MATH5004 - Lab 1

Simple Calculation & Graphs in Matlab

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

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.

```
>> 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).

One way to enter a *n*-dimensional array (*n*>2) is to concatenate two or more (*n*-1)-dimensional arrays using the cat command. For example, the following command concatenates two 3x2 arrays to create a 3x2x2 array:

```
>> C = cat(3,[1,2;3,4;5,6],[7,8;9,10;11,12])
C(:,:,1) =
    1    2
    3    4
    5    6
C(:,:,2) =
    7    8
    9   10
   11   12
```

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

```
>> whos
  Name      Size         Bytes  Class

  A         3x3            72  double array
  C         3x2x2          96  double array
  a         1x1             8  double array
  x         3x1            24  double array

Grand total is 25 elements using 200 bytes
```

Note that the argument ``3" in the cat command indicates that the concatenation is to occur along the third dimension. If *D* and *E* were kxmxn arrays, the command

>> cat(4,D,E)

would create a kxmxnx2 array (try it!).

Matlab allows arrays to have complex entries. 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, 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). The command more can be used to cause Matlab to display only one page of output at a time.

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.

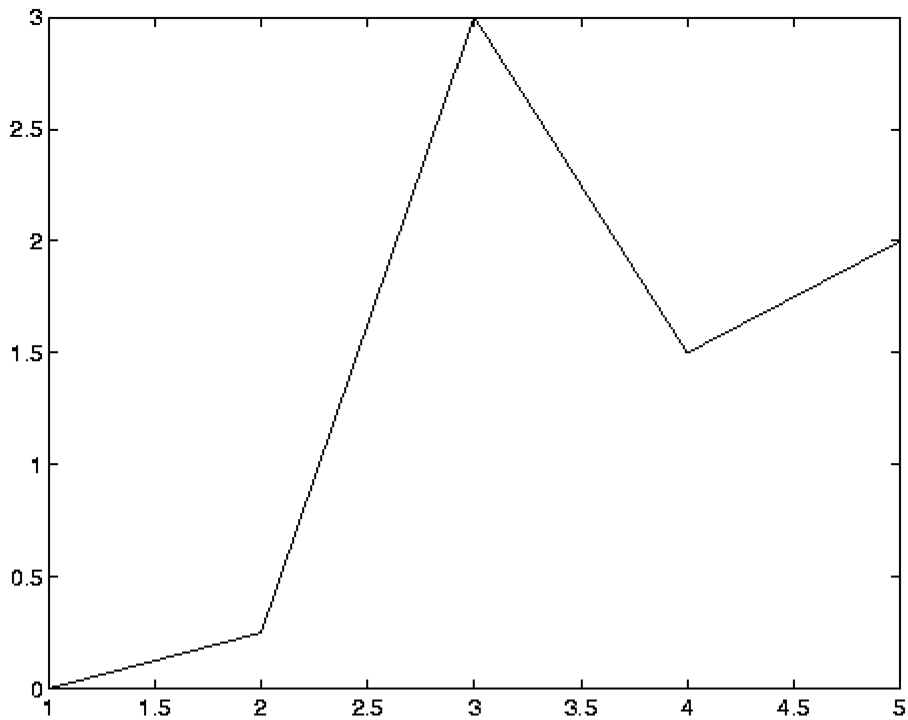

Figure 1: A simple Matlab graph

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

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

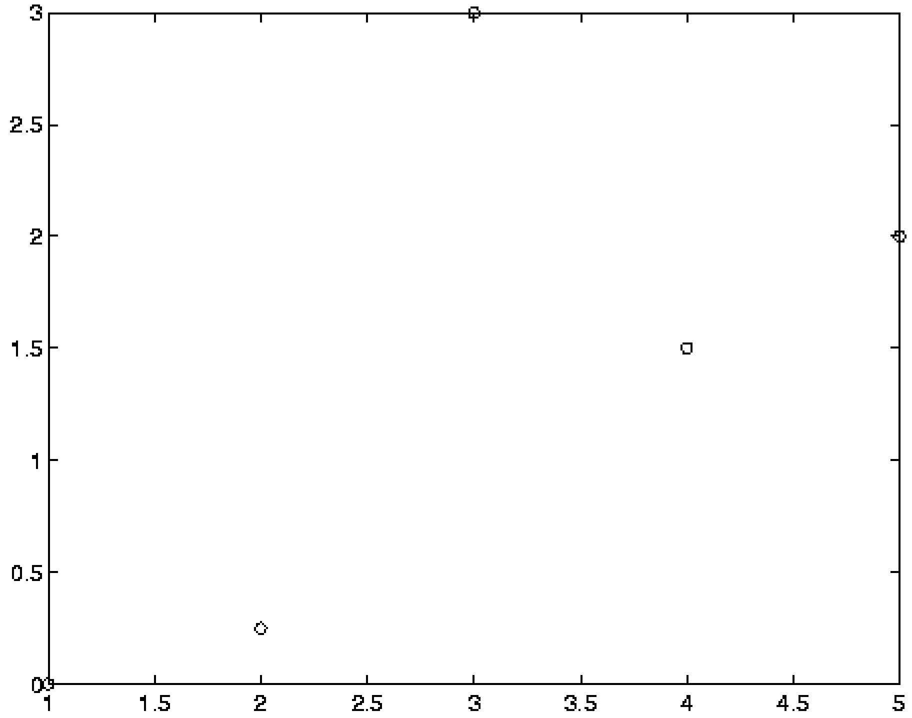

Figure 2: Another simple Matlab graph

Arithmetic operations on matrices

Matlab can perform the standard arithmetic operations on matrices, vectors, and scalars (that is, on 2-, 1-, and 0-dimensional arrays): addition, subtraction, and multiplication. In addition, Matlab defines a notion of matrix division as well as ``vectorized" operations. All vectorized operations (these include addition, subtraction, and scalar multiplication, as explained below) can be applied to *n*-dimensional arrays for any value of *n*, but multiplication and division are restricted to matrices and vectors ($n \leq 2$).

Standard operations

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

