Introduction to MATLAB

Albert Honein

Prepared for E155A, Stanford University

1 OVERVIEW

What is MATLAB? MATLAB, short for Matrix Laboratory, is a programming language for scientific calculations and graphic visualization of functions and data sets. It has hundreds of built-in routines for wide variety of computations. In addition, MATLAB has extensive add-on software modules, called toolboxes, used for a wide variety of disciplines, including statistics, image and signal processing, control systems designs, financial analysis, and optimization.

MATLAB's usage that is of interest to us includes plotting functions, two and three-dimensional graphics, plotting vector fields, calculation of definite integrals, data analysis, solution of algebraic nonlinear equations (e.g. polynomials), calculations with matrices, solution of linear systems of equations, and numerical solution of ordinary differential equations.

Where to find MATLAB? For those who own a personal computer, "The Student Edition of MATLAB" is available at the bookstore for about $100, which is $40 more than what it costs to obtain the manual, which is included with the program itself. This is a bargain price for excellent software, and it is highly recommended that you get your own copy of it to take with you when you leave Stanford.

For those who don't own a personal computer, or if you choose not to make this investment, you will find that MATLAB is available on a wide variety of platforms around campus. The fastest version is on the Unix machines connected to the Leland system. Most of the MacIntosh's on campus have MATLAB installed, including those in the Terman Engineering Computer Cluster (room 104), Terman library, Green library, first floor of Meyer library, and the computer cluster on the second floor of Tressider Union.

How to start MATLAB? On Unix workstations, the X Windows system makes working on a Unix machine much easier. When you get your account:

- Login and type `x` at the shell prompt to run the X Windows system.
- Type `matlab` at the shell prompt in any xterm window.
- Once in MATLAB the prompt `>>` is displayed. This means that you are now in MATLAB command window and you can start using MATLAB.

On the Macs:

- Open up the MATLAB folder in the applications folder and double click on the icon named MATLAB.
- If under the applications folder, MATLAB is listed as an Optional software, you need to download the whole folder of MATLAB onto the hard drive from the server. Read the Optional software instructions available there. The download might take up to 30 minutes.

We assume now that you were able to start MATLAB and you are facing the MATLAB command window with the MATLAB prompt `>>` or `EDU >>`.

2 MATLAB AS A CALCULATOR

Conventions for numbers. MATLAB uses conventional decimal notation, with an optional decimal point and leading plus or minus sign, to denote real numbers. Scientific notation is allowed: a real number followed immediately (no spaces) by the letter `e` and an integer n (positive or negative) results in a new number equal to the initial number multiplied by 10^n. Imaginary numbers are obtained by using either `i` or `j` as a suffix. Examples of valid numbers in MATLAB are: `3`, `-99`, `-1.01`, `8.027e-19`, `4.022e24`, `i`, `4i`, `-6.28j`, `-7.01e5i`.

Basic operations and precedence. The basic mathematical operators are `+`, `-`, `*`, `/`, and `^`, which are used for addition, subtraction, multiplication, division, and exponentiation, respectively. For example, typing `1+1` at the command prompt and hitting the Enter key yields:

```
ans =
     2
```

Other examples (in almost all the examples below, the expressions following `>>` are what you type at the command prompt. The other expressions are the MATLAB output after you hit the Enter key):

```
>> i*i                    >> 3i + 5                 >> j^.5
ans =                     ans =                     ans =
    -1                        5.0000 + 3.0000i          0.7071 + 0.7071i
```

Arithmetic formulas are evaluated from left to right with exponentiation being done first, followed by multiplication and division, and finally by addition and subtraction. This order is altered by using parenthesis; expressions in parentheses are evaluated first (read the examples row by row):

```
>> 6/3*2                  >> 2+3*4                  >> (2+3)*4                >> 2*2^3
ans =                     ans =                     ans =                     ans =
     4                        14                        20                        16

>> (2*2)^3                >> -2^.5                  >> (-2)^.5                >> (2+3i)^3
ans =                     ans =                     ans =                     ans =
    64                      -1.4142                    0.0000 + 1.4142i        -46.0000 + 9.0000i
```

MATLAB command window is user friendly. The Backspace, Delete, Right, and Left arrow keys can be used when typing at the command window. Also, previous expressions executed at the command window can be retrieved using the Up and Down arrow keys.

Commands in MATLAB. MATLAB allows issuing commands at the command window. Like calculating formulas, commands are issued by typing them at the command prompt and hitting the Enter key. We introduce first the `format` command which controls the display of numbers as in the following examples:

```
>> 1/3                    >> format long            >> format long e
ans =                     >> 1/3                    >> 1/3
    0.3333                ans =                     ans =
                             0.333333333333333         3.333333333333333e-01

>> format short e         >> format short           >> format Short
>> 1/3                    >> 1/3                    ??? Error using ==> format
ans =                     ans =                     Unknown command option.
   3.3333e-01                 0.3333
```

The last example shows that MATLAB is case sensitive; it distinguishes between uppercase and lowercase letters. Also, MATLAB allows writing consecutive input lines on one line if commas are used in between. For example, the two input lines in the second example above can be written on one line:

```
>> format long, 1/3
ans =
   0.333333333333333
```

Other commands that can be introduced at this stage are the `exit` and `help` commands. `exit` is used to quit MATLAB. `help` *topic* gives help on the specified topic which is, in general, a command name. `help` by itself lists all the help topics. Examples:

```
>> help exit                                        >> help abort
 EXIT   Exit from MATLAB.                           abort.m not found.
    EXIT terminates MATLAB.
```

In general, the `help` command displays elaborate explanation: be patient in using it while you become more familiar with MATLAB structure. Another help command is `lookfor`: `lookfor` *topic* searches all files used by MATLAB for *topic*.

Mathematical functions. MATLAB has a large number of built-in standard mathematical functions. At the command prompt, type `help elfun` to obtain a list of elementary mathematical functions. Familiar functions to you include:

`sqrt`, `sign`
`exp`, `log`, `log10` (Exponentials)
`sin`, `cos`, `tan`, `cot`, `sec`, `csc` (Trigonometric)
`asin`, `acos`, `atan`, `acot` (Inverse Trigonometric)
`sinh`, `cosh`, `tanh` (Hyperbolic)
`asinh`, `acosh`, `atanh` (Inverse Hyperbolic)
`abs`, `angle`, `conj`, `imag`, `real` (Complex)
`fix`, `floor`, `ceil`, `round`, `mod` (Rounding and Remainder)

Type `help` *name of the function* to get more details. These functions are used by including their arguments between parenthesis:

```
>> exp(1)              >> mod(4,3)              >> sqrt(-2)
ans =                  ans =                    ans =
    2.7183                  1                            0 + 1.4142i
```

The last example shows that taking the square root of a negative number is not an error; the appropriate complex result is produced automatically.

3 VARIABLES

Variables in MATLAB are of one kind: rectangular matrices (or arrays) of numbers/strings or formulas of numbers/strings. The numbers can be real or complex. A variable associated with a scalar is represented by a matrix of one row and one column. A variable associated with a vector is represented by either a matrix of one row (called row vector) or a matrix of one column (called column vector).

We show below how variables are created and used in MATLAB by using scalar variables. Manipulating and using variables associated with vectors and matrices are discussed in subsequent sections.

Creating and displaying variables. The simple command statement

```
>> monthly_salary=0.375*4
monthly_salary =
    1.5000
```

creates the scalar variable (1×1 matrix) `monthly_salary` and stores the value 1.5 in it. In subsequent calculations, this variable can be used like any number. MATLAB does not require any type of declarations or dimension statements. When MATLAB encounters a new variable name, it automatically creates the variable and allocates the appropriate amount of storage. If the variable already exists, MATLAB changes its contents and, if necessary, allocates new storage.

In the previous section, we evaluated arithmetic expressions without assigning them to any variable and the output was of the form `ans =` *number*. MATLAB assigns the most recent calculated expression that is not assigned to a variable to the default variable `ans`.

The value of any variable is displayed by typing its name and hitting the Enter key. The output after executing commands can be suppressed by including a semicolon `;` at the end of the command. Compare these examples:

```
>> ms = 1.5, ys=12*ms          >> ms = 1.5; ys=12*ms          >> ms = 1.5; ys=12*ms;
ms =                           ys =                           >> ys =
    1.5000                         18                             18
```

```
ys =
    18
```

Current variables in use can be listed using the commands `who` and `whos`. If we type `who` after executing all the commands from the beginning of this section, we obtain

```
>> who
Your variables are:
monthly_salary      ys
ms
```

Try `whos` for yourself.

