Introduction to MATLAB for Economists

Seth Neumuller

Wellesley College

This module is designed to introduce the advanced undergraduate or first year graduate student in economics to some of the basic features of MATLAB. While no prior programming experience is required, familiarity with matrix algebra and some basic statistics will be helpful. At the end of this module, the student will be asked to apply the skills that they have learned to a few numerical examples. MATLAB is available free of charge to Wellesley students. More information and instructions for installing MATLAB on your computer can be found here.

1 Getting Started

MATLAB is a software package that works on several different levels to aid in the analysis of mathematical problems. MATLAB got its name from a shortening of 'Matrix Laboratory' and it follows that matrices, more generally referred to as arrays, are the most natural and useful data type in MATLAB. Not only is framing problems in terms of arrays often convenient, MATLAB is also most efficient when performing computations using arrays. At its most basic level, MATLAB can be used like a calculator that specializes in matrix operations. MATLAB also allows the user to store a list of commands in a file which can then be executed at anytime. To solve more complex problems, MATLAB may be used like a programming language where functions are defined that accept a set of arguments and return output. Finally, the most recent versions of MATLAB support symbolic mathematics.

The *Command Window* functions like a calculator with the +, -, /, and * operators performing their usual functions. The usual order of operations is respected as are parentheses. Input is entered after the command prompt which is the >> sign. MATLAB allows values to be stored in variables and can then perform operations on those variables. The = sign is used to assign values on the RHS to variables on the LHS. To get a feel for this, define a few variables and perform some arithmetic operations on them as in the example below where a student computes his weekly hours spent studying.

```
>> numclass=3
numclass =
3
>> hoursclass=2
hoursclass =
2
>> hoursday=hoursclass*numclass
hoursday =
6
>> hoursweek=7*hoursclass*numclass
hoursweek =
42
```

The above example illustrates that named variables may be operated on together with scalar numerical inputs and that the result of a computation may be stored in a new variable. To view the value stored in a variable simply enter the name of the variable at the command prompt. It is important to realize that MATLAB performs computations using the value of the variable at the time the computation is processed. For instance, hoursclass was used to compute the value of hoursweek, but if the value of hoursclass is later changed, say hoursclass=1000, this will have no effect on the value of hoursweek. The point is that the variables above each store a scalar value and contain no information about the formula used in their definition.

There are a few important points to keep in mind when naming variables. Valid variable names must start with a letter, contain all alphanumeric characters and must not conflict with one of MATLAB's keywords or built-in functions. Keywords are words MATLAB reserves for special purposes and a list of these words is available by entering the command **iskeyword**. MATLAB is case sensitive meaning that commands and variable names that are spelled the same but with different capitalization are treated as different entities.

Once we have assigned a value to a variable, MATLAB remembers the name of the variable and the value assigned. Variables created in the *Command Window* are said to exist in the MATLAB workspace. Where the variables live may seem unimportant now, but it will become important when we discuss the scope of variables defined in functions. There are two important commands for managing variables in the workspace. The **who** command prints a list of the variables currently existing in the workspace, and the **clear** command clears the existing variables. Type **who** to see that your variables are indeed stored and then **clear** to start with a clean workspace. Check that the **clear** command worked by typing **who** again.

```
>> who
Your variables are:
```

```
hoursclass hoursday hoursweek numclass
>> clear
>> who
>>
```

One nice advantage of MATLAB over traditional programming languages is that it comes with a built in set of functions for performing basic mathematical operations. All built in functions contain helpful text that is displayed by entering **help** followed by the name of the function. The displayed text generally includes the values that must be passed to the function, called arguments, and a description of the values that are returned. For the time being we are interested in functions that take a single scalar argument and return a single scalar. Some examples are trigonometric functions like **sin**, **cos**, and **tan**, the absolute value function **abs**, and the square root function **sqrt**. Use the **help** command to see details about some of these functions. Spend some time playing around with these functions as in the example below.

```
>> abs(-9)-10
ans =
-1
>> abs(-9-10)
ans =
19
>> x=-.5
x = -0.5000
>> sin(x)
ans =
-0.4794
>> abs(sin(x))
ans =
0.4794
>> myvar=2;
>> sqrt(myvar)
ans =
1.4142
```

Notice that if the value of a computation is not assigned to a variable, MATLAB automatically assigns it to a variable named ans. For this reason, naming a variable ans should be avoided as it may be accidentally overwritten. Also notice that placing a semicolon **;** after a command suppresses printing to the command window.

