An introduction to MATLAB for financial engineering

Tutorial: Part II

Hugo Lamarre

HEC montreal

1 Introduction

This tutorial is the second part of two. It will cover more advanced topics: plotting a graph, finding roots, optimizing, regressing (linearly) and importing/exporting data sets. We will also go through more advanced variable types such as strings, cell arrays, structures and multidimensional arrays. See the various appendixes for how to debug, how to use the profiler and how to perform input and output checks (to add robustness to your functions).

2 Figures (plotting)

The basic graphical interface in MATLAB is the *figure*. Type

```
figure;
```

in the command window. This should open a blank figure. It is now the *active graphical device*. Any graphical instructions will be directed toward this figure. To close all previously opened figures, type

```
close all;
```

To plot basic two-dimensional graphs, use the fonction *plot*. If vectors are passed, the first two inputs are the corresponding x-axis and y-axis values. For example, if you call

```
plot(x,y)
```

MATLAB will create the points $(x(i), y(i))$ (for i going from 1 to $length(\mathrm{x})$) on the graph. By default, MATLAB will plot a blue line (i.e. it will connect the adjacent elements of (x, y) with a blue line). The vectors must of course be of the same length (but one could be a row-vector, while the other a column-vector).

Exercise 1. Plot a line that goes from *(x=1, y=1)* to *(x=100, y=200). Tip:* If you omit to open a figure and don't have one opened already, MATLAB will automatically create a new one.

Solution 1. `plot([1 100],[1 200]);`

Since MATLAB plot points, you cannot have infinite precision graphs. You have to adjust the discretized grid (x-values) on which you are plotting.

Exercise 2. Plot x^2 for x between -100 and 100. Try different levels of discretization.

Solution 2.
```
x=(-100:1:100);
plot(x,x.^2);

x=(-100:0.5:100);
plot(x,x.^2);

x=(-100:0.01:100);
plot(x,x.^2);

x=(-100:0.001:100);
plot(x,x.^2);
```

As you probably noticed in the previous example, when performing subsequent plots, MATLAB overwrites the previous plot by the new one. This is undesirable, since you will want to compare graphs at some point. To plot multiple graphs in different figures, simply initiate new figures (*figure*) before plotting.

To change the color of a line, set the third input of *plot* to the desired color (e.g. 'r' for red). You can also specify the desired line style through the same third input (e.g. ':' for a dotted line). Search for *LineSpec* in the MATLAB help for a complete list of *color & line specifiers*. To both change the color and line style, concatenate the line style & color specifiers and pass the result as the third input of *plot*. 'r' and ':' are called strings. For more information on how to generate and concatenate strings, see section 4. For example,

```
plot(x,y,[':' 'r']); %plot a red dotted line
plot(x,y,':r'); %equivalent
```

You can also pass marker specifiers to highlight the plotted points with a marker (e.g. a cross). When passed a marker specifier *only*, MATLAB will not connect the different points with a line. You can concatenate line style, markers and color specifiers (in this order) and pass the result as the third input of *plot*.

Exercise 3. Plot $sin(x)$ for x in (-6.6:0.1:6.6). Try different colors, line styles & markers. Plot a graph with markers only and one with both markers & lines.

Solution 3.
```
x=(-6.6:0.1:6.6);

figure;
plot(x,sin(x),['--' 'r']);

figure;
plot(x,sin(x),'xb');
figure;
plot(x,sin(x),'.-.k');
```

You could also modify lines through more advanced properties such as *LineWidth, MarkerEdgeColor, MarkerFaceColor, MarkerSize* with the following syntax:

```
plot(x,y,'Property',Value,...);
```

Search for *LineSpec* in the MATLAB help for more information.

Exercise 4. Advanced: Try changing the following properties: *LineWidth, MarkerEdgeColor, MarkerFaceColor, MarkerSize* for the preceding graphs (see ex.3). Use the *Color* property to specify the color of your line. Specify all colors randomly through *RGB triples* (vectors of length 3 with elements in (0,1)) instead of strings (such as 'r').

Solution 4.
```
x=(-6.6:0.1:6.6);
%using properties of lines
figure;
plot(x,sin(x),'-o','LineWidth',2,'MarkerEdgeColor',rand([3 1]),...
 'MarkerFaceColor',rand([3 1]),'MarkerSize',10,'Color',rand([3 1]));
```

To plot multiple graphs *on the same figure*, first initiate a new figure. Then, type *hold on* to tell MATLAB not to overwrite the active figure. Plot your graphs. Type *hold off* to let go.

Exercise 5. Using *hold on* & *hold off*, plot sin(x) and sin(x+$\frac{\pi}{2}$) on the same figure for x in (-6.6:0.1:6.6). Assign different line specifications to each lines.

Solution 5.
```
x=(-6.6:0.1:6.6);
figure;
hold on;
plot(x,sin(x),'.-.r');
plot(x,sin(x+pi/2),'x-.b');
hold off;
```

To add a legend, use *legend*. Pass it a cell array of strings of the same length as the number of lines. Use the curly brackets to create cell array of strings (more on this syntax in section 4). MATLAB will automatically assign the corresponding line specifications to each string of the legend. In other words, the strings in the cell array must match the order in which you plotted the different graphs!

