LMU München
Institut für Statistik
Seminar für Finanzökonometrie

MATLAB-Übung zur Ökonometrie I
WS 2007/08
serkan.yener@stat.uni-muenchen.de

Introduction to MATLAB®

Contents

1 Introduction to the "Introduction to MATLAB"

This is a short and subjective introduction to MATLAB which stems from my own experience with MATLAB programming gathered in the course of time. Consequently, these notes are far from being exhaustive, but they should be sufficient to get a quick start. For a more comprehensive and thorough reference, you are advised to consult MATLAB's help menu, or *MathWorks*'s official online documentation at `http://www.mathworks.com/access/helpdesk/help/helpdesk.html`, or books such as Hanselman and Littlefield (2004).

Before jumping in, a last word for the absolute beginners based on my personal experience. Starting to write your own programs or to get acquainted with a new programming language is a tiresome and frustrating business. As in the learning process of any foreign language, your code will be circuitous and inefficient at the start due to the lack of your "vocabulary." At the start, you will not get what you want most of the time, because

your code is erroneous. However, making errors is the key to learning a language, since it teaches you things books or notes could not teach you (which is more than one would expect). Moreover, it will provide you a deeper understanding of commands and ways to solve a problem. Essentially, programming is a trial-and-error process that necessitates much "hands-on" work. But, as time elapses, you will notice a boost of programming efficiency and find out that there is no magic in learning to program, but that it is rather simply a question of how much time you can force yourself to spend working on it.

2 Preliminary Information

When it comes to decide which programming language to use, you should clearly base your decision on the approach you want to apply to a specific problem. In economic analysis, we can make a coarse distinction between *numerical* and *symbolic* computations. MATLAB is intended for numerical problems, although there is an add-on symbolic toolbox that has to be purchased. However, if you want closed-form solutions to algebraic problems, it is better to use Maple or Mathematica. The former has a syntax which is similar to MATLAB's symbolic toolbox. In contrast, if your problem is a numerical one, you should use GAUSS, MATLAB, or R, for example.

Economist are most likely to encounter problems that call for standard methods such as matrix algebra, matrix decompositions, simulation of random numbers, numerical optimization, and solving eigenvalue problems and systems of (non)linear equations. All of these are easy to accomplish in MATLAB. Furthermore, when taking a look at the available MATLAB toolboxes, it becomes clear that you can tackle nearly any conceivable numerical problem with MATLAB, no matter whether you were a biologist or a rocket scientist.

One of the major advantages of MATLAB is its incredible flexibility in setting up your own *functions* that do some specific computations you want. This is a crucial issue, especially for people doing research, since there are many instances where, despite of the vast arsenal of so-called *built-in functions*, which MATLAB and its toolboxes offer, you need a special function custom-tailored to your specific problem at hand. Nowadays, many researcher make their MATLAB functions available online. This provides non-commercial MATLAB functions with an open-source flavor and allows the application of state-of-the-art techniques long before they are implemented in more conventional software packages such as EViews or MS Excel, for example.

There are many textbooks out which deal with the solution of economic or, more generally, mathematical problems by using MATLAB. The following list is only a rudimentary

guide to this literature, as it is based (again) on my preferences and working experience. Coombes et al. (2000) is a very good and basic introduction to the numerical solution of ordinary differential equations using MATLAB. Venkataraman (2002) is a very well-written introduction to the field of numerical optimization. It covers all kinds of problems and develops its own solutions (no black-box). However, these solutions might be too peculiar for problems that you want to solve. Morgan (2000) and Martinez and Martinez (2002) cover MATLAB solutions to statistical problems. While the former presents more cutting-edge techniques, the latter is simpler and more comprehensive. Note that both are superb on methods but do not use economic problems as illustrations. In dynamic macroeconomics, Ljungqvist and Sargent (2000) and Favero (2001) are worthwhile mentioning. Both use MATLAB for numerical simulations. Miranda and Fackler (2002) is a very interesting book which connects pure methods and their applications to economic and financial problems. Brandimarte (2002) follows the same presentation style but concentrates on finance only. Lastly, if you are interested in MATLAB's graphical capabilities and its *Graphical User Interface* (GUI), you could look up Marchand and Holland (2003), beside MATLAB's own documentation.

Furthermore, there are some non-commercial toolboxes that you can download from the internet. For example, Harald Uhlig's *Toolkit* at `http://www.wiwi.hu-berlin.de/wpol/html/toolkit.htm` or Bennett McCallum's MATLAB files at `http://business.tepper.cmu.edu/display\_faculty.aspx?id=96` constitute, by now, standard software packages which solve (numerically) linear (rational expectations) macroeconomic models. A newer and much more flexible tool is *Dynare* at `http://www.cepremap.cnrs.fr/dynare/`. *Dynare* solves RE models based on first- and second-order approximations. Moreover, it is suppose to do structural estimation. If you are more interested in econometrics, you should check out James LeSage's *Econometrics Toolbox* at `http://www.spatial-econometrics.com/`. This is an amazing and still expanding collection of almost every econometric technique you might have heard of. *Dynare* and the *Econometrics Toolbox* are very good example for the blessings of open-source software. Its transparency allows users all around the globe to perpetually check, improve, and extend procedures.

Nevertheless, you need to learn how to write and run a *program* before using a toolbox. A program can be considered as a "stand-alone" function. Thus, a program is structurally more basic than a function. Writing a program, in turn, requires a basic understanding of how MATLAB works and what it can do.

3 Getting Started

Start MATLAB from the Microsoft start menu or by double-clicking the MATLAB icon on your desktop, if available. The MATLAB window opens which should be partitioned into three sub-windows (*default setting*). Section 3.1 explains what these windows do but, for the moment, we will solely focus on the large window on the right-hand side displaying a cursor and the so-called MATLAB *command prompt* (**»**).

As usual, commands are entered via the cursor. The prompt indicates that MATLAB is ready to receive your commands. When the cursor (and the prompt) is not displayed, MATLAB is busy doing computations so that no commands can be entered. A problem, that sometimes arises in looping algorithms, is that MATLAB hooks up in endless computations. In this case, you should stop computations by pressing the Ctrl– and the C–key simultaneously.

3.1 MATLAB Windows

In this section, we shortly characterize the three sub-windows displayed in MATLAB's default setting.

The window of primary interest is the large window on the right-hand side. It is termed the *Command Window*, since it is the vehicle by which MATLAB communicates with the user by exchanging commands for output and vice versa. Although the two smaller sub-windows on the left-hand side turn out to be useful as well, I prefer to shut them down in order to get more space in the *Command Window* for displaying output. (Notice that older versions of MATLAB did not include these two sub-windows.) Actually, closing these sub-windows is an absolutely innocuous operation, as it is pretty easy to retrieve and display their information content in the *Command Window* by invoking the appropriate commands (see Chapter 5). If you choose to close the smaller sub-windows, you can bring them back by clicking the sequence of buttons *View* ⟶ *Desktop Layout* ⟶ *Default* on the MATLAB menu bar.

The upper sub-window on the left-hand side consists of two stacked windows: *Workspace* and *Current Directory*. The window labeled *Workspace* contains information about variables currently residing in the MATLAB workspace/memory. Variables residing in the workspace must have been a priori defined by the user. Notice that all variables, you have created, are lost by exiting MATLAB unless they have been saved to disk. The window labeled *Current Directory* shows the content of the current working directory/folder. You may easily change the current working directory by navigating as within the Microsoft

Explorer.