Clearing variables. Variables will retain their values unless we assign them different ones or clear them using the command `clear`. The command `clear` *list of variable_names* clears the corresponding variables while `clear` by itself clears all variables used from the beginning of the MATLAB session.

```
>> ms= 1.5; ms=4.5; ys=12*ms;
>> ms
ms =
    4.5000
>> clear ms ys
>> ms
??? Undefined function or variable 'ms'.
```

Rules for naming variables. Variable names must always begin with a letter of the alphabet and the remaining characters may other letters, numbers, or the underscore character. No other special characters can be used. `x1`, `temperature`, and `Velocity_2` are all valid variable names. MATLAB uses only the first 31 characters of a variable name. It is not recommended to use MATLAB commands like `format`, `exit`, and `help` as variable names. Remember that MATLAB is case sensitive; it distinguishes between uppercase and lowercase letters. `A` and `a` are not the same variable:

```
>> a=2
a =
     2
>> A
??? Undefined function or variable 'A'.
```

Scalar constants. MATLAB has a number of predefined mathematical constants. Some of them are:

`pi`	$\pi = 3.14159265...$
`i, j`	Imaginary units, $\sqrt{-1}$
`realmin`	Smallest floating-point number
`realmax`	Largest floating-point number

```
>> realmin, realmax
ans =
  2.2251e-308
ans =
  1.7977e+308
```

These constants are treated like any other user defined variables; their values can be changed by the user (although this is not recommended, except for `i` and `j`). They can be reassigned to their original MATLAB values by using `clear` or `clear` *a specific constant*.

4 VECTORS

Constructing row vectors. A row vector `x` is formed by putting its elements between brackets and separating them with blanks or commas:

```
>> x=[1  2  3  4  5 ]
x =
     1     2     3     4     5
```

When formulas are used as elements the use of parenthesis is recommended. Study the following examples:

```
>> a=2; x=[a+2 -3 4/2]
x =
     4    -3     2

>> x=[a +2, -3, 4/2]
x =
     2     2    -3     2
```

```
>> x=[a +2 -3 4/2]
x =
     2     2    -3     2

>> x=[(a +2), -3, 4/2]
x =
     4    -3     2
```

Constructing vectors with equally spaced elements. MATLAB has two commands for building vectors with equally spaced elements. The first is the colon notation `:` whose syntax is

$$\mathtt{x} = \textit{starting value } : \textit{increment } : \textit{maximum value}$$

Example:

```
>> x=1:.5:2.5
x =
    1.0000    1.5000    2.0000    2.5000
```

Note that if we specify 2.4 as the maximum value instead of 2.5, `x` will then be `1.0000 1.5000 2.0000`. If the increment is omitted, it is assumed to be 1:

```
>> x=1.1:7
x =
    1.1000    2.1000    3.1000    4.1000    5.1000    6.1000
```

If we need `x` to have a specific last element, we should use the `linspace` function which has the syntax

$$\mathtt{x} = \mathtt{linspace}(first\_element, last\_element, num\_of\_elements)$$

```
>> x=linspace(1,2.4,4)
x =
    1.0000    1.4667    1.9333    2.4000
```

Constructing column vectors. Column vectors are constructed by listing their elements, separated by semicolons, between brackets:

```
>> y=[1;4;5]
y =
     1
     4
     5
```

To construct longer column vectors with equally spaced data, we use the colon notation or the `linspace` function to produce a row vector and then apply the transpose operator `.'`:

```
>> x=1:2:5                          >> y=x.'
x =                                 y =
     1     3     5                       1
                                         3
                                         5
```

The complex conjugate operator is ' by itself:

```
>> z=[1+i,1+2i];z'                  >> z.'
ans =                               ans =
   1.0000 - 1.0000i                    1.0000 + 1.0000i
   1.0000 - 2.0000i                    1.0000 + 2.0000i
```

Of course, ' is equivalent to .' for real vectors.

Finally, column vectors can also be obtained using the single colon : as a subscript. `y=x(:)` puts all the elements of the `x` vector (`x` can be a row vector or a column vector) in `y` and makes it a column vector:

```
>> x=1:2:5; y=x(:)
y =
     1
     3
     5
```

Addressing vector elements. If `a` is a vector, its ith element is a scalar and is obtained by `a(i)`. `i` is referred to as a subscript and it cannot be negative or zero. It is surrounded by parenthesis and not brackets. Examples:

```
>> a=13:17                                       >> a(6)
a =                                              ???  Index exceeds matrix dimensions.
    13    14    15    16    17                   >> a(0)
>> a(3)                                          ??? Index into matrix is negative or zero.
ans =                                            See release notes on changes to logical indices.
    15
```

More than one element or blocks of elements can be accessed using vectors of natural numbers (positive integers) as subscript. Continuing with the example above, `v1=a([1,3])` will form a row vector v1 of dimension 2 with elements the first and third element of `a`, that is `13` and `15`. `a(2:4)` will give elements two through four: `14 15 16`. `a(2:2:5)` will give elements 2 and 4: `14 16`. `a([[3:-1:2],5])` will give elements 3, 2, and 5: `15 14 17`.

Array operations. When vectors are of the same dimension and type (row/column), the elementary operations used for scalars apply to vectors on an element-by-element basis. The notation for addition and subtraction is the same as for scalars: `+` and `-`. For multiplication, division, and power, the notation is `.*`, `./`, and `.^` respectively. For example:

`[2,3]+[4,1]`	will have the value	$[2+4, 3+1]$	which is	[6,4]
`[2,3]-[4,1]`	"	$[2-4, 3-1]$	"	[-2,2]
`[2,3].*[4,1]`	"	$[2*4, 3*1]$	"	[8,3]
`[2,3]./[4,1]`	"	[2/4,3/1]	"	[0.5,3]
`[2,3].^[4,1]`	"	[2^4,3^1]	"	[16,3]

The expressions `[2,3]*[4,1]` and `[2,3].*[4,1]'` will result in an error.

Operations between scalars and vectors are also defined. The different possibilities are shown in the examples below between the scalar 4 and the vector `[2,3]`:

`4+[2,3]`, `[2,3]+4`	will have the value	$[4+2,4+3]$
`4-[2,3]`	"	$[4-2,4-3]$
`[2,3]-4`	"	$[2-4,3-4]$
`4*[2,3]`, `[2,3]*4`, `4.*[2,3]`, `[2,3].*4`	"	$[4*2,4*3]$
`4./[2,3]`	"	$[4/2,4/3]$
`[2,3]/4`, `[2,3]./4`	"	$[2/4,3/4]$
`4.^[2,3]`	"	[4^2,4^3]
`[2,3].^4`	"	[2^4,3^4]

These operations are called "array operations", i.e. array addition, array subtraction, array multiplication, array division, and array power. They are exactly the scalar operations defined before but are applied at the same time to more than one scalar, or to an array of scalars. They are to be contrasted with "matrix operations" which are defined differently and are introduced later in the Linear Algebra section.

Passing vectors to built-in functions. The built-in functions mentioned before can be used with vectors the same way they are used with numbers. This time, the result will be a vector of the same nature of the input vector. Its elements are obtained by applying the function to the elements of the input vector. Example:

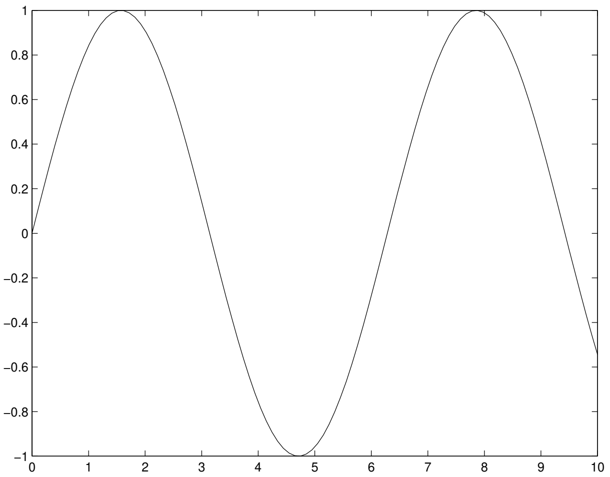
```
>> x=0:pi/2:3*pi; y=sin(x)
y =
         0    1.0000    0.0000   -1.0000    0.0000    1.0000    0.0000
```

MATLAB has many built-in functions that manipulate vectors. We will introduce them when we discuss matrices.

