Using MATLAB after learning SCILAB

By Gilberto E. Urroz, September 2002

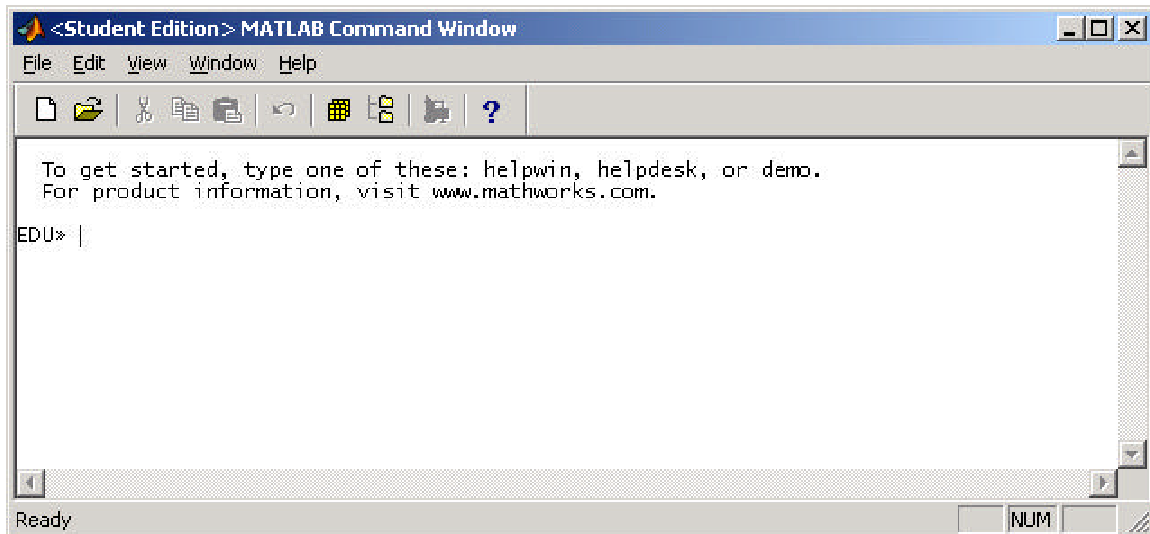
SCILAB and Matlab are very similar in operation: they both are numerical environments based on vector/matrix manipulations. They both use script files, function files, and diary files. Some differences, however, are present, and one need to be aware of them if interested in using Matlab after having learned the basic operation of SCILAB. Some of those differences are listed in the document *Matlab and Scilab comparisons* available for download in my *Scilab notes and functions* web page (a single URL):

http://www.engineering.usu.edu/cee/faculty/gurro/Software_Calculators/Scilab_Docs/SCILAB_Notes&Functions.htm

The notes in this document show examples of application of Matlab. Matlab is available in the *PC Lab* under *Start* > *Programs* > *Math Tools* > *Matlab*. The following notes refer to some Matlab functions as well as examples of Matlab operations.

Matlab interface

The following examples were created using an older Student version of Matlab, however, the interface remains similar to the current version. When you launch the Student version of Matlab that I have available, you will get the following interface:



This interface is very similar to that of SCILAB, with similar options available in the different menus shown. You can type Matlab commands in front of the prompt (in this case, the prompt is EDU>>), just like you do in SCILAB. If you need help with Matlab functions or its operation, use the online help facility by choosing the *Help* menu in the menu bar. Information about this facility is provided next.

Online Help

The MATLAB online help facility can be accessed by:

•Clicking on the question mark [?] icon on the toolbar. This opens the *Help* window.
•Type the line: *help FunctionName* in the command window
•Select *Help Desk (HTLM)* from the *Help* pull-down menu

Scripts

MATLAB scripts are collections of MATLAB commands that can be stored in a single text file and run from the command window.

•To create a script select the option *New > M-file* from the *File* menu in the toolbar. (M-file, in MATLAB, refers to text files that contain scripts or user-defined functions). This opens the *MATLAB Editor/Debugger* window where you will type your scripts. [SCILAB does not contain its own text editor, instead, you can use PFE as a text editor.]

As an exercise type the following script in the Editor window:

```
%This script includes commands for calculating the length of a vector
v = input('Enter a row vector:');
length_of_v = norm(v);
disp(['The length of vector v =']);
disp([' ']);
disp(['is:' num2str(length_of_v)]);
```

Notice that the first line in the script is a comment line. Comment lines in MATLAB start with the symbol %, rather than with // as in SCILAB.

