MATLAB Workshop – Walkthrough

TiSEM, Tilburg University

September 2020

This file contains a series of sections that introduce various aspects of MATLAB from basic program syntax to advanced data analysis and modelling using matrix algebra or optimisation. The topics are slightly tailored towards the practitioners of econometrics than towards practitioners of operations research. The topics assume basic knowledge of regression analysis and matrix algebra. This file is prepared by Tunga Kantarcı. Questions, corrections or comments can be sent to kantarci@tilburguniversity.edu.

1. Program Window

The program window consists of four panels. 'Command Window' is where you type command syntax. Type, e.g., `a = 1` in the command prompt, which creates a scalar named `a` that takes a value of `1`. Type `a` in the command prompt, which will display the result in the Command Window.

MATLAB has added the scalar you just created to the 'Workspace'. Workspace shows the variables you create and held in memory during a MATLAB session. It is currently displaying the scalar variable `a`.

Type `edit untitled` in the command prompt. A dialog box will appear asking if you want to create the file. Select yes. This will open the 'Editor'. Editor is a simple text editor that allows to type and keep program syntax.

'Current Folder' is where you can browse, delete, or rename MATLAB files. E.g., click on the 'Browse for folder' button located just above Current Folder. Then, search for the folder where you have saved the MATLAB program file `MWScript.m`. Click 'Open'. Note that you are opening a folder, not a file. Therefore, do not search for `MWScript.m`, but for the folder that contains `MWScript.m`. After you click 'Open', `MWScript.m` will appear in the Current Folder. Double click on it, to open it in the Editor. We will learn about `MWScript.m` in a future section.

```
a = 1;
edit untitled;
```

2. Interacting with MATLAB

You can interact with MATLAB in three ways. In the first way, you use the program menu, e.g., to create MATLAB program files, open MATLAB data files, create plots, etc. What you can do through the program menu is very limited, however.

In the second way, you type commands in the command prompt, in the Command Window. For this you need to know the command syntax. In this workshop you will learn some standard command syntax. You can check the MATLAB documentation for special command syntax. You will learn about the MATLAB documentation in a future section.

MATLAB keeps a history of the commands you have typed in the current or a previous session in the 'Command History'. To view the Command History, activate your cursor in the Command Window, and press the up-arrow key on your keyboard. Using the up- and down-arrow keys, you can browse the command history, and if you press the right-arrow key while you are on a command, MATLAB will paste that command to the command prompt. In this way you can adjust a previously executed command without retyping it all over again.

In the third way, you interact through a 'script' file which is a simple text editor where you can type and execute commands.

How do we execute a command contained in a script file? Go to the Editor, and have the the script file `MWScript.m` on display. Go to Section 2 in the script file, and find the command statement `a = 2;`. There are two ways to execute this command. In the first way you use the mouse. Highlight the command using the mouse, and right-click. In the menu that pops up choose 'Evaluate Selection'. The second, and for many users the more practical way is to use the keyboard. Use the up- and down-arrow keys on the keyboard to place the cursor to the left of the command `a = 2;`. Press and keep on pressing the left shift key, and use the right-arrow, or the down-arrow key to highlight the command. To execute the highlighted command, hit the F9 key on a Windows operating system, or hit together the left shift key, the function key,

and the F7 key on an OSX operating system.

```
a = 2;
```

3. Creating a Script File

A script file is a simple text file that contains command syntax exactly as you would type them in the command prompt. It takes the file extension .m. Before creating a script file, you need to specify a working directory where you will keep your script file and MATLAB will read it from. MATLAB has a default working directory but often we want to change it to one of our own preference. Type `cd 'M:\'` in the command prompt to change the current working directory to be your account folder.

Type `edit scriptfile` in the command prompt to create a script file named `scriptfile`. This will open the Editor where you can store program syntax. Alternatively, you can use the program menu to create a script file. On the program menu, go to the 'HOME' tab, click on the 'New' button, and select 'Script' from the drop-down menu. This will open the 'Editor' panel, and create a 'script file' with the name `untitled`.

Close the script file you have just created. We will work with the script file created for this workshop. To open it, double click on `MWScript.m`. Note that the name of the script file should not contain character space because otherwise MATLAB will not recognise the file.

```
cd 'M:\'
edit scriptfile
```

4. Program Syntax

MATLAB syntax takes two forms; command syntax and function syntax. In its simplest form, a command syntax is a statement. E.g., the statement `clear` clears the memory from any preloaded program code or data. The statement `5+5` executes an arithmetic operation. If you type `5+5` in the command prompt, MATLAB will return `10`.

Often we would want to assign a statement to a variable, to refer to it easily throughout the code, or to use the statement as input in other statements. E.g., you can assign the statement `5+5` to a variable named `a` by typing in the command prompt `a = 5+5`. MATLAB will evaluate the statement, and return `a = 10`. You can then use `a` in a new statement, say in `b = a+1`.

Note that before assigning a statement, or a value, to a variable, you do not have to declare the variable. Also note that if you do not assign a statement to a variable, MATLAB will automatically assign it to a variable named `ans`. In fact, this is what happened when you have typed `5+5` above. However, MATLAB will overwrite `ans` with every command that returns an output value that is not assigned to a variable.

MATLAB stores created variables in the Workspace. You can inspect variable `a` if you double click on it in the Workspace.

Function syntax, as its name suggests, is used when using built-in MATLAB functions. Functions take input arguments and return output. You enclose input arguments in parentheses, and separate them with commas. E.g., you could use the `round` function to round the mathematical pi constant to the nearest 3 decimal digits using the function syntax `round(pi,3)` which will return `3.1420` as output.

You can suppress the output produced by a command and printed in the Command Window

if you add a semicolon after the command. E.g., `round(pi,3);` will not print the output `3.1420` in the Command Window.

MATLAB is case sensitive meaning that lowercase and uppercase letters have different meanings. E.g., `Pi` is not an alternative for `pi`.

```
clear
5+5
a = 5+5
b = a+1
round(pi,3)
round(pi,3);
```

5. Creating, Indexing, and Searching Arrays

In MATLAB environment, we work with arrays. An array is an n-dimensional rectangular structure holding arbitrary data. E.g., a scalar variable is an one-dimensional array. Or, a matrix is a two-dimensional array. Hence, variable `a`, created above, is an array. In particular, it is a 1×1 matrix which stores the value 10 as its element. Create a vector array by typing `c = [2 7 5 1 8 9 0 1 1]'` in the command prompt. The transpose operator (`'`) converts the array into a column vector. To learn more about arrays, type `doc matrices and arrays` in the command prompt. Throughout this workshop you will learn about the different types of arrays.

To access selected elements of an array, we use 'array indexing'. To access the element in the second row of the vector array `c` you just created, type `c(2,1)` in the command prompt. `2` is the 'row subscript', and `1` is the 'column subscript'. To access the first three rows of the array, type `c(1:3,:)`. `:` is the 'colon operator' which allows you to specify a range of the form `start:end`. Hence, `1:3` selects all the rows from the first to the third row of the vector array. The colon operator alone, `:`, is shorthand for `1:end`. It selects all the columns from the first to the very last column of the matrix array. Because `c` is a column vector, typing `:` is equivalent to typing 1, or is not necessary. See `doc indexing` to learn more about array indexing.

