Applied Numerical Analysis Using MATLAB

SECOND EDITION

Laurene V. Fausett

Contents

1.4 USING MATLAB

As the long history of numerical techniques indicates, numerical analysis does not require any particular computer resources. On the other hand, the scale and complexity of the problems that can be solved in a practical manner are strongly influenced (indeed, greatly expanded) by the availability of high-speed computers and efficient software implementation of numerical algorithms. Understanding the basic numerical methods and the issues involved in designing and analyzing them is a first step in the use of these techniques for real-world problems. Implementing the appropriate method, or using existing software, or a combination of the two approaches, is necessary to achieve the final goal.

MATLAB has been chosen as the programming environment for the presentation of the numerical techniques in this text for two main reasons. First, MATLAB provides outstanding graphing and programming capabilities, together with the ability to solve many types of problems symbolically as well. Second, MATLAB's underlying matrix structure makes the software especially useful for focusing on the aspects of various numerical techniques that can be described conveniently in vector form. This is a particularly valuable feature, because vectorization is an important approach to parallel computing. MATLAB programs are presented for each technique discussed in subsequent chapters. These programs can also be used as the basis for programming in other languages, if that is desired. A brief summary of the most basic features of MATLAB is presented here.

MATLAB originated in the late 1970s as a "matrix laboratory" for use in courses in matrix theory, linear algebra, and numerical analysis. Although today's MATLAB has vastly expanded capabilities, the basic data element is still an array, which does not require the declaration of either dimensions or variable types (integer, real, or complex). MATLAB is an interactive system; commands may be entered in the `Command Window` or by creating scripts or functions. Scripts and functions are known as `m-files` and are discussed later. First we summarize a few of the most basic MATLAB features that are useful for working in both the `Command Window` and `m-file` settings.

1.4.1 Command Window Computations

Simple computation may be carried out in the `Command Window` by entering an instruction at the prompt, much as you would on a calculator.

MATLAB variables

Variable names in MATLAB may consist of up to 31 characters, starting with a letter and followed by any combination of letters, digits, and underscores. Variable names are case sensitive. Punctuation marks and spaces may not be included in a variable name.

MATLAB treats all variables as matrices, although scalar quantities are not entered or displayed in array notation. A row vector is a 1-by-n matrix; a column vector is an n-by-1 matrix. Note that elements in MATLAB arrays are indexed starting from 1.

A row vector may be defined as `x = [ 1 2 3 ]` or `x = [ 1, 2, 3]`. The elements, enclosed between the left bracket and the right bracket, are separated by either commas or spaces.

A column vector may be given as `y = [ 4; 5; 6 ]`, `y = [ 4, 5, 6 ]'`, or

```
y = [ 4
      5
      6 ]
```

A new row may be indicated either by a new line or by a semicolon. The transpose of a matrix is indicated by a `'`, so the transpose of a row vector is a column vector.

A matrix may be entered either by placing each row of the matrix on a new line, or by using semicolons to separate the lines. Although the "all-on-one-line" approach saves space, the matrix is more difficult to visualize in the compact form. An example of a 3-by-4 matrix is

```
Z = [ 1  2  3  4
      5  6  7  8
      9 10 11 12 ]
```

or `Z = [ 1 2 3 4; 5 6 7 8; 9 10 11 12]`.

The elements of a matrix may be accessed as in standard matrix notation, e.g., `Z(2,3) = 7`. An entire row or column of a matrix may be accessed by using a colon instead of a specific element.

```
Z(2, : ) = [ 5  6  7  8];        Z(:, 3 ) = [ 3
                                              7
                                             11 ]
```

Creating Special Vectors

In many situations, we need an array with a particular structure. For example, an array (row vector) of the integers from 1 to 10 is created by the command

```
x = 1 : 10
```

and a vector of the numbers from a to b with spacing s is created by

```
x = a : s : b
```

In other cases, we wish to have a specified number of values, evenly distributed between the first and last element. If the desired spacing is linear, the MATLAB function `linspace` can be used. For example, the command to create a vector of 100 points, evenly spaced from 1 to n, inclusive, is

```
x = linspace(1, n)
```

The user can also specify the number of points (including the first and last values) with the command

```
x = linspace(a, b, npts)
```

Creating Special Matrices

MATLAB has a number of special matrices that are very useful in applications.

- `eye(3)`: The 3×3 identity matrix.
- `ones(3)`: A 3×3 matrix of all 1's.
- `ones(1, 4)`: A row vector of all 1's.
- `zeros(2,3)`: An array of all 0's.

