Notes on Computational Mathematics: Matlab

Robert L. Higdon
Department of Mathematics
Oregon State University
Corvallis, Oregon 97331-4605

Revised April 1996

Introduction

These notes were originally developed for a course in computational mathematics given in the Department of Mathematics at Oregon State University. The goals of the course are to give an introduction to some standard mathematical software packages and to describe some mathematical topics that can be illuminated by computational examples and experimentation. The present notes were developed for the portion of the course that is concerned with Matlab.

Matlab is a software package for numerical computation and graphics developed by The MathWorks, Inc., `http://www.mathworks.com` , (508) 647-7000. MATLAB is a trademark of The MathWorks, Inc. Much more information about this package can be obtained from the *MATLAB User's Guide* and the *MATLAB Reference Guide*, which are available from The MathWorks. These notes are based on Version 4.2 of Matlab.

Matlab fundamentals

1. Beginning and ending a session
2. The `help` and `demo` facilities
3. Variables and expressions
4. Numbers
5. Command-line editing
6. Listing your variables
7. Executing operating system commands while in Matlab
8. Directories and search path
9. Printing
10. Saving your variables
11. Loading numbers from external files

1. Beginning and ending a session

To begin a Matlab session, type `matlab` at the prompt from the operating system, and then press the `return` key. While you are in Matlab, enter commands at the prompt `>>` .

Occasionally you may enter a command and then decide that you want to stop execution of that command. For example, the command may be taking a long time to execute, and you might not want to wait that long. To stop execution, type `ctrl c` ; that is, hold down the `control` key and press `c` .

To end a Matlab session, type `quit` at the Matlab prompt. It is possible to save the values of the variables that are created during the session; this is discussed later.

2. The `help` and `demo` facilities

Matlab has an extremely good on-line help facility. If you type `help` at the Matlab prompt, Matlab will display a list of help topics. One of these items, for example, is `matlab/plotxy - Two dimensional graphics.` ; if you then type `help plotxy` , you will see a list of functions related to plotting two-dimensional data. One of the items in this list is `plot - Linear plot.` ; if you type `help plot` , the result is a description of how to use the `plot` function.

If you type `demo` at the Matlab prompt, you will receive a menu that lists several demonstrations of the capabilities of Matlab. These demonstrations are very informative.

3. Variables and expressions

Variables can be assigned values that are either scalars, vectors, or matrices.

Scalar variables can be assigned values by using an equal sign, e.g., `x = 1` . The usual arithmetic operations on scalars are indicated by the following symbols: `+` for addition, `-` for subtraction, `*` for multiplication, `/` for right division, `\` for left division, `^` for exponentiation. For example, `1/4` is the same as `4\1`, and `x^2` denotes the square of `x`.

If you type a semicolon `;` at the end of a command, Matlab will not print the output of that command.

Matlab is case-sensitive. For example, `x` and `X` are not the same.

4. Numbers

The format of numerical output can be selected by the `format` command. Any of the following commands can be typed at the Matlab prompt `>>` , and the form of the numerical output is determined as indicated. For information about other formats, consult the help facility by typing `help format` at the Matlab prompt.

`format`	Default. Save as `format short` .
`format short`	Fixed-point format with 4 digits after the decimal place; converts to exponential form for large numbers.
`format short e`	Exponential format with 5 digits.
`format long`	Fixed-point format with 14 digits after the decimal point.
`format long e`	Exponential format with 15 digits.
`format rat`	Approximate by ratios of small integers.

Large and small numbers can be entered in exponential format. For example, `1e6` is the same as 10^6, and `1e-6` is the same as 10^{-6}.

The imaginary unit is denoted by both `i` and `j`. Thus `i*i` and `j*j` are both equal to -1.

The permanent variable `eps` is initially the distance from 1.0 to the next largest floating-point number that can be stored in the machine you are using. It gives a measure of the precision that is being used in floating-point computations.

5. Command-line editing

The up-arrow key enables you to display previous command lines on the current line. This is very useful if you need to correct a typographical error and re-execute a command, or if you need to execute a command that is very similar to one executed earlier. To edit a command line, use the left-arrow and right-arrow keys to move back and forth through a line, use the `delete` and/or `backspace` keys to delete text, and type at the cursor to insert text.

If you enter some characters on the current command line, and if you then press the up-arrow key, Matlab will display the most recent line that begins with those characters. If there is no such line, the preceding line is displayed.

Suppose that a command is too long to fit on one line. Finish the current line with an ellipsis consisting of at least three periods (`...`), press the `return` key, and continue on the following line.

Several short commands can be placed on one line. In this case, if you want the output of a command to be printed, end it with a comma; if you want to suppress printing of a command, end it with a semicolon.

An alternative to command-line editing is to write Matlab statements into a file and then execute the file. This will be discussed later. A variation on this method is applicable if you are working in a window system. Have one window open to the file and another window open to Matlab, and use the mouse to copy-and-paste from the file window into the Matlab window.

The command `clc` clears the Matlab command window and moves the cursor to the top of that window.

6. Listing your variables

The command `who` produces a list of all of the variables that are currently defined during the Matlab session. The command `whos` is similar to `who`, but it also gives the dimensions of each variable (when regarded as a matrix) and total memory used.

The command `clear` clears all variables, and the command `clear x` clears the variable `x`.

7. Executing operating system commands while in Matlab

It is possible to execute operating system commands while you are working in Matlab. At the Matlab prompt, type `!` and follow it with the command that you want to execute. For example, if you are using a machine that runs the Unix operating system, and if you want to execute the Unix command `ps -aux` , you can type `!ps -aux` at the Matlab prompt `>>` . (If you are intending to execute some Matlab commands that will consume a lot of computing time, it would be a good idea to execute `ps -aux` first in order to see how busy the machine is.)

If you are working with the Unix operating system, some other commands you might want to use are the following: `pwd` (print your current working directory), `ls` (list the contents of that directory), and `cd directoryname` (change directory to `directoryname`). These commands are built into Matlab; typing `pwd` at the Matlab prompt is equivalent to `!pwd`, and `ls` is equivalent to `!ls`. However, `!cd directoryname` does not have any effect, whereas `cd directoryname` works as expected.

The Matlab command `dir` is equivalent to `ls`, the Matlab command `delete filename` is equivalent to `!rm filename` on a Unix system, and the Matlab command `type filename` is equivalent to `!cat filename` on a Unix system.

8. Directories and search path

Occasionally you may need to execute Matlab programs that are contained in files external to Matlab. The programs should be located in files that either are in your current working directory or are in a directory contained in Matlab's "search path". The latter is a list of directories that Matlab will search when it is asked to read the contents of a file. To see your current search path, type `path` at the Matlab prompt. Most of the directories listed may be directories that contain the Matlab system, but there may be at least one directory listed which is accessible to the user. You could use this directory to store your files related to Matlab.

9. Printing

One way to obtain printed output from a Matlab session is to use the `diary` command to record a portion of the session, and then print the file that contains the record.

The diary facility is invoked by typing `diary filename` at the Matlab prompt `>>` . All subsequent activity at the screen is copied into a file named `filename`; this file will be located in your current working directory, except if `filename` is a path to a file in a different directory. If a file name is not specified, the output is written to a file named `diary`, which will be located in your current working directory. The command `diary off` suspends the diary; `diary on` turns it back on; the command `diary` by itself toggles the diary on and off.

One use of this command is the following. Do whatever work you are going to do with Matlab, and obtain your final results. Then turn on the diary, list all of the relevant variables, and then turn the diary off. The listing will include the names of variables, so you will be able to tell which numbers are which. Then print the file. Save paper; do not print a diary of your entire session.

10. Saving your variables

Before you end a Matlab session, you can save the values of the variables that were defined during that session. The variables will be stored in a file on the disk, and they can be loaded back into Matlab during a later session. You can also save data for the sake of feeding them into another software package.

If, in the following commands, `fname` is a simple file name, then the indicated data is stored in a file in your current working directory. It is also possible to specify a path name to a different directory.

`save fname`	Save all variables in binary form in a file named `fname.mat` .
`save fname x`	Save the variable `x` in a file named `fname.mat` .
`save fname x y z`	Save the variables `x`, `y`, and `z` in a file named `fname.mat` .
`save`	Save all variables in a file `matlab.mat` in your working directory.
`save fname x -ascii`	Save the variable `x` in a file named `fname` . Use 8-digit ASCII form. If `-double` is added to this command, 16-digit ASCII form is used.

If ASCII form is used, the numbers are saved in a form that you can read yourself and that can generally be read by other software packages. The names of the variables do not appear in the file. Otherwise, the values and names of the variables are saved in Matlab's internal binary format.

If the `save` command writes to a file that already exists, its contents are over-written.

11. Loading numbers from external files

If you have saved some variables, they can be read back into Matlab by using the `load` command, which is more or less the inverse of the `save` command. You can also use the `load` command to load numbers that were generated outside Matlab. For example, you could use a Fortran program to generate some numbers and then load them into Matlab in order to produce graphics.

Suppose that some variables have been saved in a file named `fname.mat` using the internal Matlab format. (This means that you used the `save` command without the `-ascii` option.) Type `load fname` at the Matlab prompt; when you state the file name,

do not include the suffix `.mat` . The variable names and their values will then be read into your Matlab session.

Suppose that you saved some variables in ASCII form in a file named `fname`, without any filename extension. If you type `load fname -ascii` at the Matlab prompt, the contents of the file will be loaded into Matlab and placed in an array named `fname`. If the file name has an extension, the situation is slightly different, see `help load` .

More generally, any file of numbers in ASCII form can be read into Matlab, provided that the numbers are given in fixed-length rows that are ended with carriage returns, and the numbers in each row are separated by spaces. The file need not have been created by Matlab.

