An introduction to MATLAB for financial engineering

Tutorial: Part I

Hugo Lamarre

HEC montreal

1 Introduction

1.1 Tutorial

This tutorial is the first part of two. Both parts should roughly hold into two standard three hours courses. Exercises and examples are oriented toward financial engineering. Nevertheless, the underlying theory is not attached to any particular field. Since the exercises build on each other sequentially, we *strongly* suggest going through the entire tutorial in the proposed order. In the first tutorial, we will present basic programming concepts, including the MATLAB environment (workspace), assignation of variables, linear algebra & vectorization, conditional statements and loops. The second part will be a more advanced introduction to MATLAB. We will present how to plot figures, perform basic optimizations and regressions, import/export data sets and handle errors.

1.2 Target audience

This tutorial should interest graduate students from any fields currently studying financial engineering at the master level who either

- have no programming experience whatsoever;
- lack higher-level programming skills, but might have some very basic experience with lower-lever language (e.g. C, FORTRAN,...);
- have some knowledge in other high-level languages (e.g. Visual Basic, R, ...) , but need an introduction on the specific MATLAB syntax.

This is a *basic* introduction to MATLAB *and* programming. To keep the more advanced readers interested, some programming advices and tips regarding style and robustness were included.

2 What is MATLAB?

2.1 High-level v. Low-level

MATLAB is a high-level programming language, whereas C, for example, is a low-level language. The closer is a language from *CPU* language (i.e. the most

basic language your computer speaks before 0s & 1s), the *lower* we say it is. Usually, this means that a low-level language is cumbersome to work with. For example, C has very poor error handling functionalities, which make debugging C softwares a nightmare to the uninitiated. Generally, very high-level languages, such as MATLAB, automatically handle

- memory allocations;
- variable types (e.g. string, double, matrix, cell, ...);
- number precisions (e.g. double, long double, integer, ...).

The best way to understand high-level v. low-level, is to contemplate the fact that high-level programming languages are themselves programmed in low-level languages. For example, MATLAB was initially coded in FORTRAN, whereas now it is constructed from C++ and Java. To highlight this, let's have a look at ex. 1.

Exercise 1. Open the command window. Type after >>:

```
aa=5.3
```

Solution 1. `aa = 5.3`

In ex. 1, you launched a single MATLAB command. Implicitly, MATLAB interpreted the command line $aa = 5.3$ by

- creating a variable *aa* with an associated type (here double);
- allocating the necessary memory;
- associating the value 5.3 to the variable *aa*.

MATLAB is a *live* and *interactive* environment, meaning that you can dynamically manage variables, execute code, open files, etc... We will clarify this in the following subsection.

2.2 The MATLAB environment

Remark 1. We will not go through the graphical interface of MATLAB explicitely, since each MATLAB release is different. Nevertheless, you should be able to locate the following windows:

- Command window;
- Workspace;
- Current Folder;
- Command history;
- Editor;
- Variable editor;
- Help.

Exercise 2. Through the workspace and the variable editor, change the value of aa to 2.3. Go back to the command window and type *aa*.

Solution 2. `aa`

The workspace is the interface from which you can access the MATLAB environment. Every variables are stored in the workspace. It is like a table or desk on which you can lay down information (numbers) written on pieces of paper (variables). You can always output the content of a variable in the command window by typing its variable name (in the previous example, *aa*).

Remark 2. **Identifiers** – Here are a few comments on variable names (or equivalently, *identifiers*):

- Variable names cannot start by numeric symbols, but can contain them (e.g. var12_iable is a correct identifier, but 23variable is not);
- They should be short (e.g. *stocks_that_have_no_dividend_and_no_split* will be very annoying to work with);
- They should intuitively highlight what they contain (e.g. a vector of stock prices could be named *stock_price*);
- Some keywords are reserved by MATLAB. For example, we will see that *for* initiates a specific sequence of operations. As such, one cannot name a variable *for* (this is true for almost any programming languages);
- Also, avoid naming variables the same way as built-in or proprietary functions. For example, if you need to compute a sum of numbers, naming the variable *sum* might be unwise. Indeed, *sum* is already a built-in MATLAB function;
- *Advanced:* MATLAB supports complex numbers. Thus, i and j are already given the value $\sqrt{-1}$. We suggest avoiding the use of i and j for identifiers, especially when dealing with complex numbers.
- *Advanced:* Also, to avoid any confusion, we suggest the use of *1i* as $\sqrt{-1}$, since it is not a valid identifier and thus cannot be redefined by the user.

Because of this interactive environment, we could write complete softwares with one-line commands executed through the command window. We will explore this step-by-step programming approach in ex. 3.

Remark 3. **Basic operators** – When dealing with (real and complex) numbers, you can use the standard operators: * (multiplication), / (division), + (addition), - (substraction) and ^ (power). Note that operators of the type += (to increment in C) or ++ (to increment by 1 in C) are not supported by MATLAB. To increment a variable, simply type *ii=ii+increment*.

Exercise 3. In the command window, type

```
bb=3
```

Now type

```
aa*bb
```

Solution 3.
```
bb=3
aa*bb
```

Remark 4. **Suppressing outputs** – The command window is an output device (for characters only). In ex. 3, you might have noticed that by executing $bb = 3$, the result was printed. In this case, the result is just the assignment of 3 to bb. This does not yield any relevant information! To suppress the output of a line, just add a *semicolon* (;) at the end of it. For example, try typing

```
bb=3;
```

No output should be generated by the command window. Supressing output is a necessity when dealing with large matrices. Type

```
zero_matrix=zeros(100000,5)
```

Such outputs are quite annoying and should be suppressed.

In ex. 3, the value of aa*bb was stored in the generic identifier *ans*, which always hold the last result (or *answer*). Note that this value will be overwritten as soon as another command line without assignment is executed!

Exercise 4. Assign aa*bb to cc. First, let MATLAB output the assignation, then supress it.

Solution 4.
```
cc = aa*bb
cc = aa*bb;
```

Remark 5. **Displaying in the command window** – To display text in the command window, type disp('text to display'). You can also display the content of a variable without outputting its identifier by typing disp(variable_name). Try

```
disp('Hello world');
disp(cc);
```

Compare the output generated from *disp* versus the one generated by typing directly

```
'Hello world'
cc
```

in the command window. You can also clear the command window by typing *clc*. Try it:

```
clc;
```

Remark 6. **Basic mathematical functions** –

- **sqrt**: Computes the square root;
- **exp**: Computes the exponential;
- **log**: Computes the logarithm;
- **ceil**: rounds towards infinity;
- **floor**: rounds towards negative infinity;
- **abs**: takes the absolute value.

Exercise 5. Apply all functions referred in the preceding remark (6) to cc. Apply *abs* to -2.3.

Solution 5.
```
sqrt(cc)
exp(cc)
log(cc)
ceil(cc)
floor(cc)
abs(cc)
abs(-2.3)
```

2.3 MATLAB executables – Scripts and Functions

The fact that MATLAB is very interactive can be an interesting feature for newcomers. Nevertheless, this interactivity can fastly become quite limiting. In fact, to become a skilled MATLAB programmer, we recommend using the interactive features as less as possible. For example, if you see yourself using the variable editor each time you want to modify a variable, you are probably missing out on the most efficient features offered by MATLAB.