Matrices can be combined by concatenation, as in the code

```
x = [ 1     2     3 ]
y = [ 5    10    15 ]
z = [ x     y ] = [1   2   3   5   10   15]
```

and the code

```
w =   [ x
        y ]
  =   [   1     2     3
          5    10    15 ]
```

Vector and Matrix Computation

Addition and subtraction of vectors or matrices of the same dimensions are defined in the usual way. For example, if **x** and **y** are the preceding row vectors, then standard matrix addition is the component-by-component sum:

```
x+y =  [ 1+5      2+10      3+15 ] = [ 7    12    18 ]
```

However, a special notation is required when component-by-component multiplication, division, or exponentiation is needed for vectors or matrices; such operations are indicated by a period immediately before the corresponding operation symbol. Thus, the component-by-component product of the vectors **x** and **y** is indicated as

```
x .* y =  [(1)(5)   (2)(10)   (3)(15)] = [5    20    45]
```

and the vector of the squares of the elements of vector **x** is

```
x .^ 2 =  [(1)^2   (2)^2   (3)^2] = [1    4    9]
```

Matrix Transpose

In MATLAB, the matrix transpose operation is denoted by the prime symbol, $'$; for a matrix **M** with complex elements, $\mathbf{M}'$ gives the complex-conjugate transpose of **M**; the transpose without conjugation is given by preceding the prime with a period, i.e., $\mathbf{M}.'$

Built-In Functions

MATLAB offers many common mathematical functions. Input to the trigonometric functions is in radians. The function `log(x)` is the natural logarithm; the base-10 logarithm is `log10(x)`.

Plotting

To graph a function in MATLAB, we create a vector (call it **x**) of the values of the independent variable that are to be graphed and a corresponding vector **y** of the values of the dependent variable. The graph is then generated by issuing the MATLAB command

```
plot(x, y)
```

The following set of commands generates the graph for Figure 1.1.

```
x = 0 : 0.01 : 2;          y = x.^2 -3;
plot(x,y);                 hold on
u = [0  2 ];               v = [0  0];
plot(u, v);    grid on,    hold off
```

The function is graphed first and the x-axis is plotted second, but the `hold on` command causes them to be drawn on the same set of axes. The first three lines compute the vectors **x** and **y** and plot the function (using straight lines between the indicated points); plotting **u** versus **v** gives a straight line along the x-axis. The grid lines on the graph are created using the `grid on` command. Two or three commands have been placed on each line to conserve space.

Output

A MATLAB command that is not terminated with a semicolon will display the result, together with the variable name (if there is one) or `ans`. A command followed by a semicolon will not display the result of the calculation.

The function `disp(variable)` can be used to display the value of a variable without the variable name.

To reduce the number of blank lines between results shown in the workspace, the command `format compact` is helpful.

MATLAB has several number-display formats; a few of the most useful are summarized here. The internal representation of a number is not changed when the display format is altered.

For fixed-point numbers,

`format short` displays 5 digits;

`format long` displays 16 digits.

For floating-point representation,

`format short e` gives 5 digits plus the exponent;

`format long e` gives 16 digits plus the exponent.

Two combination formats use fixed-point display unless a number is so small that floating point is necessary to show its value meaningfully.

`format short g`

`format long g`

For results that are ratios of small integers, use

`format rat`

1.4.2 M-Files

There are two types of programs (known as `m-files`) in MATLAB: scripts and functions. To generate an `m-files`, choose `New / m-file` from the `File` menu, enter the desired commands, and save the file. The differences between scripts and functions are summarized next.

Scripts

Scripts provide a set of MATLAB commands, comments, definitions of parameter values, plotting commands, and so on. A script that has been created and saved is executed by typing the file name at the MATLAB prompt in the `Command Window`, or using `save and run` or `run` from the `Debug` menu. As an example, the following script calculates the trapezoid-rule approximation to the integral of $1/x^3$ on the interval [1, 3], as in Example 1.3:

```
% S_Ex_1_3
a = 1;                  b = 3;
x = [a   b ],           y = x.^(-3)
I =(b-a)*(y(1)+y(2))/2
```

The first line is a comment, giving the name of the script; text following a percent sign (%) is treated as a comment. This script will display the vectors **x** and **y** and the approximate value of the integral I. The values of a and b are not displayed because they are followed by a semicolon.

Functions

A MATLAB function communicates with the MATLAB workspace through the variables passed into the function, the output variables it creates, and the use of global variables. A function is distinguished from a script by the fact that the first line of a function is of the form

```
function y = FunctionName(input arguments)
```

or

```
function [y, z] = FunctionName(input arguments)
```

where y (or the vector `[ y, z ]`) is the output returned by the function. The name of the function and the name of the file in which it is stored must be the same, except that the name of the file ends in ".m".

To illustrate the use of a (user-created) MATLAB function, consider the following code for finding the approximate integral of $f(x) = 1/x^3$, using the basic trapezoid rule:

```
function t = TrapEx(a, b)
t = (b-a)*(a^(-3)+b^(-3))/2;
```

The function, which is created as an m-file, and saved as `TrapEx.m`, can then be executed from the `Command Window` by entering the appropriate function call at the prompt. The response from MATLAB appears on the next line. For example:

```
t = TrapEx(1, 3)
t =          1.037
```

This function is still much more specific than we would like. A more useful function for the basic trapezoid rule allows us to specify the function to be integrated as an input argument.