The colon operator also allows to create an equally spaced vector of values using the more general form `start:step:end`. E.g., `c = 0:2:10` creates a vector array with values increasing from 0 to 10 by an increment of 2.

To search for certain elements in an array, and for their position in the array, you can use the `find` function. Execute once again the syntax `c = [2 7 5 1 8 9 0 1 1]'`. Let us investigate if `c` contains the value 1, and its position in `c`. We could use the function syntax `[row,col,v] = find(c(:,:) == 1,2,'first')`. The function accepts three input arguments. The first input argument specifies the array to search. In the example we require that the search takes place in all rows and the only column of `c` that contain the value 1. The second input argument specifies the number of instances of 1 to search. Let us search for two instances. The third argument specifies whether the search should consider the `'first'` or the `'last'` two instances of 1 in `c`. The function returns three arrays as output. `row` and `col` contain the row and column subscripts of each instance of 1 in the array. `v`, in principle, would contain the elements we have searched for. However, since we have already specified what we are searching for, `v` contains a logical value of 1 (true) for each instance of 1. That is, you could consider the first input argument without `== 1` to search for the two elements stored in the first two rows of the vector array `c`.

```
c = [2 7 5 1 8 9 0 1 1]'
c(2,1)
c(1:3,:)
c = 0:2:10
c = [2 7 5 1 8 9 0 1 1]'
[row,col,v] = find(c(:,:)  == 1,2,'first');
```

6. Commenting, Breaking, and Masking Syntax in a Script File

There are two ways to include comments in a script file. First, you can begin a line with the percent symbol (%), and everything you type after the symbol to the end of the current line is considered a comment. Second, following a command syntax, you can start with the percent symbol, and everything you type after the symbol to the end of the current line is considered a comment. See below examples for each type of syntax commenting.

In your script file you may need to type a line of code that is wider than the width of the Editor. You can simply type the long line of code. However, if you want the code to stay within the limits of the Editor, you can split the code across multiple lines by adding an ellipsis (...) at the point you want to break the code. See below an example.

You may want to mask code if you want MATLAB to ignore it. You can mask a single line of code simply by placing a percent symbol in front of the code as if you were commenting syntax. You can mask several lines of code by enclosing the code within the block comment operators %{ %}. See below some examples.

```
% You can add a line of comment.
a = 5+5; % You can annotate code.
% You can split a line of code across multiple lines.  E.g.:
a = ...
    5+5;
% You can mask multiple lines of code.  E.g.:
%{
a = 5+5;
b = a+1;
%}
% You can mask a single line of code.  E.g.:
% a = 5+5;
```

7. Importing Raw Data Files

Data files hardly come in a format that can be read directly by the software of choice. Often they come in raw text format. One such raw data format is the 'comma separated value' (csv) format. Inspect the content of the csv file we will use. To open the csv file, right click on the csv file, and in the menu that appears choose Notepad++, which is a simple text editor. Observe that the variable names are stored in the first row, the values beneath the variable names, and how both are separated from each other with a comma (,).

This section explains how to import a csv file into MATLAB, and to save it in a data format native to MATLAB. Before you proceed, clear the system memory by typing `clear` in the command prompt. We start by defining several variables which we will use as input arguments

in the import function we will consider eventually. First, we define the directory where the raw data file is located, alongside the name of the raw data file. We can do this using the statement `M:\MW - csv file.csv`. However, we need to enclose the statement in quotes, as `'M:\MW - csv file.csv'`, because the statement is a string entry rather than a numeric entry. Next, assign the statement that defines the directory to a variable named `filename`. We can now define our first input argument as `filename = 'M:\MW - csv file.csv'`.

In the raw data file, the names of the variables are stored in the top row, and the values of the variables are stored in the rows beneath. In the rows of the data file, the names and the values of the variables are separated by a delimiter, where the delimiter is a comma (`,`). We need to declare the delimiter, and we should do this by enclosing the delimiter in quotes, as `','`, because the delimiter is a string entry. Let us assign the delimiter to a variable named `delimiterIn`. We can now define our second input argument as `delimiterIn = ','`.

Raw data files usually come with variable names stored in the first row of the raw data file. This is also the case with the csv file at hand. We need to specify the row number where variable names are stored in the data file. It is the first row of our csv file. Hence we assign value `1` to a variable named `headerlinesIn` by typing `headerlinesIn = 1` in the command prompt. This defines our third input argument.

We are now ready to use the `importdata` function that allows to import the csv file into MATLAB. The function accepts the three variables created above as input arguments, and returns data in `mat` format native to MATLAB as output. In particular, consider the command `mwmatfile = importdata(filename,delimiterIn,headerlinesIn)`. `mwmatfile` is a name we give to the output that the `importdata` function will return. The command in whole instructs MATLAB to read the raw data from `filename`, with the delimiter `delimiterIn`, where variable names are stored in row `headerlinesIn`. The command will return a 'structure array', named as `mwmatfile`, that contains the imported data that is now in `mat` format. We will learn about the structure array in a future section.

We would want to save the imported data, currently being held in the system memory, to our hard drive. Note, however, that the Workspace contains several arrays, and we want to save in a file only the structure array that contains the imported data, and exclude all other arrays. We can use the `save` function to achieve this. The function syntax is `save('M:\MW - mat file','mwmatfile')`, which instructs MATLAB to select array `mwmatfile` and save it in the file `MW - mat file`.

```
clear;
filename = 'M:\MW - csv file.csv';
delimiterIn = ',';
headerlinesIn = 1;
mwmatfile = importdata(filename,delimiterIn,headerlinesIn);
save('M:\MW - mat file','mwmatfile');
```

8. Opening Data Files

Remove items from the Workspace using the command `clear`. Suppose that you just started a new session and want to open a mat file, say the mat file you just created in the preceding section. You can use the `load` command to achieve this. That is, you can type `load 'M:\MW - mat file'` in the command prompt to open the mat file.

```
clear;
```

```
load 'M:\MW - mat file';
```

9. Browsing the Data

A 'structure array' is a data type that groups related data using data containers called fields. Each field can contain data of any type and size. You can refer to the data in a structure array using 'dot indexing' of the form `structureName.fieldName`. For more information on the structure array, execute the command `doc create a structure array`.

`mwmatfile` is a structure array contained in the mat file `MW - mat file.mat`, which was loaded into the system memory in the preceding section. In the Workspace, double click on the structure array to access the three fields it contains. The first field, `data`, contains a collection of vector arrays. The other two fields contain the string variables created while importing the data. In particular, they contain the variable names. Double click on the `data` field. This will open a new tab named `mwmatfile.data`. You can use this name in command or function syntax to refer to the data contained in the `data` field. Go back to the structure array `mwmatfile`, and double click on the `textdata` field. This will open a tab named `mwmatfile.textdata`. It contains string entries such as `wage` or `female`.