Writing lengthy softwares through the command window by step-by-step programming is impracticable. We would prefer to write all the lines in a file and then execute that file once. The script file does just that. It is a text file whose name is ending with the extension *.m*. We call it a *m-file*. It contains a series of command lines that will be executed sequentially. Variables are created and cleared in the workspace as if the command lines were typed one by one in the command window. *To execute a script, simply type the m-file name (without the extension!) from the command window.*

Functions are a different class of m-files, even though they have the same extension (*.m*). The first line of functions differentiates them from scripts:

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

MATLAB automatically assigns files beginning with the word *function* to the function type. A function does *not* interact with the workspace. It has its own distinct memory space. We say that the variables inside a function are local variables. In practice, you should only be interested in retrieving the outputs of a function to the workspace. *To call a function, simply type ouputs=function_name(inputs) in either the command window, a script or another function.* It is recommended to terminate functions with the keyword *end* (even though it is not obligatory). Also, there can only be one function per m-file.

Functions can call other functions, but not other scripts. While scripts can call other scripts and other functions.

Remark 7. **Identifiers for functions** – *The name of the file containing the function has to be the same as the name of the function.* For example, if the function is called *compute_VaR*, then it should be contained in *compute_VaR.m*. Avoid naming functions by names already used by built-in MATLAB functions (e.g. sum). As any identifiers, function names should reflect their use.

Exercise 6. Create a function, named scalar_product, that takes var1 and var2 as inputs. The function should output var3, where var3 is var1*var2. Redefine aa and bb (any real number will do) and call scalar_product(aa,bb) from a script (call the script test.m). Output the result by calling the script from the command window (i.e. type test in the command window). *Tip:* save the files in the default MATLAB root. Which set of variables (aa,bb) or (var1,var2) is local and which lives in the workspace?

Solution 6. **In scalar_product.m**

```
function [var3] = scalar_product(var1, var2)
var3=var1*var2;
end
```

In test.m

```
aa=6.7;
bb=4.2;
scalar_product(aa,bb)
```

In the command window

```
test
```

(var1,var2) are local variables, while (aa,bb) are still in the workspace.

Note that MATLAB automatically returns the variables in your function matching the name of your outputs (e.g. var3 in the preceding exercise). This is in stark contrast with most programming languages, where the use of the keyword *return* is mandatory.

2.4 The path

MATLAB lives in very specific locations on your computer, namely the path. When calling a function, MATLAB will first scan its built-in functions to find a match and then scan the path for proprietary functions.

Remark 8. **Proprietary libraries and toolboxes** – One fundamental idea behind programming is reusability. When writing a specific function, we try to conceptually separate its use from the way it works. We don't really care how a television works, we just want its output! In your financial engineering career, you will have to perform certain computations over and over again.

For example, you might have to compute monthly returns from daily time series in a wide variety of situations. It would be clever to write *once* a generic function to do especially this and reuse it when necessary.

We suggest that you start as soon as possible building your own libraries. *You should maintain your path organized.* For example, you could have a subfolder named *timeseries*, which embeds another subfolder named *returns* that holds functions dealing with returns for time series.

Exercise 7. Create a subfolder named basic_op in your default MATLAB root and move the function scalar_product in this new subfolder. This is the first step towards building your own library! (Obviously, the function *scalar_product* is completly useless. We recommend deleting it after completing this tutorial.)

Solution 7. We suggest using the *Current Folder* window to perform those operations. For more advanced topics, see the functions *cd*, *mkdir* and *matlabroot*.

Exercise 8. Move back to your MATLAB root. Try to call scalar_product(aa,bb). What is the error message? Why does MATLAB can't find your function? Add basic_op to the path by right-clicking on it in the *Current Folder* window. Try to call scalar_product(aa,bb) again.

Solution 8. For more advanced topics, see the functions *addpath* and *genpath*.

3 Help

The goal of this tutorial is not to cover MATLAB exhaustively, this would be an ambitious task to say the least. In fact, it is to give you broad tools to achieve concrete results. As such, we will often *only* refer the reader to a specific function to solve a specific problem. You can then use the MATLAB help to gather further information on the cited functions. *Don't underestimate the MATLAB help, you will see yourself browsing it a lot!*

Exercise 9. Search for the function *clear* in the MATLAB help and find how to clear all the workspace at once through the command window.

Solution 9. The variables aa, bb, cc should have disappeared from the workspace. Verify it!

4 Basic methodology

As mentioned earlier, it is a dangerous path to rely to much on the interactive features of MATLAB. For example, we *strongly* recommend limiting the use of the command window for testing purposes and the use of the variable editor for exploring data. Every relevant results (including graphs) should be produced by scripts calling functions.

The script from which you are generating your end results is called the *main*. It should only contain high-level functions (i.e. functions completing complex tasks). Any reader confronted to your *main* code should be able to quickly grasp its purpose without having to dig further. This is in part why your function names (identifiers) should intuitively reflect the task they are performing. Let's say you want to write a script that computes basic descriptive statistics from a time series, your *main* should look something like this:

```
%Step 1: import the desired time series and perform basic input check
timeseries=import_timeseries_data(file_name);

%Step 2: compute returns from time series
ret=compute_returns( timeseries );

%Step 3: basic (descriptive) statistics
stat=desc_stat(ret);

%Step 4: Sharpe ratio
sharpe=sharpe_rat(ret);

%Step 5: outputting results
plot_stat(stat) %creates graphs from descriptive statistics

disp('The sharpe ratio is ')
disp(sharpe)
```

Please note the use of comments to textualize your *main*.

Even advanced programmers will sometime take shortcuts and perform basic operations in their *main* code. Still, you should be aware that fragmenting codes is a fundamental part of good programming practices.

Remark 9. **clear all** – As exposed earlier, MATLAB scripts interact with the workspace. This means that your current workspace could impact the way your script is executed. Let's say you currently have a variable named *stock* in your workspace (because of previous operations) and you try to execute a script also using an identifier *stock*. If the initialization of the variables are not robust in your script, the two variables might interfere to create subtle issues and/or errors. For this reason, it is part of good coding practices to start every script with a header such as:

```
clear all; clc;
```

Note that you could also disregard scripts and work with functions and local variables *only*. However, the workspace is handy when debugging and exploring data. We thus suggest the use of scripts for writing *mains*.

Remark 10. **Indentation** – For readability, you should always indent your codes. This is relevant when using loops and if-statements. To automatically indent a script or function in the editor, press

```
ctrl-i
```

Remark 11. **Comments** – It is very important to comment as much as possible your functions for future references. To start a comment, type %. The rest of the line won't be executed by MATLAB. Comments should *not* describe in words what the code does. Instead, they should clarify the context. For example,

```
stock_returns = diff(log(stock_price),1);
 %takes the difference of the logarithm of the variable stock_price
```

is a pointless comment. A more useful one would be

```
stock_returns = diff(log(stock_price),1);
 %Computes the log-returns of the stocks
```

In the editor, you can transform the selected text into comments by pressing

```
ctrl-r
```

or you can de-comment it by pressing

```
ctrl-t
```

This is particularly useful when debugging, since it allows you to fastly remove possibly faulty lines.

Remark 12. **Using arrows in the command window** – You can easily navigate to previously called command lines in the command window. To do so, use the *up* and *down* arrows. Try it!

5 Linear Algrebra

MATLAB was initially built with linear algebra in mind. Nowadays, MATLAB is very far from only being used for solving linear algebra problems. Still, vectors and matrices define the way you should approach programming in MATLAB. This is because MATLAB is very efficient at doing computation with matrices, especially through parallel computing.

5.1 Creating matrices or vectors

Matrices are simply array of numbers. Vectors are special cases of matrices that either have only one column or one row. When a vector forms a column (row), we say it is a column-vector (row-vector). Vectors will be useful when working with time series. Matrices or vectors are contained in *brackets* ([ ]). To go to the next column, you can either use a *whitespace* or a *comma* (,). To skip a line (e.g. go to the next row), use the *semicolon* (;). For example,

$$\begin{matrix} 1 & 2 \\ 3 & 4 \end{matrix}$$

is represented as [1 2 ; 3 4] in MATLAB. Note that an expression such as [1 2; 3] is unspecified in MATLAB, since it does not constitute a valid matrix. Try it!