```
>> 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         Bytes  Class

  A         3x3             72  double array
  B         3x3             72  double array
  C         3x2             48  double array
```

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.
Matrix multiplication is also defined:

```
>> 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.

However, no notion of multiplication is defined for multi-dimensional arrays with more than 2 dimensions:

```
>> C = cat(3,[1 2;3 4],[5 6;7 8])
C(:,:,1) =
     1     2
     3     4
C(:,:,2) =
     5     6
     7     8
>> D = [1;2]
D =
     1
     2
>> whos
  Name      Size         Bytes  Class

  C         2x2x2           64  double array
  D         2x1             16  double array
```

Grand total is 10 elements using 80 bytes

```
>> C*D
??? Error using ==> *
```

No functional support for matrix inputs.

By the same token, the exponentiation operator ^ is only defined for square 2-dimensional arrays (matrices).

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. The division operator is not defined for *n*-dimensional arrays with *n*>2.

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. The command linspace has similar results; it creates a vector with linearly spaced entries. Specifically, linspace(a,b,n) creates a vector of length *n* with entries

$$a, a+(b-a)/(n-1), a+2(b-a)/(n-1), \ldots, b.$$

```
>> linspace(0,1,11)
ans =
  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
```

There is a similar command logspace for creating vectors with logarithmically spaced entries:

```
>> logspace(0,1,11)
ans =
  Columns 1 through 7
    1.0000    1.2589    1.5849    1.9953    2.5119    3.1623    3.9811
```

```
  Columns 8 through 11
    5.0119    6.3096    7.9433   10.0000
```

See help logspace for details.

A vector with linearly spaced entries 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 array, the effect is to create a new array of the same size whose entries are the function values of the entries of the original array. For example (see Figure 3):

