Week 1: Monday, Jan 23

Safe computing

If this class were about shooting rather than computing, we'd probably start by talking about where the safety is. Otherwise, someone would shoot themself in the foot. The same is true of basic error analysis: even if it's not the most glamorous part of scientific computing, it comes at the start of the course in order to keep students safe from themselves. For this, a rough-and-ready discussion of sensitivity analysis, simplified models of rounding, and exceptional behavior is probably sufficient. We'll go a bit beyond that, but you can think of most of the material in the first week in this light: basic safety training for computing with real numbers.

Some basics

Suppose $\hat{x}$ is an approximation for x. The *absolute error* in the approximation is $\hat{x} - x$; the *relative error* is $(\hat{x} - x)/x$. Both are important, but we will use relative error more frequently for this class.

Most measurements taken from the real world have error. Computations with approximate inputs generally produce approximate outputs, even when the computations are done exactly. Thus, we need to understand the *propagation* of errors through a computation. For example, suppose that we have two inputs $\hat{x}$ and $\hat{y}$ that approximate true values x and y. If e_x and $\delta_x = e_x/x$ are the absolute and relative errors for x (and similarly for y), we can write

$$\begin{aligned}\hat{x} &= x + e_x = x(1+\delta_x)\\ \hat{y} &= y + e_y = y(1+\delta_y).\end{aligned}$$

When we add or subtract $\hat{x}$ and $\hat{y}$, we add and subtract the absolute errors:

$$\begin{aligned}\hat{x} + \hat{y} &= (x+y) + (e_x + e_y)\\ \hat{x} - \hat{y} &= (x-y) + (e_x - e_y).\end{aligned}$$

Note, however, that the relative error in the approximation of $x - y$ is $(e_x - e_y)/(x - y)$; and if x and y are close, this relative error may be much larger than δ_x or δ_y. This effect is called *cancellation*, and it is a frequent culprit in computations gone awry.

From the perspective of relative error, multiplication and division are more benign than addition and subtraction. The relative error of a product is (approximately) the sum of the relative errors of the inputs:

$$\begin{aligned}\hat{x}\hat{y} &= xy(1+\delta_x)(1+\delta_y) \\ &= xy(1+\delta_x+\delta_y+\delta_x\delta_y) \\ &\approx xy(1+\delta_x+\delta_y).\end{aligned}$$

If δ_x and δ_y are small, then $\delta_x\delta_y$ is absolutely miniscule, and so we typically drop it when performing an error analysis[1] Similiarly, using the fact that

$$(1+\delta)^{-1} = 1 - \delta + O(\delta^2),$$

the relative error in a quotient is approximately the difference of the relative errors of the inputs:

$$\begin{aligned}\frac{\hat{x}}{\hat{y}} &= \frac{x}{y}\left(\frac{1+\delta_x}{1+\delta_y}\right) \\ &\approx \frac{x}{y}(1+\delta_x)(1-\delta_y) \\ &\approx \frac{x}{y}(1+\delta_x-\delta_y).\end{aligned}$$

More generally, suppose f is some differentiable function of x, and we approximate $z = f(x)$ by $\hat{z} = f(\hat{x})$. We estimate the absolute error in the approximation by Taylor expansion:

$$\hat{z} - z = f(\hat{x}) - f(x) \approx f'(x)(\hat{x} - x).$$

The magnitude $|f'(x)|$ tells us the relationship between the size of absolute errors in the input and the output. Frequently, though, we care about how *relative* errors are magnified by the computation:

$$\frac{\hat{z}-z}{z} \approx \left(\frac{xf'(x)}{z}\right)\frac{\hat{x}-x}{x}.$$

The size of the quantity $xf'(x)/f(x)$ that tells us how relative errors on input affect the relative errors of the output is sometimes called the *condition number* for the problem. Problems with large condition numbers ("ill-conditioned" problems) are delicate: a little error in the input can destroy the accuracy of the output.

[1]This is a *first-order* error analysis or *linearized* error analysis, since we drop any terms that involve products of small errors.

A simple example

What's the average velocity of the earth as it travels around the sun?

Yes, you can ask Google this question, and it will helpfully tell you that the answer is about 29.783 km/s. But if you have a few facts at your fingertips, you can get a good estimate on your own:

- The earth is $t \approx 500$ light-seconds from the sun.
- The speed of light is $c \approx 3 \times 10^8$ m/s.
- A year is about $y \approx \pi \times 10^7$ seconds.

So the earth follows a roughly circular orbit with radius of about $r = ct \approx 1.5 \times 10^{11}$ meters, traveling $2\pi r$ meters in a year. So our velocity is about

$$\frac{2\pi ct}{y} \approx \frac{2\pi \times 1.5 \times 10^{11} \text{ m}}{\pi \times 10^7 \text{ s}} = 30 \text{ km/s}.$$

How wrong is the estimate $\hat{v} = 30$ km/s? The absolute error compared to the truth ($v = 29.783$ km/s a la Wiki answers) is about 217 m/s. Unfortunately, it's hard to tell without context whether this is a good or a bad error. Compared to the speed of the earth going around the sun, it's pretty small; compared to my walking speed, it's pretty big. It's more satisfying to know the relative error is

$$\frac{\hat{v} - v}{v} \approx 0.0073,$$

or less than one percent. Informally, we might say that the answer has about two correct digits.

What is the source of this slightly-less-than-a-percent error? Crudely speaking, there are two sources:

1. We assumed the earth's orbit is circular, but it's actually slightly elliptical. This *modeling error* entered before we did any actual calculations.
2. We approximated all the parameters in our calculation: the speed of light, the distance to the sun in light-seconds, and the length of a year. More accurate values are:

$$\begin{aligned} t &= 499.0 \text{ s} \\ c &= 299792458 \text{ m/s} \\ y &= 31556925.2 \text{ s} \end{aligned}$$

> If we plug these values into our earlier formula, we get a velocity of 29.786 km/s. This is approximated by 30 km/s with a relative error of 0.0072.

We'll put aside for the moment the ellipticity of the earth's orbit (and the veracity of Wiki Answers) and focus on the error inherited from the parameters. Denoting our earlier estimates by $\hat{t}$, $\hat{c}$, and $\hat{y}$, we have the relative errors

$$\begin{aligned}
\delta_t &\equiv (\hat{t} - t)/t = 20 \times 10^{-4} \\
\delta_c &\equiv (\hat{c} - c)/c = 7 \times 10^{-4} \\
\delta_y &\equiv (\hat{y} - t)/y = -45 \times 10^{-4}
\end{aligned}$$

According to our error propagation formulas, we should have

$$\frac{2\pi\hat{c}\hat{t}}{\hat{y}} \approx \left(\frac{2\pi ct}{y}\right)(1 + \delta_c + \delta_t - \delta_y).$$

In fact, this looks about right:

$$20 \times 10^{-4} + 7 \times 10^{-4} + 45 \times 10^{-4} = 72 \times 10^{-4}.$$

An example of error amplification

Our previous example was benign; let us now consider a more sinister example. Consider the integrals

$$E_n = \int_0^1 x^n e^{x-1}\,dx$$

It's easy to find $E_0 = 1 - 1/e$; and for $n > 0$, integration by parts[2] gives us

$$E_n = x^n e^{x-1}\big|_0^1 - n\int_0^1 x^{n-1}e^{x-1}\,dx = 1 - nE_{n-1}.$$

We can use the following MATLAB program to compute E_n for $n = 1, \ldots, n_{\max}$:

[2] If you do not remember integration by parts, you should go to your calculus text and refresh your memory.

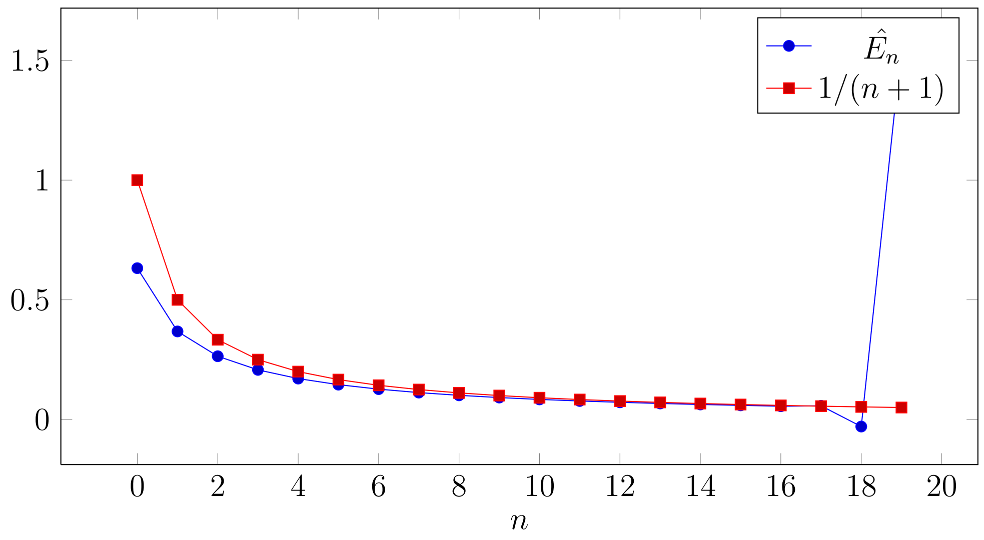


Figure 1: Computed values $\hat{E}_n$ compared to a supposed upper bound.

```
%
% Compute the integral of x^n exp(x-1) from 0 to 1
% for n = 1, 2, ..., nmax
%
function E = lec01int(nmax)

  En = 1-1/e;
  for n = 1:nmax
    En = 1-n*En;
    E(n) = En;
  end
```

The computed values of $\hat{E}_n$ according to our MATLAB program are shown in Figure 1. A moment's contemplation tells us that something is amiss with these problems. Note that $1/e \leq e^{x-1} \leq 1$ for $0 \leq x \leq 1$, so that

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

As shown in the figure, our computed values violate these bounds. What has gone wrong?

The integral is perfectly behaved; the problem lies in the algorithm. The initial value E_0 can only be represented approximately on the computer; let us write the stored value as $\hat{E}_0$. Even if this were the only error, we would find the error in later calculations grows terribly rapidly:

$$\begin{aligned}
\hat{E}_0 &= E_0 + \epsilon \\
\hat{E}_1 &= 1 - \hat{E}_0 = E_1 - \epsilon \\
\hat{E}_2 &= 1 - 2\hat{E}_1 = E_2 + 2\epsilon \\
\hat{E}_3 &= 1 - 3\hat{E}_2 = E_3 - 3 \cdot 2\epsilon \\
&\vdots \\
\hat{E}_n &= E_n + (-1)^n n!\epsilon.
\end{aligned}$$

This is an example of an *unstable* algorithm.

Sources of error

We understand the world through simplified models. These models are, from the outset, approximations to reality. To quote the statistician George Box:

All models are wrong. Some models are useful.

While modeling errors are important, we can rarely judge their quality without the context of a specific application. In this class, we will generally take the model as a given, and analyze errors from the input and the computation.

In addition to error intrinsic in our models, we frequently have error in input parameters derived from measurements. In *ill-conditioned* problems, small relative errors in the inputs can severely compromise the accuracy of outputs. For example, if the input to a problem is known to within a millionth of a percent and the condition number for the problem is 10^7, the result might only have one correct digit. This is a difficulty with the problem, not with the method used to solve the problem. Ideally, a well-designed numerical routine might warn us when we ask it to solve an ill-conditioned problem, but asking for an accurate result may be asking too much. Instead, we will seek *stable* algorithms that don't amplify the error undeservedly. That is, if the condition number for a problem is κ, a stable algorithm should amplify the input error by at most $C\kappa$, where C is some modest constant that does not depend on the details of the input.

In order to produce stable algorithms, we must control the sources of error from within the computation itself. In the order we encounter them, these include:

Roundoff error: IEEE floating point arithmetic is essentially binary scientific notation. The number 1/3 cannot be represented exactly as a finite decimal. It can't be represented exactly in a finite number of binary digits, either. We can, however, approximate 1/3 to a very high degree of accuracy.

Termination of iterations: Nonlinear equations, optimization problems, and even linear systems are frequently solved by iterations that produce successively better approximations to a true answer. At some point, we decide that we have an answer that is good enough, and stop.

Truncation error: We frequently approximate derivatives by finite differences and integrals by sums. The error in these approximations is frequently called truncation error.

Stochastic error: Monte Carlo methods use randomness to compute approximations. The variance due to randomness typically dominates the error in these methods.

Week 1: Wednesday, Jan 25

Binary floating point encodings

Binary floating point arithmetic is essentially scientific notation. Where in decimal scientific notation we write

$$\frac{1}{3} = 3.333\ldots \times 10^{-1},$$

in floating point, we write

$$\frac{(1)_2}{(11)_2} = (1.010101\ldots)_2 \times 2^{-2}.$$

Because computers are finite, however, we can only keep a finite number of bits after the binary point.

In general, a *normal floating point number* has the form

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

where $s \in \{0, 1\}$ is the *sign bit*, E is the *exponent*, and $(1.b_2 \ldots b_p)_2$ is the *significand*. In the 64-bit double precision format, $p = 52$ bits are used to store the significand, 11 bits are used for the exponent, and one bit is used for the sign. The valid exponent range for normal floating point numbers is $-1023 < E < 1024$; this leaves two exponent encodings left over for special purpose. One of these special exponents is used to encode *subnormal numbers* of the form

$$(-1)^s \times (0.b_1 b_2 \ldots b_p)_2 \times 2^{-1022};$$

the other special exponent is used to encode $\pm\infty$ and NaN (Not a Number).

For a general real number x, we will write

$$\mathrm{fl}(x) = \text{ correctly rounded floating point representation of } x.$$

By default, "correctly rounded" means that we find the closest floating point number to x, breaking any ties by rounding to the number with a zero in the last bit[1]. If x exceeds the largest normal floating point number, then $\mathrm{fl}(x) = \infty$.

[1]There are other rounding modes beside the default, but we will not discuss them in this class

Basic floating point arithmetic

For basic operations (addition, subtraction, multiplication, division, and square root), the floating point standard specifies that the computer should produce the *true result, correctly rounded.* So the MATLAB statement

```
% Compute the sum of x and y (assuming they are exact)
z = x + y;
```

actually computes the quantity $\hat{z} = \mathrm{fl}(x+y)$. If $\hat{z}$ is a normal double-precision floating point number, it will agree with the true z to 52 bits after the binary point. That is, the relative error will be smaller in magnitude than the *machine epsilon* $\epsilon_{\mathrm{mach}} = 2^{-53} \approx 1.1 \times 10^{-16}$:

$$\hat{z} = z(1+\delta), \quad |\delta| < \epsilon_{\mathrm{mach}}.$$

More generally, basic operations that produce normalized numbers are correct to within a relative error of ϵ_{mach}. The floating point standard also recommends that common transcendental functions, such as exponential and trig functions, should be correctly rounded, though compliant implementations that do not comply with this recommendation may produce results with a relative error just slightly larger than ϵ_{mach}.

The fact that normal floating point results have a relative error less than ϵ_{mach} gives us a useful *model* for reasoning about floating point error. We will refer to this as the "$1+\delta$" model. For example, suppose x is an exactly-represented input to the MATLAB statement

```
z = 1-x*x;
```

We can reason about the error in the computed $\hat{z}$ as follows:

$$\begin{aligned}
t_1 &= \mathrm{fl}(x^2) = x^2(1+\delta_1) \\
t_2 &= 1 - t_1 = (1-x^2)\left(1 + \frac{\delta_1 x^2}{1-x^2}\right) \\
\hat{z} &= \mathrm{fl}(1-t_1) = z\left(1 + \frac{\delta_1 x^2}{1-x^2}\right)(1+\delta_2) \\
&\approx z\left(1 + \frac{\delta_1 x^2}{1-x^2} + \delta_2\right),
\end{aligned}$$

where $|\delta_1|, |\delta_2| \leq \epsilon_{\mathrm{mach}}$. As before, we throw away the (tiny) term involving $\delta_1\delta_2$. Note that if z is close to zero (i.e. if there is *cancellation* in the subtraction), then the model shows the result may have a large relative error.

Exceptions

We say there is an *exception* when the floating point result is not an ordinary value that represents the exact result. The most common exception is *inexact* (i.e. some rounding was needed). Other exceptions occur when we fail to produce a normalized floating point number. 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)).

When exceptions other than inexact occur, the usual "$1+\delta$" model used for most rounding error analysis is not valid.

Limits of the "$1 + \delta$" model

Apart from the fact that it fails when there are exceptions other than inexact, the "$1+\delta$" model of floating point does not reflect the fact that some computations involve no rounding error. For example:

- If x and y are floating point numbers within a factor of two of each other, $\mathrm{fl}(x - y)$ is computed without rounding error.
- Barring overflow or underflow to zero, $\mathrm{fl}(2x) = 2x$ and $\mathrm{fl}(x/2) = x/2$.
- Integers between $\pm(2^{53} - 1)$ are represented exactly.

These properties of floating point allow us to do some clever things, such as using ordinary double precision arithmetic to simulate arithmetic with about twice the number of digits. You should be aware that these tricks exist, even if you never need to implement them – otherwise, I may find myself cursing a compiler you wrote for rearranging the computations in a floating point code that I wrote!

Finding and fixing floating point problems

Floating point arithmetic is not the same as real arithmetic. Even simple properties like associativity or distributivity of addition and multiplication only hold approximately. Thus, some computations that look fine in exact arithmetic can produce bad answers in floating point. What follows is a (very incomplete) list of some of the ways in which programmers can go awry with careless floating point programming.

Cancellation

If $\hat{x} = x(1+\delta_1)$ and $\hat{y} = y(1+\delta_2)$ are floating point approximations to x and y that are very close, then $\mathrm{fl}(\hat{x}-\hat{y})$ may be a poor approximation to $x-y$ due to *cancellation.* In some ways, the subtraction is blameless in this tail: if x and y are close, then $\mathrm{fl}(\hat{x}-\hat{y}) = \hat{x}-\hat{y}$, and the subtraction causes no additional rounding error. Rather, the problem is with the approximation error already present in $\hat{x}$ and $\hat{y}$.

The standard example of loss of accuracy revealed through cancellation is in the computation of the smaller root of a quadratic using the quadratic formula, e.g.

$$x = 1 - \sqrt{1-z}$$

for z small. Fortunately, some algebraic manipulation gives an equivalent formula that does not suffer cancellation:

$$x = \left(1 - \sqrt{1-z}\right)\left(\frac{1+\sqrt{1-z}}{1+\sqrt{1-z}}\right) = \frac{z}{1+\sqrt{1-z}}.$$

Sensitive subproblems

We often solve problems by breaking them into simpler subproblems. Unfortunately, it is easy to produce badly-conditioned subproblems as steps to solving a well-conditioned problem. As a simple (if contrived) example, try running the following MATLAB code:

```
x = 2;
for k = 1:60, x = sqrt(x); end
for k = 1:60, x = x^2;     end
disp(x);
```

In exact arithmetic, this should produce 2, but what does it produce in floating point? In fact, the first loop produces a correctly rounded result, but the second loop represents the function $x^{2^{60}}$, which has a condition number far greater than 10^{16} — and so all accuracy is lost.

Unstable recurrences

We gave an example of this problem in the last lecture notes when we looked at the recurrence

$$\begin{aligned} E_0 &= 1 - 1/e \\ E_n &= 1 - nE_{n-1}, \quad n \geq 1. \end{aligned}$$

No single step of this recurrence causes the error to explode, but each step amplifies the error somewhat, resulting in an exponential growth in error.

Undetected underflow

In Bayesian statistics, one sometimes computes ratios of long products. These products may underflow individually, even when the final ratio is not far from one. In the best case, the products will grow so tiny that they underflow to zero, and the user may notice an infinity or NaN in the final result. In the worst case, the underflowed results will produce nonzero subnormal numbers with unexpectedly poor relative accuracy, and the final result will be wildly inaccurate with no warning except for the (often ignored) underflow flag.

Bad branches

A NaN result is often a blessing in disguise: if you see an unexpected NaN, at least you *know* something has gone wrong! But all comparisons involving NaN are false, and so when a floating point result is used to compute a branch condition and an unexpected NaN appears, the result can wreak havoc. As an example, try out the following code in MATLAB.

```
x = 0/0;
if x < 0 then          disp('x is negative');
elseif x >= 0 then disp('x is non-negative');
else                   disp('Uh...');
end
```

Problems to ponder

1. In double precision, is fl(0.2) larger, smaller, or equal to 0.2?
2. How do we accurately evaluate $\sqrt{1+x} - \sqrt{1-x}$ when $x \ll 1$?
3. How do we accurately evaluate $\ln\sqrt{x+1} - \ln\sqrt{x}$ when $x \gg 1$?
4. How do we accurately evaluate $(1-\cos(x))/\sin(x)$ when $x \ll 1$?
5. How would we compute $\cos(x) - 1$ accurately when $x \ll 1$?
6. The *Lamb-Oseen vortex* is a solution to the 2D Navier-Stokes equation that plays a key role in some methods for computational fluid dynamics. It has the form
$$v_\theta(r,t) = \frac{\Gamma}{2\pi r}\left(1 - \exp\left(\frac{-r^2}{4\nu t}\right)\right)$$
How would one evaluate $v(r,t)$ to high relative accuracy for all values of r and t (barring overflow or underflow)?
7. For $x > 1$, the equation $x = \cosh(y)$ can be solved as
$$y = -\ln\left(x - \sqrt{x^2-1}\right).$$
What happens when $x = 10^8$? Can we fix it?
8. The difference equation
$$x_{k+1} = 2.25x_k - 0.5x_{k-1}$$
with starting values
$$x_1 = \frac{1}{3}, \qquad x_2 = \frac{1}{12}$$
has solution
$$x_k = \frac{4^{1-k}}{3}.$$
Is this what you actually see if you compute? What goes wrong?
9. Considering the following two MATLAB fragments:

```
% Version 1
f = (exp(x)−1)/x;

% Version 2
y = exp(x);
f = (1−y)/log(y);
```

In exact arithmetic, the two fragments are equivalent. In floating point, the first formulation is inaccurate for $x \ll 1$, while the second formulation remains accurate. Why?

10. Running the recurrence $E_n = 1 - nE_{n-1}$ *forward* is an unstable way to compute $\int_0^1 x^n e^{x-1}\,dx$. However, we can get good results by running the recurrence *backward* from the estimate $E_n \approx 1/(N+1)$ starting at large enough N. Explain why. How large must N be to compute E_{20} to near machine precision?

11. How might you accurately compute this function for $|x| < 1$?

$$f(x) = \sum_{j=0}^{\infty} \left(\cos(x^j) - 1\right)$$

Week 2: Monday, Jan 30

Overview

After this week (and the associated problems), you should come away with some understanding of

- *Algorithms* for equation solving, particularly bisection, Newton, secant, and fixed point iterations.
- *Analysis* of error recurrences in order fo find rates of convergence for algorithms; you should also understand a little about analyzing the sensitivity of the root-finding problem itself.
- *Application* of standard root-finding procedures to real problems. This frequently means some sketches and analysis done in advance in order to figure out appropriate rescalings and changes of variables, handle singularities, and find good initial guesses (for Newton) or bracketing intervals (for bisection).

A little long division

Let's begin with a question: Suppose I have a machine with hardware support for addition, subtraction, multiplication, and scaling by integer powers of two (positive or negative). How can I implement reciprocation? That is, if $d > 1$ is an integer, how can I compute $1/d$ without using division?

This is a linear problem, but as we will see, it presents many of the same issues as nonlinear problems.

Method 1: From long division to bisection

Maybe the most obvious algorithm to compute $1/d$ is binary long division (the binary version of the decimal long division that we learned in grade school). To compute a bit in the kth place after the binary point (corresponding to the value 2^{-k}), we see whether $2^{-k}d$ is greater than the current remainder; if it is, then we set the bit to one and update the remainder. This algorithm is shown in Figure 1.

```
function x = lec11division(d, n)
  % Approximate x = 1/d by n steps of binary long division.

  r = 1;       % Current remainder
  x = 0;       % Current reciprocal estimate
  bit = 0.5;   % Value of a one in the current place

  for k = 1:n
    if r > d*bit
      x = x +  bit;
      r = r − d*bit;
    end
    bit = bit/2;
  end
```

Figure 1: Approximate $1/d$ by n steps of binary long division.

```
function x = lec11bisect(d, n)
  % Approximate x = 1/d by n steps of bisection
  % At each step f(x) = dx−1 is negative at the lower
  % bound, positive at the upper bound.

  hi = 1;    % Current upper bound
  lo = 0;    % Current lower bound

  for k = 1:n
    x  = (hi+lo)/2;
    fx = d*x−1;
    if fx > 0
      hi = x;
    else
      lo = x;
    end
  end
  x = (hi+lo)/2;
```

Figure 2: Approximate $1/d$ by n steps of bisection.

At step k of long division, we have an approximation $\hat{x}$, $\hat{x} \leq 1/d < \hat{x}+2^{-k}$, and a remainder $r = 1 - d\hat{x}$. Based on the remainder, we either get a zero bit (and discover $\hat{x} \leq 1/d < \hat{x} + 2^{-(k+1)}$), or we get a one bit (i.e. $\hat{x}+2^{-(k+1)} \leq 1/d < \hat{x}+2^{-k}$). That is, the long division algorithm is implicitly computing interals that contain $1/d$, and each step cuts the interval size by a factor of two. This is characteristic of *bisection*, which finds a zero of any continuous function $f(x)$ by starting with a bracketing interval and repeatedly cutting those intervals in half. We show the bisection algorithm in Figure 2.

Method 2: Almost Newton

You might recall *Newton's method* from a calculus class. If we want to estimate a zero near x_k, we take the first-order Taylor expansion near x_k and set that equal to zero:

$$f(x_{k+1}) \approx f(x_k) + f'(x_k)(x_{k+1} - x_k) = 0.$$

With a little algebra, we have

$$x_{k+1} = x_k - f'(x_k)^{-1} f(x_k).$$

Note that if x_* is the actual root we seek, then Taylor's formula with remainder yields

$$0 = f(x_*) = f(x_k) + f'(x_k)(x_* - x_k) + \frac{1}{2} f''(\xi)(x_* - x_k)^2.$$

Now subtract the Taylor expansions for $f(x_{k+1})$ and $f(x_*)$ to get

$$f'(x_k)(x_{k+1} - x_*) + \frac{1}{2} f''(\xi)(x_k - x_*)^2 = 0.$$

This gives us an iteration for the error $e_k = x_k - x_*$:

$$e_{k+1} = -\frac{1}{2} \frac{f''(\xi)}{f'(x_k)} e_k^2.$$

Assuming that we can bound $f''(\xi)/f(x_k)$ by some modest constant C, this implies that a small error at e_k leads to a *really* small error $|e_{k+1}| \leq C|e_k|^2$ at the next step. This behavior, where the error is squared at each step, is *quadratic convergence.*

If we apply Newton iteration to $f(x) = dx - 1$, we get

$$x_{k+1} = x_k - \frac{dx_k - 1}{d} = \frac{1}{d}.$$

That is, the iteration converges in one step. But remember that we wanted to avoid division by d! This is actually not uncommon: often it is inconvenient to work with $f'(x_k)$, and so we instead cook up some approximation. In this case, let's suppose we have some $\hat{d}$ that is an integer power of two close to d. Then we can write a modified Newton iteration

$$x_{k+1} = x_k - \frac{dx_k - 1}{\hat{d}} = \left(1 - \frac{d}{\hat{d}}\right) x_k + \frac{1}{\hat{d}}.$$

Note that $1/d$ is a *fixed point* of this iteration:

$$\frac{1}{d} = \left(1 - \frac{d}{\hat{d}}\right) \frac{1}{d} + \frac{1}{\hat{d}}.$$

If we subtract the fixed point equation from the iteration equation, we have an iteration for the error $e_k = x_k - 1/d$:

$$e_{k+1} = \left(1 - \frac{d}{\hat{d}}\right) e_k.$$

So if $|d - \hat{d}|/|d| < 1$, the errors will eventually go to zero. For example, if we choose $\hat{d}$ to be the next integer power of two larger than d, then $|d - \hat{d}|/|\hat{d}| < 1/2$, and we get at least one additional binary digit of accuracy at each step.

When we plot the error in long division, bisection, or our modified Newton iteration on a semi-logarithmic scale, the decay in the error looks (roughly) like a straight line. That is, we have *linear* convergence. But we can do better!

Method 3: Actually Newton

We may have given up on Newton iteration too easily. In many problems, there are multiple ways to write the same nonlinear equation. For example, we can write the reciprocal of d as x such that $f(x) = dx - 1 = 0$, or we can write it as x such that $g(x) = x^{-1} - d = 0$. If we apply Newton iteration to g, we have

$$x_{k+1} = x_k - \frac{g(x_k)}{g'(x_k)} = x_k + x_k^2(x_k^{-1} - d) = x_k(2 - dx_k).$$

As before, note that $1/d$ is a fixed point of this iteration:

$$\frac{1}{d} = \frac{1}{d}\left(2 - d\frac{1}{d}\right).$$

Given that $1/d$ is a fixed point, we have some hope that this iteration will converge — but when, and how quickly? To answer these questions, we need to analyze a recurrence for the error.

We can get a recurrence for error by subtracting the true answer $1/d$ from both sides of the iteration equation and doing some algebra:

$$\begin{aligned} e_{k+1} &= x_{k+1} - d^{-1} \\ &= x_k(2 - dx_k) - d^{-1} \\ &= -d(x_k^2 - 2d^{-1}x_k + d^{-2}) \\ &= -d(x_k - d^{-1})^2 \\ &= -de_k^2 \end{aligned}$$

In terms of the relative error $\delta_k = e_k/d^{-1} = de_k$, we have

$$\delta_{k+1} = -\delta_k^2.$$

If $|\delta_0| < 1$, then this iteration converges — and once convergence really sets in, it is ferocious, roughly doubling the number of correct digits at each step. Of course, if $|\delta_0| > 1$, then the iteration diverges with equal ferocity. Fortunately, we can get a good initial guess in the same way we got a good guess for the modified Newton iteration: choose the first guess to be a nearby integer power of two.

On some machines, this sort of Newton iteration (intelligently started) is actually the preferred method for division.

The big picture

Let's summarize what we have learned from this example (and generalize slightly to the case of solving $f(x) = 0$ for more interesting f):

- *Bisection* is a general, robust strategy. We just need that f is continuous, and that there is some interval $[a, b]$ so that $f(a)$ and $f(b)$ have different signs. On the other hand, it is not always easy to get a bracketing interval; and once we do, bisection only halves that interval at each step, so it may take many steps to reach an acceptable answer. Also, bisection is an intrinsically one-dimensional construction.