2 Vector Algebra

Having mastered scalar algebra it is now straightforward to consider vectors. In fact, MATLAB does not have a separate 'scalar' data type and thus the discussion of scalar mathematics above may be seen in a more general framework as an exposition of the mathematics of 1-dimensional vectors. To manually define a row vector, we use the same approach as before, except now we must enclose the values in square brackets [ ] with spaces between values. The following examples involve a vector of integer values, but the elements of vectors may take on any value.

```
>> x=[8 3 9 0 5]
x =
8 3 9 0 5
```

Note that the default in MATLAB is to create a row vector. To define a column vector instead, simply use the transpose operator ' as follows.

```
>> x=[8 3 9 0 5]'
x =
8
3
9
0
5
```

The variable x contains five elements which we may access individually using index addresses. You can think of a vector of length n as a set of slots labeled 1,2,...,n with each numbered slot containing a value. The function **length** can be used to verify the number of elements in a vector.

```
>> length(x)
ans =
5
```

Note that MATLAB begins numbering at 1 in contrast to many other programming languages that begin numbering at 0. To access an element of x the variable name is followed by parentheses () containing the index, or in our terminology the slot number, of the desired element. The following command stores the sum of the second and third element of x in a new variable called sumval.

```
>> sumval=x(2)+x(3)
sumval =
```

3 Matrix Algebra

In discussing MATLAB, the terms array and matrix are used loosely and often interchangeably. Manually defining a matrix is much like defining a vector, but we now require a new character, the semicolon **;** , to denote the end of a row.

```
>> y=[1 2 3; 4 5 6; 7 8 9]
y =
1 2 3
4 5 6
7 8 9
```

Index addressing for matrices also follows as the immediate generalization of vector addressing. As in standard notation, the first index gives the row and the second the column. Additionally, matrices will accept single scalar indices but now the index, call it i, refers to the i^{th} element of the vector formed by stacking the columns from first to last. The following example illustrates both methods of addressing the matrix y.

```
>> y(2,3)
ans =
6
>> y(8)
ans =
6
```

We can also use indices together with the assignment operator = to change the value of one or more elements in the matrix.

```
>> y(2,3)=0
y =
1 2 3
4 5 0
7 8 9
```

The function **size** returns a two element vector with the first element being the number of rows and the second the number of columns, which is consistent with standard linear algebra notation.

```
>> size(y)
ans =
3 3
```

To index a column, specify the column number and use the colon sign **:** in place of a row number, indicating we want the elements of every row in the specified column. Furthermore, we can access multiple columns by using the syntax first:last in the column index where first is the first column and last is the last column. The analogous syntax is appropriate for accessing rows.

```
>> y(:,1)
ans =
1
4
7
>> y(1:2,:)
ans =
1 2 3
4 5 0
```

Finally, the syntax [A B] or [A; B] can be used to form a compound matrix from the matrices A and B. Here, the space means to add B as a new set of columns, whereas the semicolon **;** means to add B as a new set of rows. The appropriate dimensions must match for the concatenation to be valid, otherwise MATLAB will return an error.

So far, we have created vectors and matrices by manually typing in values, a technique that is cumbersome, error prone, and often impractical in more complicated numerical applications. Luckily, MATLAB provides a number of functions to automatically create certain commonly encountered vectors and matrices. to create a vector x whose values are equally spaced at step size s and go from a to b, use the command $x = a{:}s{:}b$. If s is omitted, MATLAB assumes a step size of one. Note that if $b \neq a + sK$ for some integer K, then the last value in x will be $a + s\overline{K}$, where $\overline{K}$ is the greatest integer such that $a + s\overline{K} < b$.

```
>> x=1:.1:1.5
x =
1.0000 1.1000 1.2000 1.3000 1.4000 1.5000
```

Additionally, the functions **linspace** and **logspace** are available to create vectors with linearly or logarithmically spaced elements. For instance x =linspace(1,1.5,6) will create a vector of 6 linearly spaced points with endpoints 1 and 1.5. That is, it will create the same x as we created above using the colon operator **:**.

There are also a set of functions available to create specially structured matrices of arbitrary dimension. A typical example is the function **zeros(m,n)** which returns a matrix of size mxn consisting of all zeros.

```
>> zeros(3,4)
ans =
0 0 0 0
0 0 0 0
0 0 0 0
```

Other matrix creation functions such as **ones**, **zeros**, **eye**, **diag**, and **repmat** work similarly. In general, if only one dimension argument is given, a square matrix of that dimension in both rows and columns is created. Try out a few of these functions to get comfortable with creating standardized matrices. Remember that you may use the **help** command at any time to learn about predefined functions in MATLAB.

