Chapter 2

Linear Equations

2.1 Solving Linear Systems

One of the most frequent problems encountered in scientific computation is the solution of a system of simultaneous linear equations, usually with as many equations as unknowns. Such a system can be written in the form

$$Ax = b$$

where A is a given square matrix of order n, b is a given column vector of n components, and x is an unknown column vector of n components.

Students of linear algebra learn that the solution to $Ax = b$ can be written $x = A^{-1}b$ where A^{-1} is the inverse of A. However, in the vast majority of practical computational problems, it is unnecessary and inadvisable to actually compute A^{-1}. As an extreme but illustrative example, consider a system consisting of just one equation, such as

$$7x = 21$$

The best way to solve such a system is by division,

$$x = \frac{21}{7} = 3$$

Use of the matrix inverse would lead to

$$x = 7^{-1} \times 21 = .142857 \times 21 = 2.99997$$

The inverse requires more arithmetic — a division and a multiplication instead of just a division — and produces a less accurate answer. Similar considerations apply to systems of more than one equation. This is even true in the common situation where there are several systems of equations with the same matrix A but different right hand sides b. Consequently, we shall concentrate on the direct solution of systems of equations rather than the computation of the inverse.

2.2 The MATLAB Backslash Operator

To emphasize the distinction between solving linear equations and computing inverses, MATLAB has introduced nonstandard notation using *backward slash* and *forward slash* operators, "\" and "/".

If A is a matrix of any size and shape and B is a matrix with as many rows as A, then the solution to the system of simultaneous equations

$$AX = B$$

is denoted by

$$X = A\backslash B$$

Think of this as dividing both sides of the equation by the coefficient matrix A. Because matrix multiplication is not commutative and A occurs on the left in the original equation, this is *left division*.

Similarly, the solution to a system with A on the right and B with as many columns as A,

$$XA = B$$

is obtained by *right division*,

$$X = B/A$$

This notation applies even when A is not square, so that the number of equations is not the same as the number of unknowns. However, in this chapter, we limit ourselves to systems with square coefficient matrices.

2.3 A 3-by-3 Example

To illustrate the general linear equation solution algorithm, consider an example of order three:

$$\begin{pmatrix} 10 & -7 & 0 \\ -3 & 2 & 6 \\ 5 & -1 & 5 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 7 \\ 4 \\ 6 \end{pmatrix}$$

This of course, represents the three simultaneous equations

$$\begin{aligned} 10x_1 - 7x_2 &= 7 \\ -3x_1 + 2x_2 + 6x_3 &= 4 \\ 5x_1 - x_2 + 5x_3 &= 6 \end{aligned}$$

The first step uses the first equation to eliminate x_1 from the other equations. This is accomplished by adding 0.3 times the first equation to the second equation and subtracting 0.5 times the first equation from the third equation. The coefficient 10 of x_1 in the first equation is called the first *pivot* and the quantities -0.3 and 0.5,

obtained by dividing the coefficients of x_1 in the other equations by the pivot, are called the *multipliers*. The first step changes the equations to

$$\begin{pmatrix} 10 & -7 & 0 \\ 0 & -0.1 & 6 \\ 0 & 2.5 & 5 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 7 \\ 6.1 \\ 2.5 \end{pmatrix}$$

The second step might use the second equation to eliminate x_2 from the third equation. However, the second pivot, which is the coefficient of x_2 in the second equation, would be -0.1, which is smaller than the other coefficients. Consequently, the last two equations are interchanged. This is called *pivoting*. It is not actually necessary in this example because there are no roundoff errors, but it is crucial in general.

$$\begin{pmatrix} 10 & -7 & 0 \\ 0 & 2.5 & 5 \\ 0 & -0.1 & 6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 7 \\ 2.5 \\ 6.1 \end{pmatrix}$$

Now, the second pivot is 2.5 and the second equation can be used to eliminate x_2 from the third equation. This is accomplished by adding 0.04 times the second equation to the third equation. (What would the multiplier have been if the equations had not been interchanged?)

$$\begin{pmatrix} 10 & -7 & 0 \\ 0 & 2.5 & 5 \\ 0 & 0 & 6.2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 7 \\ 2.5 \\ 6.2 \end{pmatrix}$$

The last equation is now

$$6.2x_3 = 6.2$$

This can be solved to give $x_3 = 1$. This value is substituted into the second equation:

$$2.5x_2 + (5)(1) = 2.5.$$

Hence $x_2 = -1$. Finally the values of x_2 and x_3 are substituted into the first equation:

$$10x_1 + (-7)(-1) = 7$$

Hence $x_1 = 0$. The solution is

$$x = \begin{pmatrix} 0 \\ -1 \\ 1 \end{pmatrix}$$

This solution can be easily checked using the original equations:

$$\begin{pmatrix} 10 & -7 & 0 \\ -3 & 2 & 6 \\ 5 & -1 & 5 \end{pmatrix} \begin{pmatrix} 0 \\ -1 \\ 1 \end{pmatrix} = \begin{pmatrix} 7 \\ 4 \\ 6 \end{pmatrix}$$

The entire algorithm can be compactly expressed in matrix notation. For this example, let

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ -0.3 & -0.04 & 1 \end{pmatrix}, \; U = \begin{pmatrix} 10 & -7 & 0 \\ 0 & 2.5 & 5 \\ 0 & 0 & 6.2 \end{pmatrix}, \; P = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix},$$

The matrix L contains the multipliers used during the elimination, the matrix U is the final coefficient matrix, and the matrix P describes the pivoting. In the next section, we will see that L and U are *triangular* matrices and that P is a *permutation* matrix. With these three matrices, we have

$$LU = PA$$

In other words, the original coefficient matrix can be expressed in terms of products involving matrices with simpler structure.

2.4 Permutation and Triangular Matrices

A *permutation matrix* is an identity matrix with the rows and columns interchanged. It has exactly one 1 in each row and column; all the other elements are 0. For example:

$$P = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

Multiplying a matrix A on the left by a permutation matrix, to give PA, permutes the rows of A. Multiplying on the right, AP, permutes the columns of A.

MATLAB can also use a *permutation vector* as a row or column index to rearrange the rows or columns of a matrix. Continuing with the P above, let `p` be the vector

```
p = [4 1 3 2]
```

Then `P*A` and `A(p,:)` are equal. The resulting matrix has the fourth row of A as its first row, the first row of A as its second row, and so on. Similarly, `A*P` and `A(:,p)` both produce the same permutation of the columns of `A`. The `P*A` notation is closer to traditional mathematics, PA, while the `A(p,:)` notation is faster and uses less memory.

Linear equations involving permutation matrices are trivial to solve. The solution to

$$Px = b$$

is simply a rearrangement of the components of b,

$$x = P^T b$$

An *upper triangular* matrix has all its nonzero elements above or on the main diagonal. A *unit lower triangular* matrix has ones on the main diagonal and all the rest of its nonzero elements below the main diagonal. For example:

$$U = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 0 & 5 & 6 & 7 \\ 0 & 0 & 8 & 9 \\ 0 & 0 & 0 & 10 \end{pmatrix}$$

is upper triangular, and

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 3 & 5 & 1 & 0 \\ 4 & 6 & 7 & 1 \end{pmatrix}$$

is unit lower triangular.

Linear equations involving triangular matrices are also easily solved. There are two variants of the algorithm for solving an n-by-n upper triangular system, $Ux = b$. Both begin by solving the last equation for the last variable, then the next to last equation for the next to last variable, and so on. One subtracts multiples of the columns of U from b:

```
x = zeros(n,1);
for k = n:-1:1
   x(k) = b(k)/U(k,k);
   i = (1:k-1)';
   b(i) = b(i) - x(k)*U(i,k);
end
```

The other uses inner products between the rows of U and portions of the emerging solution x:

```
x = zeros(n,1);
for k = n:-1:1
   j = k+1:n;
   x(k) = (b(k) - U(k,j)*x(j))/U(k,k);
end
```

2.5 LU Factorization

The algorithm that is almost universally used to solve square systems of simultaneous linear equations is one of the oldest numerical methods, the systematic elimination method, generally named after C. F. Gauss. Research in the period 1955 to 1965 revealed the importance of two aspects of Gaussian elimination that were not emphasized in earlier work: the search for pivots and the proper interpretation of the effect of rounding errors. Gaussian elimination and other aspects of matrix computation are studied in detail in the books by Forsythe and Moler

(1967), Stewart (1973), and Golub and VanLoan (1996). The reader who desires more information than we have in this chapter should consult these references.

In general, Gaussian elimination has two stages, the *forward elimination* and the *back substitution*. The forward elimination consists of $n-1$ steps. At the kth step, multiples of the kth equation are subtracted from the remaining equations to eliminate the kth variable. If the coefficient of x_k is "small," it is advisable to interchange equations before this is done. The elimination steps can be simultaneously applied to the right hand side, or the interchanges and multipliers saved and applied to the right hand side later. The back substitution consists of solving the last equation for x_n, then the next-to-last equation for x_{n-1}, and so on, until x_1 is computed from the first equation.

Let P_k, $k = 1, \cdots, n-1$, denote the permutation matrix obtained by interchanging the rows of the identity matrix in the same way the rows of A are interchanged at the kth step of the elimination. Let M_k denote the unit lower triangular matrix obtained by inserting the negatives of the multipliers used at the kth step below the diagonal in the kth column of the identity matrix. Let U be the final upper triangular matrix obtained after the $n-1$ steps. The entire process can be described by one matrix equation,

$$U = M_{n-1}P_{n-1}\cdots M_2P_2M_1P_1A$$

It turns out that this equation can be rewritten

$$L_1L_2\cdots L_{n-1}U = P_{n-1}\cdots P_2P_1A$$

where L_k is obtained from M_k by permuting and changing the signs of the multipliers below the diagonal. So, if we let

$$L = L_1L_2\cdots L_{n-1}$$
$$P = P_{n-1}\cdots P_2P_1$$

then we have

$$LU = PA$$

The unit lower triangular matrix L contains all the multipliers used during the elimination and the permutation matrix P accounts for all the interchanges.