In many cases, the function to be integrated can be given as an `inline function`. If the integrand is given as an `inline function`, the function for the basic trapezoid method looks very much like the mathematical statement of the formula. We begin by creating and saving the function `BasicTrap`:

```
function q = BasicTrap(f, a, b)
% compute integral of function f, which is an inline function
q = (b-a)*(f(a)+f(b))/2;
```

We can use the function `BasicTrap` by defining the function to be integrated as an inline function in the `Command Window`, and also defining the values of a and b, and then calling the `BasicTrap` function, as in this example:

```
f = inline('x.^(-3)');
a = 1;     b = 2;
q = BasicTrap(f, a, b)
```

Alternatively, the function to be integrated, and the interval of integration, can be defined in the call to `BasicTrap` by entering (for example)

```
q = BasicTrap(inline('x.^(-3)'), 1, 2)
```

Another approach is to define the function to be integrated in the same manner as the function for the basic trapezoid method. We illustrate this with the following two functions, the first of which, `MyFunc`, defines the function to be integrated, and the second of which, `BasicTrap`, implements the basic trapezoid rule. Each function is saved in a file of the same name.

```
function y = MyFunc(x)
y = x.^(-3);

function q = BasicTrap2(f, a, b)
ya = feval(f,a);          yb = feval(f,b);
q = (b-a)*(ya+yb)/2;
```

Notice that if the integrand is defined in this form, it must be evaluated using `feval` in the function `BasicTrap2`. This form of the `BasicTrap2` function will also work with inline functions.

To use the function `BasicTrap2`, we must specify the name of the function to be integrated, enclosed in single quotes:

```
q = BasicTrap2('MyFunc', 1, 3)
```

In the chapters that follow, MATLAB functions or scripts are given for each of the numerical methods presented. A brief discussion of MATLAB's built-in functions for solving numerical problems is also included at the end of each chapter.

1.4.3 Programming in MATLAB

We now consider some of the essential parts of a computer program, and discuss briefly how these structures may be implemented in MATLAB.

Starting and Finishing

A computer program to implement a numerical method must specify the input values for various parameters, the definitions of other functions used in the procedure, and the computed value(s) of the variable(s). Together these parts of the program are often called the I/O (input/output). Depending on the programming language used for implementing an algorithm, and the level of generality desired for the program, input values may be defined in the program, or passed as arguments in the call to the program.

In many programming languages, it is necessary to specify a type for each variable used in the program (integer, real, complex, and so on). In some languages there is a default type (which may depend on the name of the variable); in other languages every variable type must be declared. Also, again depending on the language, it may be necessary to specify the dimensions of any arrays (vectors or matrices) before they are used in the program. MATLAB treats all variables as matrices (of whatever dimension is needed) and does computation in complex arithmetic whenever appropriate.

Computation

The main body of a computer program consists of statements that perform the computations to implement the algorithm. In many cases these computations are performed repeatedly, either for a specified range of values for some variable, or until some condition is satisfied. Such repeated computation is usually controlled by a looping structure. MATLAB has two such structures, designated as "for-loops" and "while-loops."

There are also cases in which a statement (or group of statements) is to be executed under certain conditions, and not under other conditions. MATLAB has a variety of "if-structures" as well as "break" structures that allow escape from inside a loop, under certain conditions. The relational and logical operators that are commonly used in determining whether a "while-loop" or any of the "if-structures" is to execute are listed inside the back cover.

For-Loops

In a *for-loop*, the computations are carried out for all values of the looping index, from the beginning value to the ending value, unless some condition occurs (conditional execution is discussed next) that causes the process to exit the loop. In general, if such an exit occurs, the value of the looping index is not preserved outside the loop. A simple *for-loop* in MATLAB is given as

```
for i = 1:k
    commands
end
```

In general, the `for` statement may have the form `for i = array`.

While-Loops

In a *while-loop*, the computations are continued as long as the specified condition is true. Care should be taken that the computations will eventually terminate. The commands between the `while` and `end` statements are executed as long as the evaluation of `expression` is true (nonzero).

```
while expression
    commands...
end
```

Conditional Execution

Another key feature of any computer programming environment or language that is suitable for implementing numerical methods is the ability to control execution of one or more statements, depending on certain conditions. The *if* statement executes the statements that follow if the condition is true. The *break* statement causes the program to exit the loop in which the break occurs. The *return* or *error* statements cause the program to terminate (with an error message in the case of the *error* statement).

If

The simplest *if-else-end* construction allows a command or group of commands to be executed if a specified expression is true. The statements are skipped if the expression is false. It is indicated as

```
if expression
   commands
end
```

If there are more alternatives, the form is

```
if expression1
    commands...
elseif expression2
    commands...
elseif ...
...
else
    default commands
end
```

Break and Error

The `break` command causes MATLAB to jump outside the loop in which it occurs. The `error` command aborts function execution, displays a character string in the `Command Window`, and returns control to the keyboard.

1.4.4 Matrix Multiplication

There are many low-level (basic) computations that MATLAB can handle without the user having to program them directly. However, to illustrate the various ways in which fundamental operations can be performed, we consider the question of how to compute the product of two matrices. The three basic approaches are the "dot-product form," the "saxpy form" (scalar a $\mathbf{x}$ plus $\mathbf{y}$), and the "outer-product form." (See Golub and Van Loan, 1996.)