5 SIMPLE PLOTS

Plotting is very easy in MATLAB. The following MATLAB commands plot the function $y = \sin(x)$ for $0 \le x \le 10$:

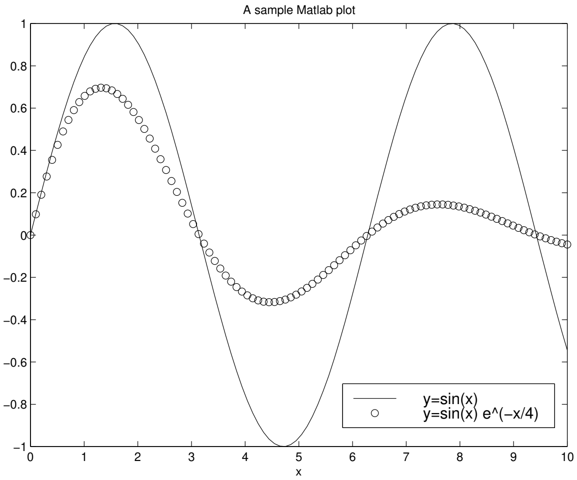
```
>> x = linspace(0,10,100);
>> y1 = sin(x);
>> plot(x,y1,'b')
```



The first two lines create the vectors `x` and `y1`. The `plot` command on the third line produces the non-labeled plot. The general form of the `plot` command is `plot(x,y,S)` where `x` and `y` are vectors of the same type and dimension and `S` is a collection of characters that tell MATLAB what color, linestyle, and data point symbol to use. Use `help plot` to find what are the characters corresponding to the different colors, symbols, and line styles in MATLAB. For example, `b` refers to blue, `o` to circle, and `:` to dotted. The command plot(x,y1,'bo:') (or plot(x,y1,'o:b')) plots a blue dotted line with a circle at each data point. The line properties as well many other plot properties can be specified using the Tools menu on the top of the window that contains the plot.

We next plot the function $y = \sin(x)\exp(-x/4)$ on the same figure:

```
>> y2 = sin(x).*exp(-.25*x);
>> hold on
>> plot(x,y2,'ro')
>> title('A sample MATLAB plot')
>> xlabel('x')
>> legend('y=sin(x)','y=sin(x) e\^(-x/4)')
```



The hold on command tells MATLAB to hold the current plot so that more elements can be added. The legend command inserts the legend in the lower right hand corner (actually MATLAB puts it in the upper right hand corner by default, but you can use the mouse to move it around). The xlabel and title commands are self explanatory. The arguments for legend, xlabel and title are all texts and will be discussed later in this handout. More plotting commands will be introduced later but you can always use the online help. Try, for example, help ylabel, help axis, help grid, help subplot, and help loglog

Printing plots. Hardcopies of graphs on Unix workstations are best achieved by using the print command. For example, print -deps fig at the command prompt creates a postscript file fig.eps which can be sent directly to the printer using the local print command at the Unix level (usually lp or lpr). On MACs and PCs it is easiest to use the menus on the window containing the plot to change the page setup and to send plots directly to the printer. The print command may still be used at the command line to produce postscript files.

6 SAVING YOUR COMMANDS – SCRIPTS

Everything we have done so far has been in MATLAB's interactive mode. However, MATLAB can execute commands stored in a regular text file. These files are called scripts or 'M-files'. Instead of writing the commands at the prompt, we write them in a script file and then simply type the name of the file at the prompt to execute the commands. It is almost **always** a good idea to work from scripts and modify them as you go instead of repeatedly typing everything at the command prompt.

Any text editor can be used to generate M-files. An M-file must be a plain ASCII file with the '.m' extension. The name of the file should not conflict with any existing MATLAB commands or variables. On PCs and MACs, M-files can be created using New under the File menu on the top of the command window. On Unix, Emacs, vi, or any other editor can be used. Unix commands can be issued from inside MATLAB by preceding them with an exclamation mark. For example, to edit an M-file called test.m with Emacs, type

```
>> !xemacs test.m &
```

This M-file can be ran by typing `test` at the prompt, of course after saving it with Emacs.

The M-file has to be in a directory that MATLAB knows about, that is in MATLAB `path` (try `help path`). On Unix system this can be the current directory where MATLAB was launched.

A very simple script file:

```
% very very simple
x = sin(pi/2.)
```

The % symbol tells MATLAB that the rest of the line is a comment. It is a good idea to use comments so you can remember what you did when you have to recycle an M-file (as will often happen). If you call this script `very.m` and save it in your current directory, it is executed at the prompt by typing `very` (no need for '.m'). The output is:

```
>> very
x =
     1
```

It is important to note that the script is executed in the same workspace memory as everything we do at the prompt. We are simply bringing the commands from the script file instead of typing them by hand at the prompt. The variables already existing before executing the script can be used by that script. Similarly, the variables in the script are available at the prompt after executing the script. In the example above, if you change the value of `pi` before calling `very`, `x` will not be 1:

```
>> clear
>> very
x =
     1
>> pi=0; very
x =
     0
```

Here is a program that plots a circle and is worth saving in a script file:

```
% specify the radius
radius = 3;

% number of points to plot
num_pts = 100;

theta = linspace(0,2*pi,num_pts);

% compute the coordinates
x = radius * cos(theta);
y = radius * sin(theta);

plot(x,y,'.')
title('A circle')

% enlarge the axes a bit
axis([-1.1,1.1,-1.1,1.1]*radius)

% make the axes square
axis square
```

This M-file and other samples for the class are available on the class website (`http://www.stanford.edu/class/me200a/`).

7 MATRICES

This section is a generalization of section 4 where we introduced MATLAB vectors. Since row and column vectors are special cases of two dimensional matrices, the discussion in section 4 follows from the discussion in this section.

Simple construction. A matrix is a rectangular array of numbers. For example,

$$A = \begin{bmatrix} 5 & 4 & 0 \\ 3 & 2 & \sin(\pi/2) \end{bmatrix}$$

is a matrix with two rows and three columns (2×3 matrix). We look at this matrix, and any other matrix, as a set of row vectors placed underneath each other or a set of column vectors placed side by side next to each other. A matrix can be constructed in MATLAB according to these two interpretations: we can place its rows between brackets and separate them by semicolons or place its columns between brackets and separate them by spaces or commas. We illustrate this by constructing the matrix above. Using the row picture:

```
>> A=[5 4 0; 3 2 sin(pi/2)]
A =
     5     4     0
     3     2     0
```

We can also define the rows separately and then put them inside the brackets:

```
>> r1=[5 4 0]; r2=[3 2 sin(pi/2)]; A=[r1 ; r2]
```

will produce the same `A` above. Note that the rows `r1` and `r2` must have the same dimension. Similarly we can construct `A` from its columns:

```
>> c1=[5 3]'; c2=[4 2]'; c3=[0  sin(pi/2)]'; A=[c1 c2 c3]
```

We can also form a matrix by appending rows, columns, or matrices to an existing matrix. The dimensions should be consistent however. Study the following examples (`A, r1`, ... same as above):

```
>> B=[A ;r2]
B =
     5     4     0
     3     2     0
     3     2     0
```

```
>> B=[A  c2]
B =
     5     4     0     4
     3     2     0     2
```

```
>> B=[A;A]
B =
     5     4     0
     3     2     0
     5     4     0
     3     2     0
```

```
>> B=[A A]
B =
     5     4     0     5     4     0
     3     2     0     3     2     0
```

Extracting matrix elements. The element in the ith row and jth column of a matrix `A` is `A(i,j)`. The subscripts `i` and `j` cannot be negative or zero. Arrays of matrix elements are identified using vector subscripts whose elements are positive numbers. This is illustrated for the following matrix:

```
>>  B=[ 1 0 2 5 ; 4 8 6 11; 7 3 9 20]
B =
     1     0     2     5
     4     8     6    11
     7     3     9    20
```

B(2,3) is the element in row 2 and column 3 which is 6:

```
>>  B(2,3)
ans =
     6
```

Consider B([2 3],[1 4]). The vector on the left of the comma ([2 3]) refers to rows while the one on the right ([1 4]) refer to columns. B([2 3],[1 4]) is a sub-matrix of B whose elements are the intersection of rows 2 and 3 and columns 1 and 4. Similarly, B([1 2],3) is sub-matrix of B whose elements are the intersection of rows 1 and 2 and column 3, which is the column vector of elements 2 and 6. These two examples are shown below:

```
>> B([2 3],[1 4])
ans =
     4    11
     7    20
```

```
>> B([1 2],3)
ans =
     2
     6
```

The numbers forming the vector subscript can be in any order and this allows us to manipulate B in any way. Compared to B([2 3],[1 4]) above, B([3 2],[4 1]) is

```
>> B([3 2],[4 1])
ans =
  20     7
  11     4
```

C=B(3:-1:1,1:4) creates a matrix C by taking the rows of B in reverse order:

```
>> C=B(3:-1:1,1:4)
C =
     7     3     9    20
     4     8     6    11
     1     0     2     5
```

The matrix B defined in this subsection will be used many times below. In order not to keep redefining it, we create an M-file called Bmat.m containing the command that creates B.

