A Practical Introduction to Matlab

Mark S. Gockenbach*

Introduction

Matlab (**Mat**rix **lab**oratory) is an interactive software system for numerical computations and graphics. As the name suggests, Matlab is especially designed for matrix computations: solving systems of linear equations, computing eigenvalues and eigenvectors, factoring matrices, and so forth. In addition, it has a variety of graphical capabilities, and can be extended through programs written in its own programming language. Many such programs come with the system; a number of these extend Matlab's capabilities to nonlinear problems, such as the solution of initial value problems for ordinary differential equations.

Matlab is designed to solve problems numerically, that is, in finite-precision arithmetic. Therefore it produces approximate rather than exact solutions, and should not be confused with a symbolic computation system (SCS) such as Mathematica or Maple. It should be understood that this does not make Matlab better or worse than an SCS; it is a tool designed for different tasks and is therefore not directly comparable.

In the following sections, I give an introduction to some of the most useful features of Matlab. I include plenty of examples; the best way to learn to use Matlab is to read this in front of a computer, trying the examples and experimenting.

1 Simple calculations and graphs

In this section, I will give a quick introduction to defining numbers, vectors and matrices in Matlab, performing basic computations with them, and creating simple graphs. All of these topics will be revisited in greater depth in later sections.

1.1 Entering vectors and matrices; built-in variables and functions; help

The first thing to learn about Matlab is that everything is a matrix. This of course implies some limitations, but it is this simplicity of design that makes

*Department of Mathematics, University of Michigan

Matlab so easy to use.

The following commands show how to enter numbers, vectors and matrices, and assign them to variables (>> is the Matlab prompt on my computer; it may be different with different computers or different versions of Matlab. I am using version 4.2c. On my Unix workstation, I start Matlab by typing matlab at the Unix prompt.):

```
>> a = 2
a =
     2
>> x = [1;2;3]
x =
     1
     2
     3
>> A = [1 2 3;4 5 6;7 8 0]
A =
     1     2     3
     4     5     6
     7     8     0
```

Notice that the rows of a matrix are separated by semicolons, while the entries on a row are separated by spaces (or commas).

A useful command is "whos", which displays the names of all defined variables and their types:

```
>> whos
              Name        Size       Elements     Bytes    Density   Complex

                 A       3 by 3            9        72       Full        No
                 a       1 by 1            1         8       Full        No
                 x       3 by 1            3        24       Full        No

Grand total is 13 elements using 104 bytes
```

Note that, as a mentioned above, each of these three variables is regarded as a matrix by Matlab. The scalar `a` is a 1×1 matrix, the vector `x` is a 3×1 matrix, and, of course, `A` is a 3×3 matrix (see the "size" entry for each variable).

As the "density" and "complex" entries suggest, Matlab allows both sparse and complex matrices. I will discuss sparse matrices later. The complex unit $i = \sqrt{-1}$ is represented by either of the built-in variables `i` or `j`:

```
>> sqrt(-1)
ans =
        0 + 1.0000i
```

This example shows how complex numbers are displayed in Matlab; it also shows that the square root function is a built-in feature.

The result of the last calculation not assigned to a variable is automatically assigned to the variable `ans`, which can then be used as any other variable in subsequent computations. Here is an example:

```
>> 100^2-4*2*3
ans =
        9976
>> sqrt(ans)
ans =
   99.8799
>> (-100+ans)/4
ans =
   -0.0300
```

The arithmetic operators work as expected for scalars. A built-in variable that is often useful is π:

```
>> pi
ans =
    3.1416
```

Above I pointed out that the square root function is built-in; other common scientific functions, such as sine, cosine, tangent, exponential, and logarithm are also pre-defined. For example:

```
>> cos(.5)^2+sin(.5)^2
ans =
     1
>> exp(1)
ans =
    2.7183
>> log(ans)
ans =
     1
```

Other elementary functions, such as hyperbolic and inverse trigonometric functions, are also defined.

At this point, rather than providing a comprehensive list of functions available in Matlab, I want to explain how to get this information from Matlab itself. An extensive online help system can be accessed by commands of the form `help <command-name>`. For example:

```
>> help ans

 ANS    The most recent answer.
        ANS is the variable created automatically when expressions
        are not assigned to anything else. ANSwer.
```

```
>> help pi

 PI     3.1415926535897....
        PI = 4*atan(1) = imag(log(-1)) = 3.1415926535897....
```

A good place to start is with the command **help help**, which explains how the help systems works, as well as some related commands. Typing **help** by itself produces a list of topics for which help is available; looking at this list we find the entry "elfun—elementary math functions." Typing **help elfun** produces a list of the math functions available. We see, for example, that the inverse tangent function (or arctangent) is called **atan**:

```
>> pi-4*atan(1)
ans =
     0
```

It is often useful, when entering a matrix, to suppress the display; this is done by ending the line with a semicolon (see the first example in the next section).

1.2 Graphs

The simplest graphs to create are plots of points in the cartesian plane. For example:

```
>> x = [1;2;3;4;5];
>> y = [0;.25;3;1.5;2];
>> plot(x,y)
```

The resulting graph is displayed in Figure 1. Notice that, by default, Matlab connects the points with straight line segments. An alternative is the following (see Figure 2):

```
>> plot(x,y,'o')
```

1.3 Arithmetic operations on matrices

Matlab can perform the standard arithmetic operations on matrices (and therefore also on vectors and scalars, since these are regarded as matrices): addition, subtraction, and multiplication. In addition, Matlab defines a notion of matrix division as well as "vectorized" operations.