Suppose that we want to browse the first 20 observations of the data field `mwmatfile.data`. For this we can use the function syntax `mwmatfile.data(1:20,:)`. `mwmatfile.data` refers to the data field we want to consider. We access the observations of interest using array indexing. That is, the row subscript `1:20` selects all the rows from the first to the twentieth row of the array. The column subscript `:` selects all the columns from the first to the last column of the data field.

If we wanted to browse only the variable `wage` contained in the first column of the data field, we could use the syntax `mwmatfile.data(:,1)`.

We could browse the data using a logical condition. Suppose that we want to browse the data on wage and education among women. Type `unique(mwmatfile.data(:,4))` to inspect the unique values the vector array `mwmatfile.data(:,4)` contains. 1 represents women. We could consider the syntax `mwmatfile.data(mwmatfile.data(:,4) == 1,[1 2 4])` to select our sample. Pay attention to the the row subscript: `mwmatfile.data(:,4) == 1`. The colon operator `(:)` selects the observations contained in all rows of the data field, `mwmatfile.data`, and `4` selects the observations contained in the fourth column of the data field. Hence `(:,4)` selects all the observations in the fourth column of the data field. `== 1` requires that these observations are for women. Consider the column subscript: `[1 2 4]`. It selects the vector arrays `wage`, `educ` and `female`. The two arguments together selects the intended sample.

We might want to inspect a variable with its observations sorted in an ascending or descending order. To make it a little more complicated, suppose that we want to sort the observations of `wage` in a descending order 'within' the observations of `educ` sorted in an ascending order. The syntax `sortrows(mwmatfile.data,[2 -1])` serves to this purpose. Pay attention in particular to the second input argument of the `sortrows` function: `[2 -1]`. It requires the second column array be sorted in an ascending order, and 'within' the sorted observations of the second column array it requires the first column be sorted in a descending order. Inspect the result stored in `ans` in the Workspace.

Note that the command `sortrows(mwmatfile.data,[2 -1]);` does not replace the original sorting of the data contained in the data field `mwmatfile.data`. That is, after sorting the data using the command `sortrows(mwmatfile.data,[2 -1]);`, if you inspect the data field `mwmatfile.data`, you will notice that the data is not sorted. If you want the sorting to apply to the data filed `mwmatfile.data`, you need to 'assign' the sorting statement

`sortrows(mwmatfile.data,[2 -1]);` to the data field name `mwmatfile.data` using the command `mwmatfile.data = sortrows(mwmatfile.data,[2 -1]);`.

We could obtain the sizes of each dimension of the data field `mwmatfile.data` using the function syntax `[N,k] = size(mwmatfile.data)`. `size` is a built-in function that accepts an array as input, and returns the sizes of the row and column dimensions of the array as output. In the example, the function syntax takes the data field `mwmatfile.data` as input, and returns the row and column dimensions of `mwmatfile.data` as output in separate variables `N` and `k`.

For more information on browsing data, execute `doc select subsets of observations`.

```
mwmatfile.data(1:20,:);
mwmatfile.data(:,1);
unique(mwmatfile.data(:,4))
mwmatfile.data(mwmatfile.data(:,4) == 1,[1 2 4]);
sortrows(mwmatfile.data,[2 -1]);
mwmatfile.data = sortrows(mwmatfile.data,[2 -1]);
[N,k] = size(mwmatfile.data);
```

10. Producing Descriptive Statistics

Built-in MATLAB functions can be used to produce descriptive statistics. To compute the mean of the elements in each column of the data field `mwmatfile.data`, we can use the function syntax `mean(mwmatfile.data,1)`. The `mean` function accepts two arguments. The first input argument specifies the array containing data. The second input argument specifies the dimension of the array along which the mean is computed. A value of `1` specifies that it is the column dimension of data field `mwmatfile.data` to calculate the mean of. Therefore the function returns a row vector containing the mean of the elements in each column of the data field. If we have set the dimension to `2`, the function would have returned a column vector containing the mean of the elements in each row of the data field. If you ignore the second input argument, MATLAB will assume that it is `1`.

Relational operators can be used in statistical functions to obtain group statistics. E.g., `mean(mwmatfile.data(mwmatfile.data(:,4) == 1,1))` produces the mean of wage among women, using the relational 'Equal to' operator (`==`). Convince yourself that this is true. Note that we have suppressed the second input argument of the `mean` function.

To produce a frequency table for `educ`, we could consider `tabulate(mwmatfile.data(:,2))`. The `tabulate` function produces a table with three columns. The first column contains the unique values of `educ`, the second column contains the number of instances of each unique value, and the third column contains the fraction of each value among all values.

In econometrics, we are hardly interested in the statistics of a single variable, but in the statistical relationships between variables. The function `[r,p] = corrcoef(mwmatfile.data(:,1),mwmatfile.data(:,3))` accepts column vectors `wage` and `exper` as input, and returns correlation coefficients as well as p-values for testing the hypothesis of no correlation as output. For more on descriptive statistics, consider `doc descriptive statistics`.

Note that when you execute the function syntax `mean(mwmatfile.data,1)`, it returns a row vector containing the value `NaN` as one of its elements. `NaN` stands for Not-a-Number. It results from operations which have undefined numerical results. The `mean` function generated a `NaN` value because the vector array contained in the second column of the data field `mwmatfile.data` contains missing observations. Browse to the end of the data field to notice the missing observations marked by `NaN` values. To ignore the `NaN` values and compute the mean of the vector

array, you can add `'omitnan'` as a third input argument in the `mean` function. That is, you can consider the function syntax `mean(mwmatfile.data,1,'omitnan')`.

To count the number of missing cases in the vector array `educ`, you can use the `isnan` function within the `sum` function as `sum(isnan(mwmatfile.data(:,2)))`. Convince yourself that the function syntax does what it intends to do.

```
mean(mwmatfile.data,1);
mean(mwmatfile.data(mwmatfile.data(:,4) == 1,1));
tabulate(mwmatfile.data(:,2))
[r,p] = corrcoef(mwmatfile.data(:,1),mwmatfile.data(:,3));
mean(mwmatfile.data,1,'omitnan');
sum(isnan(mwmatfile.data(:,2)));
```

11. Creating Variables

In the preceding sections we have accessed data stored in a structure array using array indexing of the form `structureName.fieldName`, and the colon operator, `:`. E.g., we have accessed wage data using the syntax `mwmatfile.data(:,1)`. This way of accessing vector arrays is a little tedious unless we want to access certain rows or columns of the vector arrays using the colon operator. Instead, we can assign vector arrays to variables, and access vector arrays using variable names. E.g., `wage = mwmatfile.data(:,1)` assigns the vector array `mwmatfile.data(:,1)` to a variable named `wage`.

We can generate new variables using arithmetic, logical, and relational operators. Consider the following commands that generate new variables using arithmetic operators: `wage+1`, `-wage`, and `wage.^2`. Pay attention to the dot operator, `.^`, used in the last command. It raises each element of the vector array `wage` to a power of two. Other dot operators include `.*` and `./`. Each syntax consists of a dot and a standard algebraic operator. They are used to perform element-wise algebraic operations on matrices.