Each of these approaches can be implemented in MATLAB using nested loops, or using MATLAB's vector capabilities, in terms of the appropriate combination of rows or columns of the matrices $\mathbf{A}$ and $\mathbf{B}$. Remember that $\mathbf{A}(i,:)$ indicates the i^{th} row of the matrix $\mathbf{A}$, while $\mathbf{B}(:,j)$ gives the j^{th} column of the matrix $\mathbf{B}$.

Dot-Product Form

In this form, the i,j element of the product is the dot product of the i^{th} row of $\mathbf{A}$ with the j^{th} column of $\mathbf{B}$. If $\mathbf{A}$ and $\mathbf{B}$ are 2×2 matrices:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}$$

The following MATLAB code computes the product $\mathbf{C} = \mathbf{AB}$ (where $\mathbf{A}$ is a $m \times p$ matrix and $\mathbf{B}$ is an $p \times n$ matrix), with the loops fully expanded. Matrix $\mathbf{C}$ is initialized to a zero matrix of the appropriate dimensions, although this is not actually necessary in MATLAB.

```
C = zeros(m,n)
for i = 1:m
    for j = 1:n
        for k = 1:p
            C(i, j) = C(i, j)+A(i, k)*B(k, j)
        end
    end
end
```

Using MATLAB's ability to access the rows of $\mathbf{A}$ and the columns of $\mathbf{B}$, the inner loop can be replaced by the dot product of the i^{th} row of $\mathbf{A}$ and the j^{th} column of $\mathbf{B}$, as follows:

```
for i = 1:m
    for j = 1:n
        C(i,j) =  A(i, :)*B(:, j)
    end
end
```

Saxpy Form

In this form, each column of the product is viewed as a linear combination of the columns of **A**. If **A** and **B** are 2×2 matrices:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \left[b_{11} \begin{bmatrix} a_{11} \\ a_{21} \end{bmatrix} + b_{21} \begin{bmatrix} a_{12} \\ a_{22} \end{bmatrix}, \quad b_{12} \begin{bmatrix} a_{11} \\ a_{21} \end{bmatrix} + b_{22} \begin{bmatrix} a_{12} \\ a_{22} \end{bmatrix} \right]$$

The MATLAB implementation of the Saxpy form of matrix multiplication (where again, **A** is an $m \times p$ matrix and **B** is a $p \times n$ matrix) begins by initializing **C** as the zero matrix of appropriate dimensions.

```
C = zeros(m,n)
for j = 1:n
    for k = 1:p
        C(:,j) = C(:,j)+A(:, k)*B(k, j)
    end
end
```

Outer-Product Form

In this form, the product is computed as the sum of the outer product of each column of **A** with the corresponding row of **B**. If **A** and **B** are 2×2 matrices:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11} \\ a_{21} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \end{bmatrix} + \begin{bmatrix} a_{12} \\ a_{22} \end{bmatrix} \begin{bmatrix} b_{21} & b_{22} \end{bmatrix}$$

The fully expanded form of the outer-product formulation (where **A** is an $m \times p$ matrix and **B** is a $p \times n$ matrix) computes $\mathbf{C} = \mathbf{AB}$ as a sum of outer products of each column of **A** with the corresponding row of **B**. Again, we begin by initializing **C** to be the zero matrix of appropriate dimensions.

```
C = zeros(m,n)
for k = 1:p
    for j = 1:n
        for i = 1:m
            C(i, j) = C(i, j)+A(i, k)*B(k, j)
        end
    end
end
```

A more compact form accesses the columns of **A** and the rows of **B** using colon notation.

```
C = zeros(m,n)
for k = 1:p
    C = C+ A(:, k)*B(k, :)
end
```

The following MATLAB function implements Gaussian elimination with row pivoting. Although it actually performs the row interchanges, more sophisticated programs keep track of the indices of the pivots selected at each stage without carrying out the row interchanges. Implicit row pivoting is illustrated in Sec. 4.4.

Gaussian Elimination with Pivoting

```
function  X = GaussPivot(A, B)
%  Inputs :
%     A is the n-by-n coefficient matrix
%     B is the n-by-p right-hand-side matrix
%  Outputs :
%     X is the n-by-m solution matrix
[n,  n1] = size(A);
[m1, p ] = size(B);
% program requires that n = n1 = m1
if n ~= n1,
   error ('A must be square'),
end
if n ~= m1,
    error ('A and B must have same number of rows'),
end
C = [ A  B ]                    % form augmented matrix
for i = 1:n-1
    [ pivot, k] = max(abs(C(i:n, i)));
    % k is position of pivot, relative to row i
    % do not check rows above current row
    if k > 1
        temp1        = C(i, :);
        C(i, :)      = C(i+k-1, :);
        C(i+k-1, :) = temp1;
    end
    m(i+1:n, i) = -C(i+1:n, i)/C(i, i);
    C(i+1:n, :) =  C(i+1:n, :) + m(i+1:n, i)*C(i, :);
end
%    back substitution
X(n, 1:p) = C(n, n+1:n+p)/C(n, n);
for i = n-1 : -1 : 1
    X(i, 1:p) = (C(i, n+1:n+p) - C(i, i+1:n)*X(i+1:n, 1:p)) /C(i, i);
end
```