For our example

$$A = \begin{pmatrix} 10 & -7 & 0 \\ -3 & 2 & 6 \\ 5 & -1 & 5 \end{pmatrix}$$

the matrices defined during the elimination are

$$P_1 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \; M_1 = \begin{pmatrix} 1 & 0 & 0 \\ 0.3 & 1 & 0 \\ -0.5 & 0 & 1 \end{pmatrix},$$

$$P_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}, \; M_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0.04 & 1 \end{pmatrix},$$

The corresponding L's are

$$L_1 = \begin{pmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ -0.3 & 0 & 1 \end{pmatrix}, \; L_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -0.04 & 1 \end{pmatrix},$$

The relation $LU = PA$ is called the *LU factorization* or the *triangular decomposition* of A. It should be emphasized that nothing new has been introduced. Computationally, elimination is done by row operations on the coefficient matrix, not by actual matrix multiplication. LU factorization is simply Gaussian elimination expressed in matrix notation. The triangular factors can also be computed by other algorithms; see Forsythe and Moler (1967).

With this factorization, a general system of equations

$$Ax = b$$

becomes a pair of triangular systems

$$Ly = Pb$$
$$Ux = y$$

2.6 Why Is Pivoting Necessary?

The diagonal elements of U are called *pivots*. The kth pivot is the coefficient of the kth variable in the kth equation at the kth step of the elimination. In our 3-by-3 example, the pivots are 10, 2.5, and 6.2. Both the computation of the multipliers and the back substitution require divisions by the pivots. Consequently, the algorithm cannot be carried out if any of the pivots are zero. Intuition should tell us that it is a bad idea to complete the computation if any of the pivots are nearly zero. To see this, let us change our example slightly to

$$\begin{pmatrix} 10 & -7 & 0 \\ -3 & 2.099 & 6 \\ 5 & -1 & 5 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 7 \\ 3.901 \\ 6 \end{pmatrix}$$

The (2,2) element of the matrix has been changed from 2.000 to 2.099, and the right hand side has also been changed so that the exact answer is still $(0, -1, 1)^T$. Let us assume that the solution is to be computed on a hypothetical machine that does decimal floating-point arithmetic with five significant digits.

The first step of the elimination produces

$$\begin{pmatrix} 10 & -7 & 0 \\ 0 & -0.001 & 6 \\ 0 & 2.5 & 5 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 7 \\ 6.001 \\ 2.5 \end{pmatrix}$$

The (2,2) element is now quite small compared with the other elements in the matrix. Nevertheless, let us complete the elimination without using any interchanges. The next step requires adding $2.5 \cdot 10^3$ times the second equation to the third.

$$(5 + (2.5 \cdot 10^3)(6))x_3 = (2.5 + (2.5 \cdot 10^3)(6.001))$$

On the right hand side, this involves multiplying 6.001 by $2.5 \cdot 10^3$. The result is $1.50025 \cdot 10^4$, which cannot be exactly represented in our hypothetical floating-point number system. It must be rounded to $1.5002 \cdot 10^4$. The result is then added to 2.5 and rounded again. In other words, both of the *5*'s shown in italics in

$$(5 + 1.5000 \cdot 10^4)x_3 = (2.5 + 1.5002\mathit{5} \cdot 10^4)$$

are lost in roundoff errors. On this hypothetical machine the last equation becomes

$$1.5005 \cdot 10^4 x_3 = 1.5004 \cdot 10^4$$

The back substitution begins with

$$x_3 = \frac{1.5004 \cdot 10^4}{1.5005 \cdot 10^4} = 0.99993$$

Because the exact answer is $x_3 = 1$, it does not appear that the error is too serious. Unfortunately, x_2 must be determined from the equation

$$-0.001x_2 + (6)(0.99993) = 6.001$$

which gives

$$x_2 = \frac{1.5 \cdot 10^{-3}}{-1.0 \cdot 10^{-3}} = -1.5$$

Finally x_1 is determined from the first equation,

$$10x_1 + (-7)(-1.5) = 7$$

which gives

$$x_1 = -0.35$$

Instead of $(0, -1, 1)^T$, we have obtained $(-0.35, -1.5, 0.99993)^T$.

Where did things go wrong? There was no "accumulation of rounding error" caused by doing thousands of arithmetic operations. The matrix is not close to singular. The difficulty comes from choosing a small pivot at the second step of the elimination. As a result, the multiplier is $2.5 \cdot 10^3$, and the final equation involves coefficients that are 10^3 times as large as those in the original problem. Roundoff errors that are small when compared to these large coefficients are unacceptable in terms of the original matrix and the actual solution.

We leave it to the reader to verify that if the second and third equations are interchanged, then no large multipliers are necessary and the final result is satisfactory. This turns out to be true in general: If the multipliers are all less than or equal to one in magnitude, then the computed solution can be proved to be accurate. Keeping the multipliers less than one in absolute value can be ensured by a process known as *partial pivoting*: At the kth step of the forward elimination, the pivot is taken to be the largest (in absolute value) element in the unreduced part of the kth column. The row containing this pivot is interchanged with the kth row to bring the pivot element into the (k, k) position. The same interchanges must be done with the elements of the right hand side, b. The unknowns in x are not reordered because the columns of A are not interchanged.

2.7 lutx, bslashtx, lugui

We have three functions implementing the algorithms discussed in this chapter. The first function, `lutx`, is a readable version of the built-in MATLAB function `lu`. There is one outer `for` loop on `k` that counts the elimination steps. The inner loops on `i` and `j` are implemented with vector and matrix operations, so that the overall function is reasonably efficient.

```
function [L,U,p] = lutx(A)
%LU Triangular factorization
%   [L,U,p] = lutx(A) produces a unit lower triangular
%   matrix L, an upper triangular matrix U and a
%   permutation vector p, so that L*U = A(p,:).

[n,n] = size(A);
p = (1:n)'

for k = 1:n-1

   % Find largest element below diagonal in k-th column
   [r,m] = max(abs(A(k:n,k)));
   m = m+k-1;

   % Skip elimination if column is zero
   if (A(m,k) ~= 0)

      % Swap pivot row
      if (m ~= k)
         A([k m],:) = A([m k],:);
         p([k m]) = p([m k]);
      end

      % Compute multipliers
      i = k+1:n;
      A(i,k) = A(i,k)/A(k,k);

      % Update the remainder of the matrix
      j = k+1:n;
      A(i,j) = A(i,j) - A(i,k)*A(k,j);
   end
end

% Separate result
L = tril(A,-1) + eye(n,n);
U = triu(A);
```

Study this function carefully. Almost all the execution time is spent in the statement

```
A(i,j) = A(i,j) - A(i,k)*A(k,j);
```

At the `k`th step of the elimination, `i` and `j` are index vectors of length `n-k`. The operation `A(i,k)*A(k,j)` multiplies a column vector by a row vector to produce a square, rank one matrix of order `n-k`. This matrix is then subtracted from the submatrix of the same size in the bottom right corner of `A`. In a programming language without vector and matrix operations, this update of a portion of `A` would be done with doubly nested loops on `i` and `j`.

The second function, `bslashtx`, is a simplified version of the built-in MATLAB backslash operator. It calls `lutx` to permute and factor the coefficient matrix, then uses the permutation and factors to complete the solution of a linear system.

```
function x = bslashtx(A,b)
%BSLASHTX  Solve linear system (backslash)
%   x = bslashtx(A,b) solves A*x = b

[n,n] = size(A);

% Triangular factorization
[L,U,p] = lutx(A);

% Permutation and forward elimination
y = zeros(n,1);
for k = 1:n
   j = 1:k-1;
   y(k) = b(p(k)) - L(k,j)*y(j);
end

% Back substitution
x = zeros(n,1);
for k = n:-1:1
   j = k+1:n;
   x(k) = (y(k) - U(k,j)*x(j))/U(k,k);
end
```

A third function, `lugui`, shows the steps in LU decomposition by Gaussian elimination. It is a version of `lutx` that allows you to experiment with various pivot selection strategies. At the kth step of the elimination, the largest element in the unreduced portion of the kth column is shown in magenta. This is the element that partial pivoting would ordinarily select as the pivot. You can then choose among four different pivoting strategies:

- Pick a pivot. Use the mouse to pick the magenta element, or any other element, as pivot.

- Diagonal pivoting. Use the diagonal element as the pivot.
- Partial pivoting. Same strategy as `lu` and `lutx`.
- Complete pivoting. Use the largest element in the unfactored submatrix as the pivot.

The chosen pivot is shown in red and the resulting elimination step is taken. As the process proceeds, the emerging columns of L are shown in green, and the emerging rows of U in blue.