Exercise 10. Create a column-vector named stock_price containing the following values: $56.1, 57.07, 59.3, 55.86, 55.91$.

Solution 10. `stock_price = [56.1; 57.07; 59.3; 55.86; 55.91]`

MATLAB does not distinguish identifiers (variables names) for scalars, vectors and matrices. In other words, any names can be assigned to any variable type (scalar, vector, matrix, string, cell, etc...). Adding a suffix or prefix specifying the variable type can increase the readability of your code. For example, if you have to deal with both a matrix and a vector of stock prices, it might be wise to call the former *stock_price_mat*, and the latter *stock_price_vect*. You could also develop your own convention where matrices are written in upper case and vectors in lower case.

5.2 Operators

There is (roughly) two categories of operators on matrices (and vectors) in MATLAB:

- standard linear algebra operators;
- component-wise operators.

The multiplication (*) and division (\,/) operators are used to perform standard linear algebra operations. To perform a multiplication of matrices, A*B, A and B must satisfy certain size requirements. Namely, A must be k-by-n and B n-by-m (which would yield a k-by-m matrix). By convention, a k-by-n matrix have k rows and n columns.

The division operators are tricky to work with. B/A is the same as BA^{-1} (B must be m-by-n if A is square n-by-n). $A\backslash B$ is the same as $A^{-1}B$ (B must be n-by-m if A is square n-by-n).

Most of the time in financial engineering, we are interested in applying operators component-wise. For example, suppose you have a vector containing daily returns for stock XYZ and another vector containing daily portfolio weights for XYZ. By multiplying the two vectors component-wise, you get the daily return contributions to the portfolio from XYZ. *The component-wise operators are .* and ./ for the multiplication and division respectively.* The dot should not be separated by a whitespace from * (or /). To perform a component-wise (also called elementwise) operation, the two matrices (or vectors) must have the *same size*.

Since in standard linear algebra + and - are already component-wise operators, MATLAB does not define .+ or .-.

The power operator (^) follows the same rules as the multiplication and division. Namely, .^ applies the power operator elementwise, while ^ compute the power of a square matrix. You can only elevate a square matrix (or scalar) to a scalar.

Remark 13. **Transposing** – To transpose a matrix or vector, simply add an apostrophe (') after it. One common mistake made by newcomers is to assume row-vectors are the same as column-vectors. This is not the case.

Exercise 11. Let a=[1 2 3] and b=[1;2;3]. Try a.*b and a+b. Now try a*b. What is the difference between a.*b and a*b? Why does it work in one case and not in the other? Clear a and b.

Solution 11.
```
a=[ 1 2 3 ];
b=[ 1;2;3 ];
a.*b  %error!
a+b %error!
a*b
clear a
clear b
```

a*b is a scalar product, which is defined when a is a row-vector and b a column-vector (ok!). a.*b is a component-wise operations which is undefined since a and b don't have the same size.

Remark 14. **Whoops...** – If you tried to execute the previous solution (11) all at once (either through a script or the command window), you realized that MATLAB stopped the execution before reaching the end. When MATLAB encounters a faulty line of code, it automatically stops and outputs the reason. This is useful for debugging.

Exercise 12. Define A as

$$\begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{matrix}$$

and B as

$$\begin{matrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{matrix}$$

Perform

- $A + B$;
- AB;
- $AB^\top$ (where $^\top$ is standard notation for transpose);
- A^5;
- AB *elementwise*;
- A^5 *elementwise*;
- $\frac{A}{A}$ *elementwise.*

Clear A and B. You should get errors when performing AB and A^5. Why?

Solution 12.
```
A=[1 2 3 ; 4 5 6];
B=[7 8 9; 10 11 12];
A+B
A*B %error!
A*B'
A^5 %error!
A.*B
A.^5
A./A
clear A
clear B
```

Exercise 13. Create a column-vector named *stock_weight* containing the following values: $0.1, 0.4, 0.2, 0.05$. Compute the logarithmic returns associated with *stock_price*. Assign the result to *stock_return*. Try doing this with a single line of code using the functions *diff* and *log*. *diff* returns the difference of consecutive elements in a vector, while *log* returns the logarithm elementwise. Assume the timeseries is ascending (i.e. timestamps are increasing with the index). Output the daily portfolio returns generated by this stock.

Solution 13.
```
stock_weight = [0.1; 0.4; 0.2; 0.05];
stock_return = diff(log(stock_price)); % if increasing timestamps
stock_weight .* stock_return
```

Why must we assume the time series is increasing? *diff* takes the $(k+1)^{th}$ element (for example, tomorrow) and substract from it the k^{th} element (for example, today) of a vector.

Remark 15. **Scalars** – When a scalar is added to (or subtracted from) a matrix, the operation is applied component-wise.

Exercise 14. Add 1 to every stock price in *stock_price* and assign the result to *stock_price*.

Solution 14.
```
stock_price=stock_price+1
```

Remark 16. **Modifying variables** – One must be careful when modifying a variable and re-assigning it to itself (as in ex. 14). The new variable is likely to be conceptually different. Thus, it is part of good coding practices to change its name.

Exercise 15. Redefine *stock_price* as a *row-vector* containing 56.1, 57.07, 59.3, 55.86, 55.91. Modify it so that it becomes a column vector. Add 1 to every stock price in *stock_price* and assign the result to stock_price_p1.

Solution 15.
```
stock_price=[56.1, 57.07, 59.3, 55.86, 55.91];
stock_price=stock_price'; %here the variable is not conceptually different
stock_price_p1=stock_price+1 %here it is conceptually different => renaming it!
```

Here are a few useful linear algebra basic functions:

- **det**: Computes the determinant of a square matrix;
- **inv**: Inverse a square matrix;
- **eig**: Diagonalize a square matrix (eigenvalues and eigenvectors).
- **chol**: Computes the Cholesky factorization of a square matrix. *Advanced:* this is the fastest way to verify if a matrix is positive definite.

Exercise 16. Let A=[1 2 ; 3 4] and C=[1; -1]. Solve for x the following linear problem: Ax=C. First, use the divison operator, then use the *inv* function. Verify your result by performing Ax. *Tip:* you should use the backslash operator.

Solution 16.
```
A=[1 2; 3 4];
C=[1;-1];
x1=A\C;
x2=inv(A)*C;
A*x1
A*x2
```

5.3 Indexing vectors

The i^{th} component of a *vector* can be accessed by typing *vector_name(i)*, where i is an integer. In fact, i can be

- an integer;
- a vector of integer (any length);
- a logical vector (of the same length as *vector_name*).

When passing a vector of integer, a new vector of the same length is created, with each component being the value associated with the corresponding index. MATLAB does not distinguish row-vector from column-vector when indexing.

Remark 17. **Constant increment vectors** – Let's say you want to create a vector with first value 0, last value 3 and with a constant increment of 1 (i.e. [0 1 2 3]). MATLAB offers a very lean syntax to do so: (0:1:3). More generally, you can create constant increment vectors by using: (first value: increment: last value). When *increment* is one, it can be omitted (i.e. (0:3) in the preceding example). This syntax generates row-vectors *only*. Use an apostrophe to transform it into a column-vector if necessary. If *increment* is negative, *first value* must be greater than *last value*.

Exercise 17. Print the two first stock prices (from *stock_price*) using constant increment vectors. Print the index used.

Solution 17.
```
stock_price(1:2)
(1:2)
```

Remark 18. **end** – When indexing, the keyword *end* refers to the last element of the vector. Note that you can do arithmetic with *end*. For example, end-1 refers to the penultimate element. end/2 would also be acceptable if the length of the vector is even (*indices must be integers!*).