3.2 GAUSS-JORDAN

Gauss-Jordan elimination begins by performing row reduction to zero out the below-diagonal elements (as in standard Gaussian elimination); it then performs row reduction (starting with the last row as pivot) to zero out the above-diagonal elements (back elimination). Finally, the rows are scaled so that diagonal elements are ones. This transforms the matrix into reduced row echelon form (RREF).

The elimination process is carried out in two steps for computational efficiency. Applying all the backward elimination after the forward elimination stages are completed is more efficient because each forward elimination step involves only elimination of the elements in one column of matrix **A**. The resulting computation for back elimination is of the same order as that for back substitution.

In the following function for Gauss-Jordan elimination, only the columns in which nonzero elements could appear are updated during the backward-elimination phase.

Gauss-Jordan with Row Pivoting

```
function  X  = Gauss_Jordan_P(A, B)
[n, n1] = size(A);      [m1, p ] = size(B);      C = [ A  B ]
% program requires that n = n1 = m1
if n ~= n1, error ('A must be square'), end
if n ~= m1, error ('A and B must have same number of rows'), end
for i = 1:n-1
    [ pivot, k] = max(abs(C(i:n, i)));
    % k is position of pivot, relative to row i, check rows i to n
    if k > 1
        temp1 = C(i, :);  C(i, :) = C(i+k-1, :);  C(i+k-1, :) = temp1;
    end
    m(i+1:n, i) = -C(i+1:n, i)/C(i, i);
    C(i+1:n, :) =  C(i+1:n, :) + m(i+1:n, i)*C(i, :);
end
for i = n : -1 : 2
    for k = i-1 : -1 : 1                % update rows above pivot row
        m = -C(k, i)/C(i, i);
        C(k, i)        = C(k, i) + m*C(i, i);
        C(k, n+1:n+p) = C(k, n+1:n+p) + m*C(i, n+1:n+p);
    end
end
for i = 1:n                             % scaling
    C(i, n+1:n+p) = C(i, n+1:n+p)/C(i, i);
    C(i, i) = 1;
end
X = C( : , n+1:n+p);
```

Finally, we scale the fifth equation, use it to eliminate x_5 in the last equation, and scale the last equation:

$$\begin{aligned}
x_1 - \frac{1}{2}x_2 &= \frac{1}{2}\\
x_2 - \frac{2}{3}x_3 &= \frac{1}{3}\\
-x_4 &= -1\\
-x_3 + 2x_4 - x_5 &= 0\\
x_5 - \frac{1}{2}x_6 &= \frac{1}{2}\\
x_6 &= 1
\end{aligned}$$

Solving by back substitution yields

$$\begin{aligned}
x_6 &= 1\\
x_5 &= \frac{1}{2} + \frac{1}{2}x_6 = 1\\
x_4 &= 1 \quad \text{(computed previously)}\\
x_3 &= 2x_4 - x_5 = 1 \quad \text{(skipped previously)}\\
x_2 &= \frac{1}{3} + \frac{2}{3}x_3 = 1\\
x_1 &= \frac{1}{2} + \frac{1}{2}x_2 = 1
\end{aligned}$$

3.4 FURTHER TOPICS

We conclude this chapter with some discussion of MATLAB's methods for solving linear systems, a method for estimating the condition of a matrix, and an introduction to the use of iterative refinement to reduce the residual of the solution of a linear system. Iterative methods for linear systems are discussed in Chapter 6.

3.4.1 MATLAB's Methods

There are two division symbols in MATLAB, the forwardslash (/) and the backslash (\ ). It is useful to think of each as indicating multiplication by the inverse of the quantity under the slash. In this way of interpreting the symbols, $a/b = a(b^{-1})$ and $c\backslash d = c^{-1}(d)$. Since scalar multiplication is commutative, we can write a fraction using either of these symbols. However, matrix multiplication is not commutative, and backslash division gives us a convenient way of solving the linear system $\mathbf{Ax} = \mathbf{b}$. The solution of $\mathbf{Ax} = \mathbf{b}$ is $\mathbf{x} = \mathbf{A}\backslash\mathbf{b}$, which is suggestive of $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$, where it is important that the implied multiplication by the inverse of $\mathbf{A}$ is from the left. The backslash division operator is also called the matrix left-division operator.

If **A** is an $n \times n$ matrix and **b** is a column vector with n components (or a matrix with several such columns), then $\mathbf{x} = \mathbf{A} \backslash \mathbf{b}$ is the solution of the equation $\mathbf{Ax} = \mathbf{b}$, computed by Gaussian elimination (not computed by multiplying by the inverse of **A**). A warning message is printed if **A** is badly scaled or nearly singular. Information about MATLAB's operations can be found using the command `help matlab:ops`. The matrix left-division operator is implemented in the function `mldivide`, which provides a brief description of its approach in the comments at the beginning of the function.