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

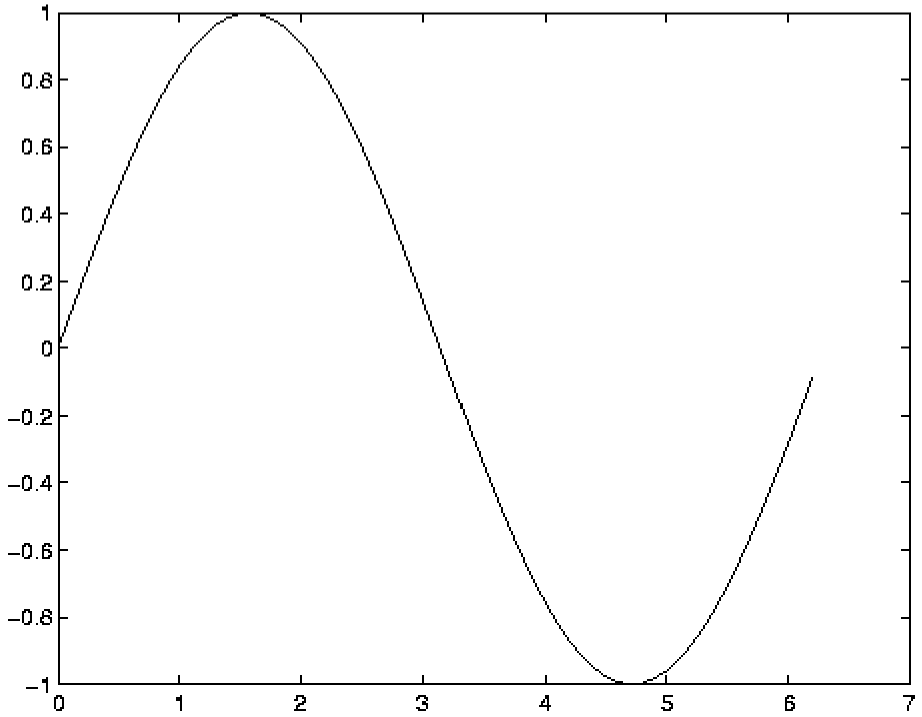

Figure 3: Graph of $y = sin(x)$

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 andA.^2. The first is only defined if A is a square matrix, while the second is defined for any *n*-dimensional array A.

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

The commands zeros and ones can be given any number of integer arguments; with *k* arguments, they each create a *k*-dimensional array of the indicated size.

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

$$a \le x \le b, c \le y \le d$$

- grid adds 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; see help max for the result when x is a *k*-dimensional array;
- min(x) analogous to max;
- abs(x) returns an array of the same size as x whose entries are the magnitudes of the entries of x;
- size(A) returns a $1 \times k$ vector with the number of rows, columns, etc. of the *k*-dimensional array A;
- length(x) returns the ``length'' of the array, 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

MATH5004 - Lab 2a

Programming in MATLAB

The capabilities of Matlab can be extended through programs written in its own 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 extension ``.m''). I will first discuss the programming mechanisms and then explain how to write programs.

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 expr1
  statements
elseif expr2
  statements
 .
 .
 .
else
  statements
end
```

The first block of statements following a nonzero *expr* executes.

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 =
```

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 cdidentifies 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 4.

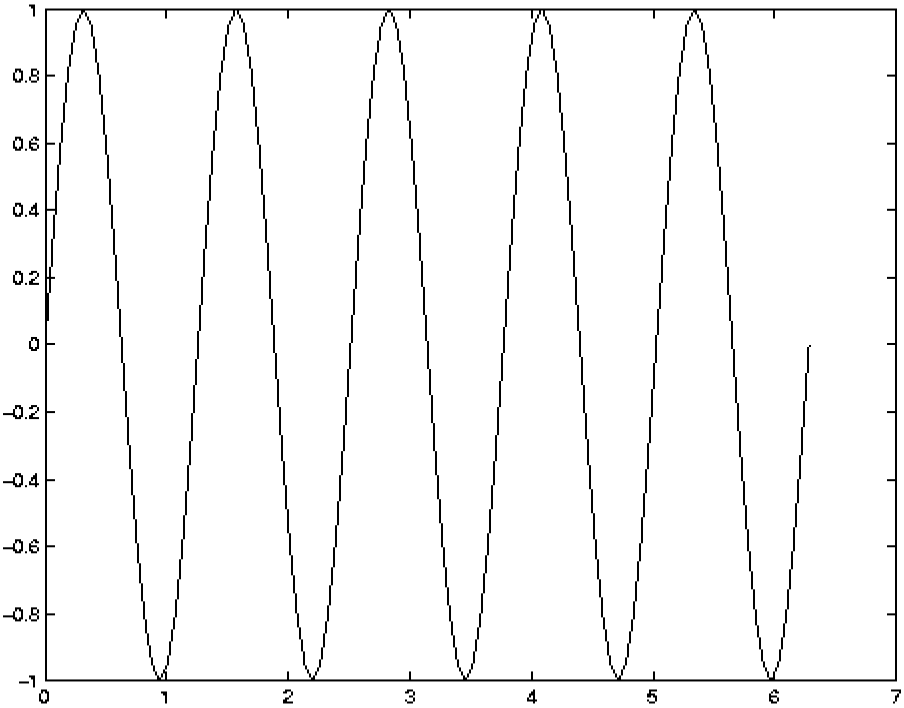

Figure 4: Effect of an m-file

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 executed 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(input1,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 5. 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 plotfor more information. Note also the use of the grid command to superimpose a cartesian grid on the graph.