•To save the script, use the option *Save As...* from the *File* menu in the toolbar. This will prompt you for the name of the M-file that will store the script. The default directory where MATLAB provides is a directory called Work within the MATLAB installation directory. You can choose to save it in this directory or change to another directory or drive. If you are using MATLAB in the PC Lab you want to save to your diskette or zip drive. Save the current script as *Script1.m*

•To run the script, select the option *Run Script...* from the *File* menu in the toolbar. This will produce an input box where you can type the address of the M-file containing the script. You can also click on the *Browse* button in the input form and navigate your way to the corresponding M-file. If using the latter option, you need to highlight the desired file, when found, and press the *Open* button in the corresponding form. This will load the name of the M-file in the input form. Once the name is loaded, press OK to run the script. MATLAB will execute the script providing the required output. If there are syntax errors in the script, the errors will be reported and execution of the script will be stopped.

As an exercise, run the script contained in the file Script.m that we saved earlier. This script requires the user to enter a row vector. The example shown here uses the vector [2, 3, -1]:

```
EDU» Enter a row vector:[2, 3, -1]
The length of vector v =
     2     3    -1

is:3.7417
```

•An alternative to running a script is to simply type the name of the script (without the suffix) in the command window. For this case, try:

```
EDU>> Script1
```

Note: In SCILAB, you'd have to use the exec command to run a script. In MATLAB, you just need to type the name of the script. If the script file *Script1.m* is located within the MATLAB Path specifications, MATLAB will search for the script name and execute it in the screen. To see the contents of the Path variable in MATLAB use:

```
EDU» Path
```

MATLAB will list all the directories included in the *Path* specification. Thus, when you invoke the name of a script file by typing it at the MATLAB prompt, MATLAB will look for that file in all the directories listed under *Path*. This is different than SCILAB, in that SCILAB does not search for script in a pre-defined set of directories. Instead, the user must specify the correct file specification.

M-files

There are two kinds of M-files:

•***Scripts***, which do not accept input arguments or return output arguments. They operate on data in the workspace.
•***Functions,*** which can accept input arguments and return output arguments. Internal variables are local to the function. (More information on user-defined functions is provided below).

Active variables in the current session

As you work within the command window you create variables that are kept active by MATLAB. To see all the active variables at any given time, use the commands *who* or *whos*, e.g.,

```
EDU» who

Your variables are:

A              b              v
ans            length_of_v
```

(***Note:*** the variable *ans* is a generic MATLAB variable that holds the last answer provided by MATLAB).

•The command *who* lists the name of the variables only.

```
EDU» whos
  Name              Size         Bytes  Class
  A                 3x3             72  double array
  ans               3x1             24  double array
  b                 3x1             24  double array
  length_of_v       1x1              8  double array
  v                 1x3             24  double array
Grand total is 19 elements using 152 bytes
```

•The command whos shows not only the variable names but also the type of matrix they are (e.g., length_of_v, a 1x1 matrix, represents a scalar value).

NOTE: Variable *ans* has the same role as in SCILAB. SCILAB also has commands who and whos.

Saving the workspace

The command window stores all the commands that you have typed since you started your session as well as the corresponding output. The command window, however, is not a file that MATLAB keeps track of (as is the case of a MAPLE Worksheet). In other words, you cannot save your command window by selecting a *Save* option from the file menu, as you can do with MAPLE Worksheets. You can , however, save the variables that you have defined into a MATLAB file referred to as the *workspace.*

If you want to save your current active variables (i.e., your workspace) use the option *Save Workspace As...* from the *File* menu in the toolbar. The workspace will be saved in a file that uses the suffix *.mat*

Alternatively, you can type the command *save* followed by the name of the workspace in the command window to save your current workspace. For example,

```
EDU>> save WorkspaceSept1
```

Loading a saved workspace

To load a saved workspace use the option *Load Workspace...* from the *File* menu, or type the command *load* in the command window followed by the name of the workspace you want to load.

Remember, saving and loading a workspace only means that you are saving the name and contents of your current active variables. All the commands used to load or generate those variables are not saved when you use the *Save Workspace...* option in the *File* menu.

Clear and clear all

The command *clear* can be used to clear the definition of a given variable. For example, the following commands load and clear the variable v1. The command who is used to check whether the variable v1 is present or not:

```
EDU» who                                 No output for who, indicating no
variables exist
EDU» v1 = [3; 1; -1]        %Variable v1 is loaded

v1 =
     3
     1
    -1

EDU» who                    %A second call to who shows that v1 exist
```

Your variables are:

```
v1
```

```
EDU» clear v1              %Clear variable v1
EDU» who                   %This call to who shows no output, v1 was cleared.
```

The command *clear all* removes all variable definitions. Use it only when you want to restart your workspace with a clean slate. If you have not saved your workspace, all definitions will be lost when you use *clear all*. [NOTE: SCILAB has also a *clear* command that works similar to MATLAB's *clear* command.]