Often it is necessary to perform the same scalar transformation on every element of a matrix. For example, to add a scalar s to every element of a matrix (or vector) A, the command is either $A + s$ or $s + A$. Other arithmetic operators work similarly, but the exponent operator $\wedge$ performs matrix powers in the linear algebra sense. To simply raise each element to a power we must make use of the dot operator **.** which, when placed in front of the exponent operator, tells MATLAB to raise each element to the desired power rather than the entire matrix. The example below illustrates both scalar matrix operations and the use of the dot operator to raise each element to a power.

```
>> A=ones(4);
>> A*3+2
ans =
5 5 5 5
5 5 5 5
5 5 5 5
5 5 5 5
>> A∧3
ans =
16 16 16 16
16 16 16 16
16 16 16 16
16 16 16 16
>> A.∧3
ans =
1 1 1 1
```

```
1 1 1 1
1 1 1 1
1 1 1 1
```

A key point in MATLAB syntax is that the usual mathematical operators take on their linear algebra meaning. Thus, when ***** is applied to two matrices, the result is matrix multiplication rather than element by element multiplication. We have already seen how this distinction is important when applying exponents to a matrix. To indicate that element by element arithmetic is desired, always use the dot operator **.** in addition to the standard mathematical operator. The following illustrates the difference for the two by two matrix case.

```
>> A = [1 1; 1 1];
>> B = [2 2; 2 2];
>> A*B
ans =
4 4
4 4
>> A.*B
ans =
2 2
2 2
```

Matrix inverses are given by the function **inv**, and an error message will appear if the matrix is (numerically) singular. It is also possible to use the right division operator **/** for which if A and B are matrices A/B returns essentially $A*B^{-1}$, although computed in a different way than the inverse function. To read more about the details of matrix inversion, get help for the function **mrdivide**.

Matrix operations are only defined on matrices of appropriate dimensions. That is, addition and summation require matrices of the same dimension, while multiplication and division require that the inner dimensions match. Define some matrices using the automatic matrix creation functions and perform some computations.

Earlier we considered the action of built in functions when passed scalar arguments. That discussion immediately generalizes to the vector or matrix case. When a function like **sin** is passed a matrix A it returns a matrix B of the same dimension with the elements of B having the value of the function evaluated at the corresponding element of A. In particular, the most efficient way to evaluate a function at a set of points is to first define a vector that acts as a grid and then to pass that vector to the desired function. The returned output will be a vector made up of the function evaluated at each grid point. Recall from above that generating a linearly spaced grid is automated by MATLAB through the use of the colon **:** operator. The following example evaluates the sine function at 9 points evenly spaced from 0 to 4π. Note that

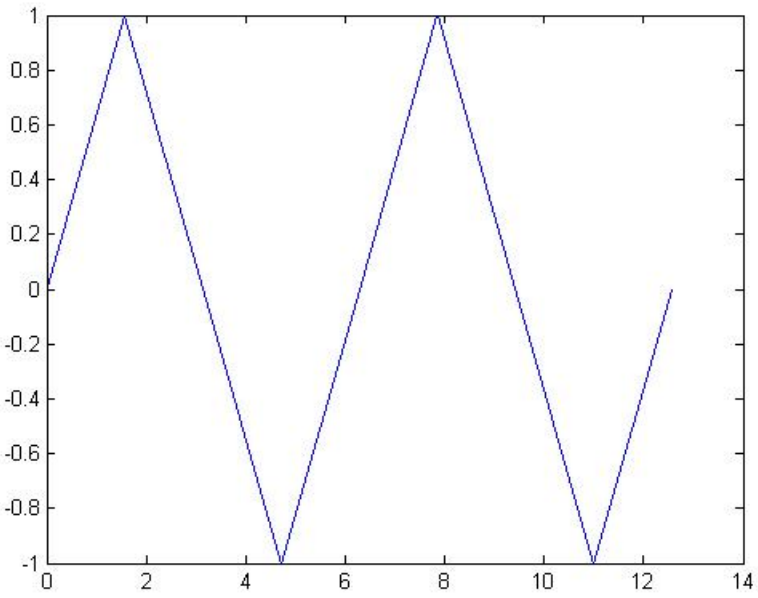
certain variables like **pi** are automatically assigned standard values by MATLAB.

```
>> x=0:4*pi/8:4*pi
x =
0 1.5708 3.1416 4.7124 6.2832 7.8540 9.4248 10.9956 12.5664
>> sin(x)
ans =
0 1.0000 0.0000 -1.0000 -0.0000 1.0000 0.0000 -1.0000 -0.0000
```