Exercise 18. Print the penultimate stock price.

Solution 18.
```
stock_price(end-1)
```

Exercise 19. Output and assign to *stock_disret* the discrete (price) returns associated with stock_price. Use a single line. *Tip:* you should use ./, constant increment vectors, *end* and indexing. First, assume the timestamps are increasing with the index, then the opposite.

Solution 19.
```
%increasing timestamps
stock_disret = stock_price(2:end)./stock_price(1:end-1)-1

%decreasing timestamps
stock_disret = stock_price(1:end-1)./stock_price(2:end)-1
```

Remark 19. **Logical variables** – Logical matrices or vectors only contain zeros and ones. A one correspond to the state TRUE, while a zero correspond to FALSE. To obtain such a matrix, you must first make a statement. The most basic statement operators are > (superior to), < (inferior to), >= (superior or equal to), <= (inferior or equal to), == (equal to) and ∼= (not equal to). There is a fundamental distinction to make between == and =. = is an assignment operator. As such, it performs an "action". == tests a condition (as does ∼=) .

To avoid issues regarding order of operations, we strongly suggest enclosing logical variables in parentheses. We also refer the reader to the is* functions (e.g. *isnan*). We will come back to logical operators in greater details in section 6.

Exercise 20. From the discret returns computed in ex. 19, create a logical vector responding to the following statement: the discrete return is greater than zero. Use that vector to output the discrete returns greater than zero through indexing. Output the index in the command window and compare it to *stock_disret*. Do 1s correspond to returns greater than zero?

Solution 20.
```
index = (stock_disret>0);
stock_disret(index)
index
stock_disret
```

Remark 20. **Errors when indexing** – Trying to retrieve the index zero or an index greater than the length of the vector are common mistakes. An index is an integer *strictly* greater than 0 that must not exceed the length of the vector.

Exercise 21. Try retrieving the *stock_price* indexed by zero. What error message do you get?

Solution 21.
```
Subscript indices must either be real positive
 integers or logicals.
```

Exercise 22. Try retrieving the *stock_price* indexed by end+1. What error message do you get? Why?

Solution 22.
```
Index exceeds matrix dimensions.
```

5.4 Indexing matrices

You can select a single value in a matrix by providing two integers. To retrieve the element at the i^{th} row and j^{th} column, you would write *matrix_name(i,j)*. The first index is referring to the rows, while the second to the columns. Note that they are separated by a comma. You can also select multiple columns or rows by providing vectors of integers.

Remark 21. **Colon to select all** – To select all the rows or columns, you can use the colon (:) operator. For example, let's say that you are interested in the j^{th} column, you would type *matrix_name(:,j)*. Indeed, you wish to retrieve all the rows from the j^{th} column. In fact, in this context, the colon operator is equivalent to (1:end).

Exercise 23. Let A=magic(10). Retreive the element at (6,8). Use the colon operator to retreive the fifth row. Don't use the colon operator to retreive the sixth column.

Solution 23.
```
A=magic(10);
A(6,8)
A(5,:)
A(1:end,6)
```

Logical indexing is also possible. The logical matrix must have the same size as the initial matrix. Since the initial size of the matrix cannot be preserved by logical indexing (think about this statement; why is it the case?), MATLAB will shrink the selected components into a vector.

Exercise 24. Create a matrix of normalized gaussian random returns of size 10 by 10 using the function *randn*. Name it *ret*. The volatility of your stocks should be $0.2/\sqrt{252}$ and the mean $0.06/252$. Using logical indexing, extract all the negative returns and print them. *Tip:* see the function *randn* in the MATLAB help.

Solution 24.
```
ret=0.2/sqrt(252) * randn(10,10) + 0.06/252;
ret_neg=ret(ret<0)
```

Remark 22. **Time series** – It is natural to let rows represent descending or ascending timestamps (for example, days) and columns different stocks (or any type of cross section, e.g. different countries, debt issue, etc...). In this tutorial, we will assume increasing timestamps. Simply put, when you move down a matrix, you get closer to today.

Note how, in the preceding example (24), $(ret < 0)$ is a logical matrix and *ret_neg* is a vector. *We will rarely proceed this way for matrices.* Indeed, by logically indexing matrices, we are losing their initial structures. In the previous example (24), we don't know which returns are associated to which stocks anymore!

Exercise 25. Retrieve the fifth column of *ret* (let's say the returns of XYZ) and assign it to *XYZ_ret*. Compute the mean of *XYZ_ret*. Filter out non-negative returns and compute the standard deviation on remaining returns. This quantity is known as downside deviation in finance. *Tip:* see the function *mean* and *std* in the MATLAB help.

Solution 25.
```
XYZ_ret=ret(:,5);
mean(XYZ_ret)
std(XYZ_ret(XYZ_ret<0))
```

Remark 23. **Colon to shrink matrices to vectors** – To transform a matrix into a vector, use the single colon operator (:). *matrix_name(:)* would return a vector filled by each column of the initial matrix sequentially.

Exercise 26. Transform the *ret* matrix into a vector.

Solution 26.
```
ret(:)
```

Until now, we have used indexing to extract data already in matrices. What if you want to modify data from a matrix? Your indexed matrix must be of the same size as your new data. You can then overwrite the matrix with the new data by typing matrix_name(index)=new_data. *When overwriting an indexed matrix or vector with a scalar, the scalar replaces all the elements at once.* This is particularly useful when indexing with logical matrix to modify data.

Exercise 27. Let A=magic(10). Replace the first column by ones using the function *ones* (see the MATLAB help). Don't forget that you must overwrite the selected column with a vector of the same length. Do the same thing for the second column using a scalar (1) instead of *ones* . Now, try overwriting elements greater than 60 with zeros.

Solution 27.
```
A=magic(10);
A(:,1)=ones(10,1)
A(:,2)=1
A(A>60)=0
```

When assigning a value at an index greater than the initial size of the matrix, MATLAB will automatically expand the matrix to hold the new data. Unspecified values are assigned zeros.

5.5 Basic functions to handle matrices and vectors

Here are a few basic functions that are useful when dealing with vectors and matrices:

- **size**: return the size of a *matrix*;
- **length**: return the length of a *vector*;

– **ones**: create a matrix of ones;
– **zeros**: create a matrix of zeros;
– **diag**: extract the diagonal or create a diagonal matrix;
– **eye**: generates the identity matrix;
– **flipud**: flips a matrix in the up-down direction.

Exercise 28. Let A=magic(10). Extract the diagonal from A and assign it to *diag_tmp*. Is *diag_tmp* a vector or matrix? Create a diagonal matrix with the (i,i)-th element equal to *diag_tmp*(i). What if you changed the identifier *diag_tmp* for *diag*?

Solution 28.
```
A=magic(10);
diag_tmp=diag(A);
diag(diag_tmp) %here, diag is a function.
%The parentheses are used to pass an input.

%Conflict between built-in function and identifier
A=magic(10);
diag=diag(A);
diag(diag) %here, diag is a variable.
%The parentheses are the indexing operators.
```

Exercise 29. Let A=magic(10). Extract the fifth column from A and assign it to b. What is the length of b (use the *length* function)? Extract a submatrix of A, named B, formed by the first 5 rows and first 7 columns. Confirm that the matrix has the proper size (i.e. 5 rows and 7 columns) by using the function *size*. What is the first element returned by size: the number of rows or columns?

Solution 29.
```
A=magic(10);
b=A(:,5);
length(b)
B=A(1:5,1:7)
size(B)
```
In MATLAB, the first dimension is always the vertical one (as in standard linear algebra). Thus, the first element returned by *size* is the length of the vertical dimension, namely the number of rows!