Saving commands and output from the command window to a text file

If you want to save the commands that you type interactively in the command window, as well as the corresponding MATLAB output, you can use the commands *diary* and *diary off*. The command *diary*, followed by a file name, creates a new file (or opens an existing file) with the given file name. All the commands you type in the command window after using the command diary are written to the specified file. To stop sending text to the selected file, type the command *diary off.*

If no file name is indicated when using *diary*, the output will be sent to a file called *diary.txt*

Here is an example of the use of *diary* and *diary off*:

```
EDU» diary myFirstDiary
EDU» A = [2, 2; 1, -1]

A =
     2     2
     1    -1

EDU» b = [5; 2]

b =
     5
     2

EDU» x = A\b

x =
    2.2500
    0.2500

EDU» diary off
```

There will be a file called *myFirstDiary.txt* storing the commands and output used in the example above. Open this file using the option *Open* from the *File* menu in the command window. You may have to change the type of file that you want to open to *All Files(*.*)* in order to have this particular file listed in the *Open* window. The file, open in the *Editor/Debugger* window, will look like this:

```
A = [2, 2; 1, -1]

A =
     2     2
     1    -1


b = [5; 2]
```

```
b =
     5
     2

x = A\b

x =
    2.2500
    0.2500

diary off
```

To create a script file out of this diary file, you can edit out the output stored in the diary file. For this case, for example, you want the diary file edited to:

```
A = [2, 2; 1, -1]
b = [5; 2]
x = A\b
```

before saving it to a script file, say *Example1.m*

[NOTE: In SCILAB, the commands used to start and end a diary file are `diary('filename')` and `diary(0)`, respectively].

User-defined functions

User-defined functions are stored as M-files. The name of the M-file and of the function should be the same. Functions operate on variables within their own workspace, separate from the main MATLAB workspace.

The general form of a user-defined function is:

```
function OutputVariable = FunctionName(InputVariables)
% comments
expression(s)
```

The first line of a function M-file starts with the keyword function. It gives the function name and order of arguments. The next several lines, up to the first blank or executable line, are comment lines that provide the help text. These lines are printed when you type

```
help FunctionName
```

The first line of the help text is the *H1* line, which MATLAB displays when you use the *lookfor* command or request help on a directory. The rest of the file is the executable MATLAB code defining the function.

The variables introduced in the body of the function, as well as the variables on the first line, are all local to the function. They represent local variables that only have meaning within the function. As an example, type the function

```
function z = expabs(x,y)
% Function expabs calculates the exponential of the absolute value of
% a vector of two components (x,y)
mag    = sqrt(x^2+y^2);
z      = exp(mag);
```

into the M-file *expabs.m*

To check the function explanation, within the command window, use:

```
EDU» help expabs

  Function expabs calculates the exponential of the absolute value of
  a vector of two components (x,y)
```

You can call the function by using a simple call as:

```
EDU» expabs(2,2)

ans =

   16.9188
```

Or you can assign the function call to a variable, e.g.,

```
EDU» a = 2; b = 3; r = expabs(a,b); r

r =

   36.8020
```

User-defined function with two or more outputs

A function can have more than one output by defining the *OutputVariable*, in the general definition given above, as a vector. For example, the following function transform Cartesian to polar coordinates:

```
       function [r,theta] = Cartesian2Polar(x,y)
       % The function Cartesian2Polar takes as arguments the Cartesian
       % coordinate (x,y)and returns the corresponding Polar coordinates       %
(r,theta).

       r     = sqrt(x^2+y^2);
       theta = atan(y/x);
```

To find out about the function defined above use:

```
EDU» help Cartesian2Polar
```

The function Cartesian2Polar takes as arguments the Cartesian coordinate (x,y)
and returns the corresponding Polar coordinates (r,theta).

A direct call to the function shows only the second output:

```
EDU» Cartesian2Polar(2,-1)

ans =

    2.2361
```

Assigning the function to a vector with two components shows the two results required:

```
EDU» [r1, t1] = Cartesian2Polar(2,-1)
```

```
r1 =

    2.2361


t1 =

   -0.4636
```

IMPORTANT NOTE: MATLAB does not allow inline functions such as those defined with the command `deff()` in SCILAB. All MATLAB functions must be typed in the MATLAB editor and saved as an *M* file with the name of the function and that of the file being the same.

Global Variables

If you want more than one function to share a single copy of a variable, simply declare the variable as global in all the functions. Do the same thing at the command line if you want the base workspace to access the variable. The global declaration must occur before the variable is actually used in a function. Although it is not required, using capital letters for the names of global variables helps distinguish them from other variables. For example, create an M-file called *falling.m*:

```
function h = falling(t)
  global GRAVITY
  h = 1/2*GRAVITY*t.^2;
```

Then interactively enter the statements:

```
global GRAVITY
GRAVITY = 32;
y = falling((0:.1:5)');
```

The two global statements make the value assigned to GRAVITY at the command prompt available inside the function. You can then modify GRAVITY interactively and obtain new solutions without editing any files.