Single colon notation. When a vector subscript is to refer to all the rows or columns in ascending order (like 1:4 in the expression for C above), we can use a single colon instead. C=B(3:-1:1,:) will give the same result as C=B(3:-1:1,1:4). Also, B(2,:) refers to the second row and B(:,3) refers to the third column:

```
>> B(2,:)
ans =
     4     8     6    11
```

```
>> B(:,3)
ans =
     2
     6
     9
```

When used by itself as a matrix subscript, the single colon creates a column vector by taking the columns of the matrix one at a time. C=B(:) will create a column vector C consisting of the first, second, third, and fourth column of B:

```
>> C=B(:); C'
ans =
     1     4     7     0     8     3     2     6     9     5    11    20
```

Changing matrix elements. Once a matrix is formed, any of its submatrices (a single element, vector, or two dimensional matrix) is accessed using the appropriate subscripts. It can be changed by simply assigning it a matrix of the same dimensions. With B the same matrix as before, B(2,3)=4 changes the element in the second row and third column to 4 while B(1,:) = [1:4] makes the first row equal to [1 2 3 4]:

```
>> Bmat;  B(2,3)=4              >> Bmat;  B(1,:) = 1:4
B =                             B =
     1     0     2     5             1     2     3     4
     4     8     4    11             4     8     6    11
     7     3     9    20             7     3     9    20
```

`B(:,[3,1,4])= [1:3; 1:3;1:3]` results in:

```
>> B(:,[3,1,4])= [1:3; 1:3;1:3]
B =
     2     2     1     3
     2     8     1     3
     2     3     1     3
```

Deleting a row or column. Vectors in a matrix are deleted by assigning `[]` to them. `B(1,:)=[]` deletes the first row of `B` while `B(:,2:4) =[]` deletes the second through the fourth column:

```
>> Bmat; B(1,:)=[]              >> Bmat;B(:,2:4) =[]
B =                             B =
     4     8     6    11             1
     7     3     9    20             4
                                     7
```

Given the vector `v=[8 9 5 6 3]`, `v([2 3 4])=[]` deletes the second through the fourth elements and results in a vector of dimension 2:

```
>> v=[8 9 5 6 3]; v([2 3 4])=[]
v =
    8     3
```

Transpose and conjugate transpose. The operator `.'`, when placed after a matrix, produces its transpose, that is, a matrix whose columns are the rows of the original matrix, placed in the same order. Equivalently, the rows of the transpose of a matrix `B` are the columns of `B`:

```
>> Bmat; B                      >> B.'
B =                             ans =
     1     0     2     5             1     4     7
     4     8     6    11             0     8     3
     7     3     9    20             2     6     9
                                     5    11    20
```

The operator `'` by itself produces the conjugate transpose. This means that it transposes the matrix and replaces its elements by their complex conjugate. If the elements of the matrix are real, then `'` and `.'` are equivalent.

MATLAB matrix utilities. The mathematical functions mentioned in sections 2 and 4 (`sin`, `cos`, ...) apply to matrices the same way they apply to vectors. For example, `sin(A)` is a matrix of the same size of `A`. If a_{ij} is the element on row i and column j of `A`, the element on row i and column j of `sin(A)` will be $\sin(a_{ij})$.

In addition to these functions, MATLAB has other useful built-in functions to manipulate matrices. We list some of them:

- `eye(m,n)` is an $m \times n$ matrix with ones on the main diagonal and zeros elsewhere (the main diagonal consists of the elements with equal row and column numbers). If $m = n$, `eye(n)` can be used instead of `eye(n,n)`.

- `zeros(m,n)` is an $m \times n$ matrix with zero elements. If $m = n$, `zeros(n)` can be used instead of `zeros(n,n)`.

- `ones(m,n)` is an $m \times n$ matrix whose elements are ones. If $m = n$, `ones(n)` can be used instead of `ones(n,n)`.
- `diag(v)` is a square diagonal matrix with vector `v` on the main diagonal.
- `diag(A)` is a column vector formed from the main diagonal of `A`.
- `diag(A,n)` is a column vector formed from the nth diagonal of `A`. $n = 0$ corresponds to the main diagonal, $n = 1$ to the first upper diagonal, $n = -1$ to the first lower diagonal, etc (see examples below).
- `tril(A)` is the lower triangular part of `A`.
- `triu(A)` is the upper triangular part of `A`.
- `max`: for vectors, `max(v)` is the largest element in `v`. For two dimensional matrices, `max(A)` is a row vector containing the maximum element from each column.
- `size(A)` is a two-element row vector containing the number of row and columns of `A`. This function can be used with `eye`, `zeros`, and `ones` to create a matrix of the same size of `A`: `ones(size(A))` creates a matrix of ones having the same size of `A`.
- `length(A)`is a number equal to the greater of the number of rows and the number of columns, that is, `max(size(A))`.
- `sum`: for vectors, `sum(v)` is the sum of the elements of `v`. For two dimensional matrices, `sum(A,1)`, or simply `sum(A)`, is a row vector representing the sum of the elements over each column. `sum(A,2)` is a column vector representing the sum of the elements over each row.

The use of these functions is illustrated below. Use `help elmat` for more MATLAB functions.

```
>> eye(2,3)
ans =
     1     0     0
     0     1     0
```

```
>> zeros(2,3)
ans =
     0     0     0
     0     0     0
```

```
>> ones(2,3)
ans =
     1     1     1
     1     1     1
```

```
>> diag([1:4])
ans =
     1     0     0     0
     0     2     0     0
     0     0     3     0
     0     0     0     4
```

```
>> Bmat; B
B =
     1     0     2     5
     4     8     6    11
     7     3     9    20
```

```
>> Bmat; diag(B)
ans =
     1
     8
     9
```

```
>> diag(B,-1)
ans =
     4
     3
```

```
>> max(B)
ans =
     7     8     9    20
```

```
>> s=size(B
s =
     3     4
```

```
>> length(B)
ans =
     4
```

```
>> length([ 4 5 3 2 ]')
ans =
     4
```

```
>> sum([ 4 5 3 2 ]')
ans =
    14
```

```
>> sum(B,1)
ans =
    12    11    17    36
```

```
>> sum(B,2)
ans =
     8
    29
    39
```

Array operations. The array operations (`+,-,.*,./,.^`) introduced earlier between two vectors are defined similarly between two matrices. Again, the matrices in question should be of the same dimension, the operators act element by element, and the resulting matrix is of the same dimension of the matrices operated on. The array operations between a matrix and a scalar are also similar to those between a vector and a scalar. Check the previous examples in section 4. We discuss matrix operations in the Linear Algebra section.

We discuss next the applications of matrices in plotting.

8 MORE PLOTTING

The function meshgrid and surface plots. Assume we want to plot the function

$$z = \frac{\sin\sqrt{x^2+y^2}}{\sqrt{x^2+y^2}}$$

over the domain $-8 \leq x \leq -2$ and $1 \leq y \leq 3$. We first create a rectangular grid in the domain, say 4 points in the x-direction (-8, -6, -4, -2) and 3 points in the y-direction (1, 2, 3). The correspondence between the grid and a matrix is obvious. At each point on the grid we need to calculate the coordinates x and y and the function z. This is best done by creating a matrix `X`, containing the x-coordinates of all the points, a matrix `Y` containing the y-coordinates, and a matrix Z containing the function values. `X` and `Y` can be created in MATLAB using the function `meshgrid` with arguments the vectors representing the points in the x and y directions:

```
>> vx=[-8:2:-2];
>> vy=[1:3];
>> [X,Y]= meshgrid(vx,vy);
```

The result is:

```
>> X                                >> Y
X =                                 Y =
    -8    -6    -4    -2                 1     1     1     1
    -8    -6    -4    -2                 2     2     2     2
    -8    -6    -4    -2                 3     3     3     3
```

The columns of `X` and `Y` represent the points of the grid with constant x-values while the rows represent the points with constant y-values.