Over- and Underdetermined Systems

The left-division operator can also be used when **A** is $n \times m$ ($n \neq m$) to obtain a solution of the over- or underdetermined system. A system is underdetermined if there are more unknowns than equations and overdetermined if there are fewer unknowns than equations. In each case, it is the number of linearly independent equations that is important. The concept of linear independence is discussed in standard texts on linear algebra.

Reduced Row Echelon Form

The MATLAB function `rref` transforms a matrix to reduced row echelon form.

The command `R = rref(A)` produces the reduced row echelon form of A using Gauss-Jordan elimination with partial pivoting. The default tolerance is used in the tests for negligible column elements.

Calling the function as `[R,j] = rref(A)` also returns a vector j such that `r = length(j)` is this algorithm's idea of the rank of **A**. The elements of vector **j** indicate which variables are the pivot variables in a linear system $\mathbf{Ax} = \mathbf{b}$. The vectors $\mathbf{A}(:, j)$ form a basis for the range of **A**. $\mathbf{R}(1 : r, j)$ is the $r \times r$ identity matrix. The following small example illustrates the use of `rref`.

```
A = [  1   1  -3   5
       3  -2   1   0
       4  -2   0   2 ]
[R,j] = rref(A)
R =
     1     0    -1     2
     0     1    -2     3
     0     0     0     0
j =
     1     2
```

Calling the function as `[R,j] = rref(A,tol)` causes the function to use the given tolerance in the rank tests.

Round-off errors may cause this algorithm to compute a different value for the rank than that found by the functions `rank`, `orth`, and `null`.

It is also possible to solve the linear system $\mathbf{Ax} = \mathbf{b}$ by using MATLAB's function `inv` for finding a matrix inverse; however, multiplication by a matrix inverse is usually not the best way to solve a linear system.

3.4.2 Condition of a Matrix

The condition number of a nonsingular matrix $\mathbf{A}$ is defined in terms of the norm of the matrix $\|\mathbf{A}\|$ and the norm of the inverse matrix, $\|\mathbf{A}^{-1}\|$. Since there are several different matrix norms, there are several different condition numbers for a matrix. Some matrix norms are derived from (or are subordinate to) a vector norm.

The three most common vector norms are

$$\|x\|_1 = |x_1| + |x_2| + \cdots + |x_n| \qquad \text{(Manhattan or taxi-cab norm)}$$

$$\|x\|_2 = (|x_1|^2 + |x_2|^2 + \cdots + |x_n|^2)^{1/2} \qquad \text{(Euclidean norm)}$$

$$\|x\|_\infty = \max |x_i| \qquad \text{(Chebyshev norm)}$$

In general, if p is a real number ≥ 1, the p-norm is

$$\|x\|_p = (|x_1|^p + |x_2|^p + \cdots + |x_n|^p)^{1/p}$$

A matrix norm that is subordinate to a vector norm is defined as

$$\|\mathbf{A}\| = \sup \|\mathbf{A}x\|/\|x\|$$

where the sup is taken over all nonzero vectors.

The matrix norm that is subordinate to the vector p-norm is denoted $\|\mathbf{A}\|_p$. The matrix norms $\|\mathbf{A}\|_1$, $\|\mathbf{A}\|_2$, and $\|\mathbf{A}\|_\infty$ have several useful properties, which can be established directly from the definitions.

$\|\mathbf{A}\|_1$ = maximum of column sums

$$= \max_j(|a_{1j}| + |a_{2j}| + \cdots + |a_{nj}|) \qquad = \max_j \sum_{i=1}^{n} |a_{ij}|$$

$\|\mathbf{A}\|_\infty$ = maximum of row sums

$$= \max_i(|a_{i1}| + |a_{i2}| + \cdots + |a_{in}|) \qquad = \max_i \sum_{j=1}^{n} |a_{ij}|$$

$\|\mathbf{A}\|_2$ = maximum eigenvalue of $(\mathbf{A}^H\mathbf{A})^{1/2}$

This norm is also known as the spectral norm.

$\mathbf{A}^H$ is the complex conjugate of the transpose of $\mathbf{A}$. The eigenvalues of $\mathbf{A}^H\mathbf{A}$ are nonnegative; the square roots of these eigenvalues are the singular values of $\mathbf{A}$.

There are also important matrix norms that are not subordinate to any vector norm. The most important of these is the Euclidean (or Schur, or Frobenius) norm

$$\|\mathbf{A}\|_E = \left(\sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij}^2 \right)^{1/2}$$

(See Wilkinson, 1965, for further discussion.) Note that the Euclidean matrix norm is not subordinate to the Euclidean vector norm (or to any other vector norm).