Vectors and matrices

1. Entering vectors and matrices
2. Array operations
3. Functions for constructing matrices
4. Extracting parts of a matrix
5. Diagonal and triangular parts of matrices
6. Building larger matrices
7. Some matrix functions

1. Entering vectors and matrices

A basic data structure in Matlab is the array. An array with one row or one column will be regarded as a vector, and others will be regarded as matrices.

A simple way to enter a (small) vector or matrix is to give an explicit list. Some examples:

(1) The command `A = [1 2 3; 4 5 6; 7 8 9]` creates a 3×3 matrix whose rows are `1 2 3` , `4 5 6` , and `7 8 9` , respectively. An equivalent command is `A = [1, 2, 3; 4, 5, 6; 7, 8, 9]` . In general, the entries within a row should be separated by spaces or by commas, and the end of a row is marked with a semicolon.

(2) The command `x = [1 2 3]` creates a row vector. An equivalent command is `x = [1, 2, 3]` .

(3) The command `y = [1; 2; 3]` creates a column vector. Another command that gives the same result is `y = [1 2 3]'` . The prime `'` denotes transposition. For vectors or matrices having complex entries, the prime `'` denotes the conjugate transpose.

Matlab provides a convenient notation for generating vectors having equally-spaced entries. The command `x = 0:5` generates the row vector `[0 1 2 3 4 5]`. The command `x = 1:0.01:3` generates the row vector `[1.00 1.01 1.02 ... 2.99 3.00]`. In general, when colon notation is used, the first number indicates the starting value, the last number indicates the ending value, and the middle number (if present) indicates the increment. Negative increments are possible; for example, the command `x = 5:-1:1` generates the vector `[5 4 3 2 1]`. If an increment is not stated explicitly, it is assumed to be 1.

Addition, subtraction, and multiplication of matrices are indicated by `+` , `-` , and `*` , respectively. For example, if `x` and `y` are 3×1 column vectors, then `y'*x` is 1×1 and `x*y'` is 3×3. Entry (m, n) of a matrix `A` can be referenced as `A(m,n)` .

2. Array operations

In Matlab, the term "array operation" is used to refer to an element-by-element operation, as opposed to the usual matrix operations. For example, suppose that `A` and `B` are square matrices having the same dimensions. The usual matrix product is denoted by `A*B`. On the other hand, the product `C = A.*B` is the matrix whose elements are the products of corresponding entries of `A` and `B`. That is, $C(i, j) = A(i, j) * B(i, j)$ for all (i, j).

Similarly, the expression `A./B` produces an element-by-element division of entries of `A` by entries of `B`.

If `B` is a square matrix, then `B^2` is the square of `B`, computed using the usual matrix multiplication. However, `B.^2` is obtained by squaring the individual entries of `B`. If `x` is a row or column vector having more than one element, then `x^2` is undefined, but `x.^2` is the vector obtained by squaring each of the elements of `x`.

Example 2.1. The command `x = 0:0.001:1;` generates a row vector consisting of the numbers from 0 through 1 with increments of 0.001. This vector has 1001 components, so a semicolon is placed at the end in order to suppress printing of the output. The command `y = x.^2 + 1;` then produces a vector `y` which consists of values of the function $x^2 + 1$ at the points 0, .001, .002, ..., .999, 1. The command `z = sin(x);` calculates values of the sine function at those points, and `w = exp(-x) .* sin(x);` computes the values of the function $e^{-x} \sin x$ at those points. The commands `plot(x,y)`, `plot(x,z)`, and `plot(x,w)` could then be used to plot these functions.

3. Functions for constructing matrices

Several functions can be used to construct matrices having special forms. For a more extensive listing, type `help specmat` .

`eye` `eye(n)` is the identity matrix of dimension n.

`ones, zeros` The command `ones(m,n)` yields an $m \times n$ matrix in which all entries are equal to 1, and the command `zeros(m,n)` yields an $m \times n$ matrix that consists entirely of zeros. For example, `ones(3,3)` is a 3×3 matrix consisting entirely of 1's, `zeros(1,10)` is a row vector consisting of ten zeros, and `zeros(4,1)` is a column vector consisting of four zeros. If a single argument is given, the result is a square matrix; for example, `ones(3)` is a 3×3 matrix consisting entirely of 1's . If a matrix `A` already exists, then the commands `ones(size(A))` and `zeros(size(A))` create matrices having the same dimensions as `A`.

`rand, randn` The command `rand(m,n)` produces an $m \times n$ matrix of pseudo-random numbers. The numbers are uniformly distributed in the interval $(0, 1)$. If a single argument is given, the result is a square matrix; the command `rand`, without any arguments, yields a single random number. If a matrix `A` is already defined, then `rand(size(A))` produces a result having the same dimensions as `A`. The function `randn` chooses pseudo-random numbers from a normal distribution having mean 0 and variance 1, and it is used in the same manner as the function `rand`. It is possible to manipulate the seeds of the random number generators; for more information, type `help rand` and `help randn` .

`diag` If `v` is a vector with n components, then `diag(v)` is an $n \times n$ diagonal matrix whose diagonal entries are the components of `v`. More generally, the command `diag(v,k)` produces a square matrix in which the components of `v` form the k'th diagonal; $k = 0$ refers to the main diagonal, $k > 0$ refers to a diagonal above the main diagonal, and $k < 0$ refers to a diagonal below the main diagonal. The matrix `diag(v,k)` has dimension $n + |k|$.

`toeplitz` A Toeplitz matrix is a matrix that is constant along each diagonal. If `c` and `r` are vectors, then `toeplitz(c)` is the symmetric Toeplitz matrix having `c` as its

first column, and `toeplitz(c,r)` is a nonsymmetric Toeplitz matrix having `c` as its first column and `r` as its first row.

Example 3.1. Suppose that you want to construct a 10×10 tridiagonal matrix having 4's on the main diagonal, -2's immediately below the main diagonal, -1's immediately above the main diagonal, and zeros elsewhere. You can use `4*eye(10) - 2*diag(ones(9,1), -1) - diag(ones(9,1), 1)` . An alternative is to use the function `toeplitz` as follows.

```
c = [4, -2, zeros(1,8)];
r = [4, -1, zeros(1,8)];
toeplitz(c,r)
```

Example 3.2. The command `x = randn(1,1000);` generates a vector containing 1000 pseudo-random numbers that are approximately normally distributed. The command `hist(x,20)` then produces a histogram with 20 bins.

4. Extracting parts of a matrix

Suppose, for example, that `A` is a 10×10 matrix. Parts of this matrix can be extracted as follows.

`A(:, 5)` is the fifth column of `A`. The colon means that the first index (row index) is allowed to assume any value, whereas the second index (column index) is fixed equal to 5.

`A(1:3, :)` is a 3×10 submatrix consisting of the first 3 rows of `A`.

`A(4:5, 6:10)` is a 2×5 submatrix consisting of those portions of rows 4 and 5 that lie in columns 6 through 10.

`A(:, [1 3 5])` is a 10×3 matrix whose columns are equal to columns 1, 3, and 5 of `A`.

`A([2 4 6], [1 3 5])` is a 3×3 matrix whose rows are taken from rows 2, 4, and 6 and columns 1, 3, and 5 of `A`. In general, if `v` and `w` are vectors having positive integer components, then `A(v,w)` is obtained by taking the elements of `A` with row indices from `v` and column indices from `w`. The components of `v` and `w` do not have to be in any particular order, and they do not have to be distinct.

`A(:, 10:-1:1)` is obtained by reversing the order of the columns of `A`. The matrix `A(10:-1:1, :)` is obtained by reversing the order of the rows of `A`.

The assignment statement `A(:, [1 3]) = B(:, [7 3])` replaces columns 1 and 3 of `A` with columns 7 and 3 of `B`, respectively, provided that `A` and `B` have the same number of rows.

If the notation `A(:)` appears on the right side of an equation, then it refers to a (long) column vector consisting of the first column of `A`, then the second column of `A`, and so forth. In particular, if `x` is either a row vector or a column vector, then `x(:)` is a column vector. This is useful if you are writing a program that uses a vector defined by the user, and you need to be sure that it is a column vector for the sake of writing your code. Similarly, `x(:)'` is guaranteed to be a row vector; however, recall that a prime `'` denotes the conjugate transpose for complex vectors, so you might want to use `conj(x(:)')` in that case.

5. Diagonal and triangular parts of matrices

If `A` is a matrix, then `diag(A)` is a column vector formed from the diagonal elements of `A`, and `diag(A,k)` is a column vector formed from the k'th diagonal of A . Diagonals above the main diagonal correspond to $k > 0$, and diagonals below the main diagonal correspond to $k < 0$. The `diag` function was also discussed above. If a vector argument is used, the result is a square matrix; if a matrix argument is used, the result is a vector.

The command `tril(A)` gives the lower triangular part of `A`, and `triu(A)` gives the upper triangular part of A . For additional features, consult the help menu.

6. Building larger matrices

A matrix can be defined by an explicit list in which the entries are matrices instead of scalars. That is, a matrix can be constructed from submatrices.

Example 6.1. Suppose that `A` is 4×2. The command `B = [A eye(4); eye(2) A']` creates a 6×6 matrix having the indicated submatrices. The first "row" of submatrices is `A eye(4)`, and the second row of submatrices is `eye(2) A'`. The command `C = [A eye(4); A' eye(2)]` creates another 6×6 matrix.

7. Some matrix functions

Here is a list of some Matlab functions that give useful information about matrices. For a more extensive list, type `help matfun` . For a list of functions related to sparse matrices, type `help sparfun` . For more information about individual functions, consult the help menu, e.g., use `help eig` or `help schur` .

`eig`	Eigenvalues and eigenvectors.
`schur`	Schur form. (Similarity transformation to triangular form).
`rref`	Reduced row echelon form.
`rank`	Rank.
`inv`	Inverse.
`sqrtm`	Square root of a matrix.
`expm`	Exponential of a matrix. If A is a square matrix, then $\exp A$ is defined to be $I + A + A^2/2! + \ldots$.
`null`	Orthonormal basis for the null space.
`qr`	QR factorization.
`svd`	Singular value decomposition.
`chol`	Cholesky factorization.

