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 $\text{nnz}(A)$).

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.

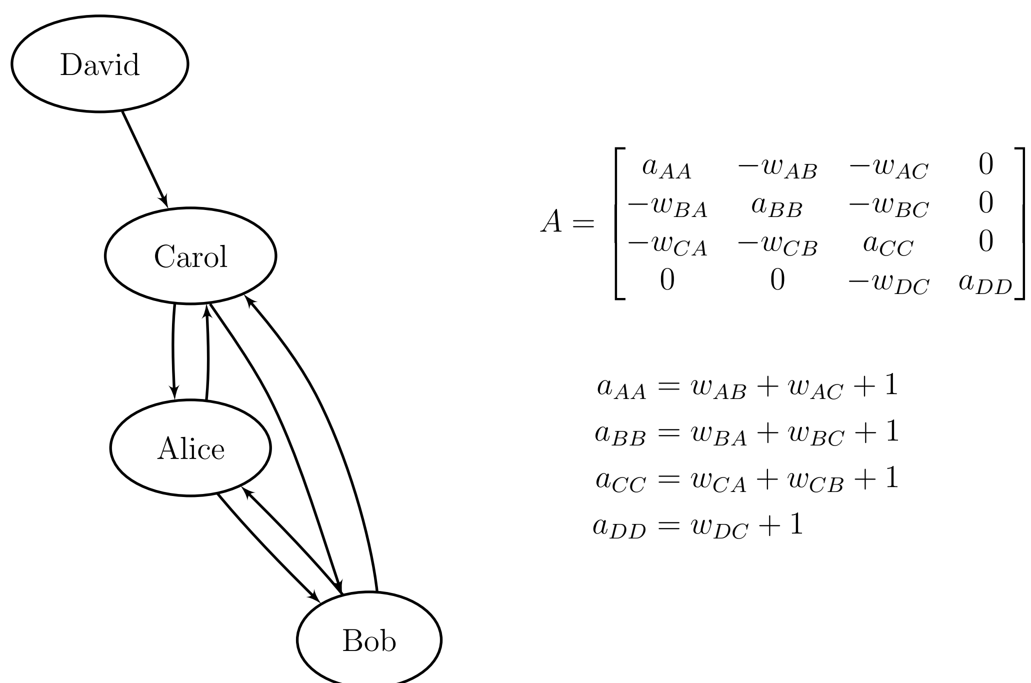

Figure 1: Small example of opinion formation in a social network. If Alice, Bob, Carol, and David have "intrinsic" opinions s_A, s_B, s_C, s_D, then we compute the "espressed" opinions x_A, x_B, x_C, x_D according to $Ax = s$.