2.8 Effect of Roundoff Errors

The rounding errors introduced during the solution of a linear system of equations almost always cause the computed solution — which we now denote by x_* — to differ somewhat from the theoretical solution, $x = A^{-1}b$. In fact, it *must* differ because the elements of x are usually not floating-point numbers. There are two common measures of the discrepancy in x_*: the *error*

$$e = x - x_*$$

and the *residual*

$$r = b - Ax_*$$

Matrix theory tells us that, because A is nonsingular, if one of these is zero, the other must also be zero. But they are not necessarily both 'small" at the same time. Consider the following example:

$$\begin{pmatrix} 0.780 & 0.563 \\ 0.913 & 0.659 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 0.217 \\ 0.254 \end{pmatrix}$$

What happens if we carry out Gaussian elimination with partial pivoting on a hypothetical three-digit decimal computer? First, the two rows (equations) are interchanged so that 0.913 becomes the pivot. Then the multiplier

$$\frac{0.780}{0.913} = 0.854 \quad \text{(to three places)}$$

is computed. Next, 0.854 times the new first row is subtracted from the new second row to produce the system

$$\begin{pmatrix} 0.913 & 0.659 \\ 0 & 0.001 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 0.254 \\ 0.001 \end{pmatrix}$$

Finally, the back substitution is carried out:

$$\begin{aligned} x_2 &= \frac{0.001}{0.001} = 1.00 \quad \text{(exactly)}, \\ x_1 &= \frac{0.254 - 0.659x_2}{0.913} \\ &= -0.443 \quad \text{(to three places)}. \end{aligned}$$

Thus the computed solution is

$$x_* = \begin{pmatrix} -0.443 \\ 1.000 \end{pmatrix}$$

To assess the accuracy without knowing the exact answer, we compute the residuals (exactly):

$$\begin{aligned} r &= b - Ax_* = \begin{pmatrix} 0.217 - ((0.780)(-0.443) + (0.563)(1.00)) \\ 0.254 - ((0.913)(-0.443) + (0.659)(1.00)) \end{pmatrix} \\ &= \begin{pmatrix} -0.000460 \\ -0.000541 \end{pmatrix} \end{aligned}$$

The *residuals* are less than 10^{-3}. We could hardly expect better on a three-digit machine. However, it is easy to see that the exact solution to this system is

$$x = \begin{pmatrix} 1.000 \\ -1.000 \end{pmatrix}$$

So the components of our computed solution actually have the wrong signs; the *error* is larger than the solution itself.

Were the small residuals just a lucky fluke? You should realize that this example is highly contrived. The matrix is incredibly close to being singular and is *not* typical of most problems encountered in practice. Nevertheless, let us track down the reason for the small residuals.

If Gaussian elimination with partial pivoting is carried out for this example on a computer with six or more digits, the forward elimination will produce a system something like

$$\begin{pmatrix} 0.913000 & 0.659000 \\ 0 & 0.000001 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 0.254000 \\ -0.000001 \end{pmatrix}$$

Notice that the sign of b_2 differs from that obtained with three-digit computation. Now the back substitution produces

$$\begin{aligned} x_2 &= \frac{-0.000001}{0.000001} = -1.00000, \\ x_1 &= \frac{0.254 - 0.659x_2}{0.913} \\ &= 1.00000, \end{aligned}$$

the exact answer. On our three-digit machine, x_2 was computed by dividing two quantities, both of which were on the order of rounding errors and one of which did not even have the correct sign. Hence x_2 can turn out to be almost anything. (In fact, if we used a machine with nine binary bits, we would obtain a completely different value.) Then this completely arbitrary value of x_2 was substituted into the first equation to obtain x_1. We can reasonably expect the residual from the first equation to be small — x_1 was computed in such a way as to make this certain. Now comes a subtle but crucial point. We can also expect the residual from the second equation to be small, *precisely because the matrix is so close to being singular*. The

two equations are very nearly multiples of one another, so any pair (x_1, x_2) that nearly satisfies the first equation will also nearly satisfy the second. If the matrix were known to be exactly singular, we would not need the second equation at all — any solution of the first would automatically satisfy the second.

Although this example is contrived and atypical, the conclusion we reached is not. It is probably the single most important fact that we have learned about matrix computation since the invention of the digital computer:

> *Gaussian elimination with partial pivoting is guaranteed to produce small residuals.*

Now that we have stated it so strongly, we must make a couple of qualifying remarks. By "guaranteed" we mean it is possible to prove a precise theorem that assumes certain technical details about how the floating-point arithmetic system works and that establishes certain inequalities that the components of the residual must satisfy. If the arithmetic units work some other way or if there is a bug in the particular program, then the "guarantee" is void. Furthermore, by "small" we mean on the order of roundoff error *relative to* three quantities: the size of the elements of the original coefficient matrix, the size of the elements of the coefficient matrix at intermediate steps of the elimination process, and the size of the elements of the computed solution. If any of these are "large," then the residual will not necessarily be small in an absolute sense. Finally, even if the residual is small, we have made no claims that the error will be small. The relationship between the size of the residual and the size of the error is determined in part by a quantity known as the *condition number* of the matrix, which is the subject of the next section.

2.9 Norms and Condition Numbers

The coefficients in the matrix and right hand side of a system of simultaneous linear equations are rarely known exactly. Some systems arise from experiments, and so the coefficients are subject to observational errors. Other systems have coefficients given by formulas that involve roundoff error in their evaluation. Even if the system can be stored exactly in the computer, it is almost inevitable that roundoff errors will be introduced during its solution. It can be shown that roundoff errors in Gaussian elimination have the same effect on the answer as errors in the original coefficients.

Consequently, we are led to a fundamental question. If perturbations are made in the coefficients of a system of linear equations, how much is the solution altered? In other words, if $Ax = b$, how can we measure the sensitivity of x to changes in A and b?

The answer to this question lies in making the idea of *nearly singular* precise. If A is a singular matrix, then for some b's a solution x will not exist, while for others it will not be unique. So if A is nearly singular, we can expect small changes in A and b to cause very large changes in x. On the other hand, if A is the identity matrix, then b and x are the same vector. So if A is nearly the identity, small changes in A and b should result in correspondingly small changes in x.

At first glance, it might appear that there is some connection between the size of the pivots encountered in Gaussian elimination with partial pivoting and *nearness to singularity*, because if the arithmetic could be done exactly, all the pivots would be nonzero if and only if the matrix is nonsingular. To some extent, it is also true that if the pivots are small, then the matrix is *close to singular*. However, when roundoff errors are encountered, the converse is no longer true — a matrix might be close to singular even though none of the pivots are small.

To get a more precise, and reliable, measure of nearness to singularity than the size of the pivots, we need to introduce the concept of a *norm* of a vector. This is a single number that measures the general size of the elements of the vector. The family of vector norms known as l_p depends on a parameter p, in the range $1 \le p \le \infty$.

$$\|x\|_p = (\sum_{i=1}^{n} |x_i|^p)^{1/p}$$

We almost always use $p = 1$, $p = 2$ or $\lim p \to \infty$

$$\begin{aligned} \|x\|_1 &= \sum_{i=1}^{n} |x_i| \\ \|x\|_2 &= (\sum_{i=1}^{n} |x_i|^2)^{1/2} \\ \|x\|_\infty &= \max_i |x_i| \end{aligned}$$

The l_1 norm is also known as the *Manhattan* norm because it corresponds to the distance traveled on a grid of city streets. The l_2 norm is the familiar Euclidean distance. The l_∞ norm is also known as the *Chebyshev* norm.

The particular value of p is often unimportant and we simply use $\|x\|$. All vector norms have the following basic properties associated with the notion of distance.

$$\begin{aligned} \|x\| &> 0 \text{ if } x \neq 0 \\ \|0\| &= 0 \\ \|cx\| &= |c|\|x\| \text{ for all scalars } c \\ \|x+y\| &\le \|x\| + \|y\|, \text{ (the } \textit{triangle inequality}) \end{aligned}$$

In MATLAB, $\|x\|_p$ is computed by `norm(x,p)` and `norm(x)` is the same as `norm(x,2)`. For example:

```
x = (1:4)/5
n1 = norm(x,1)
n2 = norm(x)
ninf = norm(x,inf)
```

produces

```
x =
    0.2000    0.4000    0.6000    0.8000

n1 =
     2.0000

n2 =
     1.0954

ninf =
     0.8000
```

Multiplication of a vector x by a matrix A results in a new vector Ax that can have a very different norm from x. This change in norm is directly related to the sensitivity we want to measure. The range of the possible change can be expressed by two numbers,

$$M = \max \frac{\|Ax\|}{\|x\|}$$
$$m = \min \frac{\|Ax\|}{\|x\|}$$

The max and min are taken over all nonzero vectors, x. Note that if A is singular, then $m = 0$. The ratio M/m is called the *condition number* of A,

$$\kappa(A) = \frac{\max \frac{\|Ax\|}{\|x\|}}{\min \frac{\|Ax\|}{\|x\|}}$$

It is not hard to see that $\|A^{-1}\| = 1/m$, so an equivalent definition of the condition number is

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

The actual numerical value of $\kappa(A)$ depends on the vector norm being used, but we are usually only interested in order of magnitude estimates of the condition number, so the particular norm is usually not very important.

Consider a system of equations

$$Ax = b$$

and a second system obtained by altering the right hand side:

$$A(x + \delta x) = b + \delta b$$

We think of δb as being the error in b and δx as being the resulting error in x, although we need not make any assumptions that the errors are small. Because $A(\delta x) = \delta b$, the definitions of M and m immediately lead to

$$\|b\| \leq M\|x\|$$

and

$$\|\delta b\| \geq m\|\delta x\|$$

Consequently, if $m \neq 0$,

$$\frac{\|\delta x\|}{\|x\|} \leq \kappa(A)\frac{\|\delta b\|}{\|b\|}$$

The quantity $\|\delta b\|/\|b\|$ is the *relative* change in the right hand side, and the quantity $\|\delta x\|/\|\mathrm{x}\|$ is the *relative* error caused by this change. The advantage of using relative changes is that they are *dimensionless*, that is, they are not affected by overall scale factors.

This shows that the condition number is a relative error magnification factor. Changes in the right hand side can cause changes $\kappa(A)$ times as large in the solution. It turns out that the same is true of changes in the coefficient matrix itself.

The condition number is also a measure of nearness to singularity. Although we have not yet developed the mathematical tools necessary to make the idea precise, the condition number can be thought of as the reciprocal of the relative distance from the matrix to the set of singular matrices. So, if $\kappa(A)$ is large, A is close to singular.

