MATLAB™ TUTORIAL

Matlab (MATrix LABoratory) is an interactive software system for numerical computations and graphics. As the name suggests, Matlab is especially designed for matrix computations: solving systems of linear equations, computing eigenvalues and eigenvectors, factoring matrices, and so forth. In addition, it has a wide variety of graphical (visualization) capabilities, and can be extended through programs written in its own programming language. An introduction to some of the most useful features of Matlab is given in the following sections. The best way to learn to use Matlab is to read this while running Matlab, trying the examples and experimenting.

Basic Features of MATLAB

Matlab, by its design, is an interpretive computing environment. When you startup Matlab, you simply come across a prompt (much like DOS or UNIX prompt) such as

```
To get started, type one of these: helpwin, helpdesk, or demo. For product information, type tour or visit www.mathworks.com.
>> _
```

Matlab interpreter awaits your commands at this point. Matlab can be used much like a calculator right away. For example, try typing

```
>>(45 + 7)/3
```

and you should get back the answer right away

```
ans =
   17.3333
```

Here `ans` is a temporary variable that stores the result of an instantaneous calculation. The result can then be used in subsequent computations. Here are some examples:

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

where `sqrt` is a built-in square root function. Matlab constitutes large number of predefined functions (`sin, cos, exp, log, tanh, abs`) you have got accustomed programming with a high-level language:

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

Like BASIC language, the results of various arithmetic operations can be assigned to the variables that are defined. The basic data type in Matlab is an *n*-dimensional array of double precision numbers. In earlier versions of Matlab, every variable was a two-dimensional array (matrix), with one-dimensional arrays (vectors) and zero-dimensional arrays (scalars as special cases). The following commands show how to enter numbers, vectors and matrices, and assign them to variables:

```
>> a = 2.12
a =
     2.12
>> x = [1; 2; 3]
x =
     1
     2
     3
>> A = [1 2 3; 4 5 6; 7 8 9]
A =
     1     2     3
     4     5     6
     7     8     9
```

Notice that the rows of a matrix are separated by semicolons (;), while the entries on a row are separated by spaces (or commas). A useful command is ``whos", which displays the names of all defined variables and their types:

```
>> whos
  Name      Size         Bytes  Class

  A         3x3             72  double array
  a         1x1              8  double array
  ans       1x1              8  double array
  x         3x1             24  double array

Grand total is 14 elements using 112 bytes
```

Note that each of these four variables is an array; the ”shape" of the array determines its exact type. The scalar `a` and `ans` are 1x1 arrays, the vector `x` is a 3x1 array, and `A` is a 3x3 array (see the ``size" entry for each variable). Note that the rules for picking up a valid name for an array (matrix/vector/scalar) are similar to those of high-level languages like C/C++.

Matlab allows arrays to have complex entries. The complex unit $i = \sqrt{-1}$ is represented by either of the built-in variables `i` or `j`:

```
>> i
ans =
        0 + 1.0000i
>> -7.9+a*i
ans =
  -7.9000 + 2.1200i
```

These examples show how complex numbers are displayed in Matlab. Most arithmetic operators as well as certain functions work for the matrices and vectors as well. For instance,

```
>> sin(x)
ans =
    0.8415
    0.9093
    0.1411
```

In this case, sine of each vector element is computed. A built-in variable that is often useful is π:

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

Strange enough, the user is not limited by the context of finite numbers in Matlab. One can assign infinity to variables:

```
>> a = inf
a =
   Inf
>> 1/a
ans =
     0
```

Here, `inf` is a built-in Matlab variable for representing infinite values (at least within the framework of IEEE 754 standard!). At this point, rather than providing a comprehensive list of functions and variables available in Matlab, it is explained how to get this information from Matlab itself. An extensive online help system can be accessed by commands of the form `help <command-name>`. For example:

```
>> help ans
ANS  Most recent answer.
    ANS is the variable created automatically
    when expressions are not assigned to
    anything else. ANSwer.
```

A good place to start is with the command `help help`, which explains how the help systems works, as well as some related

commands. Typing `help` by itself produces a list of topics for which help is available; looking at this list we find the entry ``elfun – elementary math functions." Typing `help elfun` produces a list of the math functions available.

It is often useful, when entering a vector/matrix/scalar, to suppress the display; this is done by ending the line with a semicolon (;). For instance, type

```
>> b = -9.75;
```

The scalar `b` is created in the workspace (i.e. in computer's memory). However, the semicolon at the end of the statement suppresses the output.

