Week 2: Wednesday, Feb 2

Logistics

1. Class for Wednesday, Feb 2, has been **cancelled**. These notes were my preliminary version of what I had intended to cover during this lecture. Though I will cover much the same material on Monday, I am posting these online in order to help you if you choose to start HW 2 a little early.
2. HW 1 should now be handed in during class on Monday, Feb 7.
3. HW 2 should is now due Monday, Feb 14.

Digression: A useful power series

This seems like a good place to mention a Taylor expansion that we will see more than once in the coming weeks. For $|x| < 1$, we can write an absolutely convergent series:

$$(1-x)^{-1} = \sum_{j=0}^{\infty} x^j.$$

As a concrete example of this, we have that if δ is around ϵ_{mach},

$$(1+\delta)^{-1} \approx 1 - \delta$$

The utility of this power series is not restricted to real numbers! When E is a "small" matrix, we have the matrix power series

$$(I-E)^{-1} = \sum_{j=0}^{\infty} E^j.$$

As in the scalar case, the linear approximation $(I+E)^{-1} \approx I - E$ is often useful. We will see this again when we talk about sensitivity of linear systems in the next couple lectures.

Matrices and vectors in Matlab[1]

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)
```

[1] This section should be review. If it looks completely unfamiliar, please see me.

[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."

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:

```
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.

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*.

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

[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.

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:

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

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

[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.

[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

```
% 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.

Of sizes and sensitivities

When we talked about rounding error analysis for scalar functions, I emphasized a couple points:

- I care about *relative* error more than *absolute* error.
- Some functions are *ill-conditioned*. These are hard to solve because a small relative error in the input can lead to a large relative error in the output.
- We like algorithms that are *backward stable*: that is, they return an answer which solves slightly the wrong problem. (small *backward error*). The *forward error* for a backward stable algorithm is bounded

than older versions, particularly when the loops have simple structures that the optimizer can figure out.

by the product of the backward error and the condition number, so a backward stable algorithm can return results with large forward error when the problem is ill-conditioned.

Now I want to build the same machinery for talking about problems in which the inputs and outputs are vectors.

Norm!

First, we need the concept of a *norm*, which is a measure of the length of a vector. 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):

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

In general, 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: $\|x + y\| \leq \|x\| + \|y\|$.

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:

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

Question: Why is the second equality true?

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!

Absolute error, relative error, conditioning

Suppose I want to compute $y = Ax$, where A is a square matrix, but I don't know the true value of x. Instead, I only know $\hat{x} = x + e$ and some bound on $\|e\|$. What can I say about the error in $\hat{y} = A\hat{x}$ as an approximation to y? I know the absolute error in $\hat{y}$ is

$$\hat{y} - y = A\hat{x} - Ax = A(\hat{x} - x) = Ae,$$

and if I have a consistent matrix norm, I can write

$$\|\hat{y} - y\| \leq \|\hat{A}\| \|e\|$$

Remember, though, I usually care about the relative error:

$$\frac{\|\hat{y} - y\|}{\|y\|} \leq \|A\| \frac{\|x\|}{\|y\|} \frac{\|e\|}{\|y\|}$$

If A is invertible, then $x = A^{-1}y$, and so I can write

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

Therefore,

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

The quantity $\kappa(A) = \|A\|\|A^{-1}\|$ is the condition number for matrix multiplication. It is also the condition number for multiplication by the inverse (solving a linear system).

Ill-conditioned matrices are "close to" singular in a well-defined sense: if $\kappa(A) \gg 1$, then there is a perturbation E, $\|E\| \ll \|A\|$, such that $A + E$ is singular. An exactly singular matrix (which has no inverse) can be thought of as infinitely ill-conditioned. That is because if x is a null vector for A, an arbitrarily small relative error in x can lead to an "infinite" relative error in Ax (since no nonzero vector can be a good approximation to a zero vector).