1.3.1 Standard operations

If `A` and `B` are matrices, then Matlab can compute `A+B`, `A-B`, and `A*B`, *when these operations are defined.* For example, consider the following commands:

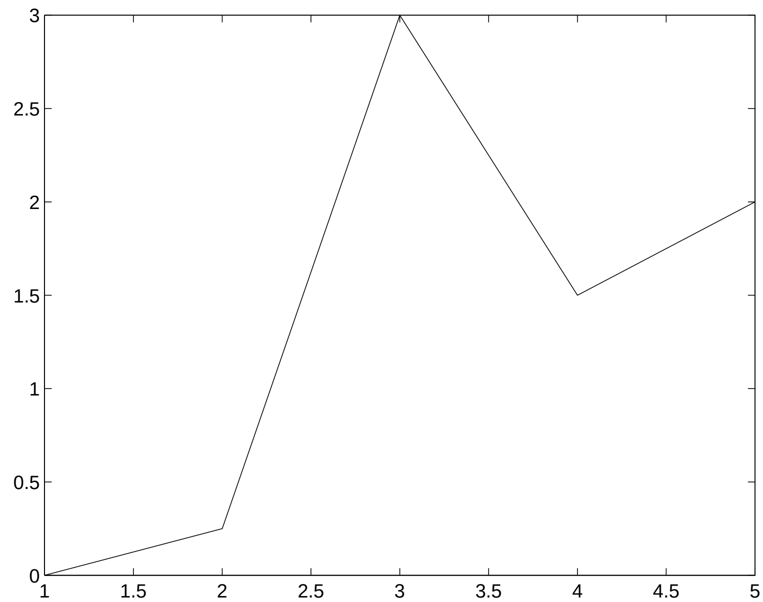

Figure 1: A simple Matlab graph

```
>> A = [1 2 3;4 5 6;7 8 9];
>> B = [1 1 1;2 2 2;3 3 3];
>> C = [1 2;3 4;5 6];
>> whos
              Name        Size       Elements     Bytes    Density   Complex

                 A       3 by 3             9        72       Full        No
                 B       3 by 3             9        72       Full        No
                 C       3 by 2             6        48       Full        No

Grand total is 24 elements using 192 bytes

>> A+B
ans =
     2     3     4
     6     7     8
    10    11    12
>> A+C
??? Error using ==> +
Matrix dimensions must agree.
```

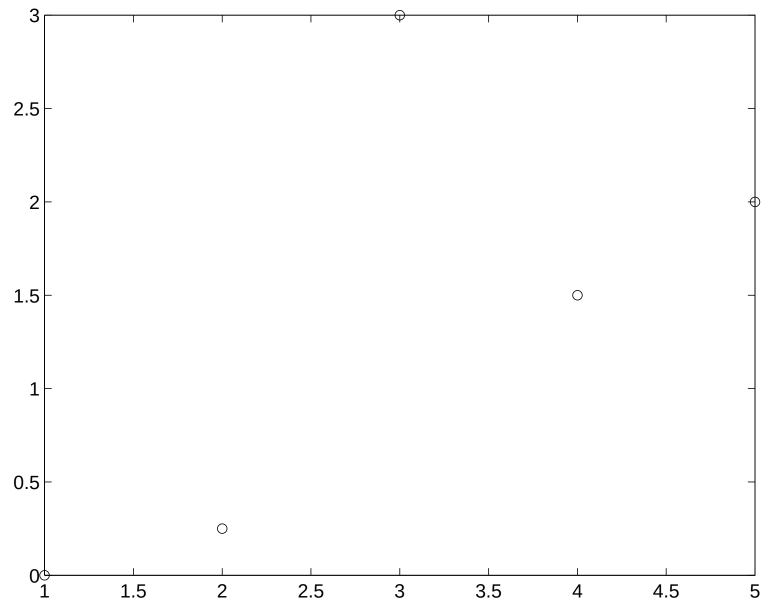

Figure 2: Another simple Matlab graph

```
>> A*C
ans =
    22    28
    49    64
    76   100
>> C*A
??? Error using ==> *
Inner matrix dimensions must agree.
```

If `A` is a square matrix and `m` is a positive integer, then `A^m` is the product of `m` factors of `A`.

1.3.2 Solving matrix equations using matrix division

If A is a square, nonsingular matrix, then the solution of the equation $Ax = b$ is $x = A^{-1}b$. Matlab implements this operation with the backslash operator:

```
>> A = rand(3,3)
A =
    0.2190    0.6793    0.5194
    0.0470    0.9347    0.8310
```

```
    0.6789    0.3835    0.0346
>> b = rand(3,1)
b =
    0.0535
    0.5297
    0.6711
>> x = A\b
x =
 -159.3380
  314.8625
 -344.5078
>> A*x-b
ans =
   1.0e-13 *
   -0.2602
   -0.1732
   -0.0322
```

(Notice the use of the built-in function **rand**, which creates a matrix with entries from a uniform distribution on the interval $(0, 1)$. See **help rand** for more details.) Thus `A\b` is (mathematically) equivalent to multiplying b on the left by A^{-1} (however, Matlab does *not* compute the inverse matrix; instead it solves the linear system directly). When used with a nonsquare matrix, the backslash operator solves the appropriate system in the least-squares sense; see **help slash** for details. Of course, as with the other arithmetic operators, the matrices must be compatible in size.

1.3.3 Vectorized functions and operators; more on graphs

Matlab has many commands to create special matrices; the following command creates a row vector whose components increase arithmetically:

```
>> t = 1:5
t =
     1     2     3     4     5
```

The components can change by non-unit steps:

```
>> x = 0:.1:1
x =
  Columns 1 through 7
         0    0.1000    0.2000    0.3000    0.4000    0.5000    0.6000
  Columns 8 through 11
    0.7000    0.8000    0.9000    1.0000
```

A negative step is also allowed.

A vector of this sort can be regarded as defining a one-dimensional grid, which is useful for graphing functions. To create a graph of $y = f(x)$ (or, to be precise, to graph points of the form $(x, f(x))$ and connect them with line segments), one can create a grid in the vector **x** and then create a vector **y** with the corresponding function values.