- *Newton iteration* is a standard workhorse based on finding zeros of successive linear approximations to f. When it converges, it converges ferociously quickly. But Newton iteration requires that we have a derivative (which is sometimes inconvient), and we may require a good initial guess.
- A *modified Newton iteration* sometimes works well if computing a derivative is a pain. There are many ways we can modify Newton method for our convenience; for example, we might choose to approximate $f'(x_k)$ by some fixed value, or we might use a secant approximation.
- It is often convenient to work with *fixed point iterations* of the form

$$x_{k+1} = g(x_k),$$

where the number we seek is a fixed point of g ($x_* = g(x_*)$). Newton-like methods are an example of fixed point iteration, but there are others. Whenever we have a fixed point iteration, we can try to write an iteration for the error:

$$e_{k+1} = x_{k+1} - x_* = g(x_k) - g(x_*) = g(x_* + e_k) - g(x_*).$$

How easy it is to analyze this error recurrence depends somewhat on the properties of g. If g has two derivatives, we can write

$$e_{k+1} = g'(x_*)e_k + \frac{1}{2}g''(\xi_k)e_k^2 \approx g'(x_*)e_k.$$

If $g'(x_*) = 0$, the iteration converges *superlinearly.* If $0 < |g'(x_*)| < 1$, the iteration converges linearly, and $|g'(x_*)|$ is the rate constant. If $|g'(x_*)| > 1$, the iteration diverges.

Week 2: Wednesday, Feb 1

Use a routine, or roll your own?

The MATLAB function `fzero` is a fast, reliable black-box root-finding algorithm based on a combination of bisection (for safety) and interpolation-based methods (for speed). If you provide an initial interval containing exactly one zero, and if the root you seek is not too sensitive, `fzero` will find the root you seek to high accuracy (the default relative error tolerance is about $2\epsilon_{\mathrm{mach}}$). I use the function often, and recommend it to you.

That said, there are a few reasons to write your own root-finding algorithms, at least some of the time:

1. Not all the world is MATLAB, and you may sometimes find that you have to write these things yourself.
2. Black box approaches are far less useful for problems involving multiple variables. Consequently, it's worth learning to write Newton-like methods in one variable so that you can learn their properties well enough to work with similar algorithms in more than one variable.
3. Actually walking through the internals of a root-finding algorithm can be a terrific way to gain insight into how to formulate your problems so that a standard root finder can solve them.

Sensitivity and error

Suppose we want to find x_* such that $f(x_*) = 0$. On the computer, we actually have $\hat{f}(\hat{x}_*) = 0$. We'll assume that we're using a nice, robust code like `fzero`, so we have a very accurate zero of $\hat{f}$. But this still leaves the question: how well do $\hat{x}_*$ and x_* approximate each other? In other words, we want to know the sensitivity of the root-finding problem.

If $\hat{x}_* \approx x_*$, then

$$f(\hat{x}_*) \approx f'(x_*)(\hat{x}_* - x_*).$$

Using the fact that $\hat{f}(\hat{x}_*) = 0$, we have that if $|\hat{f} - f| < \delta$ for arguments near x_*, then

$$|f'(x_*)(\hat{x}_* - x_*)| \lesssim \delta.$$

This in turn gives us

$$|\hat{x}_* - x_*| \lesssim \frac{\delta}{f'(x_*)}.$$

Thus, if $f'(x_*)$ is close to zero, small rounding errors in the evaluation of f may lead to large errors in the computed root.

It's worth noting that if $f'(x_*) = 0$ (i.e. if x_* is a multiple root), that doesn't mean that x_* is completely untrustworthy. It just means that we need to take more terms in a Taylor series in order to understand the local behavior. In the case $f'(x_*) = 0$, we have

$$f(\hat{x}_*) \approx \frac{1}{2} f''(x_*)(\hat{x}_* - \hat{x}_*),$$

and so we have

$$|\hat{x}_* - \hat{x}_*| \leq \sqrt{\frac{2\delta}{f''(x_*)}}.$$

So if the second derivative is well behaved and δ is on the order of around 10^{-16}, for example, our computed $\hat{x}$ might be accurate to within an absolute error of around 10^{-8}.

Understanding the sensitivity of root finding is not only important so that we can be appropriately grim when someone asks for impossible accuracy. It's also important because it helps us choose problem formulations for which it is (relatively) easy to get good accuracy.

Choice of functions and variables

Root-finding problems are hard or easy depending on how they are posed. Often, the initial problem formulation is not the most convenient. For example, consider the problem of finding the positive root of

$$f(x) = (x + 1)(x - 1)^8 - 10^{-8}.$$

This function is terrifyingly uninformative for values close to 1. Newton's iteration is based on the assumption that a local, linear approximation provides a good estimate of the behavior of a function. In this problem, a linear approximation is terrible. Fortunately, the function

$$g(x) = (x + 1)^{1/8}(x - 1) - 10^{-1}$$

has the same root, which is very nicely behaved.

There are a few standard tricks to make root-finding problems easier:

- Scale the function. If $f(x)$ has a zero at x_*, so does $f(x)g(x)$; and sometimes we can analytically choose a scaling function to make the root finding problem easier.

- Otherwise transform the function. For example, in computational statistics, one frequently would like to maximize a likelihood function

  $$L(\theta) = \prod_{j=1}^{n} f(x_j; \theta)$$

  where $f(x; theta)$ is a probability density that depends on some parameter θ. One way to do this would be find zeros of $L'(\theta)$, but this often leads to scaling problems (potential underflow) and other numerical discomforts. The standard trick is to instead maximize the log-likelihood function

  $$\ell(\theta) = \sum_{j=1}^{n} \log f(x_j; \theta),$$

  often using a root finder for $\ell'(\theta)$. This tends to be a much more convenient form, both for analysis and for computation.

- Change variables. A good rule of thumb is to pick variables that are naturally *dimensionless*[1] For difficult problems, these dimensionless variables are often very small or very large, and that fact can be used to simplify the process of coming up with good initial guesses for Newton iteration.

Starting points

All root-finding software requires either an initial guess at the solution or an initial interval that contains the solution. This sometimes calls for a little cleverness, but there are a few standard tricks:

[1]Those of you who are interested in applied mathematics more generally should look up the Buckingham Pi Theorem — it's a tremendously useful thing to know about.

- If you know where the problem comes from, you may be able to get a good estimate (or bounds) by "application reasoning." This is often the case in physical problems, for example: you can guess the order of magnitude of an answer because it corresponds to some physical quantity that you know about.
- Crude estimates are often fine for getting upper and lower bounds. For example, we know that for all $x > 0$,

  $$\log(x) \leq x - 1$$

  and for all $x \geq 1$, $\log(x) > 0$. So if I wanted to $x + \log(x) = c$ for $c > 1$, I know that c should fall between x and $2x - 1$, and that gives me an initial interval. Alternatively, if I know that $g(z) = 0$ has a solution close to 0, I might try Taylor expanding g about zero – including higher order terms if needed – in order to get an initial guess for z.
- Sometimes, it's easier to find local minima and maxima than to find zeros. Between any pair of local minima and maxima, functions are either monotonically increasing or monotonically decreasing, so there is either exactly one root in between (in which case there is a sign change between the local min and max) or there are zero roots between (in which case there is no sign change). This can be a terrific way to start bisection.

Problems to ponder

1. Analyze the convergence of the fixed point iteration

   $$x_{k+1} = c - \log(x_k).$$

   What is the equation for the fixed point? Under what conditions will the iteration converge with a good initial guess, and at what rate will the convergence occur?
2. Repeat the previous exercise for the iteration $x_{k+1} = 10 - \exp(x_k)$.
3. Analyze the convergence of Newton's iteration on the equation $x^2 = 0$, where $x_0 = 0.1$. How many iterations will it take to get to a number less than 10^{-16}?

4. Analyze the convergence of the fixed point iteration $x_{k+1} = x_k - \sin(x_k)$ for x_k near zero. Starting from $x = 0.1$, how many iterations will it take to get to a number less than 10^{-16}?

5. Consider the cubic equation
$$x^3 - 2x + c = 0.$$
Describe a general purpose strategy for finding *all* the real roots of this equation for a given c.

6. Suppose we have some small number of samples $X_1, \ldots, X_m$ drawn from a Cauchy distribution with parameter θ (for which the pdf is)
$$f(x, \theta) = \frac{1}{\pi} \frac{1}{1 + (x - \theta)^2}.$$
The *maximum likelihood estimate* for θ is the function that maximizes
$$L(\theta) = \prod_{j=1}^{m} f(X_j, \theta).$$
Usually, one instead maximizes $l(\theta) = \log L(\theta)$ — why would this make sense numerically? Derive a MATLAB function to find the maximum likelihood estimate for θ by finding an appropriate solution to the equation $l'(\theta) = 0$.

7. The Darcy friction coefficient f for turbulent flow in a pipe is defined in terms of the Colebrook-White equation for large Reynolds number Re (greater than 4000 or so):
$$\frac{1}{\sqrt{f}} = -2 \log_{10} \left(\frac{\epsilon / D_h}{3.7} + \frac{2.51}{\mathrm{Re}\sqrt{f}} \right)$$
Here ϵ is the height of the surface roughness and D_h is the diameter of the pipe. For a 10 cm pipe with 0.1 mm surface roughness, find f for Reynolds numbers of 10^4, 10^5, and 10^6. Ideally, you should use a Newton iteration with a good initial guess.

8. A cable with density of 0.52 lb/ft is suspended between towers of equal height that are 500 ft apart. If the wire sags by 50 ft in between, find the maximum tension T in the wire. The relevant equations are

$$\begin{aligned} c + 50 &= c \cosh\left(\frac{500}{2c}\right) \\ T &= 0.52(c+50) \end{aligned}$$

Ideally, you should use a Newton iteration with a good initial guess.

Week 3: Monday, Feb 6

Subtle singularity

A square matrix $A \in \mathbb{R}^{n \times n}$ is called *invertible* or *nonsingular* if there is an A^{-1} such that $AA^{-1} = I$. Otherwise, A is called *singular*. There are several common ways to characterize nonsingularity: A is nonsingular if it has an inverse, if $\det(A) \neq 0$, if $\operatorname{rank}(A) = n$, or if $\operatorname{null}(A) = \{0\}$. What would happen if we tried to test these conditions numerically?

1. A has an inverse. How do we compute it? Is it sensitive to roundoff?
2. $\det(A) \neq 0$. How do we compute determinants? The usual Laplace expansion (also called the cofactor expansion) is very expensive for large n! Also, consider what happens for $A = I/16$ when $n = 100$.
3. $\operatorname{rank}(A) = n$. How do we compute the rank? We might look for a basis for the range space; how do we get that? Is this computation sensitive to roundoff?
4. $\operatorname{null}(A) = \{0\}$. How do we compute the null space of a matrix? Is the computation sensitive to roundoff?

Even if A is singular, almost every matrix $\hat{A}$ close to A will be nonsingular. Since we usually perturb problems just by storing them in floating point, it may be too much to ask whether an interesting matrix is *exactly* singular, or to ask for the true rank. It turns out to be much more practical to ask whether A is *close to* singular and whether there is an *almost* null space. It also turns out that some constructions that look straightforward to compute, such as explicit inverses and determinants, are poorly-behaved in floating point, and so are rarely used in computational practice[1].

[1] At least, they are rarely used by people who took a class like this one and paid some attention. They are often used in codes written by people who have never taken such a class; and when codes like that break, people sometimes knock on my door. I sometimes grumble about this, but I suppose I should consider it job security.

Matrices and vectors in Matlab

Vectors and matrices are basic objects in numerical linear algebra[2]. They are also basic objects in MATLAB. For example, we can write a column vector[3] $x \in \mathbb{R}^3$ as

```
x = [1; 2; 3];
```

and a matrix $A \in \mathbb{R}^{4 \times 3}$ as

```
A = [1, 5, 9;
     2, 6, 10;
     3, 7, 11;
     4, 8, 12];
```

Internally, MATLAB uses *column major* layout — all the entries of the first column of a matrix are listed first in memory, then all the entries of the second column, and so on. This is actually visible at the user level in some contexts. For example, when I enter A as above, the MATLAB expression `A(6)` evaluates to 6; and if I write

```
fprintf('%d\n', A);
```

the output is the numbers 1 through 12, one per line.

I can multiply matrices and vectors with compatible dimensions using the ordinary multiplication operator:

```
y = A*x; % Computes y = [38; 44; 50; 56]
```

The tic operator in MATLAB computes the (conjugate) transpose of a matrix or a vector. For example:

```
b = [1; 2];  % b is a column vector
bt = b';     % bt = [1, 2] is a row vector

C = [1, 2; 3, 4];
Ct = C'; % Ct = [1, 3; 2, 4]; Ct(i,j) is C(j,i)
```

If x and y are two vectors, we can define their *inner product* (also called the *scalar product* or *dot product*) and *outer product* in terms of ordinary matrix multiplication and transposition:

[2]I suppose abstract linear maps are more basic than matrices — but you have to have matrices to compute.

[3]In this class, the word "vector" with no qualifiers will usually mean "column vector." If I want to refer to a row vector, I will write "row vector."

```
x = [1; 2];
y = [3; 4];
dotxy = x'*y;  % Inner product is 1*3 + 2*4 = 11
outer = x*y';  % Outer product is [3, 6;, 4, 8]
```

If I want to apply the inverse of a square matrix C, I can use the backslash (solve) operator

```
C = [1, 2; 3, 4];
b = [1; 2];
z = C\b; % Computes z = [0; 0.5]. Better than z = inv(C)*b.
```

Most expressions that involve a matrix inverse can be rewritten in terms of the backslash operator, and backslash is *almost always* preferable to the `inv` command.

I can take *slices* of matrices using MATLAB's colon syntax. For example, if I write

```
I = eye(6);
e3 = I(:,3);
```

then `e3` denotes e_3, the vector which is all zeros except for the third entry.

The costs of computations

Our first goal in any scientific computing task is to get a sufficiently accurate answer. Our second goal is to get it fast enough[4]. Of course, there is a tradeoff between the computer time and our time; and often, we can optimize both by making wise high-level decisions about the type of algorithm we should use, and then calling an appropriate library routine. At the same time, we need to keep track of enough details so that we don't spend days on end twiddling our thumbs and waiting for a computation that should have taken a few seconds. It is easy to goof and write slow MATLAB code. Fortunately, MATLAB has a *profiler* that can help us find where our code is spending all its time; for details, type `help profile` at the command line. Unfortunately, it doesn't always help us to know *where* we are spending a lot of time if we don't know *why*.

[4]If you really like thining about how to make things run fast enough, you might enjoy CS 5220: Applications of Parallel Computers. I'll be teaching it in the fall.

The work to multiply an $m \times n$ matrix by an $n \times p$ matrix is $O(mnp)$. If $A \in \mathbb{R}^{n\times n}$ and $B \in \mathbb{R}^{n\times p}$ are general (dense) MATLAB matrices, then the work to compute $A^{-1}B$ using the backslash operator is $O(n^3+n^2p)$[5]. Because matrix multiplication is *associative*, $(AB)C$ and $A(BC)$ are mathematically equivalent; but they can have very different performance depending on the matrix sizes. For example, if $x, y, z \in \mathbb{R}^n$ are three vectors ($n \times 1$ matrices), then evaluating $(xy^T)z$ takes $O(n^2)$ arithmetic and storage ($O(n^2)$ arithmetic and storage for the outer product and $O(n^2)$ arithmetic to multiply by z). But the equivalent expression $x(y^Tz)$ takes only $O(n)$ arithmetic and storage: $O(n)$ arithmetic and one element of storage to compute the inner product, followed by $O(n)$ arithmetic and storage to multiply x by a scalar.

Because equivalent mathematical expressions can have very different performance characteristics, it is useful to remember some basic algebraic properties of simple matrix operations:

$$\begin{aligned}
(AB)C &= A(BC) \\
(AB)^T &= B^T A^T \\
(AB)^{-1} &= B^{-1}A^{-1} \\
A^{-T} &\equiv (A^{-1})^T = (A^T)^{-1}
\end{aligned}$$

It is also helpful to remember that some matrix operations can be written more efficiently without forming an explicit matrix. For example, the following codes are equivalent:

```
% Inefficient (O(n^2))
y = diag(s)*x;     % Multiply x by a diagonal scaling matrix
z = (c*eye(n))*x;  % Multiply x by c*I

% Efficient
y = s.*x;          % .* is componentwise multiplication
z = c*x;           % Can omit multiplication by an identity
```

In addition to poor choices of parentheses, we can get terrible performance in MATLAB if we ignore silent costs. But we can also get surprisingly good

[5]The backslash operator is actually very sophisticated, and it will take advantage of any structure it can find in your matrix. If the matrix A is triangular, MATLAB will compute $A^{-1}B$ in $O(n^2p)$ time; if A is represented using MATLAB's sparse matrix features, the cost can be even lower.

performance if we play to MATLAB's strength in vector operations[6]. For example:

```
% Inefficient (O(n^2) data transfer operations)
results = [];
for k = 1:n
  results(k) = foo(k);   % Allocates a length k+1 array, copies old data in
end

% More efficient (no silent memory costs)
results = zeros(1,n);   % Pre-allocate storage
for k = 1:n
  results(k) = foo(k);
end

% Most efficient if foo is vectorized
results = foo(1:n);
```

People sometimes think MATLAB must be slow compared to a language like Java or C. But for matrix computations, well-written MATLAB is often faster than all but very carefully tuned code in a compiled language. That is because MATLAB uses very fast libraries for linear algebra operations like matrix multiplication and linear solves. Most of our codes in this class will be fast to the extent that we can take advantage of these libraries.

[6]Recent versions of MATLAB pre-compile scripts into byte code, and the compiler has an optimizer. Consequently, recent versions of MATLAB have better loop performance than older versions, particularly when the loops have simple structures that the optimizer can figure out.

Week 3: Wednesday, Feb 8

Spaces and bases

I have two favorite vector spaces[1]: $\mathbb{R}^n$ and the space $\mathcal{P}_d$ of polynomials of degree at most d. For $\mathbb{R}^n$, we have a *canonical basis*:

$$\mathbb{R}^n = \operatorname{span}\{e_1, e_2, \ldots, e_n\},$$

where e_k is the kth column of the identity matrix. This basis is frequently convenient both for analysis and for computation. For $\mathcal{P}_d$, an obvious-seeming choice of basis is the *power basis*:

$$\mathcal{P}_d = \operatorname{span}\{1, x, x^2, \ldots, x^d\}.$$

But this obvious-looking choice turns out to often be terrible for computation. Why? The short version is that powers of x aren't all that strongly linearly dependent, but we need to develop some more concepts before that short description will make much sense.

The *range space* of a matrix or a linear map A is just the set of vectors y that can be written in the form $y = Ax$. If A is *full (column) rank*, then the columns of A are linearly independent, and they form a basis for the range space. Otherwise, A is *rank-deficient*, and there is a non-trivial *null space* consisting of vectors x such that $Ax = 0$.

Rank deficiency is a delicate property[2]. For example, consider the matrix

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}.$$

This matrix is rank deficient, but the matrix

$$\hat{A} = \begin{bmatrix} 1+\delta & 1 \\ 1 & 1 \end{bmatrix}.$$

is not rank deficient for any $\delta \neq 0$. Technically, the columns of $\hat{A}$ form a basis for $\mathbb{R}^2$, but we should be disturbed by the fact that $\hat{A}$ is so close to a singular matrix. We will return to this point in some detail next week.

[1]This is a fib, but not by too much.

[2]Technically, we should probably say that rank deficiency is *non-generic* rather than "delicate."

Norm!

In order to talk sensibly about a matrix being "close to" singular or a basis being "close to" linear dependence, we need the right language.

First, we need the concept of a *norm*, which is a measure of the length of a vector. A norm is a function from a vector space into the real numbers with three properties

1. Positive definiteness: $\|x\| > 0$ when $x \neq 0$ and $\|0\| = 0$.
2. Homogeneity: $\|\alpha x\| = |\alpha|\|x\|$.
3. Triangle inequality: $\|x + y\| \leq \|x\| + \|y\|$.

One of the most popular norms is the Euclidean norm (or 2-norm):

$$\|x\|_2 = \sqrt{\sum_{i=1}^{n} |x_i|^2} = \sqrt{x^T x}.$$

We will also use the 1-norm and the ∞-norm (*a.k.a.* the max norm or the Manhattan norm):

$$\|x\|_1 = \sum_i |x_i|.$$
$$\|x\|_\infty = \max_i |x_i|$$

Second, we need a way to relate the norm of an input to the norm of an output. We do this with matrix norms. Matrices of a given size form a vector space, so in one way a matrix norm is just another type of vector norm. However, the most useful matrix norms are *consistent* with vector norms on their domain and range spaces, i.e. for all vectors x in the domain,

$$\|Ax\| \leq \|A\|\|x\|.$$

Given norms for vector spaces, a commonly-used consistent norm is the *induced* norm (operator norm):

$$\|A\| \equiv \max_{x \neq 0} \frac{\|Ax\|}{\|x\|} = \max_{\|x\|=1} \|Ax\|.$$

The matrix 1-norm and the matrix ∞-norm (the norms induced by the vector 1-norm and vector ∞-norm) are:

$$\|A\|_1 = \max_j \left(\sum_i |a_{ij}| \right) \quad \text{(max abs column sum)}$$

$$\|A\|_\infty = \max_j \left(\sum_i |a_{ij}| \right) \quad \text{(max abs row sum)}$$

If we think of a vector as a special case of an n-by-1 matrix, the vector 1-norm matches the matrix 1-norm, and likewise with the ∞-norm. This is how I remember which one is the max row sum and which is the max column sum!

The matrix 2-norm is very useful, but it is actually much harder to compute than the 1-norm or the ∞-norm. There is a related matrix norm, the Frobenius norm, which is much easier to compute:

$$\|A\|_F = \sqrt{\sum_{i,j} |a_{ij}^2|}.$$

The Frobenius norm is consistent, but it is not an operator norm[3]

MATLAB allows us to compute all the vector and matrix norms describe above with the `norm` command. For example, `norm(A, 'fro')` computes the Frobenius norm of a matrix A, while `norm(x,1)` computes the 1-norm of a vector x. The default norm, which we get if we just write `norm(A)` or `norm(x)`, is the Euclidean vector norm (*a.k.a.* the 2-norm) and the corresponding operator norm.

The ideas of vector norms and operator norms make sense on spaces other than $\mathbb{R}^n$, too. For example, one choice of norms for $\mathcal{P}_d$ is

$$\|p\|_{L^2([-1,1])} = \sqrt{\int_{-1}^{1} p(x)^2 \, dx}.$$

You will note that this looks an awful lot like the standard Euclidean norm; we also have analogues of the 1-norm and the ∞-norm in this case. The norms for spaces of functions (like $\mathcal{P}_d$) are actually a more interesting topic than the norms of $\mathbb{R}^n$, but an extended discussion is (lamentably) beyond the scope of what I can reasonably fit into this course.

[3]The first half of this sentence is basically Cauchy-Schwarz; the second half of the sentence can be seen by looking at $\|I\|_F$. If you don't understand this footnote, no worries.

Inner products

Norms are the tools we need to measure lengths and distances. *Inner products* are the tools we need to measure angles. In general, an inner product satisfies three axioms:

- *Positive definiteness*: $\langle u, u \rangle \geq 0$, with equality iff $u = 0$.
- *Symmetry*: $\langle u, v \rangle = \overline{\langle v, u \rangle}$
- *Linearity*: $\langle \alpha u, v \rangle = \alpha \langle u, v \rangle$ and $\langle u_1 + u_2, v \rangle = \langle u_1, v \rangle + \langle u_2, v \rangle$.

For every inner product, we have an associated norm: $\|u\| = \sqrt{\langle u, u \rangle}$. An important identity relating the inner product to the norm is the *Cauchy-Schwartz* inequality:

$$\langle u, v \rangle \leq \|u\| \|v\|.$$

Equality holds only if u and v are parallel. Vectors u and v are *orthogonal* if $\langle u, v \rangle = 0$. In general, the angle α between nonzero vectors u and v is *defined* by the relation

$$\cos(\alpha) = \frac{\langle u, v \rangle}{\|u\| \|v\|}.$$

If x and y are in $\mathbb{R}^n$, the standard inner product is:

$$\langle x, y \rangle = x^T y = \sum_{i=1}^{n} x_i y_i.$$

We say vectors $u_1, u_2, \ldots, u_k$ are *orthonormal* if they mutually orthogonal and have unit Euclidean length, i.e.

$$\langle u_i, u_j \rangle = \delta_{ij} = \begin{cases} 1, & i = j \\ 0, & \text{otherwise.} \end{cases}$$

Somewhat oddly, though, we define an *orthogonal matrix* to be a square matrix whose columns are orthonormal (i.e. a matrix Q such that $Q^T Q = I$). When we say a matrix is orthogonal, we usually really mean "orthogonal with respect to the standard inner product on $\mathbb{R}^n$"; if the matrix is orthogonal with respect to some other inner product, we say so explicitly.

One very useful property of orthogonal matrices is that they *preserve Euclidean length.* That is, if Q is orthogonal, then

$$\|Qx\|^2 = (Qx)^T (Qx) = x^T Q^T Q x = x^T x = \|x\|^2.$$

From time to time, I may talk about "unitary operations"; if I do, I generally mean linear maps that have this property of preserving Euclidean length[4]

Of course, other spaces can also have useful inner products. For example, a standard choice of inner products for $\mathcal{P}_d$ is

$$\langle p, q \rangle_{L^2([-1,1])} = \int_{-1}^{1} p(x)q(x)\, dx.$$

The power basis $\{1, x, x^2, \ldots, x^d\}$ is decidedly *not* orthonormal with respect to this inner product. On the other hand the *Legendre polynomials*, which play a critical role in the theory of Gaussian integration, do form an orthogonal basis for $\mathcal{P}_d$ with respect to this inner product.

Symmetric matrices and quadratic forms

The multi-dimensional version of Taylor's theorem says that we can write any sufficiently nice function from $\mathbb{R}^n \to \mathbb{R}$ as

$$f(x_0 + z) = f(x_0) + \sum_i \frac{\partial f}{\partial x_i} z_i + \frac{1}{2} \sum_{i,j} \frac{\partial^2 f}{\partial x_i \partial x_j} z_i z_j + O(\|z\|^3).$$

We sometimes write this more concisely as

$$f(x_0 + z) = f(x_0) + \nabla f(x_0)^T z + \frac{1}{2} z^T H_f(x_0) z + O(\|z\|^3),$$

where the *Hessian matrix* $H_f(x_0)$ has entries which are second partials of f at x_0. Still assuming that f is nice, we have that

$$(H_f(x_0))_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{\partial^2 f}{\partial x_j \partial x_i} = (H_f(x_0))_{ji}\,;$$

that is, the Hessian matrix is *symmetric.*

A *quadratic form* on $\mathbb{R}^n$ is function of the form

$$\phi(x) = x^T A x.$$

[4] I'll expect you to know what an orthogonal matrix is going forward, but if I ever say "unitary operation" and you forget what I mean, just ask me.

We typically assume A is symmetric, since only the symmetric part of the matrix matters.[5] Quadratic forms show up frequently throughout applied mathematics, partly because second-order Taylor expansions show up frequently. Symmetric matrices also show up more-or-less constantly; and when they do, there is often a quadratic form lurking behind the scenes.

A symmetric matrix A is *positive definite* if the corresponding quadratic form $\phi(x) = x^T Ax$ is positive definite — that is, $\phi(x) \geq 0$ for all x, with equality only at $x = 0$. You've likely seen the notion of positive definiteness before in multivariable calculus: if a function f has a critical point at x_0 and $H_f(x_0)$ is positive definite, then x_0 is a local minimum. You've also seen the notion of positive definiteness earlier in these notes, since the quadratic form associated with an inner product ($\|u\|^2 = \langle u, u\rangle$) must be positive definite. Matrices that are symmetric and positive definite occur so frequently in numerical linear algebra that we often just call them SPD matrices[6].

Quadratic forms are characterized by the fact that they are quadratic; that is, $\phi(\alpha x) = \alpha^2\phi(x)$. It is sometimes convenient to get rid of the effects of scaling vectors, and so we define the *Rayleigh quotient*:

$$\rho_A(x) = \frac{x^T Ax}{x^T x}.$$

It is interesting to differentiate $\rho_A(x)$ to try to find critical points:

$$\begin{aligned} \frac{d}{dt}\rho_A(x+tw) &= \frac{w^T Ax + x^T Aw}{x^T x} - \frac{(x^T Ax)(w^T x + x^T w)}{(x^T x)^2} \\ &= \frac{2w^T}{x^T Ax}\left(Ax - \rho_A(x)x\right). \end{aligned}$$

At a critical point, where all the directional derivatives are zero, we have

$$Ax = \rho_A(x)x,$$

i.e. x is an *eigenvector* and $\rho_A(x)$ is an *eigenvalue*. This connection between eigenvalues of symmetric matrices and ratios of quadratic forms is immensely powerful. For example, we can use it to characterize the operator two-norm

$$\|A\|_2^2 = \max_{x \neq 0} \frac{\|Ax\|^2}{\|x\|^2} = \max_{x\neq 0} \frac{x^T A^T Ax}{x^T x} = \lambda_{\max}(A^T A)$$

[5]The symmetric part of a general matrix A is $(A + A^T)/2$.

[6]Abbreviations are our way of stalling RSI. Why do you think CS has so many TLAs?

The other eigenvalues of $A^T A$ (the squared *singular values*) are also sometimes handy, and we'll talk about them later.

We can also look at the eigenvalues of a symmetric matrix A to determine whether the corresponding quadratic form is positive definite (all eigenvalues of A positive), negative definite (all eigenvalues of A negative), or indefinite.

Problems to ponder

1. We said earlier that
$$\|A\| \equiv \max_{x \neq 0} \frac{\|Ax\|}{\|x\|} = \max_{\|x\|=1} \|Ax\|.$$
Why is the equality true?

2. What are the range and null space of $\frac{d}{dx}$ viewed as a linear operator acting on $\mathcal{P}_d$? In terms of the power basis, how might you write $\frac{d}{dx}$ as a matrix?

3. Using the inner product $\langle \cdot, \cdot \rangle_{L^2([-1,1])}$, what is the angle between the monomials x^j and x^k?

4. The Cauchy-Schwartz inequality says
$$\langle u, v \rangle \leq \|u\|\|v\|.$$
The easiest way I know to prove Cauchy-Schwartz is to write
$$\phi(t) = \langle u + tv, u + tv \rangle \geq 0,$$
then use the properties of inner products to write $\phi(t)$ as a quadratic function in t with coefficients given in terms of $\|u\|^2$, $\|v\|^2$, and $\langle u, v \rangle$. Do this expansion, and write the discriminant of the resulting quadratic. This discriminant must be non-positive in order for $\phi(t)$ to be non-negative for all values of t; using this fact, show that Cauchy-Schwartz must hold.

5. Given matrices $X, Y \in \mathbb{R}^{m \times n}$, we define the *Frobenius inner product* to be
$$\langle X, Y \rangle = \mathrm{tr}(X^T Y),$$
where $\mathrm{tr}(A)$ is the sum of the diagonal elements of A. Argue that this is an inner product, and that the associated norm is the Frobenius norm.

6. Show that when we have a norm induced by an inner product,
$$(\|u+v\|^2 - \|u-v\|^2)/4 = \langle u, v \rangle$$
7. Show that the operation $p(x) \mapsto p(-x)$ is unitary for $\mathcal{P}_d$ with the inner product $L^2([-1,1])$.
8. Show that if A is an SPD matrix, then
$$\langle x, y \rangle_A = x^T A y$$
is a valid inner product (sometimes called an *energy inner product*).
9. Assuming A is symmetric, define
$$\psi(x) = \left(\frac{1}{2} x^T A x - x^T b \right).$$
Give an expression for the directional derivatives
$$\frac{d}{dt} \psi(x + tu).$$
What equation must be satisfied at a critical point (i.e. a point where all the directional derivatives are zero)?