Command/Function Duality

MATLAB commands are statements like

```
load
help
```

Many commands accept modifiers that specify operands.

```
load August17.dat
help magic
type rank
```

An alternate method of supplying the command modifiers makes them string arguments of functions.

```
load('August17.dat')
help('magic')
```

```
type('rank')
```

This is MATLAB's "*command/function duality*." Any command of the form

```
command argument
```

can also be written in the functional form

```
command('argument')
```

Vectorization

To obtain the most speed out of MATLAB, it's important to vectorize the algorithms in your M-files. Where other programming languages might use for or DO loops, MATLAB can use vector or matrix operations. A simple example involves creating a table of logarithms.

```
x = 0;
for k = 1:1001
   y(k) = log10(x);
   x = x + .01;
end
```

A vectorized version of the same code is

```
x = 0:.01:10;
y = log10(x);
```

For more complicated code, vectorization options are not always so obvious. When speed is important, however, you should always look for ways to vectorize your algorithms.

Preallocation

If you can't vectorize a piece of code, you can make your for loops go faster by preallocating any vectors or
arrays in which output results are stored. For example, this code uses the function *zeros* to preallocate the
vector created in the for loop. This makes the for loop execute significantly faster.

```
r = zeros(32,1);
for n = 1:32
    r(n) = rank(magic(n));
end
```

Without the preallocation in the previous example, the MATLAB interpreter enlarges the r vector by one element each time through the loop. Vector preallocation eliminates this step and results in faster execution.

MATLAB's search path

MATLAB's search path, referred to as matlabpath, shows all the directories in your hard disk where MATLAB will search for M-files (containing scripts or user-defined functions) that you invoke in the command window. To review the current contents of matlabpath use the command:

EDU>> path

Other operations of interest that relate to MATLAB's search path are:

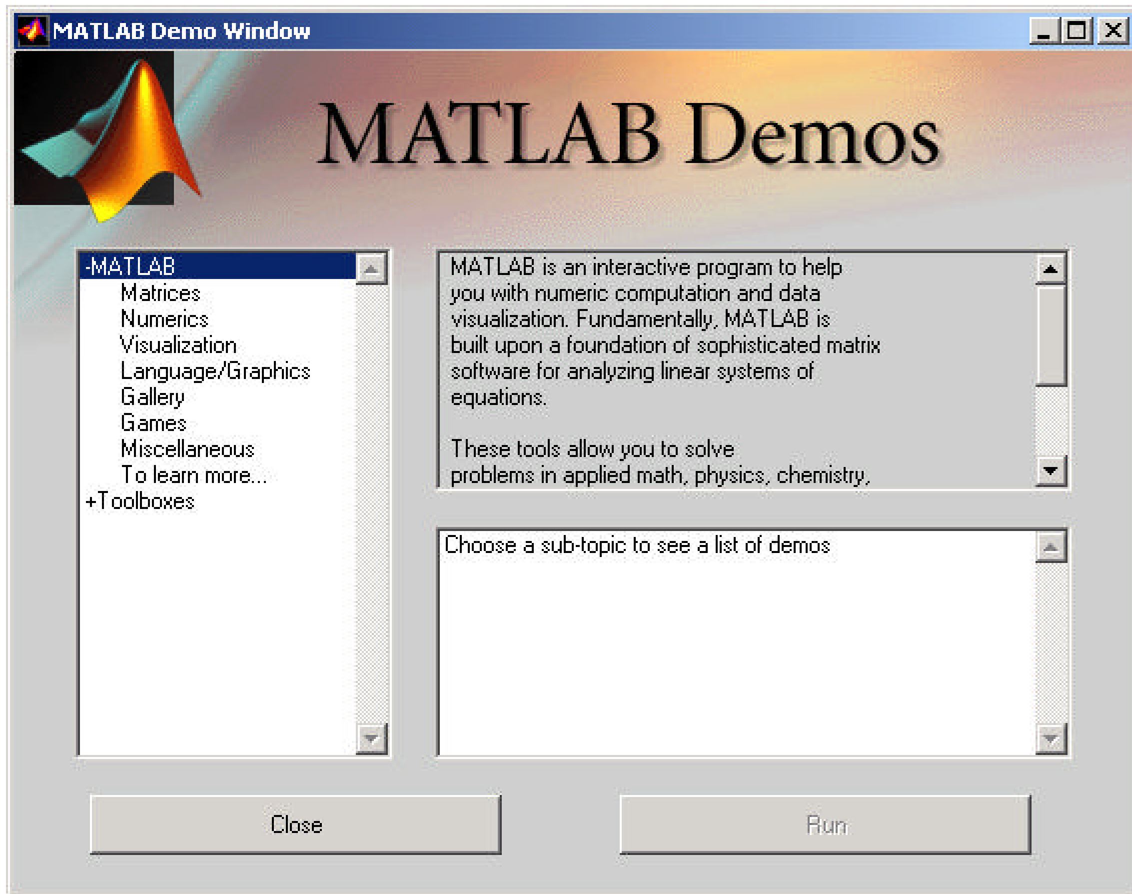
addpath dir1	Add directory dir1 to the beginning of matlabpath
cd	Show present working directory
p = cd	Return present working directory in variable p
cd myPath	Change directory to directory *myPath*
delete myFile	Deletes file *myFile.m*
dir	List files in current directory
d = dir	Return files in current directory in structured variable d
edit myFile	Open file *myFile.m* for editing [Same as *Open..* in *File* menu]
editpath	Edit matlabpath using a GUI [Same as *Set Path* in *File* menu]
exist('myFile','file')	Checks existence of file *myFile* on matlabpath
exist('myDir','dir')	Checks existence of directory *myDir* on matlabpath
inmem	Lists function M-files loaded in memory
ls	Same as dir
matlabroot	Return directory path to MATLAB executable program
pathdef.m	M-file where matlabpath is maintained
pwd	Same as cd
rmpath myDir	Remove directory *myDir* from the matlabpath
type myFile	Display m-file *myFile* in the Command window
what	Listing of all M-files and MAT-files in the current directory
which myFile	Display the directory path to *myFile.m*