It is easy to create the needed vectors to graph a built-in function, since Matlab functions are vectorized. This means that if a built-in function such as sine is applied to a matrix, the effect is create a new matrix of the same size whose entries are the function values of the entries of the original matrix. For example (see Figure 3):

```
>> x = (0:.1:2*pi);
>> y = sin(x);
>> plot(x,y)
```

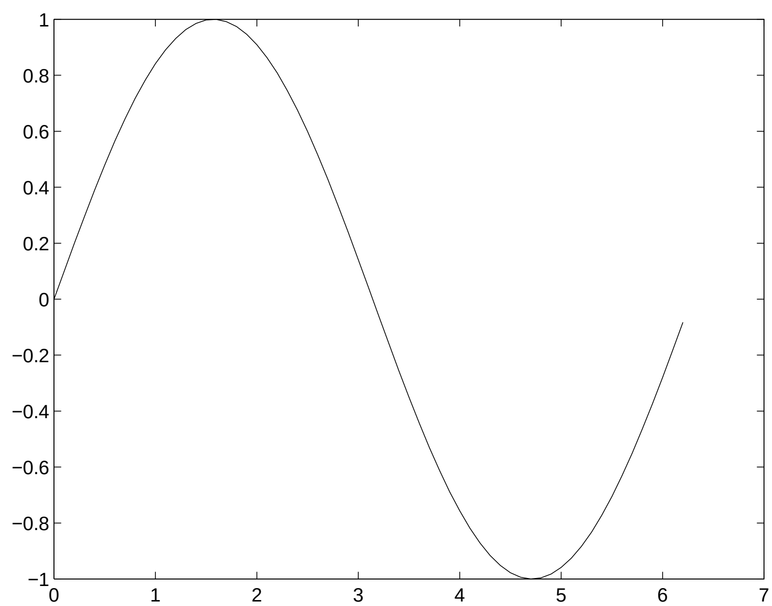

Figure 3: Graph of $y = sin(x)$

Matlab also provides vectorized arithmetic operators, which are the same as the ordinary operators, preceded by ".". For example, to graph $y = x/(1 + x^2)$:

```
>> x = (-5:.1:5);
>> y = x./(1+x.^2);
>> plot(x,y)
```

(the graph is not shown). Thus `x.^2` squares each component of `x`, and `x./z` divides each component of `x` by the corresponding component of `z`. Addition and subtraction are performed component-wise by definition, so there are no ".+" or ".-" operators. Note the difference between `A^2` and `A.^2`. The first is only defined if `A` is square, while the second is defined for any matrix (or vector) `A`.

1.4 Some miscellaneous commands

An important operator in Matlab is the single quote, which represents the (conjugate) transpose:

```
>> A = [1 2;3 4]
A =
     1     2
     3     4
>> A'
ans =
     1     3
     2     4
>> B = A + i*.5*A
B =
   1.0000 + 0.5000i   2.0000 + 1.0000i
   3.0000 + 1.5000i   4.0000 + 2.0000i
>> B'
ans =
   1.0000 - 0.5000i   3.0000 - 1.5000i
   2.0000 - 1.0000i   4.0000 - 2.0000i
```

In the rare event that the transpose, rather than the conjugate transpose, is needed, the "`.'`" operator is used:

```
>> B.'
ans =
   1.0000 + 0.5000i   3.0000 + 1.5000i
   2.0000 + 1.0000i   4.0000 + 2.0000i
```

(note that `'` and `.'` are equivalent for matrices with real entries).

The following commands are frequently useful; more information can be obtained from the on-line help system.

Creating matrices

- `zeros(m,n)` creates an $m \times n$ matrix of zeros;
- `ones(m,n)` creates an $m \times n$ matrix of ones;
- `eye(n)` creates the $n \times n$ identity matrix;
- `diag(v)` (assuming `v` is an n-vector) creates an $n \times n$ diagonal matrix with `v` on the diagonal.

formatting display and graphics

- format
  - `format short` 3.1416
  - `format short e` 3.1416e+00
  - `format long` 3.14159265358979
  - `format long e` 3.141592653589793e+00
  - `format compact` suppresses extra line feeds (all of the output in this paper is in compact format).
- `xlabel('string'), ylabel('string')` label the horizontal and vertical axes, respectively, in the current plot;
- `title('string')` add a title to the current plot;
- `axis([a b c d])` change the window on the current graph to $a \leq x \leq b, c \leq y \leq d$;
- `grid` add a rectangular grid to the current plot;
- `hold on` freezes the current plot so that subsequent graphs will be displayed with the current;
- `hold off` releases the current plot; the next plot will erase the current before displaying;
- `subplot` puts multiple plots in one graphics window.

Miscellaneous

- `max(x)` returns the largest entry of x, if x is a vector; otherwise returns a row vector with the largest entry from each column of the matrix x;
- `min(x)` analogous to `max`;
- `size(A)` returns a 1×2 vector with the number of rows and columns of A;
- `length(x)` returns the "length" of the matrix, i.e. `max(size(A))`.
- `save fname` saves the current variables to the file named fname.mat;
- `load fname` load the variables from the file named fname.mat;
- `quit` exits Matlab

2 Programming in Matlab

The capabilities of Matlab can be extended through programs written in its own, simple programming language. It provides the standard constructs, such as loops and conditionals; these constructs can be used interactively to reduce the tedium of repetitive tasks, or collected in programs stored in "m-files" (nothing more than a text file with extention ".m"). I will first discuss the programming mechanisms and then explain how to write programs.

2.1 Conditionals and loops

Matlab has a standard if-elseif-else conditional; for example:

```
>> t = rand(1);
>> if t > 0.75
      s = 0;
   elseif t < 0.25
      s = 1;
   else
      s = 1-2*(t-0.25);
   end
>> s
s =
     0
>> t
t =
    0.7622
```

The logical operators in Matlab are <, >, <=, >=, == (logical equals), and ~= (not equal). These are binary operators which return the values 0 and 1 (for scalar arguments):

```
>> 5>3
ans =
     1
>> 5<3
ans =
     0
>> 5==3
ans =
     0
```

Thus the general form of the if statement is

if *expr*
  statements
end;

the statements execute if *expr* is nonzero.

