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 $\text{rank}(A) = n$, or if $\text{null}(A) = \{0\}$. 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. $\text{rank}(A) = n$. 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. $\text{null}(A) = \{0\}$. 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:

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

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

[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.