Examples and demos

A number of examples and demonstrations of MATLAB applications are included under the Help pull-down menu. Select the option Examples and demos from the Help menu. This will produce the window *MATLAB Demos* listing a number of categories for examples and demos as shown below:

Select a particular category from the menu in the left window. For example, selecting the option *Language/Graphics* produces a sub-menu in the lower right window that starts with the option *MATLAB language introduction*. If you select this option, by pressing the button *[Run MATLAB language...*], you will start a slide show demonstrating several MATLAB scripts. The first slide in the show is shown below:

•Press the button [*Start>>*] to get the slide show started.
•Press [*Next>>*] or [*Prev<<*] to move to the next or previous slide, respectively.
•Press [*Reset*] to restart the slide show.
•Press [*AutoPlay*] to let the slide show play automatically.
•Press [*Info*] to get a description of the slide show. (You will need to press the [*Close*] button to return to the slide show.)
•Press [*Close*] to return to the *MATLAB Demos* window.



Back in the *MATLAB Demos* window you can select other options from the left window that will provide you with other examples and demonstrations. While most of these examples and demos produce slide shows similar to the one described above, the type of control buttons available may be different from those used in the *MATLAB language introduction* slide show. The title of the control buttons, however, will be self-explanatory, thus allowing the user to easily navigate through the slide show.

Complex numbers in MATLAB

Use i (or j, the preferred symbol in electrical engineering applications) as the imaginary unit. Thus, complex numbers are simply written as *x+i*y* or *x+j*y*, for example:

```
EDU» z1 = 3 - 5*i          %Define complex number z1
z1 = 3.0000 - 5.0000i


EDU» z2 = -2 + i                 %Define complex number z1
z2 = -2.0000 + 1.0000i

EDU» z1+z2, z1-z2, z1*z2, z1/z2   %sum, difference, multiplication, division
ans =1.0000 - 4.0000i
ans =5.0000 - 6.0000i
ans =-1.0000 +13.0000i
ans =-2.2000 + 1.4000i

EDU» z1^2, sqrt(z1), conj(z2)     %square, square root function, conjugate
```

```
ans = -16.0000 -30.0000i
ans = 2.1013 - 1.1897i
ans = -2.0000 - 1.0000i

EDU» real(z1), imag(z1)          %real and imaginary parts
ans = 3
ans = -5

EDU» abs(z2), angle(z2)          %magnitude and argument (for Polar represent.)
ans =2.2361
ans = 2.6779
```

Common mathematical functions in MATLAB

Name	Operation
abs(x)	Absolute value or magnitude of a complex number
acos(x)	Arc cosine
acosh(x)	Hyperbolic arc cosine
angle(x)	Phase angle of a complex number
asin(x)	Arc sine
asinh(x)	Hyperbolic arc sine
atan(x)	Arc tangent
atan2(x,y)	Four-quadrant arc tangent
atanh(x)	Hyperbolic arc tangent
ceil(x)	Upper integer value
conj(x)	Complex conjugate
cos(x)	Cosine
cosh(x)	Hyperbolic cosine
exp(x)	Exponential function, i.e., e^x
fix(x)	Round number
floor(x)	Lower integer value
gcd(x,y)	Greatest common divisor of integers x and y
imag(x)	Imaginary part of a complex number
lcm(x,y)	Least common multiple of integers x and y
log(x)	Natural logarithm function, i.e., ln x
log10(x)	Common logarithm or logarithm of base 10, i.e., log x or $\log_{10} x$
real(x)	Real part of a complex number
rem(x,y)	Remainder of x/y, i.e., r in x/y = k + r/y, where k is an integer
round(x)	Round number towards nearest integer
sign(x)	Singnum function. It returns sign of the argument as -1, 0, or 1
sin(x)	Sine
sinh(x)	Hyperbolic sine
sqrt(x)	Square root
tan(x)	Tangent
tanh(x)	Hyperbolic tangent