The lower sub-window on the left-hand side, *Command History*, contains a chronologically inverted list with commands used in past sessions. These past commands can be re-activated by double-clicking on them. However, notice that if they refer to variables not contained the current workspace, you will get an error message, since MATLAB does not know the variables involved in those commands.

3.2 MATLAB Variables

In this section, we take a glimpse at types of variables available in MATLAB. Hence, it serves the sole purpose of characterizing these types from a global perspective for someone with no programming experience at all. A detailed discussion of how to define and work with these types of variables is provided in Chapter 4. MATLAB distinguishes between four different types of variables which has important implications for the way they are internally processed: numeric, (character) string, cell, and structure variables.

Numeric variables can take on real or complex values. Furthermore, they can be represented as scalars, vectors, or matrices.

String variables are strings of characters. They are usually used for labeling output or plots and for supplying function names as input arguments to other functions.

Cell variables are considered to be a major remedy to the problem that it is not possible to put different types of variables into a single variable (while preserving their very nature) or that there may be difficulties when putting different character strings into a single entity. Thus, cells can be thought of as data containers admitting different types of variables.

Structure variables resemble cell variables. However, they are easier to deal with and seem to be strongly preferred by most MATLAB users. Structure variables are predominantly used as input and output arguments of functions, since they are able to transport all sorts of data.

4 Types of Variables

This chapter takes a closer look at the aforementioned variable types of Section 3.2, i.e., how they are entered into the workspace and how they are displayed in the *Command Window*.

4.1 Numeric Variables

MATLAB (***MAT****rix* ***LAB****oratory*) is a matrix-based programming language. This follows from the fact that matrices are the fundamental objects in MATLAB. A matrix is rectangular double-data array. Keeping this in mind, we will now learn how to define matrices.

For example, suppose that you want to read in the scalar variable

$$A = 2$$

in the MATLAB workspace. This can be interpreted as a (1×1)–matrix. This can easily be accomplish by typing

```
>> A = 2
```

and pressing the Return–key. Notice that an equality sign is interpreted by MATLAB as an assignment operator, that is, the expressions or commands on the right-hand side of the equality are assigned to the variable on the left-hand side.[1] For this to work out, every variable on the right-hand side of an equality sign must be defined. If this is done, a variable can be assigned to another or new variable. For example,

```
>> b = A
```

assigns the value of variable **A** to the newly created variable **b**. Next, observe what the following line does:

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

Obviously, the result looks like a (2×2)–matrix. From this, we may be tempted to infer that the last command turned the scalar **A** into a matrix, but this is not really correct. In general, the command **A(i,j) = x** assigns the value **x** to the element which is positioned in the $\mathbf{i}^{\text{th}}$ row of the $\mathbf{j}^{\text{th}}$ column of matrix **A**. If this element lies outside of the current dimensions of **A**, MATLAB automatically creates the respective rows and columns. Elements of these rows and columns, which have not been assigned to any specific values, are then filled up with zeros. This explains why, in the above example, the off–diagonal elements equal to zero. Thus, the reason, we did not get an error message when switching from a scalar to a matrix, is that MATLAB implicitly defines numeric variables as matrices. Extending **A** by assigning the value 1 to the element in the 3^{rd} row of the 2^{nd} column, transforms **A** to a (3×2)–matrix.

[1] In mathematics, this corresponds to expressions like $A := 2$ or $A \equiv 2$.

```
>> A(3,2) = 1
```

The same syntax can be used to retrieve information from a numeric array. For example, if you want MATLAB to display the third element of the second column of **A**, type

```
>> A(3,2)
```

Note that we left out the assignment operator (**=**) and thereby got MATLAB to return the corresponding element of **A**. More advanced techniques for handling matrices are presented in Chapter 6. Next, suppose you want to define the (3×3)–matrix

$$\boldsymbol{A} = \begin{bmatrix} 2 & 0 & 1 \\ 1 & 3 & 3 \\ -2 & 1 & 0 \end{bmatrix} .$$

Although you could proceed as before by assigning each matrix entry to the desired value, it quickly becomes quite cumbersome to address each and every single element. A more direct approach is to use one of the following syntaxes:

```
>> A = [2 0 1; 1 3 3; -2 1 0]
```

or

```
>> A = [2, 0, 1; 1, 3, 3; -2, 1, 0]
```

Note, as a general rule, that different rows of a matrix are separated by a semi-colon (**;**), whereas two elements within the same row can (but must not) be separated by a comma (**,**). Likewise, the (3×1)–column vector,

$$\boldsymbol{a} = \begin{bmatrix} 3 \\ 2 \\ 5 \end{bmatrix} ,$$

is generated by typing

```
>> a = [3; 2; 5]
```

4.2 String Variables

String variables represent series of characters enclosed in apostrophes. For example,

```
>> s = 'This is a string!'
```

assigns the sentence "This is a string!" to the variable **s**, thereby implicitly defining it as a string variable. Like numeric variables, strings can be packed into a matrix. Try

```
>> s = ['abc' 'cba']
```

and

```
>> s = ['abc'; 'cba']
```

For retrieving the second letter in the third column, type

```
>> s(2,3)
```

Note, however, that strings, which are packed into arrays, must have conformable dimensions. For example,

```
>> s = ['abc'; 'cb']
```

does not work, since the string in the first row consists of three letters (**abc**), whereas the second string consists of two letters (**cb**) only.

4.3 Cell Variables

A way of incorporating strings of different dimensions into a single variable is to use cell variables. For example,

```
>> c = {'abc'; 'cb'}
```

accomplishes this task. Notice that curled brackets are exclusively reserved for operations involving cell variables. Using the syntax

```
>> c{2}
```

lets us retrieve the element in the second row, because **c** is essentially a cell vector. Furthermore, it is possible to put numeric and string variables into a cell variable like in

```
>> c = {'abc'; 'cb'; 69}
```

This feature explains why cells are often referred to as data containers. However, we encounter a similar snag, as we did in the previous section, if we try

```
>> c = {'abc'; 'cb' 69}
```

4.4 Structure Variables

Finally, consider the example of setting up a structure variable which should incorporate information on artificial data from the US stock market. Let us name this structure variable **usa**. Individual components of the structure variable **usa** are addressed by simple name extensions, i.e., *structure fields*. For example,

```
>> usa.date = 'Oct. 2004'
```

assigns a string variable to the structure field **date** of the structure variable **usa**. Likewise,

```
>> usa.sp500 = 7000
```

and

```
>> usa.nasdaq = 5000
```

include scalars to the fields **sp500** and **nasdaq**. Lastly,

```
>> usa.correlation_matrix = [1 0.6; 0.6 1]
```

assigns a matrix to the field **correlation_matrix**. Once you have created the structure variable by defining the first field, it will be automatically updated with every consecutively entered field. Typing

```
>> usa
```

provides you the information on the components of the structure variable **usa**. In order to retrieve the information of a particular field of the structure variable like, for example, the date or the correlation matrix, simply type

```
>> usa.date
```

or

```
>> usa.correlation_matrix
```

respectively. Note that structure variables allow us to store variables of different types (numeric, string, cell, structure) and dimensions (scalar, matrix) in any arbitrary order.

5 Useful Gimmicks

In this chapter, we will see how to retrieve specific information from within the *Command Window*. Additionally, some very useful commands are presented which make programming life much easier.

5.1 Changing the Current Working Directory

One problem that often occurs when using data in different directories can be described as follows. If your data do not reside within the *current working directory*, it cannot be loaded and manipulated. There are two solutions to this problem: (a) telling MATLAB the path of the directory where the data reside or (b) "jumping" to the respective directory by changing the current working directory.