Week 4: Monday, Feb 13

Gaussian elimination in matrix terms

To solve the linear system

$$\begin{bmatrix} 4 & 4 & 2 \\ 4 & 5 & 3 \\ 2 & 3 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \\ 5 \end{bmatrix},$$

by Gaussian elimination, we start by subtracting multiples of the first row from the remaining rows in order to introduce zeros in the first column, thus eliminating variable x_1 from consideration in the last two questions. We then repeat this procedure, so that we end up with a list of equations where the first equation involves x_1, x_2, and x_3, the second equation involves x_2 and x_3, and the third equation involves only x_3.

We can summarize the transformations to the equations in terms of matrix operations. First, we subtract one times the first row from the second row and 0.5 times the first row from the third row:

$$M_1 A = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ -0.5 & 0 & 1 \end{bmatrix} \begin{bmatrix} 4 & 4 & 2 \\ 4 & 5 & 3 \\ 2 & 3 & 3 \end{bmatrix} = \begin{bmatrix} 4 & 4 & 2 \\ 0 & 1 & 1 \\ 0 & 1 & 2 \end{bmatrix}.$$

Then we subtract the new second row from the new third row:

$$M_2 M_1 A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} 4 & 4 & 2 \\ 0 & 1 & 1 \\ 0 & 1 & 2 \end{bmatrix} = \begin{bmatrix} 4 & 4 & 2 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

If we write the original system as $Ax = b$, what we have just shown is that $M_2 M_1 A = U$, where M_1 and M_2 are simple unit lower triangular matrices (sometimes called elementary transformations or Gauss transformations) and U is upper triangular. The equation $M_2 M_1 A x = M_2 M_1 b$ thus looks like

$$\begin{bmatrix} 4 & 4 & 2 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ 3 \end{bmatrix},$$

and we can compute x by *back-substitution*:

$$\begin{aligned} x_3 &= 3 &\implies x_3 &= 3 && \text{from the third row} \\ x_2 + x_3 &= 1 &\implies x_2 &= -2 && \text{from the second row} \\ 4x_1 + 4x_2 + 2x_3 &= 2 &\implies x_1 &= 1 && \text{from the first row.} \end{aligned}$$

In matrix terms, we can rewrite $M_2 M_1 A = U$ as

$$A = LU$$

where

$$L = M_1^{-1} M_2^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0.5 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0.5 & 1 & 1 \end{bmatrix}.$$

Notice that the subdiagonal elements of L are just the multipliers that we encountered during Gaussian elimination. This *matrix factorization* is what numerical analysts would usually call "Gaussian elimination". Once we have L and U, computing $A^{-1}b$ boils down to computing $L^{-1}b$ (forward substitution) followed by $U^{-1}(L^{-1}b)$ (back substitution).

Using LU factorization

When you write `x = A\b` for a general dense matrix `A` in MATLAB, two things happen:

1. MATLAB computes the factorization $PA = LU$. Here, P is a permutation matrix – this *row pivoting* just corresponds to re-ordering the equations during Gaussian elimination in order to improve numerical stability. This phase costs $O(n^3)$ time.

2. MATLAB then permutes the entries of b and solves the triangular systems $Lc = b$ and $Uc = x$ by forward and backward substitution, respectively. This phase costs $O(n^2)$ time.

You can separate these two phases into two MATLAB calls:

```
[L,U,P] = lu(A);  % This is O(n^3)
x = U\(L\(P*b)); % This is O(n^2)
```

In the second line, MATLAB is smart enough to recognize that L and U are triangular matrices, so that linear systems with them can be solved by forward and back substitution. Thus, most of the work takes place in the first line (the factorization). If you want to solve multiple linear systems involving the matrix A, it is very helpful to use the `lu` function so that you don't have to do the work of factoring A more than once.

Sparse matrices

A matrix A is *sparse* if most of the coefficients a_{ij} are zero. Sparse matrices occur frequently in practice, and they will play an important role in the first class project. MATLAB provides a compact storage support for sparse matrices, and also includes fast matrix multiplication and Gaussian elimination routines for use with sparse matrices. We can create a sparse matrix in MATLAB using the `sparse` command. There are several variants of `sparse`, but perhaps the simplest takes the positions and values of the nonzero elements in parallel vector arguments. For example, the commands

```
i = [1, 2, 3,  4,  1];
j = [1, 2, 3,  4,  4];
a = [5, 7, 9, 11, 13];
A = sparse(i,j,a);
```

produce the sparse matrix

$$A = \begin{bmatrix} 5 & 0 & 0 & 13 \\ 0 & 7 & 0 & 0 \\ 0 & 0 & 9 & 0 \\ 0 & 0 & 0 & 11 \end{bmatrix}.$$

Internally, MATLAB stores sparse matrices in a format that tracks only the positions and values of the nonzero entries. This *compressed sparse column* format is much like a packed version of the adjacency list representation used to store sparse graphs. If A is sparse, then the storage for A and the time to compute a matrix-vector product with A are both proportional to the number of nonzeros in A (sometimes written nnz(A)).

Like the multiplication operator, the `lu` command and the backslash operator in MATLAB are overloaded to handle sparse matrix inputs. However, the complexity of solving a linear system involving a sparse matrix is much more difficult to characterize than the complexity of sparse matrix-vector multiplication. Depending on the order in which variables are eliminated and the topology of the graph associated with the matrix A, Gaussian elimination may produce matrices L and U which have many more nonzeros than the matrix A. These extra nonzeros are called *fill*. In order to minimize fill during factorization of sparse matrices, we would typically reorder the variables, just as we reorder the equations in order to improve numerical stability. That is, in the sparse case we usually write $PAQ = LU$ where P is a row permutation chosen by partial pivoting and Q is a column permutation chosen to reduce fill. In general, choosing the Q that minimizes the fill is an NP-hard problem, but we have good heuristics.

If A is a MATLAB matrix stored in the sparse format, then, we can solve linear systems with A either by writing `x = A\b` or

```
[L,U,P,Q] = lu(A);
x = Q*(U\(L\(P*b)));
```

As in the dense case, factoring the matrix A with `lu` is usually much more expensive than the triangular solves with L and U.

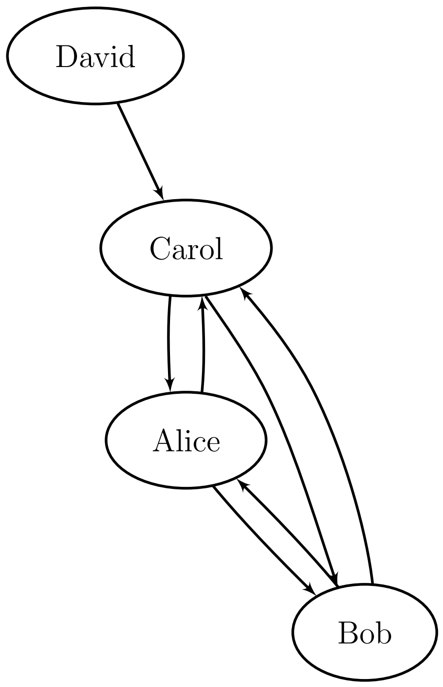
Opinions and projects

The first project for the class involves some social network analysis. We have a model of opinion formation, and we would like to quantify the sensitivity of the mean opinion to the model parameters. At the heart of the project are a number of manipulations using MATLAB sparse matrices. In particular, we have a matrix A that characterizes how people in the network weight the opinions of their peers and their own intrinsic beliefs in order to form their expressed opinions. In matrix terms, we have

$$Ax = s$$

where A reflects the connectivity among individuals, x is a vector of expressed opinions, and s represents intrinsic beliefs. We show a small example in Figure 1.

For the purposes of the project, we will assume that the structure of A is such that sparse factorization routines work well. For a general social network, though, it may not be feasible to factor A. In this case, we would typically turn to *iterative* methods to solve the linear system $Ax = s$. We will speak of such methods in more detail later in the course.



$$A = \begin{bmatrix} a_{AA} & -w_{AB} & -w_{AC} & 0 \\ -w_{BA} & a_{BB} & -w_{BC} & 0 \\ -w_{CA} & -w_{CB} & a_{CC} & 0 \\ 0 & 0 & -w_{DC} & a_{DD} \end{bmatrix}$$

$$\begin{aligned} a_{AA} &= w_{AB} + w_{AC} + 1 \\ a_{BB} &= w_{BA} + w_{BC} + 1 \\ a_{CC} &= w_{CA} + w_{CB} + 1 \\ a_{DD} &= w_{DC} + 1 \end{aligned}$$

Figure 1: Small example of opinion formation in a social network. If Alice, Bob, Carol, and David have "intrinsic" opinions s_A, s_B, s_C, s_D, then we compute the "espressed" opinions x_A, x_B, x_C, x_D according to $Ax = s$.

Week 4: Wednesday, Feb 15

A summary

From Monday, you should have learned:

1. Gaussian elimination can be seen as the computation of a matrix factorization $PA = LU$, where L is a unit lower triangular matrix L whose entries are the multipliers used in the elimination process; U is an upper triangular matrix; and P is a permutation matrix corresponding to row re-ordering during partial pivoting.

2. Solving a linear system by Gaussian elimination consists of two steps: factoring the matrix (which costs $O(n^3)$) and solving triangular systems with forward and backward substitution (which costs $O(n^2)$).

3. Most of the entries in a *sparse* matrix are zero. We can represent a sparse matrix compactly by only storing the location and values of the nonzero entries. Gaussian elimination on sparse matrices *sometimes* yields sparse factors, but the order of elimination matters. The MATLAB call

   ```
   [L,U,P,Q] = lu(A);
   ```

   factors a sparse matrix A as $PAQ = LU$, where P, L, and U are as before, and the permutation matrix Q is automatically computed in order to try to keep L and U sparse.

Today, we'll look at

1. Condition numbers and some basic error analysis for linear systems.

2. *Cholesky* factorization for symmetric, positive definite matrices.

3. How Cholesky factorization actually gets implemented.

Partial pivoting

I only said a little last time about the role of the permutation matrix P in the factorization. The reason that P is there is to help control the size of the elements in L. For example, consider what happens when we factor the following matrix without pivoting:

$$A = \begin{bmatrix} \epsilon & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ \epsilon^{-1} & 1 \end{bmatrix} \begin{bmatrix} \epsilon & 1 \\ 0 & 1-\epsilon^{-1} \end{bmatrix}.$$

If we round u_{22} to $-\epsilon^{-1}$, then we have

$$\begin{bmatrix} 1 & 0 \\ \epsilon^{-1} & 1 \end{bmatrix} \begin{bmatrix} \epsilon & 1 \\ 0 & -\epsilon^{-1} \end{bmatrix} = \begin{bmatrix} \epsilon & 1 \\ 1 & 0 \end{bmatrix} \neq A;$$

that is, a rounding error in the (huge) u_{22} entry causes a complete loss of information about the a_{22} component.

In this example, the l_{21} and u_{22} terms are both huge. Why does this matter? When L and U have huge entries and A does not, computing the product LU must inevitably involve huge cancellation effects, and we have already seen the danger of cancellation in previous lectures. The *partial pivoting* strategy usually used with Gaussian elimination permutes the rows of A so that the multipliers at each step (the coefficients of L) are at most one in magnitude. Even with this control on the elements of L, it is still possible that there might be “pivot growth”: that is, elements of U might grow much larger than those in A. But while it is possible to construct test problems for which pivot growth is exponential, in practice such cases almost never happen.

Alas, even when GEPP works well, it can produce answers with large relative errors. In some sense, though, the fault lies not in our algorithms, but in our problems. In order to make this statement precise, we need to return to a theme from the first week of classes, and discuss condition numbers.

Warm-up: Error in matrix multiplication

Suppose $\hat{y} = (A+E)x$ is an approximation to $y = Ax$. What is the error in using $\hat{y}$ to approximate y? We can write an equation for the absolute error:

$$\hat{y} - y = Ex,$$

and using norms, we have

$$\|\hat{y} - y\| \leq \|E\|\|x\|.$$

This is all well and good, but we would really like an expression involving *relative* errors. To get such an expression, it's helpful to play around with norms a little more. Assuming A is invertible, we have

$$\|x\| = \|A^{-1}y\| \leq \|A^{-1}\|\|y\|,$$

so that

$$\frac{\|\hat{y} - y\|}{\|y\|} \leq \|A\|\|A^{-1}\|\frac{\|E\|}{\|A\|}.$$

That is, the quantity

$$\kappa(A) = \|A\|\|A^{-1}\|$$

serves as a *condition number* that relates the size of relative error in the computed result $\hat{y}$ to the size of relative error in the matrix A. Note that this condition number is a function of the *problem formulation*, and does not depend on the way that we implement matrix multiplication.

It is a straightforward (if tedious) exercise in rounding error analysis to show that if we compute $y = Ax$ in the usual way in floating point arithmetic, the computed result $\hat{y}$ will actually satisfy $\hat{y} = (A + E)x$, where $|E_{ij}| \lesssim n\epsilon_{\mathrm{mach}}|A_{ij}|$. That is, $\hat{y}$ is the *exact* result for a slightly perturbed problem. The perturbation E is called a *backward error*. For the matrix norms we have discussed, this element-wise inequality implies the norm inequality $\|E\|/\|A\| \leq n\epsilon$. Thus, the relative error in matrix multiplication is bounded by

$$\frac{\|\hat{y} - y\|}{\|y\|} \leq \kappa(A) \cdot n\epsilon.$$

Since the numerical computation always has a small backward error, we say the algorithm is *backward stable* (or sometimes just *stable*). If the problem is additionally *well-conditioned* (so that $\kappa(A)$ is small), then the *forward relative error* $\|\hat{y} - y\|/\|y\|$ will be small. But if the condition number is large, the forward error may still be big.

From multiplication to linear solves

Now suppose that instead of computing $y = Ax$, we want to solve $Ax = b$. How sensitive is this problem to changes in A? We know already how to differentiate A^{-1} with respect to changes in A; using this knowledge, we can write a first-order sensitivity formula relating small changes δA in the system matrix to small changes δx in the solution:

$$\delta x \approx -A^{-1}(\delta A)A^{-1}b = -A^{-1}(\delta A)x.$$

Taking norms gives

$$\|\delta x\| \lesssim \|A^{-1}\|\|\delta A\|\|x\|,$$

which we can rearrange to get

$$\frac{\|\delta x\|}{\|x\|} \lesssim \kappa(A)\frac{\|\delta A\|}{\|A\|}.$$

That is, the condition number $\kappa(A) = \|A\|\|A^{-1}\|$ once again relates relative error in the matrix to relative error in the result. Another very useful result is that

$$\frac{\|\hat{x} - x\|}{\|x\|} \lesssim \kappa(A)\frac{\|r\|}{\|b\|},$$

where $r = b - A\hat{x}$ is the *residual error*, or the extent to which the approximate solution $\hat{x}$ fails to satisfy the equations.

Gaussian elimination with partial pivoting is almost always backward stable in practice. There are some artificial examples where "pivot growth" breaks backward stability, but this never seems to occur in practice; and if it does occur, one can cheaply evaluate the relative residual in order to evaluate the solution. What this means in practice is that solving linear systems with Gaussian elimination with partial pivoting almost always results in a small relative residual (on the order of some modestly growing function in n times ϵ_{mach}, for example). However, a small relative residual only translates to a small relative error if the condition number is also not too big!

Cholesky factorization

For matrices that are symmetric and positive definite, the *Cholesky factorization*

$$A = LL^T$$

is an attractive alternative to Gaussian elimination. Here, the *Cholesky factor* L is a lower triangular matrix; by convention, the diagonal of L is chosen to be positive. Sometimes, the Cholesky factorization is written in the equivalent form

$$A = R^T R$$

where R is upper triangular; this is the convention used by default in MATLAB. One way to see this factorization is as a generalization of the posive square root of a positive real number[1]

The Cholesky factorization is useful for solving linear systems, among other things. Cholesky factors also show up in statistical applications, such as sampling a multivariate normal with given covariance; and the existence of a (nonsingular) Cholesky factor is equivalent to A being positive definite, so Cholesky factorization is sometimes also used to check for positive definiteness. Even if we're only interested in linear systems, the Cholesky factorization has a very attractive feature compared to Gaussian elimination: it can be stably computed without pivoting. Because pivoting is sort of a pain, I'm going to leave the discussion of the algorithmic details of Gaussian elimination to the book, but I will walk through some ideas behind Cholesky factorization.

Let's start with the Cholesky factorization of a 2-by-2 matrix:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{12} & a_{22} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 \\ l_{21} & l_{22} \end{bmatrix} \begin{bmatrix} l_{11} & l_{21} \\ 0 & l_{22} \end{bmatrix}.$$

We can write this matrix equation as three scalar equations, which we can easily use to solve for the Cholesky factor

$$\begin{aligned} a_{11} &= l_{11}^2 & l_{11} &= \sqrt{a_{11}} \\ a_{12} &= l_{21} l_{11} & l_{21} &= a_{12}/l_{11} \\ a_{22} &= l_{22}^2 + l_{21}^2 & l_{22} &= \sqrt{a_{22} - l_{21}^2}. \end{aligned}$$

This picture actually generalizes. Now suppose we write down a *block* matrix formula:

$$\begin{bmatrix} a_{11} & a_{21}^T \\ a_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 \\ l_{21} & L_{22} \end{bmatrix} \begin{bmatrix} l_{11} & l_{21}^T \\ 0 & L_{22} \end{bmatrix}.$$

[1]It's worth noting that the matrix square root of an SPD matrix A is actually a symmetric positive definite matrix B such that $A = B^2$. So while the Cholesky factor is *a* generalization of square roots to matrices, it is not the generalization that gets called "the matrix square root."

Here, we're thinking of a_{11} and l_{11} as scalars, a_{21} and l_{21} as column vectors, and A_{22} and L_{22} as matrices. Working out the multiplication, we again have three equations:

$$\begin{aligned} a_{11} &= l_{11}^2 & l_{11} &= \sqrt{a_{11}} \\ a_{21} &= l_{21} l_{11} & l_{21} &= a_{21} l_{11}^{-1} \\ A_{22} &= L_{22} L_{22}^T + l_{21} l_{21}^T & L_{22} L_{22}^T &= A_{22} - l_{21} l_{21}^T. \end{aligned}$$

We can compute the first column of the Cholesky factor by the first two of these equations, and the remaining equation tells us how to express the rest of L as the Cholesky factor for a smaller matrix. Here's the idea in MATLAB:

```
function L = lec08chol(A)

  n = length(A);
  L = zeros(n);

  for j = 1:n

    % Compute column j of L
    L(j,j) = sqrt(A(j,j));
    L(j+1:n,j) = A(j+1:n,j)/L(j,j);

    % Update the trailing submatrix (a "Schur complement")
    A(j+1:n,j+1:n) = A(j+1:n,j+1:n)-L(j+1:n,j)*L(j+1:n,j)';

  end
```

Actually, MATLAB uses an even more sophisticated algorithm based on a block factorization

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{12} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} L_{11} & L_{21} \\ 0 & L_{22} \end{bmatrix}.$$

The L_{11} part of the factor is the Cholesky factorization of A_{11}, which is computed by a small Cholesky factorization routine; the block $L_{21} = A_{21} L_{11}^{-1}$ is computed by triangular solves; and then L_{22} is computed by a block factorization of the Schur complement $A_{22} - L_{21} L_{21}^T$. This organization turns out to be very useful for writing *cache-efficient code* that is able to do a lot of work on a small part of the matrix before moving on to other parts of the computation.

Problems to ponder

1. Suppose you were given P, L, and U such that $PA = LU$. How would you solve $A^T x = b$?

2. Suppose $A = LU$. How could you compute $\det(A)$ efficiently using the L and U factors[2]?

3. Show that the product of two unit lower triangular matrices is again unit lower triangular.

4. I claimed that if A has a nonsingular Cholesky factor, then A is SPD. Why is that so?

5. Suppose
$$A = \begin{bmatrix} \epsilon & 1 \\ 1 & 1 \end{bmatrix}$$
What is the one-norm condition number $\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1$?

6. I claimed in class that
$$\frac{\|\hat{x} - x\|}{\|x\|} \leq \kappa(A) \frac{\|r\|}{\|b\|}.$$
Using the formula $r = A\hat{x} - b = A(\hat{x} - x)$ and properties of norms, argue why this must be true.

7. What is the Cholesky factor of A?
$$A = \begin{bmatrix} 4 & 4 & 2 \\ 4 & 20 & 34 \\ 2 & 34 & 74 \end{bmatrix}$$
What is the determinant?

[2]When I taught multivariable calculus, I actually started off with this method for computing determinants. It has a nice interpretation if you think about an elementary operation in Gaussian elimination as a *shear transformation*, which preserves volumes.

8. Write an $O(n)$ code to compute the Cholesky factor of an SPD tridiagonal matrix given the diagonal entries $a_1, \ldots, a_n$ and off-diagonal entries $b_1, b_2, \ldots, b_{n-1}$:
$$A = \begin{bmatrix} a_1 & b_1 & & & \\ b_1 & a_2 & b_2 & & \\ & b_2 & a_3 & b_3 & \\ & & \ddots & \ddots & \ddots \\ & & & b_{n-1} & a_n \end{bmatrix}$$

9. *Harder.* In order to test whether or not a matrix A is singular, one sometimes uses a *bordered linear system.* If $A \in \mathbb{R}^{n\times n}$, we choose $b, c \in \mathbb{R}^n$ and $d \in \mathbb{R}$ at random and try to solve the equation
$$\begin{bmatrix} A & b \\ c^T & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$
If the extended matrix is singular, then A almost certainly has a null space of at least dimension two; otherwise, with high probability, $y = 0$ iff A is singular. Why does this make sense?

Week 5: Wednesday, Feb 22

Least squares: the big idea

Least squares problems are a special sort of minimization. Suppose $A \in \mathbb{R}^{m \times n}$ and $m > n$. In general, we will not be able to exactly solve *overdetermined* equations $Ax = b$; the best we can do is to minimize the *residual* $r = b - Ax$. In least squares problems, we minimize the two-norm of the residual[1]:

$$\text{Find } \hat{x} \text{ to minimize } \|r\|_2^2 = \langle r, r \rangle.$$

This is not the only way to approximate the solution to an overdetermined system, but it is attractive for several reasons:

1. It's *really* mathematically attractive. $\|x\|^2$ is a smooth function of x, and the solution to the least squares problem is a linear function of b ($x = A^\dagger b$ where $A^\dagger$ is the Moore-Penrose pseudoinverse of A)

2. There's a nice picture that goes with it – the least squares solution is the projection of b onto the span of A, and the residual at the least squares solution is orthogonal to the span of A.

3. It's a mathematically reasonable choice in statistical settings when the data vector b is contaminated by Gaussian noise.

Normal equations

One way to solve the least squares problem is to attack it directly. We know $\|r\|^2 = \|b - Ax\|^2$; and from a given x, the directional derivative in any direction δx is

$$\nabla_x \|r\|^2 \cdot \delta x = 2\langle A\delta x, b - Ax \rangle = 2\delta x^T (A^T b - A^T A x).$$

The minimum occurs when all posible directional derivatives are zero, which gives us the *normal equations*[2]

$$A^T A x = A^T b.$$

[1] Minimizing the two-norm is equivalent to miminizing the squared two-norm.

[2] They are called the *normal* equations because they specify that the residual must be normal (orthogonal) to every vector in the span of A.

Rearranging, we have

$$x = (A^T A)^{-1} A^T b = A^\dagger b;$$

the matrix $A^\dagger = (A^T A)^{-1} A^T$ is the *Moore-Penrose pseudoinverse* of A (sometimes just called the pseudoinverse).

If the columns of A are not too close to linearly dependent, we would usually just form the normal equations and solve them by using Cholesky factorization to write

$$A^T A = R^T R,$$

where R is an upper triangular matrix.

QR factorization

Another approach is to write a *QR factorization*:

$$A = QR = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \begin{bmatrix} R_1 \\ 0 \end{bmatrix} = Q_1 R_1$$

where $Q \in \mathbb{R}^{m \times m}$ is orthogonal ($Q^T Q = I$) and R is upper triangular. The columns of $Q_1 \in \mathbb{R}^{m \times n}$ form an orthonormal basis for the range space of A, and the columns of Q_2 span the orthogonal complement. The factorization $A = Q_1 R_1$ is sometimes called the "economy" QR factorization.

Multiplication by an orthogonal matrix does not change lengths, so

$$\|r\|^2 = \|Q^T r\|^2 = \left\| \begin{bmatrix} R_1 \\ 0 \end{bmatrix} x - Q^T b \right\|^2 = \|R_1 x - Q_1^T b\|^2 + \|Q_2^T b\|^2.$$

The second part of this expression ($\|Q_2^T b\|^2$) is error that we cannot reduce; but $R_1 x - Q_1^T b$ can be made exactly equal to zero. That is, the solution to the least squares problem is

$$x = R_1^{-1} Q_1^T b.$$

In Matlab, we can compute the QR factorization using the `qr` routine:

```
[Q, R ] = qr(A);    % Full QR
[Q1,R1] = qr(A,0); % Economy QR
```

We also use QR implicitly if we solve a least-squares system using the ever-useful backslash operator:

```
x = A\b;   % Minimize norm(Ax−b) via a QR factorization
```

Sensitivity and conditioning

At a high level, there are two pieces to solving a least squares problem:

1. Project b onto the span of A.
2. Solve a linear system so that Ax equals the projected b.

Correspondingly, there are two ways we can get into trouble in solving least squares problem: either b may be nearly orthogonal to the span of A, or the linear system might be ill-conditioned.

Let's consider the issue of b nearly orthogonal to A first. Suppose we have the trivial problem

$$A = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad b = \begin{bmatrix} \epsilon \\ 1 \end{bmatrix}.$$

The solution to this problem is $x = \epsilon$; but the solution for

$$A = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad \hat{b} = \begin{bmatrix} -\epsilon \\ 1 \end{bmatrix}.$$

is $\hat{x} = -\epsilon$. Note that $\|\hat{b} - b\|/\|b\| \approx 2\epsilon$ is small, but $|\hat{x} - x|/|x| = 2$ is huge. That is because the projection of b onto the span of A (i.e. the first component of b) is much smaller than b itself; so an error in b that is small relative to the overall size may not be small relative to the size of the projection onto the columns of A.

Of course, the case when b is nearly orthogonal to A often corresponds to a rather silly regression, like trying to fit a straight line to data distributed uniformly around a circle, or trying to find a meaningful signal when the signal to noise ratio is tiny. This is something to be aware of and to watch out for, but it isn't exactly subtle: if $\|r\|/\|b\|$ is close to one, we have a numerical problem, but we also probably don't have a very good model. A more subtle issue problem occurs when some columns of A are nearly linearly dependent (i.e. A is ill-conditioned).

The *condition number of A for least squares* is

$$\kappa(A) = \|A\|\|A^\dagger\| = \kappa(R_1) = \sqrt{\kappa(A^T A)}.$$

We generally recommend solving least squares via QR factorization because $\kappa(R_1) = \kappa(A)$, while forming the normal equations squares the condition number. If $\kappa(A)$ is large, that means:

1. Small relative changes to A can cause large changes to the span of A (i.e. there are some vectors in the span of $\hat{A}$ that form a large angle with all the vectors in the span of A).

2. The linear system to find x in terms of the projection onto A will be ill-conditioned.

If θ is the angle between b and the range of A[3], then the sensitivity to perturbations in b is

$$\frac{\|\Delta x\|}{\|x\|} \leq \frac{\kappa(A)}{\cos(\theta)} \frac{\|\delta b\|}{\|b\|},$$

while the sensitivity to perturbations in A is

$$\frac{\|\Delta x\|}{\|x\|} \leq \left(\kappa(A)^2 \tan(\theta) + \kappa(A)\right) \frac{\|E\|}{\|A\|}.$$

Even if the residual is moderate, the sensitivity of the least squares problem to perturbations in A (either due to roundoff or due to measurement error) can quickly be dominated by $\kappa(A)^2 \tan(\theta)$ if $\kappa(A)$ is at all large.

In regression problems, the columns of A correspond to explanatory factors. For example, we might try to use height, weight, and age to explain the probability of some disease. In this setting, ill-conditioning happens when the explanatory factors are correlated — for example, perhaps weight might be well predicted by height and age in our sample population. This happens reasonably often. When there is some correlation, we get moderate ill conditioning, and might want to use QR factorization. When there is a lot of correlation and the columns of A are truly linearly dependent (or close enough for numerical work), we have a *rank-deficient* problem. We will talk about rank-deficient problems next lecture.

Problems to Ponder

1. If x minimizes $\|b - Ax\|^2$, argue that $r \perp Ax$.

2. Show that if x is minimizes $\|Ax - b\|$, then $\|Ax\|^2 + \|r\|^2 = \|b\|^2$.

3. Suppose that $A \in \mathbb{R}^{m \times n}$, $m > n$, and that A has full column rank. Then $A^T A$ is symmetric and positive definite. Why?

[3]Note that b, Ax, and r are three sides of a right triangle, so $\sin(\theta) = \|r\|/\|b\|$.

4. Suppose $A^T A = LL^T$, where L is a lower triangular Cholesky factor. Show that the columns of AL^{-T} are orthonormal.

5. Show that minimizing $\|Ax-b\|$ is equivalent to solving the linear system
$$\begin{bmatrix} I & A \\ A^T & 0 \end{bmatrix} \begin{bmatrix} r \\ x \end{bmatrix} = \begin{bmatrix} b \\ 0 \end{bmatrix}.$$
Can you think of an advantage of writing the least square problem in this way?

6. Find an orthonormal basis for $\mathcal{P}_2$ with the $L^2([-1,1])$ inner product.

7. How would you find the quadratic $p(x)$ to minimize
$$\int_{-1}^{1} (p(x) - f(x))^2 \, dx?$$

8. Suppose $A = \mathbb{R}^{m \times n}$, $m > n$ is full rank, and that $b \in \mathbb{R}^n$. The linear system $A^T x = b$ is *underdetermined.* How would you find the solution that minimizes $\|x\|$?

9. *Maybe only if you've had some stats:* Suppose the entries of $z \in \mathbb{R}^n$ are independent standard normal random variables. Show that for any orthogonal matrix Q, the entries of $Q^T z$ are again independent standard normal random variables.

Week 5: Monday, Feb 27

Least squares reminder

Last week, we started to discuss least squares solutions to overdetermined linear systems:

$$\text{minimize } \|Ax - b\|_2^2$$

where $A \in \mathbb{R}^{m \times n}$, $x \in \mathbb{R}^n$, $b \in \mathbb{R}^m$ with $m > n$. We described two different possible methods for computing the solutions to this equation:

- Solve the *normal equations*

  $$A^T A x = A^T b,$$

  which we derived by finding the critical point for the function $\phi(x) = \|Ax - b\|^2$.

- Compute the *QR decomposition*

  $$A = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \begin{bmatrix} R_{11} \\ 0 \end{bmatrix} = Q_1 R_{11},$$

  where $Q = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix}$ is an orthogonal matrix and R_{11} is upper triangular. Use the fact that multiplication by orthogonal matrices does not change Euclidean lengths to say

  $$\begin{aligned} \|Ax - b\|^2 &= \|Q^T(Ax - b)\|^2 \\ &= \left\| \begin{bmatrix} R_{11} \\ 0 \end{bmatrix} x - \begin{bmatrix} Q_1^T b \\ Q_2^T b \end{bmatrix} \right\|^2 \\ &= \|R_{11}x - Q_1^T b\|^2 + \|Q_2^T b\|^2. \end{aligned}$$

  The second term in the last expression is independent of b; the first term is nonnegative, and can be set to zero by solving the triangular linear system $R_{11}x = Q_1^T b$

So far, our discussion has mostly depended on the *algebra* of least squares problems. But in order to make sense of the sensitivity analysis of least squares, we should also talk about the *geometry* of these problems.