At some point in time, it is possible to access certain elements of the matrices defined earlier. For instance, typing

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

will give the A matrix element (which is typed earlier!) at the second row and the third column. Similarly, if

```
>> A(:,2)
ans =
     2
     5
     8
```

is typed it is observed that Matlab has returned the entire second column of A. Here, colon (:) serves as a symbol to signal Matlab interpreter to omit the row numbers and to return the entire column. Likewise, when

```
>> A(3,:)
ans =
     7     8     9
```

is typed one gets the third row elements of A. Matlab also enables the user to access sub matrices:

```
>> A(1:2,2:3)
ans =
     2     3
     5     6
```

In this case, asking Matlab to consider the rows of A from 1 to 2 creates a new matrix (2x2) (1:2). Within these rows, Matlab is asked to take elements in columns 2 to 3.

Matlab enables the user to combine different matrices to form new ones. For instance, in this example:

```
>> B = [A; zeros(1,3); x']
B =
     1     2     3
     4     5     6
     7     8     9
     0     0     0
     1     2     3
```

A new matrix `B` (5x3) is created using `A` (3x3) and `x` (3x1) defined earlier. As can be noticed, some new features of Matlab is utilized in this example. The most important one is `zeros(n,m)` function which generates a n by m matrix full of zero element. The other one is `(')` operator which takes the transpose of a given matrix. In this case, x (3x1) is transposed to form a new row vector (1x3). In the given example, a row vector full of zeros and the transposed x matrix is appended to our old matrix A. It should be noted how semi colons are utilized to combine these different Matlab variables. It is just like typing the elements of a vector. However, care must be exercised on the dimensions of these variables as they all must be conformable. This time, trying the following:

```
>> B = [x ones(3,1) A]
B =
     1     1     1     2     3
     2     1     4     5     6
     3     1     7     8     9
```

Once again, another important Matlab function: `ones(n,m)`! It is pretty much like `zeros` function but it creates an n by m matrix full of ones instead. Notice that this time, the semicolon is not used to separate out the elements. The thing done in this example is very similar to typing a row vector. However, this time row vector elements are not scalars (1x1), but matrices/vectors having 3 rows. These concepts may look complicated in the first glance. But as one becomes more experienced in Matlab, this will be a second nature.

Standard matrix operations

Now basic matrix operations in Matlab will be considered. It can be started by typing the following matrices:

```
>> A = [1 2 3; 4 5 6; 7 8 9];
>> B = [1 1 1; 2 2 2; 3 3 3];
>> C = [1 2; 3 4; 5 6];
```

Matlab enables the user to perform arithmetic operations (+, –, *) on matrices. For example, consider the following commands:

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

Matrix multiplication is also defined:

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

Just like the consepts learned in linear algebra, the matrices must be conformable to conduct arithmetic operations on them. Nevertheless, there are some exceptions in Matlab:

```
>> A–3
ans =
    -2    -1     0
     1     2     3
     4     5     6
```

Here, the inner matrix dimensions do not agree. However, scalars (1x1) are considered as major exceptions. The result obtained is that the scalar 3 is subtracted from every element of the matrix A. In some cases, it is not desired to perform regular matrix multiplication. Instead, it is wished to multiply the corresponding elements of two matrices. Considering the following example:

```
>> A.*B
ans =
     1     2     3
     8    10    12
    21    24    27
```

(.*) is the right operator for the job! Typing `help ops` will give informstion on other operator, which are not covered in this section!

Now comes another important issue: solving linear equations using Matlab. It is known that if *A* is a square, nonsingular matrix, then the solution of the equation $Ax = b$ is $x = A^{-1}b$. Matlab implements this operation with the backslash operator:

```
>> A = [1 1 –4; 2 –3 2; 5 –8 10];
>> b = [8; 1; 2];
>> x = A\b
x =
    5.5000
    3.5000
    0.2500
```

Thus A\b is (mathematically) equivalent to multiplying *b* on the left by A^{-1} (however, Matlab does *not* compute the inverse matrix; instead it solves the linear system directly). When used with a non-square matrix, the backslash operator solves the appropriate system in the least-squares sense; more details are available if help slash is typed. Of course, as with the other

arithmetic operators, the matrices must be compatible in size. As an alternative, trying the following statement:

```
>> x = inv(A)*b
x =
    5.5000
    3.5000
    0.2500
```

In this case, matrix inversion command of Matlab (`inv`) is employed. Matlab has a wide variety of useful matrix operation commands (a.k.a. functions) including `det` (determinant), `norm` (matrix norm), etc. Typing `help matfun` will give more information about these specialized functions for matrices.