Some of the basic properties of the condition number are easily derived. Clearly, $M \geq m$, and so

$$\kappa(A) \geq 1$$

If P is a permutation matrix, then the components of Px are simply a rearrangement of the components of x. It follows that $\|Px\| = \|x\|$ for all x, and so

$$\kappa(P) = 1$$

In particular, $\kappa(I) = 1$. If A is multiplied by a scalar c, then M and m are both multiplied by the same scalar, and so

$$\kappa(cA) = \kappa(A)$$

If D is a diagonal matrix, then

$$\kappa(D) = \frac{\max|d_{ii}|}{\min|d_{ii}|}$$

The last two properties are two of the reasons that $\kappa(A)$ is a better measure of nearness to singularity than the determinant of A. As an extreme example, consider a 100-by-100 diagonal matrix with 0.1 on the diagonal. Then $det(A) = 10^{-100}$, which is usually regarded as a small number. But $\kappa(A) = 1$, and the components of Ax are simply 0.1 times the corresponding components of x. For linear systems of equations, such a matrix behaves more like the identity than like a singular matrix.

The following example uses the l_1 norm.

$$A = \begin{pmatrix} 4.1 & 2.8 \\ 9.7 & 6.6 \end{pmatrix}$$

$$b = \begin{pmatrix} 4.1 \\ 9.7 \end{pmatrix}$$
$$x = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

Clearly, $Ax = b$, and

$$\|b\| = 13.8, \quad \|x\| = 1$$

If the right hand side is changed to

$$\tilde{b} = \begin{pmatrix} 4.11 \\ 9.70 \end{pmatrix}$$

the solution becomes

$$\tilde{x} = \begin{pmatrix} 0.34 \\ 0.97 \end{pmatrix}$$

Let $\delta b = b - \tilde{b}$ and $\delta x = x - \tilde{x}$. Then

$$\|\delta b\| = 0.01$$
$$\|\delta x\| = 1.63$$

We have made a fairly small perturbation in b that completely changes x. In fact, the relative changes are

$$\frac{\|\delta b\|}{\|b\|} = 0.0007246$$
$$\frac{\|\delta x\|}{\|x\|} = 1.63$$

Because $\kappa(A)$ is the maximum magnification factor,

$$\kappa(A) \geq \frac{1.63}{0.0007246} = 2249.4$$

We have actually chosen the b and δb that give the maximum, and so for this example with the l_1 norm

$$\kappa(A) = 2249.4$$

It is important to realize that this example is concerned with the *exact* solutions to two slightly different systems of equations and that the method used to obtain the solutions is irrelevant. The example is constructed to have a fairly large condition number so that the effect of changes in b is quite pronounced, but similar behavior can be expected in any problem with a large condition number.

Suppose we want to solve a problem in which $a_{11} = 0.1$, all the other elements of A and b are integers, and $\kappa(A) = 10^{12}$. Suppose further that we use IEEE double-precision floating-point arithmetic with 53 bits in the fraction and that we can

somehow compute the exact solution to the system actually stored in the computer. Then the only error is caused by representing 0.1 in binary, but we can expect

$$\frac{\|\delta x\|}{\|x\|} \approx 10^{12} \times 2^{-53} \approx 2 \cdot 10^{-4}$$

In other words, the simple act of storing the coefficient matrix in the machine might cause changes in the fourth significant figures of the true solution.

The condition number also plays a fundamental role in the analysis of the roundoff errors introduced during the solution by Gaussian elimination. Let us assume that A and b have elements that are exact floating-point numbers, and let x_* be the vector of floating-point numbers obtained from a linear equation solver such as the function we shall present in the next section. We also assume that exact singularity is not detected and that there are no underflows or overflows. Then it is possible to establish the following inequalities:

$$\begin{aligned} \frac{\|b - Ax_*\|}{\|A\|\|x_*\|} &\leq \rho\epsilon, \\ \frac{\|x - x_*\|}{\|x_*\|} &\leq \rho\kappa(A)\epsilon \end{aligned}$$

Here ϵ is the relative machine precision `eps` and ρ is defined more carefully later, but it usually has a value no larger than about 10.

The first inequality says that the *relative residual* can usually be expected to be about the size of roundoff error, no matter how badly conditioned the matrix is. This was illustrated by the example in the previous section. The second inequality requires that A be nonsingular and involves the exact solution x. It follows directly from the first inequality and the definition of $\kappa(A)$ and says that the *relative error* will also be small if $\kappa(A)$ is small but might be quite large if the matrix is nearly singular. In the extreme case where A is singular, but the singularity is not detected, the first inequality still holds, but the second has no meaning.

To be more precise about the quantity ρ, it is necessary to introduce the idea of a *matrix norm* and establish some further inequalities. Readers who are not interested in such details can skip the remainder of this section. The quantity M defined earlier is known as the norm of the matrix. The notation for the matrix norm is the same as for the vector norm,

$$\|A\| = \max \frac{\|Ax\|}{\|x\|}$$

Again, the actual numerical value of the matrix norm depends on the underlying vector norm. It is easy to compute the matrix norms corresponding to the l_1 and l_∞ vector norms. In fact, it is not hard to show that

$$\begin{aligned} \|A\|_1 &= \max_j \sum_i |a_{i,j}| \\ \|A\|_\infty &= \max_i \sum_j |a_{i,j}| \end{aligned}$$

Computing the matrix norm corresponding to the l_2 vector norm involves the singular value decomposition, which is discussed in a later chapter. MATLAB computes matrix norms with `norm(A,p)` for `p = 1, 2,` or `inf`.

The basic result in the study of roundoff error in Gaussian elimination is due to J. H. Wilkinson. He proved that the computed solution x_* exactly satisfies

$$(A + E)x_* = b$$

where E is a matrix whose elements are about the size of roundoff errors in the elements of A. There are some rare situations where the intermediate matrices obtained during Gaussian elimination have elements that are larger than those of A, and there is some effect from accumulation of rounding errors in large matrices, but it can be expected that if ρ is defined by

$$\frac{\|E\|}{\|A\|} = \rho\epsilon$$

then ρ will rarely be bigger than about 10.

From this basic result, we can immediately derive inequalities involving the residual and the error in the computed solution. The residual is given by

$$b - Ax_* = Ex_*$$

and hence

$$\|b - Ax_*\| = \|Ex_*\| \le \|E\|\|x_*\|$$

The residual involves the product Ax_* so it is appropriate to consider the *relative residual*, which compares the norm of $b - Ax$ to the norms of A and x_*. It follows directly from the above inequalities that

$$\frac{\|b - Ax_*\|}{\|A\|\|x_*\|} \le \rho\epsilon$$

When A is nonsingular, the error can be expressed using the inverse of A by

$$x - x_* = A^{-1}(b - Ax_*)$$

and so

$$\|x - x_*\| \le \|A^{-1}\|\|E\|\|x_*\|$$

It is simplest to compare the norm of the error with the norm of the computed solution. Thus the *relative error* satisfies

$$\frac{\|x - x_*\|}{\|x_*\|} \le \rho\|A\|\|A^{-1}\|\epsilon$$

Hence

$$\frac{\|x - x_*\|}{\|x_*\|} \le \rho\kappa(A)\epsilon$$

The actual computation of $\kappa(A)$ requires knowing $\|A^{-1}\|$. But computing A^{-1} requires roughly three times as much work as solving a single linear system. Computing the l_2 condition number requires the singular value decomposition and even more work. Fortunately, the exact value of $\kappa(A)$ is rarely required. Any reasonably good estimate of it is satisfactory.

MATLAB has several functions for computing or estimating condition numbers.

- `cond(A)` or `cond(A,2)` computes $\kappa_2(A)$. Uses `svd(A)`. Suitable for smaller matrices where the geometric properties of the l_2 norm are important.
- `cond(A,1)` computes $\kappa_1(A)$. Uses `inv(A)`. Less work than `cond(A,2)`.
- `cond(A,inf)` computes $\kappa_\infty(A)$. Uses `inv(A)`. Same as `cond(A',1)`.
- `condest(A)` estimates $\kappa_1(A)$. Uses `lu(A)` and a recent algorithm of Higham and Tisseur. Especially suitable for large, sparse matrices.
- `rcond(A)` estimates $1/\kappa_1(A)$. Uses `lu(A)` and an older algorithm developed by the LINPACK and LAPACK projects. Primarily of historical interest.

2.10 Sparse Matrices and Band Matrices

Sparse matrices and band matrices occur frequently in technical computing. The *sparsity* of a matrix is the fraction of its elements that are zero. The MATLAB function `nnz` counts the number of nonzeros in a matrix, so the sparsity of `A` is given by

```
density  = nnz(A)/prod(size(A))
sparsity = 1 - density
```

A *sparse matrix* is a matrix whose sparsity is nearly equal to 1. The *band width* of a matrix is the maximum distance of the nonzero elements from the main diagonal.

```
[i,j] = find(A)
bandwidth = max(abs(i-j))
```

A *band matrix* is a matrix whose band width is small.

As you can see, both properties are matters of degree. An n-by-n diagonal matrix with no zeros on the diagonal has sparsity $1 - 1/n$ and bandwidth 0, so it is an extreme example of both a sparse matrix and a band matrix. On the other hand, an n-by-n matrix with no zero elements, such as the one created by `rand(n,n)`, has sparsity equal to zero, band width equal to $n - 1$, and so is far from qualifying for either category.

The MATLAB sparse data structure stores the nonzero elements together with information about their indices. The sparse data structure also provides efficient handling of band matrices, so MATLAB does not have a separate band matrix storage class. The statement

```
S = sparse(A)
```

converts a matrix to its sparse representation. The statement

```
A = full(S)
```

reverses the process. However, most sparse matrices have orders so large that it is impractical to store the full representation. More frequently, sparse matrices are created by

```
S = sparse(i,j,x,m,n)
```

This produces a matrix `S` with

```
[i,j,x] = find(S)
[m,n] = size(S)
```

Most MATLAB matrix operations and functions can be applied to both full and sparse matrices. The dominant factor in determining the execution time and memory requirements for sparse matrix operations is the number of nonzeros, `nnz(S)`, in the various matrices involved.