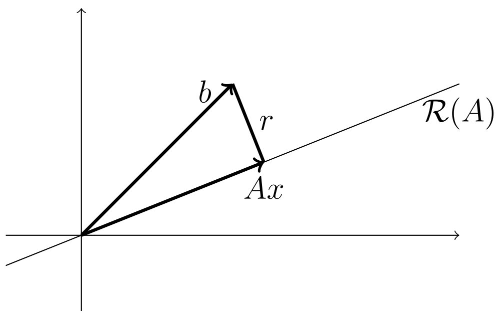


Figure 1: Schematic of the geometry of a least squares problem. The residual vector $r = Ax - b$ is orthogonal to any vector in the range of A.

Least squares: a geometric view

The normal equations are often written as

$$A^T Ax = A^T b,$$

but we could equivalently write

$$\begin{aligned} r &= Ax - b \\ A^T r &= 0. \end{aligned}$$

That is, the normal equations say that at the least squares solution, the residual $r = Ax - b$ is orthogonal to all of the columns of A, and hence to any vector in the range of A.

By the same token, we can use the QR decomposition to write

$$\begin{aligned} r &= Q_2 Q_2^T b, \\ Ax &= Q_1 Q_1^T b. \end{aligned}$$

That is, the QR decomposition lets us write b as a sum of two orthogonal components, Ax and r. Note that the Pythagorean theorem therefore says

$$\|Ax\|^2 + \|r\|^2 = \|b\|^2.$$

Figure 1 illustrates the geometric relations between b, r, A, and x. It's worth spending some time to stare at and comprehend this picture.

Sensitivity and conditioning

At a high level, there are two pieces to solving a least squares problem:

1. Project b onto the span of A.
2. Solve a linear system so that Ax equals the projected b.

Correspondingly, there are two ways we can get into trouble in solving least squares problem: either b may be nearly orthogonal to the span of A, or the linear system might be ill-conditioned.

Let's consider the issue of b nearly orthogonal to A first. Suppose we have the trivial problem

$$A = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad b = \begin{bmatrix} \epsilon \\ 1 \end{bmatrix}.$$

The solution to this problem is $x = \epsilon$; but the solution for

$$A = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad \hat{b} = \begin{bmatrix} -\epsilon \\ 1 \end{bmatrix}.$$

is $\hat{x} = -\epsilon$. Note that $\|\hat{b} - b\|/\|b\| \approx 2\epsilon$ is small, but $|\hat{x} - x|/|x| = 2$ is huge. That is because the projection of b onto the span of A (i.e. the first component of b) is much smaller than b itself; so an error in b that is small relative to the overall size may not be small relative to the size of the projection onto the columns of A.

Of course, the case when b is nearly orthogonal to A often corresponds to a rather silly regression, like trying to fit a straight line to data distributed uniformly around a circle, or trying to find a meaningful signal when the signal to noise ratio is tiny. This is something to be aware of and to watch out for, but it isn't exactly subtle: if $\|r\|/\|b\|$ is close to one, we have a numerical problem, but we also probably don't have a very good model. A more subtle issue problem occurs when some columns of A are nearly linearly dependent (i.e. A is ill-conditioned).

The *condition number of A for least squares* is

$$\kappa(A) = \|A\|\|A^\dagger\| = \kappa(R_1) = \sqrt{\kappa(A^T A)}.$$

We generally recommend solving least squares via QR factorization because $\kappa(R_1) = \kappa(A)$, while forming the normal equations squares the condition number. If $\kappa(A)$ is large, that means:

1. Small relative changes to A can cause large changes to the span of A (i.e. there are some vectors in the span of $\hat{A}$ that form a large angle with all the vectors in the span of A).
2. The linear system to find x in terms of the projection onto A will be ill-conditioned.

If θ is the angle between b and the range of A[1], then the sensitivity to perturbations in b is

$$\frac{\|\Delta x\|}{\|x\|} \leq \frac{\kappa(A)}{\cos(\theta)} \frac{\|\delta b\|}{\|b\|},$$

while the sensitivity to perturbations in A is

$$\frac{\|\Delta x\|}{\|x\|} \leq \left(\kappa(A)^2 \tan(\theta) + \kappa(A)\right) \frac{\|E\|}{\|A\|}.$$

Even if the residual is moderate, the sensitivity of the least squares problem to perturbations in A (either due to roundoff or due to measurement error) can quickly be dominated by $\kappa(A)^2 \tan(\theta)$ if $\kappa(A)$ is at all large.

Ill-conditioned problems

In regression problems, the columns of A correspond to explanatory factors. For example, we might try to use height, weight, and age to explain the probability of some disease. In this setting, ill-conditioning happens when the explanatory factors are correlated — for example, perhaps weight might be well predicted by height and age in our sample population. This happens reasonably often. When there is some correlation, we get moderate ill conditioning, and might want to use QR factorization. When there is a lot of correlation and the columns of A are truly linearly dependent (or close enough for numerical work), or when there A is contaminated by enough noise that a moderate correlation seems dangerous, then we may declare that we have a *rank-deficient* problem.

What should we do when the columns of A are close to linearly dependent (relative to the size of roundoff or of measurement noise)? The answer depends somewhat on our objective for the fit, and whether we care about x on its own merits (because the columns of A are meaningful) or we just about Ax:

[1]Note that b, Ax, and r are three sides of a right triangle, so $\sin(\theta) = \|r\|/\|b\|$.

1. We may want to balance the quality of the fit with the size of the solution or some similar penalty term that helps keep things unique. This is the *regularization* approach.

2. We may want to choose a strongly linearly independent set of columns of A and leave the remaining columns out of our fitting. That is, we want to fit to a subset of the available factors. This can be done using the leading columns of a pivoted version of the QR factorization $AP = QR$. This is sometimes called *parameter subset selection*. MATLAB's backslash operator does this when A is numerically singular.

3. We may want to choose the "most important" directions in the span of A, and use them for our fitting. This is the idea behind *principal components analysis*.

We will focus on the "most important directions" version of this idea, since that will lead us into our next topic: the singular value decomposition. Still, it is important to realize that in some cases, it is more appropriate to add a regularization term or to reduce the number of fitting parameters.

Singular value decomposition

The *singular value decomposition* (SVD) is important for solving least squares problems and for a variety of other approximation tasks in linear algebra. For $A \in \mathbb{R}^{m \times n}$[2], we write

$$A = U\Sigma V^T$$

where $U \in \mathbb{R}^{m \times m}$ and $V \in \mathbb{R}^{n \times n}$ are orthogonal matrices and $\Sigma \in \mathbb{R}^{m \times n}$ is diagonal. The diagonal matrix Σ has non-negative diagonal entries

$$\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_n \geq 0.$$

The σ_i are called the *singular values* of A. We sometimes also write

$$A = \begin{bmatrix} U_1 & U_2 \end{bmatrix} \begin{bmatrix} \Sigma_1 \\ 0 \end{bmatrix} \begin{bmatrix} V_1 & V_2 \end{bmatrix}^T = U_1 \Sigma_1 V_1^T$$

where $U_1 \in \mathbb{R}^{m \times n}$, $\Sigma_1 \in \mathbb{R}^{n \times n}$, $V_1 \in \mathbb{R}^{n \times n}$. We call this the *economy* SVD.

[2]We will assume for the moment that $m \geq n$. Everything about the SVD still makes sense when $m < n$, though.

We can interpret the SVD geometrically using the same picture we drew when talking about the operator two norm. The matrix A maps the unit ball to an ellipse. The axes of the ellipse are $\sigma_1 u_1$, $\sigma_2 u_2$, etc, where the σ_i give the lengths and the u_i give the directions (remember that the u_i are normalized). The columns of V are the vectors in the original space that map onto these axes; that is, $Av_i = \sigma_i u_i$.

We can use the geometry to define the SVD as follows. First, we look for the major axis of the ellipse formed by applying A to the unit ball:

$$\sigma_1^2 = \max_{\|v\|=1} \|Av\|^2 = \max_{\|v\|=1} v^T (A^T A) v.$$

Some of you may recognize this as an eigenvalue problem in disguise: σ_1^2 is the largest eigenvalue of $A^T A$, and v_1 is the corresponding eigenvector. We can then compute u_1 by the relation $\sigma_1 u_1 = Av_1$. To get σ_2, we restrict our attention to the spaces orthogonal to what we have already seen:

$$\sigma_2^2 = \max_{\|v\|=1, v \perp v_1, Av \perp u_1} \|Av\|^2.$$

We can keep going to get the other singular values and vectors.

Norms, conditioning, and near singularity

Given an economy SVD $A = U\Sigma V^T$, we can give satisfyingly brief descriptions (in the two norm) of many of the concepts we've discussed so far in class. The two-norm of A is given by the largest singular value: $\|A\|_2 = \sigma_1$. The pseudoinverse of A, assuming A is full rank, is

$$(A^T A)^{-1} A = U \Sigma^{-1} V^T,$$

which means that $\|A^\dagger\|_2 = 1/\sigma_n$. The condition number for least squares (or for solving the linear system when $m = n$) is therefore

$$\kappa(A) = \sigma_1/\sigma_n.$$

Another useful fact about the SVD is that it gives us a precise characterization of what it means to be "almost" singular. Suppose $A = U\Sigma V^T$ and E is some perturbation. Using invariance of the matrix two norm under orthogonal transformations (the problem du jour), we have

$$\|A + E\|_2 = \|U(\Sigma + \tilde{E})V^T\| = \|\Sigma + \tilde{E}\|,$$

where $\|\tilde{E}\| = \|U^T E V\| = \|E\|$. For the diagonal case, we can actually characterize the smallest perturbation $\tilde{E}$ that makes $\Sigma + \tilde{E}$ singular. It turns out that this smallest perturbation is $\tilde{E} = -\sigma_n e_n e_n^T$ (i.e. something that zeros out the last singular value of Σ). Therefore, we can characterize the smallest singular value as the *distance to singularity*:

$$\sigma_n = \min\{\|E\|_2 : A + E \text{ is singular}\}.$$

The condition number therefore is a *relative distance to singularity*, which is why I keep saying ill-conditioned problems are "close to singular."

The SVD and rank-deficient least squares

If we substitute $A = U\Sigma V^T$ in the least squares residual norm formula, we can "factor out" U just as we pulled out the Q factor in QR decomposition:

$$\|Ax - b\| = \|U\Sigma V^T x - b\| = \|\Sigma \tilde{x} - \tilde{b}\|, \text{ where } \tilde{x} = V^T x \text{ and } \tilde{b} = U^T b.$$

Note that $\|\tilde{x}\| = \|x\|$ and $\|\tilde{b}\| = \|b\|$.

If A has rank r, then singular values $\sigma_{r+1}, \ldots, \sigma_n$ are all zero. In this case, there are many different solutions that minimize the residual — changing the values of $\tilde{x}_{r+1}$ through $\tilde{x}_n$ does not change the residual at all. One standard way to pick a unique solution is to choose the minimal norm solution to the problem, which corresponds to setting $\tilde{x}_{r+1} = \ldots = \tilde{x}_n = 0$. In this case, the Moore-Penrose pseudoinverse is defined as

$$A^\dagger = V_+ \Sigma_+^{-1} U_+^T$$

where $\Sigma_+ = \operatorname{diag}(\sigma_1, \sigma_2, \ldots, \sigma_r)$ and U_+ and V_+ consist of the first r left and right singular vectors.

If A has entries that are not zero but small, it often makes sense to use a *truncated SVD*. That is, instead of setting $\tilde{x}_i = 0$ just when $\sigma_i = 0$, we set $\tilde{x}_i = 0$ whenever σ is small enough. This corresponds, if you like, to perturbing A a little bit before solving in order to get an approximate least squares solution that does not have a terribly large norm.

Why, by the way, might we want to avoid large components? A few reasons come to mind. One issue is that we might be solving linear least squares problems as a step in the solution of some nonlinear problem, and a large solution corresponds to a large step — which means that the local, linear model might not be such a good idea. As another example, suppose we are looking at a model of patient reactions to three drugs, A and B. Drug A has a small effect and a horrible side effect. Drug B just cancels out the horrible side effect. Drug C has a more moderate effect on the problem of interest, and a different, small side effect. A poorly-considered regression might suggest that the best strategy would be to prescribe A and B together in giant doses, but common sense suggests that we should concentrate on drug C. Of course, neither of these examples requires that we use a truncated SVD — it might be fine to use another regularization strategy, or use subset selection.

Week 5: Wednesday, Feb 29

Of cabbages and kings

The past three weeks have covered quite a bit of ground. We've looked at linear systems and least squares problems, and we've discussed Gaussian elimination, QR decompositions, and singular value decompositions. Rather than doing an overly hurried introduction to iterative methods for solving linear systems, I'd like to go back and show the surprisingly versatile role that the SVD can play in thinking about all of these problems.

Geometry of the SVD

How should we understand the singular value decomposition? We've already described the basic algebraic picture:

$$A = U\Sigma V^T,$$

where U and V are orthonormal matrices and Σ is diagonal. But what about the geometric picture?

Let's start by going back to something we glossed over earlier in the semester: the characterization of the matrix 2-norm. By definition, we have

$$\|A\|_2 = \max_{x\neq 0} \frac{\|Ax\|_2}{\|x\|_2}$$

This is equivalent to

$$\|A\|_2^2 = \max_{x\neq 0} \frac{\|Ax\|_2^2}{\|x\|_2^2} = \max_{x\neq 0} \frac{x^T A^T A x}{x^T x}.$$

The quotient $\phi(x) = (x^T A^T A x)/(x^T x)$ is differentiable, and the critical points satisfy

$$0 = \nabla\phi(x) = \frac{2}{x^T x}\left(A^T A x - \phi(x) x\right)$$

That is, the critical points of ϕ – including the value of x that maximizes ϕ – are eigenvectors of A. The corresponding eigenvalues are values of $\phi(x)$. Hence, the largest eigenvalue of $A^T A$ is $\sigma_1^2 = \|A\|_2^2$. The corresponding

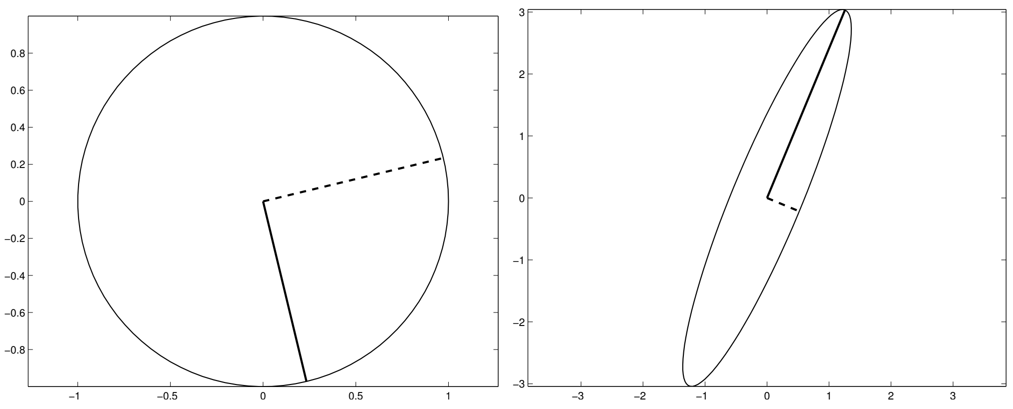


Figure 1: Graphical depiction of an SVD of $A \in \mathbb{R}^{2\times 2}$. The matrix A maps the unit circle (left) to an oval (right); the vectors v_1 (solid, left) and v_2 (dashed, left) are mapped to the major axis $\sigma_1 u_1$ (solid, right) and the minor axis $\sigma_2 u_2$ (dashed, right) for the oval.

eigenvector v_1 is the right singular vector corresponding to the eigenvalue σ_1^2; and $Av_1 = \sigma_1 u_1$ gives the first singular value.

What does this really say? It says that v_1 is the vector that is stretched the most by multiplication by A, and σ_1 is the amount of stretching. More generally, we can *completely* characterize A by an orthonormal basis of right singular vectors that are each transformed in the same special way: they get scaled, then rotated or reflected in a way that preserves lengths. Viewed differently, the matrix A maps vectors on the unit sphere into an ovoid shape, and the singular values are the lengths of the axes. In Figure 1, we show this for a particular example, the matrix

$$A = \begin{bmatrix} 0.8 & -1.1 \\ 0.5 & -3.0 \end{bmatrix}.$$

Conditioning and the distance to singularity

We have already seen that the condition number for linear equation solving is

$$\kappa(A) = \|A\|\|A^{-1}\|$$

When the norm in question is the operator two norm, we have that $\|A\| = \sigma_1$ and $\|A^{-1}\| = \sigma_n^{-1}$, so

$$\kappa(A) = \frac{\sigma_1}{\sigma_n}$$

That is, $\kappa(A)$ is the ratio between the largest and the smallest amounts by which a vector can be stretched through multiplication by A.

There is another way to interpret this, too. If $A = U\Sigma V^T$ is a square matrix, then the smallest E (in the two-norm) such that $A - E$ is *exactly* singular is $A - \sigma_n u_n v_n^T$. Thus,

$$\kappa(A)^{-1} = \frac{\|E\|}{\|A\|}$$

is the *relative distance to singularity* for the matrix A. So a matrix is ill-conditioned exactly when a relatively small perturbation would make it exactly singular.

For least squares problems, we still write

$$\kappa(A) = \frac{\sigma_1}{\sigma_n},$$

and we can still interpret $\kappa(A)$ as the ratio of the largest to the smallest amount that multiplication by A can stretch a vector. We can also still interpret $\kappa(A)$ in terms of the distance to singularity – or, at least, the distance to rank deficiency. Of course, the actual sensitivity of least squares problems to perturbation depends on the angle between the right hand side vector b and the range of A, but the basic intuition that big condition numbers means problems can be very near singular – very nearly ill-posed – tells us the types of situations that can lead us into trouble.

Orthogonal Procrustes

The SVD can provide surprising insights in settings other than standard least squares and linear systems problems. Let's consider one interesting one that comes up when doing things like trying to align 3D models with each other.

Suppose we are given two sets of coordinates for m points in n-dimensional space, arranged into rows of $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{m \times n}$. Let's also suppose the matrices are (approximately) related by a rigid motion that leaves the origin fixed. How can we recover the transformation? That is, we want an orthogonal matrix W that minimizes $\|AW - B\|_F^2$. This is sometimes called an *orthogonal Procrustes problem*, named in honor of the legendary Greek king Procrustes, who had a bed on which he would either stretch guests or cut off their legs in order to make them fit perfectly.

We can write $\|AW - B\|_F^2$ as

$$\|AW - B\|_F^2 = (\|A\|_F^2 + \|B\|_F^2) - \operatorname{tr}(W^T A^T B),$$

so minimizing the squared residual is equivalent to maximizing $\operatorname{tr}(W^T A^T B)$. Note that if $A^T B = U \Sigma V^T$, then

$$\operatorname{tr}(W^T A^T B) = \operatorname{tr}(W^T U \Sigma V^T) = \operatorname{tr}(V W U^T \Sigma) = \operatorname{tr}(Z\Sigma),$$

where $Z = VWU^T$ is orthogonal. Now, note that

$$\operatorname{tr}(Z\Sigma) = \operatorname{tr}(\Sigma Z) = \sum_i \sigma_i z_{ii}$$

is maximal over all orthogonal matrices when $z_{ii} = 1$ for each i. Therefore, the trace is maximized when $Z = I$, corresponding to $W = UV^T$.

Problems to Ponder

1. Suppose $A \in \mathbb{R}^{n \times n}$ is invertible and $A = U \Sigma V^T$ is given. How could we use this decomposition to solve $Ax = b$ in $O(n^2)$ additional work?
2. What are the singular values of A^{-1} in terms of the singular values of A?
3. Suppose $A = QR$. Show $\kappa_2(A) = \kappa_2(R)$.
4. Suppose that $A^T A = R^T R$, where R is an upper triangular Cholesky factor. Show that AR^{-1} is a matrix with orthonormal columns.
5. Show that if V and W are orthogonal matrices with appropriate dimensions, then $\|VAW\|_F = \|A\|_F$.
6. Show that if $X, Y \in \mathbb{R}^{m \times n}$ and $\operatorname{tr}(X^T Y) = 0$ then $\|X\|_F^2 + \|Y\|_F^2 = \|X + Y\|_F^2$.
7. Why do the diagonal entries of an orthogonal matrix have to lie between -1 and 1? Why must an orthogonal matrix with all ones on the diagonal be an identity matrix?

Week 6: Monday, Mar 5

Iterative and Direct Methods

So far, we have discussed *direct* methods for solving linear systems and least squares problems. These methods have several advantages:

- They are general purpose. It helps to recognize some basic structural properties (sparsity, symmetry, etc), and you need to understand conditioning. Otherwise, you can often trust that MATLAB's backslash operation is doing something reasonable.
- They are robust. More specifically, direct methods are generally backward stable.
- There are good, fast standard libraries.

The main challenges of direct methods involve scaling. Forming and factoring a large matrix can be expensive.

Iterative methods for solving linear systems have a lot of knobs to twiddle, and they often have to be tailored for specific types of systems in order to converge well. But when they are tailored, and when the parameters are set right, they can be very efficient.

A Model Problem

There is a standard model problem for introducing iterative methods for linear systems: a discretized Poisson equation. In lecture, I talked about the two-dimensional case (which is the same case that is in the book); but in order to present the ideas in a simple way, let me write in these notes about the 1D case.

In order to set up this model problem, we need the following approximation: if $u(x)$ is twice differentiable, then

$$u''(x) = \frac{u(x-h) - 2u(x) + u(x+h)}{h^2} + O(h^2).$$

We can use this *finite difference approximation* to solve differential equations. For example, suppose we want to approximate the solution to

$$\begin{aligned} -u''(x) &= f(x) \text{ for } 0 \leq x \leq 1 \\ u(0) &= u(1) = 0. \end{aligned}$$

The standard approach would be to sample the interval $[0, 1]$ with a mesh of points ih for $i = 0, 1, 2, \ldots, N+1$ (so $h = 1/(N+1)$), and let $u_i \approx u(ih)$ and $g_i = h^2 f(ih)$. Then

$$\begin{aligned} -u_{i-1} + 2u_i - u_{i+1} &= -g_i \text{ for } i = 1, 2, \ldots, N \\ u_0 = u_{N+1} &= 0. \end{aligned}$$

Listing the equations in order, we have

$$Tu = -g,$$

where T is a tridiagonal matrix with 2 on the main diagonal and -1 on the first sub and superdiagonals. For example, for $N = 5$, we have $T \in \mathbb{R}^{5\times5}$ given by

$$T = \begin{bmatrix} 2 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{bmatrix}.$$

Relax!

Suppose we wanted to solve a system like $Tu = -g$ using an iterative method. That is, we are willing to put aside the machinery we've built for directly solving the system through a factorization, and instead we will construct a sequence of guesses $u^{(k)}$ that will converge to the true solution as $k \rightarrow \infty$. How should we do this?

The key point here is that we don't necessarily care that $u^{(k+1)}$ should be the true answer – it should just be more right than $u^{(k)}$. So it is natural to try to *relax* the problem so that we can "fix up" the solution by a little bit at each step. For example, if we believe that $u^{(k)}$ is a good guess, then we might try to fix up $u^{(k+1)}$ by making sure that the variable at each point in the new steps satisfies the balance equation at that same point (assuming that the neighbor data comes from the old step). That is, for each i we would compute a new approximate solution value using

$$-u_{i-1}^{(k)} + 2u_i^{(k+1)} - u_{i+1}^{(k)} = -g_i.$$

This is *Jacobi* iteration.

Suppose we programmed Jacobi iteration sweeping from $i = 1$ up to $i = N$[1]:

```
% Perform a single Jacobi iteration, computing unew from u
unew(1) = ( u(2)−g(1) )/2;
for i = 2:N−1
  unew(i) = ( u(i−1)+u(i+1)−g(i) )/2;
end
unew(N) = ( u(N−1)−g(1) )/2;
```

Notice that at the time we have computed $u_i^{(k+1)}$ in this code, we have also computed $u_{i-1}^{(k+1)}$. Wouldn't it be better to update $u_i^{(k+1)}$ using this new value, instead of the old one? This natural idea is sometimes called *Gauss-Seidel* iteration:

$$-u_{i-1}^{(k+1)} + 2u_i^{(k+1)} - u_{i+1}^{(k)} = -g_i.$$

When we program a Gauss-Seidel iteration, we can get away with just a single vector for the approximate solution that is overwritten during each sweep:

```
% Perform a Gauss−Seidel sweep, overwriting u with updated guesses
u(1) = ( u(2)−g(1) )/2;
for i = 2:N−1
  u(i) = ( u(i−1)+u(i+1)−g(i) )/2;
end
unew(N) = ( u(N−1)−g(1) )/2;
```

Unfortunately, as we have presented them so far, it seems like it would be a mess to analyze the convergence of either Jacobi or Gauss-Seidel. In order to stay sane during such convergence analysis, we would like a clean notation, and it is to this topic we now turn.

[1] I have assumed the MATLAB vector `u` only holds the active variables $u_1, \ldots, u_N$. If I kept a little extra space for the boundary values $u_0 = u_{N+1} = 0$, I could get rid of the special-case updates for u_1 and u_N.

The Matrix Splitting Perspective

Consider the following general approach to constructing fixed-point iterations to solve $Ax = b$:

1. Split A into two pieces: $A = M - N$. The matrix M should ideally "look like" A, but it should be easy to solve linear systems involving M (where it might not be so easy with A).

2. Iterate on
$$Mx^{(k+1)} = Nx^{(k)} + b,$$
or, equivalently,
$$x^{(k+1)} = x^{(k)} - M^{-1}(Ax^{(k)} - b). \tag{1}$$

The fixed point for the iteration (1) is clearly $x_* = A^{-1}b$. Furthermore, both Jacobi and Gauss-Seidel iteration can be written in terms of a matrix splitting: for Jacobi, we take M to be the diagonal part of A, and for Gauss-Seidel, we take M to be the lower triangular part.

Remember now that we have a general strategy for analyzing the convergence of fixed point iterations, which is to subtract the fixed point equation from the iteration equation in order to get an equation for error propogation. In this case,

$$e^{(k+1)} = e^{(k)} - M^{-1}Ae^{(k)} = (I - M^{-1}A)e^{(k)}.$$

Now, notice that for any consistent choice of norms,

$$\|e^{(k+1)}\| = \|(I - M^{-1}A)e^{(k)}\| \leq \|(I - M^{-1}A)\|\|e^{(k)}\|,$$

so that if $\|I - M^{-1}A\| < 1$, the iteration converges. The converse is not quite true, and in order to make a precise statement about convergence we need to reason about the *spectral radius* of $I - M^{-1}A$. But this norm-based bound is good enough for our present purposes, and we will leave the spectral analysis to another time.

Week 6: Wednesday, Mar 7

From Stationary Methods to Krylov Subspaces

Last time, we discussed *stationary methods* for the iterative solution of linear systems of equations, which can generally be written in the form

$$x^{(k+1)} = x^{(k)} - M^{-1}(Ax^{(k)} - b).$$

Stationary methods are simple, and they make good building blocks for more sophisticated methods, but for most purposes they have been superceded by faster-converging *Krylov subspace* methods. The book describes one of these, the conjugate gradient method (CG), in a little detail. In these notes, we will describe the same iteration, but from a slightly different perspective.

When all you have is a hammer...

Suppose that we want to solve $Ax = b$, but the only information we have about A is a subroutine that can apply A to a vector. What should we do? If the only operation at hand is matrix mutiplication, and the only vector staring us in the face is b, a natural approach would be to take b and start multiplying by A to get b, Ab, A^2b, etc. Taking linear combinations of these vectors gives us a *Krylov subspace*:

$$\mathcal{K}_k(A, b) = \operatorname{span}\{b, Ab, A^2b, \ldots, A^{k-1}b\}.$$

Given the examples that we've seen so far, you might reasonably wonder why we're assuming that we can do so little with A. In fact, there are many cases where working with the entries of A is a pain, but evaluating a matrix-vector product can be done efficiently. One set of examples comes from image and signal processing, where many linear operations can be applied efficiently using Fourier transforms. Another example, which we may encounter again after the break when we talk about Newton methods for solving systems of linear equations, is approximate multiplication by a Jacobian matrix using finite differences. Since this example reinforces some calculus concepts that have occurred repeatedly, let's go through it in a little detail.

Suppose $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$ has two continuous derivatives. At a point x, the *Jacobian matrix* $J(x)$ is a matrix of partial derivatives of f:

$$J_{ij} = \frac{\partial f_i}{\partial x_j}(x).$$

The Jacobian matrix can be used to calculate *directional derivatives* using the chain rule; if u is some direction, then

$$\frac{\partial f}{\partial u}(x) = \left.\frac{d}{dt}\right|_{t=0} f(x+tu) = J(x)u.$$

In a calculus course, you might learn that multiplying the Jacobian matrix by a direction vector is a way to compute a directional derivative. Here, we take the opposite approach: to approximate the product of a Jacobian matrix and a vector, we approximate a directional derivative:

$$J(x)u = \left.\frac{d}{dt}\right|_{t=0} f(x+tu) \approx \frac{f(x+hu)-f(x)}{h}.$$

If f is a "black box" function that is hard to differentiate analytically, this numerical approach to computing matrix-vector products with the Jacobian is sometimes incredibly useful.

From Linear Systems to Optimization

There are two basic ingredients to Krylov subspace methods:

1. Build a sequence of Krylov subspaces.
2. Find approximate solutions in those subspaces.

The conjugate gradient method (CG) pulls out approximate solutions to $Ax = b$ when A is a symmetric positive definite matrix by turning the problem of linear system solving into an equivalent optimization problem. That is, the functional

$$\phi(x) = \frac{1}{2}x^T Ax - x^T b,$$

has a single critical point at

$$0 = \nabla\phi(x) = Ax - b.$$

By the second derivative test, this critical point is a global minimum for ϕ. The conjugate gradient method finds an approximate solution $x^{(k)} \in \mathcal{K}_k(A, b)$ by minimizing $\phi(x)$ over the Krylov subspace. In exact arithmetic, this is guaranteed to converge to the true solution in at most n steps, but in practice we usually get very good approximations in far fewer than n steps.

The *what* of the CG algorithm, then, is straightforward: at step k, the method produces an approximate solution $x^{(k)}$ that minimizes $\phi(x)$ over the kth Krylov subspace. The *how* of the algorithm involves a lot of beautiful mathematical connections and a little bit of magic. I will not discuss the implementation of CG further, except to point to the very popular article by Jonathan Shewchuck on "The Conjugate Gradient Method without the Agonizing Pain," which provides a gentle introduction to the particulars[1].

