Tutorial

This is an interactive introduction to MATLAB. A sequence of commands has been provided for you to try. After each command you should type the `Return` or `Enter` key to execute the command. MATLAB runs on many different systems. You should consult with your instructor or the MATLAB documents on the method for getting into the MATLAB system. In the course of the tutorial if you get stuck on what a command means type

`help` *command name*

and then try the command again. You should record the outcome of the command either in the space provided on the right side of the page or in a notebook.

1. BUILDING MATRICES

MATLAB has many types of matrices which are built into the system. A 7×7 matrix with random entries is found by typing the commands followed by a return:

```
rand(7)
rand(2,5)
help rand
```

Another special matrix, called a Hilbert matrix is

```
hilb(5)
help hilb
```

A 5×5 magic square is given by

```
magic(5)
help magic
```

Here are some of the standard matrices from linear algebra.

```
eye(6)
zeros(4,7)
ones(5)
```

You can also build matrices of your own with any entries that you may want.

```
[1 2 3 5 7 9]
[1, 2, 3;4, 5, 6;7, 8, 9]
[1  2   <RETURN>  3  4   <RETURN>  5  6]
```

Here `<RETURN>` means to type the `Return` or `Enter` key.

```
[eye(2);zeros(2)]
[eye(2);zeros(3)]
```

You should have got an error message on this. Why?

```
[eye(2),ones(2,3)]
```

2. VARIABLES

MATLAB has built-in variables like `pi`, `eps`, and `ans`. You can learn their values.

```
pi
eps
help eps
ans
```

At any time if you want to know the active variables try:

```
who
help who
```

The variable `ans` will keep track of the last output which was not assigned to another variable.

```
magic(6)
ans
x=ans
x
x=[x,ones(6)]
x
```

Since you have created a new variable, `x`, it should appear as an active variable.

```
who
```

To remove a variable try this.

```
clear x
x
who
```

3. FUNCTIONS

```
a=magic(4)
```

Take the transpose of `a`.

```
a'
```

If `a` has complex entries `a'` is not the transpose, rather it is the transpose of the conjugate of `a`.

```
3*a
-a
a+(-a)
b=max(a)
max(b)
```

Some functions can return more than one value. In the case of `max` it will return the maximum value and also the column index where the maximum value occurs.

```
[m,i]=max(b)
min(a)
b=2*ones(6)
a*b
a
```

Usually a dot in front of an operation will change the operation. In the case of multiplication, `a.*b` will make it into an entry-by-entry multiplication instead of the usual matrix multiplication.

`a.*b` (there is a dot there!)

```
x=5
x^2
a*a
a^2
```

`a.^2` (another dot)

```
a
triu(a)
tril(a)
diag(a)
diag(diag(a))
c=rand(4,5)
size(c)
[m,n]=size(c)
m
d=.5-c
```

There are a many functions which we apply to scalars that can be applied to matrices also.

```
sin(d)
exp(d)
log(d)
abs(d)
```

MATLAB has functions to round floating point numbers to integers. These are `round`, `fix`, `ceil`, and `floor`. The next few commands will help you determine how these work.

```
f=[-.5  .1  .5]
round(f)
fix(f)
ceil(f)
floor(f)
sum(f)
prod(f)
```

4. RELATIONS AND LOGICAL OPERATIONS

In this section you should think of 1 as "true" and 0 as "false." The notations `&`, `|`, $\sim$ stand for "and," "or," and "not," respectively. The notation `==` is a check for equality.

```
a=[1,  0,  1,  0]
b=[1,  1,  0,  0]
a==b
```

```
a<=b
~a
a&b
a&~a
a|b
a|~a
```

There is a function , `any` , to determine if there is a nonzero entry in a vector, and a function , `all` , to determine if all the entries are nonzero.

```
a
any(a)
all(a)
c=zeros(1,4)
d=ones(1,4)
any(c)
all(d)
e=[a',b',c',d']
any(e)
all(e)
any(all(e))
```