Examples of MATLAB graphics

Example 1 - Plot lists of numbers representing points in the plane

The following MATLAB Script produces a simple line graph of two-dimensional data. The data is entered as lists of numbers into variables *time* and *temps*. Notice the use of the continuation character ... to continue the lists in a second line. The function *mean* is used to obtain the average of the values in *temps*. The statements *plot*, *title*, *xlabel*, *ylabel*, and *grid* are used to produce the accompanying plot. These statements are separated by commas to indicate that they are related statements used in the production of the graph.

```
%  Compute average temperature and
%  plot the temperature data.
%

time = [0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, ...
      3.5, 4.0, 4.5, 5.0];
temps = [105, 126, 119, 129, 132, 128, 131,...
      135, 136, 132, 137];
average = mean(temps)
plot (time, temps), title('Temperature Measurements'),
xlabel ('Time, minutes')
ylabel ('Temperature, degrees F'), grid
```

Example 2 - Plot a function

To produce the plot of a function you first need to produce a range of values of the independent variable, then create a variable to store the values of the function. In the example below, because we are multiplying two functions, namely, exp(-0.05*x) and cos(2.5*x-2), both applied to vectors of numbers, the multiplication sign between the two functions is preceded by a period indication term-by-term multiplication. Omission of the period in this case results in an error.

```
% Plot a function y = f(x) = exp(-0.05*x)*cos(2.5*x-2)
%
x = 0:0.1:40;
y = exp(-0.05*x).*cos(2.5*x-2);
plot (x,y), title('Plot of function exp(-0.05x)cos(2.5x-2)'),
xlabel ('x')
ylabel ('y= f(x)'), grid
```

Alternatively, you could plot a function using the command *fplot*. For this case, try:

```
% Plot a function y = f(x) = exp(-0.05*x)*cos(2.5*x-2)using fplot
%
fplot('exp(-0.05*x)*cos(2.5*x-2)',[0 50]);
title('fPlot of function exp(-0.05x)cos(2.5x-2)'),
xlabel ('x')
ylabel ('y= f(x)'), grid
```

Example 3 - Defining line styles, markers, and colors

The function plot can contain specifications to change the line style, the type of point marker (if any), and the color of the plot. The specifications are listed, enclosed between question marks, after the names of the variables in the plot. The specifications available are:

Symbol	*Color*	*Symbol*	*Marker*	*Symbol*	*Line style*
b	blue	.	point	-	solid line
g	green	0	circle	:	dotted line
r	red	x	x-mark	-.	dash-dot line
c	cyan	+	plus sign	--	dashed line
m	magenta	*	star		
y	yellow	s	square		
k	black	d	diamond		
		?	triangle (down)		
		?	triangle up		
		<	triangle left		
		>	triangle right		
		p	pentagram		
		h	hexagram		

The following MATLAB script shows how to produce three different plots with different line styles, markers, and colors:

```
%      Plots with different line styles, markers, and colors

x = 0:0.5:40;
y = sin(0.5*x);
z = cos(0.5*x);
w = exp(-0.05*x);

plot(x,y,'b-d',x,z,'m-.^',x,w,'r--O'), xlabel ('x'),ylabel ('y= f(x)')
```

Example 4 – Plot in polar coordinates

Polar plots use the function polar. An example follows.

```
%A plot in polar coordinates

theta = 0:pi/50:2*pi;
r = 2.5.*(1+2*cos(theta));
polar(theta, r), title('Example of a polar plot')
```

Example 5 - Parametric plots

You can generate the points *(x(t), y(t))* and use plot to plot y vs. x. For example,

```
EDU» t = [0:0.1:20];
EDU» x = sin(2*t);
EDU» y = cos(t/2);
EDU» plot(x,y)
```

Example 6 – Plotting complex numbers

Complex numbers of the form *x+iy* can be represented by vectors in the x-y plane by using the function *compass.* For example:

```
EDU» x = randn(10,1); %generates 10 random values of x
EDU» y = randn(10,1); %generates 10 random values of y
EDU» compass(x,y)
```

Example 7 - Plotting with logarithmic scales

Use the commands *loglog(x,y)*, *semilogx(x,y)*, and *semilogy(x,y)* to generate a log-log plot, a semi-logartithmic plot in the x-axis, or a semi-logarithmic plot in the y-axis, respectively. Examples:

```
EDU» x = 10:1:10000; y = log(x/2)+1;
EDU» loglog(x,y)
EDU» semilogx(x,y)
EDU» semilogy(x,y)
```

The last plot is shown below:

Example 8 – Plotting a curve in space using parametric equations

A curve in space defined using parametric equations connects the points *(x(t), y(t), z(t))*, where *t* is the parameter that generates the curve. To plot a curve in space, generate a list of values of *t*, then use the function *plot3*. An example follows:

```
EDU» t=[0:0.1:100];
EDU» plot3(sin(t),cos(t),t.^4)
```