In addition to solving linear systems (with the backslash operator), Matlab performs many other matrix computations. Among the most useful is the computation of eigenvalues and eigenvectors with the `eig` command. If `A` is a square matrix, then `ev = eig(A)` returns the eigenvalues of `A` in a vector, while `[V,D] = eig(A)` returns two matrices V and D. `V` is a matrix whose columns are eigenvectors of A, while `D` is a diagonal matrix whose diagonal entries are eigenvalues. Here is an example:

```
>> A = [1 2 3; 4 5 6; 7 8 9];
>> eig(A)
ans =
   16.1168
   -1.1168
   -0.0000
>> [V,D] = eig(A)
V =
   -0.2320   -0.7858    0.4082
   -0.5253   -0.0868   -0.8165
   -0.8187    0.6123    0.4082
D =
   16.1168         0         0
         0   -1.1168         0
         0         0   -0.0000
```

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

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

Vectors and graphs

Matlab includes a number of functions (commands) to create special matrices; the following command creates a row vector whose components increase arithmetically:

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

The components can change by non-unit steps:

```
>> x = 0:.1:1
x =
  Columns 1 through 4
         0    0.1000    0.2000    0.3000
  Columns 5 through 8
    0.4000    0.5000    0.6000    0.7000
  Columns 9 through 11
    0.8000    0.9000    1.0000
```

A negative step is also allowed. The command `linspace` has similar results; it creates a vector with linearly spaced entries. Specifically, `linspace(a,b,n)` creates a vector of length *n* with entries {a, a+(b-a)/(n-1), a+2(b-a)/(n-1),…}:

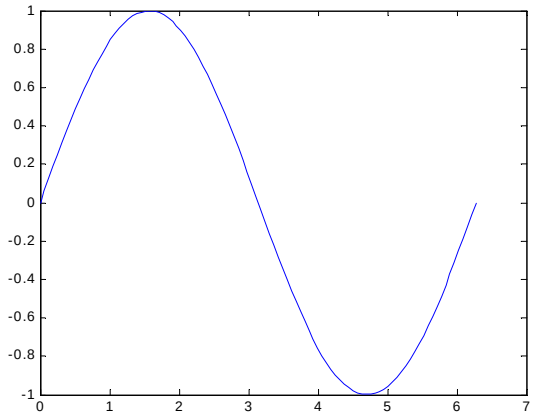
```
>> linspace(0,1,11)
ans =
  Columns 1 through 4
         0    0.1000    0.2000    0.3000
  Columns 5 through 8
    0.4000    0.5000    0.6000    0.7000
  Columns 9 through 11
    0.8000    0.9000    1.0000
```

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

It is easy to create the needed vectors to graph a built-in function since most Matlab functions are designed to return a vector (matrix) if the input argument is a vector (matrix). This means that if a built-in function such as sine is applied to an array, the effect is to create a new array of the same size whose entries are the function values of the entries of the original array. For example,

```
>> x = linspace(0,2*pi,100);
>> y = sin(x);
>> plot(x,y)
```

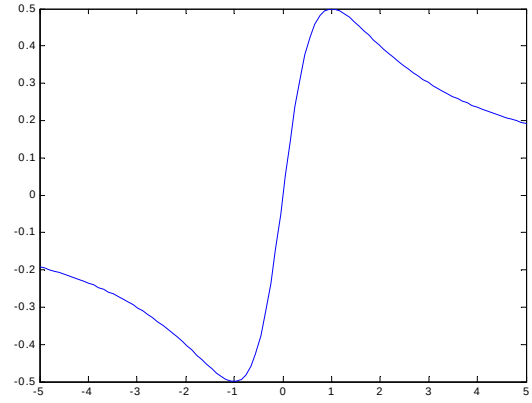
Matlab opens up a graphics window and displays the result:



As a second example, the following function is plotted $y = x/(1 + x^2)$ for $-5 \le x \le 5$. To graph, one should type the following:

```
>> x = linspace(-5,5,100);
>> y = x./(1+x.*x);
>> plot(x,y)
```