5. COLON NOTATION

MATLAB also offers some very powerful ways for creating vectors.

```
x=-2:1
length(x)
-2:.5:1
-2:.2:1
a=magic(5)
a(2,3)
```

Now we will use the colon notation to get a column of `a`.

```
a(2,:)
a(:,3)
a
a(2:4,:)
a(:,3:5)
a(2:4,3:5)
a(1:2:5,:)
```

You can put a vector in the row or column position of `a`.

```
a(:,[1,  2,  5])
```

```
a([2,  5],[2,  4,  5])
```

You can also make assignment statements using a vector or a matrix.

```
b=magic(5)
b([1  2],:)=a([2  1],:)
a(:,[1  2])=b(:,[3  5])
a(:,[1  5])=a(:,[5  1])
a=a(:,5:-1:1)
```

This has been a sample of the basic MATLAB functions and the matrix manipulation techniques. At the end of this tutorial there is a listing of some of MATLAB's functions. The functions that you have available will vary slightly from version to version of MATLAB. By typing `help` you can access a listing of all functions in the version that you are using.

6. MISCELLANEOUS FEATURES

You may have discovered by now that MATLAB is case sensitive, that is "`a`" is not the same as "`A`."

The MATLAB display only shows 5 digits in the default mode. The fact is that MATLAB always keeps and computes in a double precision 16 decimal places and rounds the display to 4 digits. The command

```
format long
```

will switch to display all 16 digits and

```
format short
```

will return to the shorter display. It is also possible to toggle back and forth in the floating point format display with the commands

```
format short e
```

and

```
format  long  e
```

It is not always necessary for MATLAB to display the results of a command to the screen. If you do not want the matrix `A` displayed, type `A;` to suppress it. When MATLAB is ready to proceed, the prompt `>>` will appear. Try this on a matrix.

Occasionally you will have spent much time creating matrices in the course of your MATLAB session and you would like to use these same matrices in your next session. You can save these values in a file by typing

`save` *filename*

This creates a file *filename.mat* which contains the values of the variables from your session. If you do not want to save all variables there are two options. One is to clear the variables off with the command

```
clear a b c
```

which will remove the variables `a,b,c`. The other option is to use the command

`save` *filename* `x y z`

which will save the variables `x,y,z` in the file *filename.mat*. The variables can be reloaded in a future session by typing

`load` *filename*

When you are ready to print out the results of a session, you can store the results in a file and print the file from the operating system using the "print" command appropriate for your operating system. The file is created using the command

`diary` *filename*

Once a file name has been established you can toggle the diary with the commands

```
diary on
```

and

```
diary off
```

The command `diary` *filename*
will copy anything which goes to the screen (other than graphics) to the specified file, *filename.* Since this is an ordinary ASCII file, you can edit it later.

7. PROGRAMMING IN MATLAB

MATLAB is also a programming language. By creating a file with the extension *.m* you can easily write and run programs. If you were to create a program file *myfile.m* in the MATLAB language, then you can make the command *myfile* from MATLAB and it will run like any other MATLAB function. You do not need to compile the program since MATLAB is an interpretative (not compiled) language. Such a file is called an *m-file.* I am going to describe the basic programming constructions. While there are other constructions available, if you master these you will be able to write clear programs.

I. Assignment

Assignment is the method of giving a value to a variable. You have already seen this in the interactive mode. We write `x=a` to give the value of `a` to `x`. Here is a short program illustrating the use of assignment.

```
function r=mod(a,d)

% r=mod(a,d). If a and d are integers,
% then r is the integer remainder of a
% after division by d. If a and d are
% integer matrices, then r is the matrix
% of remainders after division by
% corresponding  entries. Compare with
% MATLAB's rem.

r=a-d.*floor(a./d);
```

You should make a file named *mod.m* and enter this program exactly as it is written. Now assign some integer values for `a` and `d`. Run