Example 9 – Plot of a function z = f(x,y)

To plot a function z = f(x,y) you can use either the *mesh* command or the *surf* command. While both commands will produce similar plots, the *mesh* command generates only a mesh of the data points, while the *surf* command will fill in the mesh with surface patches. To generate evenly spaced points in the (x,y) plane, use the command *meshgrid*. Try the following example:

```
EDU» x = -10:0.25:10;              %Generates values of x
EDU» y = -5:0.1:5;                 %Generates values of y
EDU» [X,Y] = meshgrid(x,y);        %Generate a grid of values in (x,y)
EDU» Z = sin(X).*cos(Y);           %Calculate z = f(X,Y)
EDU» mesh(X,Y,Z)                   %Get a mesh plot
```

With the same data (X,Y,Z), use the surf command to get:

```
EDU» surf(X,Y,Z)                   %Get a surface plot
```

Example 10 – Variants of the mesh *and* surf *command*

The commands *meshc*, *meshz*, and *waterfall* are variants of the *mesh* command that add different features to the graph. The command *meshc* adds a contour plot projection to the mesh graph. The command *meshz* produces a 'solid-cut' view of the graph. The command *waterfall* produces a figure described with parallel contours, resembling the streamlines in a

smooth waterfall. Also, a variant of the *surf* command, *surfc*, produces a surface plot with contour projections. Try the following exercises:

```
EDU» x = [-2:0.1:2];
EDU» [X,Y] = meshgrid(x);  % same as meshgrid(x,x)
EDU» Z = sin(X).*cos(Y);
EDU» mesh(X,Y,Z)           % figure 1 shows regular mesh plot
EDU» meshc(X,Y,Z)          % figure 1 shows mesh with contours
EDU» meshz(X,Y,Z)          % figure 1 shows mesh with solid cuts
EDU>> waterfall(X,Y,Z)     % figure 1 shows a 'waterfall-style' mesh
EDU» surfc(X,Y,Z)          % figure 1 shows surface plot with contours
```

The last result is shown below.

Example 11 - Contour and density plots

Contour plots and density plots can be used to represent a function z = f(x,y) in the plane. Check out the following examples:

```
EDU» x = -10:0.25:10;
EDU» y = -5:0.1:5;
EDU» [X,Y] = meshgrid(x,y);
EDU» Z = sin(X).*cos(Y);
EDU» contour(X,Y,Z)        %generate a contour plot

EDU>> pcolor(X,Y,Z)               %generate a color (density) plot
```

Example 12 - Combining contour and density plots

Using the same data of example 11 we can produce a combined contour-density plot as follows:

```
%  This script generates a combined contour-density plot

x = -10:0.5:10;
y = -5:0.25:5;
[X,Y] = meshgrid(x,y);
Z = sin(X).*cos(Y);
pcolor(X,Y,Z)            % generate color density plot (includes grid)
shading interp              % remove grid from the density plot
hold on                     % retains current figure for more graphic output
contour(X,Y,Z,10,'k')    % plot 10 contour lines in black ('k')
hold off                    % releases current figure
title('Example of contour-density plot combination')
```

Try all these examples on your own using MATLAB. To learn more about MATLAB graphics, read the document **PLOTLIB examples** located at the web site (a single URL):

http://www.engineering.usu.edu/cee/faculty/gurro/Software_Calculators/Scilab_Docs/SCILAB_Notes&Functions.htm

Although developed for SCILAB, the commands in the PLOTLIB library are very similar to MATLAB's graphing functions. Try the examples in the PLOTLIB document using MATLAB, and you'll find that all the functions described in the PLOTLIB document work in MATLAB without modification.

Programming structures in MATLAB

The programming structures in MATLAB are identical to those in SCILAB, with the execption of the *if* or *elseif* commands, that do not have a *then* particle as in SCILAB. Thus, the following function in SCILAB:

```
function [y] = f(x)
if x > 2 then
   y = x+2
else
   y = x
end
```

will be written as

```
function [y] = f1(x)
if x > 2
   y = x+2;
else
   y = x;
end
```

in MATLAB.

Also, notice that the statements `y = x+2` and `y = x` in the MATLAB function need to be followed by a semi-colon. Otherwise, the result will be shown in the main MATLAB window. In SCILAB that is not a problem since the statements within a SCILAB window are not output to the main SCILAB screen, unless an output statement (e.g., `disp`, `printf`) is used.

Comparison particles in MATLAB

The comparison particles in MATLAB are the same as in SCILAB (e.g., `>`, `>=`, `==`, etc.), except for the "not equal" particle that in MATLAB is written as `~=`. The corresponding SCILAB particle is `<>`.