The command `wage(:) >= 20 & wage(:) <= 40` creates a variable using the logical 'and' operator (`&`), but also the relational 'greater than or equal to', and the 'less than or equal to' operators (`>=`, `<=`). It creates in particular a dummy variable taking a value of 1 when `wage` is between 20 and 40, and 0 otherwise. Note that in the command syntax we did not need to specify a column dimension for `wage` because `wage` consists of only one column of data. That is, `wage(:,1) >= 20 & wage(:,1) <= 40` would have produced the same vector array. For more information on the operators, execute `doc operators and elementary operations`.

We can use the built-in mathematical functions to transform variables. E.g., `log(wage)` takes the natural logarithm of `wage`, or `exp(wage)` performs the exponential transformation. To see the full list of built-in mathematical functions, execute `doc mathematical functions`.

A useful feature of MATLAB is that when you hover your mouse cursor over an array in the Editor, a window will pop up and show the size and the content of the array. To enable this feature, in the program menu, go to the 'VIEW' tab, and check 'Enable data tips while editing'. Experiment this feature by moving your mouse cursor on `wage` in the Editor.

```
wage = mwmatfile.data(:,1);
wage+1;
-wage;
wage.^2;
wage(:) >= 20 & wage(:) <= 40;
```

```
log(wage);
exp(wage);
```

12. Manipulating Variables

To keep certain variables, say `N`, `k`, and `mwmatfile`, and remove all others in the Workspace, use the syntax `clearvars -except N k mwmatfile`.

To delete a vector array contained in a matrix array, assign the vector array an empty array. E.g., to delete the vector array `white` in the matrix array `mwmatfile.data`, assign `white` an empty array using the command `mwmatfile.data(:,6) = []`.

The vector array `educ` in the matrix array `mwmatfile.data` contains observations with missing values marked by `NaN` values. We may want to delete observations with missing values, but also the observations with non-missing values contained in the same row but in other columns of the data sheet, because observations with missing values cannot be used in combination with observations wth non-missing values to produce, say, multivariate statistics. Consider the command `mwmatfile.data(isnan(mwmatfile.data(:,2)),:) = []`. The function syntax `isnan(mwmatfile.data(:,2))`, as the row subscript of the array `mwmatfile.data`, selects the rows of `educ` that contain `NaN` values. The colon operator (`:`), as the column subscript of the array `mwmatfile.data`, selects all the columns of the matrix array. The command in whole deletes the rows that contain `NaN` values.

The `wage` data contains values larger than 30. Suppose that we believe that these are outliers, and that we want to replace them with missing values. Convince yourself that the command `mwmatfile.data(mwmatfile.data(:,1) > 30,1) = NaN` replaces values of `wage` larger than 30 with NaN values.

```
clearvars -except N k mwmatfile;
mwmatfile.data(:,6) = [];
mwmatfile.data(isnan(mwmatfile.data(:,2)),:) = [];
mwmatfile.data(mwmatfile.data(:,1) > 30,1) = NaN;
```

13. Data Types

An array can hold different data types. Type `doc Fundamental MATLAB Classes` to see the data types.

`mwmatfile.data` holds data that is of the numeric type. Therefore, `mwmatfile.data` is a numeric array. Type `isnumeric(mwmatfile.data)` in the command prompt to confirm this. The numeric data type itself has different types. For example, the first column of `mwmatfile.data` contains `NaN` entries. `NaN` is a special numeric data type. That is why `isnumeric(mwmatfile.data)` did not complain about the `NaN` entries.

`mwmatfile.textdata` holds data that is of the cell type. Therefore, `mwmatfile.textdata` is a cell array. Type `iscellstr(mwmatfile.textdata)` in the command prompt to confirm this. A cell array is a type of array that can hold different types of data. Here `mwmatfile.textdata` holds data that is of the string type.

Built-in functions can convert one data type into another. We do not pursue this here.

```
doc Fundamental MATLAB Classes
isnumeric(mwmatfile.data)
```

```
iscellstr(mwmatfile.textdata)
```

14. Scalar vs Vector Oriented Coding

In MATLAB, or in comparable software like R, we can carry out matrix operations in two ways. In the first way we work with the elements of a vector array, and use loops to carry out matrix operations on each element in an iterative process. In the second way we work with all elements of a vector array at a time when carrying out matrix operations. Vectorised code is easier to understand, less prone to errors, and often runs faster than the corresponding code containing loops. Hence we might want to vectorise our code as much as possible. You can read about the advantages of coding using vectors over using program loops at `doc vectorisation`.

Open the data field `mwmatfile.data` stored in structure array `mwmatfile` in the Workspace. Note the sorted values of `educ`. Suppose that we want to delete the rows of the data field if a unique value of `educ` is ordered first among its kind. E.g., there are six cases where `educ` takes the unique value of 4. We want to delete the rows if it is the first case of 4 among the ordered six cases. We will consider two different programs to delete the rows. The programs are presented at the end of the section.

First, consider the program containing a for loop. A for loop is a program that repeats an operation for a specified number of times. `i` is the 'index' of the loop. `uniq` contains the 'values' the index takes. It contains the unique values of `educ`. A for loop iterates over columns rather than rows, and therefore we transpose `uniq` using the transpose operator (`'`). The next line contains the 'statement' of the loop that repeats itself for the index values. `find(mwmatfile.data(:,2) == i,1,'first')` conducts a search of the row where the vector array `educ` contains the first value of its kind. Since we have specified the `find` function as a row subscript of the matrix array `mwmatfile.data`, it acts as selecting the row it conducts the search for. The column subscript (`:`) selects all the columns of the matrix array. We assign the selected rows and columns of the matrix array an empty array (`[]`) to delete the specified rows. `end` ends the loop.

Consider the second program containing only vector operations. The `ismember` function takes the unique elements of `educ` contained in `uniq`, and all elements of `educ` contained in the vector array `mwmatfile.data(:,2)` as input arguments. The function returns two arrays as output. First, it returns an array containing 1 (true) if the data in `uniq` is found in `mwmatfile.data(:,2)`. We do not need this array, and we can use the tilde operator ($\sim$) to ignore it as output. Second, it returns an array named `tag`, containing the 'lowest index' in `mwmatfile.data(:,2)` for each row in `uniq` that is also a row in `mwmatfile.data(:,2)`. That is, `tag` contains the row numbers containing the first instances of the unique values of `educ` among the sorted values of `educ`. `tag` contains 0 whenever `unique` is not a row of `mwmatfile.data(:,2)`.

`unique` and `ismember` are members of the set of functions that carry out set operations which prove useful when you need to access certain values in vector arrays. For more information, see `doc set operations`.

```
% Scalar-oriented coding
uniq = unique(mwmatfile.data(:,2));
for i = uniq'
    mwmatfile.data(find(mwmatfile.data(:,2) == i,1,'first'),:)  = [];
end
% Vector-oriented coding
```

```
uniq = unique(mwmatfile.data(:,2));
[~,tag] = ismember(uniq,mwmatfile.data(:,2));
mwmatfile.data(tag',:)  = [];
```

15. Efficient calculation in MATLAB

Suppose that you want to multiply a square matrix with an identity matrix. Note that for this particular matrix multiplication the off-diagonal elements of the identity matrix are not relevant to the final matrix you want to end up with because all these elements are zeros and do not contribute to the multiplication. If this is so, why to use the off-diagonal elements while carrying out the matrix multiplication that costs computation time and computer power?