Remark 24. **Functions applying to vectors v. scalars** – Some functions naturally apply to vectors. Here are a few examples:

– **sum**: compute the sum of a vector;
– **cumsum**: compute the cumulative sum of a vector (see also *cumprod* in the help);
– **diff**: compute the difference between consecutive vector elements (higher orders also available);
– **sort**: sort the elements of a vector;
– **max/min**: returns the maximum or minimum element of a vector;

– Basic statistical functions: see **mean, std, skewness** and **kurtosis**.

When passing a matrix as input to these functions, MATLAB must know on which dimension to apply the function (i.e. apply it to the rows or to the columns?). Built-in MATLAB functions usually have a *dim* input to specify this. To apply the function on columns (rows), *dim* must be equal to 1 (2). If the function returns a scalar when passed a vector (e.g. sum), the resulting output will be a vector when passed a matrix. More precisely, if *dim* is 1 (2), the result will be a row (column) -vector.

By default, *dim* is one. In other words, MATLAB applies functions on columns if *dim* is unspecified.

Some other functions, like the basic mathematical functions *sqrt, exp, log, ceil, floor and abs*, naturally applies to scalar. In this case, MATLAB applies the function elementwise.

Refer to the MATLAB help to know how a function behave when passed a matrix.

Exercise 30. Compute the mean of *ret* for each rows (i.e. the mean returns for the cross section of stocks). Is the result a row or column -vector? Do the same for each columns (i.e. the mean return of each stocks).

Solution 30.
```
mean(ret,2) %cross section mean
mean(ret) %stock mean
```
Note how MATLAB applies the function horizontally when *dim* is set to 2. The result is a column-vector, since MATLAB computes one mean per row.

Exercise 31. Compute the cumulative return for the whole period (1 to 4) from *stock_disret* using *cumprod* (see help). Try doing it in one line of code. What is the terminal return?

Solution 31.
```
cumret=cumprod(stock_disret+1,1)-1
cumret(end)
```

Remark 25. **Empty matrices** – The empty matrix has very specific properties in MATLAB. To create it, simply type []. You can test if a matrix is empty by using the function *isempty*. Indexing with the empty matrix will return the empty matrix. You can also delete entries in a matrix by overwriting them with the empty matrix.

Exercise 32. Let A=magic(10). Delete the first column by replacing it with the empty matrix. Try doing the same thing by indexing. Type A([]) (i.e. indexing with the empty matrix). What do you expect to retrieve? Now try deleting the element at (4,5) of A using the empty matrix.

Solution 32.
```
A=magic(10);
A(:,1)=[]
A=magic(10);
A=A(:,2:end)
A([])
A(4,5)=[]; %error!
```

5.6 Concatenation

Often, you will want to glue many vectors together to create a matrix or glue many matrices together to create a larger matrix (or perhaps even glue vectors and matrices together!). This is called "concatenating". The size of the variables must match in the concatenating dimension. For example, if you wish to concatenate a column-vector and a matrix horizontally, the length of the vector should match the number of rows in the matrix. There are two ways to concatenate:

- bracket ([]) operators;
- built-in concatenation functions *vertcat* and *horzcat*.

The brackets are used in the same ways as for building matrices or vectors from numbers. For example, assuming a, b and c are vectors of equal length, you could construct a matrix by typing [a b c]. If you type [a; b; c], you would get a vector. The function *vertcat* (*horzcat*) are used to concatenate vectors or matrices vertically (horizontally). Even though there is no fundamental distinction between the brackets and the built-in functions, we suggest using *vertcat* and *horzcat* for robustness and readability.

Exercise 33. Let a=(1:3), b=(4:6) and c=(7:9). Create a matrix with a,b and c as columns and another one as rows. Use both the bracket operator and the built-in functions *vertcat* and *horzcat*. *Tip:* remember that constant increment vectors are generated as rows.

Solution 33.
```
a=(1:3);
b=(4:6);
c=(7:9);

ascol1=[a' b' c']
ascol2=horzcat(a',b',c')

asrow1=[a;b;c]
asrow2=vertcat(a,b,c)
```

You will sometime wish to concatenate starting from an empty matrix. One interesting feature is that both the built-in concatenation functions and the bracket operators can handle empty matrices. *Advanced:* This might be useful for initiating loops when preallocation is impossible.

Exercise 34. Let A=[] (the empty matrix) and B=[1;2;3]. Try [A B]. Now try, horzcat(A,B).

Solution 34.
```
A=[];
B=[1;2;3];
[A B]
horzcat(A,B)
```

Let's now have a look at error messages you might get when concatenating.

Exercise 35. Let A=[1;2;3] and B=[1 2 3]. Try [A B]. What error message do you get and why?

Solution 35.
```
Error using horzcat

Dimensions of matrices being concatenated are not consistent.
```

5.7 More advanced topics: replicating, sorting and finding

repmat is used to tile and replicate matrices or vectors. It can be very useful for performing elementwise operations between vectors and matrices. The first input is the vector or matrix to replicate, the second is the number of replications to perform in each dimension.

Exercise 36. Using *repmat*, create a matrix containing ten column-vector [1 2 3].

Solution 36.
```
repmat([1;2;3],[1 10])
```

Exercise 37. Advanced: Generate a matrix of gaussian random returns with each column having different volatilities and means. Name it *ret2*. Let the vector of daily volatilities be [0.2 0.18 0.3 0.05 0.55]/$\sqrt{252}$ and of daily means [0.01 0.2 0.2 0.12 0.03]/252. Generate 1000 daily returns. Compute the standard deviations and means to confirm that you generated the correct distribution of returns.

Solution 37.
```
vol = [ 0.2 0.18 0.3 0.05 0.55 ]/sqrt(252);
mean_ret = [ 0.01 0.2 0.2 0.12 0.03 ]/252;
ret2 = randn(1000,5).*repmat(vol,[1000 1]) + repmat(mean_ret,[1000 1]) ;

[mean_ret ; mean(ret2)]
[vol ; std(ret2)]
```

Exercise 38. Compute the sharpe ratio of the returns in *ret2*. Sort the Sharpe ratios in descending order (using sort) and output the best Sharpe ratio. Do the same using the max function.

Solution 38.
```
sharpe = mean(ret2,1)./std(ret2,1);
sharpe_sorted=sort(sharpe',1,'descend');
sharpe_sorted(1)
max(sharpe)
```

sortrows sort the rows of a matrix. It maintain the row structure of a matrix. By default, it will sort the matrix using the first column, but you could specify a column index.

Exercise 39. Let *stock_vol* be a matrix containing the daily price returns and trading volumes of a stock. More precisely, let

```
stock_vol= [ 0.03 10; 0.01 5;  0.14 100;  -0.05 203 ; 0.001 2 ];
```

where the first column are the prices and the second column the volumes. Sort the rows of the matrix according to volume in descending order. *Tip: sortrows* does not support the use of 'descend' or 'ascend' (as *sort*). Instead, you must provide a negative index to set the mode to *descending*.

Solution 39.
```
stock_vol= [ 0.03 10; 0.01 5;  0.14 100;  -0.05 203;  0.001 2];
stock_vol_sorted=sortrows(stock_vol,-2)
```

You might wish to retrieve the numeric index for the elements satisfying a certain statement. The function *find* finds the k^{th} first or last indices satisfying a statement (where k is some given integer).

The input to *find* must be logical. The function will then take care of finding the k^{th} first or last indices corresponding to 1s (TRUEs).

Exercise 40. Find the last volume greater or equal to 100 in *stock_vol*. Remember that volumes are in the second column. With the index returned by *find*, retrieve the corresponding return.