`mod(a,d)`

This should run just like any built-in MATLAB function. Type

`help mod`

This should produce the seven lines of comments which follow the `%` signs. The `%` signs generally indicate that what follows on that line is a comment which will be ignored when the program is being executed. MATLAB will print to the screen the comments which follow the "function" declaration at the top of the file when the `help` command is used. In this way you can contribute to the help facility provided by MATLAB to quickly determine the behavior of the function. Type

`type  mod`

This will list out the entire file for your perusal. What does this one line program mean? The first line is the "function declaration." A file with the function declaration at the top is called a *function file.* In it the name of the function (which is always the same as the name of the file without the extension *.m*), the input variables (in this case `a` and `d`), and the output variables (in this case `r`) are declared. Next come the "help comments" which we have already discussed. Finally, we come to the meat of the program. The variable `r` is being assigned the value `a-d.*floor(a./d);` The operations on the right hand side of the assignment have the meaning which you have just been practicing (the / is division) with the "`.`" meaning the entry-wise operation as opposed to a matrix operation. Finally, the "`;`" prevents printing the answer to the screen before the end of execution. You might try replacing the "`;`" with "`,`" and running the program again just to see the difference.

II. Branching

Branching is the construction

`if` *condition, program* `end`

The *condition* is a MATLAB function and *program* is a program segment. The entire construction executes the *program* just in case the value of *condition* is not 0. If that value is 0, the control moves on to the next program construction. You should keep in mind

that MATLAB regards `a==b` and `a<=b` as functions with values 0 or 1. Frequently, this construction is elaborated with

```
if  condition ,
    program1
else
    program2
end
```

In this case if *condition* is 0, then *program2* is executed. Another variation is

```
if condition1,
    program1
elseif condition2,
    program2
end
```

Now if *condition1* is not 0, then *program1* is executed. If *condition1* is 0 and if *condition2* is not 0, then *program2* is executed. Otherwise control is passed on to the next construction. Here is a short program to illustrate branching.

```
function b=even(n)

% b=even(n). If n is an even integer,
% then b=1, otherwise b=0.

if mod(n,2)==0,
   b=1;
   else b=0;
end
```

III. For Loops

A "for loop" is a construction of the form

```
for i=1:n,  program, end
```

Here we will repeat *program* once for each index value `i`. Here are some sample programs. The first is matrix addition.

```
function c=add(a,b)

% c=add(a,b). This is the function
% which adds the matrices a and b.
% It duplicates the MATLAB function
% a+b.

[m,n]=size(a);
[k,l]=size(b);
if m~=k | n~=l,
  error('matrices not the same size');
  return,
end
c=zeros(m,n);
for i=1:m,
   for j=1:n,
      c(i,j)=a(i,j)+b(i,j);
   end
end
```

The next program is matrix multiplication.

```
function c=mult(a,b)

% c=mult(a,b). This is the matrix product
% of the matrices a and b. It duplicates
% the MATLAB function c=a*b.

[m,n]=size(a);
[k,l]=size(b);
if n~=k,
   error('matrices are not compatible');
   return,
end,
c=zeros(m,l);
for i=1:m,
   for j=1:l,
      for p=1:n,
         c(i,j)=c(i,j)+a(i,p)*b(p,j);
      end
   end
end
```

For both of these programs you should notice the branch construction which follows the size statements. This is included as an error message. In the case of `add`, an error is made if we attempt to add matrices of different sizes, and in the case of `mult` it is an error to multiply if the matrix on the left does not have the same number of columns as the number of rows of the matrix on the right. Had these messages not been included and the error was made, MATLAB would have delivered another error message saying that the index exceeds the matrix dimensions. You will notice in the error message the use of single quotes. The words surrounded by the quotes will be treated as text and sent to the screen as the value of the variable `c`. Following the message is the command `return`, which is the directive to send the control back to the function which called `add` or return to the prompt. I usually only recommend using the `return` command in the context of an error message. Most MATLAB implementations have an error message function, either `errmsg` or `error`, which you might prefer to use. In the construction