There are two routines presented at the end of the section. The first routine employs the `eye` function to carry out the matrix multiplication of interest. The built-in `eye` function creates an identity matrix of size determined by the input argument of the function.

The second routine employs the `speye` function to make the same matrix multiplication. The `speye` function creates a sparse identity matrix. This means that it considers only the diagonal elements of the identity matrix of interest, and disregards the off-diagonal elements.

Observe that the computation time of the first routine is substantially higher than that of the second routine. Hence, whenever possible, we would want to use sparse matrices to save time and computer power. To learn more about sparse matrices, type `doc sparse` in the command prompt to find out.

```
A = [1 2 3; 4 5 6; 7 8 9];
square_matrix = repmat(A,200);
tic
efficientnot = square_matrix*eye(600);
toc
tic
efficientyes = square_matrix*speye(600);
toc
```

16. Assign Names to Vector Arrays

We have learned working with vector arrays. We can now put this knowledge into use to analyse economic phenomena. First, let us make our data ready. Clear the memory, reload the mat file containing our data, delete the rows where `educ` takes a value of `NaN` in the matrix array, and obtain the size of the row and column dimensions of the matrix array, using the first four lines of the syntax presented at the end of the section.

In our data analysis we will work with vector arrays, rather than with a matrix array, and therefore we need to extract vector arrays from our matrix array. In our dataset we have six vector arrays in total. We could assign each vector array to a variable and access vector arrays using variable names. For this we could use, e.g., `wage = mwmatfile.data(:,1)`. However, it is tedious to assign each vector array in the dataset to a variable for six times. Instead, we can write a for loop to automate creating variables. In particular, our aim is to assign vector arrays to variables stored in `mwmatfile.data(:,i)` where `i` is a vector array. Consider the command `eval([cell2mat(mwmatfile.textdata(i)) '= mwmatfile.data(:,i);'])`. The first expression, `cell2mat(mwmatfile.textdata(i))`, takes the string data stored in `mwmatfile.textdata`

for `i`, which is essentially the name of variable `i`. String data is stored in a cell array, and the cell array needs to be converted to an ordinary array, to use it in the `eval` function. This can be achieved with the `cell2mat` function. The second expression `'= mwmatfile.data(:,i);'` takes the numeric data stored in `mwmatfile.data` for `i`, which is essentially variable `i`. The `eval` function assigns the string name to the numeric data. The loop repeats the procedure for each `i` from `1` to `k` where `k` is the size of the column dimension of the data matrix we have defined above. Check `doc genvarname` to learn more about naming variables.

```
clear;
load 'M:\MW - mat file';
mwmatfile.data(isnan(mwmatfile.data(:,2)),:)  = [];
[N,k] = size(mwmatfile.data);
for i = 1:k
    eval([cell2mat(mwmatfile.textdata(i)) '= mwmatfile.data(:,i);'])
end
```

17. Regression Analysis Using Matrix Algebra

Consider the linear regression model with multiple regressors that take the form $y = \mathbf{X}\boldsymbol{\beta} + u$. $\mathbf{X}$ is a N×K matrix. N denotes the number of observations. K denotes the number of vector arrays in $\mathbf{X}$. There is one vector array for the constant term, and k vector arrays for k regressors. Hence K = 1+k. Each vector array is of size N×1. $\boldsymbol{\beta}$ is K×1. It represents the true coefficient vector. The error term is N×1, and follows a normal distribution. Our aim is to estimate the parameter vector $\boldsymbol{\beta}$. You may recall from your introductory econometrics course that solving the F.O.C. of the least squares problem leads to the OLS parameter estimates, $\hat{\boldsymbol{\beta}}_{OLS}$, that take the form $(\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y}$. This means that the matrix operation we need to carry out in MATLAB is `(X'*X)\X'*y`. `y` is the dependent variable. In our example it is the `wage`. For notational convenience, rename `wage` as `y`. `X` is the systematic component of the regression equation. It includes a constant and a set of regressors. Let us denote the constant with `c`. Create the constant term using the command `x_0 = ones(N,1)`. The function `ones` creates a vector array of all ones where the size of the row dimension of the array is `N`. Let us consider `educ` and `exper` as two regressors. We can create `X` as `[x_0 educ exper]`. Consider the matrix operation `(X'*X)\X'*y` that produces a vector array with OLS estimates of the true parameter vector $\boldsymbol{\beta}$. `*` is the matrix multiplication operator that multiply two matrices which have a common inner dimension. `\` is the matrix left division operator to solve the system of linear equations `(X'*X)\X'*y`. Instead, we could consider `inv(X'*X)*X'*y` where `inv` is the inverse function. The former way is faster and more robust than the latter. See `doc mldivide` for more information.

```
y = wage;
x_0 = ones(N,1);
X = [x_0 educ exper];
B_hat_OLS = (X'*X)\X'*y;
B_hat_OLS = inv(X'*X)*X'*y;
```

18. Creating Graphs

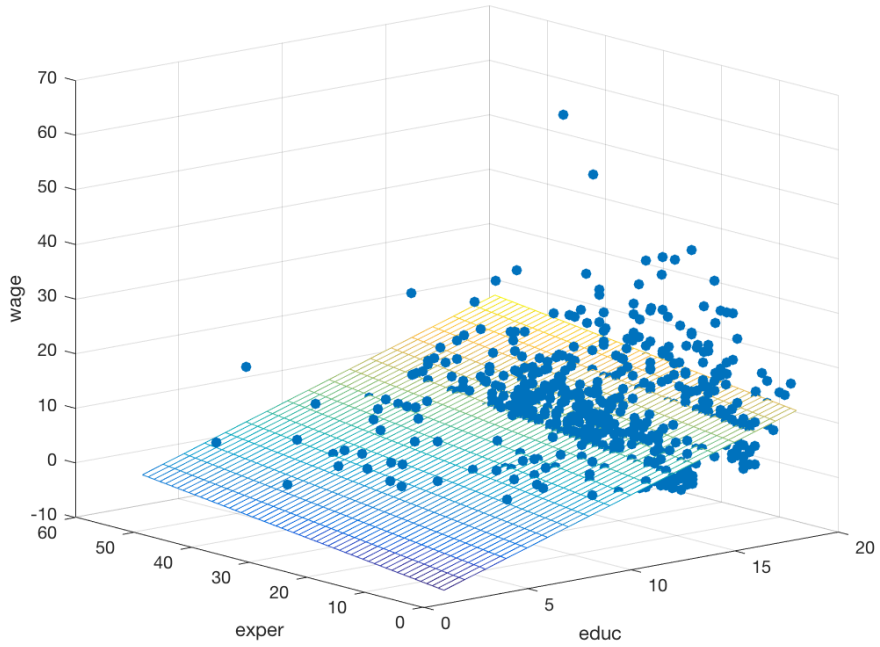
It is helpful to produce a scatter plot to visualise the relationships between the dependent variable and the independent variables to form expectations about the signs and significance of the effects of the independent variables. Here we learn how to produce a three dimensional scatter plot. We will then also combine it with a best-fit plane.