Line continuation

Line continuation in MATLAB is the same as SCILAB, i.e., use the three dots at the end of a line to continue a statement in the next line, e.g.,

```
EDU» y = sin(x) + cos(x) + ...
exp(x)
```

Matrix functions

The use of the colon operator (:) and the functions *rand*, *ones*, *zeros*, *eye*, *inv*, *det*, *norm*, *rank*, *trace*, *cond*, *lu* for matrix operations are very similar in both MATLAB and SCILAB. To find eigenvalues and eigenvectors use the function *eig*, e.g.,

```
EDU» A = round(10*rand(4,4))  %create matrix A(4x4)

A =
```

```
     9     1     1     3
     9     4     2     2
     4     8     2     0
     9     0     6     7
```

Calculate eigenvalues only:

```
EDU» eig(A)

ans =

  16.0318
  -0.8002
   3.3842 + 3.0467i
   3.3842 - 3.0467i
```

Calculate eigenvalues and eigenvectors:

```
EDU» [vect,lam] = eig(A)

vect =

   0.4093             0.1786             0.0630 - 0.2640i   0.0630 + 0.2640i
   0.4824            -0.3096            -0.2632 - 0.0323i  -0.2632 + 0.0323i
   0.3917             0.6292            -0.5867 + 0.3419i  -0.5867 - 0.3419i
   0.6681            -0.6901             0.4335 + 0.4549i   0.4335 - 0.4549i


lam =

  16.0318                  0                  0                  0
        0            -0.8002                  0                  0
        0                  0             3.3842 + 3.0467i        0
        0                  0                  0             3.3842 - 3.0467i
```

Calculate generalized eigenvector value for matrices A and B:

```
EDU» A = round(10*rand(3,3))

A =

     4     4     2
     9     8     7
     5     5     8

EDU» B = round(10*rand(3,3))

B =

     2     5     4
     7     2     9
     3     7     9

EDU» [vect,lam] = eig(A,B)

vect =

    0.8714   -0.8381   -0.6830
    0.2939   -0.1072    0.7304
   -0.3927   -0.5348    0.0034
```

```
lam =

    2.3612         0         0
         0    1.1147         0
         0         0    0.0853
```

As in SCILAB, you can use function `sparse` to create sparse matrices.

Polynomial roots

Use function *roots* in MATLAB to calculate the roots of a polynomial given a vector of values representing the polynomial coefficients in decreasing order of powers, e.g., the polynomial p = $3x^3+5x^2-2x+1$, has roots:

```
EDU» c = [3,5,-2,-1]

c =

     3     5    -2    -1

EDU» roots(c)

ans =

   -1.9232
    0.5639
   -0.3074
```

Solution to equations

The equivalent of function *fsolve* in SCILAB is MATLAB'S *fzero* equation. Example:

Define the function:

```
function [y] = f000(x)
y = exp(-x) + sin(2*x) - 25
```

and save it to file f000.m. Then do the following:

```
EDU» x = -10:0.1:0; y = f000(x); plot(x,y); grid   %produces plot – shows root
near -5
EDU» xsol = fzero('f000',-3)                             %call 'fzero'
Zero found in the interval: [-0.28471, -5.7153].

xsol =

   -5.5175
```

Integration

Integration can be accomplished in MATLAB by using function `quad`, e.g., using function *f000(x)* defined above, the integral in the limits x = -6 to x = -5 is given by:

```
EDU» quad('f000',-6,-5)
```

```
ans =

    5.8575
```

References

The following are some good references for MATLAB (which can also be used for SCILAB, if we keep in mind the differences between the two programs):

1. Etter, D.M. and D.C. Kuncicky, 1999, "Introduction to MATLAB®", E-Source, Prentice Hall, Upper Saddle River, New Jersey

2. Palm III, W.J., "*Introduction to MATLAB® for Engineers,*" B.E.S.T Series, McGraw-Hill, Boston.

3. Hahn, B.D., 1997, "*Essential MATLAB® for Scientists and Engineers,*" Arnold, London.

4. Harman, T.L., J. Dabney, and N. Richert, 2000, "*Advanced Engineering Mathematics with MATLAB® - Second Edition,*" Brooks/Cole – Thomson Learning, Australia

5. Mathews, J.H. and K.D. Fink, 1999, "*Numerical Methods Using MATLAB – Third Edition,*" Prentice Hall, Upper Saddle River, New Jersey

6. Middleton, G.W., , "*Data Analysis in the Earth Sciences using MATLAB®*," Prentice Hall, Upper Saddle River, New Jersey

7. Nakamura, S., 2002, "*Numerical Analysis and Graphic Visualization with MATLAB – Second Edition,*" Prentice Hall PTR, Upper Saddle River, New Jersey

8. Van Loan, C.F., 2000, "*Introduction to Scientific Computing – A Matrix-Vector Approach Using MATLAB®,*" Prentice Hall, Upper Saddle River, New Jersey