4 Plots and Histograms

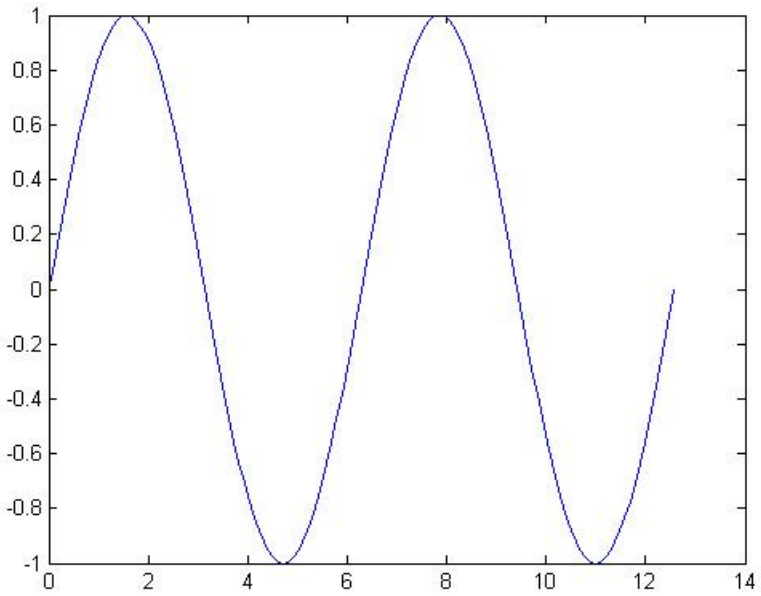
MATLAB also offers a wide variety of graphical tools, of which the most frequently used are plots and histograms. The function **plot(y)** plots the values of the vector y versus their index number. Alternatively, suppose we want to plot the values of the sine function above versus the corresponding values of x. Then we would proceed as follows:

```
>> y=sin(x);
>> plot(x,y)
```



With only 9 grid points, this plot is a rather poor approximation of the sine function. To increase the precision of the plot, simply increase the number of grid points, making the grid finer.

```
>> x=0:4*pi/128:4*pi;
>> y=sin(x);
>> plot(x,y)
```

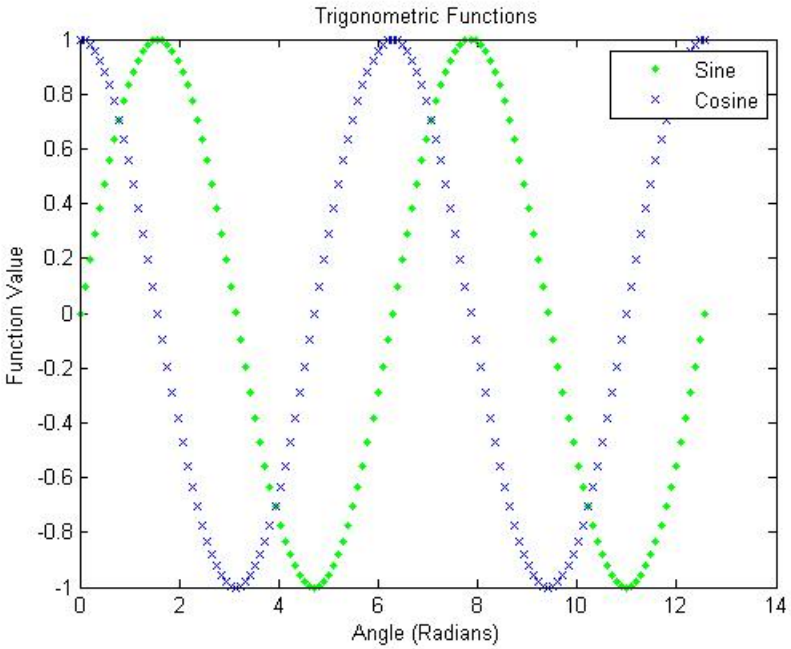


You may have noticed that this new plot overwrote the old one in the original *Figure* window. To avoid this, new *Figure* windows can be created with the command **figure**. Additionally, **figure(s)** makes active the s^{th} *Figure* window with windows being numbered starting at one in the order they are created. These window management considerations will become important when working with codes that create multiple figures.