Example 7.1. Generate a 3×3 matrix `A` of pseudo-random numbers, concatenate the identity matrix on the right to create an augmented matrix, compute the reduced

row-echelon form of the augmented matrix, and extract the right half of the result. This imitates the method for computing the inverse of a matrix that is usually taught in linear algebra courses. (This is not the way the Matlab function `inv` computes the inverse.) The final line verifies that the matrix `C` really is the inverse of `A`, at least approximately.

```
A = rand(3)
aug = [A, eye(3)]
B = rref(aug)
C = B(:, 4:6)
format long, A*C, format
```

Example 7.2. If a matrix A is 4×2 and a matrix B is 2×4, then the product $C = AB$ has rank 2 or less. This is tested by the following sequence of commands.

```
A = rand(4,2)
B = rand(2,4)
C = A*B
rank(C)
```

Example 7.3. The function `schur` returns the Schur form of a matrix. If A is a square real matrix, and if the eigenvalues of A are real, then there exists a real orthogonal matrix U and an upper triangular matrix T such that $U^{-1}AU = T$. The eigenvalues of A are located on the diagonal of T. However, if some of the eigenvalues of A are complex, then the matrix T cannot be diagonal if U is real. In that case, Matlab produces a "real Schur form", in which the matrix T contains 2×2 blocks on the diagonal; the eigenvalues of these 2×2 blocks are complex eigenvalues of A. A "complex Schur form" is also available in Matlab; type `help schur` .

In the first line of the following sequence of commands, the real Schur form of a random real matrix is produced. The second statement assumes that a 2×2 block is found in rows 1 and 2 and columns 1 and 2 of the Schur matrix T. The third statement yields a comparison of the eigenvalues of this block and the eigenvalues of the original matrix A.

```
A = randn(4); T = schur(A)
B = T(1:2, 1:2)
eig(B), eig(A)
```

Example 7.4. For any matrix A (not necessarily square), there exists a matrix Q having orthonormal columns and an upper triangular matrix R such that $A = QR$. The QR factorization has numerous applications, including data fitting. In the following sequence, the first two statements produce the QR factorization of a random 3×3 matrix. The columns of Q then constitute a collection of orthonormal vectors that are "random", in some sense. You can think of these vectors as defining a "random" system of orthogonal coordinates for R^3. The last statement verifies that the columns are actually orthonormal; the columns of Q are orthonormal if and only if Q^TQ is the identity matrix.

```
A = randn(3)
[Q, R] = qr(A)
format long, Q'*Q, format
```

Two-dimensional graphics

1. Help menus
2. The `plot` command
3. Plotting several curves on the same axes
4. Line types
5. Axis control
6. Labelling plots
7. Screen control
8. Plotting several axes in the same graph window
9. Using several graph windows
10. Setting properties of graphics objects
11. Other types of plots

1. Help menus

The command `help graphics` returns a list of general-purpose graphics functions; `help plotxy` gives a list of Matlab functions for plotting two-dimensional data; `help plotxyz` gives a list of functions for plotting three-dimensional data. You can then refer to the help items for individual commands, e.g., type `help plot` to learn about the function `plot`.

2. The `plot` command

The basic Matlab command for plotting a curve is `plot(x,y)` . Here, `x` is a vector that contains the horizontal coordinates of some points on the curve, and `y` is a vector that contains the corresponding vertical coordinates. The two vectors must have the same number of components. The vectors do not have to be named `x` and `y`.

Example 2.1. The following commands produce a graph of $y = x^2$ for $0 \leq x \leq 1$.

```
x = 0 : .01 : 1;
y = x.^2;
plot(x,y)
```

A Matlab graphics window will appear automatically on the screen, and the graph will appear in the window. In this example, the vector `x` consists of the numbers 0, .01, .02, ..., .99, 1.00. In general, Matlab draws a piecewise linear function that connects the data points; the graph will appear smooth if the spacing between the grid points is sufficiently small. The "array operations" that are built into Matlab are very useful for generating vectors of vertical coordinates. For example, if `x` is a vector, then `x^2` is undefined. However, `x.^2` denotes the vector that is obtained by squaring the components of `x`.

The contents of the graphics window will be sent to the default printer if you type the command `print` at the Matlab prompt. It is also possible to use the `print` command to save a graph in a file; for more information, type `help print` .

When you use the `plot` command, y does not have to be a function of x; Matlab simply takes the components of the vectors `x` and `y` in order, and draws a polygonal line that connects the points in that order.

Example 2.2. Graph the unit circle centered at the origin. Without the command `axis('square')`, the graph would be an ellipse, due to different scaling of the horizontal and vertical axes. (See Section 5, *Axis control*.)

```
theta = 0 : pi/60 : 2*pi;
x = cos(theta);
y = sin(theta);
plot(x,y)
axis('square')
```

Example 2.3. The following commands use complex numbers to produce the same result as in Example 2.2. In the present case, `real(z)` is the vector of horizontal coordinates, and `imag(z)` is the vector of vertical coordinates. The command `plot(z)` could be used in place of `plot(real(z),imag(z))` .

```
theta = 0 : pi/60 : 2*pi;
z = exp(i*theta);
plot( real(z), imag(z) )
axis('square')
```

3. Plotting several curves on the same axes

Suppose that the vectors `x1` and `y1` contain horizontal and vertical coordinates for a curve, and suppose that the vectors `x2` and `y2` contain the coordinates for another curve. The command `plot(x1,y1,x2,y2)` plots both curves on the same graph. The vectors `x1` and `x2` could be the same. This procedure can be generalized to any number of curves.

Example 3.1. The following commands produce plots of the curves $y = x$, $y = x^2$, $y = x^3$, and $y = x^4$ on the same graph.

```
x = 0 : .01 : 1;
y1 = x; y2 = x.^2; y3 = x.^3; y4 = x.^4;
plot(x,y1, x,y2, x,y3, x,y4)
```

If several curves are to be plotted simultaneously, and if they all use the same vector of horizontal coordinates, then another method can be used to plot the curves.

Example 3.2. The following sequence of commands produces the same result as the preceding example.

```
x = 0 : .01 : 1;
Y = [x; x.^2; x.^3; x.^4];
plot(x,Y)
```

In general, suppose that `x` is a row or column vector having n components, and suppose that `Y` is a matrix that has either n rows or n columns. The command `plot(x,Y)` graphs the rows or columns of `Y` versus `x`, depending on which dimensions match.

4. Line types

When you plot a curve, it is possible to specify any of several different line types. Suppose that the horizontal and vertical coordinates are contained in vectors `x` and `y`, respectively. Some commands:

`plot(x,y,'-')`	Plot a solid curve.
`plot(x,y,'--')`	Plot a dashed curve.
`plot(x,y,':')`	Plot a dotted curve.
`plot(x,y,'-.')`	Plot a dash-dot curve.

An alternative is to produce point plots, in which the data points are plotted but are not connected by curves. To mark the points with small circles, use `'o'` as the third argument in the `plot` command; for asterisks, use `'*'`; for x's, use `'x'`; for plusses, use `'+'`; for dots, use `'.'` .

If you are working on a machine that has a color monitor, it is also possible to specify the colors of curves and data points. For more information, type `help plot` .

The above facilities can be used when plotting several curves on the same axes. For example, a command of the form `plot(x1,y1,'--',x2,y2,':')` produces a dashed curve and a dotted curve.

5. Axis control

The `axis` command can be used to control the ranges of x- and y-coordinates that are plotted. (Unless you say otherwise, Matlab will choose the ranges automatically.) For example, the command `axis([0 10 -1 1])` specifies that the graph window will show the region $0 \le x \le 10$, $-1 \le y \le 1$. The same effect is obtained by the sequence of commands `v = [0 10 -1 1]; axis(v)` .

The `axis` command should be invoked *after* the graph is plotted. In general, it is possible to plot a graph once and then execute the `axis` command several times to alter the appearance of the plot.

Example 5.1. Generate the monic polynomial whose roots are 0.9, 1.0, 1.1, and 10.0. Then plot the polynomial over the range $-5 \le x \le 15$, together with the zero function. When the `plot` command is executed, Matlab chooses the vertical scaling, and the resulting plot suggests that there is a root near $x = 1$ and a root near $x = 10$. However, the subsequent `axis` command has the effect of zooming in to the range $0.5 \le x \le 1.5$, $-0.1 \le y \le 0.1$, and it is then apparent that there are three roots near $x = 1$. This example illustrates how computer graphics could be misleading due to scaling effects.

```
v = poly([0.9 1.0 1.1 10.0]);
x = -5 : .01 : 15;
y = polyval(v,x); z = zeros(length(x),1);
plot(x,y,x,z)
axis([0.5 1.5 -0.1 0.1])
```

Under the default configuration, the horizontal and vertical axes will generally have different length scales. The command `axis('equal')` forces equal increments on the x- and y-axes to have the same length on the plot.

Example 5.2. The following statements produce graphs of $y = x$ and $y = x^2$ for $0 \leq x \leq 2$. In this plot, the line $y = x$ intersects the horizontal axis at angle 45°. The angle would be different if the command `axis('equal')` were not executed. The command `axis('square')` forces the graph box to be square, which otherwise would not be the case.

```
x = 0 : .1 : 2;
plot(x, x, x, x.^2)
axis('equal')
axis('square')
```

The `axis` command has several additional features; for more information, type `help axis` .

6. Labelling plots

Suppose that a plot is currently residing in the graphics window. Some commands:

`xlabel('info')`	Place the character string `info` immediately below the x-axis.
`ylabel('info')`	Place the character string `info` next to the y-axis.
`title('info')`	Place the character string `info` above the graph.