A matrix with band width equal to 1 is known as a *tridiagonal* matrix. It is worthwhile to have a specialized function for one particular band matrix operation, the solution of a tridiagonal system of simultaneous linear equations.

$$\begin{pmatrix} b_1 & c_1 & & & & \\ a_1 & b_2 & c_2 & & & \\ & a_2 & b_3 & c_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & a_{n-2} & b_{n-1} & c_{n-1} \\ & & & & a_{n-1} & b_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_{n-1} \\ d_n \end{pmatrix}$$

The function `tridisolve` is included in the NCM directory. The statement

```
x = tridisolve(a,b,c,d)
```

solves the tridiagonal system with subdiagonal `a`, diagonal `b`, superdiagonal `c`, and right hand side `d`. We have already seen the algorithm that `tridisolve` uses, it is Gaussian elimination. In this context, Gaussian elimination is also known as the *Thomas algorithm*. In many situations involving tridiagonal matrices, the diagonal elements dominate the offdiagonal elements, so pivoting is unnecessary. The right hand side is processed at the same time as the matrix itself. The body of `tridisolve` begins by copying the right hand side to a vector that will become the solution.

```
x = d;
n = length(x);
```

The forward elimination step is a simple `for` loop.

```
for j = 1:n-1
   mu = a(j)/b(j);
   b(j+1) = b(j+1) - mu*c(j);
   x(j+1) = x(j+1) - mu*x(j);
end
```

The `mu`'s would be the multipliers on the subdiagonal of L if we were saving the LU factorization. Instead, the right hand side is processed in the same loop. The back substitution step is another simple loop.

```
x(n) = x(n)/b(n);
for j = n-1:-1:1
   x(j) = (x(j)-c(j)*x(j+1))/b(j);
end
```

Because `tridisolve` does not use pivoting, the results might be inaccurate if `abs(b)` is much smaller than `abs(a)+abs(c)`. More robust, but slower, alternatives that do use pivoting include generating a full matrix with `diag`

```
T = diag(a,-1) + diag(b,0) + diag(c,1);
x = T\d
```

or generating a sparse matrix with `spdiags`

```
S = spdiags([a b c],[-1 0 1],n,n);
x = S\d
```

2.11 PageRank and Markov Chains

One of the reasons why Google™ is such an effective search engine is the PageRank™ algorithm developed by Google's founders, Larry Page and Sergey Brin, when they were graduate students at Stanford University. PageRank is determined entirely by the link structure of the World Wide Web. It is recomputed about once a month and does not involve the actual content of any web pages or individual queries. Then, for any particular query, Google finds the pages on the Web that match that query and lists those pages in the order of their PageRank.

Imagine surfing the Web, going from page to page by randomly choosing an outgoing link from one page to get to the next. This can lead to dead ends at pages with no outgoing links, or cycles around cliques of interconnected pages. So, a certain fraction of the time, simply choose a random page from the Web. This theoretical random walk is known as a *Markov chain* or *Markov process*. The limiting probability that an infinitely dedicated random surfer visits any particular page is its PageRank. A page has high rank if other pages with high rank link to it.

Let W be the set of web pages that can reached by following a chain of hyperlinks starting at some root page and let n be the number of pages in W. For Google, the set W actually varies with time, but by the end of 2002, n was over 3 billion. Let G be the n-by-n connectivity matrix of the Web, that is $g_{ij} = 1$ if there is a hyperlink from page j to page i and zero otherwise. The matrix G can be huge, but it is very sparse. Its jth column shows the links on the jth page. The number of nonzeros in G is the total number of hyperlinks in W.

Let r_i and c_j be the row and column sums of G.

$$r_i = \sum_j g_{ij}, \quad c_j = \sum_i g_{ij},$$

The quantities r_j and c_j are the *in-degree* and *out-degree* of the jth page. Let p be the probability that the random walk follows a link. Google usually takes $p = 0.85$. Then $1 - p$ is the probability that an arbitrary page is chosen. Let A be the n-by-n matrix whose elements are

$$a_{ij} = pg_{ij}/c_j + \delta, \text{ where } \delta = (1 - p)/n.$$

Notice that A comes from scaling the connectivity matrix by its column sums. The jth column is the probability of jumping from the jth page to the other pages on the Web. Most of the elements of A are equal to δ, the probability of jumping from one page to another without following a link. When $n = 3 \cdot 10^9$, then $\delta = 5 \cdot 10^{-11}$.

The matrix A is the transition probability matrix of the Markov chain. Its elements are all strictly between zero and one and its column sums are all equal to one. An important result in matrix theory known as the *Perron-Frobenius Theorem* applies to such matrices. It concludes that a nonzero solution of the equation

$$x = Ax$$

exists and is unique to within a scaling factor. When this scaling factor is chosen so that

$$\sum_i x_i = 1$$

then x is the *state vector* of the Markov chain and is Google's PageRank. The elements of x are all positive and less than one.

The vector x is the solution to the singular, homogenous linear system

$$(I - A)x = 0$$

For modest n, an easy way to compute x in MATLAB is to start with some approximate solution, such as the PageRanks from the previous month, or

```
x = ones(n,1)/n
```

Then simply repeat the assignment statement

```
x = A*x
```

until successive vectors agree to within a specified tolerance. This is known as the *power method* and is about the only possible approach for very large n. In practice, the matrices G and A are never actually formed. One step of the power method would be done by one pass over a database of Web pages, updating weighted reference counts generated by the hyperlinks between pages.

The best way to compute PageRank in MATLAB is to take advantage of the particular structure of the Markov matrix. The equation

$$x = Ax$$

can be written

$$x = \left(pGD + \delta ee^T\right) x$$

where e is the n-vector of all ones and D is the diagonal matrix formed from the reciprocals of the outdegrees.

$$d_{jj} = \frac{1}{c_j}$$

We want to have

$$e^T x = 1$$

so the equation becomes

$$(I - pGD)x = \delta e$$

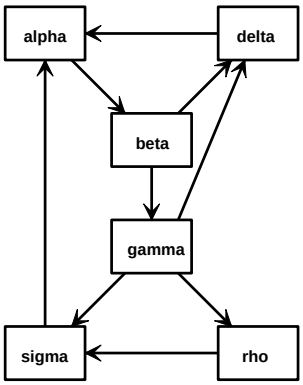
As long as p is strictly less than one, the coefficient matrix $I - pGD$ is nonsingular and these equations can be solved for x. This approach preserves the sparsity of G, but it breaks down as $p \to 1$ and $\delta \to 0$.

It is also possible to use a single step of an algorithm known as *inverse iteration*.

```
e = ones(n,1)
I = eye(n,n)
x = (I - A)\e
x = x/sum(x)
```

At first glance, this appears to be a very dangerous idea. Because $I - A$ is theoretically singular, with exact computation some diagonal element of the upper triangular factor of $I - A$ should to be zero and this computation should fail. But with roundoff error, the computed `I - A` is probably not exactly singular. Even if it is singular, roundoff during Gaussian elimination will most likely prevent any exact zero diagonal elements. We know that Gaussian elimination with partial pivoting always produces a solution with a small residual, relative to the computed solution, even if the matrix is badly conditioned. The vector obtained with the backslash operation, `(I - A)\e`, will usually have very large components. When it is rescaled by its sum, the residual will be scaled by the same factor and become very small. Consequently, the two vectors `x` and `A*x` will almost always be equal to each other to within roundoff error. In this setting, solving the singular system with Gaussian elimination blows up, but it blows up in exactly the right direction.

Here is the graph for a very small example, with $n = 6$ instead of $n = 3$ billion.



Pages on the Web are identified by strings known as *Universal Record Locators*, or *URLs*. Most URLs begin with `http` because they use the *Hypertext Transfer Protocol*. In MATLAB we can store the URLs as an array of strings in a *cell array*. This example involves a 6-by-1 cell array.

```
U = {'http://www.alpha.com'
     'http://www.beta.com'
     'http://www.gamma.com'
     'http://www.delta.com'
     'http://www.rho.com'
     'http://www.sigma.com'}
```

Two different kinds of indexing into cell arrays are possible. Parentheses denote subarrays, including individual cells, and curly braces denote the contents of the cells. If `k` is a scalar, then `U(k)` is a 1-by-1 cell array consisting of the `k`th cell in `U`, while `U{k}` is the string in that cell. Thus `U(1)` is a single cell and `U{1}` is the string `'http://www.alpha.com'`. Think of mail boxes with addresses on a city street. `B(502)` is the box at number 502, while `B{502}` is the mail in that box.

We can generate the connectivity matrix by specifying the pairs of indices `(i,j)` of the nonzero elements. Because `alpha.com` is connected to `beta.com`, the `(2,1)` element of G is nonzero. The nine connections are described by

```
j = [ 1 2 2 3 3 3 4 5 6]
i = [ 2 3 4 4 5 6 1 6 1]
```

A sparse matrix is stored in a data structure that requires memory only for the nonzero elements and their indices. This is hardly necessary for a 6-by-6 matrix with only 27 zero entries, but it becomes crucially important for larger problems. The statements

```
n = 6
G = sparse(i,j,1,n,n)
```

generate the sparse representation of an `n`-by-`n` matrix with ones in the positions specified by the vectors `j` and `i`. Then the statement

```
full(G)
```

displays

```
 0     0     0     1     0     1
 1     0     0     0     0     0
 0     1     0     0     0     0
 0     1     1     0     0     0
 0     0     1     0     0     0
 0     0     1     0     1     0
```

We need to scale G by its column sums

```
c = sum(G)
```

It has been proposed that future versions of MATLAB allow the expression

```
G./c
```

to divide each column of `G` by the corresponding element of `c`. Until this is available, it is best to use the `spdiags` function to create a sparse diagonal matrix

```
D = spdiags(1./c',0,n,n)
```

The sparse matrix product `G*D` will then be computed efficiently. The statements

```
p = .85
delta = (1-p)/n
e = ones(n,1)
I = speye(n,n)
x = (I - p*G*D)\(delta*e)
```

compute PageRank by solving the sparse linear system with Gaussian elimination.

```
x =
 0.2675
 0.2524
 0.1323
 0.1697
 0.0625
 0.1156
```

The Markov transition matrix is

```
A = p*G*D + delta
```

In this tiny example, the smallest element in `A` is $\delta = .15/6 = .0250$.

```
A =
 0.0250    0.0250    0.0250    0.8750    0.0250    0.8750
 0.8750    0.0250    0.0250    0.0250    0.0250    0.0250
 0.0250    0.4500    0.0250    0.0250    0.0250    0.0250
 0.0250    0.4500    0.3083    0.0250    0.0250    0.0250
 0.0250    0.0250    0.3083    0.0250    0.0250    0.0250
 0.0250    0.0250    0.3083    0.0250    0.8750    0.0250
```

Notice that the column sums of `A` are all equal to one.