`for i=1:n`, *program*, `end`

the index `i` may (in fact usually does) occur in some essential way inside *program*. MATLAB will allow you to put any vector in place of the vector `1:n` in this construction. Thus the construction

`for i=[2,4,5,6,10]`, *program*, `end`

is perfectly legitimate. In this case *program* will execute 5 times and the values for the variable `i` during execution are 2,4,5,6,10, in that order. The MATLAB developers went one step further. If you can put a vector in, why not put a matrix in? So, for example,

`for i=magic(7)`, *program*, `end`

is also legal. Now $program$ will execute 7 (=number of columns) times, and the values of `i` used in *program* will be successively the columns of `magic(7)`.

IV. While Loops

A "while loop" is a construction of the form

`while` *condition*, *program*, `end`

where *condition* is a MATLAB function, as with the branching construction. The program *program* will execute successively as long as the value of *condition* is not 0. While loops carry an implicit danger in that there is no guarantee in general that you will exit a while loop. Here is a sample program using a while loop.

```
function l=twolog(n)

% l=twolog(n). l is the floor of
% the base 2 logarithm of n.

l=0;
m=2;
while m<=n
   l=l+1;
   m=2*m;
end
```

V. Recursion

Recursion is a devious construction which allows a function to call itself. Here is a simple example of recursion

```
function y=twoexp(n)

% y=twoexp(n). This is a recursive program
% for computing y=2^n. The program halts
% only if n is a nonnegative integer.

if n==0,
   y=1;
else
   y=2*twoexp(n-1);
end
```

The program has a branching construction built in. Many recursive programs do. The condition `n==0` is the base of the recursion. This is the only way to get the program to stop calling itself. The "else" part is the recursion. Notice how the `twoexp(n-1)` occurs right there in the program which is defining `twoexp(n)`! The secret is that it is calling a lower value, `n-1`, and it will continue to do so until it gets down to `n=0`. A successful recursion is calling a lower value.

There are several dangers in using recursion. The first is that, like while loops, it is possible for the function to call itself forever and never return an answer. The second is that recursion can lead to redundant calculations which, though they may terminate, can be time consuming. The third danger is that while a recursive program is running it needs extra space to accomodate the overhead of the recursion. In numerical calculations on very large systems of equations memory space is frequently at a premium, and it should not be wasted on program overhead. With all of these bad possibilities why use recursion? It is not always bad; only in the hands of an inexperienced user. Recursive programs can be easier to write and read than nonrecursive programs. Some of the future projects illustrate good and poor uses of recursion.

VI. Miscellaneous Programming Items

It is possible to place a matrix valued function as the condition of a branching construction or a while loop. Thus the condition might be a matrix like `ones(2)` or `eye(2)`. How would a construction like

```
if condition,
    program1,
else
    program2,
end
```

behave if *condition* is `eye(2)`? If all of the entries of *condition* are nonzero, *program1* will execute. So if *condition* is `eye(2)`, *program2* will execute. If *condition* is `ones(2)`, *program1* will execute since all of the entries are nonzero. A problematic construction occurs when you have

`if A~=B`, *program*, `end`.

You would like *program* to execute if the matrices `A` and `B` differ on some entry. Under the convention, *program* will only execute when they differ on *all* entries. There are various ways around this. One is the construction

`if A==B, else` *program*, `end`

which will pass control to the "else" part if `A` and `B` differ on at least one entry. Another is to convert `A==B` into a binary valued function by using `all(all(A==B))`. The inside `all` creates a binary vector whose i^{th} entry is 1 only if the ith column of `A` is the same as the ith column of `B`. The outside `all` produces a 1 if all the entries of the vector are 1. Thus if `A` and `B` differ on at least one entry, then `all(all(A==B))=0`. The construction

`if ~all(all(A==B))`, *program*, `end`

then behaves in the desired way.

Essentially, the same convention holds for the while construction.

`while` *condition*, *program*, `end`.