Following the two arguments specifying the x and y values, the plot function accepts an additional argument that specifies the color, symbol, and line type to be used for the plot. See the help documentation associated with the **plot** function for more information. In addition, MATLAB provides a set of functions to make labeling plots easy. These include **title**, **legend**, **xlabel**, and **ylabel**. Finally, to add additional data series to a figure, you can either use the **hold** command to keep the current figure active or use the same **plot** function with multiple arguments. See the example below for how to create a well labeled plot with multiple data series.

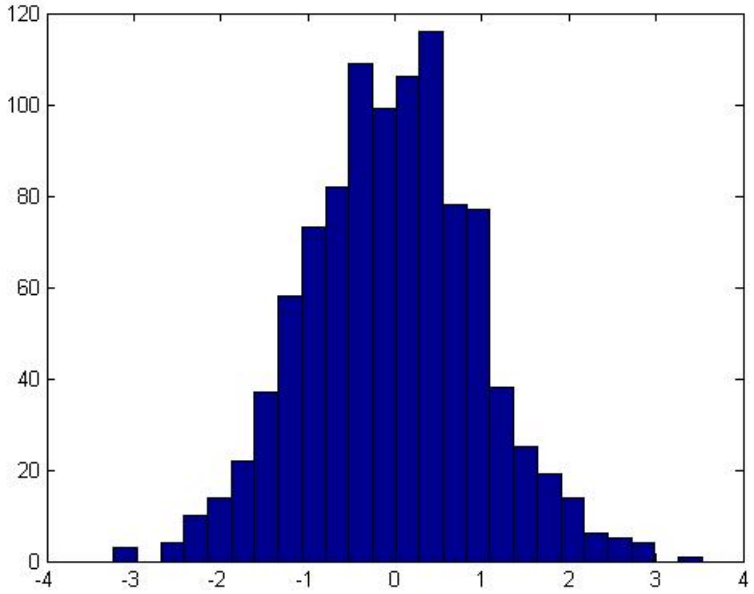
```
>> plot(x,sin(x),'g.',x,cos(x),'bx')
>> title('Trigonometric Functions')
>> xlabel('Angle (Radians)')
>> ylabel('Function Value')
```

>> legend('Sine','Cosine')



A natural way for graphically viewing probability distributions, or more generally for plotting certain types of data, is through the use of histograms. A histogram of draws from a distribution graphs the number of draws that fall in a given range. In the remainder, each range will be referred to as a 'bin.' The function **hist** counts the number of observations that fall into each bin and returns two vectors, one giving the set of bins and the other the number of observations in each bin. In particular **[num bin]=hist(y,x)** sorts the data in y into bins with centers given by the vector x and returns the centers in 'bin' and the numbers in each bin in 'num.' If x is a scalar it specifies the number of bins. To best illustrate this feature, we will first clear the workspace and then use the **randn** function to generate a column vector of 1,000 normally distributed random draws from the standard normal distribution (mean zero and variance one). Finally, we will create a histogram of the random draws with 25 bins.

```
>> clear
>> x=randn(1000,1);
>> hist(x,25)
```



Sometimes it may be more appropriate to display a smooth representation of the distribution as opposed to a jagged histogram. The function **ksdensity** is extremely useful for this application. See the associated help documentation for more information.

5 M-Files

So far, we have used MATLAB as a sophisticated calculator with advanced graphing and linear algebra capabilities. While this aspect of MATLAB is useful, you will undoubtedly encounter analysis that is too complex to be implemented with a calculator. Examples of this include computing the equilibrium of a model, complex econometric analysis, or any task that may be need to be carried out more than once and can be easily automated. In this section, we take a first step toward using MATLAB as a programmable mathematical environment.

MATLAB allows users to store blocks of commands as scripts called M-Files that can be saved and then modified or run at any time. To create a new M-File, select *File→New→Blank M-File*. Within a script M-File, any commands that are valid in the *Command Window* may be typed. You may save an M-File in any convenient location as *filename.m* and then execute it by either typing the name of the M-File (without the *.m* extension) in the *Command Window* or by simply selecting the green arrow in the *Editor* window.

To make your code more readable to yourself and others, MATLAB provides for comments to be included in script M-Files. Comments are just that, descriptions

of the code that are ignored by MATLAB when the M-File is executed. To enter comments, simply start with a percent sign % followed by text. It is a very good idea to include comments explaining each section of code. In general, the more comments the better!