The first step of computing PageRank with inverse iteration is

```
e = ones(n,1)
I = eye(n,n)
x = (I - A)\e
```

This produces a warning message about illconditioning and a vector with elements on the order of 10^{16}. On one particular computer the elements of `x` happen to be negative and their sum is

```
s = sum(x)
  = -6.6797e+016
```

Other computers with different roundoff error might give other results. But in all cases, the rescaled solution

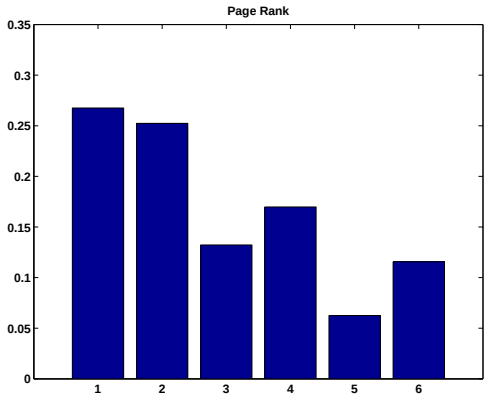
```
x = x/sum(x)
```

is the same as the `x` computed by sparse backslash. This x satisfies the equation

$$x = Ax$$

to within roundoff error.

The `bar` graph of `x` is



When the URLs are sorted in PageRank order and listed along with their in- and out-degrees, the result is

```
   page-rank  in  out  url
1    0.2675    2    1  http://www.alpha.com
2    0.2524    1    2  http://www.beta.com
4    0.1697    2    1  http://www.delta.com
3    0.1323    1    3  http://www.gamma.com
6    0.1156    2    1  http://www.sigma.com
5    0.0625    1    1  http://www.rho.com
```

We see that `alpha` has a higher PageRank than `delta` or `sigma`, even though they all have the same number of links, and that `beta` is ranked second because it basks in `alpha`'s glory. A random surfer will visit `alpha` almost 27% of the time and `rho` just about 6% of the time.

Our collection of NCM programs includes `surfer.m`. A statement like

```
[U,G] = surfer('http://www.xxx.zzz',n)
```

starts at a specified URL and tries to surf the Web until it has visited `n` pages. If successful, it returns an `n`-by-1 cell array of URLs and an `n`-by-`n` sparse connectivity matrix. The function uses `urlread`, which was introduced in MATLAB 6.5, along with underlying Java utilities to access the Web. Surfing the Web automatically is a dangerous undertaking and this function must be used with care. Some URLs contain typographical errors and illegal characters. There is a list of URLs to avoid that includes `.gif` files and Web sites known to cause difficulties. Most importantly, `surfer` can get completely bogged down trying to read a page from a site that appears to be responding, but that never delivers the complete page. When this happens, it may be necessary to have the computer's operating system ruthlessly terminate MATLAB. With these precautions in mind, you can use `surfer` to generate your own PageRank examples.

The statement

```
[U,G] = surfer('http://www.harvard.edu',500)
```

accesses the home page of Harvard University and generates a 500-by-500 test case. You can obtain the same data set from the NCM directory with

```
load harvard500
```

The statement

```
spy(G)
```

produces a `spy` plot that shows the nonzero structure of the connectivity matrix. The statement

```
pagerank(U,G)
```

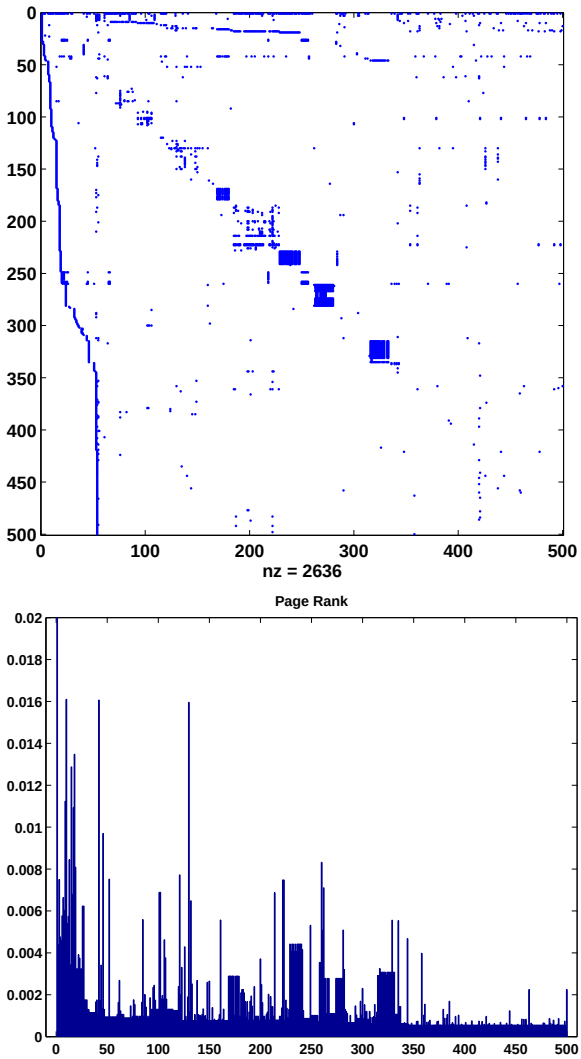
computes page ranks, produces a bar graph of the ranks, and prints the most highly ranked URLs in PageRank order.

For the `harvard500` data, the dozen most highly ranked pages are

```
     page-rank  in  out  url
  1    0.0823  195   26  http://www.harvard.edu
 10    0.0161   21   18  http://www.hbs.edu
 42    0.0161   42    0  http://search.harvard.edu:8765/custom/query.html
130    0.0160   24   12  http://www.med.harvard.edu
 18    0.0135   45   46  http://www.gse.harvard.edu
 15    0.0129   16   49  http://www.hms.harvard.edu
  9    0.0112   21   27  http://www.ksg.harvard.edu
```

```
 17   0.0109   13    6   http://www.hsph.harvard.edu
 46   0.0097   18   21   http://www.gocrimson.com
 13   0.0084    9    1   http://www.hsdm.med.harvard.edu
260   0.0083   26    1   http://search.harvard.edu:8765/query.html
 19   0.0081   23   21   http://www.radcliffe.edu
```

The URL where the search began, `www.harvard.edu`, dominates. Like most universities, Harvard is organized into various colleges and institutes, including the Kennedy School of Government, the Harvard Medical School, the Harvard Business School and Radcliffe Institute. You can see that the home pages of these schools have high PageRank. With a different sample, such as the one generated by Google itself, the ranks would be different.

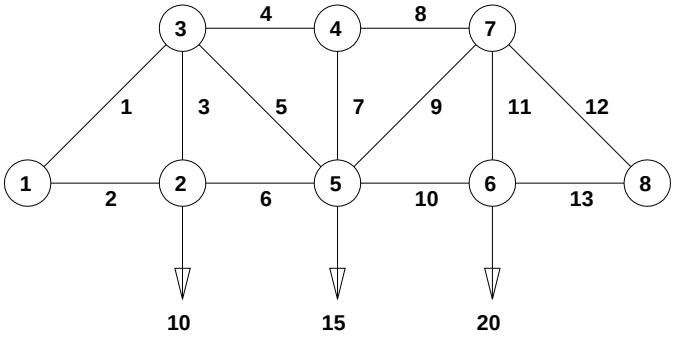


Exercises

2.1. Alice buys three apples, a dozen bananas and one cantaloupe for $2.36. Bob buys a dozen apples and two cantaloupes for $5.26. Carol buys two bananas and three cantaloupes for $2.77. How much do single pieces of each fruit cost? (You might want to set `format bank`.)

2.2. What MATLAB function computes the reduced row echelon form of a matrix? What MATLAB function generates magic square matrices? What is the reduced row echelon form of the magic square of order six?

2.3. The following diagram depicts a plane truss having 13 members (the numbered lines) connecting 8 joints (the numbered circles). The indicated loads, in tons, are applied at joints 2, 5, and 6, and we want to determine the resulting force on each member of the truss.