Matlab provides two types of loops, a for-loop (comparable to a Fortran do-loop or a C for-loop) and a while-loop. A for-loop repeats the statements in the loop as the loop index takes on the values in a given row vector:

```
>> for i=[1,2,3,4]
      disp(i^2)
   end
     1
     4
     9
    16
```

(Note the use of the built-in function `disp`, which simply displays its argument.) The loop, like an if-block, must be terminated by `end`. This loop would more commonly be written as

```
>> for i=1:4
      disp(i^2)
   end
     1
     4
     9
    16
```

(recall that `1:4` is the same as `[1,2,3,4]`).

The while-loop repeats as long as the given expr is true (nonzero):

```
>> x=1;
>> while 1+x > 1
      x = x/2;
   end
>> x
x =
   1.1102e-16
```

2.2 Scripts and functions

A script is simply a collection of Matlab commands in an m-file (a text file whose name ends in the extension ".m"). Upon typing the name of the file (without the extension), those commands are executed as if they had been entered at the keyboard. The m-file must be located in one of the directories in which Matlab automatically looks for m-files; a list of these directories can be obtained by the command `path`. (See `help path` to learn how to add a directory to this list.) One of the directories in which Matlab always looks is the *current working directory*; the command `cd` identifies the current working directory, and `cd newdir` changes the working directory to `newdir`.

For example, suppose that plotsin.m contains the lines

```
x = 0:2*pi/N:2*pi;
y = sin(w*x);
plot(x,y)
```

Then the sequence of commands

```
>> N=100;w=5;
>> plotsin
```

produces Figure 2.2.

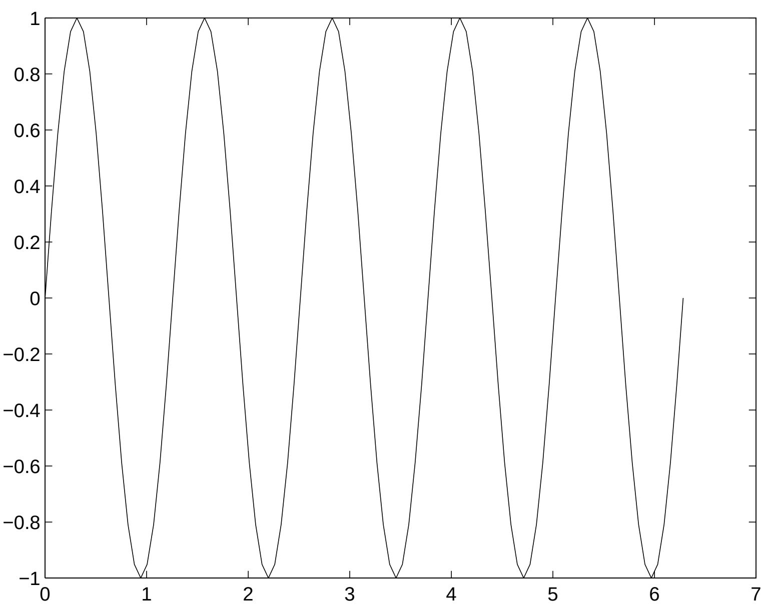

Figure 4: Effect of an m-file

As this example shows, the commands in the script can refer to the variables already defined in Matlab, which are said to be in the global workspace (notice the reference to `N` and `w` in plotsin.m). As I mentioned above, the commands in the script are exectuted exactly as if they had been typed at the keyboard.

Much more powerful than scripts are functions, which allow the user to create new Matlab commands. A function is defined in an m-file that begins with a line of the following form:

```
function [output1,output2,...]  = cmd_name(intput1,input2,...)
```

The rest of the m-file consists of ordinary Matlab commands computing the values of the outputs and performing other desired actions. It is important to note that when a function is invoked, Matlab creates a local workspace. The commands in the function cannot refer to variables from the global (interactive) workspace unless they are passed as inputs. By the same token, variables created as the function executes are erased when the execution of the function ends, unless they are passed back as outputs.

Here is a simple example of a function; it computes the function $f(x) = sin(x^2)$. The following commands should be stored in the file `fcn.m` (the name of the function within Matlab is the name of the m-file, without the extension):

```
function y = fcn(x)
y = sin(x.^2);
```

(Note that I used the vectorized operator `.^` so that the function `fcn` is also vectorized.) With this function defined, I can now use `fcn` just as the built-in function `sin`:

```
>> x = (-pi:2*pi/100:pi)';
>> y = sin(x);
>> z = fcn(x);
>> plot(x,y,x,z)
>> grid
```

The graph is shown in Figure 2.2. Notice how `plot` can be used to graph two (or more) functions together. The computer will display the curves with different line types—different colors on a color monitor, or different styles (e.g. solid versus dashed) on a black-and-white monitor. See `help plot` for more information. Note also the use of the `grid` command to superimpose a cartesian grid on the graph.

2.3 A nontrivial example

Notice from Figure 2.2 that $f(x) = sin(x^2)$ has a root between 1 and 2 (of course, this root is $x = \sqrt{\pi}$, but we feign ignorance for a moment). A general algorithm for nonlinear root-finding is the method of bisection, which takes a function and an interval on which function changes sign, and repeatedly bisects the interval until the root is trapped in a very small interval.

A function implementing the method of bisection illustrates many of the important techniques of programming in Matlab. The first important technique, without which a useful bisection routine cannot be written, is the ability to pass the name of one function to another function. In this case, `bisect` needs to know the name of the function whose root it is to find. This name can be passed as a string (the alternative is to "hard-code" the name in `bisect.m`, which means that each time one wants to use `bisect` with a different function, the file `bisect.m` must be modified. This style of programming is to be avoided.).