We next calculate the matrix `Z` and plot it using the command `mesh`. The remainders of the commands are:

```
>> R=sqrt(X.^2 + Y.^2);
>> Z=sin(R)./R; mesh(X,Y,Z), xlabel('x'), ylabel('y'), zlabel('z')
```

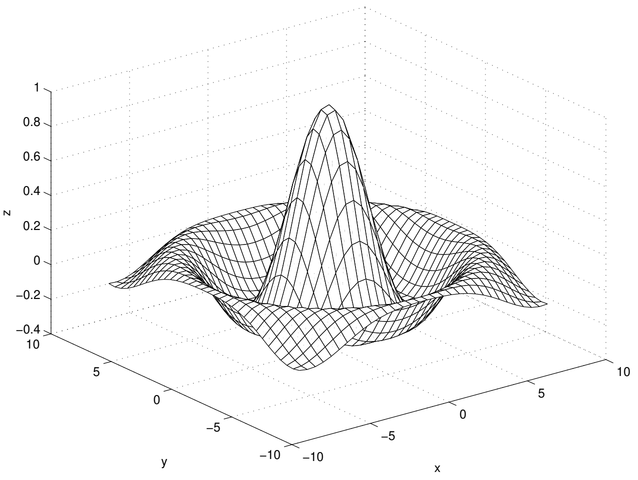
We illustrate this by plotting the same function on a larger domain with more points. The commands and the plots are:

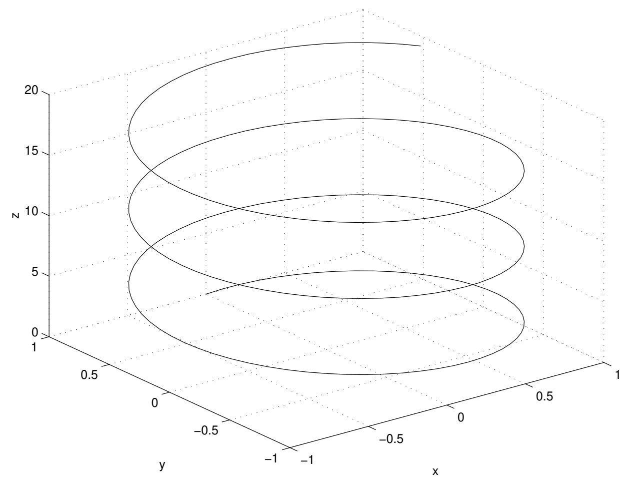
```
>> [X,Y] = meshgrid(-8:.51:8, -8:.51:8);
>> R=sqrt(X.^2 + Y.^2); Z=sin(R)./R; mesh(X,Y,Z)
>> xlabel('x'), ylabel('y'),  zlabel('z')
```

Note that a step size of 0.51 was used instead of 0.5 to avoid dividing by zero. Try it!

Three dimensional line plots. These are done in MATLAB using the command `plot3`. The following is an example of plotting a circular helix:

```
>> t=0:.1:20; x=sin(t); y=cos(t); z=t; plot3(x,y,z)
>> xlabel('x'), ylabel('y'), zlabel('z')
```





9 LINEAR ALGEBRA

Matrix multiplication and division. Given two matrices A and B with elements a_{ij} and b_{ij}, their matrix multiplication is $C = AB$ where the elements of C are given by $c_{ij} = \sum_k a_{ik}b_{kj}$. In order for C to be defined, the number of columns of A should be the same as the number of rows of B. Given two matrices `A` and `B` in MATLAB that satisfy this condition, the matrix multiplication is given by `C=A*B`. If `A` is a row vector and `B` is a column vector of the same size of `A`, `C` is then a scalar equal to their dot product.

The inverse of a square matrix `A` is given by `inv A`. There are two "matrix division" symbols in Matlab, `\` and `/` — if `A` is a nonsingular square matrix, then `A\B` and `B/A` correspond formally to left and right multiplication of `B` (which must be of the appropriate size) by `inv A`, that is `inv(A)*B` and `B*inv(A)`, but the result is obtained directly without the computation of the inverse. Thus, to solve a system `A*x=b` for the unknown vector `x`, type

```
>> A=[1 2 3; 4 5 6; 7 8 0];
>> b=[5 8 -7]';
>> x=A\b
```

which results in

```
x =
  -1
   0
   2
```

To check this result, just type A*x which verifies that

```
ans =
   5
   8
  -7
```

To learn more about this type help slash.

You will learn that when solving the system of equations A*x=b using Gauss elimination, the matrix [A b] will be reduced to row echelon form before obtaining the solution. This can obtained in MATLAB using rref. Continuing with A and b above, the row echelon form of [A b] is:

```
>> m=[A b]; rref(m)
ans =
     1     0     0    -1
     0     1     0     0
     0     0     1     2
```

We see that the last column of rref(m) is the solution in this case.

Commands for matrix factoring and decomposition. Matlab has several advanced matrix functions which will become useful to you as your knowledge of linear algebra grows. These are summarized in the following section—don't be concerned if most of these functions are unfamiliar to you now. One command which we will encounter early in the quarter is LU decomposition:

```
>> A=[1 2 3; 4 5 6; 7 8 0];
>> [L,U,P]=lu(A)
```

results in a lower triangular matrix L, an upper triangular matrix U, and a permutation matrix P such that P*A = L*U

```
L =
    1.0000         0         0
    0.1429    1.0000         0
    0.5714    0.5000    1.0000
U =
    7.0000    8.0000         0
         0    0.8571    3.0000
         0         0    4.5000
P =

     0     0     1
     1     0     0
     0     1     0
```

Checking this with P\(L*U) confirms that the original matrix A is recovered.

The following commands are but a small subset of what Matlab has to offer:

```
>> R = chol(X)
```

produces an upper triangular R so that R'*R = X. If X is not positive definite, an error message is printed.

```
>> [V,D] = eig(X)
```

produces a diagonal matrix D of eigenvalues and a full matrix V whose columns are the corresponding eigenvectors so that X*V = V*D.

```
>> p = poly(A)
```

If `A` is an N by N matrix, `poly(A)` is a row vector with N+1 elements which are the coefficients of the characteristic polynomial, `det(lambda*eye(A) - A)`.
If `A` is a vector, `poly(A)` is a vector whose elements are the coefficients of the polynomial whose roots are the elements of `A`.

```
>> [P,H] = hess(A)
```

produces a unitary matrix `P` and a Hessenberg matrix `H` so that `A = P*H*P'` and `P'*P = eye(size(P))`.

```
>> [L,U,P] = lu(X)
```

produces a lower triangular matrix `L`, an upper triangular matrix `U`, and a permutation matrix `P` so that `P*X = L*U`.

```
>> Q = orth(A)
```

produces an orthonormal basis for the range of `A`. Note that `Q'*Q = I`, the columns of `Q` span the same space as the columns of `A` and the number of columns of `Q` is the rank of `A`.

```
>> [Q,R] = qr(X)
```

produces an upper triangular matrix `R` of the same dimension as `X` and a unitary matrix `Q` so that `X = Q*R`.

```
>> [U,S,V] = svd(X)
```

produces a diagonal matrix S, of the same dimension as `X` and with nonnegative diagonal elements in decreasing order, and unitary matrices `U` and `V` so that `X = U*S*V'`.

Sample M-file. This script is a good example to look later in the quarter.

```
% matrix.m
echo on, clc
% This code uses MATLAB's eig program to find eigenvalues of
% large matrices. When we only want to display eigenvalues we use eig.
%
% Special types of matrices have eigenvalues in special positions.
% Here are experiments from a random R:
% press any key
pause, clc

R = rand(4)
eig(R)

eig(R + R')

% Notice that R + R' is symmetric, with real eigenvalues.
% Of course R (since it was random) may have real eigenvalues too.

eig(R - R')

% The matrix R - R' is skew-symmetric, with imaginary eigenvalues.
% press any key
pause, clc

eig(R'*R)

% This matrix R'*R is symmetric, with real POSITIVE eigenvalues.

eig(R*R')
```

```
% R*R' has the same eigenvalues as R'*R. But different eigenvectors!
% press any key
pause, clc

% You can create matrices with desired eigenvalues 1,2,3,4 from any
% invertible S times lambda times S inverse.

S = rand(4);
lambda = diag([1 2 3 4]);
A = S * lambda * inv(S);
eig(A)
echo off
```

10 THE FOR LOOP

Perhaps the most common and useful computational structure in programming is the loop structure, wherein a block of statements is executed repeatedly with some of its parameters changed. MATLAB has two types of explicit loops: the *for* loop, where the repetition is carried over for a predetermined number of times, and the *while* loop, where the repetition stops only when a certain condition is met. The *while* loop will be discussed in a later section.