For the truss to be in static equilibrium, there must be no net force, horizontally or vertically, at any joint. Thus, we can determine the member forces by equating the horizontal forces to the left and right at each joint, and similarly equating the vertical forces upward and downward at each joint. For the eight joints, this would give 16 equations, which is more than the 13 unknown factors to be determined. For the truss to be statically determinate, that is, for there to be a unique solution, we assume that joint 1 is rigidly fixed both horizontally and vertically, and that joint 8 is fixed vertically. Resolving the member forces into horizontal and vertical components and defining $\alpha = 1/\sqrt{2}$, we obtain the following system of equations for the member forces f_i:

$$
\begin{aligned}
\text{Joint 2:} \quad & f_2 = f_6 \\
& f_3 = 10 \\
\text{Joint 3:} \quad & \alpha f_1 = f_4 + \alpha f_5 \\
& \alpha f_1 + f_3 + \alpha f_5 = 0 \\
\text{Joint 4:} \quad & f_4 = f_8 \\
& f_7 = 0 \\
\text{Joint 5:} \quad & \alpha f_5 + f_6 = \alpha f_9 + f_{10}
\end{aligned}
$$

$$\alpha f_5 + f_7 + \alpha f_9 = 15$$

Joint 6: $f_{10} = f_{13}$

$$f_{11} = 20$$

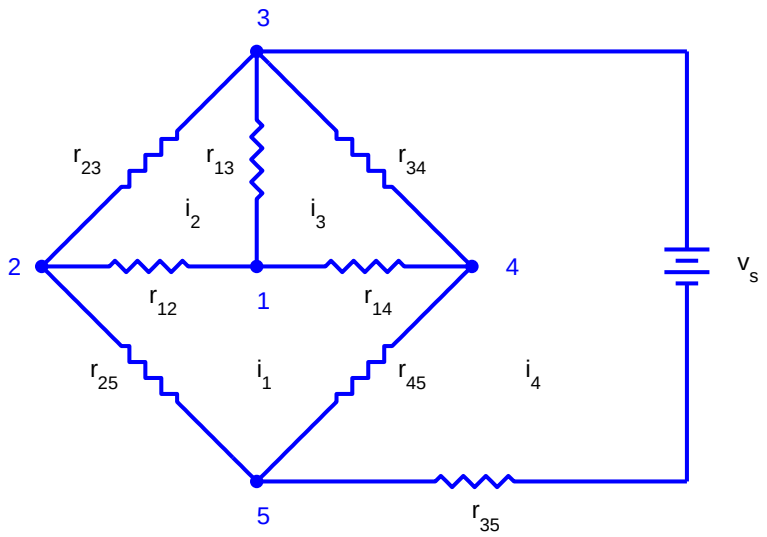
Joint 7: $f_8 + \alpha f_9 = \alpha f_{12}$

$$\alpha f_9 + f_{11} + \alpha f_{12} = 0$$

Joint 8: $f_{13} + \alpha f_{12} = 0$

Solve this system of equations to find the vector f of member forces.

2.4. Here is the circuit diagram for a small network of resistors.



There are five nodes, eight resistors, and one constant voltage source. We want to compute the voltage drops between the nodes and the currents around each of the loops.

Several different linear systems of equations can be formed to describe this circuit. Let $v_k, k = 1, \ldots, 4$ denote the voltage difference between each of the first four nodes and node number 5 and let $i_k, k = 1, \ldots, 4$ denote the clockwise current around each of the loops in the diagram. *Ohm's law* says that the voltage drop across a resistor is the resistance times the current. For example, the branch between nodes 1 and 2 gives

$$v_1 - v_2 = r_{12}(i_2 - i_1)$$

Using the *conductance*, which is the reciprocal of the resistance, $g_{kj} = 1/r_{kj}$, Ohm's law becomes

$$i_2 - i_1 = g_{12}(v_1 - v_2)$$

The voltage source is included in the equation

$$v_3 - v_s = r_{35} i_4$$

Kirchoff's voltage law says that the sum of the voltage differences around each loop is zero. For example, around loop 1,

$$(v_1 - v_4) + (v_4 - v_5) + (v_5 - v_2) + (v_2 - v_1) = 0$$

Combining the voltage law with Ohm's law leads to the *loop* equations for the currents.

$$Ri = b$$

Here i is the current vector,

$$i = \begin{pmatrix} i_1 \\ i_2 \\ i_3 \\ i_4 \end{pmatrix}$$

b is the source voltage vector,

$$b = \begin{pmatrix} 0 \\ 0 \\ 0 \\ v_s \end{pmatrix}$$

and R is the resistance matrix.

$$\begin{pmatrix} r_{25} + r_{12} + r_{14} + r_{45} & -r_{12} & -r_{14} & -r_{45} \\ -r_{12} & r_{23} + r_{12} + r_{13} & -r_{13} & 0 \\ -r_{14} & -r_{13} & r_{14} + r_{13} + r_{34} & -r_{34} \\ -r_{45} & 0 & -r_{34} & r_{35} + r_{45} + r_{34} \end{pmatrix}$$

Kirchoff's current law says that the sum of the currents at each node is zero. For example, at node 1,

$$(i_1 - i_2) + (i_2 - i_3) + (i_3 - i_1) = 0$$

Combining the current law with the conductance version of Ohm's law leads to the *nodal* equations for the voltages.

$$Gv = c$$

Here v is the voltage vector,

$$v = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{pmatrix}$$

c is the source current vector,

$$c = \begin{pmatrix} 0 \\ 0 \\ g_{35} v_s \\ 0 \end{pmatrix}$$

and G is the conductance matrix.

$$\begin{pmatrix} g_{12}+g_{13}+g_{14} & -g_{12} & -g_{13} & -g_{14} \\ -g_{12} & g_{12}+g_{23}+g_{25} & -g_{23} & 0 \\ -g_{13} & -g_{23} & g_{13}+g_{23}+g_{34}+g_{35} & -g_{34} \\ -g_{14} & 0 & -g_{34} & g_{14}+g_{34}+g_{45} \end{pmatrix}$$

You can solve the linear system obtained from the loop equations to compute the currents and then use Ohm's law to recover the voltages. Or, you can solve the linear system obtained from the node equations to compute the voltages and then use Ohm's law to recover the currents. Your assignment is to verify that these two approaches produce the same results for this circuit. You can choose your own numerical values for the resistances and the voltage source.

2.5. The Cholesky algorithm factors an important class of matrices known as *positive definite* matrices. Andre-Louis Cholesky (1875-1918) was a French military officer involved in geodesy and surveying in Crete and North Africa just before World War I. He developed the method now named after him to compute solutions to the normal equations for some least squares data-fitting problems arising in geodesy. His work was posthumously published on his behalf in 1924 by a fellow officer, Benoit, in the *Bulletin Geodesique*.
A real symmetric matrix $A = A^T$ is *positive definite* if any of the following equivalent conditions hold:

- The *quadratic form*

  $$x^T A x$$

  is positive for all nonzero vectors x.
- All *determinants* formed from symmetric submatrices of any order centered on the diagonal of A are positive.
- All *eigenvalues* $\lambda(A)$ are positive.
- There is a real matrix R so that

  $$A = R^T R$$

These conditions are difficult or expensive to use as the basis for checking if a particular matrix is positive definite. In MATLAB the best way to check positive definiteness is with the `chol` function. See

```
help chol
```

Which of the following families of matrices are positive definite?

```
M = magic(n)
H = hilb(n)
P = pascal(n)
I = eye(n,n)
R = randn(n,n)
R = randn(n,n); A = R' * R
```

```
R = randn(n,n); A = R' + R
R = randn(n,n); I = eye(n,n); A = R' + R + n*I
```

If the matrix R is upper triangular, then equating individual elements in the equation $A = R^T R$ gives

$$a_{kj} = \sum_{i=1}^{k} r_{ik} r_{ij}, \; k \le j$$

Using these equations in different orders yields different variants of the Cholesky algorithm for computing the elements of R. What is one such algorithm?

2.6. This example shows that a badly conditioned matrix does not necessarily lead to small pivots in Gaussian elimination. The matrix is the n-by-n upper triangular matrix A with elements

$$a_{ij} = \begin{cases} -1, & i < j \\ 1, & i = j \\ 0, & i > j \end{cases}$$

Show how to generate this matrix in MATLAB with `eye`, `ones`, and `triu`.
Show that

$$\kappa_1(A) = n 2^{n-1}$$

For what n does $\kappa_1(A)$ exceed `1/eps`?
This matrix is not singular, so Ax cannot be zero unless x is zero. However, there are vectors x for which $\|Ax\|$ is much smaller than $\|x\|$. Find one such x.
Because this matrix is already upper triangular, Gaussian elimination with partial pivoting has no work to do. What are the pivots?
Use `lugui` to design a pivot strategy that will produce smaller pivots than partial pivoting. (Even these pivots do not completely reveal the large condition number.)

2.7. The matrix factorization

$$LU = PA$$

can be used to compute the determinant of A. We have

$$det(L)det(U) = det(P)det(A)$$

Because L is triangular with ones on the diagonal, $det(L) = 1$. Because U is triangular, $det(U) = u_{11} u_{22} \cdots u_{nn}$. Because P is a permutation, $det(P) = +1$ if the number of interchanges is even and -1 if it is odd. So

$$det(A) = \pm u_{11} u_{22} \cdots u_{nn}.$$

Modify the `lutx` function so that it returns four outputs:

```
function [L,U,p,sig] = lutx(A)
%LU Triangular factorization
%  [L,U,p,sig] = lutx(A) computes a unit lower triangular
%  matrix L, an upper triangular matrix U, a permutation
%  vector p and a scalar sig, so that L*U = A(p,:) and
%  sig = +1 or -1 if p is an even or odd permutation.
```

Write a function `determ(A)` that uses your modified `lutx` to compute the determinant of A. In MATLAB, the product $u_{11}u_{22}\cdots u_{nn}$ can be computed with `prod(diag(U))`.

2.8. Modify the `lutx` function so that it uses explicit `for` loops instead of MATLAB vector notation. For example, one section of your modified program will read

```
% Compute the multipliers
for i = k+1:n
   A(i,k) = A(i,k)/A(k,k);
end
```

Compare the execution time of your modified `lutx` program with the original `lutx` program and with the built-in `lu` function by finding the order of the matrix for which each of the three programs takes about 10 seconds on your computer.

2.9. Let

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}, \quad b = \begin{pmatrix} 1 \\ 3 \\ 5 \end{pmatrix}$$

(a) Show that the set of linear equations $Ax = b$ has infinitely many solutions. Describe the set of possible solutions.
(b) Suppose `bslashtx` is used to solve $Ax = b$ on a hypothetical computer that does exact arithmetic. Because there are infinitely many solutions, it is unreasonable to expect one particular solution to be computed. What does happen?
(c) Use `bslashtx` to solve $Ax = b$ on an actual computer with floating-point arithmetic. What solution is obtained? Why? In what sense is it a "good" solution? In what sense is it a "bad" solution.
(d) Explain why the built-in backslash operator, `x = A\b`, gives a different solution from `x = bslashtx(A,b)`.