Solution 40.
```
id=find(stock_vol(:,2)>=100,1,'last');
stock_vol(id,1)
```

Remark 26. **Warning regarding *find*** – If you want to find all the elements satisfying a certain statement, you should use logical indexing instead of *find* to improve performance. For example, the following two lines produce identical results:

```
stock_vol(find(stock_vol(:,2)>=100),1)
stock_vol((stock_vol(:,2)>=100),1)
```

but the second one is more efficient *and* readable. *Use find only when interested in the k^{th} first or last indices.*

Remark 27. **Counting with logical vectors** – You will encounter situations where you need to count the number of elements satisfying a certain statement. Even though logical vectors are specific variable types, the number they contain are usual integers (0s and 1s)! As such, they can be summed.

Exercise 41. Compute the percentage of daily negative returns for the stock returns contained in *stock_vol*. Clear all variables from the workspace.

Solution 41.
```
sum(stock_vol(:,1)<0)/length(stock_vol(:,1))
Clear all;
```

6 Statements and for loops

6.1 Logical operators

We already briefly covered logical operators. We will now go into greater details. Remember that >, < , >=,<=, == and ~= can be used to compare variables. When matrices or vectors are compared, they must be of the same size. The comparisons are made elementwise. The negation operator in MATLAB is ~ (as opposed to ! in C/C++). In other words, $\sim 1 = 0$ and $\sim 0 = 1$.

What if you want to test if two variables are not equal? ~= does just that. Equivalently, you could first test for equality using == and then apply the negation operator:

```
~(variable_name==test_variable)
```

In fact, > could also be written as the negation operator and <=. How?

The negation operator is applied elementwise for matrices. If BOOL is a logical matrix, then ~BOOL would be BOOL with all the 1s transformed into 0s and vice-versa.

Exercise 42. Type ~1 and ~0 in the MATLAB command window.

Solution 42.
```
~1
~0
```

Exercise 43. Let A=[1 2 3; 4 5 6; 7 8 9]. Create a logical matrix to test if elements of A are not equal to 5 by using ~=. Print it. Generate the same matrix by using the negation (~) and equality (==) operators only. *This is reminescient of the versatility of MATLAB: there is usually more than one way to perform a certain task. The best one is often a matter of preferences.*

Solution 43.
```
A=[1 2 3; 4 5 6; 7 8 9];
(A~=5)
~(A==5)
```

You might want to test for multiple conditions at the same time. Let's say you have *condition1* and *condition2*. You might ask yourself: is both of them respected? or is at least one of them respected? In the former case, we are referring to a AND-type operator, while in the latter to a OR-type operator. MATLAB proposes two sets of operators:

- & (AND) and | (OR): for elementwise logical operations (i.e. for matrixes, vectors or scalars);
- && (AND) and || (OR): short-circuiting operators for logical scalars only.

Exercise 44. With the same A as in 43, create a logical matrix indexing elements greater than 3, but smaller than 7. Also, create a logical matrix indexing elements equal to 5 or 8.

Solution 44.
```
A=[1 2 3; 4 5 6; 7 8 9];
(A>3 & A<7)
(A==5 | A==8)
```

Remark 28. **Robustness** – For robustness, we strongly suggest using && and || when dealing with scalars. Indeed, if you mistakenly assume a certain variable is a scalar, you will get an error message.

Remark 29. **Advanced: Short-circuiting** – The && and || allow for short-circuit behavior. This means that MATLAB won't go through the whole statement if it can reach a conclusion prematurely. For example, consider (*statement1* && *statement2*). If *statement1* is false, MATLAB does not need to evaluate *statement2* (Why?)! It can already reach a conclusion, namely FALSE. In some situation, using short-circuiting operators is necessary.

Short-circuiting can also improve performance. Indeed, when short-circuiting, try evaluating computationally expensive statements at last, so that they are evaluated only if absolutely necessary.

Exercise 45. Advanced: Let a=[1 2 3 4] and b=[1 2]. Try to generate the following logical scalar (also called *boolean*) (length(a)==length(b)) && (a*b==5) and (length(a)==length(b)) & (a*b==5). Which operator, && or &, produces an error? Why?

Solution 45.
```
a=[ 1 2 3 4 ];
b=[ 1 2 ];
(length(a)==length(b)) && (a*b==5)
(length(a)==length(b)) & (a*b==5)
```
In this situation, you must use short-circuiting operators. Otherwise, MATLAB might try to perform a scalar product on vectors of different length!

If you have multiple variables *of the same size*, you can use the AND and OR operators to jointly test multiple statements on different variables.

Exercise 46. With the same A as in 43 and B=magic(3). Create a logical matrix indexing elements of A greater than 3, but smaller than 7 *or* elements of B equal to 5.

Solution 46.
```
A=[1 2 3; 4 5 6; 7 8 9];
B=magic(3)
(A>3 & A<7) | (B==5)
```

6.2 Conditional statements

The most fundamental building block in programming is the *if*-statement. If a certain statement is true, then MATLAB executes a specific part of code. The basic syntax is:

```
if(statement)
 ... %execute this code if statement is true (i.e. statement is 1)
end
```

If the statement is false, you could wish to perform certain tasks. This is acheived with the *if-else* syntax. If a certain statement is true, then MATLAB executes one part of code. Otherwise, it executes another part of code:

```
if(statement)
 ... %execute this code if statement is true (i.e. statement is 1)
else
...  %execute this code if statement is false (i.e. statement is 0)
end
```

You can embed (nest) multiple *if-else* to give rise to the *elseif* syntax:

```
if(statement_1)
... %execute this code if statement_1 is true
elseif(statement_2)
... %execute this code if statement_2 is true and statement_1 is false

...

elseif(statement_k)
... %execute this code if statement_k is true and all preceding statements are false
else
... %execute this code if all preceding statements are false
end
```

Note how the keyword *end* is used to terminate the *if* syntax. This is very different from its use in indexing.

Exercise 47. Write a program that tosses a coin. First, generate a random number between 0 and 1 using the function *rand*. If the number is between 0 and 1/2, display 'head' using the function *disp*. If it is between 1/2 and 1, display 'tail'. *Tip:* Since there are only two possible events, use the if-else syntax.

Solution 47.

```
coin=rand;

if(coin>0 && coin<1/2)
disp('head');
else
disp('tail');
end
```

Exercise 48. Write a program that tosses a six-sided dice. Use the *elseif* syntax and output the number of dots.

Solution 48.
```
dice=rand;

if(dice>0 && dice<1/6)
disp('1');
elseif (dice>=1/6 && dice<2/6)
disp('2');
elseif (dice>=2/6 && dice<3/6)
disp('3');
elseif (dice>=3/6 && dice<4/6)
disp('4');
elseif (dice>=4/6 && dice<5/6)
disp('5');
else
disp('6');
end
```

When using the *if* or *elseif* syntax, the statements must be *logical scalars* (booleans). To transform logical vectors into logical scalars, two functions are particularly useful:

- **all**: returns 1 if all elements of the vector are 1s.
- **any**: returns 1 if any element of the vector is a 1.

When a matrix is passed, the functions are applied on each columns or rows (as usual, you can change the dimension through *dim*). To test if all or any elements of a matrix satisfy a statement, simply apply the respective functions twice or vectorize the matrix using the colon operator (:).

Exercise 49. Let A=magic(10). If there is any number equal to 42 or 60, display '42 or 60 found'. Code the algorithm using at least two different syntaxes. Use && and || when comparing scalars.