The function `text` can be used to place a character string at an arbitrary position on the plot. If `x` and `y` are scalars, the command `text(x,y,'info')` places the lower left corner of the character string `info` at position `(x,y)` in the graphics screen, where `x` and `y` are measured in the units of the current plot. For more information about `text`, type `help text` .

The function `gtext` is similar to `text`, except that the text is placed graphically. Execute the command `gtext('info')`, use the mouse to move the pointer to the desired location in the graph window, and then press any mouse button or any key. The lower left corner of the character string `info` is then placed at that position.

7. Screen control

A Matlab graphics window can be resized and/or moved by using the same techniques that you would use for any other window in the window system that you are using. To clear the contents of the graphics window, type `clf` or `clg` .

The command `hold on` holds the current graph on the screen. Subsequent graphing commands will add to the current plot; everything that is already in the graphics window will be retained, and the axes will not change. The command `hold off` turns off this mode.

The command `ginput` can be used to read coordinates of points on the screen. Execute the command, use the mouse to move cross-hairs to the desired point, and click any button. For details, type `help ginput` .

8. Plotting several axes in the same graph window

It is possible to divide the graph window into several subwindows and then place a plot in each subwindow. The command `subplot(m,n,p)` divides the graph window into an $m \times n$ array of subwindows and then selects the p'th subwindow for the next plot. The subwindows are numbered row-wise. If the p'th subwindow already contains a plot, then `subplot(m,n,p)` causes that window to become the current window.

Example 8.1. Display the graphs of $y = x$, $y = x^2$, $y = x^3$, and $y = x^4$ in a 2×2 array of plots.

```
x = 0 : .01 : 1;
%    Divide the graph window into a 2x2 array of windows, and
%    and plot y = x in the first window.
subplot(2,2,1)
plot(x, x)
title('y = x')

%    Place plots in the other windows.  Several commands
%    can be placed on one line.
subplot(2,2,2), plot(x, x.^2), title('y = x^2')
subplot(2,2,3), plot(x, x.^3), title('y = x^3')
subplot(2,2,4), plot(x, x.^4), title('y = x^4')

%    Go back and put y-labels on the first and fourth plots.
subplot(2,2,1), ylabel('first plot')
subplot(2,2,4), ylabel('last plot')
```

9. Using several graph windows

The command `figure` can be used to create new graphics windows. If, for example, you have just started a Matlab session, then a graphics command would create a graph in a window labelled `Figure No. 1` . Subsequent graphics commands would over-write the contents of that window. If you wish to retain the contents of that window while creating new graphs, then execute the command `figure` to create a new window; if `Figure No. 1` is the only existing graphics window, then the new window is labelled `Figure No. 2` . Subsequent executions of `figure` create additional windows. At any stage in a Matlab session, graphics commands will affect the "current" figure. The command `figure(n)` makes figure n the current figure, and the command `gcf` ("get current figure") gives you the number of the current figure.

The command `print` prints the current figure; the commands `print -f1` and `print -f2` print figures 1 and 2, respectively. The command `clf` clears the contents of the current figure. The command `close` closes the current figure window, and `close(n)` closes figure n.

Example 9.1. Suppose that the only open graphics window is `Figure No. 1` . The following commands plot the graph of $y = x$ in `Figure No. 1` , $y = x^2$ in `Figure No. 2` , and $y = x^3$ in `Figure No. 3` . The third figure is then printed.

```
x = 0:0.01:1;
y = x;
plot(x,y)
figure
plot(x, x.^2)
figure, plot(x, x.^3), print -f3
```

10. Setting properties of graphics objects

The function `set` and related functions make it possible to set a few properties of graphics objects. Type `help set` , and also refer to the other functions mentioned there.

Example 10.1. In the following, the second statement places circles at each of the data points and assigns the "handle" `p` to the graph. The third statement makes the circles twice as large as the default size, which is "6". To see a listing of various characteristics of the graph, type `get(p)` .

```
x = 0:0.1:1;
p = plot(x, exp(-x), 'o')
set(p, 'MarkerSize', 12)
```

11. Other types of plots

Parametric plots can be produced by using the `plot` command.

Example 11.1. Graph the spiral curve $x = t\cos t$, $y = t\sin t$ for $0 \le t \le 2\pi$. The command `axis('equal')` eliminates the distortion that would have occurred due to different horizontal and vertical scales, and the command `axis([-7 7 -7 7])` causes the point $(0, 0)$ to be in the center of the graph window.

```
t = 0:.01:2*pi;
x = t.*cos(t); y = t.*sin(t);
plot(x,y)
axis('equal'), axis([-7 7 -7 7]), axis('square')
```

Logarithmic plots are also possible. The following commands are used in the same manner as `plot`, and they yield the indicated results.

`loglog`	x- and y-axes are logarithmic.
`semilogx`	x-axis is logarithmic, y-axis is linear.
`semilogy`	y-axis is logarithmic, x-axis is linear.

The `polar` command is used to produce polar plots. In the argument list, list the angle (in radians) and then the radius. The `polar` command will not accept multiple plots, so it will be necessary to use the `hold` command if you want more than one curve in the same plot.

Example 11.2. Graph the curves $\rho = \cos 2\theta$ and $\rho = \cos 4\theta$ for $0 \le \theta \le 2\pi$. The second curve is plotted as a dotted curve.

```
theta = 0 : .01 : 2*pi;
rho2 = cos(2*theta); rho4 = cos(4*theta);
polar(theta, rho2)
hold on, polar(theta,rho4,':'), hold off
title('polar plot')
```

The function `fplot` can be used to plot functions that are defined symbolically; Matlab will choose the sample points for you. However, the function needs to be defined in an M-file.

Example 11.3. Suppose that a file named `fsqr.m` is located in a directory in Matlab's search path, and suppose that this file consists of the following statements.

```
function y = fsqr(x)
y = x.^2;
```

If you then execute the command `fplot('fsqr', [0 10])` in your Matlab session, the result is a graph of $y = x^2$ for $0 \leq x \leq 10$. Various options are available; type `help fplot` .

Three-dimensional graphics

1. Defining arrays of independent and dependent variables
2. Contour plots
3. Plotting implicit functions
4. Surface plots
5. Parametric plots

1. Defining arrays of independent and dependent variables

Matlab contains commands for producing contour plots and surface plots. In each case, Matlab plots the data contained in a rectangular matrix. The entries in such a matrix are regarded as z-coordinates, and the row and column indices correspond to the independent variables. (For parametric plots, the situation is a little more complicated; see Section 5.) The purpose of this section is to show how to generate arrays that can be plotted.

If the z-coordinates are to be calculated from an explicit formula involving independent variables x and y , it is very useful to generate matrices containing values of these variables.

Example 1.1. Suppose that you want to plot a function of (x, y) for $0 \leq x \leq 1.5$ and $0 \leq y \leq 1$, with increment 0.5 in each variable. (In practice, the increment should generally be much smaller than this.) Arrays containing values of these variables are generated by the following commands.

```
x = 0:0.5 :1.5;
y = 0:0.5 :1;
[X,Y] = meshgrid(x,y);
```

Recall that Matlab is case-sensitive, so that `X` is not the same as `x`. The output from the `meshgrid` command is as follows.

$$\texttt{X} = \begin{matrix} 0 & 0.5 & 1.0 & 1.5 \\ 0 & 0.5 & 1.0 & 1.5 \\ 0 & 0.5 & 1.0 & 1.5 \end{matrix} \qquad \texttt{Y} = \begin{matrix} 0 & 0 & 0 & 0 \\ 0.5 & 0.5 & 0.5 & 0.5 \\ 1.0 & 1.0 & 1.0 & 1.0 \end{matrix}$$

The matrix `X` thus contains values of x-coordinates, and matrix `Y` contains values of y-coordinates. In the matrix `Y`, the row index increases with increasing values of y. Don't worry about the values of y being upside down; this is taken care of automatically by the contour plot and surface plot routines.

If you want to plot the function $z = x^2 + e^{-y} \sin x$, the array of z-values is then computed by using the Matlab statement `Z = X.^2 + exp(-Y).*sin(X);` . Similarly, values of the function $z = xy$ are obtained by the command `Z = X.*Y;` .

2. Contour plots

The Matlab function `contour` produces contour plots of functions of two real variables; the Matlab function `contour3` produces three-dimensional contour plots, in which

contours are placed on a three-dimensional surface. Information about these and other facilities can be obtained via the `help` menu; for example, try `help plotxyz` and `help graphics` .

Example 2.1. Here, we produce a contour plot of the surface $z = e^{-y} \sin x$ for $0 \le x \le \pi$ and $0 \le y \le 1$.

```
x = 0 : pi/30 : pi;
y = 0 : .1 : 1;
[X,Y] = meshgrid(x,y);
Z = exp(-Y) .* sin(X);
contour(Z)
```

The first four statements produce values of z on a rectangular grid. On this grid, x varies from 0 to π in increments of $\pi/30$, and y varies from 0 to 1 in increments of 0.1 . The last statement produces a contour plot. In this plot, the contour curves are not labelled, and the coordinate axes are labelled by matrix indices, not by values of the actual coordinates x and y. Matlab chooses the values of z for which contours are plotted.

Example 2.2. In the preceding example, replace the statement `contour(Z)` with `contour(x,y,Z)`. In the resulting plot, the horizontal coordinate is labelled as varying from 0 to π, and the vertical coordinate is labelled as varying from 0 to 1. The contour curves are not labelled, and the contour interval is chosen by Matlab.

Example 2.3. In this example, the contour levels are specified explicitly, and each contour is labelled with the corresponding value of z.

```
x = 0 : pi/30 : pi;
y = 0 : .1 : 1;
[X,Y] = meshgrid(x,y);
Z = exp(-Y) .* sin(X);
v = .2 : .2 : 1;
cdata = contour(x,y,Z,v);
clabel(cdata)
```

The vector `v` contains the values of z for which contours are to be drawn. In general, these values do not need to be evenly spaced, nor do they need to be given in any particular order.