The *for* loop is constructed in the following manner:

> `for` *loop control variable* = *array*
>
> > Set of MATLAB statements that will be executed for a number of times equal to the number of columns of array. At iteration number n, the *loop control variable* is assigned the column n of *array*.
>
> `end`

For most purposes, *array* is a row vector of the form `[`*initial_value* `:` *increment* `:` *final_value*`]`. Of course, *increment* can be omitted when it is 1.

The following script is an example of summing the components of a vector:

```
rv=[1.1, 2, -1.2, 3];
sum_rv=0;
for i=1:length(rv)
  sum_rv=sum_rv+rv(i);
end
sum_rv
```

The same task can also be performed using the following *for* loop, where the loop control variable is equal to one of the vector components at each iteration:

```
rv=[1.1, 2, -1.2, 3];
sum_rv=0;
for x=rv
  sum_rv=sum_rv+x;
end
sum_rv
```

The answer in both cases should be the same as the one obtained using the `sum` command introduced earlier: `sum(rv)`. As in other computer languages, the value of the loop control variable (`i` in the first script and `x` in the second one) cannot be changed by the commands inside the loop; it can only be used.

Implied loops. Many MATLAB commands operate with an implied loop. An example is the function `sum`, which represents the *for* loop script above. Most of the functions and commands we have used to construct and manipulate vectors and matrices contain implied loops as well. For instance, the command `x=[0:.5:5]` can be performed with

```
for i=1:11
  x(i)=(i-1)/2;
end
```

Nested loops. *for* loops can be nested as desired. Consider the following script that constructs a matrix:

```
for n=1:3
  for m=1:4
    A(m,n)=n^2+m^2;
  end
end
A
```

The result of running the script is

```
A =
     2     5    10
     5     8    13
    10    13    18
    17    20    25
```

For each value of n, the inner loop is executed for all values of m.

11 TEXT

We have been using text to produce labels and titles for plots. We will now formally discuss MATLAB texts, which are called character strings or simply strings. A string is assigned to a variable by enclosing it between single quotes. `c='How are you doing?'` will make `c` a string variable with value `'How are you doing?'`:

```
>> c='How are you doing?'
c =
How are you doing?
```

Because the single quote is used to signal the end of the string, we must type it twice in case we want it to be included within some string. In the two examples below, the first one is not valid but the second one is:

```
>> c='What's up?'
??? c='What's
            |
Missing operator, comma, or semi-colon.
```

```
>> c='What''s up?'
c =
What's up?
```

Strings as arrays. Each string is an array by itself. `c='What''s  up?'` is a row vector of length 11. Note that there are two spaces between `s` and `u`. Each of them, as well as `''`, counts as one character. Vector manipulation can be done on the string `c` like any other row vector:

```
>> c='What''s up?';c(1)
ans =
W
```

```
>> c(3:8)
ans =
at's u
```

```
>> c(4:-1:1)
ans =
tahW
```

If we have the following two row vectors of numbers, `r1=[1 2]` and `r2=[3 4]`, `[r1 r2]` will create the row vector `[ 1 2 3 4]`. Similarly, if we enclose strings between brackets and separate them by spaces or commas, we obtain another string which is the concatenation of the original ones. `c=['I ','am ','fine.']` will produce a string `c` whose value is `'I am fine.'`.

As with matrices, strings can have multiple rows. `class1=['Rony'; 'Mary'; 'Mona'; 'John']` creates a string matrix with one name per row. We can assess one name the usual way we assess one row of a matrix: `class1(3,:)` will give `Mona`. Being a matrix, all the rows of a string must have the same number of columns. `class2=['Nick'; 'Julie'; 'Min'; 'John']` will result in an error since not all the names have the same length. We can fix this problem by appending blanks to the shorter names: `class2=['Nick '; 'Julie'; 'Min  '; 'John ']`. Another way to fix this problem is to use the MATLAB function `char` which is introduced below.

Manipulating strings. MATLAB has many built-in functions to manipulate strings. We introduce some of them below: (some of the descriptions are obtained using the `help` command)

- `s = char(s1,s2,s3,...)` forms the string `s` containing the strings s1,s2,s3,... as rows. It automatically pads each string with blanks in order to form a valid matrix.
- `s = lower(string_var)` forms the string `s` by converting `string_var` to lowercase.
- `s= upper(string_var)` forms the string `s` by converting `string_var` to uppercase.
- `s = int2str(x)` rounds the number `x` to an integer and converts the result into a string `s`.
- `s = num2str(x)` converts the number `x` into a string representation `s` with about 4 digits and an exponent if required. This is useful for labeling plots with the `title`, `xlabel`, `ylabel`, and `text` commands.

Examples:

```
>> char('1','two','three')
ans =
1
two
three
```

```
>> lower('Seven UP')
ans =
seven up
```

```
>> upper('I am down')
ans =
I AM DOWN
```

```
>> int2str(4.51)
ans =
5
```

```
>> int2str(-4.49)
ans =
-4
```

```
>> num2str(3.30)
ans =
3.3
```

Special characters. As we have seen before, some characters have special role when displaying a string into a plot using `title`, `gtext`, `legend`, etc. The caret ^ makes the character that follows it a superscript while the underscore _ makes the character that follows it a subscript. A whole set of characters can be made superscript or subscript by enclosing them between curly braces and preceding them by ^ or _. In order to display the characters ^ { } _ on a plot, they need to be preceded by a back slash \. For example, the title $\mathtt{e}^{\mathtt{(-x/4)}}$ can be obtained using `title('e^{(-x/4)}')` while the title e^{-x/4} can be obtained using `title('e\^\{-x/4\}')`.

Controlling the display. Assume we wrote a long script that computed the average value of the temperature (`T`) during the month (`m`) of the year 2000 in California. Assume also that the computation yielded `T=120.1` for `m='January'`. Writing `T` on a single line in the script will display both the name and the value of `T` when the line is executed:

```
T
120.1000
```

Similarly for `m`. A better way is to use the function `disp`. If `x` is a number, `disp(x)` displays its value without printing its name. If `x` is a string, the corresponding text is displayed. We can use the function `disp` along with `num2str` to produce a good display of the results. Including the two lines

```
s=['Tav during the month of ',m,' is ',num2str(T),'F'];
disp(s)
```

in our script will give the following output:

```
Tav during the month of January is 120.1F
```

Interacting with scripts. We can also write MATLAB scripts that prompt the user for some input during the execution of the commands. This is possible using the function `input`. For example, `sal=input('What is your yearly salary?  ')` displays `What is your yearly salary?` at the prompt and waits for input from the keyboard. The input is assigned to the variable `sal` and in general, can be a number, a string, or a matrix. Examples of valid inputs are: `24000` for a number, `'I am jobless'` for a string, and `[2000, 12]` for a row vector.