Here, arithmetic operator preceded by (.) is utilized. As discussed before, "dot" operators perform the arithmetic operation on the corresponding elements of two vectors (or matrices). In this case, firstly $1 + x^2$ is calculated using array multiplication (.*). The result is a vector (1 by 100). Hence employing array division (./), the vector `x` is divided by this result in order to form the vector y. The plot is shown below:



Typing `help plot` will give more information on this very useful command. Matlab is not limited by only 2-D graphs. It has a wide variety of advanced visualization commands including `mesh, surf, plot3d,` and `contourf.`

Programming in Matlab

So far, all Matlab commands are keyed sequentially at Matlab prompt. As might have already been realized by now, this is a very tedious and slow process. Thus, Matlab offers a practical solution for this problem. One can type in all the commands in a text file having an extension of "m". This "script" file must be located in the current work directory. When the name of this "m" file is entered at prompt level (as if it were a Matlab command), Matlab reads the content of this file and executes each command in this file one by one sequentially. In fact, what is created is a Matlab program (code) called "m-code." Matlab provides a number of standard programming constructs, such as loops (`for … end`; `while … end`) and conditionals (`if … else … end`).

The use of conditionals in Matlab programs is very similar to those of common high-level languages. The general form of if statement is as follows:

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

Here is an example for the if statement:

```
if x == 0
  y = 0;
elseif x == 1
   y = 1;
else
   y = 1 - 2*(x - 0.25);
end
```

The logical operators in Matlab are <, >, <=, >=, == (logical equal), and ~= (not equal). These are the Boolean operators returning the values 0 and 1 (for scalar arguments).

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

```
for k = [1,2,3,4]
   disp(k^2)
end
```

Here, `disp` (display) is a Matlab function that simply displays the argument. If this piece of code is run (or just typed at the Matlab prompt), one gets the following output:

```
     1
     4
     9
    16
```

The loop, like an if-block, must be terminated by an `end` statement. This loop can be alternatively written as

```
for k = 1:4
    disp(k^2)
end
```

Recalling that 1:4 is the same as [1,2,3,4]. Similarly, Matlab offers another useful loop control command: `while`. The general form of the while statement is

```
while expr
    statements
end
```

The while-loop repeats the `statements` as long as the given `expr` is true. Here is a non-trivial example:

```
x = 1;
while 1 + x > 1
  x = x/2;
end
disp(2*x)
```

If this strange looking code is run, one will get `2.2204e-016,` which is the relative floating-point accuracy of the computer used (i.e. machine epsilon). Another important command one needs to remember is the `break` command. It simply helps the user to terminate `for` or while `loop` that are currently running.

While programming in Matlab, one may need to define custom subroutines (procedures/functions). Unlike high-level languages, these functions cannot be included in the main program (i.e. main collection of Matlab commands). Each function in Matlab has to be an individual m-file (m-code). A function is defined in an m-file that begins with a line of the following form:

```
function [output1,output2,...]
              = cmd_name(input1,input2,...)
```

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

```
function [y,z] = fcn(x)
z = 1 + x.*x;
y = 1./z;
plot(x,y)
```

At Matlab prompt, this new function can be assessed as if it were a Matlab command:

```
>> x = linspace(-5,5,100);
>> fcn(x)
```

This new function plots out the function $y = 1/(1+x^2)$. It is worthwhile to note that it has two output arguments that are ignored. Try the following statements:

```
>> y1 = fcn(x);
>> [y2, z2] = fcn(x);
```

One will find out that the function is now able to pass different output arguments (vectors) to the workspace. This is a probably one of the most powerful features of Matlab which enables development of auxiliary Matlab commands/functions. In fact, all of the functions in Matlab toolboxes were formed this way.

Where to go from here

As mentioned, the best way to learn Matlab is definitely to practice! The textbook (Ogata, Modern Control Engineering, 1997, 3/e) has some reading materials on Matlab (pp. 960-982). There are numerous textbooks on Matlab and its specific applications. One may also find very nice Matlab tutorials on the web (For instance, the web site below can be checked out).

Acknowledgement

Matlab is a trademark of MathWorks, Inc. (Mass., USA) (www.mathworks.com). This text is partly adapted from http://www.math.mtu.edu/~msgocken/intro/node1.html.