Preconditioning

For the fastest convergence, Krylov subspace methods like conjugate gradients should be used with a *preconditioner*. That is, instead of solving $Ax = b$, we solve

$$M^{-1}Ax = M^{-1}b,$$

where applying M^{-1} should approximate A^{-1} in some very loose sense[2] be reasonably inexpensive. Thinking back to the last lecture, we recall that $M^{-1}A$ also showed up when we were analyzing the stationary iteration

$$y^{(k+1)} = y^{(k)} - M^{-1}(Ay^{(k)} - b)$$

Note that if we start this iteration at $y^{(0)} = 0$, then

$$y^{(k)} \in \mathcal{K}_k(M^{-1}A, M^{-1}b).$$

[1]I personally prefer the terser — more painful? — treatment in standard numerical linear algebra texts like those of Demmel or Golub and Van Loan, or even just the "template" given in the book *Templates for the Solution of Linear Systems* (which is freely available online). Then again, I grew up to be a numerical analyst. Your mileage may vary.

[2]Really, M^{-1} should be chosen so that the eigenvalues of $M^{-1}A$ are clustered, though this characterization tends to be more useful for analysis than for preconditioner design. I tried to walk through this in lecture, and realized about 3/4 of the way through my explanation that I'd lost everybody by pulling in too much about eigenvalues too quickly. Sorry about that. Don't worry, it won't be on any exams.

That is, the first k steps of a stationary iteration with the splitting matrix M form a basis for a *preconditioned* Krylov subspace $\mathcal{K}_k(M^{-1}A, M^{-1}b)$. Since Krylov subspace methods try to find the best possible solution within a subspace (where the definition of "best" varies from method to method), using M as a preconditioner for a Krylov subspace method typically yields better approximations than using M as the basis for a stationary method. However, the M matrices used in classical stationary methods such as the Jacobi, Gauss-Seidel, and SOR iterations are frequently used as default preconditioners. It is often possible to construct much better preconditioners for specific classes of matrices using a combination of hard analysis and physical intuition, but this is beyond the scope of what we will cover in this class.

Problems to ponder

1. A is *strictly diagonally dominant* if for each i,
$$a_{ii} > \sum_{j \neq i} |a_{ij}|.$$
Show that Jacobi iteration necessarily converges for strictly diagonally dominant matrices.

2. If A is a sparse rectangular matrix, MATLAB provides a so-called "Q-less" QR decomposition in which R is a sparse matrix and $Q = AR^{-1}$ is never formed explicitly. Suppose the computed factor $\hat{R}$ is contaminated by a small amount of noise. Describe a stationary method that nonetheless uses this computed factor to compute $\operatorname{argmin}_x \|Ax - b\|^2$ accurately in a few steps[3].

3. Show that if $\phi(x) = \frac{1}{2}x^T Ax - x^T b$ and A is symmetric, then $\nabla\phi(x) = Ax - b$.

4. Given a guess x^{old}, consider the update $x^{\text{new}} = x^{\text{old}} + te_i$. For what value of t is $\phi(x^{\text{new}})$ minimized? For the one-dimensional Poisson model problem, show that updating the ith component of x according to this rule is equivalent to step i in a Gauss-Seidel sweep[4]

5. Suppose $x^* = \operatorname{argmin}_{x \in \mathcal{U}} \phi(x)$. Show that $Ax^* - b$ is orthogonal to every vector in $\mathcal{U}$.

6. Show that $\langle x^* - A^{-1}b, x^* \rangle_A = 0$ (use the previous question).

7. Show that minimizing $\phi(x^*)$ over all vectors in $\mathcal{U}$ also minimizes $\|x^* - A^{-1}b\|_A^2$ over all vectors in $\mathcal{U}$.

[3]If you are following along offline, type `help qr` in MATLAB — the online documentation describes exactly the iterative refinement strategy I have in mind.

[4]This is true for matrices other than the model problem, too; in general, Gauss-Seidel for a symmetric positive definite system will monotonically reduce the value of ϕ.

Week 7: Monday, Mar 12

Newton and Company

Suppose $f : \mathbb{R}^n \to \mathbb{R}^n$ is twice differentiable. Then

$$f(x + \delta x) = f(x) + f'(x)\delta x + O(\|\delta x\|^2),$$

where $f'(x)$ denotes the Jacobian matrix at x. The idea of Newton iteration in this multi-dimensional setting is the same as the idea in one dimension: in order to get x_{k+1}, set the linear approximation to f about x_k to zero. That is, we set

$$f(x_k) + f'(x_k)(x_{k+1} - x_k) = 0,$$

which we can rearrange to

$$x_{k+1} = x_k - f'(x_k)^{-1} f(x_k).$$

The basic form of Newton iteration remains the same in the multi-dimensional case, and so do the basic convergence properties. As usual, we obtain a recurrence for the error by subtracting the fixed point equation from the fixed point equation:

$$\begin{array}{r} f(x_k) + f'(x_k)(x_{k+1} - x_k) = 0 \\ f(x_k) + f'(x_k)(x_* - x_k) = O(\|x_* - x_k\|^2) \\ \hline f'(x_k)e_{k+1} = O(\|e_k\|^2) \end{array}$$

As long as $f'(x_*)$ is nonsingular (i.e. x_* is a *regular* root) and the second derivatives of f are continuous near x_*, this iteration is quadratically convergent to regular roots (where f' is nonsingular) from close enough starting points. But the iteration may well diverge if the starting point is not good enough. What changes in the multi-dimensional case is the cost of a Newton step. If the dimension n is large, the cost of forming and factoring the Jacobian matrix may start to dominate the other costs in the iteration. For this reason, one frequently uses modified Newton iterations in which solutions with $f'(x_k)^{-1}$ are approximated in some way.

There are several ways to approximate $f'(x_k)^{-1}$, and these provide different tradeoffs between the reduction in error per step and the cost per step. The simplest flavor of modified Newton iteration might be

$$x_{k+1} = x_k - \hat{J}^{-1} f(x_k),$$

where $\hat{J}$ is an approximation to the Jacobian (e.g. computed and factored at x_0, and then frozen for successive iterations). When does this converge? Remember our basic analysis method for fixed point iterations: take the iteration equation and subtract the fixed point equation in order to get an iteration for the error. Here, that gives us

$$e_{k+1} = \left(I - \hat{J}^{-1} f'(x_*)\right) e_k + O(\|e_k\|^2),$$

and taking norms gives

$$\|e_{k+1}\| \leq \left\|I - \hat{J}^{-1} f'(x_*)\right\| \|e_k\| + O(\|e_k\|^2),$$

so convergence is assured (for x_0 close to x_*) when $\|I - \hat{J}^{-1} f'(x)\| < 1$.

Dealing with Newton

Newton iteration and closely-related variants are a workhorse in nonlinear equation solving. Unfortunately, as we have seen, Newton's method is only *locally* convergent. A good part of the art of nonlinear equation solving is in dealing with this local convergence property. In one dimension, we can combine Newton's method (or secant method, or other iterations) with bisection in order to get something that is simultaneously robust and efficient; Charlie talked about this last Wednesday. But bisection is a one-dimensional construction. In higher dimensions, what can we do?

As it turns out, there are several possible strategies:

1. *Get a good guess*: If you have some method of getting a good initial guess, Newton iteration is terrific. Getting a good guess is application-specific.

2. *Modify the problem*: There are usually many equivalent ways to write an equation $f(x) = 0$. Some of those ways of writing things may lead to better convergence. For example, if $f(x)$ has a zero close to the origin and approaches some constant value far away from the origin, we might want to look at an equation like $f(x)(\|x\|^2 + 1) = 0$. If $f(x)$ has a pole that causes problems, we might want to multiply through by a function that removes that pole. In general, it pays to have a good understanding of the properties of the functions you are solving, and to

try to minimize the effects of properties that confuse Newton iteration. Alas, this is also application-specific.

3. *Choose a specialized iteration*: Newton iteration is our workhorse, but it isn't the only horse around. Other iterations may have better convergence properties, or they may be cheap enough that you are willing to let them run for many more iterations than you would want to take with Newton. This is application-specific; but for lots of applications, you can find something reasonable in a textbook or paper.

4. *Use a line search*: What goes wrong with Newton iteration? The Newton direction should always take us in a direction that reduces $\|f(x)\|$, but the problem is that we might overshoot. We can fix this problem by taking steps of the form

   $$x_{k+1} = x_k - \alpha_k f'(x_k)^{-1} f(x_k),$$

   where α_k is chosen so that $\|f(x_{k+1})\| < \|f(x_k)\|$. Refinements of this strategy lead to iterations that converge to *some* root from almost everywhere, but even the basic strategy can work rather well. Ideally, $\alpha_k \rightarrow 1$ eventually, so that we can get the quadratic convergence of Newton once we've gotten sufficiently close to a root.

5. *Use a trust region*[1]: The reason that Newton can overshoot the mark is because we keep using a linear approximation to f about x_k far beyond where the linear approximation is accurate. In a *trust region* method, we define a sphere of radius ρ around x_k where we think linear approximation is reasonable. If the Newton step falls inside the sphere, we take it; otherwise, we find a point on the surface of the sphere to minimize $\|f(x_k) + f'(x_k)(x_{k+1} - x_k)\|^2$.

6. *Use a continuation strategy*: Sometimes there is a natural way of gradually transitioning from an easy problem to a hard problem, and we can use this in a solver strategy. Suppose $f(x; s)$ is a family of functions parameterized by s, where solving $f(x; 1) = 0$ is hard and solving $f(x; 0) = 0$ is easy. Then one approach to solving $f(x; 1) = 0$ is:

[1] I will not ask you about trust regions on any homework or exam! But it is a sufficiently widely-used technique that you might want to at least recognize the term.

```
xguess = 0;  % Initial guess for the easy problem
for s = 0:ds:1
  % Solve f(x; s) = 0 using the solution to f(x; s-h) = 0 as
  % an initial guess
  xguess = basic_solver(f, s, xguess);
end
x = xguess;
```

There are many, many variants on this theme.

Week 7: Wednesday, Mar 14

Line search revisited

In the last lecture, we briefly discussed the idea of a *line search* to improve the convergence of Newton iterations. That is, instead of always using the Newton update
$$x^{k+1} = x^k - f'(x^k)^{-1} f(x^k),$$
we allow ourselves to use a scaled version of the step
$$x^{k+1} = x^k - \alpha_k f'(x^k)^{-1} f(x^k),$$
where α_k is chosen to ensure that the iteration actually makes progress. Here, "progress" is typically measured in terms of the residual norm $\|f(x^{k+1})\|$. At the bare minimum, we want to make sure that the residual goes down at each step, but we can prove a bit more with a slightly stricter criterion:
$$\|f(x^{k+1})\| < (1 - \sigma\alpha_k)\|f(x^k)\|$$
where σ is chosen to be some small value (say 10^{-4}). In practice, this looks something like this:

```
% Get Newton step
[f,J] = eval_f(x);
d = J\f;

% Line search
alpha = 1;
for k = 1:maxstep

  % Try step
  xnew = x-alpha*d;
  fnew = eval_f(xnew);

  % Accept if satisfactoy
  if norm(fnew) < (1-sigma*alpha)*norm(f)
    x = xnew;
    f = fnew;
```

```
      break;
    end

    % Otherwise, cut alpha in half and try again
    alpha = alpha/2;

  end
```

This line search strategy essentially relies on the fact that we can characterize a solution of $f(x) = 0$ in terms of a minimization of $\|f(x)\|$. Of course, this relationship goes the other way, too: for a differentiable objective function, we can write a nonlinear system of equations that define necessary condtions for a minimum.

Iterations for optimization

Suppose $g : \mathbb{R}^n \rightarrow \mathbb{R}$ is twice continuously differentiable near x_0. Then you might remember that Taylor's theorem gives

$$g(x + z) = g(x) + g'(x)z + \frac{1}{2}z^T H_g(x)z + O(\|z\|^3),$$

where H_g is the *Hessian matrix*

$$[H_g(x)]_{ij} = \frac{\partial^2 g(x)}{x_i x_j}.$$

A necessary conditions for x_* to be a local minimum or maximum of g is that $g'(x) = 0$. This suggests one way of trying to find a local minimum of g is simply Newton iteration (with a line search):

$$x^{k+1} = x^k - \alpha_k H_g(x^k)^{-1} \nabla g(x^k).$$

Unfortunately, even if Newton iteration converges to a critical point (a point where the gradient of g is zero), there is nothing to guarantee that this will be a minimum rather than a maximum. In order to make sure that we converge to a minimum, we would like to make sure not that $\|\nabla g\|$ decreases at each step, but that g decreases at each step! There are two ensuring this decrease:

1. We need the Newton direction (or some other search direction) to at least be a *descent* direction. That is, we want
$$x^{k+1} = x^k + \alpha_k d^k$$
where $\nabla g(x^k) \cdot d^k < 0$.

2. Once we have a descent direction, we want to make sure that the steps we take are short enough that we actually decrease g by some sufficient amount. The condition we use might look something like
$$g(x^{k+1}) \leq g(x^k) + \alpha^k \sigma \nabla g(x^k) \cdot d^k$$

Under what conditions can we guarantee that the Newton direction is actually a descent direction? If the Newton direction is
$$d^k = -H_g(x^k)^{-1} \nabla g(x^k),$$
then the descent condition looks like
$$\nabla g(x^k)^T d^k = -\nabla g(x^k)^T H_g(x^k)^{-1} \nabla g(x^k),$$
which is a quadratic form in $H_g(x^k)^{-1}$. So a sufficient condition for the Newton iteration to be a descent direction is that $H_g(x^k)$ is positive definite (and therefore that $H_g(x^k)^{-1}$ is positive definite). This suggests the following modification to the Newton approach to minimizing g:

- If the Hessian matrix $H_g(x^k)$ is positive definite, search in the Newton direction
$$d^k = -H_g(x^k)^{-1} \nabla g(x^k).$$
- If the Hessian is not positive definite at x^k, use a modified Newton direction
$$d^k = -\hat{H}^{-1} \nabla g(x^k).$$
where $\hat{H}$ is some positive definite matrix. Convergence tends to be fastest when $\hat{H}$ approximates the Hessian in some way (subject to the constraint of being positive definite), but one can also be lazy and just choose $\hat{H} = I$ (i.e. follow the direction of steepest descent).

Note that while it is possible to choose to a local minimum by choosing the steepest descent direction $-\nabla g(x^k)$ at *every* step, this approach can yield painfully slow convergence.

Problems to Ponder

1. Write a (guarded) Newton iteration to find the intersection of three spheres in three dimensional space, i.e. find x_* such that
$$\begin{aligned} \|x_* - x_a\| &= r_a \\ \|x_* - x_b\| &= r_b \\ \|x_* - x_c\| &= r_c \end{aligned}$$
Assume for the moment that there are exactly two solutions. If you find one, how might you easily find the other?

2. Consider the steepest descent iteration
$$x_{k+1} = x_k - \alpha_k \nabla \phi(x_k)$$
applied to
$$\phi(x) = \frac{1}{2} \begin{bmatrix} x_1 \\ x_t \end{bmatrix}^T \begin{bmatrix} 1 & 0 \\ 0 & 10^6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_t \end{bmatrix},$$
and suppose that α_k is chosen by *exact line search*: that is α_k is chosen to reduce $\phi(x_{k+1})$ as much as possible. Starting from $\begin{bmatrix} 1 & 1 \end{bmatrix}^T$, what are the iterates produced by this iteration? What can you say about the rate of convergence?

3. What is $\nabla_x \phi(x)$ for $\phi(x) = \|f(x)\|^2$? Argue based on your computation that the Newton direction is a descent direction for this objective function.

4. Write the critical point equations for minimizing $\|f(x) - b\|^2$.

5. The Gauss-Newton iteration for minimizing $\|f(x) - b\|^2$ is
$$\begin{aligned} p_k &= \left(J(x_k)^T J(x_k)\right)^{-1} J(x_k)^T (f(x_k) - b) \\ x^{k+1} &= x^k - \alpha_k p_k \end{aligned}$$
where $J(x_k)$ is the Jacobian of f. Argue that p_k is always a descent direction.

Week 9: Monday, Mar 26

Function approximation

A common task in scientific computing is to approximate a function. The approximated function might be available only through tabulated data, or it may be the output of some other numerical procedure, or it may be the solution to a differential equation. The approximating function is usually chosen because it is relatively simpler to evaluate and analyze. Depending on the context, we might want an approximation that is accurate for a narrow range of arguments (like a Taylor series), or we might want guaranteed global accuracy over a wide range of arguments. We might want an approximation that preserves properties like monotonicity or positivity (e.g. when approximating a probability density). We might want to exactly match measurements at specified points, or we might want an approximation that "smooths out" noisy data. We might care a great deal about the cost of forming the approximating function if it is only used a few times, or we might care more about the cost of evaluating the approximation after it has been formed. There are a huge number of possible tradeoffs, and it is worth keeping these types of questions in mind in practice.

Though function approximation is a huge subject, we will mostly focus on approximation by polynomials and piecewise polynomials. In particular, we will concentrate on *interpolation*, or finding (piecewise) polynomial approximating functions that exactly match a given function at specified points.

Polynomial interpolation

This is the basic polynomial interpolation problem: given data $\{(x_i, y_i)\}_{i=0}^d$ where all the t_i are distinct, find a degree d polynomial $p(x)$ such that $p(x_i) = y_i$ for each i. Such a polynomial always exists and is unique.

The Vandermonde approach

Maybe the most obvious way to approach to this problem is to write

$$p(x) = \sum_{j=0}^{d} c_j x^j,$$

where the unknown x_j are determined by the interpolation conditions

$$p(x_i) = \sum_{j=0}^{d} c_j x_i^j = y_j.$$

In matrix form, we can write the interpolation conditions as

$$Ac = y$$

where $a_{ij} = x_i^j$ (and we're now thinking of the index j as going from zero to d). The matrix A is a *Vandermonde matrix*. The Vandermonde matrix is nonsingular, and we can solve Vandermonde systems using ordinary Gaussian elimination in $O(d^3)$ time.

This is usually a bad way to compute things numerically. The problem is that the condition numbers of Vandermonde systems grow exponentially with the system size, yielding terribly ill-conditioned problems even for relatively small problems.

The Lagrange approach

The problem with the Vandermonde matrix is not in the basic setup, but in how we chose to represent the space of degree d polynomials. In general, we can write

$$p(x) = \sum_{j=0}^{d} c_j q_j(x)$$

where $\{q_j(x)\}$ is some other basis for the space of polynomials of degree at most d. The power basis $\{x^j\}$ just happens to be a poor choice from the perspective of conditioning.

One alternative to the power basis is a basis of *Lagrange polynomials*:

$$L_i(x) = \frac{\prod_{j \neq i}(x - x_i)}{\prod_{j \neq i}(x_j - x_i)}.$$

The polynomial L_i is characterized by the property

$$L_i(x_j) = \begin{cases} 1, & j = i \\ 0, & \text{otherwise.} \end{cases}$$

Therefore, if we write the interpolating polynomial in the form

$$p(x) = \sum_{j=0}^{d} c_j L_j(x),$$

the interpolation conditions yield the linear system

$$Ic = y,$$

i.e. we simply have

$$p(x) = \sum_{j=0}^{d} y_j L_j(x),$$

It is trivial to find the coefficients in a representation of an interpolant via Lagrange polynomials. But what if we want to evaluate the Lagrange form of the interpolant at some point? The most obvious algorithm costs $O(d^2)$ per evaluation, which is more expensive than the $O(d)$ cost of evaluating a polynomial in the usual monomial basis using Horner's rule.

Horner's rule

There are typically two tasks in applications of polynomial interpolation. The first task is getting some representation of the polynomial; the second task is to actually evaluate the polynomial. In the case of the power basis $\{x^j\}_{j=0}^d$, we would usually evaluate the polynomial in $O(d)$ time using Horner's method. You have likely seen this method before, but it is perhaps worth going through it one more time.

Horner's scheme can be written in terms of a recurrence, writing $p(x)$ as $p_0(x)$ where

$$p_j(x) = c_j + x p_{j+1}(x)$$

and $p_d(x) = c_d$. For example, if we had three data points, we would write

$$\begin{aligned}
p_2(x) &= c_2 \\
p_1(x) &= c_1 + x p_2(x) = c_1 + x c_2 \\
p_0(x) &= c_0 + x p_1(x) = c_0 + x c_1 + x^2 c_2.
\end{aligned}$$

Usually, we would just write a loop:

```
function px = peval(c,x)
  px = c(end)*x;
  for j = length(c)-1:-1:1
    px = c(j) + x.*px;
  end
```

But even if we would usually write the loop with no particular thought to the recurrence, it is worth remembering how to write the recurrence.

The idea of Horner's rule extends to other bases. For example, suppose we now write a quadratic as

$$p(x) = c_0 q_0(x) + c_1 q_1(x) + c_2 q_2(x).$$

An alternate way to write this is

$$p(x) = q_0(c_0 + q_1/q_0(c_1 + c_2 q_2/q_1));$$

more generally, we could write $p(x) = q_0(x)p_0(x)$ where $p_d(x) = c_d$ and

$$p_j(x) = c_j + p_{j+1}(x)q_{j+1}(x)/q_j(x).$$

In the case of the monomial basis, this is just Horner's rule, but the recurrence holds more generally.

The Newton approach

The Vandermonde approach to interpolation requires that we solve an ill-conditioned linear system (at a cost of $O(d^3)$) to find the interpolating polynomial. It then costs $O(d)$ per point to evaluate the polynomial. The Lagrange approach gives us a trivial linear system for the coefficients, but it then costs $O(d^2)$ per point to evaluate the resulting representation. Newton's form of the interpolant will give us a better balance: $O(d^2)$ time to find the coefficients, $O(d)$ time to evaluate the function.

Newton's interpolation scheme uses the polynomial basis

$$\begin{aligned} q_0(x) &= 1 \\ q_j(x) &= \prod_{k=1}^{j}(x - x_k), \quad j > 0. \end{aligned}$$

If we write

$$p(x) = \sum_{j=0}^{d} c_j q_j(x),$$

the interpolating conditions have the form

$$Uc = y,$$

where U is an upper triangular matrix with entries

$$u_{ij} = q_j(x_i) = \prod_{k=j}^{d} (t_i - t_j)$$

for $i = 0, \ldots d$ and $j = 0, \ldots, d$. Because U is upper triangular, we can compute the coefficients c_j in $O(d^2)$ time; and we can use the relationship $q_j(x) = (x - x_j)q_{j-1}(x)$ as the basis for a Horner-like scheme to evaluate $p(x)$ in $O(d)$ time (this is part of a problem on HW 5).

In practice, we typically do not form the matrix U in order to compute x. Instead, we express the components of x in terms of *divided differences.* That is, we write

$$c_j = y[x_1, \ldots, x_{j+1}]$$

where the coefficients $y[x_i, \ldots, x_j]$ are defined recursively by the relationship

$$\begin{aligned} y[x_i] &= y_i, \\ y[x_i, x_{i+1}, \ldots, x_j] &= \frac{y[x_i, x_{i+1}, \ldots, x_{j-1}] - y[x_{i+1}, \ldots, x_j]}{x_i - x_j}. \end{aligned}$$

Evaluating the x_j coefficients by divided differences turns out to be numerically preferable to forming U and solving by back-substitution.

Week 9: Wednesday, Mar 28

Summary of last time

We spent most of the last lecture discussing three forms of polynomial interpolation. In each case, we were given function values $\{y_i\}_{i=0}^d$ at points $\{x_i\}_{i-1}^d$, and we wanted to construct a degree d polynomial such that $p(x_i) = y_i$. We do this in general by writing

$$p(x) = \sum_{j=0}^{d} c_j \phi_j(x),$$

where the functions $\phi_j(x)$ form a basis for the space of polynomials of degree at most d. Then we use the interpolation conditions to determine the coefficients c_j via a linear system

$$Ac = y,$$

where $A_{ij} = \phi_j(x_i)$. In the last lecture, we considered three choices of basis functions $\phi_j(x)$:

1. Power basis:
$$\phi_j(x) = x^j.$$
2. Lagrange basis:
$$\phi_j(x) = \frac{\prod_{i \neq j}(x - x_i)}{\prod_{i \neq j}(x_j - x_i)}.$$
3. Newton basis:
$$\phi_j(x) = \prod_{i<j}(x - x_i).$$

The power basis yields an ill-conditioned system matrix (the Vandermonde matrix). The Lagrange basis leads to a trivial linear system, but it takes $O(d)$ time to evaluate each Lagrange polynomial and so $O(d^2)$ time to evluate the interpolant. The Newton basis is a nice compromise: the coefficients can be computed in $O(d^2)$ time as the solution to an upper triangular system or through a divided difference recurrence, and the polynomial itself can be evaluated in $O(d)$ time using an algorithm like Horner's rule.

Divided differences and derivatives

The coefficients in the Newton form of the interpolant are *divided differences*. For a given function f known at sample points $\{x_i\}_{i=1}^n$, we can evaluate divided differences recursively:

$$\begin{aligned} f[x_i] &= f(x_i), \\ f[x_i, x_{i+1}, \ldots, x_j] &= \frac{f[x_i, x_{i+1}, \ldots, x_{j-1}] - f[x_{i+1}, \ldots, x_j]}{x_i - x_j}. \end{aligned}$$

This recurrence is numerically preferable to finding the coefficients of the Newton interpolant by back substitution.

You might recognize the first divided difference $f[x_1, x_2]$ as a derivative approximation. In fact, if f is a differentiable function, then the mean value theorem tells us that $f[x_1, x_2] = f'(\xi)$ for some ξ between x_1 and x_2. Thus if f is a continuously differentiable function, it makes sense to define

$$f[x_i, x_i] \equiv f'(x_i).$$

This gives us a natural way to solve *Hermite* interpolation problems in which we specify both function values and derivatives at specified points.

More generally, it turns out that if $f \in C^{m-1}$, then

$$f[x_1, x_2, \ldots, x_m] = \frac{f^{(m-1)}(\xi)}{(m-1)!}, \quad \text{some } \xi \in (\min\{x_i\}, \max\{x_i\})$$

Therefore, in the limiting case as we let all the x_j approach some common point x_0, the Newton form of the interpolant degenerates into a Taylor approximation.

Error in polynomial approximation

The relation between divided differences and derivatives is incredibly useful in reasoning about how well polynomial interpolants approximate an underlying function. Suppose we approximate $f \in C^n$ by a polynomial p of degree $n-1$ that interpolates f at points $\{x_i\}_{i=1}^n$. At any point x, we can write $f(x) = p^*(x)$, where $p^*(x)$ is the degree n polynomial interpolating f at

$\{x_i\}_{i=1}^n \cup \{x\}$. This may seem somewhat silly, but it gives us the error representation

$$\begin{aligned} f(x) - p(x) &= p^*(x) - p(x) \\ &= f[x_1, \ldots, x_n, x] \prod_{i=1}^n (x - x_i) \\ &= \frac{f^{(n)}(\theta)}{n!} \prod_{i=1}^n (x - x_i). \end{aligned}$$

If x lies within h of all the values x_i and $|f^{(n)}| \leq M_n$ on the interval bounded by the points in question, then we have

$$|f(x) - p(x)| \leq \frac{M_n h^n}{n!}.$$

This bound suggests that high-order polynomial interpolation of a smooth function over a bounded interval can provide very accurate approximations to the function values, with two catches. First, the h^n term may not be small (especially in *extrapolation*, where x lies outside the convex hull of the data points). Second, M_n may grow quickly as a function of n. Note that these two effects are not independent; for example, we can scale the nodal coordinates to make h smaller, but then M_n gets commensurately bigger. The standard example of these effects, due to Runge, is the function

$$\phi(t) = \frac{1}{1 + 25t^2}.$$

Polynomial approximations to $\phi(t)$ by interpolation on a uniform mesh on $[-1, 1]$ oscillate wildly toward the end points of the interval, and it is not true in this case that ever higher-degree interpolating polynomials provide ever-better function approximations. This is a general problem, known as the *Runge phenomena*, and there are two standard fixes. The first fix is to use something other than polynomials (piecewise polynomial functions are particularly popular). We will talk about this option next week. The second approach involves optimizing the location of the sample points, a topic which we will turn to now.

Chebyshev interpolation

Suppose we want a polynomial interpolant that accurately represents some function on a bounded interval. Earlier, we showed that

$$f(x) - p(x) = \frac{f^{(n)}(\theta)}{n!} \prod_{i=1}^{n} (x - x_i).$$

If $|f^{(n)}(x)| \leq M$ on the interval $[a, b]$, then

$$|f(x) - p(x)| \leq \frac{M}{n!} \prod_{i=1}^{n} (x - x_i).$$

So one natural approach to trying to build accurate interpolants is to try to minimize some norm that measures the size of

$$\psi(x) = \prod_{i=1}^{n} (x - x_i)$$

over the interval $[a, b]$. If we care about the values pointwise, it makes sense to try to choose the interpolation points to minimize

$$\|\psi\|_{L^\infty([a,b])} = \max_{x \in [a,b]} |\psi(x)|.$$

This leads to the choice of *Chebyshev points* on $[-1, 1]$

$$\xi_i = \cos\left(\frac{2i-1}{2n}\pi\right), \quad i = 1, \ldots, n.$$

For more general intervals, we can simply apply an affine mapping to get the interpolation points

$$x_i = a + \frac{b-a}{2}(\xi_i + 1).$$

If we choose the Chebyshev points as interpolation nodes, then

$$\|\psi\|_{L^\infty([a,b])} = 2^{1-n}$$

and so we have the error bound

$$|f(x) - p(x)| \leq \frac{M}{2^{n-1} n!}$$

for $x \in [a, b]$.

Problems to ponder

1. Suppose $p(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3$. Write a linear system for the coefficients a_j such that $p(0) = p_0$, $p'(0) = q_0$, $p(1) = p_1$, $p'(1) = q_1$.

2. For the points $x_1 = -1, x_2 = 0, x_3 = 1$ and the values of $y_1 = 0, y_2 = 1, y_3 = 1$, write the interpolating polynomial in power form, Lagrange form, and Newton form.

3. In lecture, I described Horner's rule for evaluating a polynomial
$$p(x) = \sum_{j=0}^{d} c_j x^j$$
in terms of the recurrence
$$\begin{aligned} p_{d+1}(x) &= 0 \\ p_j(x) &= x p_{j+1}(x) + c_j. \end{aligned}$$
What is the equivalent recurrence for evaluating the Newton form of the interpolant?

4. Describe how to find coefficients c_i such that
$$g(x) = c_1 + c_2 x + c_3 \sin(x) + c_4 \cos(x)$$
interpolates $f(x)$ at distinct points x_1, x_2, x_3, x_4.

5. In class, we wrote the Lagrange interpolant as
$$p(x) = \sum_{j=1^n} y_i L_i(x)$$
where $L_i(x) = \prod_{k \neq i}(x - x_k)$. The book describes computation of the Lagrange interpolant via the barycentric formula
$$p(x) = \frac{\sum_{j=1^n} w_j y_j/(x - x_j)}{\sum_{j=1^n} w_j/(x - x_j)}$$
where $w_j^{-1} = \prod_{i \neq j}(x_j - x_i)$. Why are these formulas equivalent, and what is the advantage of barycentric interpolation?

Week 10: Monday, Apr 2

Hermite interpolation

For standard polynomial interpolation problems, we seek to satisfy conditions of the form

$$p(x_j) = y_j,$$

where y_j is frequently a sampled function value $f(x_j)$. If all we know is function values, this is a reasonable approach. But sometimes we have more information. *Hermite* interpolation constructs an interpolant based not only on equations for the function values, but also for the derivatives.