The function eval. `eval(s)`, where `s` is a string, causes MATLAB to execute the string as an expression or statement. For example, typing `eval('sin(pi/2)')` at the prompt is exactly like typing `sin(pi/2)`. The result will be `1`. Using `eval`, we can input to a script a command as a string and then execute it. Consider the following script:

```
s=input('What is your yearly salary? ');
formula=input('What is the corresponding formula? ');
yearly_sal=eval(formula);
tax=.3*yearly_sal;
disp(['your tax is $',num2str(tax)]);
```

If we input `[12 1500]` to the first question and `'s(1)*s(2)'` to the second question, `yearly_sal` will be 18000 and the output will be

```
your tax is $5400
```

Taylor series example. The Taylor series of $\sin x$ is given by

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots .$$

The following script plots $\sin x$ and its first 6 approximations:

```
clf; x=[0:0.1:5]; y=sin(x); plot(x,y)
axis([ 0 5 -2 5]), grid on,  hold on,

y=x; deno=1; plot(x,y), pause
for i=2:6
  n=2*i-1; deno=-deno*(n-1)*n;
  y=y + x.^n/deno;
  plot(x,y), pause
end
```

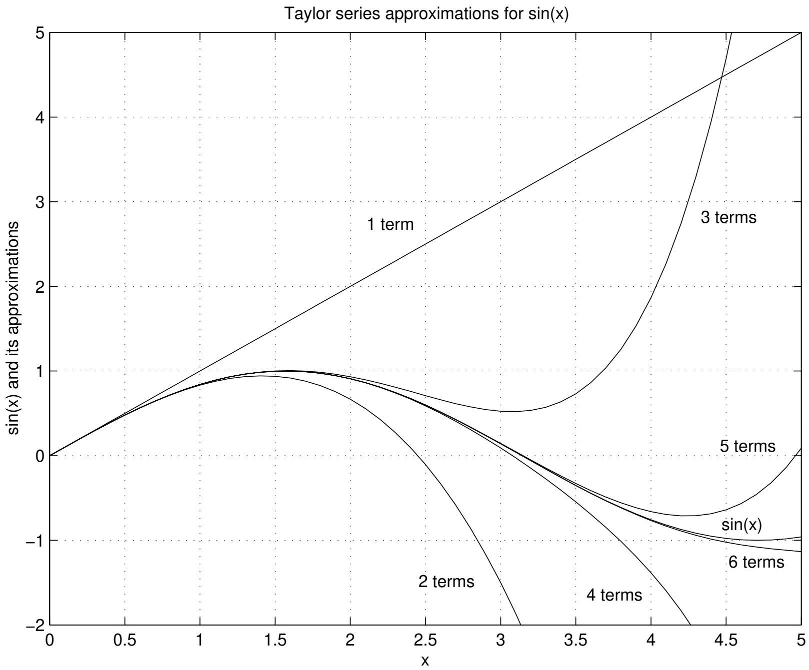
The `clf` command clears the current figure if there is already one. The `pause` command causes the script to stop and wait for the user to strike any key before continuing. Without `pause`, the script above will be executed very fast and will yield the figure containing the 7 curves without giving us a chance to see which curve corresponds to each iteration. A variation of the `pause` command is `pause(n)` which pauses the script for `n` seconds before continuing. No need for a keyboard strike to continue.

We add to the script above the necessary commands to label the different curves. For this, we construct a string matrix containing the labels using the `char` function. We also use the function `eval` to write a series of commands in a compact form. The extended script is given below. The three dots at the end of the first line are used to tell MATLAB that the second line is a continuation of the first one.

```
labels=char('1 term', '2 terms', '3 terms', ...
       '4 terms','5 terms', '6 terms', 'sin(x)');
pl='plot(x,y); gtext(labels(i,:))';

clf; x=[0:0.1:5]; y=sin(x); i=7; eval(pl);
axis([ 0 5 -2 5]), grid on,  hold on,
```

```
y=x; deno=1; i=1; eval(pl)
for i=2:6
  n=2*i-1; deno=-deno*(n-1)*n;
  y=y + x.^n/deno;
  eval(pl)
end
title('Taylor series approximations for sin(x)')
xlabel('x'), ylabel('sin(x) and its approximations')
print -deps taylor
```



12 USER-DEFINED FUNCTIONS

We have experienced the power of MATLAB through its wide range of useful functions. By functions we refer to all kind of MATLAB commands and not only the elementary mathematical ones. The functions that we have used so far can be classified according to three categories:

- commands that take one or more input arguments, compute the required results (one or more) using the input, and pass these results back to us: `y=log(x)`, `s=char(s1,s2,s3)`, `[X,Y]=meshgrid(x,y)`, ...
- commands that perform some required task using one or several input arguments but do not pass any result back: `xlabel('x')`, `plot(x,y)`, ...
- simple commands that perform some required task and do not require any input arguments nor pass any result back: `format`, `clear`, `grid on`, `clf`, `pause`, ...

We will learn in this section how to extend MATLAB's library by creating our own functions.

Constructing functions. A user-defined function consists of two parts: a function definition line and a body of commands that perform tasks and compute results. The whole content is written in a separate text file with a `.m` extension (like scripts). This file is called a function file.

The function definition line has the name of the function and a list of the input and output arguments, if there are any. Its syntax is :

`function` [*out_var1, out_var2, ...*] = *function_name* (*in_var1, in_var2, ...*)

The input and output arguments are the usual variables (scalars, vectors, or arrays of numbers or strings). The function name should be the same as the name of the function file (without the `.m` extension) and it must follow the same naming rule as for a MATLAB variable. We list below some examples of definition lines and their corresponding function files.

function definition line	function file
`function [perimeter,area]=rectangle(b,h)`	`rectangle.m`
`function [area]=rectangle(b,h)`	`rectangle.m`
`function area=rectangle(b,h)`	`rectangle.m`
`function [ ]=plot_circle(r)`	`plot_circle.m`
`function plot_circle(r)`	`plot_circle.m`
`function [ ]=just_say_hi( )`	`just_say_hi.m`
`function just_say_hi`	`just_say_hi.m`

Note that if there is one output variable [ ] can be omitted, if there are no output variables [ ]= can be omitted, and if there are no input variables ( ) can be omitted. We provide below 4 examples that illustrates different situations.

Example 1 - one output variable.

```
function dydt=f(t,y)
    dydt= y./t;
```

The function file must be called `f.m`. The function `f(t,y)` is accessible for subsequent calculations in MATLAB like any other command. Its output is the element by element division of its second input argument by its first input argument (therefore the order of the input and output arguments is important). Examples:

```
>> u=2; v=3.6; w=f(u,v)        >> f(2,4)        >> f(4,2)
w =                            ans =            ans =
    1.8000                          2               0.5000
```

The variables `t`, `y`, `dydt` have dummy names and are local to the function. It is not necessary when calling the function `f` that the inputs are named `t` and `y` and the output is named `dydt`.

The arguments `t` and `y` (and thus the output `dydt`) can be vectors or matrices since the body of the function is array operations compatible. Try `t = [1:.1:4]; y=exp(t); plot(y,f(t,y))`.

Example 2 - two output variables. Assume that the function `meshgrid` that we used before is not available. The following function gives the same results:

```
function [X,Y]=my_meshgrid(x,y)
    for i=1:length(y)
        X(i,:)=x;
    end
    for i=1:length(x)
        Y(:,i)=y.';
    end
```

Compare the example below to the first example in Section 8.

```
>> vx=[-8:2:-2]; vy=[1:3]; [X,Y]= my_meshgrid(vx,vy);
```

The result is

```
>> X                                >> Y
X =                                 Y =
    -8    -6    -4    -2                 1     1     1     1
    -8    -6    -4    -2                 2     2     2     2
    -8    -6    -4    -2                 3     3     3     3
```

Example 3 - no output variables.

```
function plot_circle(r)
    % number of points to plot
    num_pts = 100;
    theta = linspace(0,2*pi,num_pts);
    % compute the coordinates
    x = r * cos(theta);
    y = r * sin(theta);
    plot(x,y,'.')
    % enlarge the axes a bit
    axis([-1.1,1.1,-1.1,1.1]*r)
    % make the axes square
    axis square
```

Try executing the following:

```
>> for x=1:5,  plot_circle(x), hold on, end
```

Example 4 - no input or output variables.

```
function hi
    disp('Just want to say HI. Bye.')
```

Executing `hi`, we get

```
>> hi
Just want to say HI. Bye.
```

Comparing function files and script files. As with script files, function files are not entered in the command window but rather are external text files created with a text editor. Both files have to be in a directory that MATLAB knows about; this can either be the current directory or any directory on the MATLAB path. Without the definition line, the function file becomes a script file.

A function file is different than a script file in that it communicates with MATLAB workspace only through the output variables it creates. Intermediate variables within the function (like those in my_meshgrid and `plot_circle`) do not appear or interact with the MATLAB workspace. They are not defined after the function is executed.

Using a user-defined function instead of a script file of commands has two advantages. First, we do not have to worry about any interaction between variables that exist in the workspace prior to calling the function and the variables created and used within the function. For example, changing the value of `pi` before calling `plot_circle` does not affect the value of `pi` used in the third line of `plot_circle`. Similarly, using the variable `x` in `plot_circle` and as a counter in the `for` loop in Example 3 doesnt create any problems. Remember that the value of the loop counter shouldn't be changed inside the loop. Second, the function allows using a list of input and output variables. We can thus easily repeat a lengthy set of commands with different parameters.