Solution 49.
```
A=magic(10);

%first way
if( any(any(A==42)) || any(any(A==60)) )
disp('42 or 60 found');
end

%second way
if(any(any(A==42 | A==60)))
disp('42 or 60 found');
end

%third way
if any(A(:)==42) || any(A(:)==60)
```

```
disp('42 or 60 found');
end

%fourth way
if any(A(:)==42 | A(:)==60)
disp('42 or 60 found');
end
```

6.3 While

The *while* loop is the simplest loop. The syntax is:

```
while(statement)
... %execute this code while statement is true (i.e. 1)
end
```

For this loop to make sense, the statement must be *dynamic*. In other words, at each iteration the variables tested in *statement* must change! (An iteration is the execution of one instance of the content of a loop.) When MATLAB encounters a *while* loop, it will first test the statement. If it is TRUE, MATLAB enters the loop for the first time. Otherwise, it jumps straight to the keyword *end* and disregard the content of the loop completely. After executing the code once and reaching *end* (i.e. the first iteration), it will test the statement again. If *statement* is TRUE, MATLAB executes the code contained in the *while* loop for the second time (i.e. the second iteration). Otherwise, it jumps to *end*. MATLAB will keep iterating until *statement* is FALSE.

Exercise 50. Generate random numbers between 0 and 1 (see *rand*) until a number greater than 0.99 is generated. Print it.

Solution 50.
```
rand_number=0; %initialization
while(rand_number<=0.99)
rand_number=rand;
end
disp(rand_number);
```

When dealing with loops, we often have to perform an initialization phase. This initialization usually includes the preallocation of variables (more on this in the subsection on *for* loops). In the preceding example (52), *rand_number* has to be smaller than 0.99 for MATLAB to enter the *while* loop. We arbitarly set it to 0. If *rand_number* was not declared before the statement, an error message would be printed.

Remark 30. **Infinite loop** – It is possible to create loops for which *statement* is always true. Usually, this is due to a programming error and is unwanted. In such a case, MATLAB will never exit the loop. To manually stop MATLAB, press

```
ctrl-c
```

Exercise 51. Create an infinite loop by setting *statement* to 1. Display a random number between 0 and 1 (*rand*) at each iteration. Stop MATLAB.

Solution 51.
```
while(1)
disp(rand)
end
```

Exercise 52. Display 1 to 10 in the commance window. To do so, first initiate *ii* to 1. Then, create a *while* loop that keeps iterating until *ii* is smaller or equal to 10. At each iteration, display *ii* and then increment it by 1. What if you first increment *ii* and then display *ii*? Does it produce the same outcome?

Solution 52.
```
%correct way
ii=1;
while(ii<=10)
disp(ii);
ii=ii+1;
end

%wrong way
ii=1;
while(ii<=10)
ii=ii+1;
disp(ii);
end
```

The keyword *break* will force MATLAB to exit a *while* (or *for*) loop. You could use an if-statement with *break* to break a loop if a certain condition is met. We often fix a maximum number of iteration acceptable to avoid infinite loops.

Exercise 53. By defining a variable that counts the number of iteration, make sure that the loop in ex. 51 does not exceed 10000 iterations. Use an if-statement and the keyword *break*.

Solution 53.
```
iter=0;

while(1)

if iter>10000
break;
end

disp(rand)

iter=iter+1;
end
```

You can nest multiple while loops. This is rarely useful. When using *break* in nested loops, MATLAB quits the immediate loop which contains *break*, by jumping to the corresponding *end*.

Exercise 54. Using two nested *while* loops, fill a 10-by-10 matrix where each element is the sum of the vertical and horizontal indices. You should use a similar logical structure as in ex. 52, but with two counting variables: *ii* and *jj*. Indent your code (ctrl-i) for readability. Use the keyword *break* to let elements having both indices (horizontal or vertical) greater than 5 equal to zero.

Solution 54.
```
ii=1;
while(ii<=10)
    jj=1;
    while(jj<=10)
        A(ii,jj)=ii+jj;
        jj=jj+1;
    end
    ii=ii+1;
end
clear A;

%with a break for elements with ii>5 or jj>5
ii=1;
while(ii<=10)
    jj=1;
    while(jj<=10) %break only affect this loop
        if(ii>5 && jj>5)
             break;
        end
        A(ii,jj)=ii+jj;
        jj=jj+1;
    end
    ii=ii+1;
end
```

6.4 for loops

Logical structures presented in exercise 52 and 54 are widely used when programming. More precisely, the logical sequence is:

```
%INITIALIZATION
var=initial_value;

%LOOP
while(ii<=terminal_value)
```

```
... %PERFORM ACTIONS

%INCREMENT
ii=ii+increment;

end
```

MATLAB (as most other programming languages) offers an equivalent and leaner way to execute the same sequence:

```
for(ii=(initial_value:increment:terminal_value))

...PERFORM ACTIONS

end
```

MATLAB will first set *ii* to *initial_value* and go through the first iteration. When the code contained by the *for* loop is executed and *end* is reached, it will increment *ii* by *increment* and go through the second iteration. MATLAB will keep iterating until *terminal_value* is reached. Both set of parentheses (in the *for* loop syntax) are *not* mandatory. Nevertheless, they could increase readability.

Exercise 55. Display 1 to 10 in the command window using the *for* loop.

Solution 55.
```
%with two sets of parentheses
for(ii=(1:10))
disp(ii);
end

%with one set of parentheses
for(ii=1:10)
disp(ii);
end

%without parentheses
for ii=1:10
disp(ii);
end
```

Of course, if you don't use parentheses, you have to add a *whitespace* between the keyword *for* and the identifier *ii*.

Exercise 56. Write a loop to compute the sum of the elements of (1:10).

Solution 56.
```
sum=0;
for(ii=1:10)
```

```
sum=sum+ii;
end
sum

sum(1:10)
clear sum; % Avoid conlicts with built-in sum
```

It is first more intuitive to see *for* loops as incrementing loops. In fact, 99% of the time you will use them as such. Still, the *for* syntax is, in fact, even more powerful. It can loop on any vector (not necessarily constant increment vectors!).

```
for var=vector

...PERFORM ACTIONS

end
```

MATLAB will go through every element in *vector*. At the first iteration, *var* will be the first element of *vector*. At the second iteration, it will be the second element, and so on until the last iteration is reached.

Exercise 57. Let a=[490 123 31 43]. Display all the elements of a sequentially using the for loop. Use *pause* (see the MATLAB help) and ask the user to press *ENTER* to continue.

Solution 57. `a=[490 123 31 43];`

```
for(ii=a)
disp(ii);
disp('Please press enter to continue');
pause
end
```

Remark 31. **Preallocation** – You will often use *for* loops to fill data in matrices or vectors. When you try to write data at an index exceeding the size of the matrix or vector, MATLAB will automatically reassign memory to hold the new data. It will also automatically fill missing data with zeros. This process is computationally expensive, especially when compounded through loops. If you know, a priori, the size of the matrix, you should preallocate the memory by using the function *zeros*.

Exercise 58. Let a=1. Using isscalar, test if a is a scalar. What is the size of a? Now, set a(10,10) to 1. What happened to a? Is it scalar? What is its size? What is the value of a(5,5)?

Solution 58.
```
a=1;
isscalar(a)
size(a)
a(10,10)=1;
isscalar(a)
size(a)
a(5,5)
```

MATLAB changed the variable type from scalar to matrix and the *size* parameters from [1 1] to [10 10]. It also allocated 99 new memory slots.

Exercise 59. Advanced: In this exercise, you will assess the impact of preallocating the memory in MATLAB. You will fill a vector of length 100 000 with random numbers between 0 and 1. Display the computational time with the operators tic and toc (see the MATLAB help) and compute the efficiency gain. *Preallocation is very important!*

Solution 59.
```
%Without preallocation

clear unif_var;
tic;
for(i=1:100000)
unif_var(i)=rand;
end
without_time=toc;

clear unif_var;

%With preallocation
tic;
unif_var=zeros(100000,1);
for(i=1:100000)
unif_var(i)=rand;
end
with_time=toc;

disp('It is');
disp(without_time/with_time);
disp('times faster to preallocate')
```

As opposed to nested *while* loops, nested *for* loops are very useful. Since they are computationally expensive, they should be used only when absolutely necessary.