Consider the lines of syntax presented at the end of the section. The first line uses the `scatter3` function produce a three dimensional scatter plot. In particular, it displays circles at the locations specified by three vectors representing respectively the x, y, and z dimensions of the Cartesian coordinate system. `'filled'` fills in the circles. In the second line, `hold on` retains in the axes the plot we just created so that the next plot we will add to the axes does not delete the existing plot. The third line creates a vector array with equally spaced values starting from the minimum and running until the maximum value of `educ`. The fourth line repeats the command for `exper`. In the fifth line, `[educFIT,experFIT] = meshgrid(educfit,experfit)` returns 2-D grid coordinates based on the coordinates contained in vectors `educfit` and `experfit`. `educFIT` is a matrix where each row is a copy of `educfit`, and `experFIT` is a matrix where each column is a copy of `experfit`. The grid represented by the coordinates `educFIT` and `experFIT` has `length(educfit)` rows and `length(experfit)` columns. In line six, we predict `wage` at each value of `educFIT` and `experFIT` using the estimated parameters of our regression model in the preceding section. In line seven, `mesh(educFIT,experFIT,wageFIT)` draws a mesh plot.

The syntax in the remaining lines of the code at the end of the section determine the labels and the viewpoint of the mesh plot. See `doc surface and mesh plots` for more information on 3-D plots.

In the plot, note how the slope of the mesh confirms the magnitudes of the coefficient estimates of `educ` and `exper` relative to each other. That is, the slope of the plot is steeper at larger values of `educ` than at larger values of `exper`, confirming the larger magnitude of the coefficient estimate of `educ` than that of `exper`.

```
scatter3(educ,exper,wage,'filled')
hold on
educfit = min(educ):1:max(educ);
experfit = min(exper):1:max(exper);
[educFIT,experFIT] = meshgrid(educfit,experfit);
wageFIT = B_hat_OLS(1) + B_hat_OLS(2)*educFIT + B_hat_OLS(3)*experFIT;
mesh(educFIT,experFIT,wageFIT)
xlabel('educ')
ylabel('exper')
zlabel('wage')
view(-40,15)
```



19. Working with a Custom-built Function

So far we have been working with a script file to keep and execute program code. A script file is the most basic program file type of MATLAB. A second type of program file is the function file. Like the script file, the function file takes the file extension .m. A function file contains only a function that accepts input arguments and returns output. We call the function from a script file, and therefore the function acts as input for the command we execute in a script file. In our script file we have already been calling built-in MATLAB functions. In this section we will learn working with a custom built function.

Double click on the function file `MWFunctionLSS.m` to open it the Editor. Inspect the content of the function file. The first line always declares a function name, the input and the output arguments. We name the output as `LSS`, the function itself as `MWFunctionLSS`, and the input arguments as `y` and `X`. The name of the function file should match the name of the function in the function file. The lines beneath the first line contain executable statements which compute OLS estimates and related regression statistics. Statements should be assigned names comprising the name of the output argument followed by a dot and a custom name. E.g., we assign `length(y)` to `LSS.N`. `end` marks the end of the executable statements of the function.

`MWFunctionLSS.m` contains new command syntax. E.g., `length` is a built-in function that simply returns the number of elements contained in a vector array. Note the `rdivide` function, `./`, used in `ones(LSS.N)./LSS.N`. `rdivide` stands for right array division, and it divides each element of `ones(LSS.N)` by the corresponding element of `LSS.N`. Compare `rdivide` with `ldivide`, `mldivide`, and `mrdivide` in MATLAB documentation files.

To access the function defined in `MWFunctionLSS.m` in a script file, we need to make the directory where the function file is located on our hard drive known to MATLAB. Right click on the tab holder of the function file, and choose 'Change Current Folder to ...'. If the option is greyed out, and hence not selectable, it means that the Current Folder is already showing

the directory where the function file is located. We can now call the custom-built function in our script file. Consider the function syntax `LSS = MWFunctionLSS(y,X)`. We have already defined the input arguments `y` and `X`. Executing the function syntax returns a structure array named `LSS`. See the created structure array in the Workspace. `LSS` contains standard regression statistics similar to those produced by widely used econometric software such as Stata. E.g., `LSS.t` produces the t-statistic that shows if estimated coefficients are statistically significant at a given significance level. You can also access the individual statistics produced by the `LSS` function from your script file. E.g., to obtain the OLS coefficient estimate for `educ`, use the array indexing of the form `LSS.B_hat_OLS(2)`.

You can compare the output of the custom-built function `MWFunctionLSS` with that of the built-in function `fitlm` which contains estimates of the parameters of a linear regression model and various other regression statistics similar to those produced by `MWFunctionLSS`. Consider the function syntax `lrm = fitlm(X,y,'Intercept',false)`. The first two input arguments of the function specify the systematic part and the dependent variable of the regression. The last two input arguments are optional pair arguments, and instruct that the model does not include a constant. This is because the `fitlm` function includes a constant by default which we want to avoid because `X` already includes a constant. Inspect the output the function has just returned in the Workspace. This demonstrates that it is not difficult to build your own function similar to a built-in MATLAB function.

```
LSS = MWFunctionLSS(y,X);
LSS.B_hat_OLS(2);
lrm = fitlm(X,y,'Intercept',false)
```

Content of MWFunctionLSS.m:

```
% MATLAB Workshop - Function File - Least Squares Statistics

function LSS = MWFunctionLSS(y,X)
% Number of observations and column dimension of X
LSS.N             = length(y);
LSS.K             = size(X,2);
% OLS estimates, predictions, residuals
LSS.B_hat_OLS     = (X'*X)\(X'*y);
LSS.y_hat         = X*LSS.B_hat_OLS;
LSS.u_hat         = y-LSS.y_hat;
% Analysis of variance
LSS.ess           = LSS.y_hat'*LSS.y_hat;
LSS.tss           = y'*y;
LSS.rss           = LSS.u_hat'*LSS.u_hat;
LSS.R2_uc         = LSS.ess/LSS.tss;
LSS.Mi            = eye(LSS.N)-ones(LSS.N)./LSS.N;
LSS.tss_c         = y'*LSS.Mi*y;
LSS.R2_c          = 1-LSS.rss/LSS.tss_c;
% Inference
LSS.B_hat_OLS_VCE = 1/(LSS.N-LSS.K)*((X'*X)\X'*(LSS.u_hat'*LSS.u_hat)*X)/(X'*X);
LSS.B_hat_OLS_SEE = sqrt(diag(LSS.B_hat_OLS_VCE));
```

```
LSS.t               = LSS.B_hat_OLS./LSS.B_hat_OLS_SEE;
LSS.p               = (1-cdf(makedist('Normal'),abs(LSS.t)))*2;
end
```

20. The Optimisation Problem of an Econometrician

Consider once again the linear regression model with multiple regressors that take the form $y = \mathbf{X}\boldsymbol{\beta} + u$. In Section 15, we have used the F.O.C. of the least squares problem to obtain estimates of the true parameter vector $\boldsymbol{\beta}$. In this section we will minimise the least squares objective function using a built-in optimisation algorithm to obtain the same estimates of $\boldsymbol{\beta}$.