In order to apply the second solution, we need to know what the current working directory is and how to change it. To get the path of the **p**resent **w**orking **d**irectory, type

```
>> pwd
```

Alternatively, type

```
>> cd
```

in order to retrieve the **c**urrent **d**irectory.

For moving the current working directory one directory upwards, type

```
>> cd ..
```

If you want MATLAB to display all directories and files contained in the current directory, type **`dir`** (directory) or **`ls`** (list).[2] Initially, the current working directory was the *work* directory (default setting). We can now "jump" back to the *work* directory by changing the current working directory to the *work* directory:

```
>> cd work
```

Check whether you were successful!

[2]Files have a corresponding extension, directories don't.

5.2 Retrieving Information from the Workspace

After having defined b, $\boldsymbol{a}$, and $\boldsymbol{A}$, we expect MATLAB to know these variables. This can be checked by typing

```
>> whos
```

which gives the names, dimensions, storage requirements, and types of variables in the MATLAB workspace memory. Note that this corresponds exactly to the information we lose when closing the window which contains the workspace variables.

5.3 Clearing the Workspace

If you want to delete, say, matrix $\boldsymbol{A}$ from the workspace memory, you can accomplish this by using the **clear**-command

```
>> clear A
```

or if you want to delete all variables being currently in the workspace, use

```
>> clear all
```

5.4 Getting Help from the Command Window

If you want to know more about what a certain command does and how it is used, you can invoke the **help**–command. Type

```
>> help whos
```

or

```
>> help clear
```

in order to learn more about these two commands.

When getting started with MATLAB, one usually does not know any commands for, e.g., inverting a square matrix. In these cases, you can use the **lookfor**–command which searches MATLAB files for a certain buzzword. For example, to find the command that computes the inverse of a square matrix, type

```
>> lookfor inverse
```

What should come up, among other things, is the line

```
INV    Matrix inverse.
```

so we know that the **inv**–command is what we are looking for. Use

```
>> help inv
```

to check it and to see how it is implemented.

5.5 Scrolling Back the Command History

Since we have deleted all variables from the workspace, we would have to define matrix $\boldsymbol{A}$ again in the usual way. However, this can be elegantly circumvented by pressing the ↑–key on your keyboard. By doing so, we scroll back the command history, i.e., the list of all past commands. You should stop when

```
>> A = [2 0 1; 1 3 3; -2 1 0]
```

re-appears and press the Return–key. Do the same for vector $\boldsymbol{a}$! Check with **whos** to see whether you were successful! Note that this corresponds exactly to the information we lose when closing the *Command History* window .

5.6 Clearing the Command Window

There may be occasions, where the *Command Window* is overcrowded with redundant code and output. In such instances, you can **cl**ear the **c**ommand window by typing

```
>> clc
```

5.7 Determining the Dimension of Arrays

When writing a program, one is often confronted with the problem of setting up a matrix whose dimension should conform to the dimension of an already existing matrix with possibly changing or unknown dimension. For these case, the **size**-command turns out to be extremely useful. For example, type

```
>> size(A)
```

in order to determine the dimension of matrix $\boldsymbol{A}$. Using **help size** shows that **size** returns a (1×2)-row vector whose first element is the number of rows and whose second element is the number of columns.

6 Matrix Manipulation

In the last chapter, we first introduce commands for the efficient generation of special, but often needed matrices. Next, we show how to transform matrices and to conduct algebraic matrix computations.

6.1 Special Arrays

There are special commands for generating frequently used matrices, which have easy structures, so that you do not need to define them explicitly. For example, the **eye**-command creates an identity matrix. The **zeros**– or **ones**–command create arrays consisting, solely, of 0's or 1's, respectively. Use the **help**–command to learn how these commands are invoked.

MATLAB knows empty arrays, i.e., matrices with dimension (0×0). They are invoked by applying the **[]**–operator. For example, in order to turn vector $\boldsymbol{a}$ into an empty array, type

```
>> a = []
```

Check the dimensions of $\boldsymbol{a}$ by **whos** and **size**!

• Exercise 1 •

(a) Create a (3×1)-null vector!
(b) Create a (3×1)-unity vector!
(c) Create a (3×3)-matrix consisting solely of zeros!
(d) Create a (3×3)-matrix consisting solely of ones!
(e) Create a four-dimensional identity matrix!

6.2 Partitioning and Merging Matrices

As already seen in Section 4.1, the syntax for assigning values to a matrix can also be used to extract values of a matrix. For example, assume that we want to extract the element positioned in the first row of the first column of $\boldsymbol{A}$. Then, type

```
>> A(1,1)
```

Similarly, for returning the element of the second row of the third column of $\boldsymbol{A}$, type

```
>> A(2,3)
```

If this value is going to be used in future computations, it should be assigned to a specific variable. To this end, you can simply assign **A(2,3)** to a new variable, say, **b** as in Section 4.1:

```
>> b = A(2,3)
```

In order to extract a whole row or column, use the colon notation (**:**). For example, to extract the first row $\boldsymbol{A}$, type

```
>> A(1,:)
```

This expression instructs MATLAB to take the first row of **A** and extract all of this row's elements. To extract the second column of $\boldsymbol{A}$, type

```
>> A(:,2)
```

This expression instructs MATLAB to take the second column of **A** and extract all of this column's elements. To extract the first two elements of the third column of $\boldsymbol{A}$, type

```
>> A(1:2,3)
```

This should be understood as taking rows 1 to 2 and extracting the third elements of these rows. Put differently, this command extracts the elements lying in the intersection of the indexed rows and columns. There is another important application of the (**:**)–operator.

Suppose you want to compute the vectorization of $\boldsymbol{A}$,

$$\text{vec}[\boldsymbol{A}] = \begin{bmatrix} 2 \\ 1 \\ -2 \\ 0 \\ 3 \\ 1 \\ 1 \\ 3 \\ 0 \end{bmatrix},$$

that is, stacking all columns of matrix $\boldsymbol{A}$.[3] In order to vectorize $\boldsymbol{A}$, type

```
>> A(:)
```

Sometimes one has to address the last row or column of a matrix, but does not know its dimension. In this instance, the last row of a matrix can be extracted by

```
>> A(end,:)
```

Sometimes the result of a command is a high-dimensional matrix, e.g., (100×100). In such cases, we can suppress the result from being printed in the *Command Window* by putting a semi-colon at the end of the command.

There are situations where one wants to merge existing matrices in order to set up a new matrix. This is done by defining a new matrix array and placing the existing matrices in the desired order (*concatenation* or merging). For example, look at the results of

```
>> [A ones(size(A))]
```

and

```
>> [A; ones(size(A))]
```

The **ones**-command can be very convenient for copying rows or columns of a matrix. For example,

```
>> A(:,ones(1,2))
```

[3] Cf. Magnus and Neudecker (1999), p.30.

duplicates the first column of **A**. Similarly,

```
>> A(:,ones(1,4))
```

or

```
>> A(:,ones(2))
```

puts 4 (horizontally concatenated) copies of the first column of $\boldsymbol{A}$ into a new matrix.

• Exercise 2 •

(a) Extract the upper left-hand sub-matrix

$$\begin{bmatrix} 2 & 0 \\ 1 & 3 \end{bmatrix}$$

from matrix $\boldsymbol{A}$!