Estimating the Condition of a Matrix

In a fairly broad sense, a computational problem is "ill-conditioned" if its solutio
is very sensitive to small changes in the parameters that define the problem. Fo
problems involving a matrix (solving a linear system, finding the eigenvalues
the matrix, and so on), it may be useful to describe this sensitivity in terms of
number, called the condition number of the matrix.

For any matrix norm, we define the condition number $k(\mathbf{A}) = \|\mathbf{A}\|\|\mathbf{A}^{-1}$
Although the actual value found for the condition number will in general be differe
for different norms, a matrix that is ill-conditioned in one norm will also be i
conditioned in all other matrix norms.

However, exact computation of the condition number of a matrix (using a
of the matrix norms) is computationally quite expensive. The following algorith
(Hager, 1984) gives an estimate of the condition number $k(\mathbf{A})$ using the 1-norm
based on an estimate of the 1-norm of $\mathbf{A}^{-1}$.

Estimate of Condition Number of A

```
A = [  60   30   20
       30   20   15
       20   15   12 ];
[ n  nn ] = size(A);    b( 1:n, 1 )= 1/n;   p = 0;   kmax = 3;
for kk = 1:kmax
   x = A\b
   if ( norm(x, 1) - p <= 0), disp('converged, norm x > p'), break
   else
       p = norm(x, 1)
   end
   for k = 1: n
       if x(k, 1) >= 0
          y(k, 1) = 1;
       else
          y(k, 1) = -1;
       end
   end
   z = A'\y                          %  Solve A'z = y
   [zz, j] = max(abs(z));            % index such that | z(j) | is maximal
   if abs(z(j))<= z'*b
       disp('converged, abs(z(j))<= zT*b');,    break
   else
       b( 1:n, 1 )  =  0;            b(j, 1) = 1;
   end
end
p
c = p* norm( A,1)                    % estimate of norm of inv(A)
                                     % || A ||1 is max absolute column sum
```

4.2 MATRIX TRANSFORMATIONS

In the next section we consider the problem of factoring a square matrix $\mathbf{A}$ into the product of an orthogonal matrix $\mathbf{Q}$ and an upper (right) triangular matrix $\mathbf{R}$. Thus we seek to write $\mathbf{A} = \mathbf{QR}$, where $\mathbf{R}$ is right triangular, $\mathbf{Q}$ is orthogonal (so that $\mathbf{Q}^{-1} = \mathbf{Q}^T$), and both are real.

In the analysis of LU factorization in Sec. 4.1.1 we saw that the elementary row operations used in Gaussian elimination can be described in terms of multiplication by certain simple matrices (which we denoted as $\mathbf{M}$ in the analysis) even though that is not the computationally efficient way to perform those operations.

In this section we consider two other matrix transformations. These are also convenient to describe in terms of multiplication by a matrix of a specific form, but are more efficiently carried out in practice without actually forming the transformation matrix or performing the matrix multiplication. As with Gaussian elimination, we are interested in transforming a given matrix to a matrix which has zeros in certain specified locations. The important characteristic of each transformation matrix we consider in this section is that each is an orthogonal matrix, so its inverse is just the transpose of the matrix.

First we investigate Householder transformations. In terms of matrix multiplication, this can be described as multiplication by a Householder matrix, which is a matrix of the form

$$\mathbf{H} = \mathbf{I} - 2\mathbf{w}\mathbf{w}^T$$

where $\mathbf{w}$ is a unit column vector (i.e., $||\mathbf{w}||_2 = \mathbf{w}^T\mathbf{w} = 1$). It can also be written in the form

$$\mathbf{H} = \mathbf{I} - \frac{2}{\mathbf{v}'\mathbf{v}}\mathbf{v}\mathbf{v}'$$

where $\mathbf{v}$ is a nonzero (column) vector.

By the appropriate choice of the vector $\mathbf{v}$ we can form $\mathbf{H}$ so that

$$\mathbf{Hx} = \pm||\mathbf{x}||\mathbf{e}_1$$

i.e., all components of the product are zero except the first.

This is accomplished by taking

$$\mathbf{v} = \mathbf{x} \pm ||\mathbf{x}||_2\mathbf{e}_1$$

In applications it is often the case that we need to zero-out all of the elements of the column of a matrix, below a certain point. In such cases we take the vector $\mathbf{x}$ to be the lower portion of the given column.

The second type of transformation is called a Givens rotation (or transformation). It is described as multiplication by an orthogonal matrix, which is the identity matrix with four additional nonzero elements: $g_{ii} = g_{jj} = c$, and $g_{ij} = -s$, $g_{ji} = s$ (with $i \leq j$ and $c^2 + s^2 = 1$). By selecting the appropriate values for c and s, a Givens rotation can be used to reduce a specific matrix element to zero.

The key characteristic for Householder matrices and Givens rotation matrices is that they are orthogonal, so we have $\mathbf{H}\mathbf{H}^T = \mathbf{I}$ and $\mathbf{G}\mathbf{G}^T = \mathbf{I}$. However, as we will see, Householder matrices are also symmetric, so in fact we have $\mathbf{H}\mathbf{H} = \mathbf{I}$.