The statement `cdata = contour(x,y,Z,v);` produces the plot and stores data about the plot in an array named `cdata` . (It is not necessary to use the name `cdata` for this array.) The semicolon at the end of this statement is used to suppress printing of the array. For information about what is stored in the array, type `help contourc` at the Matlab prompt.

The final statement `clabel(cdata)` produces labels of the contour curves. Matlab chooses the positions to place the labels, and each curve is labelled at least once. If you wish to choose the positions yourself, use the statement `clabel(cdata,'manual')` instead of `clabel(cdata)`. Use the mouse to point to positions on the contour curves where labels are to be placed. Clicking the mouse places a label. To quit this mode, press the `return` key.

Example 2.4. In the preceding example, replace the last two lines by `contour3(x,y,Z, v)` . The result is a three-dimensional graph in which contour curves lie on the surface which is the graph of $z = e^{-y} \sin x$. The position from which the surface is viewed can be adjusted by using the function `view`, as described in Section 4, *Surface plots*.

The commands `title`, `xlabel` and `ylabel` can be used with the `contour` and `contour3` commands in the same manner as they are used with the `plot` command. A command `zlabel` can be used with `contour3`. Also, the functions `text` and `gtext` can be used with `contour`, and the three-dimensional version of `text` can be used with `contour3`.

3. Plotting implicit functions

One application of contour plotting is to plot curves that are defined implicitly. If you want to plot a curve of the form $f(x, y) = 0$, make a contour plot of f with one contour level $z = 0$.

Example 3.1. Plot the curve(s) defined by $e^{xy} = (1 + x + y)$.

```
x = -5 : .1 : 5;
y = -5 : .1 : 5;
[X,Y] = meshgrid(x,y);
Z = exp(X.*Y) - (1 + X + Y);
v = [0 0];
contour(x,y,Z,v)
```

The vector `v = [0 0]` is an improvisation. If the vector `v = 0` or `v = [0]` were used, Matlab would produce no contour curves; the command `contour(x,y,Z,n)` produces n contour levels if n is an integer. (See `help contour` .)

4. Surface plots

Matlab has several commands for plotting surfaces in three dimensions. Some examples are the following.

`mesh`	Represent the surface as a wire-frame mesh.
`meshc`	Represent the surface as a wire-frame mesh, and also display a contour plot in the (x, y) plane.
`surf`	Same as `mesh`, except that the "panes" between the mesh curves are colored (or shaded).
`surfc`	Same as `surf`, except that a contour plot is also shown in the (x, y) plane.

For more information on these and other facilities, consult the `help` menu, e.g., type `help plotxyz` and `help graphics` . The functions `xlabel`, `ylabel`, `zlabel`, `title`, and `text` can be used to label surface plots. Some aspects of the `mesh` and `meshc` functions are illustrated in the following examples. The `surf` and `surfc` functions are used in the same manner.

Example 4.1. Here, we produce a mesh plot of the surface $z = e^{-y} \sin x$ for $0 \le x \le \pi$ and $0 \le y \le 1$.

```
x = 0 : pi/30 : pi;
y = 0 : .1 : 1;
[X,Y] = meshgrid(x,y);
Z = exp(-Y).*sin(X);
mesh(Z), xlabel('x'), ylabel('y'), zlabel('z')
```

In this plot, the origin is closest to the observer, the x-axis points to the right and into the background, the y-axis points backward and to the left, and the z-axis points upward. The x- and y-axes are labelled by matrix indices, not by the actual coordinate values.

The function $u(x, y) = e^{-y} \sin x$ is a solution of the Laplace equation $u_{xx} + u_{yy} = 0$. This plot illustrates the strong maximum and minimum principles for that equation. If a nonconstant function satisfies the Laplace equation on a connected open set, then the function can have no interior local maxima and no interior local minima. (Loosely speaking, the equation $u_{xx} = -u_{yy}$ implies that the concavities in the x- and y-directions must have opposite signs.) Thus the maximum and minimum values must be found on the boundary.

Example 4.2. In the preceding example, replace the last line with

```
mesh(x,y,Z), xlabel('x'), ylabel('y'), zlabel('z')
```

In this case, the horizontal axes are labelled according to the true coordinate values. However, the x-axis might be longer than the interval $[0, \pi]$. If you want to make sure that the x-axis extends only over the interval $[0, \pi]$, type `axis([0 pi 0 1 0 1])` after the graph has been plotted.

Example 4.3. The function `view` can be used to change the point from which the surface is viewed. Suppose `x`, `y`, and `Z` are as defined in Example 4.1.

```
mesh(x,y,Z), xlabel('x'), ylabel('y'), zlabel('z')
axis([0 pi 0 1 0 1])
view(120,30)
```

In this example, the x-axis points forward and to the left, the y-axis points to the right, and the z-axis points upward.

In general, the first argument in function `view` is an angle of rotation (in degrees) about the z-axis, and the second argument is an angle of elevation above or below the (x, y) plane. If the first argument is zero, then the x-axis points to the right, and the y-axis points into the background. Positive values of the first argument indicate counterclockwise rotation of the observer, as viewed from the positive z-axis. The default value of the first argument is -37.5. The default angle of elevation is 30; positive values mean that the observer is above the (x, y) plane.

Example 4.4. Suppose `x`, `y`, and `Z` are as defined in Example 4.1.

```
meshc(x,y,Z), xlabel('x'), ylabel('y'), zlabel('z')
axis([0 pi 0 1 0 1])
```

The function `meshc` produces a mesh plot, just like the function `mesh`, and it also creates a contour plot in the (x, y) plane. In this example, the default viewpoint is used, but the viewpoint can be changed by using the `view` function.

5. Parametric plots

The functions `plot3` and `comet3` can be used to plot parametric curves in three dimensions; consult the help items for these functions. The function `comet` is a two-dimensional analogue of `comet3` .

The `mesh` and `surf` functions can be used to plot surfaces for which z is not a function of x and y. Instead, write x, y, and z as functions of two independent variables, and plot a "parametric" surface.

Example 5.1. If the axis of a torus is the z-axis, then the torus can be parameterized in the form $x = (a + b\cos\psi)\cos\theta$, $y = (a + b\cos\psi)\sin\theta$, $z = b\sin\psi$, for $0 \le \theta \le 2\pi$, $0 \le \psi \le 2\pi$. Here, a is the distance from the z-axis to the center of a cross-section, b is the radius of a cross-section, θ is an angle of rotation about the z-axis, and ψ is an angle of rotation within a cross-section. Here, we plot a torus for which $a = 2$ and $b = 1$.

```
theta = 0 : pi/16 : 2*pi;
psi = 0 : pi/16 : 2*pi;
[T, P] = meshgrid(theta, psi);
a = 2; b = 1;
X = ( a + b*cos(P) ) .* cos(T);
Y = ( a + b*cos(P) ) .* sin(T);
Z = b*sin(P);
mesh(X,Y,Z)
axis([-3, 3, -3, 3, -3, 3])
axis('square')
```

If the `axis` commands were not executed, then the default scaling of axes would be used, and the cross-sections of the torus would appear to be ellipses. With the given sequence of commands, the cross-sections are approximately circular.

Linear systems and data fitting

1. The operator `\`
2. Comparison with `inv(A)`
3. Condition number
4. Solving several systems with the same coefficient matrix
5. Overdetermined systems and data fitting
6. Other functions for data fitting
7. Sparse matrices

1. The operator `\`

Suppose that A is an $n \times n$ matrix and b is an $n \times 1$ column vector. The linear system $Ax = b$ can be solved by executing the command `x = A\b` . The operation `\` represents "left division", and it can be thought of as dividing A into b. When this operation is invoked, Matlab checks several possibilities.

(1) First, Matlab checks whether A is triangular or can be obtained from a triangular matrix by permuting some rows. If so, Matlab solves the system via a (permuted) back-substitution.

(2) If the test in (1) fails, then Matlab checks whether the matrix is real symmetric (or Hermitian) and has positive diagonal elements. If so, Matlab attempts to compute a Cholesky factorization $A = LL^*$, where L is lower triangular, and L^* is the conjugate transpose of L. Such a factorization exists if and only if A is Hermitian and positive definite, and a necessary condition for positive definiteness is that the matrix have positive diagonal elements. If the factorization succeeds, then the system is equivalent to $L(L^*x) = b$, which is then solved by solving the two triangular systems $Ly = b$, $L^*x = y$. If successful, this method requires less than half the time required by general Gaussian elimination, which is described next.

(3) If the tests in (1) and (2) fail, then Matlab uses Gaussian elimination with partial pivoting to compute a factorization $A = LU$, where U is upper triangular and L is either lower triangular or is obtained from a lower triangular matrix by permuting some rows. The system is then equivalent to $L(Ux) = b$, which is solved by solving the two triangular systems $Ly = b$ and $Ux = y$. In a sense, the matrix L records the row operations that are performed when A is reduced to upper triangular form, and U is the upper triangular form itself. Solving $Ly = b$ has the same net effect as performing on b the same row operations that are performed on A, and solving $Ux = y$ is the back-substitution. The Matlab function `lu` can be used to obtain the factors L and U explicitly; a situation where this is useful is described in Section 4.

2. Comparison with `inv(A)`

The command `x = inv(A)*b` could also be used to solve the system $Ax = b$. Here, `inv(A)` is the inverse of A. This is obtained by computing the factorization $A = LU$ described in Section 1, inverting the factors L and U, and then using $A^{-1} = U^{-1}L^{-1}$.