13 DECISION MAKING

Assume we want to plot the function

$$f(x) = \begin{cases} 1 & \text{if } 0 \leq x \leq 1 \\ -1 & \text{otherwise} \end{cases}$$

for x=[-1:.1:2]. We need first to construct the corresponding vector f and then we can use the command plot(x,f). This can be achieved if we can write commands that are equivalent to:

> for i=1:length(x)
>   *if* x(i) ≥ 0 *and* x(i) ≤ 1 *let* f(i)=1, *otherwise, let* f(i)=-1
> end

MATLAB allows this by providing a set of operators to compare numbers and a decision-making structure to decide whether or not to execute some commands. We discuss these below.

Relational Operators. Comparison between numbers is done through 6 operators called the relational operators. They are:

Relational Operator	Meaning
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal to
~=	not equal to

The comparison is done by placing a relational operator between the numbers: 3<2, x~=y, etc. Expressions of this from are called logical or relational expressions. The result of a comparison (or the value of a logical expression) is either true or false. MATLAB assigns a numerical value of +1 for true and 0 for false. The value of a logical expression can therefore be used in mathematical operations. Examples:

```
>> 3 < 2
ans =
     0
```

```
>> 4 >= 4
ans =
     1
```

```
>> x=3; y=(x==4) - 4; y
y =
    -4
```

Note that = and == mean different things: == compares two variables and returns ones where they are equal and zeros where they are not; = on the other hand is used to assign the output of an operation to a variable. Also, the arithmetic operators take precedence over the relational operators: when x=3, (x==4) - 4 is equal to 0 - 4=-4 while x==4 - 4 is equal to x==0 which is 0.

As for the arithmetic operators, the relational operators can also be applied between arrays of equal size, or between an array and a scalar, on an element by element basis. Consider the following examples:

```
>> x = [6 3 9]; y = [10 -2 9];
>> z  = x < y
z =
     1     0     0
```

```
>> x=-3:1:3
x =
    -3    -2    -1     0     1     2     3
>> x>=0
ans =
     0     0     0     1     1     1     1
```

In the first example, each element of the row vector x is compared to the corresponding element of the row vector y; in the second one, each element of x is compared to 0.

We illustrate a good use of logical expressions in the following example:

```
>> x=[-4:1]/5
x =
   -0.8000   -0.6000   -0.4000   -0.2000         0    0.2000
>> sin(x)./x
Warning: Divide by zero.
ans =
    0.8967    0.9411    0.9735    0.9933       NaN    0.9933
```

Computing the function $\sin(x)/x$ at $x = 0$ gives an error since division by 0 is undefined in MATLAB although $\sin(0)/0$ is 1. Replacing the 0 in x by a very small number (the constant eps for example, which is 2.2204e-16) gives the correct limiting answer:

```
>> x=x+(x==0)*eps;
>> sin(x)./x
   -0.8000   -0.6000   -0.4000   -0.2000         0    0.2000
>> sin(x)./x
ans =
    0.8967    0.9411    0.9735    0.9933    1.0000    0.9933
```

`x==0` is a vector of the same length of `x` and is 1 whenever `x` is zero, and 0 otherwise. Adding `(x==0)*eps` to `x` replaces the zeros in `x` (only one zero in this case) by `eps`.

Logical Operators. Consider the example stated at the beginning of this Section. In order to set `f(i)` to 1, the logical expressions `x(i)>=0` and `x(i)<=1` need to be true. In other words, 'both of them' need to be true. This motivates combining two logical expressions to give another one. This is done in MATLAB through 3 operators called the logical operators. They are:

Logical Operator	Meaning
&	and
|	or
~	not

Logical expressions are combined using the logical operators according to the following rules:

$$
\begin{array}{ccccc}
\textit{true} & \& & \textit{true} & = & \textit{true} \\
\textit{true} & \& & \textit{false} & = & \textit{false} \\
\textit{false} & \& & \textit{false} & = & \textit{false} \\
\textit{true} & | & \textit{true} & = & \textit{true} \\
\textit{true} & | & \textit{false} & = & \textit{true} \\
\textit{false} & | & \textit{false} & = & \textit{false} \\
 & \tilde{}\,\textit{true} & & = & \textit{false} \\
 & \tilde{}\,\textit{false} & & = & \textit{true}
\end{array}
$$

The following examples illustrate the use of logical operators with scalars and vectors:

```
>> a = 2; b = 8;
>> c =  a > b  &  b < 20;
>> c
c =
     0
```

```
>> x = [6 3 9]; y = [10 -2 9];
>> z = (x + y > 10)  &  (x .* y > 0);
>> z
z =
     1     0     1
```

Using the relational and logical operators, we can now construct `f` of the example at the beginning of this Section as follows: (without the use of the *if* structure which is discussed next)

```
>> x=[-1:.1:2];
>> f=(x>=0)  & (x<=1);
>> f=-1 + 2*f;
```

The second line gives a vector of the same length of `x` which is 1 for $0 \leq x \leq 1$ and 0 otherwise. The third line gives the desired `f`, which is 1 for $0 \leq x \leq 1$ and -1 otherwise. Try it!

if-else-end structures. (parts of this subsection are taken from the User's Guide of the 'Student Edition of MATLAB') Many times, sequences of commands must be conditionally evaluated based on a relational test. In programming languages this logic is provided by some variation of an `if-else-end` structure. The simplest `if-else-end` structure is

> `if` *expression*
>   commands executed if *expression* is true
> `end`

If the logical expression is false, the block of statements is ignored and the program proceeds to the next statement after the `end`. Consider for example the script `apple.m` below:

```
n=input('How many apples are you buying? ');
cost=n*.5; discount='0%';
if n>=10
    cost=0.8*cost;
    discount='20%';
end
disp(['The cost is $',num2str(cost)]);
disp(['It reflects ',discount,' discount'])
```

We run it twice:

```
>> apple
How many apples are you buying? 13
The cost is $5.2
It reflects 20% discount
```

```
>> apple
How many apples are you buying? 3
The cost is $1.5
It reflects 0% discount
```

In cases where there are two alternatives the `if-else-end` structure is

> `if` *expression1*
>   commands executed if *expression1* is true
> `else`
>   commands executed if *expression1* is false
> `end`

As an example, we use the `if-else-end` structure to construct the function `f` given earlier:

```
x=[-1:.1:2];
for i=1:length(x)
    if (x(i)>=0) & (x(i)<=1)
        f(i)=1;
    else
        f(i)=-1;
    end
end
```

When there are three or more alternatives, the `if-else-end` structure takes the form

> `if` *expression1*
>   commands executed if *expression1* is true
> `elseif` *expression2*
>   commands executed if *expression2* is true
> `elseif` *expression3*
>   commands executed if *expression3* is true
> `elseif` $\cdots$
> $\vdots$
> `elseif` *expression_n*
>   commands executed if *expression_n* is true

```
else
    commands executed if no other expression is true
end
```

As an example, consider the following script which we call `test.m`:

```
a = 2;   b = 8;   c = 5;
if a>= b
    c = 10;
elseif  b == c + 3
    c = 15;
elseif  b  >  a
    c = 20;
else
    c = 25;
end
```

Running it yields

```
>> test; c
c =
    15
```

Note in the above example that both of the `elseif` conditional statements are true. However, after the first test that results in true, the next statement to be executed will be that following the `end` statement.

while loops. As opposed to a `for` loop that evaluates a group of commands a fixed number of times, a `while` loop evaluates a group of commands an indefinite number of times until a condition is met.

The general form of a `while` loop is

```
while expression
    commands executed as long as expression remains true
end
```

The function `my_row.m` below is equivalent to constructing a row vector using the colon notation introduced in Section 4.

```
function x=my_row(xa,inc,xb)
    i=1; x(i)=xa;
    while (x(i)+inc) <= xb
        i=i+1;
        x(i)=x(i-1)+inc;
    end
```

We execute this function with examples from Section 4:

```
>> my_row(1.,.5,2.5)
ans =
    1.0000    1.5000    2.0000    2.5000

>>  my_row(1.,.5,2.4)
ans =
    1.0000    1.5000    2.0000

>> my_row(1.,.5,2.9)
ans =
    1.0000    1.5000    2.0000    2.5000
```

Please refer to the links on the class website to learn lots more about MATLAB. I would greatly appreciate any comments on this handout (alberth@stanford).