The built-in function `feval` is needed to evaluate a function whose name is known (as a string). Thus, interactively

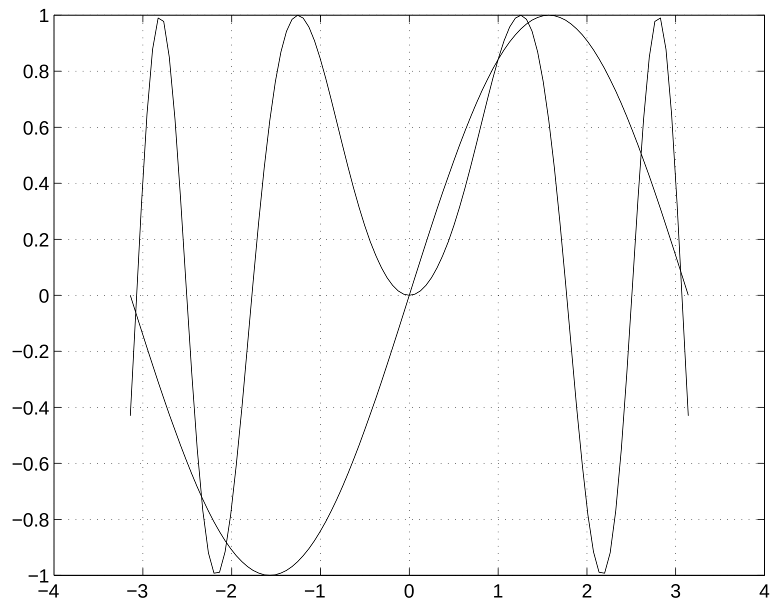

Figure 5: Two curves graphed together

```
>> fcn(2)
ans =
   -0.7568
```

and

```
>> feval('fcn',2)
ans =
   -0.7568
```

are equivalent (notice that single quotes are used to delimit a string). A variable can also be assigned the value of a string:

```
>> str = 'fcn'
str =
f
>> feval(str,2)
ans =
   -0.7568
```

This does not violate my earlier statement that everything is a matrix, since Matlab stores a string variable as an array of ASCII values (it also must store a flag indicating whether these numbers are to be interpreted as ASCII; see `help strings`).

The following Matlab program uses the string facility to pass the name of a function to `bisect`. A % sign indicates that the rest of the line is a comment.

```
function c = bisect(fn,a,b,tol)

% c = bisect('fn',a,b,tol)
%
%   This function locates a root of the function fn on the interval
%   [a,b] to within a tolerance of tol.  It is assumed that the function
%   has opposite signs at a and b.

% Evaluate the function at the endpoints and check to see if it
% changes sign.

fa = feval(fn,a);
fb = feval(fn,b);

if fa*fb >= 0
   error('The function must have opposite signs at a and b')
end

% The flag done is used to flag the unlikely event that we find
% the root exactly before the interval has been sufficiently reduced.

done = 0;

% Bisect the interval

c = (a+b)/2;

% Main loop

while abs(a-b) > 2*tol & ~done

   % Evaluate the function at the midpoint

   fc = feval(fn,c);

   if fa*fc < 0       % The root is to the left of c
      b = c;
      fb = fc;
      c = (a+b)/2;
   elseif fc*fb < 0   % The root is to the right of c
      a = c;
```

```
      fa = fc;
      c = (a+b)/2;
   else                % We landed on the root
      done = 1;
   end

end
```

Assuming that this file is named `bisect.m`, it can be run as follows:

```
>> x = bisect('fcn',1,2,1e-6)
x =
    1.7725
>> sqrt(pi)-x
ans =
  -4.1087e-07
```

Not only can new Matlab commands be created with m-files, but the help system can be automatically extended. The `help` command will print the first comment block from an m-file:

```
>> help bisect

  c = bisect('fn',a,b,tol)

    This function locates a root of the function fn on the interval
    [a,b] to within a tolerance of tol.  It is assumed that the function
    has opposite signs at a and b.
```

(Something that may be confusing is the use of both `fn` and `'fn'` in `bisect.m`. I put quotes around `fn` in the comment block to remind the user that a string must be passed. However, the variable `fn` is a string *variable* and does not need quotes in any command line.)

Notice the use of the `error` function near the beginning of the program. This function displays the string passed to it and exits the m-file.

At the risk of repeating myself, I want to re-emphasize a potentially troublesome point. In order to execute an m-file, Matlab must be able to find it, which means that it must be found in a directory in Matlab's path. The current working directory is always on the path; to display or change the path, use the `path` command. To display or change the working directory, use the `cd` command. As usual, `help` will provide more information.

3 Advanced matrix computations

3.1 Eigenvalues and other numerical linear algebra computations

In addition to solving linear systems (with the backslash operator), Matlab performs many other matrix computations. Among the most useful is the computation of eigenvalues and eigenvectors with the `eig` command. If `A` is a square matrix, then `ev = eig(A)` returns the eigenvalues of `A` in a vector, while `[V,D] = eig(A)` returns the spectral decompostion of `A`: `V` is a matrix whose columns are eigenvectors of A, while `D` is a diagonal matrix whose diagonal entries are eigenvalues. The equation `AV = VD` holds. If `A` is diagonalizable, then `V` is invertible, while if `A` is symmetric, then `V` is orthogonal ($V^TV = I$).

Here is an example:

```
>> A = [1 3 2;4 5 6;7 8 9]
A =
     1     3     2
     4     5     6
     7     8     9
>> eig(A)
ans =
  15.9743
  -0.4871 + 0.5711i
  -0.4871 - 0.5711i
>> [V,D] = eig(A)
V =
  -0.2155              0.0683 + 0.7215i   0.0683 - 0.7215i
  -0.5277             -0.3613 - 0.0027i  -0.3613 + 0.0027i
  -0.8216              0.2851 - 0.5129i   0.2851 + 0.5129i
D =
  15.9743                   0                  0
        0             -0.4871 + 0.5711i        0
        0                   0            -0.4871 - 0.5711i
>> A*V-V*D
ans =
   1.0e-14 *
  -0.0888              0.0777 - 0.1998i   0.0777 + 0.1998i
        0             -0.0583 + 0.0666i  -0.0583 - 0.0666i
        0             -0.0555 + 0.2387i  -0.0555 - 0.2387i
```

There are many other matrix functions in Matlab, many of them related to matrix factorizations. Some of the most useful are:

- `lu` computes the LU factorization of a matrix;
- `chol` computes the Cholesky factorization of a symmetric positive definite matrix;

- `qr` computes the QR factorization of a matrix;
- `svd` computes the singular values or singular value decomposition of a matrix;
- `cond, condest, rcond` computes or estimates various condition numbers;
- `norm` computes various matrix or vector norms;

3.2 Sparse matrix computations

Matlab has the ability to store and manipulate sparse matrices, which greatly increases its usefulness for realistic problems. Creating a sparse matrix can be rather difficult, but manipulating them is easy, since the same operators apply to both sparse and dense matrices. In particular, the backslash operator works with sparse matrices, so sparse systems can be solved in the same fashion as dense systems. Some of the built-in functions apply to sparse matrices, but others do not (for example, `eig` can be used on sparse symmetric matrix, but not on a sparse nonsymmetric matrix).