Exercise 60. You will now perform the same task as in ex. 54, by using nested *for* loops. Namely, fill a 10-by-10 matrix where each element is the sum of the vertical and horizontal indices. Note how the *for* syntax is more readable than the *while* syntax. *Tip*: preallocate this time!

Solution 60.

```
A=zeros(10,10);

for(ii=1:10)
    for(jj=1:10)

    A(ii,jj)=ii+jj;

    end
end
```

Remark 32. **Preallocation for robustness** – Preallocation is not only important for performance issues. As argued earlier in remark 9, scripts can often interfere with the workspace (when the same identifier is used for multiple purposes in different scripts). By preallocating all variables in a script, you reset any duplicated variables in the workspace, eliminating possible interferences.

Remark 33. **Nested *for* loops** – If possible, you should always have the shorter loops at the higher level. For example, if loop A runs through 10 elements and loop B through 100000, the outermost loop should be loop A. In ex. 61, you will assess the perfomance gain from following this guideline.

Exercise 61. Advanced: Create two nested loops filling a 100000-by-10 matrix with the sum of the vertical and horizontal indices (as in 60). Assess the computational burden of filling columns versus rows first. It should be faster to have the shortest loop at the outermost (highest) level.

Solution 61.

```
A=zeros(100000,10);

%slow way
tic;
for(ii=1:100000)
    for(jj=1:10)
    A(ii,jj)=ii+jj;
    end
end
colfirst_time=toc;

%fast way
tic;
for(jj=1:10)
    for(ii=1:100000)
    A(ii,jj)=ii+jj;
    end
```

```
end
rowsfirst_time=toc;

disp('It is');
disp(colfirst_time/rowsfirst_time);
disp('times faster to have the smallest loop to the higher level.');
```

Remark 34. **Invariant expressions** – One common mistake is to let invariant expressions inside loops. They should be moved outside. For example, consider the following code that computes the excess returns of a stock:

```
price = [56.2 57.1 54.78 53.21];
ret=diff(log(price));

for(tt=1:length(ret))
   RFR=0.06/252; %daily risk-free rate
   excess_ret(tt)=(ret(tt)-RFR)
end
```

The risk-free rate is computed at each iteration of the loop. This is a waste of computational resources since RFR is a constant! Often, determining what can be taken out of a loop is a challenging task. Go through every line in your loops and ask yourself: is the output of this line varying at each iteration? If not, it can be safely taken out. In the previous example, the correct code would simply be

```
price = [56.2 57.1 54.78 53.21];
ret=diff(log(price));
RFR=0.06/252; %daily risk-free rate: a constant!

for(tt=1:length(ret))
   excess_ret(tt)=(ret(tt)-RFR)
end
```

Remark 35. **Vectorization – Avoid *for* loops!** – *This is the most important remark of this tutorial!* Most newcomers will see the *for* loop as their holy grail and overuse it. We *stronly* suggest to avoid *for* loops as much as possible in high-level languages such as MATLAB. Most tasks can be performed faster with operators and built-in functions. You should not use *for* loops to perform sums of vectors, elementwise operations, searches of index or any other task that can be equivalently fulfilled by MATLAB built-in functions.

In fact, you should aspire to cast most of your financial engineering problems into a vector/matrix framework and use MATLAB operators and built-in functions to perform the needed tasks; this is called *vectorization*. Most of the preceding and following exercises/remarks on *for* loops can be easily vectorized. For example, to compute the excess returns in remark 34, simply type

```
price = [56.2 57.1 54.78 53.21];
ret=diff(log(price));
RFR=0.06/252;

excess_ret=ret-RFR; %a single line instead of a for loop!
```

Indeed, when substracting a scalar from a vector, you learned that the operation is performed elementwise. There is no need for a *for* loop here!

Also, to compute the sum of (1:10) (see ex. 56), simply type

```
sum(1:10)
```

No need for a *for* loop here either ! Those are two very simple examples of vectorization.

6.5 Switch case

The switch case syntax is used to perform actions conditional on a test variable being equal to a value. We must know, a priori, the set of values the test variable can take. Those are called *cases*. Most of the time, the test variable will be an integer. The syntax is:

```
switch test_variable

    case value_1
    ... %execute this code if test_variable == value_1 is true
    case value_2
    ...%execute this code if test_variable == value_2 is true
    ...
    otherwise
    ...%execute this code if test_variable is not equal to the previous values

end
```

You can have as many cases as you wish. Note that for the *switch case* to be well-defined, the values tested must be mutually exclusive!

Exercise 62. Using the function *randi*, generate an integer between 1 and 5. Display the generated value using a switch case.

Solution 62.

```
mynumber=randi(5);

switch mynumber

    case 1
    disp('The random number is 1.');
```

```
    case 2
    disp('The random number is 2.');

    case 3
    disp('The random number is 3.');

    case 4
    disp('The random number is 4.');

    case 5
    disp('The random number is 5.');

otherwise
error('Unexpected random number')

end
```

6.6 Exercises on vectorization

In this section, we will propose (poorly written) codes and algorithms performing very basic tasks using loops. You will have to first understand what the code does and then rewrite it correctly (without loops!) using vectorization.

Exercise 63.
```
A = zeros(11,1);
k = 0;
for s = 1.0: -0.1: 0.0
    k = k+1;
    A(k) = s;
end
A=A';
```

Solution 63.
```
A=(1:-0.1:0);
```

Exercise 64.
```
% Keep A as in the previous exercise
d = size(A,2);
target = 0.5; %find this target value in A

for i = 1:d
    if (A(i) == target)
        indx=i;
    end
end
```

Solution 64.
```
indx=find(A==target);
```

Exercise 65.
```
A = [2 5 7 3 ; 2 12 1 8];
lig = size(A,1);
col = size(A,2);
target = 1; % find this target value in A

for m = 1:1:lig
    for n = 1:1:col
        if n==1
            if A(m,n) == target
               indx1=m;
               indx2=n;
            end
        elseif (n==2)
            if A(m,n) == target
               indx1=m;
               indx2=n;
            end
        elseif (n==3)
            if A(m,n) == target
               indx1=m;
               indx2=n;
            end
        elseif (n==4)
            if A(m,n) == target
               indx1=m;
               indx2=n;
            end
        end
    end
end
```

Solution 65.
```
[indx1, indx2]=find(A==target);
```

Exercise 66. Advanced: In the following exercise, you will vectorize the simulation of a geometric brownian motion (GBM).

```
nper = 100; %number of periods
mu = 0.01;  %mean of the GBM
sigma = 0.05; %variance of the GBM
nsim= 10;  %number of simulations
S_0 = 100; %initial stock value

S=zeros(nper+1,nsim);
S(1,:)=S_0;

for nn=1:nsim
```

```
   for tt=2:nper+1

   S(tt,nn)=S(tt-1,nn)*exp((mu-sigma^2/2)/nper + sigma*randn/sqrt(nper));

   end
end

t=[zeros(1,nsim);repmat((1:nper)'/nper,[1 nsim])];
```

Solution 66.

```
nper = 100; %number of periods
mu = 0.01;  %mean of the GBM
sigma = 0.05; %variance of the GBM
nsim= 10;  %number of simulations
S_0 = 100; %initial stock value


S = [S_0*ones(1,nsim);S_0*exp(( repmat((1:nper)'/nper,1,nsim)*(mu-sigma^2/2) +...
 sigma*cumsum(randn(nper,nsim),1)/sqrt(nper)))];
t = [0*ones(1,nsim); repmat((1:nper)'/nper,1,nsim)];
```