For example, consider the important special case of finding a cubic polynomial that satisfies proscribed conditions on the values and derivatives at the endpoints of the interval $[-1, 1]$. That is, we require

$$\begin{aligned} p(1) &= f(1) & p(-1) &= f(-1) \\ p'(1) &= f'(1) & p'(-1) &= f'(-1). \end{aligned}$$

As with polynomial interpolation based just on function values, we can express the cubic that satisfies these conditions with respect to several different bases: monomial, Lagrange, or Newton.

For the monomial basis, we have

$$p(x) = c_0 + c_1 x + c_2 x^2 + c_3 x^3,$$

which yields the linear system

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 0 & 1 & 2 & 3 \\ 0 & 1 & -2 & 3 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} f(1) \\ f(-1) \\ f'(1) \\ f'(-1) \end{bmatrix}.$$

For the Newton basis, we have expressions like

$$p(x) = f[-1]+f[-1,-1](x+1)+f[1,-1,-1](x+1)^2+f[1,1,-1,-1](x+1)^2(x-1),$$

where the divided differences for f are

$$\begin{aligned}
f[1] &= f(1) \\
f[-1] &= f(-1) \\
f[1,1] &= f'(1) \\
f[-1,-1] &= f'(-1) \\
f[1,-1] &= \left(f(1) - f(-1)\right)/2 \\
f[1,-1,-1] &= \left(f[1,-1] - f[-1,-1]\right)/2 \\
f[1,1,-1] &= \left(f[1,1] - f[1,-1]\right)/2 \\
f[1,1,-1,-1] &= \left(f[1,1,-1] - f[1,-1,-1]\right)/2.
\end{aligned}$$

We leave the Lagrange basis as a problem to ponder (or look up).

Piecewise polynomial approximations

Polynomials are convenient for interpolation for a few reasons: we know how to manipulate them symbolically, we can evaluate them quickly, and there is a theorem of analysis (the Weierstrass approximation theorem) that says that any continuous function on some interval $[a,b]$ can be uniformly approximated by polynomials. In practice, though, high-degree polynomial interpolation does not always provide fantastic function approximation. An alternative approach that retains the advantages of working with polynomials is to work with *piecewise* polynomial functions.

Piecewise linear interpolation

Perhaps the simplest example is piecewise linear interpolation; if function values $f(x_j)$ are given at points $x_1 < x_2 < x_3 < \ldots < x_n$, then we write the approximating function $\hat{f}(x)$ as

$$\hat{f}(x) = \frac{f(x_j)(x - x_j) + f(x_{j+1})(x_{j+1} - x)}{x_{j+1} - x_j}, \quad x \in [x_j, x_{j+1}].$$

Alternately, we can write

$$\hat{f}(x) = \sum_{j=1}^{n} \phi_j(x) f(x_j)$$

where $\phi_j(x)$ is a "hat function":

$$\phi_j(x) = \begin{cases} (x - x_{j+1})/(x_j - x_{j+1}), & x \in [x_j, x_{j+1}], \\ (x - x_{j-1})/(x_j - x_{j-1}), & x \in [x_{j-1}, x_j], \\ 0 & \text{otherwise.} \end{cases}$$

This last you may recognize as similar in spirit to using a basis of Lagrange polynomials for polynomial interpolation.

Using piecewise linear interpolation to approximate a function f yields $O(h^2)$ error (where h is the distance between interpolation points), assuming f has two continuous derivatives. This level of accuracy is adequate for many purposes. Beyond the basic error behavior, though, piecewise linear interpolation has several virtues when *structural* properties are important. For example, the maximum and minimum values of a piecewise linear interpolant are equal to the maximum and minimum values of the data. And if f is positive or monotone (like a probability density or cumulative density function), then any piecewise linear interpolant inherits these properties.

Piecewise cubic interpolation

If f is reasonably smooth and the data points are widely spaced, it may make sense to use higher-order polynomials. For example, we might decide to use a *cubic spline* $\hat{f}(x)$ characterized by the properties:

- Interpolation: $\hat{f}(x_i) = f(x_i)$
- Twice differentiability: $\hat{f}'$ and $\hat{f}''$ are continuous at $\{x_2, \ldots, x_{n-1}\}$

The interpolation and differentiability constraints give us $4n - 2$ constraints on the $4n$-dimensional space of piecewise polynomial functions that are defined by general cubics on each interval $[x_j, x_{j+1}]$. In order to uniquely determine the spline, we need some additional constraint; common choices are

- Specified values of f' at x_1 and x_n (clamped conditions)
- A natural spline: $f''(x_1) = f''(x_n) = 0$
- Not-a-knot conditions: f''' is continuous at x_2 and x_{n-1}
- Periodicity: $f'(x_1) = f'(x_n), f''(x_1) = f''(x_n)$

For the clamped conditions and not-a-knot conditions, one has the error bound

$$\|p - f\|_\infty \leq c\|f'''\|_\infty h^4$$

where $\|\cdot\|_\infty$ is the L^∞ norm (max norm) on some interval of interest and h is the maximum space between interpolation nodes.

In addition to spline conditions, one can choose piecewise cubic polynomials that satisfy Hermite interpolation conditions (sometimes referred to by the acronym PCHIP or Piecewise Cubic Hermite Interpolating Polynomials). That is, the function values and derivatives are specified at each nodal point. If we don't actually have derivative values prescribed at the nodal points, then we can assign these values to satisfy additional constraints. We gain this flexibility at the cost of some differentiability; piecewise cubic Hermite interpolants are in general not twice continuously differentiable.

As in the case of polynomial interpolation, there are several different bases for the space of piecewise cubic functions. Any choice of locally supported basis functions (basis functions that are only nonzero on only a fixed number of intervals $[x_j, x_{j+1}]$) leads to a banded linear system which can be solved in $O(n)$ time to find either cubic splines or piecewise Hermite cubic interpolants. One common choice of basis is the B-spline basis, which you can find described in the book.

Week 11: Monday, Apr 9

Maximizing an interpolating quadratic

Suppose that a function f is evaluated on a reasonably fine, uniform mesh $\{x_i\}_{i=0}^n$ with spacing $h = x_{i+1} - x_i$. How can we find any local maxima inside the mesh interval (x_0, x_n)?

A natural first approximation is to simply find local maxima in the discrete sequence $\{f(x_i)\}_{i=0}^n$. I would usually do that by looking for a place where differences between adjacent points change from positive to negative (the discrete analog of looking for a critical point where the derivative changes from positive to negative):

```
% [idx] = find_local_max(fi)
%
% Based on samples fi of a function on a uniform mesh over
% an interval, find the indices of mesh points where there
% are discrete local maxima.

function [idx] = find_local_max(fi)

  d_fi = fi(2:end)-fi(1:end-1);
  idx  = find( d_fi:end-1) > 0 & d_fi(2:end) <= 0 );
  idx  = idx+1;
```

Unfortunately, unless we use a rather fine mesh, this method is unlikely to give us more than a couple digits of accuracy. A simple method of improving the accuracy of the result is to fit a polynomial interpolant to the data near the discrete local maximum, and use a maximum of the interpolating polynomial as an estimate for the local maximum of f. The simplest variant of this is to fit a quadratic; let's look in a little detail at how this works.

Suppose x_j is an interior mesh point where f has a discrete local maximum. We would like to find a corrected estimate of the local maximum, $x_* = x_j + z$, by maximizing a quadratic interpolant through x_{j-1}, x_j, and

x_{j+1}. In terms of the correction z, the interpolation conditions are

$$\begin{aligned} p(0) &= f(x_j + 0) = f(x_j) \\ p(h) &= f(x_j + h) = f(x_{j+1}) \\ p(-h) &= f(x_j - h) = f(x_{j-1}) \end{aligned}$$

In a homework exercise, we saw how to differentiate a polynomial interpolant written in the Newton basis. For variety, let's now write things in terms of the Lagrange polynomials for $\{0, h, -h\}$:

$$\begin{aligned} p(z) &= \frac{p(0)}{h^2}(h^2 - z^2) + \frac{p(h)}{2h^2} z(z+h) + \frac{p(-h)}{2h^2} z(z-h) \\ &= p(0) + \left(\frac{p(h) - p(-h)}{2h} \right) z + \frac{1}{2} \left(\frac{p(h) - 2p(0) + p(-h)}{h^2} \right) z^2. \end{aligned}$$

Note that this last expression is just $p(z)$ expressed in Taylor series form:

$$p(z) = p(0) + p'(0)z + \frac{1}{2}p''(0)z^2.$$

where

$$\begin{aligned} p'(0) &= p[h, -h] = \frac{p(h) - p(-h)}{2h}, \\ p''(0) &= 2p[h, 0, -h] = \frac{p(h) - 2p(0) + p(-h)}{h^2}. \end{aligned}$$

Therefore, the maximum z_* for p satisfies

$$z_* = -\frac{p'(0)}{p''(0)}, \qquad p(z_*) = p(0) - \frac{p'(0)^2}{2p''(0)}.$$

It's worth comparing this maximization to what we would do if we took x_j as an initial guess at the maximum and did one step of Newton iteration to improve our guess:

$$x^{\text{new}} = x_j - \frac{f'(x_j)}{f''(x_j)}$$

The correction z_* looks just like what we would compute in one Newton step, but with the approximations $f'(x_j) \approx p'(0)$ and $f''(x_j) \approx p''(0)$!

Two ways to numerical differentiation

One way to approximate derivatives is by *interpolation*. If we can use interpolation to estimate function values, why not use it to estimate derivatives as well? The basic procedure here is:

- Interpolate f at some nodes $x_0, \ldots, x_n$.
- Differentiate the interpolating polynomial in order to approximate derivatives of f. Usually, one is interested in the derivative at one of the node points.

In general, if the interpolation points $x_0, \ldots, x_n$ all lie within an interval of length h, and if f has enough continuous derivatives in that interval, we have

$$p^{(k)}(x_j) = O(h^{n+1-k}).$$

The error analysis is relatively straightforward, and is in the book; but I did not drag you through the algebra in class, and do not intend to do so here.

This, therefore, is one way of thinking about numerical differentiation. Another way to get to the same end is to manipulate Taylor series. For example, in the previous section we derived the *centered difference* approximations by differentiating a quadratic interpolant:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}, \quad f''(x) \approx \frac{f(x+h) - 2f(0) + f(x-h)}{h^2}.$$

We could have also said "we have $f(x)$, $f(x+h)$, and $f(x-h)$; what linear combination of these values best approximates $f'(x)$ (or $f''(x)$)?" That is, we somehow want to choose coefficients a_+, a_0, a_- so that we get a good approximation of $f'(x)$ of the form

$$f'(x) \approx \hat{f}'(x) \equiv a_0 f(x) + a_+ f(x+h) + a_- f(x-h).$$

Note that we can Taylor expand the terms in $\hat{f}'(x)$ about x to get

$$\begin{aligned} \hat{f}'(x) = & (a_0 + a_+ + a_-) f(x) \\ & + h(a_+ - a_-) f'(x) \\ & + \frac{h^2}{2}(a_+ + a_-) f''(x) \\ & + \frac{h^3}{6}(a_+ - a_-) f'''(x) \\ & + O(h^4) \end{aligned}$$

We can adjust the three coefficients to match the first three terms in this series with the the (trivial) Taylor series for $f'(x)$ by solving the linear system

$$\begin{aligned}(a_0 + a_+ + a_-) &= 0 \\ h(a_+ - a_-) &= 1 \\ \frac{h^2}{2}(a_+ + a_-) &= 0.\end{aligned}$$

This gives us

$$a_0 = 0, \quad a_\pm = \pm\frac{1}{2h}$$

or

$$\hat{f}'(x) = \frac{f(x+h) - f(x-h)}{2h} = f'(x) + O(h^2)$$

Let's walk through the same exercise for computing the second derivative. We want a formula of the form

$$\hat{f}''(x) = b_0 f(x) + b_+ f(x+h) + b_- f(x-h),$$

and Taylor expanding each term in the right hand side about zero gives

$$\begin{aligned}\hat{f}'(x) = & (b_0 + b_+ + b_-)f(x) \\ &+ h(b_+ - b_-)f'(x) \\ &+ \frac{h^2}{2}(b_+ + b_-)f''(x) \\ &+ \frac{h^3}{6}(b_+ - b_-)f'''(x) \\ &+ O(h^4)\end{aligned}$$

Setting the first three terms in this series to match $f''(x)$, we get the equations

$$\begin{aligned}(b_0 + b_+ + b_-) &= 0 \\ h(b_+ - b_-) &= 0 \\ \frac{h^2}{2}(b_+ + b_-) &= 1,\end{aligned}$$

which has the solution

$$b_0 = -\frac{2}{h^2}, \quad b_\pm = \frac{1}{h^2}.$$

Notice that because $b_+ - b_-$, we also automatically get that

$$\frac{h^3}{6}(b_+ - b_-)f''(x) = 0,$$

and so

$$\hat{f}''(x) - f''(x) = O(h^2),$$

which is one better order of accuracy than we might have expected from looking too uncautiously at the bound based on the derivation via polynomial interpolation.

Week 11: Wednesday, Apr 11

Truncation versus rounding

Last week, we discussed two different ways to derive the centered difference approximation to the first derivative

$$f'(x) \approx f[x+h, x-h] = \frac{f(x+h) - f(x-h)}{2h}.$$

Using Taylor series, we were also able to write down an estimate of the *truncation error*:

$$f[x+h, x-h] - f'(x) = \frac{h^2}{6} f'''(x) + O(h^4).$$

As h grows smaller and smaller, $f[x+h, x-h]$ becomes a better and better approximation to $f'(x)$ — at least, it does in exact arithmetic. If we plot the truncation error $|h^2/6 f'''(x)|$ against h on a log-log scale, we expect to see a nice straight line with slope 2. But Figure 1 shows that something rather different happens in floating point. Try it for yourself!

The problem, of course, is cancellation. As h goes to zero, $f(x+h)$ and $f(x-h)$ get close together; and for h small enough, the computed value of $f(x+h) - f(x-h)$ starts to be dominated by rounding error. If the values of $f(x+h)$ and $f(x-h)$ are computed in floating point as $f(x+h)(1+\delta_1)$ and $f(x-h)(1+\delta_2)$, then the computed finite difference is approximately

$$\hat{f}[h, -h] = f[h, -h] + \frac{\delta_1 f(x+h) - \delta_2 f(x-h)}{2h},$$

and if we manage to get the values of $f(x+h)$ and $f(x-h)$ correctly rounded, we have

$$\left| \frac{\delta_1 f(x+h) - \delta_2 f(x-h)}{2h} \right| \leq \frac{\epsilon_{\mathrm{mach}}}{h} \left(\max_{x-h \leq \xi \leq x+h} |f(\xi)| \right) \approx \frac{\epsilon_{\mathrm{mach}}}{h} f(x).$$

The total error in approximating $f'(x)$ by $f[x+h, x-h]$ in floating point therefore consists of two pieces: truncation error proportional to h^2, and rounding error proportional to $\epsilon_{\mathrm{mach}}/h$. The total error is minimized when these two effects are approximately equal, at

$$h \approx \left(\frac{6 f(x)}{f'''(x)} \epsilon_{\mathrm{mach}} \right)^{1/3},$$

i.e. when h is close to $\epsilon_{\mathrm{mach}}^{1/3}$. From the plot in Figure 1, we can see that this is right — the minimum observed error occurs for h pretty close to $\epsilon_{\mathrm{mach}}^{1/3}$ (around 10^{-5}).

Of course, the analysis in the previous paragraph assumed the happy circumstance that we could get our hands on the correctly rounded values of $f(x+h)$ and $f(x-h)$. In general, we might have a little more error inherited from the evaluation of f itself, which would just make the optimal h (and the corresponding optimal accuracy) that much larger.

Richardson extrapolation

Let's put aside our concerns about rounding error for a moment, and just look at the truncation error in the centered difference approximation of $f'(x)$. We have an estimate of the form

$$f[x+h, x-h] - f'(x) = \frac{h^2}{6} f'''(x) + O(h^4).$$

Usually we don't get to write down such a sharp estimate for the error. There is a good reason for this: if we have a very sharp error estimate, we can use the estimate to reduce the error! The general trick is this: if we have $g_h(x) \approx g(x)$ with an error expansion of the form

$$g_h(x) = g(x) + Ch^p + O(h^{p+1}),$$

then we can write

$$a g_h(x) + b g_{2h}(x) = (a+b) g(x) + C(a + 2^p b) h^p + O(h^{p+1}).$$

Now find coefficients a and b so that

$$\begin{aligned} a + b &= 1 \\ a + 2^p b &= 0; \end{aligned}$$

the solution to this system is

$$a = \frac{2^p}{2^p - 1}, \quad b = -\frac{1}{2^p - 1}.$$

Therefore, we have

$$\frac{2^p g_h(x) - g_{2h}(x)}{2^p - 1} = g(x) + O(h^{p+1});$$

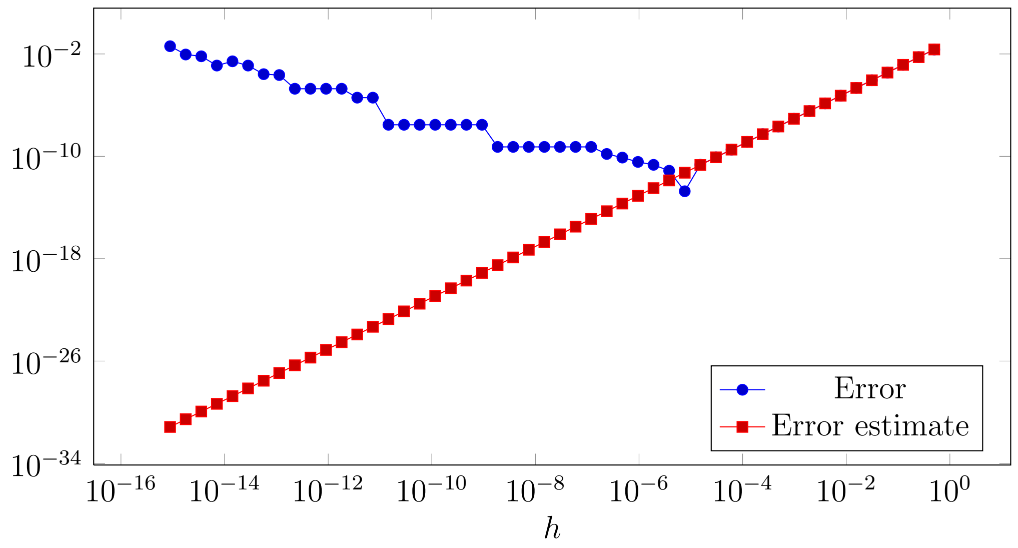


Figure 1: Actual error and estimated truncation error for a centered difference approximation to $\frac{d}{dx}\sin(x)$ at $x = 1$. For small h values, the error is dominated by roundoff rather than by truncation error.

```
%
% Compute actual error and estimated truncation error
% for a centered difference approximation to sin'(x)
% at x = 1.
%
h      = 2.^-(1:50);
fd     = ( sin(1+h)-sin(1-h) )./h/2;
err    = fd-cos(1);
errest = -h.^2/6 * cos(1);

%
% Plot the actual error and estimated truncation error
% versus h on a log-log scale.
%
loglog(h, abs(err), h, abs(errest));
legend('Error', 'Error estimate');
xlabel('h');
```

Figure 2: Code to produce Figure 1.

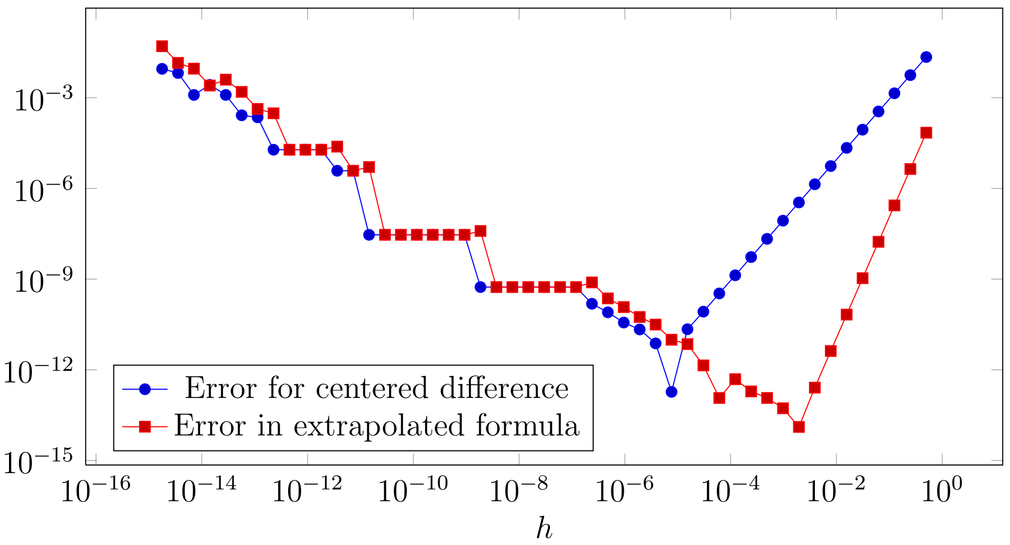


Figure 3: Actual error and estimated truncation error for a centered difference approximation to $\frac{d}{dx}\sin(x)$ at $x = 1$. For small h values, the error is dominated by roundoff rather than by truncation error.

that is, we have cancelled off the leading term in the error.

In the case of the centered difference formula, only even powers of h appear in the series expansion of the error; so we actually have that

$$\frac{4f[x+h, x-h] - f[x+2h, x-2h]}{3} = f'(0) + O(h^4).$$

An advantage of the higher order of accuracy is that we can get very small truncation errors even when h is not very small, and so we tend to be able to reach a better optimal error before cancellation effects start to dominate; see Figure 3.

Problems to ponder

1. Suppose that $f(x)$ is smooth and has a single local maximum between $[h, -h]$, and let $p_h(x)$ denote the quadratic interpolant through 0, h, and $-h$. Argue that if the second derivative of f is bounded away from zero near 0, then the actual maximizing point x_* for f satisfies
$$x_* = -\frac{p_h'(0)}{p_h''(0)} + O(h^2).$$

2. Suppose we know $f(x)$, $f(x+h)$, and $f(x+2h)$. Both by interpolation and by manipulation of Taylor series, find a formula to estimate $f'(x)$ of the form $c_0 f(x) + c_1 f(x+h) + c_2 f(x+2h)$. Using Taylor expansions about x, also estimate the truncation error.

3. Consider the *one-sided* finite difference approximation
$$f'(x) \approx f[x+h, x] = \frac{f(x+h) - f(x)}{h}.$$
   (a) Show using Taylor series that
$$f[0,h] - f'(0) = \frac{1}{2} f''(0)h + O(h^2).$$
   (b) Apply Richardson extrapolation to this approximation.

4. Verify that the extrapolated centered difference approximation to $f'(x)$ is the same as the approximation derived by differentiating the quartic that passes through f at $\{x-2h, x-h, x, x+h, x+2h\}$.

5. Richardson extrapolation is just one example of an *acceleration* technique that can turn a slowly-convergent sequence of estimates into something that converges more quickly. We can use the same idea in other cases. For example, suppose we believe a one-dimensional iteration $x_{k+1} = g(x_k)$ converges linearly to a fixed point x_*. Then
   (a) Suppose the rate constant $C = g'(x_*)$ is known. Using
$$e_{k+1} = Ce_k + O(e_k^2),$$
   show that
$$\frac{x_{k+1} - Cx_k}{1 - C} = x_* + O(e_k^2)$$

(b) Show that the rate constant $g'(x_*)$ can be estimated by

$$C_k \equiv \frac{x_{k+2} - x_{k+1}}{x_{k+1} - x_k} \rightarrow g'(x_*)$$

(c) If you are bored and feel like doing algebra, show that

$$y_k \equiv \frac{x_{k+1} - C_k x_k}{1 - C_k} = \frac{x_k x_{k+2} - x_{k+1}^2}{x_{k+2} - 2x_{k+1} + x_k},$$

and using the techniques developed in the first two parts, that $y_k - x_* = O((x_k - x_*)^2)$.

The transformation from the sequence x_k into the (more rapidly convergent) sequence y_k is sometimes known as *Aitken's delta-squared process*. The process can sometimes be applied repeatedly. You may find it entertaining to try running this transformation repeatedly on the partial sums of the alternating harmonic series

$$S_n = \sum_{j=1}^{n} \frac{(-1)^{j+1}}{j},$$

which converges very slowly to $\ln(2)$. Without any transformation, S_{20} has an error of greater than 10^{-2}; one step of transformation reduces that to nearly 10^{-5}; and with three steps, one is below 10^{-7}.

Week 12: Monday, Apr 16

Panel integration

Suppose we want to compute the integral

$$\int_a^b f(x)\,dx$$

In estimating a derivative, it makes sense to use a locally accurate approximation to the function around the point where the derivative is to be evaluated. But if f is at all interesting on the interval $[a, b]$, it probably does not make sense to approximate the integral of f by integrating a quadratic interpolant. On the other hand, f may be approximated quite well by a quadratic interpolant on small subintervals, so it may make sense to define a mesh of points

$$a = a_0 < a_1 < a_2 < \ldots < a_n = b$$

and to then compute

$$\int_a^b f(x)\,dx = \sum_{i=0}^{n-1} \int_{a_i}^{a_{i+1}} f(x)\,dx,$$

where the integrals on each panel $[a_i, a_{i+1}]$ involves a local polynomial approximation. Let us now turn to a method to compute these panel integrals.

Simpson's rule

Now, suppose we do exactly the same manipulations we used to find the centered difference approximation, but aim at coming up with an integration rule. That is, given $f(-h)$, $f(0)$, and $f(h)$, how can we estimate $\int_{-h}^{h} f(x)\,dx$? Again, we can derive the same answer by either interpolation or by the method of undetermined coefficients. Let's use interpolation first:

$$\begin{aligned}
\int_{-h}^{h} f(x)\,dx &\approx \int_{-h}^{h} p(x)\,dx \\
&= \int_{-h}^{h} \left[f(-h)L_{-h}(x) + f(0)L_0(x) + f(h)L_h(x)\right]\,dx \\
&= w_- f(-h) + w_0 f(0) + w_+ f(h), \\
w_- &= \int_{-h}^{h} L_{-h}(x)\,dx = h/3 \\
w_0 &= \int_{-h}^{h} L_0(x)\,dx = 4h/3 \\
w_+ &= \int_{-h}^{h} L_h(x)\,dx = h/3.
\end{aligned}$$

What about the method of undetermined coefficients? Let's start by integrating a Taylor expansion of f about 0:

$$\int_{-h}^{h} f(x)\,dx = 2\left[f(0)h + f''(0)h^3/6 + f^{(4)}(0)h^5/120 + O(h^7)\right]$$

We want to match terms in this Taylor expansion to the terms in the Taylor expansion of a linear combination of $f(-h)$, $f(0)$, and $f(h)$:

$$\begin{aligned}
I(h) &= c_- f(-h) + c_0 f(0) + c_+ f(h) \\
&= (c_- + c_0 + c_+) f(0) + (c_+ - c_-) f'(0)h + (c_+ + c_-) f''(0)h^2/2 \\
&\quad + (c_+ - c_-) f^{(3)}(0)h^3/6 + (c_+ + c_-) f^{(4)}(0)h^4/24 + O(h^5)
\end{aligned}$$

If we match the constant, linear, and quadratic terms between the two expansions, we have

$$\begin{aligned}
c_- + c_0 + c_+ &= 2h \\
c_+ - c_- &= 0 \\
c_+ + c_- &= 2h/3
\end{aligned}$$

Solving gives us $c_+ = c_- = h/3$ and $c_0 = 4h/3$, and

$$\int_{-h}^{h} f(x)\,dx - I(h) = O(h^5).$$

A brief digression on changing variables

How do I get from a rule on the domain $[-h, h]$ to a rule on the domain $[a_i, a_{i+1}]$? Define $h = (a_{i+1} - a_i)/2$ and $a_{i+1/2} = (a_{i+1} + a_i)/2$. Then we can define $x = a_{i+1/2} + z$, and since $dx/dz = 1$, the change of variables formula gives

$$\int_{a_i}^{a_{i+1}} f(x)\,dx = \int_{-h}^{h} f(a_{i+1/2} + z)\,dz.$$

For example, Simpson's rule on the interval $[a_i, a_{i+1}]$ is

$$\int_{a_i}^{a_{i+1}} f(x)\,dx = \frac{b-a}{6}\left[f(a_i) + 4f(a_{i+1/2}) + f(a_{i+1})\right].$$

Newton-Cotes rules

Simpson's rule is a member of the family of *Newton-Cotes* rules based on interpolation over a uniform mesh. The first three Newton-Cotes rules are

1. Midpoint: $\int_{-h}^{h} f(x)\,dx \approx 2hf(0)$
2. Trapezoidal: $\int_{-h}^{h} f(x)\,dx \approx h\,[f(-h) + f(h)]$
3. Simpson's: $\int_{-h}^{h} f(x)\,dx \approx h/3\,[f(-h) + 4f(0) + f(h)]$

The midpoint and trapezoidal rules have order $O(h^3)$ per panel, and Simpson's rule has $O(h^5)$ per panel. If the panel sizes are fixed, we generally have $O(1/h)$ panels to cover the domain $[a, b]$, so the absolute error in approximating an integral by composite midpoint or trapezoidal integration is $O(h^2)$, and the error for composite Simpson's rule is $O(h^4)$.

In general, n-point Newton-Cotes rules exactly integrate polynomials of degree $n - 1$ if n is even, degree n if n is odd (the degree of polynomial we integrate exactly is called the degree of the quadrature rule). Newton-Cotes rules with more than three or four points are uncommon in practice; for $n \geq 11$, the Newton-Cotes rules always have at least one negative weight, and cancellation causes problems in finite precision. Instead, Newton-Cotes rules are usually used in panel integration schemes, often with adaptive panel sizes based on local error estimates.

Error estimates

We have already seen one approach to writing a formula for the error in a quadrature rule: Taylor expand everything in sight, and get a formula involving high-order derivatives of f. Unfortunately, we may not always have easy access to bounds on the derivatives of f. In practice, we would therefore usually estimate the error by comparing the results of two integration rules with different errors over a panel.

For example, on $[-h, h]$, let us write the integral, the midpoint rule, and the trapezoidal rule as

$$\begin{aligned} I[f] &= \int_{-h}^{h} f(x)\,dx = 2hf(0) + f''(0)h^3/3 + O(h^5) \\ Q_M[f] &= 2hf(0) \\ Q_T[f] &= h\left[f(-h) + f(h)\right] = 2hf(0) + f''(0)h^3 + O(h^5). \end{aligned}$$

The error in the midpoint rule and the trapezoidal rule are thus

$$\begin{aligned} Q_M[f] - I[f] &= -f''(0)h^3/3 + O(h^5) \\ Q_T[f] - I[f] &= 2f''(0)h^3/3 + O(h^5). \end{aligned}$$

The error in the trapezoidal rule is thus about twice the size of the error in the midpoint rule. Moreover, we can estimate $Q_M[f] - I[f]$ even if we don't have direct access to the integral $I[f]$:

$$Q_M[f] - I[f] = (Q_M[f] - Q_T[f])/3 + O(h^5).$$

Note that this suggests we could get a more accurate formula by correcting $Q_M[f]$ with our estimate of the error:

$$I[f] = Q_M[f] - (Q_M[f] - Q_T[f])/3 + O(h^5).$$

But note that

$$Q_M[f] - (Q_M[f] - Q_T[f])/3 = \frac{h}{3}\left[f(-h) + 4f(0) + f(h)\right],$$

which is just Simpson's rule.