In general, the command `x = inv(A)*b` is less efficient than `x = A\b`. The inverse of A is a square matrix X such that $AX = I$. In effect, when the inverse of A is computed, one has solved the n linear systems $Ax_j = e_j$ for $1 \le j \le n$, where x_j is the j'th column of X, and e_j is the j'th column of I. But the ultimate goal was to solve one linear system.

Example 2.1. Compare the speed and accuracy of `x = A\b` and `x = inv(A)*b`. Generate a 100×100 matrix of random numbers, choose an "exact" solution vector of random numbers, compute a right-hand side, solve the system, compute execution times, and compute norms of the differences between the "exact" solution and the computed solution. In repeated tests, the command `A\b` has been about twice as fast as `inv(A)*b`. The errors are comparable, with `A\b` generally giving slightly smaller errors. In Matlab, text that follows a percent sign (`%`) is a comment and is not executed.

```
%    Pick an exact solution, and compute a right side.
A = rand(100);
exact = rand(100,1);
b = A*exact;

%    Use A\b.
start = cputime; x = A\b; t = cputime-start
err = norm(x - exact)

%    Use inv(A).
start = cputime; xinv = inv(A)*b; tinv = cputime-start
errinv = norm(xinv - exact)
```

The function `cputime` returns the CPU (central processing unit) time in seconds that has been used by Matlab since the current session was started. The CPU time used by a set of commands can then be determined by executing `cputime` immediately before and after those commands are executed and then comparing the results. Other Matlab functions related to timing are `clock`, `etime`, `tic`, and `toc`.

3. Condition number

The command `cond(A)` gives the condition number $\|A\|_2\|A^{-1}\|_2$, where $\|\cdot\|_2$ is the operator matrix norm corresponding to the Euclidean vector norm. That is, if $\|x\|_2{}^2 = \sum_{i=1}^n |x_i|^2$ for all vectors $x \in C^n$, then $\|A\|_2$ is defined by $\|A\|_2 = \max_{x\neq 0}\Big(\|Ax\|_2/\|x\|_2\Big)$ for any $n \times n$ matrix A. In the case where A is real and symmetric (or complex and Hermitian), the condition number with respect to this norm is equal to the ratio of the largest and smallest absolute values of the eigenvalues of A. The quantity $\log_{10}\mathrm{cond}(A)$ gives a rough estimate of the number of decimal digits of accuracy that can be lost during the process of solving a linear system in finite-precision arithmetic. A problem is said to be "ill-conditioned" if the solution is highly sensitive to the effects of perturbations such as roundoff errors, and the condition number gives a quantitative measure of this property.

Example 3.1. The $n \times n$ Hilbert matrix is defined by $a_{ij} = 1/(i+j-1)$. The following statements generate the 10×10 Hilbert matrix, an "exact" solution $x = (1, \ldots, 1)^T$, and a

right-hand side $b = Ax$. The operator `\` is then used to compute a solution to the linear system. On a workstation using double-precision arithmetic with about 16 decimal digits, the computed solution agrees with the vector $(1, \ldots, 1)^T$ to about three decimal places, and the condition number is about 10^{13}.

```
A = hilb(10);
x = ones(10,1);
b = A*x;
format long, A\b
cond(A)
```

Example 3.2. Another way to view the situation described in the preceding example is as follows. Suppose that $Ax = b$, and for the sake of simplicity suppose that the perturbations in the data are confined to the right side. Let r denote the perturbation in the right side, and let e denote the corresponding perturbation in the solution. Thus $A(x + e) = b + r$; since $Ax = b$, we then have $Ae = r$, or $e = A^{-1}r$. If `A` is the 10×10 Hilbert matrix, then the command `inv(A)` yields a result whose largest entries have a magnitude between 10^{12} and 10^{13}. The components of e thus can be far larger than the components of r.

Example 3.3. One can also use eigenvalues to describe the behavior observed in Example 3.1. As in Example 3.2, let r denote the perturbation in the right side, and let e denote the corresponding perturbation in the solution, so that $Ae = r$. Let $\lambda_1, \ldots, \lambda_n$ denote the eigenvalues of A, and let $x_1, \ldots, x_n$ denote corresponding eigenvectors with $\|x_j\|_2 = 1$ for all j. The eigenvectors can be assumed orthogonal, since A is symmetric. Because the eigenvectors are linearly independent, there exist constants $\alpha_1, \ldots, \alpha_n$ so that $r = \alpha_1 x_1 + \ldots + \alpha_n x_n$. The eigenvalue-eigenvector relationship $Ax_j = \lambda_j x_j$ then implies that the perturbation e in the solution is given by $e = (\alpha_1/\lambda_1)x_1 + \ldots + (\alpha_n/\lambda_n)x_n$.

If A is the 10×10 Hilbert matrix, then the commands `format short e`, `eig(A)` show that the smallest eigenvalue of A is approximately 1.1×10^{-13}, and the largest is approximately equal to 1.75. The expression $e = (\alpha_1/\lambda_1)x_1 + \ldots + (\alpha_n/\lambda_n)x_n$ then reveals that e could be quite large relative to r.

4. Solving several systems with the same coefficient matrix

Suppose that you need to solve several linear systems that have the same coefficient matrix but different right sides. This problem can be formulated as a matrix equation $AX = B$, where the columns of B are the various right sides, and the columns of X are the corresponding solutions. The system $AX = B$ can be solved by the command `X = A\B` . The matrix B need not be square. The algorithm used in this case is the same as described in Section 1, *The operator* `\` .

An alternative is to use the `lu` function to compute the factorization $A = LU$, where L and U are described in Section 1. The system can then be solved by the command `x = u\(l\b)` . This method is useful if you will encounter different right sides at different times, but want to process the coefficient matrix once and for all. For large problems, this approach is far more efficient than executing `A\b` repeatedly.

5. Overdetermined systems and data fitting

Consider the linear system $Ax = b$, where A is $m \times n$ with $m > n$, and x and b are column vectors having n and m components, respectively. Because $m > n$, this system may not have any solutions. In this case, Matlab computes the "least squares" solution of the system, which is a vector x that minimizes the Euclidean norm $\|Ax - b\|_2$ of the residual vector $Ax - b$. The command syntax is the same as before; use `x = A\b` .

It can be shown that the minimizing x satisfies the square system $A^T Ax = A^T b$, which is known as the system of "normal equations". This system is equivalent to $A^T(Ax-b) = 0$; the residual vector $Ax - b$ is thus orthogonal to the column space of A, so that the vector Ax is the orthogonal projection of b onto the column space. Matlab's algorithm is based on factoring the matrix A into the product of a matrix Q with orthonormal columns and a square matrix R that is upper triangular. (This is the "QR factorization".) The normal equations are then equivalent to $R^T Q^T QRx = R^T Q^T b$. If A has full rank, then R is nonsingular. The system is then equivalent to $Rx = Q^T b$, which can be solved by back-substitution. The QR factorization has the effect of using an orthogonal coordinate system for the column space of A when computing the orthogonal projection of b onto that space. This method is used for reasons of numerical accuracy; see Example 5.2. Extensive information on least-squares problems can be found in the book *Matrix Computations* by G. Golub and C. Van Loan.

Example 5.1. Find the line of best fit to a set of data points. Denote the data points by (x_i, y_i) for $1 \le i \le m$, and let $p(x) = c_1 x + c_2$ denote the line of best fit. Ideally, you would like to have $p(x_i) = y_i$ for all i. In other words, you want $c_1 x_i + c_2 = y_i$ for all i. This can be expressed as an overdetermined system having unknowns c_1 and c_2. The first column of the coefficient matrix consists of x_1, ..., x_m, and the second consists of ones. One criterion for determining a line of "best fit" is to find coefficients c_1 and c_2 which minimize the quantity $\sum_{i=1}^m (p(x_i) - y_i)^2 = \sum_{i=1}^m (c_1 x_i + c_2 - y_i)^2$. This is equivalent to solving the overdetermined system in the least-squares sense. This problem can also be solved by using the Matlab function `polyfit`; see Section 6, *Other functions for data fitting.*

In the following statements, the data points are obtained by introducing random perturbations to the line $y = x$. The quantities `x(:)` and `y(:)` are column vectors, and the matrix `A` has dimension 11×2.

```
x = 0:.1:1;
y = x + 0.05*randn(1,11);
A = [x(:), ones(11,1)];
b = y(:);
coeffs = A\b
line = coeffs(1)*x + coeffs(2);
plot(x,y,'o',x,line)
```

Numerical accuracy can be a source of difficulty in data fitting problems. For example, if you fit data with a polynomial, and if you represent the polynomial as a linear combination of 1, x, x^2, ..., then the columns of the coefficient matrix could be almost linearly dependent, in a sense. The normal equations then constitute an ill-conditioned

system. (The Hilbert matrix can be obtained from a continuous analogue of this situation.) Dealing with this possibility is a motivation for solving least-squares problems via QR factorization instead of by a direct solution of the normal equations.

Example 5.2. The following "M-file" `pfit.m` solves an overdetermined system by using Matlab's built-in least-squares algorithm, which is based on a QR factorization of the coefficient matrix. It then solves the normal equations directly. For the sake of comparison, the quantity $\|Ax - b\|_2$ is computed in each case. The method based on QR factorization generally gives much better results; for example, try `n=15` and `nints=30`.