2.10. This chapter describes two algorithms for solving triangular systems. One subtracts columns of the triangular matrix from the right hand side; the other uses inner products between the rows of the triangular matrix and the emerging solution.
(a) Which of these two algorithms does `bslashtx` use?
(b) Write another function, `bslashtx2`, that uses the other algorithm.

2.11. The inverse of a matrix A can be defined as the matrix X whose columns x_j solve the equations

$$Ax_j = e_j$$

where e_j is the jth column of the identity matrix.
(a) Starting with the function `bslashtx`, write a MATLAB function

```
X = myinv(A)
```

that computes the inverse of `A`. Your function should call `lutx` only once and should not use the built-in MATLAB backslash operator or `inv` function.
(b) Test your function by comparing the inverses it computes with the inverses obtained from the built-in `inv(A)` on a few test matrices.

2.12. When built-in MATLAB `lu` function is called with only two output arguments

```
[L,U] = lu(A)
```

the permutations are incorporated into the output matrix `L`. The `help` entry for `lu` describes `L` as "psychologically lower triangular." Modify `lutx` so that it does the same thing. You can use

```
if nargout == 2, ...
```

to test the number of output arguments.

2.13. The pivot selection strategy known as *complete pivoting* is one of the options available in `lugui`. It has some slight numerical advantages over *partial pivoting*. At each state of the elimination the element of largest magnitude in the entire unreduced matrix is selected as pivot. This involves both row and column interchanges and produces two permutation vectors `p` and `q` so that

```
L*U = A(p,q)
```

Modify `lutx` and `bslashtx` so that they use complete pivoting.

2.14. The function `golub` in the `NCM` directory is named after Stanford's professor Gene Golub. The function generates badly conditioned test matrices with random integer entries that do not produce small pivots with Gaussian elimination with no pivoting.
(a) How does `condest(golub(n))` grow with increasing order `n`? Because these are random matrices you can't be very precise here, but you can give some qualitative description.
(b) What atypical behavior do you observe with the diagonal pivoting option in `lugui(golub(n))`?
(c) What is `det(golub(n))`? Why?

2.15. The function `pascal` generates symmetric test matrices based on Pascal's triangle.
(a) How are the elements of `pascal(n+1)` related to the binomial coefficients generated by `nchoosek(n,k)`?
(b) How is `chol(pascal(n))` related to `pascal(n)`?
(c) How does `condest(pascal(n))` grow with increasing order `n`?
(d) What is `det(pascal(n))`? Why?
(e) Let `Q` be the matrix generated by

```
Q = pascal(n);
Q(n,n) = Q(n,n) - 1;
```

How is `chol(Q)` related to `chol(pascal(n))`? Why?
(f) What is `det(Q)`? Why?

2.16. The object of this exercise is to investigate how the condition numbers of random matrices grow with their order. Let R_n denote an n-by-n matrix with normally distributed random elements. You should observe experimentally that there is an exponent p so that

$$\kappa_1(R_n) = O(n^p)$$

In other words, there are constants c_1 and c_2 so that most values of $\kappa_1(R_n)$ satisfy

$$c_1 n^p \le \kappa_1(R_n) \le c_2 n^p$$

Your job is to find p, c_1, and c_2.
Here is an M-file to start your experiments. The text is also in the file `NCM/randncond.m`. The program generates random matrices with normally distributed elements and plots their l_1 condition numbers versus their order on a loglog scale. The program also plots two lines that are intended to enclose most of the observations. (On a loglog scale, power laws like $\kappa = cn^p$ produce straight lines.)

```
% RANDNCOND  Condition of random matrices

nmax = 100;
n = 2:nmax;
kappalo = n.^(1/2);
kappahi = 500*n.^3;

shg
clf reset
h = loglog(n,[kappalo; kappahi],'-',nmax,NaN,'.');
set(h(1:2),'color',[0 .5 0]);
set(gca,'xtick',[2:2:10 20:20:nmax])
kappamax = 1.e6;
axis([2 nmax 2 kappamax])
stop = uicontrol('pos',[20 10 40 25], ...
   'style','toggle','string','stop','value',0);

h = h(3);
set(h,'erasemode','none','color','blue')
while get(stop,'value') ~= 1
   n = ceil(rand*nmax);
   A = randn(n,n);
   kappa = cond(A,1);
```

```
    set(h,'xdata',n,'ydata',kappa)
    drawnow
end
```

(a) Modify the program so that the two lines have the same slope and enclose most of the observations.
(b) Based on this experiment, what is your guess for the exponent p in $\kappa(R_n) = O(n^p)$? How confident are you?
(c) The program uses `('erasemode','none')`, so you cannot print the results. What would you have to change to make printing possible?

2.17. For $n = 100$, solve this tridiagonal system of equations three different ways.

$$\begin{aligned} 2x_1 - x_2 &= 1 \\ -x_{j-1} + 2x_j - x_{j+1} &= j,\ j = 2 \ldots n-1 \\ -x_{n-1} + 2x_n &= n \end{aligned}$$

(a) Use `diag` three times to form the coefficient matrix and then use `lutx` and `bslashtx` to solve the system.
(b) Use `spdiags` once to form a sparse representation of the coefficient matrix and then use the backslash operator to solve the system.
(c) Use `tridisolve` to solve the system.
(d) Use `condest` to estimate the condition of the coefficient matrix.

2.18. Use `surfer` and `pagerank` to compute PageRanks for some subset of the Web that you choose. Do you see any interesting structure in the results?

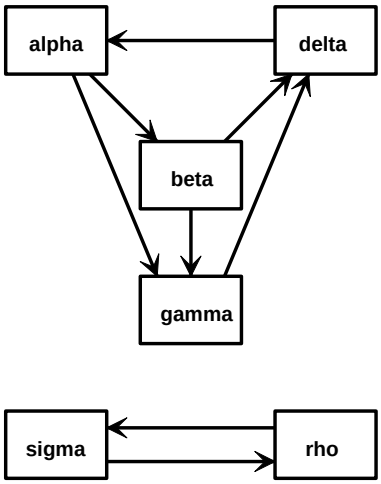
2.19. Suppose that `U` and `G` are the URL cell array and the connectivity matrix produced by `surfer` and that `k` is an integer. What are

```
U{k}, U(k), G(k,:), G(:,k), U(G(k,:)), U(G(:,k))
```

2.20. The connectivity matrix for the `harvard500` data set has four small, almost entirely nonzero submatrices that produce dense patches near the diagonal of the `spy` plot. You can use the zoom button to find their indices. The first submatrix has indices around 170 and the other three have indices in the 200s and 300s. Mathematically, a graph with every node connected to every other node is known as a *clique*. Identify the organizations within the Harvard community that are responsible for these near cliques.

2.21. The function `surfer` uses a subfunction, `hashfun`, to speed up the search for a possibly new URL in the list of URLs that have already been processed. Find two different URLs on the MathWorks home page, `http://www.mathworks.com` that have the same `hashfun` value.

2.22. Here is the graph of another six-node subset of the Web. In this example there are two disjoint subgraphs.



(a) What is the connectivity matrix, G?
(b) What are the page ranks when the hyperlink transition probability p is the default value, 0.85?
(c) Describe what happens with this example to both the definiton of page rank and the computation done by `pagerank` in the limit $p \to 1$.

2.23. The function `pagerank(U,G)` computes page ranks by solving a sparse linear system. It then plots a bar graph and prints the dominant URLs.
(a) Create `pagerank1(G)` by modifying `pagerank` so that it justs computes the page ranks, but does not do any plotting or printing.
(b) Create `pagerank2(G)` by modifying `pagerank1` to use inverse iteration instead of solving the sparse linear system. The key statements are:

```
x = (I - A)\e
x = x/sum(x)
```

What should be done in the unlikely event that the backslash operation involves a division by zero?
(c) Create `pagerank3(G)` by modifying `pagerank1` to use the power method instead of solving the sparse linear system. The key statements are:

```
while termination_test
   x = A*x;
end
```

What is an appropriate test for terminating the power iteration?
(d) Use your functions to compute the page ranks of the six-node example discussed in the text. Make sure you get the correct result from each of your three functions.
(e) Measure the execution time of each of your three functions on the `harvard500` data or some other large problem. Which function is the fastest? Why? If the times are too small to measure accurately, use something like

```
tic
for k = 1:100
   x = pagerank(U,G);
end
average_time = toc/100
```

2.24. Here is yet another function for computing PageRank. This version uses the power method, but does not do any matrix operations. Only the link structure of the connectivity matrix is involved.

```
function [x,cnt] = pagerankpow(G)
% PAGERANKPOW  PageRank by power method.
% x = pagerankpow(G) is the PageRank of the graph G.
% [x,cnt] = pagerankpow(G) counts the number of iterations.

% Link structure

[n,n] = size(G);
for j = 1:n
   L{j} = find(G(:,j));
   c(j) = length(L{j});
end

% Power method

p = .85;
delta = (1-p)/n;
x = ones(n,1)/n;
z = zeros(n,1);
cnt = 0;
while max(abs(x-z)) > .0001
   z = x;
   x = zeros(n,1);
   for j = 1:n
      if c(j) == 0
         x = x + z(j)/n;
      else
         x(L{j}) = x(L{j}) + z(j)/c(j);
      end
   end
   x = p*x + delta;
   cnt = cnt+1;
end
```

(a) How do the storage requirements and execution time of this function compare with the three `pagerank` functions from the previous exercise?
(b) Use this function as a template to write a function in some other programming language that computes PageRank.