(b) Generate matrix

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 3 \\ -2 & 1 & 0 \end{bmatrix}$$

out of matrix $\boldsymbol{A}$ using only one command!

(c) Delete the third row of matrix $\boldsymbol{A}$! (Hint: Use an empty array by invoking the **[]**–operator.)

(d) Extract the first and the third columns of matrix $\boldsymbol{A}$ simultaneously! (Hint: Try to find an appropriate indexing vector for addressing the respective columns.)

(e) Bring the original matrix $\boldsymbol{A}$ back into the MATLAB workspace and duplicate the second row of matrix $\boldsymbol{A}$ so that it constitutes a (2×3)-matrix!

(f) Duplicate

$$\begin{bmatrix} 2 & 0 \\ 1 & 3 \end{bmatrix}$$

from matrix $\boldsymbol{A}$ so that it constitutes a (2×4)–matrix, that is, concatenate it horizontally! Alternatively, consider the **reshape**– and **repmat**-commands! Which one can be used to replicate the desired result? Why?

(g) Generate matrix

$$\boldsymbol{A} = \begin{bmatrix} 100 & 0 & 100 \\ 1 & 3 & 3 \\ 100 & 1 & 100 \end{bmatrix}$$

out of matrix $\boldsymbol{A}$ using only one command! (Hint: Indexation works like an intersection operation.)

6.3 Matrix Algebra

Concerning matrix algebra, we will mainly deal with matrix addition (`+`), subtraction (`-`), and multiplication (`*`). Remember that you must be cautious about the matrices' dimensions.

Other useful operations are the so-called element-by-element multiplication[4] (`.*`) and division (`./`). Assuming that the dimensions of the matrices are the same, each element of the first matrix is multiplied or divided by the corresponding element of the second one, respectively. Furthermore, standard scalar exponentiation (`^`) can be extended to exponentiating each element of an array (`.^`).

• Exercise 3 •

Define

$$\boldsymbol{X} = \begin{bmatrix} 2 & 0 & 1 \\ 1 & 3 & 3 \\ -2 & 1 & 0 \end{bmatrix}, \quad \boldsymbol{Z} = \begin{bmatrix} 3 & 2 \\ -1 & 3 \\ 0 & -1 \end{bmatrix}, \quad \boldsymbol{y} = \begin{bmatrix} 3 \\ 2 \\ 5 \end{bmatrix}.$$

and compute (but think about what you are doing):

(a) $2\boldsymbol{X}$,

(b) $\boldsymbol{X}\boldsymbol{Z}$,

(c) $\boldsymbol{Z}'\boldsymbol{Z}$ (Hint: type a prime (`'`) sign after a matrix to generate its transpose),

(d) $\boldsymbol{Z}\boldsymbol{Z}'$,

(e) $\boldsymbol{X}\boldsymbol{Z}'$,

(f) $\boldsymbol{y}\boldsymbol{Z}$,

(g) $(\boldsymbol{X}'\boldsymbol{X})^{-1}\boldsymbol{X}'\boldsymbol{y}$ (Hint: Use the `inv`–command to compute a matrix's inverse), and

(h) concatenate $\boldsymbol{y}$ and $\boldsymbol{Z}$ horizontally to yield a (3×3)–matrix and conduct an element-by-element multiplication with matrix $\boldsymbol{X}$.

7 Miscellaneous

Of course, before you can start to program, you need to know a minimum amount of commands which may serve as basic ingredients. To this end, this chapter shortly introduces some commands an econometrician might often need to call upon. However, this overview is, again, very selective.

[4] This operation is also known as the *Hadamard Product*. Cf. Magnus and Neudecker (1999), p.45.

7.1 Random Number Generator

New simulation techniques in econometrics and financial engineering heavily rely upon the generation of random numbers. The basic version of MATLAB contains two different random number generators. The **rand**–command generates uniformly distributed random numbers, whereas the **randn**–command generates standard normal random numbers. This might seem to be a quite strong restriction, but notice that you could look up a book on simulation methods, which usually describes how to generate almost any random number out of uniform random numbers, or you may purchase MATLAB's *Statistics Toolbox*, which contains a wide variety of random number generators.

In order to generate a sample of standard normal variables, i.e., realizations of a random variable with mean $\mu = 0$ and variance $\sigma^2 = 1$, use the **randn**–command which creates an array made up of realizations of a standard normal variable. For example, by typing

```
>> x = randn(100,100);
```

you generate a (100×100)–matrix of realizations drawn from the standard normal distribution and assign it to the new variable **x**. As displaying such a large matrix does not provide us with much useful insight, we suppress its print-out. Instead, it might be more useful to take a look at a smaller part of **x**. This can be accomplished by typing

```
>> x(1:10,1:2)
```

for example, if you want MATLAB to display the elements positioned in the upper left (10×2)–array of **x**. Compare your realizations with those of your classmates. As expected, you will find out that all samples differ. Moreover, by generating and printing some more samples with

```
>> x = randn(100,100); x(1:10,1:2)
```

we can notice that every repetition produces a different set of realizations.

However, there may be occasions where you want to control for the randomness in simulations.[5] This can easily be done as computers cannot generate realizations of truly random variables. Computers generate *pseudo-random variables* in a deterministic way. Think of a sine curve where the abscissa represents the number of draws. Then, it is easy to see that functional values re-occur, if the number of draws is sufficiently high.

[5]Such a situation could be a Monte Carlo simulation where changing sample sizes and parameters affect simulated statistics. In order to check for the correctness of the program or the impact of changing sample sizes and parameters, one may want to eliminate the effects of randomness.

But, of course, the cycles of pseudo-random number generators are much more complex and chaotic than a simple sine function, and their frequency is extremely low. In order to obtain the same sample of realizations, we have to fix the starting point (*seed*) in the pseudo-random number cycle. For example, let us fix the seed at position 23 with MATLAB's **state**–option[6] and generate **x** again

```
>> randn('state',23); x=randn(100,100); x(1:10,1:2)
```

Re-run your commands and compare what your classmates obtain!

• Exercise 4 •

(a) Fix the seed at **state=23** and generate a sample of the random variable $X \sim \mathrm{N}(\mu, \sigma^2)$ with $\mu = -0.5$, $\sigma^2 = 1.5625$, and sample size $n = 15$!

(b) Fix the seed at **state=23** and generate a sample of the random variable $X \sim \mathrm{Unif}(a, b)$ with $\mathrm{supp}(X) = [5, 10]$ and sample size $n = 25$!

7.2 Saving and Loading Data

Saving data to disk is almost inevitable. Moreover, when doing applied research with real-world data, it is necessary to load-in external data to the MATLAB workspace for econometric analysis. Hence, this section shows how to save and load data to disk using two different file formats: *.mat*-files and *.xls*-files.

Files in the *.mat*–format are more natural to MATLAB, since *.mat*–files optimize processing speed. As an illustration, assume that you want to simulate 1000 realizations of a white noise process drawn from the standard normal distribution and to save the resulting sample as an *.mat*–file in the current working directory. First, check the content of the current working directory by inspecting the corresponding sub-window or typing

```
>> ls
```

Presumably, the current working directory is empty, but at least there should be no file named "econ.mat". Now, generate 1000 realizations of the standard normal distribution and assign it to the variable **x**:

```
>> randn('state',23); x=randn(1000,1);
```

[6]It is still possible to initiate the seed by MATLAB's **seed**–option, however, this is not efficient and should not be used in its current version.