The program *program* will execute successively as long as every entry in *condition* is not 0, and the control passes out of the loop when at least one entry of *condition* is 0. Another problem occurs when you have a conjunction of conditions, as in

`if` *condition1* & *condition2*, *program*, `end`

Of course, *program* will execute if both *condition1* and *condition2* are not zero. Suppose that *condition1* is 0 and *condition2* causes an error message. This might happen for `i<=m & A(i,j)==0` where `m` is the number of columns of `A`. If `i>m`, then you would like to pass the control, but since `A(i,j)` makes no sense if `i>m` an error message will be dished up. Here you can nest the conditions.

```
if i<=m,
    if A(i,j)==0,
        program
    end
end
```

VII. Scripts

We have seen m-files which have the function declaration at the top. In practice these files create new MATLAB functions. In creating a function which we will call `fun(a)` we might use a variable like `x`. Now suppose the the variable `x` has a value in your session. What happens to the value of `x` after you make a call to `fun(a)`? Nothing. The only way to change the value of `x` when running `fun` is to assign `x=fun(a)`. The `x` inside the program `fun.m` behaves independently from the variable `x` in your session. This makes function files very natural to use. You do not need to worry about whether you have used a variable inside of a program which already has a value in your session.

A ***script*** is an m-file without the function declaration at the top. A script treats variables differently than a function file. In a script, if `x` appears in a program, which we will call `scrpt`, and `x` has a value in your session, then a call to `scrpt` might change the value of `x`. If you do not make that function declaration, then the variables in your session can be altered. Sometimes this is useful, but I recommend that you use function files.

VIII. Clearing

In the interactive portion of the tutorial you saw how to use the `clear` function to remove a variable from your session. It is also possible to use `clear` to remove a function from your session. When a function is "read" by MATLAB into the machine it resides in the machine's memory. If you find that you are short of memory it is possible to clear the function with the command `clear functions`. Occasionally, when you are in the edit-run cycle, the machine will not acknowledge changes which have been made in the function you are editing. When you suspect this is the case try clearing the function before running again.

IX. Suggestions

These are a few pointers about programming and programming in MATLAB in particular.

1) You should use the indented style that you have seen in the above programs. It makes the programs easier to read, the program syntax is easier to check, and it encourages you to think in terms of building your programs in blocks.

2) Put lots of comments in your program to tell the reader in plain English what is going on. Some day that reader will be you, and you will wonder what you did.

3) Put error messages in your programs like the ones above. As you go through this manual, your programs will build on each other. Error messages will help you debug future programs.

4) Always structure your output as if it will be the input of another function. For example, if your program has "yes-no" type output, do not have it return the words "yes" and "no," rather return 1 or 0, so that it can be used as a condition for a branch or while loop construction in the future.

5) In MATLAB, try to avoid loops in your programs. MATLAB is optimized to run the built-in functions. For a comparison, see how much faster `A*B` is over `mult(A,B)`. You will be amazed at how much economy can be achieved with MATLAB functions.

6) If you are having trouble writing a program, try to get a small part of it running and build on that.

A Short List of Key Words and Symbols

This is a starter list of words and symbols for MATLAB. For the complete list as well as any m-files which are included in your implementation of MATLAB type `help`. More information is in the *MATLAB User's Guide* by The MathWorks, Inc. and published by Prentice-Hall, Inc. There is a on page matlab-index for finding functions used in this book. The *MATLAB Primer* by Kermit Sigmon published by CRC Press, Inc. is also a useful reference guide to MATLAB.

magic	round	help	function	==
eye	floor	format short	if	~=
diag	abs	format long	else	<
ones	sqrt	who	elseif	<=
zeros	sin	clear	end	>
rand	cos	exit	while	>=
hilb	tan	ans	return	&
tril	asin	load	=	|
triu	acos	save	;	~
sum	atan	diary	:	any
prod	exp	type	%	all
max	log	dir	,	+
min	eps			-
size	pi			*
length	sign			.*
				'
				/
				./
				^
				.^