Consider the function `fminunc` in the last line of the list of commands presented at the end of the section. `fminunc` stands for 'find minimum of unconstrained multivariable function'. It is an iterative algorithm that attempts to find the local minimiser of an objective function in a given interval. `fminunc` is a derivative-based search algorithm and does not guarantee a global minimum. It turns out that any derivative-based optimisation algorithm does not guarantee a global minimum.

`fminunc` accepts three input arguments. The first input argument specifies that the objective function is `MWFunctionSSR(y,X,B_hat)`, and it is to be evaluated at `B_hat`. The objective function is specified in `MWFunctionSSR.m`.

The second input argument is `B_hat_ig`. `ig` stands for 'initial guess'. `fminunc` uses an iterative algorithm that tries to find the `B_hat` that minimizes the objective function. The algorithm starts its search of the minimum with an initial guess of `B_hat_ig`. `B_hat_ig` can be a scalar, vector, or a matrix. `B_hat_ig` must be of the same size of `B_hat`.

The third input argument is `options`. It is a structure array containing all options about the way the algorithm is searching for the minimum. It accepts a number of input arguments. The first input argument specifies the analytical form of the first derivative of the objective function contained in the function `MWFunctionSSR`. It can be either gradient or Jacobian. An analytical form of the first derivative of the objective function is a required option because `fminunc` is a derivative-based search algorithm. Let us choose the gradient as the analytical form of the first derivative. This is specified with `'GradObj'` in the options. The second input argument of `options` can be `'on'` or `'off'`. `'on'` means that `MWFunctionSSR` contains a user-defined gradient that has a closed form solution. In this case the minimisation of the objective function is most precise. `'off'` means that `MWFunctionSSR` does not contain a gradient, and `fminunc` will need to approximate the gradient using numerical methods. In this case the price we pay is that the algorithm can get stuck in some regions of the parameter space and fail to converge. `MaxFunEvals` indicates the maximum number of function evaluations allowed. It takes a positive integer, and we set it to an arbitrary number of `100` iterations. `MaxIter` is the maximum number of iterations allowed. It takes a positive integer, and we set it to `100`. The last two input arguments are optional pair arguments. `Display` shows the steps, or the final outcome of the minimisation. E.g., `iter` displays output at each iteration, `final` displays just the final output, or `notify` displays output only if the function does not converge. Type `doc optimset` in the command prompt to see the full list of options.

The following are the output arguments of the `fminunc` function. `B_hat_OLS_Section_20` is where the objective function attains its minimum. Hence, `B_hat_OLS_Section_20` is the solution of the objective function. `SSR` is the value of the objective function at the solution `B_hat_OLS_Section_20`. `exitflag` indicates the reason `fminunc` stopped. The possible reasons are coded with an integer. E.g., a value of `0` means that the number of iterations exceeded `MaxIter`, or number of function evaluations exceeded `MaxFunEvals`. Type `doc fminunc` in the

command prompt, and go to the 'Output Arguments' section to see the other integers `exitflag` could return, and their respective descriptions. `output` is a structure array containing, e.g., the number of iterations taken, or the number of function evaluations. Type `doc fminunc`, and go to the 'Output Arguments' section to learn about all the elements of `output`. `GradObj` is the value of the gradient of the objective function at the solution `B_hat_OLS_Section_20`. Finally, `hesssian` is the value of the Hessian of the objective function at the solution.

Consider the syntax contained in `MWFunctionSSR.m`. The first line declares a function name, the input arguments of the function, and the output it returns. It accepts the vector array `y`, the matrix array `X`, and the parameter vector `T_hat` as input arguments. The second line defines a vector array, named as `T_hat`, that stores the values minimising the objective function. It is a K×1 vector because we have K parameters to estimate. In the next line, `SSR` represents the value of the objective function at the optimising `B_hat`. It returns a scalar value. The idea is that we choose as estimates those values of `B_hat` that minimise the square of the residual `y-X*B_hat`, which is a scalar. `GradObj` defines the gradient of the objective function. It returns the derivative of the objective function at the optimising values of `B_hat`. Hence `GradObj` is a vector with dimensions equal to the dimensions of `B_hat`. That is, `GradObj` is a 3×1 vector. It does not matter if it is defined as a row or column vector; both are accepted by `fminunc`. `end` ends the function.

```
B_hat_ig = [1 1 1]';
options = optimset('GradObj','on','MaxFunEvals',100,'MaxIter',100, ...
    'Display','iter');
[B_hat_OLS_Section_20,SSR,exitflag,output,GradObj,hessian] = ...
    fminunc(@(B_hat_OLS)MWFunctionSSR(y,X,B_hat),B_hat_ig,options);
```

Content of MWFunctionSSR.m:

```
% MATLAB Workshop - Function File - Sum of Squared Errors

function [SSR,GradObj] = MWFunctionSSR(y,X,T_hat)
B_hat = T_hat(:);
SSR = (y-X*B_hat)'*(y-X*B_hat);
GradObj = 2*(X'*X)*B_hat-2*X'*y;
end
```

21. The Optimisation Problem of an Agent

In the preceding section we have dealt with an econometric problem, which was estimating the parameters of a regression model using a built-in optimisation algorithm. In this section we will deal with an economic problem, and in particular, with maximising consumer utility using a built-in optimisation algorithm.

Consider a consumer choosing a bundle of goods to maximise her utility. Assume that her utility is represented by a function of the form $u(x_1, x_2) = x_1^{1/2} x_2^{1/2}$. This is a simple Cobb Douglas utility function. Note that it is non-linear. $x1$ and $x2$ represent the quantities of two goods, and u represents utility. Assume also that the agent is subject to a budget constraint represented by the inequality $I \geq p_1 x_1 + p_2 x_2$. p_1 and p_2 represent the prices of the two goods,

and I represents the total income available for consumption.

We will use the built-in `fmincon` function to find the bundle of goods maximising the utility of the consumer. `fmincon` is a function that finds the minimum of a function subject to linear inequality constraints. Execute `doc fmincon` to inspect the formal representation of the optimisation problems it solves.

Consider the function syntax presented in the last line of the list of commands at the end of the section. The `fmincon` function accepts a number of input arguments depending on the structure of the optimisation problem. In our example it accepts seven arguments. The first input argument specifies that the objective function is `@MWFunctionCDU` declared in `MWFunctionCDU.m`.

Let `X` denote the vector of choice variables which are quantities of each good the agent wants to consume ($x1$ and $x2$). `fmincon` uses an iterative algorithm that tries to find the `X` that minimise the objective function subject to a constraint. The algorithm starts its search of the minimum with an initial guess. `X_ig` is the initial guess for `X`. `ig` stands for 'initial guess'. This is the second input argument.