We have seen that entering a variable name accesses a stored variable at the command prompt whereas typing the name of an M-File executes the M-File. The natural question is then how MATLAB determines whether to look for a stored variable or M-File. The answer is that it begins by looking for a variable of the name entered, if none is found it then looks for an M-File matching the entered name. To find an M-File, MATLAB prioritizes the directories it looks through according to a stored 'search path' and executes the first M-File of the correct name it finds. To modify the search path select *File→Set Path* which opens a dialog box that allows you to change the order in which folders are prioritized and allows for the addition of new folders. Given the way MATLAB searches for M-Files, it is not a good idea to use variable names that are the same as M-Files that you want to make use of.

6 Logical Operations

The following section begins by surveying the set of logical and relational functions that return 1 or 0, corresponding to true or false, respectively. Once these operations are understood, it is possible to create control structures like loops and if-else statements that rely on the evaluation of logical functions.

Relational operators return a 1 or 0 based on comparing two arrays. Comparisons that MATLAB supports may be made through the use of the following operators

<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equal to
~=	Not equal to

When two matrices of the same dimension are compared the result is a matrix of the same dimension of zeros and ones representing the results of comparing corresponding elements. Also, a matrix may be compared with a scalar returning a matrix of zeros and ones of the same dimension as the matrix representing the results of comparing each element with the scalar. This is illustrated in the following example.

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

```
ans =
1 1 1
1 1 0
0 0 0
>> A<B
ans =
1 1 1
0 0 0
0 0 0
```

Note that since these relational operators perform evaluations based on stored numerical values the inherent imprecision of machines can become problematic. For this reason it is often useful to check if the difference of two values is smaller in magnitude than a certain threshold, rather than that they are numerically equivalent. We will examine this strategy at a later point in this module.

The following example showcases one useful application of relational operators. Suppose we have a matrix representing a data set and we want to censor all data points below a certain value, say 5, and replace them with zeros. This is easily accomplished using the nicely structured true-false array that MATLAB relational operators return.

```
>> A=[10 1 3; 6 4 13; 2 20 8];
>> A=A-A.*(A<5)
A =
10 0 0
6 0 13
0 20 8
```

Logical operators act on logical values (true or false) and are a useful and powerful way to combine or act on information obtained from relational operators. MATLAB logical operators are tabulated below.

&	Element by element AND
|	Element by element OR
∼	NOT
&&	Scalar AND
||	Scalar OR

These operators can be thought of as acting on either one or two logical values with NOT returning true when the input is false, OR returning true when one or both are true, and AND returning true only if both are true. Like relational operators,

these operators return a matrix of the same dimension as the dimension of the matrix or matrices being processed. The following example illustrates the use of these logical operators in conjunction with relational operators.

```
>> x=1:5;
>> x<=2
ans =
1 1 0 0 0
>> ~(x<=2)
ans =
0 0 1 1 1
>> (x>=2)&(x<=3)
ans =
0 1 1 0 0
```

7 Loops

There are two kinds of loops in MATLAB, **for** loops and **while** loops. Loops allow a block of code to be executed repeatedly while certain conditions hold. A **while** loop must adhere to the following syntax

```
while "expression"
  "code"
end
```

This tells MATLAB to continue executing the commands in "code" as long as "expression" has all nonzero elements. Each time before "code" is executed MATLAB checks to make sure "expression" is still all nonzero, and if it is it executes "code" again. If "expression" contains a zero element, MATLAB does not execute "code" and moves on to the command immediately following end. Usually "expression" will be an expression involving a relational or logical operator since we usually want "code" to execute while a certain logical condition holds. It is important to initialize "expression" to have all nonzero elements if we want the while loop to iterate at least once. To create a vector with elements from 1 to 10, without using the colon operator, the following code is appropriate.

```
index=1;
while index<=10
  x(index)=index;
  index=index+1;
```

```
end
```

We can then check the resulting output in the *Command Window*.

```
>> x
x =
1 2 3 4 5 6 7 8 9 10
```

A **for** loop, on the other hand, must follow the syntax

```
for count=a:s:b
  "code"
end
```

MATLAB will execute the loop $\lfloor (b-a)/s \rfloor + 1$ times (where $\lfloor . \rfloor$ means round down to the nearest integer). The variable "count" will have the value a on the first loop execution and the value $a + Ks$ on the K^{th} iteration. Clearly a **while** loop can do anything a **for** loop can, but a **for** loop is often more convenient. Moreover, MATLAB in some cases can execute **for** loops more efficiently than **while** loops.

A discussion of loops would not be complete without mentioning how to escape from an infinite loop. Such loops can be accidentally created by starting a loop that never will generate a zero element in "expression." To escape such a loop, or in general to stop MATLAB from executing any further commands, press Ctrl-c.