Degree of an integration rule

Suppose we write

$$
\begin{aligned}
I_h[f] &= \int_0^h f(x)\,dx \\
Q_h[f] &= h \sum_{j=1}^n w_j f(hx_j)
\end{aligned}
$$

We have in mind that the quadrature rule $Q_h[f]$ is supposed to approximate $I_h[f]$. What we want to show now is that we can analyze the quality of that approximation just based on whether or not $Q_h[x^m] = I_h[x^m]$ for small values of m.

Suppose $Q_h[f]$ has degree d; that means that $Q_h[f]$ integrates polynomials of degree $\leq d$ exactly. Using Taylor's theorem with remainder, we can write

$$
f(x) = p(x) + \frac{f^{(d+1)}(\xi)}{(d+1)!} x^{d+1},
$$

where p is a degree d polynomial (the degree d Taylor approximation). Suppose $|f^{(d+1)}| < M$; then we have

$$
|I_h[f-p]| \leq \frac{M_d}{(d+2)!} h^{d+2} = O(h^{d+2})
$$

and

$$
|Q_h[f-p]| \leq \sum_{j=1}^n |w_j| \frac{M_d}{(d+1)!} h^{d+2} = O(h^{d+2}).
$$

Therefore

$$
\begin{aligned}
|I_h[f] - Q_h[f]| &= |I_h[f-p] - Q_h[p-f]| \\
&\leq |I_h[f-p]| + |Q_h[f-p]| = O(h^{d+2}).
\end{aligned}
$$

This tells us that the local truncation error (the error per panel) of a degree d integration rule is $O(h^{d+2})$; in a composite rule where there are $O(h^{-1})$ panels, we have a total error of $O(h^{d+1})$.

Week 12: Wednesday, Apr 18

Adaptive error control

Last time, we discussed Simpson's rule for quadrature:

$$I[f] = \int_a^b f(x)\,dx \approx \frac{b-a}{6}\left(f(a) + 4f(c) + f(b)\right), \quad c \equiv \frac{a+b}{2}$$

Simpson's rule has a local error of $O(h^5)$ where h is the size of one panel[1], and the composite rule has an error of $O(h^4)$. Let $S(a,b)$ denote the Simpson's rule estimate for the panel from a to b. Then we know that

$$\begin{aligned} I[f] &= S(a,b) + Ch^5 + O(h^7) \\ I[f] &= S(a,c) + S(c,b) + 2C(h/2)^5 + O(h^7), \end{aligned}$$

Combining these estimates, we have that the error in the two-panel rule is approximately

$$E(a,b) = \frac{S(a,c) + S(c,b) - S(a,b)}{15}.$$

One way to take advantage of this error estimate is by using extrapolation to increase the degree of our method. If we wanted, we could keep continually uniformly subdivide the mesh on which we have sampled the function in order to get ever higher-degree quadrature rules; this is sometimes called *Romberg integration*. Another way to use the error estimate, though, is to *adaptively* refine our mesh. That is, we keep a running tally of the estimated error on each panel, and any panel that seems to contribute too much error gets subdivided. There are multiple ways in which to decide the order in which one should process the panels that need to be subdivided. The most elegant version might be a priority queue (subdivide the panel with the highest error estimate first), but there are also simpler recursive variants that try to keep subdividing until a leaf panel of size h has estimated error of at most ηh. MATLAB's `quad` uses an adaptive Simpson's rule, but I'm not sure which refinement strategy it uses.

Note that while in principle $E(a,b)$ is an error estimate for $S(a,c)+S(c,b)$, in practice we use this as an error estimate for the purposes of things like

[1]For this lecture, h is the size of a panel, and not $2h$.

adaptive refinement... but we also return the extrapolated estimate $S(a, c) + S(c, b) + E(a, b)$, even though $E(a, b)$ is not technically an error estimate for the extrapolated rule. We like to try to have our cake and eat it to.

Raising the degree

An interpolatory quadrature rule through n points has degree $n - 1$, and so yields (total) error that decreases at least like $O(h^n)$, assuming that the function in question is sufficiently smooth. In some cases, though, we know that we get lucky and do even better. For example, the midpoint rule ($n = 1$) has degree 2, and Simpson's rule ($n = 3$) has degree 4. Why is this the true?

For convenience, let us consider a quadrature rule on $[-1, 1]$. A quadrature rule with n points has degree $n+s$ for $s \geq 0$, that means it computes any polynomial of degree up to $n + s$ exactly. In particular, if $x_1, \ldots, x_n$ are the nodes, we can define the degree n polynomial $q(x) = (x-x_1) \ldots (x-x_n)$, and our rule should be able to integrate $q(x)x^j$ exactly for $0 \leq j \leq s$. But notice that $q(x)x^j$ is exactly zero at each of the quadrature nodes, so the quadrature rule returns exactly zero at each of these points. Therefore, the quadrature rule can have degree $n + s$ for $s \geq 0$ only if it satisfies the conditions

$$\int_{-1}^{1} q(x)x^j \, dx = 0, \quad 0 \leq j \leq s.$$

This says that with respect to the standard inner product for functions on $[-1, 1]$, the polynomial q should be *orthogonal* to x^j for $0 \leq j \leq s$. Note that we must have $s < n$, since otherwise we would have that $\int_{-1}^{1} q(x)^2 \, dx$ was zero.

As it happens, the *Legendre polynomials* $P_k(x)$ satisfy the property that $P_0(x), \ldots, P_d(x)$ forms an orthogonal basis (with respect to the standard inner product on $[-1, 1]$) for the degree d polynomials. The first few Legendre polynomials are

$$\begin{aligned} P_0(x) &= 1 \\ P_1(x) &= x \\ P_2(x) &= (3x^2 - 1)/2, \end{aligned}$$

and we can compute higher-order Legendre polynomials by a recurrence:

$$(k + 1)P_{k+1}(x) = (2k + 1)xP_k(x) - kP_{k-1}(x).$$

Interpolatory quadrature rules based on interpolation through the zeros of Legendre polynomials are *Gauss-Legendre* quadrature rules. The midpoint rule is the lowest-order such rule; the second rule is

$$\int_{-1}^{1} f(x)\,dx \approx f(-\sqrt{1/3}) + f(\sqrt{1/3}).$$

In general, n-point Gauss-Legendre quadrature rules have degree $2n-1$; the two-point Gauss-Legendre rule has degree 3, for example.

There are a few variants on the Gaussian integration theme. One involves constraining the nodes for computational convenience. For example, if we insist that the interval endpoints must be quadrature nodes, we arrive at the Gauss-Lobatto rules (degree $2n-3$). The Gauss-Kronrod rules involve a pair consisting of an n-point Gauss quadrature rule together with a $2n+1$ point rule that re-uses the Gauss quadrature nodes; these rules are popular for adaptive quadrature, since the Gauss rule and the Kronrod rule can be compared in order to get an error estimate.

High order vs adaptivity

The default MATLAB `quad` routine does not use Gauss-Kronrod quadrature rules (though `quadgk` does). Instead, it uses an adaptive Simpson's rule. There's a good reason for this: *high order convergence is only achieved for functions with lots of derivatives.* If a function has a discontinuity, it will be hard to get better than $O(h)$ global error; if there is a discontinuous derivative, it is hard to beat $O(h^2)$ error; and so on. Methods based on Gauss quadrature or Chebyshev interpolation (the Clenshaw-Curtis methods) converge ferociously quickly for smooth integrands. But simple adaptive methods based on Simpson's rule often converge fast enough for most purposes, while remaining remarkably robust to nonexistent – or just very large – higher order derivatives.

Special behaviors

Consider the integral

$$I(z) = \int_{-z}^{z} \frac{\cos(x)}{\sqrt{|x|}}\,dx$$

How could we compute $I(\pi/2)$ numerically?

First, note that the function is even, so we can compute

$$I(z) = 2 \int_0^z x^{-1/2} \cos(x) \, dx.$$

This function is now awkward because the integrand diverges at $x = 0$. We can deal with this in a few different ways:

1. **Subtracting off the singularity**: Write

   $$I(z) = 2 \left[\int_0^z x^{-1/2} \, dx + \int_0^z x^{-1/2} \left(\cos(x) - 1 \right) \, dx \right].$$

   The first term can be handled analytically:

   $$\int_0^z x^{-1/2} \, dx = 2\sqrt{z}.$$

   The second term has a removable singularity at the origin ($O(x^{3/2})$ as $x \rightarrow 0$), and we can treat the integrand as zero at that point.

2. **Integration by parts**: If we integrate by parts, we have

   $$\int_0^z x^{-1/2} \cos(x) \, dx = 2\sqrt{z} \cos(z) + \int_0^z 2\sqrt{x} \sin(x) \, dx$$

3. **Change of variables**: If we let $t^2 = x$, then we can use the change of variables formula to recast the integral with as

   $$I(z) = 2 \int_0^{z^2} t^{-1} \cos(t^2)(2t \, dt) = 4 \int_0^{z^2} \cos(t^2) dt$$

   As it happens, we could at this point declare victory, since this integral is closely related to the Fresnel integral

   $$C(x) = \int_0^x \cos(\pi t^2/2) \, dt,$$

   and there are libraries that will compute Fresnel integrals for you.

4. **Special quadrature rules**: The *Gauss-Jacobi* family of quadrature rules approximates integrals of the form

$$\int_{-1}^{1} f(x)(1-x)^{\alpha}(1+x)^{\beta}\,dx$$

If we set $\alpha = 0$, and $\beta = -1/2$, then this gives us a rule for computing something with an inverse-square-root singularity at $x = -1$. If we apply the change of variables

$$y = \frac{z}{2}(1+x),$$

we have

$$\int_{-1}^{1} f(y(x))(1+x)^{-1/2}\,dx = \left(\frac{2}{z}\right)^{1/2} \int_{0}^{z} y^{-1/2} f(y)\,dy,$$

so we have a quadrature rule that deals with integrals with this sort of singularity. Let $f(y) = \cos(y)$ and we're all set.

These are most ot the tricks I know for dealing with integrals with singularities or unbounded domains. Fortunately, these tricks tend to work well for a variety of problems.

Problems to ponder

1. How would you write an n-panel composite Simpson rule in MATLAB *without* any repeated function evaluations at the same point.
2. Show that running extrapolation on the trapezoidal rule results in Simpson's rule.
3. How might you numerically compute

$$\int_{0}^{\infty} \ln(x)\exp(-x)\,dx$$

to a relative error tolerance of around 10^{-6}, ideally without too many function evaluations?

4. Consider the integral $\int_0^z x^{-1/2} \cos(x)\, dx$ from our "subtracting the singularity" example. How many derivatives does the integrand have at zero? How does this compare to the original integral?

5. Adaptive quadrature methods can be tricked! How would you find a polynomial such that one-panel or two-panel Simpson quadrature on $[-1, 1]$ both return zero, even though the true integal is positive?

6. Suppose $f(x)$ is convex, i.e. $f''(x) \geq 0$ everywhere. Show that in this case, the composite midpoint rule underestimates the true integral.

7. Describe a strategy for devising Simpson-like rules for integrals of the form
$$\int_a^b x^\alpha f(x)\, dx$$
where $\alpha > -1$ is given and $f(x)$ is assumed to be smooth. Your rule should sample the function at a, b, and at some point in between – how would you choose the location of that point to get the highest possible order of accuracy?

   Note: it's fine to express the coefficients in this rule in terms of integrals that you know how to work out symbolically, even if you don't want to run through the algebra.

Week 13: Monday, Apr 23

Ordinary differential equations

Consider ordinary differential equations of the form

$$y' = f(t, y) \tag{1}$$

together with the initial condition $y(0) = y_0$. These are equivalent to integral equations of the form

$$y(t) = y_0 + \int_0^t f(s, y(s))\, ds. \tag{2}$$

Only the simplest sorts of differential equations can be solved in closed form, so our goal for the immediate future is to try to solve these ODEs numerically.

For the most part, we will focus on equations of the form (1), even though many equations of interest are not posed in this form. We do this, in part, because we can always convert higher-order differential to first-order form by adding variables. For example,

$$my'' + by' + ky = f(t)$$

becomes

$$\begin{bmatrix} y \\ v \end{bmatrix}' = \begin{bmatrix} v \\ f(t) - \frac{b}{m}v - \frac{k}{m}y \end{bmatrix}.$$

Thus, while there are methods that work directly with higher-order differential equations (and these are sometimes preferable), we can always solve high-order equations by first converting them to first-order form by introducing auxiliary variables. We can also always convert equations of the form

$$y' = f(t, y)$$

into *autonomous systems* of the form

$$u' = g(u)$$

by defining an auxiliary equation for the evolution of time:

$$u = \begin{bmatrix} y \\ t \end{bmatrix}, \quad g(u) = \begin{bmatrix} f(t, y) \\ 1 \end{bmatrix}.$$

Basic methods

Maybe the simplest numerical method for solving ordinary differential equations is *Euler's method*. Euler's method can be written in terms of finite difference approximation of the derivative, Hermite interpolation, Taylor series, numerical quadrature using the left-hand rule, or the method of undetermined coefficients. For the moment, we will start with a derivation by Taylor series. If $y(t)$ is known and $y' = f(t, y)$, then

$$y(t+h) = y(t) + hf(t, y(t)) + O(h^2)$$

Now, drop the $O(h^2)$ term to get a recurrence for $y_k \approx y(t_k)$:

$$y_{k+1} = y_k + h_k f(t_k, y_k)$$

where $t_{k+1} = t_k + h_k$.

We can derive another method based on first-order Taylor expansion about the point $t + h$ rather than t:

$$y(t) = y(t+h) - hf(t+h, y(t+h)) + O(h^2).$$

If we drop the $O(h^2)$ term, we get the approximation $y(t_k) = y_k$ where

$$y_{k+1} = y_k + hf(t_{k+1}, y_{k+1}).$$

This is the *backward Euler* method, sometimes also called implicit Euler. Notice that in the backward Euler step, the unknown y_{k+1} appears on both sides of the equations, and in general we will need a nonlinear equation solver to take a step.

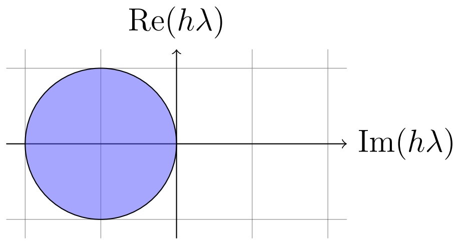
Another basic method is the *trapezoidal rule*. Let's think about this one by quadrature. If we apply the trapezoidal rule to (2), we have

$$\begin{aligned} y(t+h) &= y(t) + \int_t^{t+h} f(s, y(s))\, ds \\ &= y(t) + \frac{h}{2}\left(f(t, y(t)) + f(t+h, y(t+h))\right) + O(h^3) \end{aligned}$$

If we drop the $O(h^3)$ term, we have

$$y_{k+1} = y_k + \frac{h}{2}\left(f(t_k, y_k) + f(t_{k+1}, y_{k+1})\right).$$

Like the backward Euler rule, the trapezoidal rule is implicit: in order to compute y_{k+1}, we have to solve a nonlinear equation.



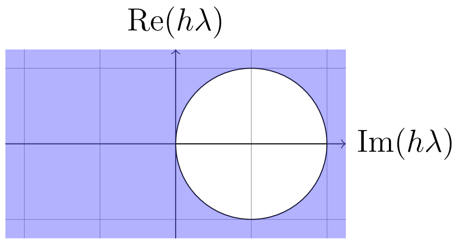


Figure 1: Regions of absolute stability for Euler (left) and backward Euler (right). For values of $h\lambda$ in the colored region, the numerical method produces decaying solutions to the test problem $y' = \lambda y$.

Stability regions

Consider what happens when we apply Euler and backward Euler to a simple linear test problem

$$y' = \lambda y$$

with a fixed step size h. Note that the solutions $y(t) = y_0 \exp(\lambda t)$ decay to zero whenever $\mathrm{Re}(\lambda) < 0$. This is a qualitative property we might like our numerical methods to reproduce. Euler's method yields

$$y_{k+1} = (1 + h\lambda)y_k,$$

which gives decaying solutions only when $|1 + h\lambda| < 1$. The set of values $h\lambda$ where Euler produces a decaying solution is called the *region of absolute stability* for the method. This region is shown in Figure 1.

Backward Euler produces the iterates

$$y_{k+1} = (1 - h\lambda)^{-1} y_k$$

Therefore, the discrete solutions decay whenever $|(1 - h\lambda)^{-1}| < 1$ — or, equivalently, whenever $|1 - h\lambda| > 1$. Thus, the region of absolute stability includes the entire left half plane $\mathrm{Re}(\lambda) < 0$ (see Figure 1), and so backward Euler produces a decaying solution when $\mathrm{Re}(\lambda) < 0$, no matter how large or small h is. This property is known as A-stability.

Euler and trapezoidal rules

So far, we have introduced three methods for solving ordinary differential equations: forward Euler, backward Euler, and the trapezoidal rule:

$$
\begin{aligned}
y_{n+1} &= y_n + hf(t_n, y_n) && \text{Euler} \\
y_{n+1} &= y_n + hf(t_{n+1}, y_{n+1}) && \text{Backward Euler} \\
y_{n+1} &= y_n + \frac{h}{2}\left(f(t_n, y_n) + f(t_{n+1}, y_{n+1})\right) && \text{Trapezoidal}
\end{aligned}
$$

Each of these methods is *consistent* with the ordinary differential equation

$$y' = f(t, y).$$

That is, if we plug solutions to the exact equation into the numerical method, we get a small *local error.* For example, for forward Euler we have *consistency of order 1*,

$$\mathcal{N}_h y_h(t_{n+1}) \equiv \frac{y(t_{n+1}) - y(t_n)}{h_n} - f(t_n, y(t_n)) = O(h_n),$$

and for the trapezoidal rule we have second-order consistency

$$\mathcal{N}_h y_h(t_{n+1}) \equiv \frac{y(t_{n+1}) - y(t_n)}{h_n} - f(t_n, y(t_n)) = O(h_n^2).$$

Consistency + 0-stability = convergence

Each of the numerical methods we have described can be written in the form

$$\mathcal{N}_h y^h = 0,$$

where y^h denotes the numerical solution and $\mathcal{N}_h$ is a (nonlinear) difference operator. If the method is consistent of order p, then the true solution gives a small *residual* error as h goes to zero:

$$\mathcal{N}_h y = O(h^p).$$

As we have seen in the past, however, small *residual* error is not the same as small *forward* error. In order to establish convergence, therefore, we need one more property. Formally, a method is *zero-stable* if there are constants h_0 and K so for any mesh functions x^h and z^h on an interval $[0,T]$ with $h \leq h_0$,

$$\|d^h\|_\infty \equiv \|x^h - z^h\|_\infty \leq K \left\{ |x_0 - z_0| + \left\| \mathcal{N}_h x^h(t_j) - \mathcal{N}_h z^h(t_j) \right\|_\infty \right\}.$$

Zero stability essentially says that the difference operators $\mathcal{N}_h$ can't become ever more singular as $h \to 0$: they are invertible, and the inverse is bounded by K. If a method is consistent and zero stable, then the error at step n is

$$|y(t_n) - y^h(t_n)| = |e_n| \leq K \max_j |d_j| = O(h^p).$$

The proof is simply a substitution of y and y^h into the definition of zero stability. The only tricky part, in general, is to show that the method is zero stable. Let's at least do this for forward Euler, to see how it's done — but you certainly won't be required to describe the details of this calculation on an exam!

We assume without loss of generality that the system is autonomous ($y' = f(y)$). We also assume that f is *Lipschitz continuous*; that is, there is some L so that for any x and z,

$$|f(x) - f(z)| \leq L|x - y|.$$

It turns out that Lipschitz continuity of f plays an important rule not only in the numerical analysis of ODEs, but in the theory of existence and uniqueness of ODEs as well: if f is not Lipschitz, then there might not be a unique solution to the ODE. The standard example of this is $u' = 2\,\mathrm{sign}(u)\sqrt{|u|}$, which has solutions $u = \pm t^2$ that both satisfy the ODE with initial condition $u(0) = 0$.

We can rearrange our description of $\mathcal{N}_h$ to get

$$\begin{aligned} x_{n+1} &= x_n + hf(x_n) + h\mathcal{N}_h[x](t_n) \\ z_{n+1} &= z_n + hf(z_n) + h\mathcal{N}_h[z](t_n). \end{aligned}$$

Subtract the two equations and take absolute values to get

$$|x_{n+1} - z_{n+1}| \leq |x_n - z_n| + h|f(x_n) - f(z_n)| + h|\mathcal{N}_h[x](t_n) - \mathcal{N}_h[z](t_n)|$$

Define $d_n = |x_n - z_n|$ and $\theta = \max_j |\mathcal{N}_h[x](t_j) - \mathcal{N}_h[z](t_j)|$. Note that by Lipschitz continuity, $|f(x_n) - f(z_n)| < Ld_n$; therefore,

$$d_{n+1} \leq (1 + hL)d_n + h\theta.$$

Let's look at the first few steps of this recurrence inequality:

$$\begin{aligned}
d_1 &\leq (1+hL)d_0 + h\theta \\
d_2 &\leq (1+hL)^2 d_0 + [(1+hL)+1]\, h\theta \\
d_3 &\leq (1+hL)^3 d_0 + \left[(1+hL)^2 + (1+hL) + 1\right] h\theta
\end{aligned}$$

In general, we have

$$\begin{aligned}
d_n &\leq (1+hL)^n d_0 + \left[\sum_{j=0}^{n-1} (1+hL)^j\right] h\theta \\
&\leq (1+hL)^n d_0 + \left[\frac{(1+hL)^n - 1}{(1+hL)-1}\right] h\theta \\
&\leq (1+hL)^n d_0 + L^{-1}\left[(1+hL)^n - 1\right]\theta
\end{aligned}$$

Now note that

$$(1+hL)^n \leq \exp(Lnh) = \exp(L(t_n - t_0)) \leq \exp(LT),$$

where T is the length of the time interval we consider. Therefore,

$$d_n \leq \exp(LT)d_0 + \frac{\exp(LT)-1}{L} \max_j |\mathcal{N}_h[x](t_j) - \mathcal{N}_h[z](t_j)|.$$

While you need not remember the entire argument, there are a few points that you should take away from this exercise:

1. The basic analysis technique is the same one we used when talking about iterative methods for solving nonlinear equations: take two equations of the same form, subtract them, and write a recurrence for the size of the difference.

2. The Lipschitz continuity of f plays an important role. In particular, if LT is large, $\exp(LT)$ may be very inconvenient, enough so that we have to take very small time steps to get good error results according to our theory.

As it turns out, in practice we will usually give up on global accuracy bounds via analyzing Lipschitz constant. Instead, we will use the same sort of local error estimates that we described when talking about quadrature: look at the difference between two methods that are solving the same equation with different accuracy, and use the difference of the numerical methods as a proxy for the error. We will discuss this strategy — and more sophisticated Runge-Kutta and multistep methods — next lecture.

Week 13: Wednesday, Apr 25

The Runge-Kutta concept

Runge-Kutta methods evaluate $f(t, y)$ multiple times in order to get higher order accuracy. For example, the classical Runge-Kutta scheme is

$$\begin{aligned}
K_0 &= f(t_n, y_n) \\
K_1 &= f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2} K_0\right) \\
K_2 &= f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2} K_1\right) \\
K_3 &= f\left(t_n + h, y_n + hK_2\right) \\
y_{n+1} &= y_n + \frac{h}{6}\left(K_0 + 2K_1 + 2K_2 + K_3\right).
\end{aligned}$$

Note that if f is a function of time alone, this is simply Simpson's rule. This is no accident.

Runge-Kutta methods are frequently used in pairs where a high-order method and a lower-order method can be computed with the same evaluations. Perhaps the most popular such methods are the Fehlberg 4(5) and Dormand-Prince 4(5) pairs — the MATLAB code `ode45` uses the Dormand-Prince pair. The difference between the two methods is then used as an estimate of the *local* error in the lower-order method. If a local error estimate seems too large, it is natural to try again with a shorter step based on an asymptotic expansion of the error. This method of step control works well on many problems in practice, but it is not foolproof (as we will see in HW 7). For example, in some settings the adaptive error control may suggest a time step which is fine for local error, but terrible for stability.

Adaptive time stepping routines generally use tolerances for both absolute and relative errors. A time step is accepted if

$$|e_i| < \max\left(\text{rtol}_i |y_i|, \text{atol}_i\right)$$

where rtol_i and atol_i are the tolerances for the ith component of the solution vector. The error tolerances have default values (10^{-3} relative and 10^{-6} absolute), but in practice it may be a good idea to set the tolerances yourself.

In principle, comparing two methods gives us an error estimate only for the lower-order method. However, one often takes a step with the higher-order method (at least for non-stiff problems). This cheat works well in practice, but we use the dignified-sound name of *local extrapolation* to dodge awkward questions about its mathematical legitimacy.

There are a bewildering variety of Runge-Kutta methods. Some are explicit, others are implicit. Some preserve interesting structural properties. Some are based on equally-spaced interpolation points, others evaluate on Gauss-Legendre points. In some, the stages can be computed one at a time; in others, the stages all depend on each other. But these methods are beyond the scope of the current discussion.

Matlab's ode45

For most non-stiff problems, `ode45` is a good first choice of integrators. The basic calling sequence is

$$[\text{tout},\ \text{yout}] = \textbf{ode45}(\text{f, tspan, y0});$$

The function `f(t,y)` returns a column vector. On output, `tout` is a column vector of evaluation times and `yout` is a matrix of solution values (one per row). Usually, `tspan` has two entries: `tspan = [t0 tmax]`. However, we can also specify points where we want solution values. In general, the underlying ODE solver does not put time steps at each of these points; instead it fills in the values using polynomial interpolation (this is called *dense output*).

The `ode45` function takes an optional output called `opt` that contains a structure produced by `odeset`. Using `odeset`, we can set error tolerances, put bounds on the step size, indicate to the solver that certain components must be non-negative, look for special events, or request diagnostic output.

The multistep concept

The Runge-Kutta methods proceed from time t_n to time t_{n+1}, then stop looking at t_n. An alternative is to use not only the behavior at t_n, but also the behavior at previous times t_{n-1}, t_{n-2}, etc. Methods that do this are *multistep methods*. Most such methods are based on linear interpolation.

For non-stiff problems, the *Adams family* are the most popular multi-step methods. The k-step explicit Adams methods (or Adams-Bashforth methods) interpolate $f(t_j, y_j)$ at points $t_{n-k}, \ldots, t_n$ with a degree k polynomial $p(t)$. Then in order to estimate

$$y(t_{n+1}) = y(t_n) + \int_{t_n}^{t_{n+1}} f(s, y(s))\, ds,$$

one computes

$$y_{n+1} = y_n + \int_{t_n}^{t_{n+1}} p(s)\, ds.$$

The *implicit* Adams methods (or Adams-Moulton methods) also interpolate through the unknown point. Though they are not A-stable, Adams-Moulton methods have larger stability regions and smaller error constants than Adams-Bashforth methods. Often, the two are used together to form a predictor-corrector pair: predict with Adams-Bashforth, then correct to Adams-Moulton. Because these methods are typically used for non-stiff problems, fixed point iteration often provides an adequate corrector.

With multistep methods, we can adapt not only the time step, but also the order. Very high-order methods may be appropriate when the solution is smooth and we want to either minimize the number of time steps or to meet very strict accuracy requirements. The MATLAB routine `ode113` implements a variable-order Adams-Bashforth-Moulton predictor-corrector solver.

The Adams methods interpolate the function values f; the *backward differentiation formulas* (BDF) instead interpolate y. The next step y_{n+1} is chosen so that the polynomial interpolating (t_{n-k}, y_{n-k}) through (t_{n+1}, y_{n+1}) has derivative at t_{n+1} equal to $f(t_{n+1}, y_{n+1})$. MATLAB's solver `ode15s` uses a variable-order *numerical differentiation formula* (a close relative of BDF). The `ode15s` code would be a typical first choice for stiff problems.

Example: the van der Pol equation

The van der Pol oscillator is a model nonlinear differential equation that shows up rather quickly in most discussions of the topic. The differential equation is:

$$x'' - \mu(1 - x^2)x' + x = 0.$$

When $\mu = 0$, this is a simple harmonic oscillator, and solutions have the form

$$x(t) = x(0)\cos(t) + x'(0)\sin(t)$$

When μ is nonzero, the picture gets slightly more complicated. You may or may not remember from a physics, calculus, or ODE class that the ODE

$$x'' + bx' + x = 0$$

has decaying oscillating solutions for $b > 0$ and exponentially growing solutions for $b < 0$. The coefficient b is interpreted as damping (with $b < 0$ corresponding to "anti-damping" behavior where solutions gain energy over time). In the case of the van der Pol equation, b is replaced by a nonlinear term which is negative when $|x| < 1$ and positive when $|x| > 1$. So the effective behavior, seen in practice, is that there is a balance between growth behavior for small x and decay behavior for larger x. The result is that the solution bounces back and forth between slow motions for $x > 1$ and $x < -1$ with fast transitions in between, and the speed of those transitions is governed by the magnitude of μ.

Now let's write code to actually solve the system. The ODE solvers in MATLAB require that we express our problem as a first-order system in standard form, which we do by introducing the auxiliary variable $y = x'$:

$$\begin{bmatrix} x \\ y \end{bmatrix}' = \begin{bmatrix} y \\ \mu(1-x^2)y - x \end{bmatrix} = f_{vdp}(x, y, \mu).$$

This is a demo system in the MATLAB documentation, so MATLAB already has a function `vanderpoldemo(x,y,mu)` to evaluate the right hand side f_{vdp} of this system. Based on the advice given in lecture, we should choose `ode45` if the problem is non-stiff (μ modest) and `ode15s` if the problem is stiff (μ large). Our script, `runvdp` will compute the solution using both methods and compare both the results and the timings. For $\mu = 1$, we find that both solvers perform adequately; for $\mu = 100$, `ode45` takes 5 seconds and 26368 steps while `ode15s` takes 0.45 seconds and 761 steps. For $\mu = 200$, `ode45` takes 20 seconds and over 104868 steps, while `ode 15s` takes 0.50 seconds. and just over 1010 steps. Both solvers return results that are nearly indistinguishable visually (see Figure 1).

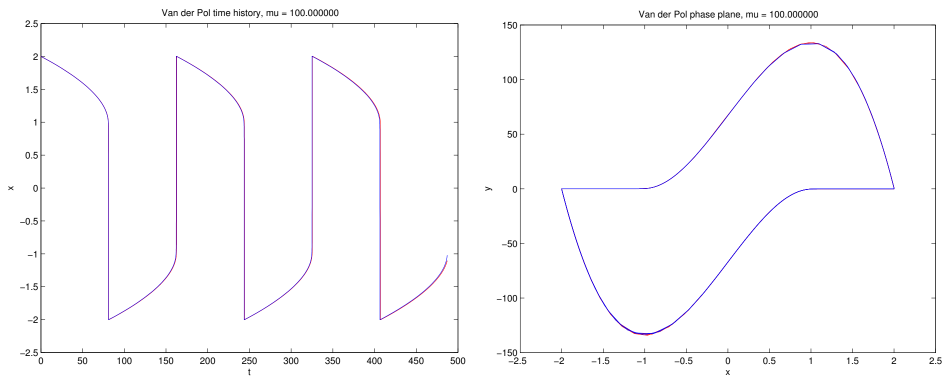


Figure 1: Van der Pol solutions for $\mu = 100$, via `ode45` (red) and `ode15s` (blue). The left plot shows x vs t; right shows $x(t)$ vs $y(t) = x'(t)$.