3.2.1 Creating a sparse matrix

If a matrix `A` is stored in ordinary (dense) format, then the command `S = sparse(A)` creates a copy of the matrix stored in sparse format. For example:

```
>> A = [0 0 1;1 0 2;0 -3 0]
A =
     0     0     1
     1     0     2
     0    -3     0
>> S = sparse(A)
S =
   (2,1)        1
   (3,2)       -3
   (1,3)        1
   (2,3)        2
>> whos
              Name        Size       Elements     Bytes    Density   Complex

                 A       3 by 3             9        72       Full        No
                 S       3 by 3             4        60     0.4444        No

Grand total is 13 elements using 132 bytes
```

Unfortunately, this form of the `sparse` command is not particularly useful, since if `A` is large, it can be very time-consuming to first create it in dense format. The command `S = sparse(m,n)` creates an $m \times n$ zero matrix in sparse format. Entries can then be added one-by-one:

```
>> A = sparse(3,2)
A =
   All zero sparse: 3-by-2
>> A(1,2)=1;
>> A(3,1)=4;
>> A(3,2)=-1;
>> A
A =
   (3,1)        4
   (1,2)        1
   (3,2)       -1
```

(Of course, for this to be truly useful, the nonzeros would be added in a loop.)

Another version of the `sparse` command is `S = sparse(I,J,S,m,n,maxnz)`. This creates an $m \times n$ sparse matrix with $(I(k), J(k))$ entry equal to $S(k), k = 1, \ldots, length(S)$. The optional argument **`maxnz`** causes Matlab to pre-allocate storage for **`maxnz`** nonzero entries, which can increase efficiency in the case when more nonzeros will be added later to `S`.

There are still more version of the `sparse` command. See `help sparse` for details.

The most common type of sparse matrix is a banded matrix, that is, a matrix with a few nonzero diagonals. Such a matrix can be created with the `spdiags` command. Consider the following matrix:

```
>> A
A =
    64   -16     0   -16     0     0     0     0     0
   -16    64   -16     0   -16     0     0     0     0
     0   -16    64     0     0   -16     0     0     0
   -16     0     0    64   -16     0   -16     0     0
     0   -16     0   -16    64   -16     0   -16     0
     0     0   -16     0   -16    64     0     0   -16
     0     0     0   -16     0     0    64   -16     0
     0     0     0     0   -16     0   -16    64   -16
     0     0     0     0     0   -16     0   -16    64
```

This is a 9×9 matrix with 5 nonzero diagonals. In Matlab's indexing scheme, the nonzero diagonals of A are numbers -3, -1, 0, 1, and 3 (the main diagonal is number 0, the first subdiagonal is number -1, the first superdiagonal is number 1, and so forth). To create the same matrix in sparse format, it is first necessary to create a 9×5 matrix containing the nonzero diagonals of A. Of course, the diagonals, regarded as column vectors, have different lengths; only the main diagonal has length 9. In order to gather the various diagonals in a single matrix, the shorter diagonals must be padded with zeros. The rule is that the extra zeros go at the bottom for subdiagonals and at the top for superdiagonals. Thus we create the following matrix:

```
>> B = [
   -16   -16    64     0     0
   -16   -16    64   -16     0
   -16     0    64   -16     0
   -16   -16    64     0   -16
   -16   -16    64   -16   -16
   -16     0    64   -16   -16
     0   -16    64     0   -16
     0   -16    64   -16   -16
     0     0    64   -16   -16
];
```

(notice the technique for entering the rows of a large matrix on several lines). The `spdiags` command also needs the indices of the diagonals:

```
>> d = [-3,-1,0,1,3];
```

The matrix is then created as follows:

```
S = spdiags(B,d,9,9);
```

The last two arguments give the size of `S`.

Perhaps the most common sparse matrix is the identity. Recall that an identity matrix can be created, in dense format, using the command `eye`. To create the $n \times n$ identity matrix in sparse format, use `I = speye(n)`.

Another useful command is `spy`, which creates a graphic displaying the sparsity pattern of a matrix. For example, the above penta-diagonal matrix `A` can be displayed by the following command; see Figure 3.2.1:

```
>> spy(A)
```

4 Advanced Graphics

Matlab can produce several different types of graphs: 2D curves, 3D surfaces, contour plots of 3D surfaces, parametric curves in 2D and 3D. I will leave the reader to find most of the details from the online help system. Here I want to show, by example, some of the possibilities. I will also explain the basics of producing 3D plots.