4.2.1 Householder Transformation

A Householder matrix such that $\mathbf{Hx} = \pm\|\mathbf{x}\|\mathbf{e}_1$ can be written as

$$\mathbf{H} = \mathbf{I} - \frac{2}{\mathbf{v}'\mathbf{v}}\mathbf{v}\mathbf{v}'$$

where

$$\mathbf{v} = \mathbf{x} + \text{sign}(x_1)\|\mathbf{x}\|_2\mathbf{e}_1$$

We formulate this as an algorithm as follows:

```
g = norm(x) = (x_1^2 + ... + x_n^2)^(1/2)   % g = ||x||
p = sign(x_1)
v = [ x_1 + p*g,  x_2, ..., x_n ]
s = norm(v) = (2*g*(g+p*x_1)^(1/2)          % after a little algebra
w = v/s
```

Now, in practice we need to zero-out all elements of a vector from position $k+1$ through n. To do this, we work on the k-to-n components of $\mathbf{x}$; by initializing the vector $\mathbf{w}$ to 0 the first $k-1$ components are ignored. Note that if $s = 0$, all of the required components are already zero, and vector $\mathbf{w}$ should be set to the zero vector, so that $\mathbf{H} = \mathbf{I}$. The following function actually forms the Householder matrix (although this is not usually done in practice).

```
function H = Householder(x, k)
[n, nn] = size(x);
w = zeros(n, 1);                    % initialize w
g = norm(x(k : n));                 % work on relevant components of x
p = sign(x(k));                     s = sqrt(2*g*(g+p*x(k)));
v(k) = (x(k) + p*g)/s;              w(k+1:n) = x(k+1:n)/s;
H = eye(n) - 2*w*w';
```

To find $\mathbf{HA}$ efficiently, we do not actually construct the matrix $\mathbf{H}$. Instead, we define $\mathbf{u} = \mathbf{A}^T\mathbf{w}$, so that $\mathbf{u}^T = \mathbf{w}^T\mathbf{A}$, and compute

$$\begin{aligned}\mathbf{HA} &= (\mathbf{I} - 2\mathbf{w}\mathbf{w}^T)\mathbf{A} \\ &= \mathbf{A} - 2\mathbf{w}\mathbf{w}^T\mathbf{A} \\ &= \mathbf{A} - 2\mathbf{w}\mathbf{u}^T\end{aligned}$$

Computation of $\mathbf{u}$ requires the multiplication of a matrix and a vector, that of $\mathbf{wu}'$ requires the outer product of two vectors, and that of $\mathbf{A} - \mathbf{wu}'$ requires the subtraction of one matrix from another. Altogether these steps require $4n^2$ flops if $\mathbf{A}$ is an $n \times n$ matrix and $\mathbf{w}$ is $n \times 1$.

On the other hand, explicit formation of the matrix $\mathbf{H} = \mathbf{I} - 2\mathbf{ww}'$ requires the outer product of a vector with itself and multiplication of the matrices $\mathbf{H}$ and $\mathbf{A}$, for a total of $2n^3$ flops. This is an order of magnitude more computational effort than it takes to compute $\mathbf{A} - \mathbf{wu}'$.

In a similar manner, to find $\mathbf{AH}$ efficiently, use

$$\mathbf{AH} = \mathbf{A}(\mathbf{I} - 2\mathbf{w}\mathbf{w}^T) = \mathbf{A} - 2\mathbf{A}\mathbf{w}\mathbf{w}^T$$

Efficient computation of $\mathbf{HAH}$ is considered in connection with the discussion of similarity transformation to Hessenberg form.

Transformation to Upper Triangular Form

The following function converts the general matrix **A** to the upper triangular matrix **B**, by computing **B** = **HA** (without actually constructing the Householder matrices). Householder transformations are applied to columns 1 through $n-1$ of matrix **A**. In the k^{th} column, we zero-out positions $k+1$ through n. This function zeros-out the first components of **x** rather than initializing the vector **w** to zero and restricting the components of **x** that are used in computing $g = \text{norm}(\mathbf{x})$.

The matrices are shown schematically below, with blanks indicating entries which are zero.

```
x  x  x  x  x
x  x  x  x  x
x  x  x  x  x
x  x  x  x  x
x  x  x  x  x

A
```

```
x  x  x  x  x
   x  x  x  x
      x  x  x
         x  x
            x

B
```

Householder Transformation to Triangular

```
function B = House_to_triang(A)
[n, nn] = size(A);
for k = 1:n-1
    x = A(:,k)              % work on the kth column of A
    for j = 1:k-1
        x(j) = 0            % work with components k...n
    end
    g = norm(x);
    p = sign(x(k));
    s = sqrt(2*g*(g+p*x(k)));
    if s ~= 0
        w = x/s
        w(k) = (x(k) + p*g)/s;
        u = A'*w
        B = A - 2*w*u';
    end
    A = B
end
```

The next example illustrates the use of Householder transformations to convert a matrix to upper triangular form. In Sec. 4.3.1 we will see that essentially the same process can be used to find the QR factorization of matrix **A**.