Problems to ponder

1. Describe how to use `ode45` and `plot` to display the solution to an initial value problem

$$mx'' + bx' + kx = \begin{cases} 1, & 0 \leq t \leq 1 \\ 0, & 1 \leq t \leq t_{\text{final}} \end{cases}$$

   where $x(0) = x'(0) = 0$.

2. For implicit methods like backward Euler or the trapezoidal rule, we need to solve a nonlinear equation at each update. For backward Euler, for example, we have

$$y_{n+1} - hf(y_{n+1}) - y_n = 0.$$

   What is Newton's iteration for computing y_{n+1} given y_n?

3. For a *non-stiff* problem where f' is not too large, note that we could also use fixed point iteration:

$$y_{n+1}^{\text{new}} = y_n + hf(y_{n+1}^{\text{old}}).$$

   Assuming f is Lipschitz with constant L, show this fixed point iteration converges when $Lh < 1$.

```
% runvdp(mu)
%   Demonstrate relative performance of ode45 and ode15s for
%   the van der Pol oscillator as a function of mu.
%
function runvdp(mu)

  tau = (3-2*log(2))*mu + 4.676*mu^(-1/3); % Estimated period
  tspan = [0, 3*tau];                      % Go about 3 cycles
  y0 = [2; 0];                             % Initial conditions
  opt = odeset('Stats', 'on');             % Print diagnostics
  ode = @(t,y) vanderpoldemo(t,y,mu);      % MATLAB already has f_vdp

  fprintf('\n--- ODE45 solve ---\n');
  tic; [tn,yn] = ode45(ode, tspan, y0, opt); toc

  fprintf('\n--- ODE15s solve ---\n');
  tic; [ts,ys] = ode15s(ode, tspan, y0, opt); toc

  figure(1);
  plot(tn,yn(:,1), 'r-', ts,ys(:,1), 'b-');
  xlabel('t');
  ylabel('x');
  title(sprintf('Van der Pol time history, mu = %f', mu));

  figure(2);
  plot(yn(:,1), yn(:,2), 'r-', ys(:,1), ys(:,2), 'b-');
  xlabel('x');
  ylabel('y');
  title(sprintf('Van der Pol phase plane, mu = %f', mu));
```

Figure 2: Script to compare `ode45` and `ode15s` for the van der Pol oscillator.

4. Suppose we have used a time-stepping algorithm to compute $y_k \approx y(t_k)$. To get from step k to $k+1$, consider using either one or two steps of forward Euler:
$$\begin{aligned} y_{k+1} &= y_k + hf(y_k) \\ z_{k+1/2} &= y_k + \frac{h}{2} f(y_k) \\ z_{k+1} &= z_{k+1/2} + \frac{h}{2} f(z_{k+1}) \end{aligned}$$
Write the first term in a Taylor expansion for the local error $y_{k+1} - u(t_{k+1})$, where u is the solution to the initial value problem
$$u'(t) = f(u(t)), \quad u(t_k) = y_k$$
How could you combine z_{k+1} and y_{k+1} to estimate this local error?

5. Consider the test equation
$$y' = \lambda y.$$
Suppose we approximate the solution at time $t_k = kh$ using forward Euler. Show that if $\hat{y}_k$ is the kth step of the forward Euler method, then $\hat{y}_k = z(t_k)$ where $z(t)$ is the solution to the modified differential equation
$$z' = \hat{\lambda} z.$$
How is $\hat{\lambda}$ related to λ? Repeat the exercise for backward Euler.

6. Based on the description in these notes, derive the backward differentiation formula based on interpolation of y at points t_n and t_{n+1}.

Week 14: Monday, Apr 30

Introduction

So far, our discussion of ODE solvers has been rather abstract. We've talked some about how to evaluate ODE solvers, how ODE solvers choose time steps in order to control error, and the different classes of ODE solvers that are available in MATLAB. We have not, however, tackled any concrete example problems other than trivial linear test problems. In part, this is because I have a hard time writing solutions and drawing believable pictures at the board for anything but trivial test problems. So today, let's try a MATLAB-oriented lecture.

1 A ballistics problem

The next problem is a classic of both scientific computing and certain classes of computer games: ballistics calculations. We will solve this problem via a *shooting* method: that is, we will base our solution method on initial value problem solvers, and we try to choose initial conditions in order to satisfy the problem constraints.

1.1 Model without air drag

Let's start with a simple model, one that many of you have probably seen in an introductory physics class. A projectile is fired from a launcher with fixed speed; as a function of the launch angle, where will it hit the ground? In the simplest version of this model, the only force acting on the ball after launch is gravity, so Newton's law tells us

$$m\mathbf{a} = -mg\mathbf{e}_y$$

where $\mathbf{e}_y$ is a unit vector in the vertical direction, m is the particle mass, and $\mathbf{a} = \mathbf{x}'' = (x'', y'')$ is the acceleration vector. If our launcher is positioned at the origin, then the initial conditions for a launch with speed s at angle θ are

$$\begin{aligned} x(0) &= 0, & x'(0) &= s\cos(\theta) = v_{0,x}, \\ y(0) &= 0, & y'(0) &= s\sin(\theta) = v_{0,y}. \end{aligned}$$

Subject to these initial conditions, we can compute the solution analytically:

$$\begin{aligned} x(t) &= v_{0,x} t \\ y(t) &= v_{0,y} t - gt^2/2. \end{aligned}$$

The trajectory returns to the ground at time $t_{\mathrm{final}} = 2v_{0,y}/g$ and at position

$$x_{\mathrm{final}} = x(t_{\mathrm{final}}) = \frac{2}{g} v_{0,y} v_{0,x} = \frac{s^2}{g} \sin(2\theta).$$

Therefore, we can reach a target at distances $d \leq s^2/g$ with launch angles θ that satisfy $\sin(2\theta) = gd/s^2$. In general, if we can hit the target at all there will be two trajectories that work. One will have angle between 0 and $\pi/4$, while the other has an angle between $\pi/4$ and $\pi/2$.

1.2 Model with air drag

In practice, projectiles are affected not only by gravity, but also by air resistance. For a reasonable range of projectiles, and assuming that the projectile does not go so high that changes in atmospheric pressure are an issue, the drag force due to air resistance acts in the direction opposite the velocity, with a magnitude proportional to the square of the velocity. That is,

$$m\mathbf{a} = -mg\mathbf{e}_y - mc\|\mathbf{v}\|\mathbf{v}.$$

If we also consider a constant horizontal wind velocity $w\mathbf{e}_x$, we arrive at

$$m\mathbf{a} = -mg\mathbf{e}_y - mc\|\mathbf{v} - w\mathbf{e}_x\|(\mathbf{v} - w\mathbf{e}_x).$$

The coefficient c is a complicated function of the size, shape, and mass of the projectile, along with the temperature and pressure of the air. For the moment, we will suppose it is simply given.

The differential equation without air drag was simple enough for us to analyze by hand. This model is harder, so we turn to numerical methods. To use MATLAB's ODE solvers, we need to put the model into first-order form:

$$\begin{bmatrix} \mathbf{x} \\ \mathbf{v} \end{bmatrix}' = \begin{bmatrix} \mathbf{v} \\ -g\mathbf{e}_y - c\|\mathbf{v} - w\mathbf{e}_x\|(\mathbf{v} - w\mathbf{e}_x) \end{bmatrix} = f_{\mathrm{ballistics}}(\mathbf{x}, \mathbf{v}, g, c, w).$$

We code this model in a MATLAB script `fball` (Figure 1).

When setting up an ODE model, there are usually ample opportunities to make nonphysical assumptions or errors in mathematics or programming. For this reason, it is good practice to sanity-check our computations. In our case, there are two natural checks:

1. If the coefficient c is zero, this problem simplifies to the model we discussed in the last section. Our numerical ODE solver should therefore recover the same solution we found in our hand analysis.

2. If the coefficient c is positive, then air drag slows down the projectile, so it should not go as far as it would go in the case of $c = 0$.

We check both of these behaviors with the script `testball1` (Figure 3). A visual comparison of trajectories computed with and without air drag is showin in Figure 2.

1.3 Computing points of impact

Now that we have a code that we believe gives plausible trajectories, we need to figure out where those trajectories hit the ground as a function of the launch angle. In order to do this, we want not to stop the simulation at a specific *time*, but when a specific *event* occurs: namely, when the vertical component of the projectile position tries to go from positive to negative. MATLAB provides *event detection* as part of the ODE suite: we define an interesting event in terms of a zero crossing of some test function of the state vector (position and velocity), and we do something special when at points when the test function goes from positive to negative or vice versa. In our case, we are interested in when the y position component of the solution goes from positive to negative values, and we would like to terminate the computation when that occurs. Then we want to extract the final x position. This computation is done in `ftarget` using a test function `hitground` to detect the impact event (Figure 4).

1.4 Computing targeting solutions

At this point, we are interested in the function $f_{\text{target}}(\theta)$ that computes the impact distance as a function of the launch angle. For the case when $c = 0$, we know that this function is proportional to $\sin(2\theta)$. The case $c > 0$ is a little more complicated, but even in this case, $f_{\text{target}}(\theta)$ is very smooth. We

```
% yp = fball(t,y,opt)
%   Compute the right hand side of an ODE for projectile motion
%   in the presence of wind and air drag.  The opt structure should
%   describe a scaled air drag coefficient (c), the gravitational field (g),
%   and the horizontal wind speed (w).
%
function yp = fball(t,y,opt)

  g = opt.g;   % Gravitational field
  c = opt.c;   % Scaled drag coefficient
  w = opt.w;   % Horizontal wind speed

  % Unpack position and velocity
  x = y(1:2);
  v = y(3:4);

  % Velocity relative to wind
  vv     = v;
  vv(1) = vv(1)-w;

  % Compute acceleration
  a      = -c*norm(vv)*vv;
  a(2)  = a(2)-g;

  % Return
  yp = [v; a];
```

Figure 1: Right-hand side for ballistics model.

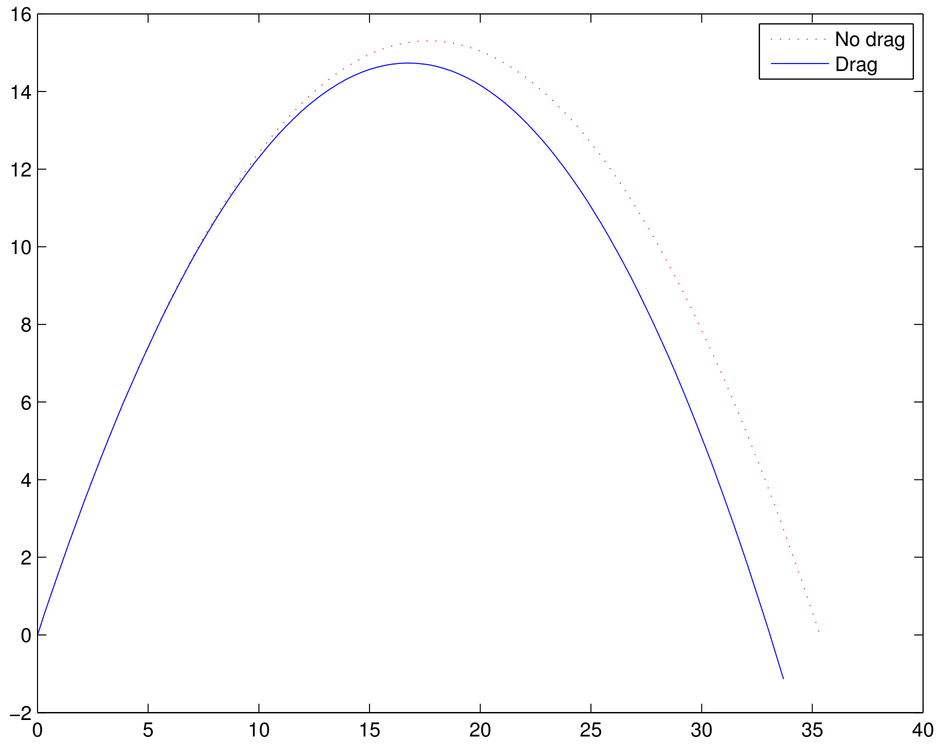


Figure 2: Trajectories with and without drag.

```
% -- Sanity check trajectories computed with and without drag --

theta = pi/3;
v0 = s*[cos(theta); sin(theta)];

% Compute the reference trajectory (absent air resistance)
%   x(y) = s*cos(theta)*t
%   y(t) = -g*t^2/2 + s*sin(theta)*t
% up to time tfinal = 2*s*sin(theta)/g
%
 tfinal = 2*v0(2)/g;
 tref = linspace(0,tfinal)';
xref = v0(1)*tref;
yref = (v0(2)-g/2*tref).*tref;

% Compute the same reference trajectory with ode45
y0 = [0; 0; v0];
refopt = opt;
refopt.c = 0;
[tout,yout] = ode45(@(t,y) fball(t,y,refopt), tref, y0);

% Compute a similar trajectory with air drag on (no wind)
dopt = opt;
dopt.w = 0;
[toutd,youtd] = ode45(@(t,y) fball(t,y,dopt), tref, y0);

% Do a comparison between analytical and numerical solutions (no drag)
fprintf('Max x error: %g\n', norm( xref-yout(:,1), inf));
fprintf('Max y error: %g\n', norm( yref-yout(:,2), inf));

% Visually compare solutions for drag and no drag
plot(xref, yref, 'r:', youtd(:,1), youtd(:,2), 'b-');
legend('No drag', 'Drag');
```

Figure 3: Test script to check ballistics ODE.

```
% xfinal = ftarget(thetas, opt)
%   Compute impact points as a function of angles for the  ballistics
%   ODE with parameters given in opt. In addition to the basic ODE
%   parameters, opt.s should be set to the launch speed.
%
function xfinal = ftarget(thetas, opt)

  xfinal = 0*thetas;
  for j = 1:length(thetas)
    theta = thetas(j);
    v0 = (opt.s)*[cos(theta); sin(theta)];
    y0 = [0; 0; v0];
    tfinal = 2*v0(2)/opt.g;
    odeopt = odeset('Events', @hitground);
    [tout,yout] = ode45(@(t,y) fball(t,y,opt), [0 tfinal], y0, odeopt);
    xfinal(j) = yout(end,1);
  end

function [value,isterminal,direction] = hitground(t,y)

  value      = y(2);  % Check for zero crossings of y position
  isterminal = 1;     % Terminate on a zero crossing
  direction  = -1;    % We only care about crossing y > 0 to y < 0
```

Figure 4: Compute point of impact as a function of θ.

could compute f_{target} and derivatives at arbitrary points based on the ODE [1], but the evaluation costs a little bit; if we included a few more complicating factors in our evaluations, we might reasonably be reluctant to do too many trajectory computations with `ftarget`. So let's use the tools that we build previously, and fit a polynomial approximation f_{target} at a Chebyshev grid on $[0, \pi/2]$.

The function $f_{\text{target}}(\theta)$ in general is a unimodal function: it increases on $[0, \theta_{\text{max}}]$ and then decreases on $[\theta_{\text{max}}, \pi/2]$. If we want to hit a target at distance d, then, there are two things that can happen:

1. If $d > f_{\text{target}}(\theta_{\text{max}})$ then we cannot hit the target at any angle.

2. If $d < f_{\text{target}}(\theta_{\text{max}})$, then there are generally two solutions to the equation $f_{\text{target}}(\theta) - d = 0$: one on the interval $[0, \theta_{\text{max}}]$ and the other on the interval $[\theta_{\text{max}}, \pi/2]$.

We can find the two solutions, if they exist, using MATLAB's `fzero` function (Figure 5). In each step, we work with a polynomial approximation to f_{target} rather than working with f_{target} directly.

1.5 An example trajectory

As an example, let's consider a concrete example with a high drag coefficient ($c = 0.05$ m^{-1}) and some wind ($w = -2.5$ m/s). We want to hit a target at distance $d = 10$ m. The trajectories computed for the two solution angles are shown in Figure 6; the residual error $f_{\text{target}}(\theta)$ is on the order of 10^{-7} for this problem, which is almost certainly smaller than errors due to uncertainty in the model parameters.

2 Particle in a box

In the previous example, we solved a two-point boundary value problem by shooting (the two points being the point of launch and of impact). In this

[1]The derivative of the trajectory with respect to the launch angle θ can be computed in terms of an extended ODE system, a so-called *variational equation*. Using this variational equation, we could compute derivatives of the impact location as a function of θ. But using a polynomial interpolant will turn out to be a simpler way of approximating the function and its derivatives.

```
% thetas = find_angle(d, opt)
%   Find angles to hit a target at distance d in the ballistics problem.
%   If the target is unreachable, give an error message.

function [thetas] = find_angle(d, opt)

  % Fit a Chebyshev polynomial to the targeting behavior
  [D,z] = cheb(20);
  thetac = (z+1)*pi/4;
  impactsc = 0*thetac;
  for k = 1:length(thetac)-1
    impactsc(k) = ftarget(thetac(k), opt);
  end

  % Find the farthest-traveling trajectory
  zcrit = chebopt(impactsc);
  topt  = chebeval(thetac,    zcrit );
  xopt  = chebeval(impactsc, zcrit );

  % If we fall below that point, quit
  if d > xopt, error('Target out of range'); end
  g = @(z) chebeval(impactsc, z)-d;
  zs = [ fzero(g, [-1,zcrit ]); fzero(g, [ zcrit ,1]) ];
  thetas = chebeval(thetac, zs );
```

Figure 5: Find targeting angles for a given distance.

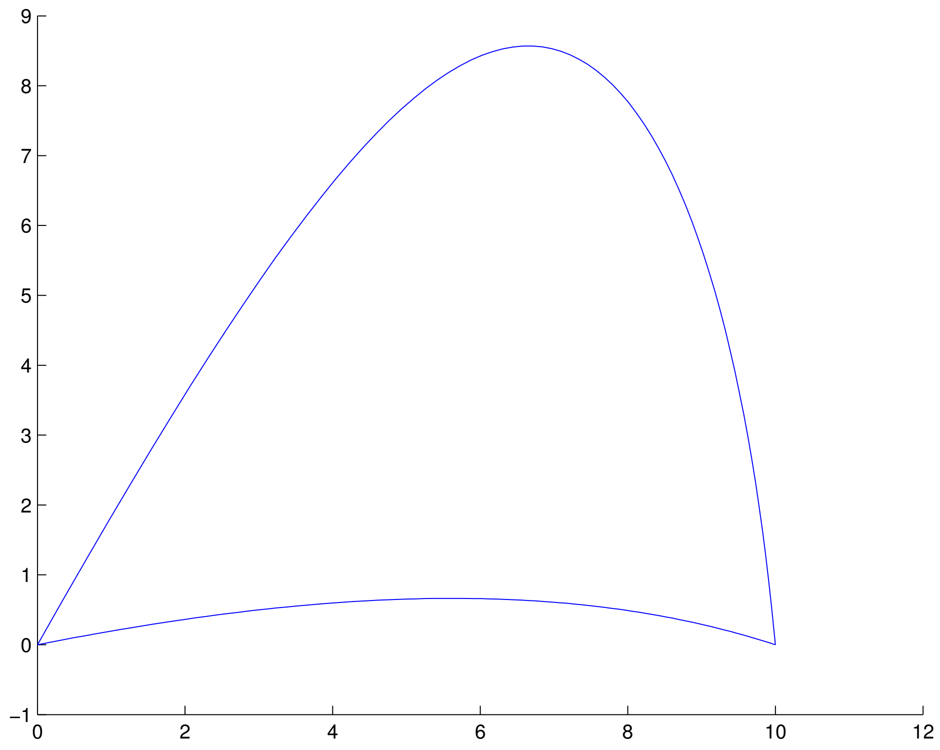


Figure 6: Sample targeting solutions with wind and a high drag coefficient.

example, we will use a finite difference method. The problem we want to solve is a classic example model in quantum mechanics: the particle in a box.

The basic equation of quantum mechanics is Schrödinger's equation; for problems with one space variable, this is

$$H\psi = \left(-\frac{\hbar}{2m} \frac{d^2}{dx^2} + V(x) \right) \psi = E\psi.$$

This is an eigenvalue problem: we want to find E such that the equation has a nontrivial solution, and there are only a discrete set of such E. For the duration of this discussion, let us assume $\hbar/2m = 1$. Now, suppose we have a potential energy $V(x)$ which is zero on $[0, 1]$ and ∞ elsewhere. The squared wave function ψ represents the probability that a particle at energy E is at any given position, and particles can't make it past the infinite energy barriers at 0 and 1, so we actually have a two-point boundary value problem:

$$\begin{aligned} \frac{d^2\psi}{dx^2} &= E\psi, \quad x \in (0, 1) \\ \psi(0) &= 0 \\ \psi(1) &= 0. \end{aligned}$$

A little thought shows the appropriate solutions are $\psi_k = \sin(k\pi x)$ and $E_k = (k\pi)^2$. Let's suppose we did not know that and look at a way to compute the solution numerically.

The standard second-order accurate approximation of a second derivative at a point x is

$$\psi''(x) \approx \frac{\psi(x-h) - \psi(x) + \psi(x+h)}{h^2}.$$

Now suppose we have a regular mesh of points $x_0 = 0$ to $x_{N+1} = 1$, with $x_j = jh$, $h = 1/(N + 1)$. Then we can compute

$$-\psi''(x_i) \approx \frac{-\psi(x_{i-1}) + 2\psi(x_i) - \psi(x_{i+1})}{h^2}$$

Therefore, we can compute an approximation $u_i \approx \psi(x_i)$ by approximating the differential equation at the interior points $x_1, \ldots, x_N$:

$$h^{-2} \left[-u_{i-1} + 2u_i - u_{i+1} \right] - Eu_i = 0.$$

At the end points, we use the boundary conditions $u_0 = u_{N+1} = 0$, which gives us the end conditions

$$\begin{aligned} h^{-2}\left[2u_1 - u_2\right] - Eu_1 = 0. \\ h^{-2}\left[-u_{N-1} + 2u_N\right] - Eu_N = 0. \end{aligned}$$

Putting everything together, we can write the discrete problem concisely as

$$(h^{-2}T - E)u = 0.$$

where T is the standard tridiagonal stencil

$$T = \begin{bmatrix} 2 & -1 & & & & \\ -1 & 2 & -1 & & & \\ & -1 & 2 & -1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & -1 & 2 & -1 \\ & & & & -1 & 2 \end{bmatrix}$$

Therefore, the eigenvalues and eigenvectors for the continuous problem can be approximated by eigenvalues and eigenvectors for a discrete problem. The function `particlebox` (Figure 7) computes the first four eigenvalues / energy levels using this approximation using a finite difference mesh with N interior points. Because we know the exact solutions in this problem, it is easy for us to assess the convergence of our code as a function of h; we do this with the script `particleboxcvg` (Figure 8). Using `particleboxcvg`, we can see that the smallest eigenvalue of the discrete problem estimates the continuous eigenvalue to accuracy $O(h^2)$.

```
% E = particlebox(N)
%   Compute the first four energy levels for the "particle in a box"
%   model using a finite-difference discretization for d^2/dx^2
%   with N interior mesh points.

function E = particlebox(N)

  % Points go 0 to N+1; 0 and N+1 satisfy BCs
  h = 1/(N+1);
  T = -diag(ones(N-1,1),-1) + 2*eye(N) - diag(ones(N-1,1),1);

  % Estimate eigenvalues and eigenvectors
  [V,D] = eig(T/h^2);
  E = diag(D);

  % Compute the first four energy levels
  E = E(1:4);
```

Figure 7: Finite difference computation of the first four energy levels for a particle in a box.

```
% Do a simple convergence study
N = 10;
h = []; E = [];
for j = 1:5
  h(j) = 1/(N+1);
  E(:,j) = particlebox(N);
  N = N*2;
end
loglog(h, pi^2-E(1,:));

% Rate of convergence
est_order = log( (pi^2-E(1,end-1))/(pi^2-E(1,end)) )/log(2);
fprintf('Estimated order of convergence: %f\n', est_order);
```

Figure 8: Convergence analysis of a finite difference computation of the first four energy levels for a particle in a box.

Week 14: Wednesday, May 2

Summary

Error analysis and floating point

You should know about relative vs absolute error, forward error, backward error, residual, and condition numbers. You should remember the $1+\delta$ model for rounding, and be able to apply it in simple situations. You should know what underflow, overflow, and cancellation are.

Example questions:

1. What are the forward error and residual error for approximating the larger root of $f(x) = x^2 - 2$ by $\hat{x} = 1.5$? What is the condition number for this problem?

2. Which is more suitable for computation near $x = 0$: $\cos(x) - 1$, or $\sin(x)/(1 + \cos(x))$? Why?

3. Suppose I wanted to sum up numbers $z_1, \ldots, z_n$. A standard approach would be to write a loop to compute successive partial sums.

```
s = 0;
for j = 1:N
  s = s + z(j);
end
```

   This loop really runs the recurrence $s_j = s_{j-1} + z_j$ starting at $s_0 = 0$. If I do this in floating point, how could I keep a running error estimate on the partial sums?

Linear algebra, linear systems, and least squares

You should know how to manipulate matrices and vectors in MATLAB, and have some notion of the relative costs of equivalent matrix expressions. You should know about the 1-norm, 2-norm, and infinity norm; about induced operator norms; and about the Frobenius norm. You should remember that the 2-norm (and the operator 2-norm and Frobenius norm) are invariant under orthogonal transformations, and that this can be used to simplify least squares problems. You should know the condition number for solving linear systems, and you should remember the factors that go into the sensitivity analysis for solving least squares problems (it's fine if you don't remember the exact formulas for the condition number in the latter case). You should know how to solve linear systems and least squares problems using MATLAB's backslash operator. You should know something about the normal equations, and about the relation between the solution, right hand side, and residual in a least squares problem. You should know the factorizations $PA = LU$, $A = QR$, and $A = U\Sigma V^T$. You should know what sparsity means.

Example questions:

1. Use norm bounds to show that the iteration $x_{j+1} = Ax_j + f$ converges if $\|A\| < 1$, and bound the magnitude of the limiting value.
2. Given $PA = LU$, write an $O(n^2)$ code fragment to compute $A^{-T}b$.
3. Describe a method to approximately minimize the sum of squared *componentwise relative errors* $d_j = (Ax - b)_j / b_j$.

Iterations, equation solving, and optimization

You should know about bisection, the Newton idea and its variants, and the concept of fixed point iteration. You should understand what is meant by rates of convergence (linear, superlinear, quadratic, etc). You should be able to reason about the error in iterations by subtracting a fixed point equation from the iteration equation; you should also be able to do Taylor series manipulations needed to understand these methods. For optimization, you should know what a descent direction is, and what it means to do a line search. You should know how to reason about stationary iterations for linear systems in terms of splittings. You should know that CG corresponds to minimization over a Krylov subspace, though you do not need to know any further details of the algorithm.

Example questions:

1. Suppose $f(x_*) = 0$ and we know $f'(x_*)$. Argue that the fixed point iteration
$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_*)}$$
converges quadratically to x_* when started close enough.

2. Suppose $|f'(x)| \leq \alpha < 1$ for any x. Show that if $x_* = f(x_*)$ is a fixed point, then the iteration $x_{k+1} = f(x_k)$ converges at least linearly to x_*.

3. If we know f is continuously differentiable and $f'(a)f'(b) < 1$, and we have a routine to compute f', describe how to find a minimum or maximum on (a, b).

4. Show that if $\|I - A\| < 1$, the iteration
$$x_{k+1} = x_k + (b - Ax_k)$$
converges to $A^{-1}b$.

Interpolation, differentiation, and integration

You should know three different approaches to polynomial interpolation: the Vandermonde approach (writing the polynomial in a power basis); the Lagrange approach; and the Newton approach. You should know how Newton divided differences relate to derivatives, and how this relation can be used to provide error bounds on how well a polynomial interpolant approximates a function (assuming bounds on derivatives of various orders). You should know the basic ideas of piecewise polynomial interpolation, though I did not emphasize this in class and will not emphasize it on the final.

You should know how to derive rules for numerical differentiation and integration by either artfully canceling out terms in Taylor expansions (the method of undetermined coefficients) or by differentiating and integrating an interpolating function. You should understand what is meant by the order of a quadrature rule. You should understand the basic ideas of Gaussian quadrature, and why their order of accuracy ($2n - 1$) is the greatest possible for any quadrature rule that involves function evaluations at n given points. You should also understand the ideas of Richardson extrapolation and of error estimates based on comparing different rules for differentiation and quadrature.

Example questions:

1. Write the interpolant through $f(0) = 1$, $f(1) = 2$, and $f(2) = 1$ in terms of the power basis, the Lagrange basis, and the Newton basis.

2. Suppose a_n and a_{2n} are two approximations to $\int_{-1}^{1} f(x)\,dx$ computed by the composite midpoint rule with n intervals and $2n$ intervals, respectively. Write an estimate of the error in a_{2n} based on comparing

the two methods. Your estimate should be asymptotically exact as $n \to \infty$.

3. Given a quadrature rule

$$I_h[f] = \sum_{j=1}^{n} w_j f(x_j) \approx \int_a^b f(x)\,dx,$$

give an example of a polynomial for which the quadrature rule cannot compute the true integral.

ODE solvers

You should know how to convert ODEs to standard first-order form for use with MATLAB's ODE solvers. You should be able to write something that makes use of MATLAB's ODE solvers (given a reminder of the basic calling sequence). You should know the formulas for forward and backward Euler, and the implicit trapezoidal rule. You should know the basic ideas of consistency and zero stability of ODE solvers, if not many details; and you should understand the basic idea of using pairs of methods for local error control (and how such local error control can still fail to yield good solutions in some cases). You should be able to reason about whether methods applied to the test problem $y' = \lambda y$ converge to zero for different values of $h\lambda$, and you should understand what is meant by the region of absolute stability for a method.

Example questions:

1. Describe how to solve the IVP $mu'' + bu' + ku = g(t)$, $u(0) = u_0$, $u'(0) = v_0$ using the MATLAB solver `ode45`. Remember that `ode45` has the calling sequence

```
[tout,yout] = ode45(f,tspan,y0);
```

where `f` is a function that takes arguments `t, y`.

2. Consider the ODE

$$\begin{bmatrix} x \\ y \end{bmatrix}' = \begin{bmatrix} y \\ -x \end{bmatrix}$$

with initial conditions $x(0) = 1$ and $y(0) = 0$. The true solution to this problem is $x(t) = \cos(t)$, $y(t) = \sin(t)$, and so $r(t)^2 \equiv x(t)^2 + y(t)^2$ is equal to 1. Show that if we discretize the problem using forward Euler with fixed step size h, then $r_n^2 \equiv x_n^2 + y_n^2 = (1+h^2)^n$.