4.1 Putting several graphs in one window

The subplot command creates several plots in a single window. To be precise, `subplot(m,n,i)` creates mn plots, arranged in an array with m rows and n columns. It also sets the next plot command to go to the ith coordinate system (counting across the rows. Here is an example (see Figure 4.1):

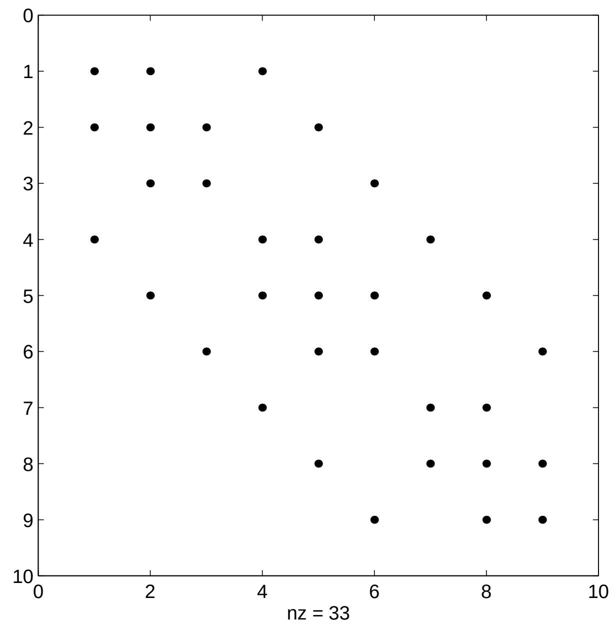

Figure 6: The sparsity pattern of a matrix

```
>> t = (0:.1:2*pi)';
>> subplot(2,2,1)
>> plot(t,sin(t))
>> subplot(2,2,2)
>> plot(t,cos(t))
>> subplot(2,2,3)
>> plot(t,exp(t))
>> subplot(2,2,4)
>> plot(t,1./(1+t.^2))
```

4.2 3D plots

In order to create a graph of a surface in 3-space (or a contour plot of a surface), it is necessary to evaluate the function on a regular rectangular grid. This can be done using the meshgrid command. First, create 1D vectors describing the grids in the x- and y-directions:

```
>> x = (0:2*pi/20:2*pi)';
>> y = (0:4*pi/40:4*pi)';
```

Next, "spread" these grids into two dimensions using meshgrid:

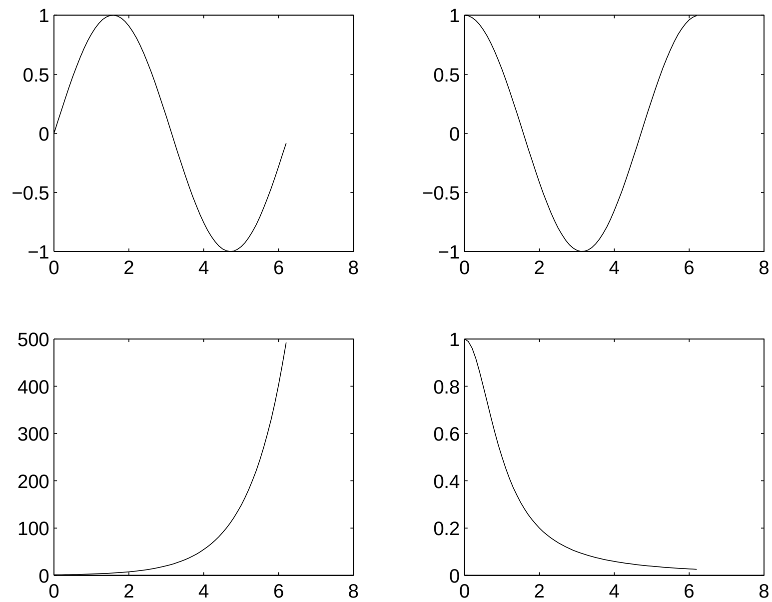

Figure 7: Using the `subplot` command

```
>> [X,Y] = meshgrid(x,y);
>> whos
              Name         Size          Elements     Bytes    Density   Complex

                 X       41 by 21             861      6888       Full        No
                 Y       41 by 21             861      6888       Full        No
                 x       21 by 1               21       168       Full        No
                 y       41 by 1               41       328       Full        No

Grand total is 1784 elements using 14272 bytes
```

The effect of meshgrid is to create a vector `X` with the x-grid along each row, and a vector `Y` with the y-grid along each column. Then, using vectorized functions and/or operators, it is easy to evaluate a function $z = f(x, y)$ of two variables on the rectangular grid:

```
>> z = cos(X).*cos(2*Y);
```

Having created the matrix containing the samples of the function, the surface can be graphed using either the `mesh` or the `surf` commands (see Figures 4.2 and 4.2, respectively):

```
>> mesh(x,y,z)
>> surf(x,y,z)
```

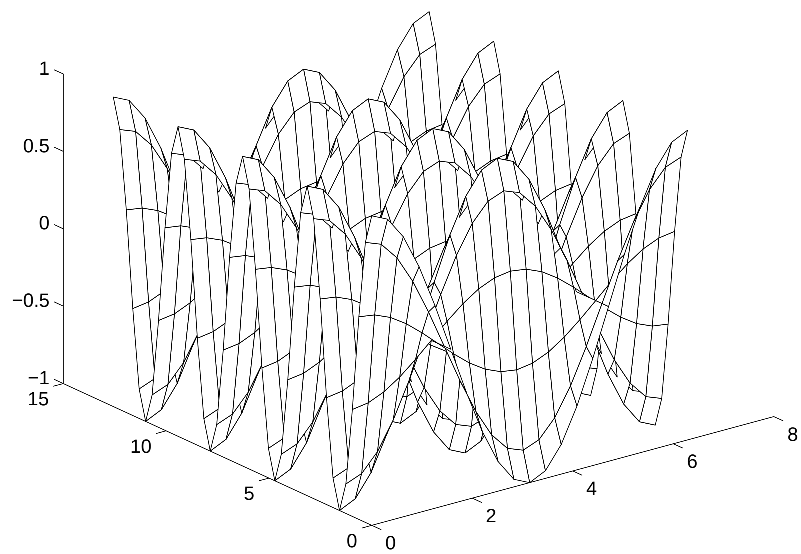

Figure 8: Using the **mesh** command

(The difference is that `surf` shades the surface, while `mesh` does not.) In addition, a contour plot can be created (see Figure 4.2):

```
>> contour(x,y,z)
```

Use the `help` command to learn the additional options. These commands can be very slow if the grid is fine.