In order to save **x** to an *.mat*-file named "econ", simply type

```
>> save econ x
```

and, finally, clear the workspace:

```
>> clear all
```

Now, verify with the **whos**– and **ls**–commands that your sample vector **x** has been deleted from the workspace, but that it is now saved as "econ.mat" to the current working directory. Alternatively, you could look up the *Workspace* and *Current Directory* sub-windows. Note that you can provide MATLAB with a list of variable names after the **save**–command that you want to be saved, but the first name is automatically defined to be the name of the file to be created. In order to re-load **x** from "econ.mat", type

```
>> load econ
```

and verify with the **whos**-command or the corresponding sub-window that **x** is back again in the workspace.

The **save**– and **load**–commands also support the *ASCII*-format. For this and other features of these two functions, use the **help**–command. Nevertheless, formats specialized for other software packages like *Excel*-spreadsheets, for example, have their own input/output functions in MATLAB. In order to save **x** to an *.xls*-file named "econ", type

```
>> xlswrite('econ',x)
```

and clear the workspace:

```
>> clear all
```

Verify with the **whos**– and **ls**–commands that your sample vector **x** was erased from the workspace, but that it is now saved as "econ.xls" to the current working directory. Alternatively, you could look up the *Workspace* and *Current Directory* sub-windows. In order to re-load **x** from "econ.xls", type

```
>> x = xlsread('econ');
```

and verify with the **whos**–command or the corresponding sub-window that **x** is back again in the workspace. Note that you have to provide the file name as a string variable (**'econ'**) without file extension to both *.xls*-functions.

7.3 Loops and Branching

There are different types of looping and branching structures depending on the software package you use. The two most important looping structures in MATLAB are the *for*- and the *while*-loop.

A *for*-loop executes a series of commands for a pre-specified number of iterations. In order to introduce *for*-loops consider the problem of summing up all realization of your simulated data in "econ.mat".

```
>> clear all
>> load econ
>> n = length(x);
>> s = 0;
>> for i = 1:1:n
s = s + x(i);
end
>> s
```

The syntax **i = 1:1:n** (or **i = 1:n**) instructs MATLAB to set up a vector starting from **1** with increments **1** up to the number **n** computed by **length(x)**.[7] The **length**–command computes the dimension or length of a vector. We intend to store the value of the sum of **x** in variable **s** which has to be initiated before it can be used in the for-loop. This has to be done because **s** is used in an iterative manner within the loop, i.e., it changes after each iteration, and we have to fix its starting value somewhere before. Of course, the most reasonable choice is to fix **s** at **0**.

In the **for**–line, **i** starts at **1** and increases with unit-increments until it reaches **n**. On every run, the series of commands between the **for**– and the **end**–line are executed. Hence, the loop runs **n**-times, and the expression between the **for**– and **end**–line are implemented **n**-times. On the first run, the first entry of the random sample vector, **x(1)**, is added to **s = 0**. On the second run, **s = s + x(i);** adds the first element of **x**, which is stored in **s**, to the second element of the random sample vector, **x(2)**. On the third run, **s = s + x(i);** adds the sum of the first two element of **x**, stored in **s**, to the third element of the random sample vector, **x(3)**, etc. Consequently, the sum in **s** is updated on each run of the **for**–loop by adding the current element in the random sample vector. At the end of the loop, **s** represents the total sum of all elements of **x**. You can verify your result by invoking the MATLAB built-in function **sum**.

[7] Note that this is equivalent to **i = [1:1:length(x)];** as it produces the same result. However, **i = [1:1:length(x)];** is more consistent because it accentuates that an array is created.

In contrast to *for*-loops, *while*-loops execute a series of statements iteratively for an unspecified number iteration until a specific termination criterion is met. This structure is a very important building block in numerical search algorithms which iteratively search for a solution, but stop when the computed value does not change significantly. Consequently, *while*-loops are much more flexible than *for*-loops but may be computationally more demanding.

It is worthwhile mentioning that looping structures should be avoided whenever possible in MATLAB. MATLAB is much faster when algorithms are based on its built-in functions and/or are *vectorized*. There is no recipe for the latter. It is rather a matter of experience. However, to give a short example of what vectorization in the present context means consider the following code

```
>> unit = ones(size(x));
>> x'*unit
```

which simply computes the sum of **x** as the inner product of **x** and a conformable unit vector. Although this is much more efficient than using a *for*-loop, you are advised to apply MATLAB's **sum**–function as there is no need to define a conformable unit vector.

Branching structures control the flow of computations at certain points of the program. In MATLAB, two very useful branching structures are **if**/**else** and **switch**. For example, assume that you want to trim the data set in **x**, e.g., eliminate all values in **x** that are larger than a pre-specified threshold value, say $\bar{x} = 0.5$, such that

$$x_{trim} = \begin{cases} x & \text{if } x \leqslant \bar{x} \\ 0 & \text{else.} \end{cases}$$

This can be accomplished by

```
>> x_trim = x;
>> for i = 1:1:n
if x(i)>0.5
x_trim(i) = 0;
else
x_trim(i) = x(i);
end
end
>> [x x_trim]
```

Notice, however, that the resulting x_{trim} does not really constitute a trimmed version of our data set, since we have simply replaced values larger than 0.5 by 0. In order to eliminate these elements, we can code

```
>> x_trim = 0;
>> for i = 1:1:n
if x(i)<=0.5
x_trim = [x_trim; x(i)];
end
end
>> x_trim(1) = [];
>> x_trim
```

The solution to this problem is conceptually more cumbersome as, at the beginning, we do not known the dimension of the resulting **x_trim**–array. Thus, we initiate it as a scalar, **x_trim = 0;**, and stack in **x_trim** all values of **x** smaller than or equal to 0.5. In this way, the dimension of **x_trim** increases each time an **x(i)<=0.5** is encountered. This has two disadvantages: Firstly, we have to delete the first element because it simply corresponds to the initialization of **x_trim** used in the first call of **if**. Secondly, increasing the dimension of arrays is computationally very inefficient and should be avoided whenever possible by anticipating the final dimension a priori.

• Exercise 5 •

(a) Compute the sum of the elements in **x** using the *while*-loop!

(b) Compute the absolute value of the random sample in **x** and store this new series in **x_abs**! To this end, use the *if*-branching structure! Compare your result to that of the MATLAB built-in function **abs**!

7.4 Plotting

MATLAB has many (easy-to-use) plotting and graphing devises. The most important is the **plot**–function. The simplest usage of **plot** is

```
>> plot(x)
```

This usage, however, might blur the understanding of how **plot** really works. The **plot**–command creates an (x, y)-graph in the Euclidean plane. To this end, it requires two inputs: a vector for the x-axis and a vector for the y-axis with the conformable

length. The latter is represented by the vector of realizations in **x**. The former must be created. It must be a vector with increasing increment and the same length as **x**. For example, define a (time) vector **t** as

```
>> t = 1:1:length(x);
```

Check the sizes of your variables! Now you can plot your random sample in a graph by

```
>> plot(t,x)
```

Indeed, this looks like a time series plot of a white noise process. Note that we have used a column and a row vector in the **plot**–command. This does not cause any problems as long as the lengths of the vectors are the same, since the assignment for setting up the the ordered pairs (x, y) is obvious. Now furnish your plot by adding a title

```
>> title('Gaussian White Noise Process with \mu=0 and \sigma^2=1')
```

an x-label

```
>> xlabel('t')
```

a y-label

```
>> ylabel('x(t)')
```

a legend

```
>> legend('series x')
```

and a grid

```
>> grid
```

Notice that MATLAB does not open a new figure window when calling **plot** again but automatically erases the old graph from the screen. If you want to keep a current graph and plot a new one into another figure window, you have to open up a second window by typing

```
>> figure(2)
```

Consecutive figure windows must be called accordingly by changing the input argument of **figure**. You should use **close all** at the beginning of your programs in order to close all open figure windows left over from prior computations.