8 if-else Statements

if statements work similarly to the **while** loop, but only execute once. The most simple **if** statement uses the syntax

```
if "expression"
  "code"
end
```

If "expression" contains all nonzero elements, then "code" is executed, otherwise MATLAB moves to end and begins executing the subsequent commands. **if** statements allow a block of code to be executed in the alternative that "expression" does in fact contain zeros. This is achieved using the **else** command as follows.

```
if "expression"
  "code"
else
```

```
  "othercode"
end
```

Finally, multiple alternatives are considered by using the **elseif** command.

```
if "expression"
  "code"
elseif "expression1"
  "code1"
elseif "expression2"
  "code2"
...
elseif "expressionN"
  "codeN"
else
  "othercode"
end
```

MATLAB considers each alternative in order and once it finds a true expression evaluates the subsequent block of code and then moves to the end. For example, if "expression" and "expression1" are false but "expression2" is true, then "code2" is executed. If "expression5" is also true, it is of no importance as "code5" will not be executed since after executing "code2" the **if** statement is terminated. If none of the numbered expressions are true, then "othercode" will be executed.

If statements are particularly useful for deciding when to exit a **for** loop. For example, suppose we are writing a code that manually computes the root of the function $f(x)$ for various values of x. Given that MATLAB performs evaluations based on numerical values, it is unlikely that MATLAB will find a value of x such that $f(x) = 0$ exactly. In most situations we only need to know the desired root to a few significant digits, and any additional precision is unnecessary. In this case, we would like to exit the **for** loop as soon as we find a value of x such that $f(x)$ is 'close' to zero. The command **break** is used to exit a **for** loop and can be implemented within a **for** loop as follows.

```
if abs(f(x)) < 10^(-3)
  break
end
```

where $f(x)$ is the value of the function evaluated at the current value of x.

9 Functions

Not only does MATLAB come with a large number of useful built in functions, but MATLAB allows the user to define new functions. This is where the true power of MATLAB lies. So far, we have used M-Files to store lists of commands that are executed in order. Functions also use the M-File structure, but with the important difference that they can both accept input arguments and return output. All functions must begin with a function header, the syntax for which is

function output = functionname(arguments)

The word "function" identifies this as a function M-file rather than a script and must be present.[1] The function can have any name subject to the same rules as variable names, and the M-File should be saved with the same name as the function (functionname.m, for example). Functions can take arbitrarily many arguments and return arbitrarily many output variables. While comments are not required, it is good practice to include both the function call and a brief description of the function directly following the header. These comments are what MATLAB displays when the help command is used followed by the function name. The function should assign a variable named "output" and it is this value that will be returned by the function.

Functions act like black boxes in that given an input they provide an output with no indication of the steps taken to arrive at the new value. Furthermore, each function only has access to the variables declared in it, or passed to it through arguments, and cannot interact directly with variables defined in the MATLAB workspace. Note the contrast here with script M-Files, where every variable defined in a script is also available at the command prompt and visa versa. Though these scope of variable considerations may at times be frustrating, they impose a valuable structure on code and help promote efficient compartmentalization.

To quickly define a mathematical function MATLAB allows the use of anonymous functions. Anonymous functions come in very handy when using the built in optimization routines in MATLAB. An anonymous function can be defined as follows.

fun =@(arg) "expression involving arg"

Where "arg" defines the arguments that will be passed to the function "fun". For instance, suppose we wanted to create a function that takes one argument, x, and returns the square of the sine function when x is nonnegative and zero otherwise.

```
>> testfunc =@(x) (x>=0)*sin(x)^2;
>> testfunc(pi/2)
```

[1]Note that while "output", "functionname", and "arguments" are all arbitrary names chosen by the user, "function" is a keyword that must begin all function headers.

```
ans =
1
>> testfunc(-pi/2)
ans =
0
```

10 Univariate Optimization

MATLAB provides robust and somewhat customizable built in optimization routines. Although an algorithm for finding the zeros of a function may at first seem only useful in certain special cases, it is actually a very general tool. Whenever we want to solve $y = g(x)$, we may equivalently consider solving for the zeros of $f(x)$, where $f(x) = y - g(x)$. In this sense, zero finding is a general way of solving an equation. The built in function for zero finding of a scalar valued function is **fzero**. The function call **fzero(@fun, x0)** returns a zero of the function "fun." "x0" is the value at which MATLAB begins looking for a zero, and depending on how well behaved "fun" is, the zero finding routine may be quite sensitive to this choice.