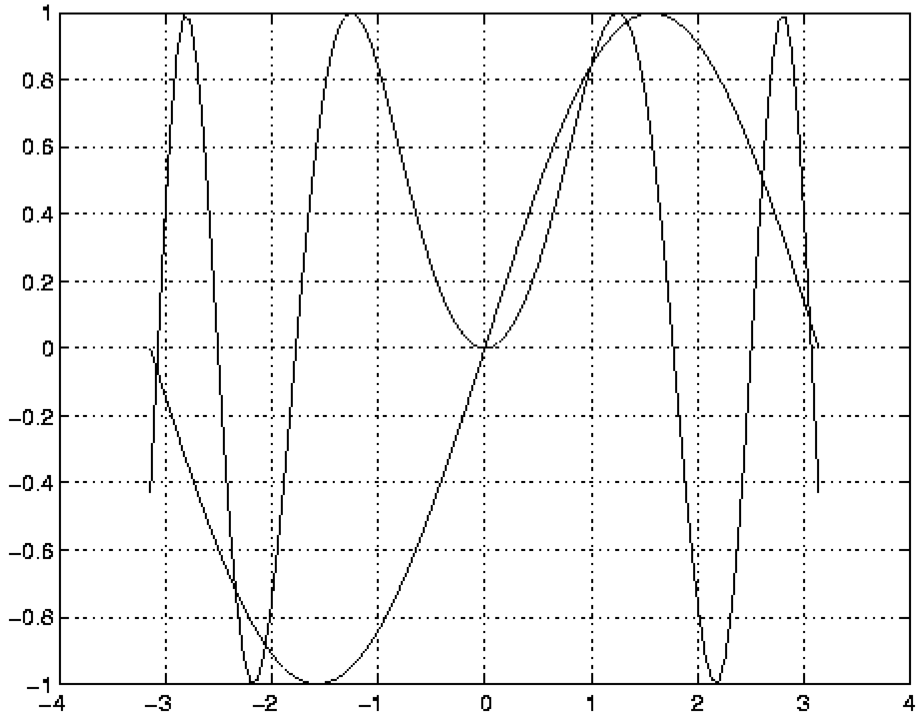

Figure 5: Two curves graphed together

A nontrivial example

Notice from Figure 5 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

```
>> 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
```

See help strings for information on how Matlab handles 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.

MATH5004 - Lab 2b

Advanced matrix computation

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 decomposition 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 ($V^T V = I$).

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;

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).

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         Bytes  Class

  A         3x3             72  double array
  S         3x3             64  sparse array

Grand total is 13 elements using 136 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 mxn sparse matrix with entry (*I*(*k*),*J*(*k*)) 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 versions 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 9x9 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 9x5 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 6:

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

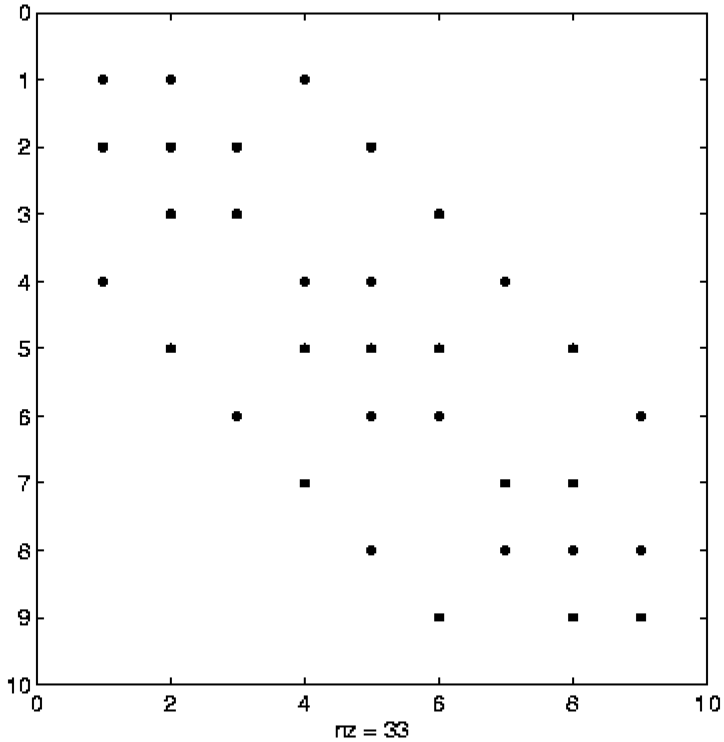

Figure 6: The sparsity pattern of a matrix