 100% relative error with $\hat{s}_{30} = 0$.

Now, consider what would happen even if we had the *exact* value of x_{29} and tried to use it to compute x_{30}. We know that the true value of x_{29} should be very close to $2^{-58}\pi^2$, or about 3×10^{-17}. Therefore, when we compute $1 - x_{29}$ and round to 16 places, we get a computed value of 1. This yields

$$\hat{x}_{29} = 1 - \sqrt{1} = 0,$$

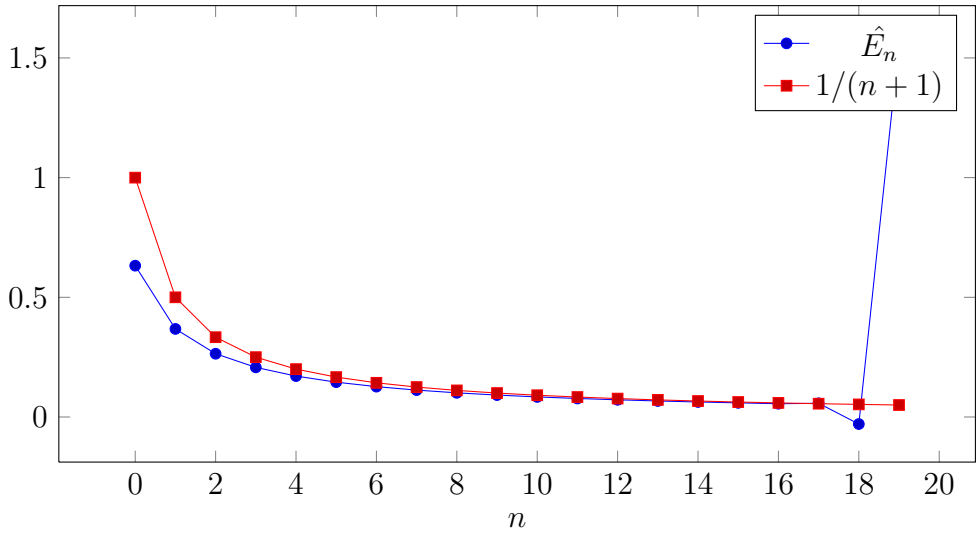

Figure 3: Computed values $\hat{E}_n$ compared to a supposed upper bound.

at which point the calculation falls flat on its face.

This problem with inaccurate calculation of the smaller root of a quadratic is a classic example of the perils of *cancellation*. It is described in Heath, and there is an exercise on it in at the end of the first chapter of Moler. I recommend you read these references before tackling the homework problem in which you are asked to modify `lec01pi` so that $\hat{s}_k$ maintains good accuracy by modifying the calculation of $\hat{x}_k$.

An Integral Iteration

Consider $E_n = \int_0^1 x^n e^{x-1}\,dx$. We can derive a recurrence for E_n by repeated integration by parts:

$$\begin{aligned}\int_0^1 x^n e^{x-1}\,dx &= \left[x^n e^{x-1}\right]_0^1 - \int_0^1 n x^{n-1} e^{x-1}\,dx \\ &= 1 - nE_{n-1}.\end{aligned}$$

We know that the true values of E_n satisfy

$$\frac{1}{e(n+1)} < E_n < \frac{1}{n+1}$$

with E_n approaching the upper bound of $1/(n+1)$ as n gets large. In contrast, Figure 3 shows what happens to the computed floating point values $\hat{E}_n$ when we run the recurrence forward. Things appear to be fine until around $n = 17$. But we compute a negative number for $\hat{E}_{18}$ and a number that is far too large for $\hat{E}_{19}$. If we continue running the recurrence forward, we will find $\hat{E}_{20} \approx -30$ and $\hat{E}_{30} \approx -3.3 \times 10^{15}$! Again, the question: what went wrong?

Analyzing the Integral Iteration

Suppose that we were able to evaluate the recurrence for E_n with no errors save for the roundoff error in evaluating E_0. That is, suppose that the true integrals E_n and the approximations $\hat{E}_n$ exactly satisfy the recurrences

$$\begin{aligned} E_n &= 1 - nE_{n-1}, & E_0 &= 1 - 1/e, \\ \hat{E}_n &= 1 - n\hat{E}_{n-1}, & \hat{E}_0 &= 1 - 1/e + \epsilon_0. \end{aligned}$$

We can derive a recurrence for $\epsilon_n = \hat{E}_n - E_n$ by subtracting the recurrences for $\hat{E}_n$ and E_n:

$$\epsilon_n = (1 - n\hat{E}_{n-1}) - (1 - nE_{n-1}) = n(\hat{E}_{n-1} - E_{n-1}) = n\epsilon_{n-1}.$$

Therefore, $\epsilon_n = n!\,\epsilon_0$. This is a truly ferocious amplification of the initial error. Note that $18! \approx 6.4 \times 10^{15}$, so even without any intermediate rounding errors, the roundoff error in the initial condition $\hat{E}_0$ will be magnified until after 18 steps it is of the same order of magnitude as the true value E_{18}.

Unlike the previous example, there is no simple equivalent formulation of the recurrence that will make the problem disappear. The method is intrinsically *unstable*. There is a fix, though. If we run the recurrence *backward* from the initial condition $\hat{E}_N = 0$ for sufficiently large N, we quickly get very accurate estimates for the true values of E_n. The same forces that amplify error when we recurse forward will supress errors when we recurse backward.

The Takeaway

Approximation is an inherent part of scientific computing. Even the simple, seemingly "exact" quadratic formula can be dangerous if we fail to recognize the approximation due to floating point arithmetic. Furthermore, small initial errors can be amplified over the course of a calculation until the final

result computed bears no resemblance to the correct answer. We need to understand these approximations if we are to build codes that return meaningful results (not to mention building useful test cases!).