Minimization, like zero finding, is more general than it sounds since maximizing a function is the same as minimizing the negative of a function. MATLAB offers specific routines depending on the linearity of the functions and constraints involved. MATLAB also allows the user to supply gradients to help the minimization procedure. Here we discuss the unconstrained minimization of a univariate function. The built in function for general nonlinear unconstrained minimization is **fminunc**. The function call **fminunc(fun, x0)** attempts to find a minimum of "fun" starting at "x0" and returns the minimizing argument value if successful. Again, the choice of "x0" can in some cases be critical. Suppose that we want to maximize rather than minimize a function "fun." We can first create an anonymous function that is the negative of "fun", and then ask MATLAB to minimize this new function using **fminunc** which is equivalent to solving for the maximum of our original function.

```
>>negfun=@(x) -fun(x);
>>xmax = fminunc(negfun,x0);
```

Problem 1

Ordinary least squares (OLS) is one of the oldest and simplest tools available to the econometrician. In the linear model $Y = X\beta + \epsilon$*, OLS provides the following estimate for the slope coefficients* $\hat{\beta} = (X'X)^{-1}X'Y$*. In this exercise, consider the stochastic single equation model* $y = 10x + 5 + \epsilon$*, in which* $\epsilon \sim N(0, 1)$*. Note that this equation can equivalently be written in matrix form as follows:*

$$y = \begin{bmatrix} 1 & x \end{bmatrix} \begin{bmatrix} 5 \\ 10 \end{bmatrix} + \epsilon.$$

To be consistent with our formula for the OLS estimator, x *and* y *are column vectors of identical length containing observations generated by the above process, 1 is a column vector of ones the same length as* x *and* y*, and* ϵ *is column vector of residuals the same length as* x *and* y*. Take care in the computations to follow these conventions.*

Create an M-File that starts by generating 100 artificial observations of this process. To do this, let x *take on the values 0.01, 0.02,...,1.00, draw corresponding values for* ϵ *at random from the standard normal distribution, and compute the implied vector* y*. You should now have a column vector containing values of* x*, a column vector containing values of* y*, and additionally should create a column vector consisting of 100 ones. Concatenate the vector of ones with the vector* x *to form the matrix* X *referred to in the single equation model above. Now compute the OLS estimator* $\hat{\beta}$ *using the formula listed above above. The result should be a 2-vector with the first element corresponding to the intercept parameter and the second element corresponding to the slope.*

Plot the artificial observations along with the OLS regression line $y = X\hat{\beta}$ *in a single figure with an appropriate title, a legend, and axes labels. Finally, compute the 100 OLS residuals defined as* $e = y - X\hat{\beta}$ *and display them in a well labeled histogram. Remember to add comments to your M-File as needed so that someone other than yourself can understand the steps that you are undertaking.*

Problem 2

Consider a competitive firm that produces output using a Cobb-Douglas technology. The firm uses as inputs capital, K, and labor, L, and takes their market prices r and w, respectively, as given. The profit function, assuming the relative price of the firm's output is 1, is then

$$\Pi(K, L) = K^{\theta}(AL)^{1-\theta} - rK - wL.$$

where θ and A are known constants. The first order condition for maximizing the firm's profit with respect to K implies that the firm should choose K such that its marginal product of capital is equal to its price, or

$$\theta K^{\theta-1}(AL)^{1-\theta} = r.$$

(a) *Suppose that labor $L = 1$ is supplied inelastically by consumers, and that $A = 10$, $\theta = 1/3$, $r = 1$, and $w = 2$. Create a function that takes K as an argument and returns the difference between the marginal product of capital and its price.*

(b) *In a separate M-File, write a script that plots the output of the function you created in Part (a) for various values of K, along with the corresponding value of* Π.

(c) In the same M-File that you created in Part (b), compute the profit maximizing value of K in three different ways:

1. *Using fzero to find the zero of the first order condition*
2. *By defining an anonymous function that returns the negative of the profit for any value of K and then using fminunc to find the minimum of this new function*
3. *Computing the zero of the first order condition using the bisection method. First guess upper and lower bounds such that the optimum value of K is likely to be contained in the interval defined by these bounds. Starting from the midpoint of this interval, evaluate the difference between the marginal product of capital and the rental rate of capital. If positive, then set the lower bound equal to the current value of K. If negative, then set the upper bound equal to the current value of K. Compute the midpoint of this new interval and evaluate the difference between the marginal product of capital and the rental rate of capital at this new value of K. Adjust the bounds accordingly. Repeat this algorithm until the absolute value of the difference between the marginal product of capital and the rental rate of capital is within some reasonable tolerance, say* 10^{-3}.