If a file having a name of the form `fname.m` is located in a directory in Matlab's search path, then the contents of that file can be executed by typing `fname` at the Matlab prompt.

```
%    PFIT.M
%    Generate a random perturbation of the line y = x for
%    0 <= x <= 1 and then compute a polynomial that gives
%    a best fit to the data points in the least-squares sense.
%    USAGE: At the console, define the following:
%        n = degree of polynomial used in the fit.
%        nints = number of subintervals of [0, 1].

x = 0 : 1/nints : 1;
y = x + 0.05*randn(1,nints+1);

%    Set up the coefficient matrix and right-hand side.
A = ones(nints+1,1);
for k = 1:n,
    A = [x(:).^k  A];
end
b = y(:);

%    Use Matlab's built-in least-squares algorithm.
%    Also compute a norm of the residual.
c = A\b;
r = norm(A*c - b)

%    Solve the normal equations directly and
%    compute a norm of the residual.
A1 = A'*A; b1 = A'*b; c1 = A1\b1;
r1 = norm(A*c1 - b)

%    Plot.
xplot = 0 : .001 : 1;
yplot = polyval(c, xplot);
yplot1 = polyval(c1, xplot);
plot(x, y, 'o', xplot, yplot, '-', xplot, yplot1, ':'  )
axis([0 1 -0.5 1.5])
```

6. Other functions for data fitting

The Matlab function `polyfit` computes the coefficients of the polynomial of specified degree which gives the best least-squares fit to a given set of data points. If vectors `x` and `y` contain the horizontal and vertical coordinates of the data points, and if you want the polynomial of degree n or less that fits the data best, use the command `polyfit(x,y,n)` . The output is a vector of coefficients of the polynomial, arranged in order of decreasing powers. Given a vector of horizontal coordinates, the corresponding values of the polynomial can then be computed with the function `polyval`, as illustrated in Example 5.2. If the number of data points is $n+1$, then the graph of the polynomial passes through all of the points, i.e., the polynomial interpolates the data set.

The Matlab function `spline` computes a cubic spline function that interpolates a given set of data. A cubic spline function is a piecewise cubic polynomial that has continuous derivatives of orders 2 and less. If you want to interpolate a large number of data points with a smooth curve that does not wiggle much, splines are probably better than polynomials.

Example 6.1. Let $f(x) = 1/(x^2+1)$ for $-5 \le x \le 5$. The following commands plot f together with two different functions that interpolate f at the points $x_0 = -5$, $x_1 = -4$, ..., $x_{10} = 5$. A dotted curve is used to graph the polynomial of degree 10 that interpolates f at x_0, ..., x_{10}; and a dashed curve is used to plot the cubic spline function that interpolates f at those points. The polynomial oscillates greatly near the endpoints of the interval $[-5, 5]$, but the cubic spline function approximates f closely over the entire interval. This example was devised by Runge to illustrate the behavior of polynomial interpolation at evenly spaced points.

```
x = -5:5;
y = 1 ./ (x.^2 + 1);
xplot = -5 :0.05 :5;
yplot = 1 ./ (xplot.^2 + 1);
polycoeffs = polyfit(x,y, length(x)-1);
ypoly = polyval(polycoeffs, xplot);
yspline = spline(x,y,xplot);
plot(xplot,yplot, xplot,ypoly,':', xplot,yspline,'--', x,y,'o')
axis([-5, 5, -.5, 2])
```

Matlab also includes several functions for performing fast Fourier transforms and related operations; type `help datafun` for a list of function names.

7. Sparse matrices

A matrix is said to be "sparse" if most of its entries are zero. Numerous problems in scientific computing involve matrices that are extremely large but also extremely sparse. If computations are performed with such matrices, the execution times and storage requirements can be reduced dramatically if the sparsity is exploited. Matlab has facilities for storing such matrices efficiently, and for matrices stored in this special way Matlab can solve linear systems by means of the `\` operation, compute LU and Cholesky factorizations, and compute least-squares solutions to overdetermined systems. For details, consult the Matlab *User's Guide* and Matlab *Reference Guide*. For a list of functions related to sparse matrices, type `help sparfun` .

Programming

1. M-files
2. Control structures
3. Efficiency

1. M-files

It is possible to write programs in the Matlab language. These should be placed in "M-files", which are files consisting of Matlab statements and whose names end with the suffix `.m` . When you execute an M-file, Matlab has to be able to find the file, so the file should be located either in your current working directory or in a directory in Matlab's search path.

Two kinds of M-files are possible: script files and function files. A script file is simply a sequence of Matlab statements, and these are executed when the name of file (without the `.m` suffix) is typed at the Matlab prompt. The final results are the same as would be obtained if each of the Matlab statements were typed at the prompt `>>` in your Matlab session. In particular, none of the variables in a script file are "local". If the first line of an M-file contains the word `function`, the file is a function file, and all variables created within the file are local to that program.

Example 1.1. Suppose that the following Matlab statements are contained in a file `angcol.m` which is located in a directory in Matlab's search path. In this function file, variables `c1`, `c2`, and `d` are defined, and variables `A`, `m`, and `n` are used. If these variables had already been defined in your Matlab session, their values would not be changed by the action of this function. The lines that begin with the character `%` are comments and are not executed.

When the following function is invoked by a statement at the Matlab prompt, it is not necessary to use the same names as those in the M-file. For example, the statement `angcol(eye(5),4,5)` produces the angle between the fourth and fifth columns of the 5×5 identity matrix, and the statement `degrees = angcol(hilb(10),9,10)` sets the variable `degrees` equal to the angle between columns 9 and 10 of the 10×10 Hilbert matrix. The outputs of these commands are `90` and `0.9560`, respectively.

```
function d = angcol(A, m, n)
%    angcol(A, m, n) is the angle, in degrees, between
%    columns m and n of matrix A.
c1 = A(:,m); c1 = c1 / norm(c1);
c2 = A(:,n); c2 = c2 / norm(c2);
d = acos(c1'*c2)*180/pi;
```

Example 1.2. A function file can also return multiple arguments. Suppose that the following function is contained in a file named `angcol2.m` . If the command `[degrees, radians] = angcol2(hilb(10),9,10)` is executed at the Matlab prompt `>>`, then the variable `degrees` is set equal to the degree measure of the angle between columns 9 and 10

of the 10×10 Hilbert matrix, and the variable `radians` is set equal to the radian measure of that angle. The command `angcol2(hilb(10),9,10)` returns only the first argument, namely, the degree measure.

```
function [d,r] = angcol2(A, m, n)
%    [d,r] = angcol(A, m, n) produces the angle between
%    columns m and n of matrix A. The quantity d is the
%    angle in degrees; r is the angle in radians.
c1 = A(:,m); c1 = c1 / norm(c1);
c2 = A(:,n); c2 = c2 / norm(c2);
r = acos(c1'*c2);
d = r*180/pi;
```

An M-file can be accessed by the `help` command, if it is located in your current working directory or in Matlab's search path. If a file named `fname.m` is a script file, then the command `help fname` will produce a listing of all comment lines that appear before the first executable statement in the M-file. If the file is a function file, then `help fname` will produce a listing of all comment lines that appear after the `function` statement and before the next executable statement.

2. Control structures

The Matlab language includes `for` loops, `while` loops, and an if-then-else control structure. These are very similar to ones found in traditional programming languages. For more information, consult the help facility; type `help for` , `help while` , and `help if` . Because of Matlab's usage of array operations, a Matlab program typically does not need as many loops as do programs written in traditional languages. In fact, for reasons of efficiency, loops should be avoided if possible; see Section 3. However, there are situations when loops are appropriate.

Example 2.1. Consider an algebraic equation of the form $f(x) = 0$, where f is a differentiable real-valued function of one real variable. Approximate solutions can be computed with Newton's method, which is the iteration $x_{n+1} = x_n - f(x_n)/f'(x_n)$.

In order to use Newton's method to compute a root, it is first necessary to determine a starting value x_0; this can typically be determined from a plot of the function f. For example, if $f(x) = x^3 - 4x^2 + 3x + 1$, then a plot indicates that there are roots near $x = 0$, $x = 1.5$, and $x = 3$. In the following M-file `newton.m`, the statement `x = [0 1.5 3];` initializes the vector `x` to contain these starting values. The formula for Newton's method that is contained in the `for` loop is stated in terms of a "vector" division; this means that the entries in vector `x` are operated on independently, so that the final value of `x` is a list of roots corresponding to the various starting values.

In general, if the starting values are not chosen appropriately, Newton's method might not converge. The `for` loop places an upper bound on the number of iterations that can be performed. On the other hand, when the iterations are essentially repeating themselves, it is appropriate to stop; in that event, the `if` statement causes execution to break out of the loop.

```
%    NEWTON.M
%    Use Newton's method to find the roots of a function f.
%    As new iterates are computed, they are added to the
%    bottom of the array named iters.
f = 'x.^3 - 4*x.^2 + 3*x + 1';
fprime = '3*x.^2 - 8*x + 3';

x = [0 1.5 3];
iters = x;
for k = 1:20,
    xnew = x - eval(f) ./ eval(fprime);
    iters = [iters; xnew];
    if max(abs( (xnew - x)./xnew )) < 10*eps, break, end;
    x = xnew;
end

%    Switch to long format, display the final answer, and then
%    turn off the long format.
format long, iters, format
```

3. Efficiency

The efficiency of a Matlab program can be greatly enhanced by using array operations instead of loops as much as possible. For example, to evaluate the function $y = x^2 + \sin x$ for all x in a given vector `x`, do not use a loop over the components of `x`, but instead use the command `y = x.^2 + sin(x);` . A different example involving matrix operations is the following.

Example 3.1. Create a 10000×2 matrix of random numbers, and then form the dot product of the two columns. In the following sequence of commands, the dot product is computed in a style that is common in programs written in languages such as Fortran 77. The function `cputime` is used to determine the execution time. During repeated tests on a Sun Sparcstation 2, the computation of the inner product required about 2.7 seconds.

```
n=10000; A = rand(n,2);
start = cputime;
x=0;
for k = 1:n,
    x = x + A(k,1)*A(k,2);
end
cputime-start
```

The following commands use the "vector" capabilities of Matlab. On a Sparcstation 2, the execution time is between 0.1 and 0.2 seconds.

```
start = cputime;
c1 = A(:,1); c2 = A(:,2); c2'*c1
cputime-start
```

Eigenvalues

1. The `eig` function
2. Comparison with roots of the characteristic polynomial
3. Sensitivity of roots of polynomials
4. The QR method

1. The `eig` function

Suppose that A is a square matrix. The command `eig(A)` yields the eigenvalues of A. The command `[V,D] = eig(A)` yields a diagonal matrix D whose diagonal entries are the eigenvalues of A and a matrix V whose columns are corresponding eigenvectors. The columns of V are given in the same order as the diagonal entries of D; thus $AV = VD$.