Another useful plotting function is the **subplot**–command which will be illustrated in the next chapter.

8 Programs

8.1 Saving and Running Programs

Before setting up a program, you are advised to think about the organization of your program files. Open the *Windows Explorer* and create a new folder named "econ" where to save your programs.

Your starting point for setting up a program is to create a program file which is called *m-file* in MATLAB. Open the *File*–menu and choose *New*. Then click on *M-file* in order to open up the so-called *MATLAB m-file editor*. Alternatively, you can simply click on the icon that looks like a sheet of paper. Any commands that should be executed by the program should be written into the editor.

This new m-file has to be saved by selecting *Save As...* from the *File*–menu in the m-file editor. Save it as "tutorial.m". To run this program, switch back to the command window, type its name (*tutorial*) and press the Return–key. Each time you modify your program, it should be saved before it is run. Otherwise the old version of your program will be executed. Alternatively, pressing the F5–key in the m-file editor automatically saves and runs the program.

8.2 An Example

As your first program, load-in the data from your "econ.mat" file and transform the white noise series according to the formula

$$y_t = \mu + \sigma x_t$$

in order to simulate four normally distributed series with different means and variances. Finally, use the **`subplot`**–command to plot the four series arranged as in the following table:

$\mu_1 = 0.5$ $\sigma_1^2 = 0.5$	$\mu_1 = 0.5$ $\sigma_2^2 = 1.5$
$\mu_2 = -0.5$ $\sigma_1^2 = 0.5$	$\mu_2 = -0.5$ $\sigma_2^2 = 1.5$

```
 1 - clear all
 2 - close all
 3 - load econ
 4 - mu1 = 0.5;
 5 - mu2 = -0.5;
 6 - sigma21 = 0.5;
 7 - sigma22 = 1.5;
 8 - t = 1:length(x);
 9 - subplot(2,2,1)
10 - y1 = mu1 + sqrt(sigma21)*x;
11 - plot(t,y1)
12 - title('Gaussian Process with \mu=0.5 & \sigma^2=0.5')
13 - subplot(2,2,2)
14 - y2 = mu1 + sqrt(sigma22)*x;
15 - plot(t,y2)
16 - title('Gaussian Process with \mu=0.5 & \sigma^2=1.5')
17 - subplot(2,2,3)
18 - y3 = mu2 + sqrt(sigma21)*x;
19 - plot(t,y3)
20 - title('Gaussian Process with \mu=-0.5 & \sigma^2=0.5')
21 - subplot(2,2,4)
22 - y4 = mu2 + sqrt(sigma22)*x;
23 - plot(t,y4)
24 - title('Gaussian Process with \mu=-0.5 & \sigma^2=1.5')
```

8.3 Paths

However, when trying to execute "tutorial.m", you will probably notice that it does not run. The reason for this is that it will fail, if the newly created folder "econ" is not in the *MATLAB file (search) path*. What you have to do, before starting the program, is to tell MATLAB about the new program directory. Once this is accomplished, MATLAB will recognize all files residing in this directory. Proceed as follows:

(1) Choose *File* ⟶ *Set Path...* from the MATLAB menu bar. A new window opens showing all paths and folders that MATLAB knows.

(2) Next, press the button *Add Folder*, search for the newly generated folder "econ," and mark it by clicking once on it.

(3) Press *OK* and notice how the *MATLAB search path* is updated.

(4) Press *Save* to save this setting and, finally, quit by clicking *Close*.

Now your program should run.

9 Functions

Functions are essential the same as programs, but they differ in one fundamental respect. Programs generate their required input from within, whereas functions require the user to supply them as input variables. Thus, programs are often considered as stand-alone functions.

9.1 An Example

Open a new m-file and save it as "mystatistics.m". Then write in the first lines of your function:

```
1  function [mu, sigma2] = mystatistics(x);
2  %MYSTATISTICS computes the sample mean and variance.
3  %-------------------------------------------------
4  % This function computes the mean and variance
5  % of a vector of random realizations.
6  %=================================================
7  % USAGE:  [mu, sigma2] = mystatistics(x)
8  % OUTPUT: mu = sample mean (scalar)
9  %         sigma2 = sample variance (scalar)
10 % INPUT:  x = data (vector)
11 %=================================================
12 % By Your Name, Date.
```

The buzzword **function** tells MATLAB that this file is not a program but rather a function. The expression right after **function** shows how to call the function. Thus, it is your personal decision how to define the function name, the inputs, and the outputs. Of course, you should choose a function name that does not already exist. Expressions after a **%**–sign will not be interpreted as commands but as comments. You can add comment lines anywhere in an m-file. Comments should be used liberally since, in the course of time, you might not remember anymore what you did in your m-file... and why!

As you might already have guessed, the objective of our function is to compute (or estimate) the mean and variance of a random sample,

$$\hat{\mu} = \frac{1}{N}\sum_{i=1}^{N} x_i \qquad \text{and} \qquad \hat{\sigma}^2 = \frac{1}{N-1}\sum_{i=1}^{N}(x_i - \hat{\mu})^2 \,,$$

which is easily accomplished by

```
13 - n = length(x);% compute sample size
14 - mu = 0;% set initial value for for-loop
15 - for i = 1:1:n
16 -     mu = mu + x(i);
17 - end
18 - mu = mu/n;% sample mean
19 - sigma2 = 0;% set initial value for for-loop
20 - for i = 1:1:n
21 -     sigma2 = sigma2 + (x(i)-mu)^2;
22 - end
23 - sigma2 = sigma2/(n-1);% sample variance
```

Before getting to work with **mystatistics**, type

```
>> help mystatistics
```

in the command window. Don't forget to save **mystatistics**! Also try

```
>> lookfor mean
```

and

```
>> lookfor variance
```

The first help line — the so-called *H1 line* — is the line that the **lookfor**–command looks up and prints if there is a match. Therefore, you should choose a text segment that contains (i) the important buzzword and (ii) a short explanation about what this function does. Next, type

```
>> clear all
>> randn('state',23), x = randn(1000,1);
>> [x_mu,x_var] = mystatistics(x)
```

in the command window. Of course, MATLAB has its own built-in function for calculating sample mean and sample variance. Compare your results by calling

```
>> [x_mu mean(x)]
```

and

```
>> [x_var var(x)]
```

• Exercise 6 •

(a) Write a function named "myfunction" for computing
   - the rank,
   - the eigenvalues, and
   - the principal minors

   of a real symmetric matrix!
   Hint: Look for MATLAB built-in functions that bear the brunt of computing ranks, eigenvalues, and determinants. Note that, for any $(n \times n)$ square matrix $\boldsymbol{A}$, a principal submatrix of $\boldsymbol{A}$ is obtained by deleting corresponding rows and columns:

$$\boldsymbol{A}_{(1\times 1)} = a_{11} \ , \quad \boldsymbol{A}_{(2\times 2)} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \ , \quad \ldots \ , \quad \boldsymbol{A}_{(n\times n)} = \boldsymbol{A} \ .$$

   The determinant of a principal submatrix is called a principal minor.