All the other input arguments, following the first two input arguments, specify the components of the constraint function. `fmincon` accepts linear inequality constraints of the form $Ax \leq b$. The input arguments A and b are made known to `fmincon` by specifying them in correct order in `fmincon`. That is, first, A is specified, and then b is specified. Our linear inequality constraint is $p_1x_1 + p_2x_2 \leq I$. Let us define `P` to be a vector array containing good prices p_1 and p_2, and assume that they are 4 and 7, respectively. Let us define `I` as a scalar variable representing the total income available for consumption, and assume that it is 100 euros. We can now specify `P` and `I` as input arguments in `fmincon`. The following two input arguments of `fmincon` are for a possible equality constraint, and we leave them as blank with the empty array operator (`[]`) since we have no equality constraint to declare. `fmincon` allows to specify a lower bound for the choice variables. Let us define `lb` to be a vector array containing the smallest quantities of each good to be consumed, and assume that they are both 0. We then specify `lb` as the last input argument of `fmincon`.

The following are the output arguments of the `fmincon` function. `X_optimal` is where the objective function attains its minimum. It represents the optimal consumption bundle. `CDU` is the value of the objective function at the solution `X_optimal`. It represents the utility from consuming the optimal consumption bundle. `exitflag` indicates the reason `fmincon` stopped. The Workspace shows that it takes the integer 1. Type `doc fmincon` in the command prompt, and go to the 'Output Arguments' section to read about the meaning of integer 1. For brevity we do not discuss it here.

Consider the content of `MWFunctionCDU.m`. The first line declares a function name, the input argument of the function, and the output it returns. The input argument of the function is the vector of choice variables `X`. In fact, our utility function takes two input arguments (x_1 and x_2). However, fmincon is designed to work with a single vector of variables. Therefore, the second and third lines of the function define the vector array `X` that contains the quantities of two goods to be consumed. `X` is a 2×1 vector because the agent consumes two goods. In the next line, `CDU` defines the objective function. We take the negative of the objective function because fmincon is using a minimisation, and not a maximisation algorithm. `end` ends the function.

This example is taken from Adams, A., Clarke, d., and Quinn, S., 2015. Microeconometrics and MATLAB: An Introduction. Oxford University Press. The code is modified, however.

```
X_ig = [15,5];
P = [4,7];
I = 100;
```

```
lb = [0,0];
[X_optimal,CDU,exitflag] = fmincon(@MWFunctionCDU,X_ig,P,I,[],[],lb);
```

Content of MWFunctionCDU.m:

```
% MATLAB Workshop - Function File - Cobb Douglas Utility

function CDU = MWFunctionCDU(X)
x1 = X(1);
x2 = X(2);
CDU = -(x1^0.5)*(x2^0.5);
end
```

22. Debugging Program Code

At the end of this section you will find a for loop. This is the same for loop used in Section 13 with the exception that it now contains an error, or as it is usually called, a bug. In this section you will learn how to debug your code.

Have the script file `MWScript.m` on display. Go to the program toolbar. Select the Editor tab. At the right hand side, you will see the 'Run' pane. Click on the green 'Run' button. MATLAB will start executing the code in the script file, but will break the execution due to an error. In the Command Window MATLAB will tell you about the error, and even show the number of the line the error occurs in the script file. However, it is not so clear what the error is about. We will take a number of simple steps to locate, understand, and fix the error.

In a script file, executable lines are indicated by a dash (-). We know that the error occurs on line 210, which is part of Section 21. Go to this section, and click on the dash at line 206 which is the first executable line of this section. The dash will turn into a breakpoint (•). This asks MATLAB to pause the execution of the entire code at this point. If we do not receive an error until this pause, we know that the error is not in part of the code before the pause. It makes sense to set multiple breakpoints (pauses) so that we can carry out our search of the error progressively from one breakpoint to the other, along the script file. Let us set another breakpoint at line 208 since we suspect that the error is not related to loading the data.

Once again, go to program toolbar, then the Editor tab, and then the 'Run' pane. Click on the 'Run' button. Since we have set breakpoints, pressing the Run button will now put MATLAB in the 'Debug mode', and produce the following results. In the Editor tab, the Run pane changes to the 'Debug' pane, and the Run button changes to 'Continue' button. MATLAB pauses at the first breakpoint. Indeed, go to line 206, and notice the green right-arrow just to the right of the breakpoint which indicates the pause. MATLAB does not execute the line where the pause occurs until it resumes running. Notice that we did not receive an error, and hence realise that the error should be in the remaining part of the script file. Go back to the Debug pane, and press the Continue button to continue execution of the code until the next breakpoint. Notice the green right-arrow at line 208 where MATLAB pauses for a second time. Again, we did not receive an error, and hence realise that the error should be ahead in the code. Meanwhile, notice that the content of the Workspace is updated with respect to the code executed between the breakpoints. Go back to the Debug pane, but now press the 'Step' button, to continue execution of only the next line of the code. The Step button allows you to slow

down with your check because you suspect that the error is about to occur. MATLAB executes line 208 and returns no error, and moves the green right-arrow to line 209. Press the Step button. MATLAB executes line 209 and returns no error. Notice how the debugging feature of MATLAB allows you step in the execution of a for loop, making code debugging easy. Press the Step button. MATLAB returns an error and automatically quits Debug mode, inviting you to fix the error.

The error is that the column dimension of the data field `mwmatfile.data` is 6, but on line 210, the array indexing is not correct. In particular, in the syntax `mwmatfile.data(:,7)`, we specify the column subscript as 7 but the data field has 6 columns. Hence, MATLAB complains that 'Index exceeds matrix dimensions'. Therefore, the fix to the error is to replace 7 with a smaller number. You have just debugged your code!

```
clear;
load 'M:\MW - mat file';
uniq = unique(mwmatfile.data(:,2));
for i = uniq'
    mwmatfile.data(find(mwmatfile.data(:,7) == i,1,'first'),:)  = [];
end
```

23. Help System

MATLAB's help system consists of two main parts: the MATLAB Documentation, and the MATLAB Central. Consider first the MATLAB Documentation. On the program menu, click on 'Help' and select 'Documentation'. This will bring the 'MATLAB Documentation' window. In the search field type, e.g., 'descriptive statistics'. MATLAB will present the available documentation on descriptive statistics. Choose, e.g. 'Descriptive Statistics - Range, central tendency, standard deviation, variance, correlation' to access the list of commands that allow you to produce descriptive statistics.

Alternatively, you can access the documentation through the Command Window. Execute the command `doc descriptive statistics` to access the documentation on descriptive statistics. You can also access the documentation of a specific command through the Command Window. Type, e.g., `doc mean`, to bring up the documentation of the `mean` command. `help mean` also provides help on the `mean` command but the content it provides is much less comprehensive.

MATLAB documentation is also available online at the web address http://nl.mathworks.com/help/matlab/.

For your specific questions, you can get help at the MATLAB Central, the discussion forum of MATLAB users. The web address of the forum is http://nl.mathworks.com/matlabcentral/#ask-and-answer. To access the forum, you first need to create a 'MathWorks Account' at https://nl.mathworks.com/mwaccount/register?uri=%2Fmwaccount%2F

```
doc descriptive statistics
doc mean
help mean
```