Example 1.1. Generate a matrix of normally distributed pseudo-random real numbers, compute the eigenvalues, and then plot the eigenvalues as points in the complex plane. Complex eigenvalues of a real matrix occur in complex conjugate pairs, and this is illustrated by the symmetry of the graph about the horizontal axis.

```
n = 20;
A = randn(n);
v = eig(A);
plot( real(v), imag(v), 'o'), axis('equal')
```

Example 1.2. Now suppose that the pseudo-random numbers are approximately uniformly distributed in the interval $[0, 1]$; in the above code, the command `A = randn(n);` is replaced by `A = rand(n);` . The distribution of eigenvalues is very different. Try this for larger values of n, e.g., $n = 50$, $n = 80$, $n = 100$, and see the pattern that develops. A heuristic for understanding the pattern is the following. For uniformly distributed numbers in the interval $[0, 1]$, the expected value is 0.5. If each entry of an $n \times n$ matrix is 0.5, then $n/2$ is an eigenvalue with multiplicity one, and zero is an eigenvalue with multiplicity $n - 1$.

2. Comparison with roots of the characteristic polynomial

The `eig` function uses the QR method to compute eigenvalues and eigenvectors. (See Section 4.) In principle, the eigenvalues could also be found by computing the roots of the characteristic polynomial $\det(A - \lambda I)$. However, the latter approach is not very useful for numerical computations. This is illustrated by the following example.

Example 2.1. Compute the eigenvalues of the $n \times n$ tridiagonal matrix A which has 2's on the main diagonal, -1's immediately above and below the main diagonal, and zeros elsewhere. The exact eigenvalues are known to have the form $\lambda_k = 2 + 2\cos(k\pi/(n+1))$ for $1 \le k \le n$.

In the following M-file `evmethods.m`, the command `p = poly(A)` produces a vector containing the coefficients of the characteristic polynomial of A. The coefficients are listed in order of decreasing powers. The command `roots(p)` then yields the roots of that

polynomial. The algorithm used by the function `roots` is actually based on eigenvalues. For any polynomial, there is a "companion matrix" whose eigenvalues are the roots of that polynomial. The function `roots` sets up the companion matrix of the polynomial, and then uses the `eig` function to compute eigenvalues. In effect, the present example compares the following two methods: (1) apply the `eig` function to A; (2) apply the `eig` function to the companion matrix of the characteristic polynomial of A.

If $n \geq 22$, the computed roots of the characteristic polynomial are complex, when computed on a Sun workstation in double precision arithmetic. However, the matrix A is real and symmetric, so its eigenvalues must be real. For smaller values of n, the computed roots of the characteristic polynomial are real, but the larger eigenvalues contain substantial errors. For large values of n (e.g., $n = 40$ or $n = 50$), the computed roots of the characteristic polynomial display an interesting pattern in the complex plane.

```
%    EVMETHODS.M
%    Compare two methods for computing eigenvalues of a matrix.
%    (1) Use Matlab's eig function.
%    (2) Compute roots of the characteristic polynomial.
%    USAGE: At the console, define
%        n = dimension of the test matrix

%    Define the test matrix.
B = diag(ones(n-1,1), 1);
A = 2*eye(n) - B - B' ;

%    Compute exact eigenvalues and sort in increasing order.
z = pi/(n+1) :  pi/(n+1) :  pi*n/(n+1) ;
exact = sort(2 + 2*cos(z)); exact = exact(:);

%    Compute eigenvalues using Matlab's eig function,
%    and sort in increasing order.
xeig = sort(eig(A)); xeig = xeig(:);

%    Compute roots of the characteristic polynomial.
p = poly(A);
xpoly = sort(roots(p)); xpoly = xpoly(:);

%    Display results.
label = 'Eigenvalues from eig, error, e-values from poly, error'
[xeig, abs(xeig-exact), xpoly, abs(xpoly-exact)]

%    Plot the computed eigenvalues in the complex plane.
plot(real(xeig), imag(xeig), 'o', real(xpoly), imag(xpoly), 'x')
axis('equal')
```

3. Sensitivity of roots of polynomials

In Example 2.1, the basic difficulty is that the roots of a high-degree polynomial can be very sensitive to perturbations in the coefficients of the polynomial. In other words, the root-finding problem is "ill-conditioned". Perturbations can always be expected in finite-precision computations, so the characteristic polynomial is generally not used to compute eigenvalues of matrices.

Example 3.1. The ill-conditioning of roots of polynomials can be illustrated with the polynomial $p_n(x) = (x-1)(x-2)\cdot\ldots\cdot(x-n)$. In a Matlab session, define a value of n, and then execute the commands

```
v = poly(1:n); roots(v)
```

The first command generates the coefficients of p_n, when it is expressed in the form $p_n(x) = x^n + a_{n-1}x^{n-1} + \ldots + a_1x + a_0$. The second command computes the roots of the polynomial. To see all of the digits computed by the machine, type `format long` at the Matlab prompt.

One would expect that command `roots(v)` should return the integers 1, 2, ..., n. However, this is not exactly the case unless n is quite small. On a Sun workstation using double precision arithmetic with about 16 decimal digits, the output of `roots(v)` is not exactly 1, 2, ..., n if $n \geq 4$. If $n = 10$, about five digits are lost in some of the roots; if $n = 20$, some of the roots have only about four good digits remaining; if $n = 21$, two of the computed roots are complex; if $n = 22$, ten of the computed roots are complex.

In Example 3.1, the unexpected behavior is due to the effects of roundoff errors that are generated during the course of the computation. One can think of these roundoff errors as generating small perturbations in the problem that is being solved; in some cases, these small perturbations have an enormous impact upon the final solution that is computed. This phenomenon can be visualized geometrically, as described in the following example.

Example 3.2. Consider a polynomial of the form $p(x) = x^n + a_{n-1}x^{n-1} + \ldots + a_1x + a_0$, and suppose that the coefficient a_{n-1} is perturbed by a small amount ϵ. The perturbed polynomial is then $\hat{p}(x) = x^n + (a_{n-1} + \epsilon)x^{n-1} + \ldots + a_1x + a_0 = p(x) + \epsilon x^{n-1}$. The perturbed polynomial $\hat{p}$ is obtained from p by moving the graph of p up or down by an amount ϵx^{n-1}. The quantity ϵ may be small, but the quantity ϵx^{n-1} could be huge, depending on the values of x and n. The roots of $p(x) = 0$ could be changed a great deal, and some could become complex.

These ideas can be illustrated by executing the following M-file, `pert.m` . For example, if $n = 7$ and $\epsilon = -.001$, then two of the roots are complex. If $n = 20$ and $\epsilon = 10^{-9}$, then six of the roots are complex. The case $n = 20$ was analyzed by J. H. Wilkinson.

```
%    PERT.M
%    Demonstrate some effects of perturbing the coefficient
%    of x^(n-1) in a polynomial of degree n.
%    USAGE: At the console, define the following:
%        n = degree of the polyomial
%        epsilon = amount to be added to the coefficient of x^(n-1)
```

```
v = poly(1:n);
vpert = v; vpert(2) = vpert(2) + epsilon;
r = roots(v);
rpert = roots(vpert);
[r(:)  rpert(:)]

%    Graph the unperturbed polynomial, the perturbed polynomial,
%    and the amount of perturbation in the function values.
%    The amount of perturbation is epsilon * x^(n-1).

x = 0 : .05 : n+1;
y = polyval(v,x);
ypert = polyval(vpert,x);
perturbation = epsilon * x.^(n-1);

%    Determine a vertical scale for the graph, and then plot.
maxy = max(abs(y(21:1+20*n)));
maxypert = max(abs(ypert(21:1+20*n)));
maxperturb = max(abs(perturbation(21:1+20*n)));
maxy = 1.1* max( [maxy, maxypert, maxperturb] );
plot(x, y, x, ypert, '--', x, perturbation, ':', ...
    x, zeros(size(x)),'-', 1:n, zeros(1,n), 'o')
axis([0 n+1 -maxy maxy])
title('Perturbations of roots of f(x) = (x-1)(x-2)...(x-n)')
```

4. The QR method

Extensive information about the QR method can be found in the book *Matrix Computations* by G. Golub and C. Van Loan and in various works referenced therein.

For a square matrix A, the basic QR method is defined as follows. Factor A into the product of an orthogonal matrix Q and an upper triangular ("right triangular") matrix R, and then reverse the order of the factors. Repeat this procedure over and over. In Matlab, the QR method can be implemented manually by repeating the statements `[Q,R] = qr(A); A = R*Q` . This algorithm generates a sequence of similar matrices that converges to a limiting matrix whose eigenvalues can be found easily. Because all of the matrices are similar, the eigenvalues do not change. If A is real and symmetric, the limiting matrix is diagonal. In general, the limiting matrix is triangular or nearly triangular.

The rate of convergence of the above method can be very slow. In order to improve the rate of convergence, the "QR method with shifts" is generally used in practice. This method can be implemented manually in Matlab by repeating the sequence `[Q,R] = qr(A - c*eye(n)); A = R*Q + c*eye(n)`. Different constants c can be used in different iterations. Golub and Van Loan state that the constants c can be taken to be approximate eigenvalues of A.

Before the QR iteration is started, the matrix A is usually transformed to Hessenberg form via an orthogonal similarity transformation. (A Hessenberg matrix is zero below the first subdiagonal.) If A is real and symmetric, the Hessenberg form is tridiagonal. The zeros are introduced in order make the QR factorization more efficient.