(b) Test "myfunction" on the following real symmetric matrices:

$$\boldsymbol{A} = \begin{bmatrix} 2 & -1 \\ -1 & 1 \end{bmatrix} \qquad \boldsymbol{B} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \qquad \boldsymbol{C} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

$$\boldsymbol{D} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \\ 3 & 3 & 1 \end{bmatrix} \qquad \boldsymbol{E} = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 1 \\ 2 & 1 & 5 \end{bmatrix} \qquad \boldsymbol{F} = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 1 & 3 & 0 \\ 0 & 3 & 1 & 0 \\ 0 & 0 & 0 & 4 \end{bmatrix} .$$

(c) Can you infer from your results obtained in (b), what the eigenvalues tell you about singularity (rank deficiency) and positive definiteness of a real symmetric matrix.
   Hint: According to Magnus and Neudecker (1999), p.24, a symmetric $(n \times n)$–matrix $\boldsymbol{A}$ is positive definite iff the determinants of all principal minors are positive: $\det\left(\boldsymbol{A}_{(k\times k)}\right) > 0$ for all $k = 1, 2, \ldots, n$.

9.2 Functions as Inputs to Functions

Section 9.1 introduced the general setting of functions and their usage in MATLAB. Moreover, there is a further important application of a function as inputs to other functions. As an example, let us consider the problem of estimating a statistical model for

the Gaussian white noise processes of Section 8.2 by *Maximum Likelihood Estimation* (MLE).

For parametric MLE, we need a reasonable model that comprises all essential characteristics of the Gaussian white noise processes of Section 8.2. Due to the properties of white noise, such a parametric model is statistical equivalent to the problem of estimating the parameters of the underlying distribution from which a random sample is drawn. This follows immediately from the fact that the random variables (on the sample path) of any white noise process are independently and identically distributed (*iid*).[8] If we additionally assume the underlying distribution to be Gaussian, we can apply MLE in order to determine the mean and variance of the underlying normal distribution. To be more precise, we assume that all realizations, x_i, of our random sample of size n are drawn from the same normal distribution, $X_i \sim \mathrm{N}(\mu, \sigma^2)$, such that

$$f(x_i; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{-\frac{(x_i - \mu)^2}{2\sigma^2}\right\} \qquad \forall\, i = 1, \ldots, n\ ,$$

where μ and σ^2 denote the mean and the variance, respectively. For independent random variables, we can decompose the likelihood (marginal density) function in the following way:

$$\begin{aligned} L(\mu, \sigma^2; x_1, \ldots, x_n) &= f(x_1, \ldots, x_n; \mu, \sigma^2) \\ &= f(x_1; \mu, \sigma^2) \cdots f(x_n; \mu, \sigma^2)\ . \end{aligned}$$

This gauges the likelihood that the realizations $x_1, \ldots, x_n$ occur given the presumption that $X_i \sim \mathrm{N}(\mu, \sigma^2)$ which implies that all X_i are identically distributed. This piece of information can be used to derive a more compact formulation of the likelihood function using the product operator,

$$L(\mu, \sigma^2) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{-\frac{(x_i - \mu)^2}{2\sigma^2}\right\}\ ,$$

where we have omitted the x_i's as arguments of L as they are constants. Thus, the only variable parts of the likelihood function are the mean and the variance. Intuitively, the best estimates of μ and σ^2 are those values which maximize the likelihood for the given

[8] The definition of a white noise process requires its increments to be uncorrelated only. Thus, the condition of independent increments appears to be too stringent. However, if the increments of the white noise process are normally distributed, then the increments are not only uncorrelated but even independent.

random sample $(x_1, \ldots, x_n)$ to occur, i.e.,

$$\hat{\mu} = \underset{\mu \in \mathbb{R}}{\operatorname{argmax}} \, L(\mu, \sigma^2)$$
$$\hat{\sigma}^2 = \underset{\sigma^2 \in \mathbb{R}_+}{\operatorname{argmax}} \, L(\mu, \sigma^2) \; .$$

Since differentiating a product is cumbersome, we simplify the computational burden by taking the logarithm of L:

$$\ell(\mu, \sigma^2) \equiv \ln \left\{ L(\mu, \sigma^2) \right\} = -\frac{n}{2} \ln(2\pi\sigma^2) - \frac{1}{2\sigma^2} \sum_{i=1}^{n} (x_i - \mu)^2 \; .$$

Then the the first-order conditions read

$$\frac{\partial}{\partial \mu} \ell(\mu, \sigma^2) \stackrel{!}{=} 0$$
$$\frac{\partial}{\partial \sigma^2} \ell(\mu, \sigma^2) \stackrel{!}{=} 0 \; .$$

Basic calculus shows that the optimal or maximum likelihood estimators have closed forms:

$$\hat{\mu} = \frac{1}{n} \sum_{i=1}^{n} x_i$$
$$\hat{\sigma}^2 = \frac{1}{n} \sum_{i=1}^{n} (x_i - \hat{\mu})^2 \; .$$

Note that the first estimator $\hat{\mu}$ corresponds to the sample mean. The second estimator $\hat{\sigma}^2$ would correspond to the sample variance, if the denominator were $n - 1$ instead of n. Since the sample variance is an unbiased estimator of the true variance, we conclude that the maximum likelihood estimator is a biased estimator of σ^2.

The closed-form expression of the ML estimators could be coded in MATLAB which would provide us with the solution. However, in order to use this example for illustrating how to insert a function into another function, we propose a solution based on one of MATLAB's numerical optimization routines. In addition to the pedagogical effect, numerical optimization is an important tool because there are many situations in econometric analysis where a closed-form solution to first-order conditions does not exist.

We will use the function **`fminsearch`** which searches for the unconstrained minimum of a general, nonlinear objective function. In contrast to many other optimization routines, which are part of the *Optimization Toolbox*, **`fminsearch`** is included in the

baseline MATLAB package. First, type

```
>> help fminsearch
```

in order to learn how it has to be invoked. For the ML problem at hand, the easiest syntax for calling **fminsearch** is

```
[para,fval,exitflag] = fminsearch('loglike',para0,[],x)
```

The first output argument (**para**) contains the solution ($\hat{\mu}$ and $\hat{\sigma}^2$) to our optimization problem. The second output argument (**fval**) contains the value of the log-likelihood function evaluated at $\hat{\mu}$ and $\hat{\sigma}^2$. The third output argument (**exitflag**) is a binary variable taking on either **1** (if convergence were successful) or **0** (if convergence failed).

The first input argument (**'loglike'**) specifies the objective function to be used in the optimization where the name of the function must be supplied as a string variable. For the ML problem at hand, this boils down to the log-likelihood function $\ell(\mu, \sigma^2)$. It is important to note that since **fminsearch** searches for the minimum of an objective function, we have to multiply the log-likelihood function by -1 in order to turn the minimization into a maximization. The second input argument (**para0**) is a vector containing the initial or starting values of the variables ($\hat{\mu}$ and $\hat{\sigma}^2$) over which the optimization takes place. As general note: In numerical optimization, starting values have to be supplied which necessitates that the user knows the admissible parameter space. The third input argument is a structure variable managing the flow and modes of the optimization procedure. However, for this simple problem, there is no need to dwell upon technicalities, so we use an empty array (**[]**) as a placeholder. Then the default setting of **fminsearch** applies. The fourth input argument (**x**) contains (constant) parameters of the objective function. For the ML problem at hand, μ and σ^2 are the variables and $x_1, \ldots, x_n$ are the parameters of the log-likelihood function.