4.3 Parametric plots

It is easy to graph a curve $(f(t), g(t)$ in 2-space. For example (see Figure 4.3):

```
>> t = (0:2*pi/100:2*pi)';
>> plot(cos(t),sin(t))
>> axis('square')
```

(note the use of the `axis('square')` command to make the circle round instead of elliptic). See also the commands `comet, comet3`.

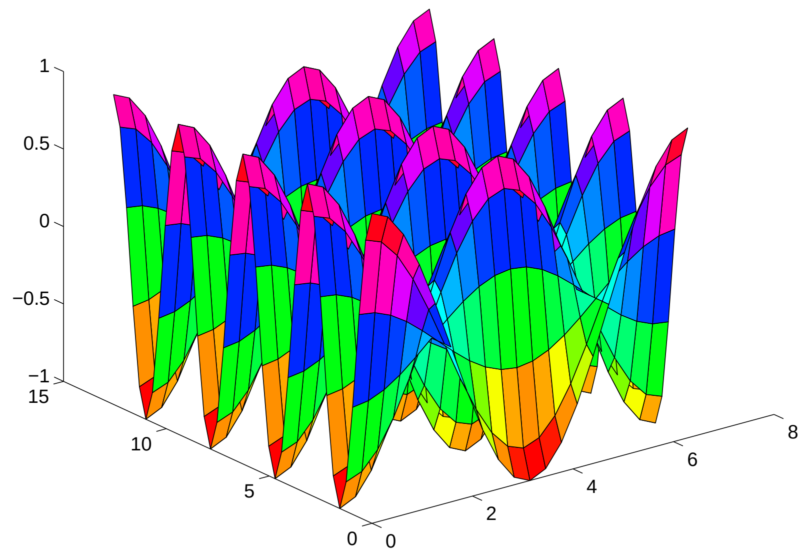

Figure 9: Using the `surf` command

5 Solving nonlinear problems in Matlab

Matlab provides a few functions for the solution of nonlinear problems, such as numerical integration, initial value problems in ordinary differential equations, root-finding, and optimization. In addition, optional "Toolboxes" provide a variety of such functions aimed at a particular type of computation, for example, optimization, spline approximation, signal processing, and so forth. I will not discuss the optional toolboxes.

More details about the following commands may be obtained from the help command:

- `quad, quad8` numerical integration (quadrature);
- `ode23, ode45` Runge-Kutta-Fehlberg methods for initial value problems;
- `fzero` root-finding (single variable);
- `fmin` nonlinear minimization (single variable);
- `fmins` nonlinear minimization (several variables);
- `spline` creates a cubic spline interpolant of give data.

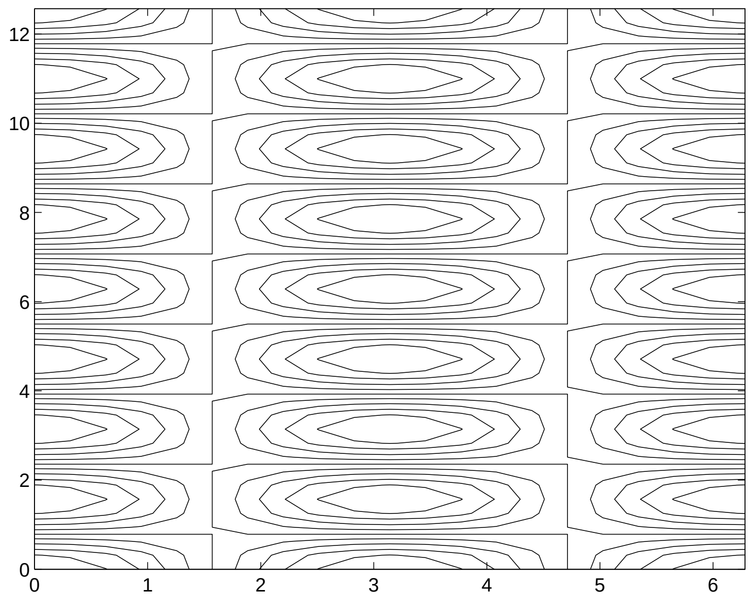

Figure 10: Using the `contour` command

6 Efficiency in Matlab

User-defined Matlab functions are interpreted, not compiled. This means roughly that when an m-file is executed, each statement is read and then executed, rather than the entire program being parsed and compiled into machine language. For this reason, Matlab programs can be much slower than programs written in a language such as Fortran or C.

In order to get the most out of Matlab, it is necessary to use built-in functions and operators whenever possible. For example, the following two command sequences have the same effect:

```
>> t = (0:.001:1)';
>> y=sin(t);
```

and

```
>> t = (0:.001:1)';
>> for i=1:length(t)
     y(i) = sin(t(i));
   end
```

However, on my computer, the explict for-loop takes 46 times as long as the vectorized sine function.

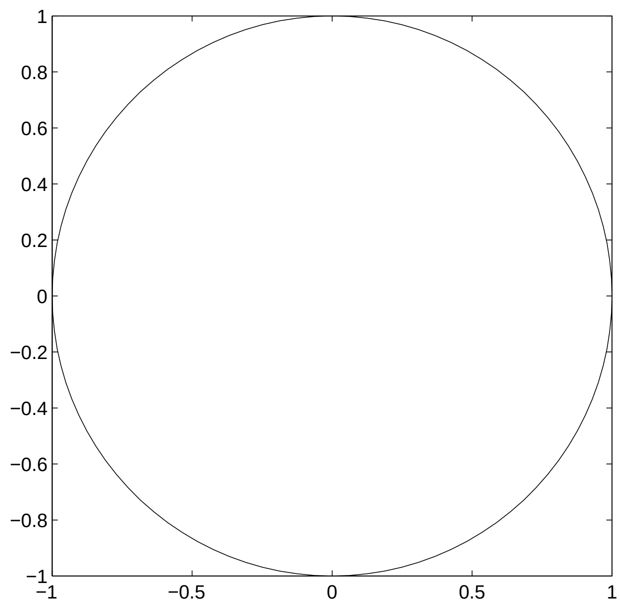

Figure 11: A parametric plot in 2D