Week 4: Wednesday, Jan 16

Problem du jour[1]

1. Suppose $A = LU$ is given. Given a fast method to compute $\det(A)$.

2. 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 $b, c \in bbR^n$ 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?

Logistics

1. The prelim is on Tuesday at 7:30. It is a one-hour, closed book exam (you can bring a page of handwritten notes – both sides of a sheet of paper). If you will need to take a make-up because of a conflict with another class, please let me know so that we can figure out timing. A practice exam will be posted shortly.

2. Project 1 is due Monday. Don't procrastinate. It's not entirely trivial, and figuring out the project is actually a good way to study for some aspects of the exam.

Bordered systems: a digression

We've discussed in the past why the determinant might not be a good basis for a test for singularity. An alternative test which is sometimes used in numerical bifurcation analysis involves a *bordered system*. An example is

[1] Because I was asked: "du jour" is French for "of the day."

given in the problem of the day: we can test whether A is singular by looking at the value of y in the solution to the linear system

$$\begin{bmatrix} A & b \\ c^T & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

Let's see if we can figure out *why* this should be the case. Let's first suppose that A is just "close" to singular rather than being exactly singular, and that we can write $A = LU$. Then we can write the LU factorization of the extended matrix as

$$\begin{bmatrix} A & b \\ c^T & d \end{bmatrix} = \begin{bmatrix} L & 0 \\ c^T U^{-1} & 1 \end{bmatrix} \begin{bmatrix} U & L^{-1}b \\ 0 & d - c^T U^{-1} L^{-1} b \end{bmatrix}$$

Notice that

$$\begin{bmatrix} L & 0 \\ c^T U^{-1} & 1 \end{bmatrix}^{-1} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix},$$

so

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} A & b \\ c^T & d \end{bmatrix}^{-1} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} U & L^{-1}b \\ 0 & d - c^T U^{-1} L^{-1} b \end{bmatrix}^{-1} \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

The last row of this equation gives us

$$y = \left(d - c^T A^{-1} b\right)^{-1}.$$

As A becomes closer and closer to singular, we expect $c^T A^{-1} b$ to tend toward infinity, so that y tends toward zero.

Gaussian elimination in Matlab again

Last time, we discussed two MATLAB calls for factoring a dense or sparse matrix A using Gaussian elimination with partial pivoting:

`[L,U,P] = lu(A)`	Compute $PA = LU$ for dense problems
`[L,U,P,Q] = lu(A)`	Compute $PAQ = LU$ for sparse problems

Here P is a permutation matrix that reorders equations (for stability) and Q is a permutation matrix that reorders variables (for sparsity). But why should we ever call the `lu` function instead of just using backslash to solve with A? After all, using backslash to solve with A also does factorization. The answer is that calling `lu` is an $O(n^3)$ time operation in the dense case,

just like using backslash with A. Solving with L and U, on the other hand, costs $O(n^2)$. There are often times when we want to solve a sequence of linear systems with the same matrix and different right-hand sides; for example, we could compute the first-order Taylor approximation of $(A+tE)^{-1}b$ about $t = 0$ this way:

```
[L,U,P] = lu(A);
x0  =  U\(L\(P*b));
x0p = -(U\(L\(P*(E*x0))));
xbar = x0 + t*x0p;
```

If we did not do the LU factorization at the start, we would pay for the factorization twice: once when computing `x0` and once when computing `x0p`.

Partial pivoting

I have said little 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:

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

In the book, Heath points out that 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.

Error in Gaussian elimination

Barring the possibility of extreme pivot growth, Gaussian elimination with partial pivoting is *backward stable*. That means that we can explain the effects of roundoff in the computation in terms of a small perturbation to the original problem. That is, we exactly solve $\hat{A}\hat{x} = \hat{b}$, where $\|\hat{A} - A\|/\|A\|$ and $\|\hat{b} - b\|/\|b\|$ are typically around $n\epsilon_{\text{mach}}$. Recall that backward errors are connected to forward errors through the condition number, so

$$\frac{\|\hat{x} - x\|}{\|x\|} \lesssim \kappa(A) \left(\frac{\|\hat{A} - A\|}{\|A\|} + \frac{\|\hat{b} - b\|}{\|b\|} \right).$$

In practice, then, Gaussian elimination results in solutions with a small *forward* error whenever the matrix A is well-conditioned.

An alternative error bound applies even in the case when we solve $Ax = b$ using some method beside Gaussian elimination (or even in the event of large pivot growth)[2]. If $\hat{x}$ is an approximation to $A^{-1}b$, the *residual* is

$$r = b - A\hat{x}.$$

Notice that $A\hat{x} = (b - r)$, so we can see the residual as a backward error in b for the approximate solution $\hat{x}$. With a little algebra, we can again connect this backward error to the forward error through the condition number:

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

Summary: Using Gaussian elimination wisely

To summarize the past two lectures, here is our advice on solving most linear systems:

[2] I think I neglected to actually talk about this in class, but the basic idea — relating a residual to the forward error — will come up again in later lectures, so I want to mention it in these notes.

1. Solve *dense* linear systems using Gaussian elimination with partial pivoting, either by explicitly computing an LU factorization or by using software (like MATLAB's backslash operator) that does the LU factorization for you behind the scenes. There is good, fast software available for this procedure, and it is backward stable in practice (though this will not save you from the need to be aware of ill conditioning – which you can detect cheaply based on MATLAB's `condest` and related routines).

2. Factoring a dense linear system costs $O(n^3)$. Subsequent solves with the triangular factors (forward and backward substitution) cost $O(n^2)$ for each right-hand side. Therefore, if you will use the same matrix in multiple problems, it pays to compute the factorization once and re-use it.

3. For matrices with special structure, there are variants of Gaussian elimination that are potentially much faster than $O(n^3)$. One type of special structure that occurs in many practical problems is *sparsity*, where most of the entries of the matrix are zero. MATLAB has a special data type for sparse matrices, and specialized routines to factor and solve with these matrices. These routines are very fast assuming that there is not too much *fill* (nonzeros in L and U in locations where A is nonzero). To try to minimize fill, sparse factorizations usually also involve a *column* permutation, and so we write $PAQ = LU$.

4. The performance behavior of sparse Gaussian elimination depends on the graph structure associated with the matrix. For many common structures, sparse factorization methods work terrifically well; but for some matrices, there is too much fill, and we would rather use *iterative methods*. We will discuss iterative methods for linear systems later in the course, around the same time we talk about iterative methods for the solution of nonlinear equations.