Consequently, we note that since **fminsearch** takes care of the computational burden, all that remains to be done is the coding of the log-likelihood function. Generally, the objective function has to be supplied to the optimization function as a MATLAB function. The most flexible form of a function has been presented in Section 9.1. Accordingly, open an m-file and save it as "loglike.m". This function should be capable of computing the value of $\ell(\mu, \sigma^2)$ for some supplied values of μ and σ^2 and the random sample **x**:

```
 1    function f = loglike(para,x);
 2    %LOGLIKE log-likelihood function of normal density.
 3    %-----------------------------------------------------
 4    % This function computes the log-likelihood function
 5    % for determining the mean and variance of a normally
 6    % distributed iid random sample via MLE.
 7    %=====================================================
 8    % USAGE:  f = loglike(para,x)
 9    % OUTPUT: f = value of log-likelihood function (scalar)
10    % INPUT:  para = (1) mean  and (2) variance (vector)
11    %         x = data (vector)
12    %=====================================================
13    % By Your Name, Date.
14
15    mu = para(1);% 1.element is the mean
16 -  sig2 = para(2);% 2.element is the variance
17 -  n = length(x);% determine sample size
18 -  f = -n/2*log(2*pi*sig2)-sum((x-mu).^2)/(2*sig2);
19 -  f = -f;
```

A purist might protest that the code of the log-likelihood function should read as

```
f = -n/2*log(2*pi*sig2)-sum((x-mu*ones(size(x))).^2)/(2*sig2);
```

since operation **x-mu** in line **19** is not defined. However, in order to make life easier MATLAB processes scalar-vector addition by transforming the scalar into a conformable vector with identical entries.

As an illustration of how to do MLE, we present a program ("mle.m") which uses the data simulated from the standard distribution in "econ.mat" for generating realizations of the Gaussian processes as in Section 8.2. These Gaussian processes are a bit more general than pure white noise processes as their moments are allowed to differ from those of a standard normal random variable. According to Section 8.2, we have four parameter sets to be estimated:

1) $\mu_1 = 0.5$ and $\sigma_1^2 = 0.5$.

2) $\mu_1 = 0.5$ and $\sigma_2^2 = 1.5$.

3) $\mu_2 = -0.5$ and $\sigma_1^2 = 0.5$.

4) $\mu_2 = -0.5$ and $\sigma_2^2 = 1.5$.

```
 1 - clear all
 2 - close all
 3 - load econ
 4
 5   % parameters
 6 - mu1 = 0.5;
 7 - mu2 = -0.5;
 8 - sigma21 = 0.5;
 9 - sigma22 = 1.5;
10
11   % starting values
12 - para0 = [0 1];
13
14   % MLE for 1.parameter set
15 - y1 = mu1 + sqrt(sigma21)*x;
16 - [para1,fval1,exitflag1] = fminsearch('loglike',para0,[],y1)
17
18   % MLE for 2.parameter set
19 - y2 = mu1 + sqrt(sigma22)*x;
20 - [para2,fval2,exitflag2] = fminsearch('loglike',para0,[],y2)
21
22   % MLE for 3.parameter set
23 - y3 = mu2 + sqrt(sigma21)*x;
24 - [para3,fval3,exitflag3] = fminsearch('loglike',para0,[],y3)
25
26   % MLE for 4.parameter set
27 - y4 = mu2 + sqrt(sigma22)*x;
28 - [para4,fval4,exitflag4] = fminsearch('loglike',para0,[],y4)
```

• Exercise 7 •

This MLE, we have presented, is rather a constrained maximization problem than an unconstrained one since the variance has to be positive, i.e., $\sigma^2 > 0$. For general constrained optimization problems, MATLAB's **fmincon** is a powerful procedure:

```
[para,fval,exitflag] = fmincon('loglike',para0,[],[],[],[],lb,ub,[],[],x)
```

(a) Analyze the usage of **fmincon** and how to implement the bounds $-\infty < \mu < \infty$ and $0 < \sigma^2 < \infty$!

Hint: The absolute value of the smallest possible number in MATLAB is **eps**, whereas the largest possible number is **inf**.

(b) Write a program that estimates the parameters of the above example by constrained MLE!

• Exercise 8 •

A kernel density estimator is a "smoothed" histogram and is defined as

$$\hat{f}_h(x) = \frac{1}{nh} \sum_{i=1}^{n} K\left(\frac{x - X_i}{h}\right) ,$$

where $K(\cdot)$ is a kernel function and h is the bandwidth parameter that controls for the degree of smoothing. If the underlying density $f(x)$ is normal with $\mathrm{N}(\mu, \sigma^2)$ and $K(\cdot)$ is the Gaussian kernel, i.e.,

$$K(u) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{u^2}{2}\right) ,$$

then $h = 1.059\sigma n^{-1/5}$ is the optimal choice which minimizes the *integrated mean squared error*. See Silverman (1986), Wand and Jones (1994), and Wasserman (2006) for more details.

(a) Load "data.mat" and use **hist** to visualize the absolute frequency of the data!

Hint: The default setting of **hist** produces an oversmoothed histogram. A better choice for the number of bins is $0.1n$.

(b) Code a function which implements the kernel density estimator!

Hint: The function should have three input arguments: a vector containing the data, a vector of grid points, and the bandwidth parameter.

(c) Plot the kernel density estimate of "data.mat" and compare it to a plot of a normal density that is adapted to "data.mat"! What are the implications for MLE based on the normality assumption?

Hint: Use **linspace** to generate the vector of grid points and **normpdf** to generate the graph of a normal density. For a normal density, the notion of *adaptation* means that the first two moments of the normal density and the first two moments of the density to which it is adapted should be equal.

References

Brandimarte, P. (2002). *Numerical Methods in Finance: A MATLAB-Based Introduction*. Wiley, New York.

Coombes, K. R., Hunt, B. R., Lipsman, R. L., Osborn, J. E. and Stuck, G. J. (2000). *Differential Equations with MATLAB*. Wiley, New York.

Favero, C. A. (2001). *Applied Macroeconometrics*. Oxford University Press, New York.

Hanselman, D. C. and Littlefield, B. L. (2004). *Mastering MATLAB 7*. Prentice Hall, Upper Saddle River.

Ljungqvist, L. and Sargent, T. J. (2000). *Recursive Macroeconomic Theory*. MIT Press, Cambridge.

Magnus, J. R. and Neudecker, H. (1999). *Matrix Differential Calculus with Applications in Statistics and Econometrics*. Revised ed. Wiley, Chichester.

Marchand, P. and Holland, O. T. (2003). *Graphics and GUIs with MATLAB*. 3rd ed. Chapman & Hall/CRC, Boca Raton.

Martinez, W. L. and Martinez, A. R. (2002). *Computational Statistics Handbook with MATLAB*. Chapman & Hall/CRC, Boca Raton.

Miranda, M. J. and Fackler, P. L. (2002). *Applied Computational Economics and Finance*. MIT Press, Cambridge.

Morgan, B. J. T. (2000). *Applied Stochastic Modelling*. Arnold, London.

Silverman, B. W. (1986). *Density Estimation for Statistics and Data Analysis*. Chapman & Hall/CRC, Boca Raton.

Venkataraman, P. (2002). *Applied Optimization with MATLAB Programming*. Wiley, New York.

Wand, M. P. and Jones, M. C. (1994). *Kernel Smoothing*. Chapman & Hall/CRC, Boca Raton.

Wasserman, L. (2006). *All of Nonparametric Statistics*. Springer, New York.