Exercise 6. Add a legend to the previous graph.

Solution 6.
```
leg_str={'f(x)=sin(x)','f(x)=sin(x+pi/2)'};
%plotted sin(x) first => first element!

legend(leg_str); %applying legend
```

You can also change the label of the x-axis and y-axis with *xlabel* and *ylabel* respectively by passing the desired string.

Exercise 7. Add labels to the previous graph.

Solution 7.
```
xlabel('x');
ylabel('f(x)');
```

Finally, you can specify the limits of the x-axis and y-axis with *xlim* and *ylim* respectively. To do so, pass a vector of length 2, where the first element corresponds to the lower limit and the second to the higher limit.

Exercise 8. Let the x-axis from the previous graph go from 0 to π and the y-axis from 0 to 1.

Solution 8.
```
xlim([0 pi]);
ylim([0 1]);
```

To generate figures with multiple axes, use *subplot*. *subplot* tells MATLAB to create axes in tiled position. It is executed before plotting each sub-graph:

```
figure;

subplot(nrow, ncol, 1);
plot(x,y);

...

subplot(nrow, ncol, nrow*ncol);
plot(x,y);
```

nrow and *ncol* must be constant for each call of *subplot* for a given figure. The third input of *subplot* is the index (going from 1 to *nrow***ncol*) which tells MATLAB where to plot the next graph. Exceptionally, MALTAB lets indices increment on rows. Imagine a 2-by-2 array of graphs , the first index would refer to the top-left axis, the second to the top-right, the third to the bottom-left and the last to the bottom-right. You can merge adjacent tiles by specifying two or more indices.

Exercise 9. Using *subplot*, generate a 2-by-2 array of axes. On the two top axes (i.e. at (1,1) and (1,2)), plot sin(x) and sin(x+$\frac{\pi}{2}$). Merge the two lower axes and plot a single graph: sin(x-$\frac{\pi}{2}$).

Solution 9.
```
x=(-6.6:0.1:6.6);
figure;

subplot(2,2,1)
plot(x,sin(x),'.-.r');

subplot(2,2,2)
plot(x,sin(x+pi/2),'x-.b');

subplot(2,2,[3 4]);
plot(x,sin(x-pi/2),':k');
```

You can also pass matrices to *plot* multiple lines at once. Each column is assigned to a different line. Colors are attributed automatically.

Exercise 10. Using the algorithm for simulating geometric brownian motions proposed in the first part of this tutorial, plot each trajectories with a different line by passing a matrix. Don't call *hold on* & *hold off*.

Solution 10.
```
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

 %each column corresponds to the evolution of one path
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)];

figure;
plot(t,S);
```

See *hist* and *scatter* for histogram plots and scatter plots respectively.

3 Root finding, Optimization and Linear Regression

3.1 Root finding

Finding a root refers to finding a value of x (a scalar!), such that f(x) is 0. The basic tool to solve such problem in MATLAB is *fzero*. It basically takes two input: a *function handle* which points to the function to solve and an initial starting point (a scalar, namely *x0*). The function could be external (i.e. a m-file) or anonymous. If it is external, it must be assigned to a *function handle* to be called indirectly by *fzero* later on. This function handle must take a single scalar as input. We will first introduce anonymous functions, which are a special type of function handles.

Remark 1. **Anonymous functions** – In the first part of this tutorial, you learned how to build functions. Usually, functions are held in separate *m-files* for performance and tractability issues. You will encounter situations (e.g. optimizing) where declaring functions *on-the-fly* is more tractable. To do so, you will use what is called *anonymous functions*. Those are assigned to standard function identifiers in either scripts or other functions. The operator is

```
anon_func = @(variable_name1, variable_name2,...) (function_output)
```

For example, you could write a function computing squares as:

```
sqr_id = @(x) (x*x);
%this anonymous function
%computes the square of a variable
```

There is no restriction on x. We don't have to specify, a priori, if x is a scalar, a vector or a matrix. But if a vector is passed to *sqr_id*, an error message will be displayed as usual (the vector dimensions don't agree). Try sqr_id([1 2])! Anonymous functions are also often used to fix parameters. This will be explored in the next exercise (11).

Exercise 11. Let's say you are tired of specifying *dim* in *sum*. Indeed, most of the time, you find yourself computing sums accross the second dimension (i.e. *dim* is 2!). You would like a function, called *sum_row* that does just that, but without having to specify *dim* to 2. Use anonymous functions. Try your function on repmat((1:10),[10 1]).

Solution 11. `sum_row=@(x) sum(x,2); %fixing second parameter of sum`
`sum_row(repmat((1:10),[10 1]))`

Remark 2. **Passing external functions as inputs – *function handles*** – To pass a function as input, the function must be an object (or variable) living in the workspace. When creating an anonymous function, you assign it to a function handle. To transform an external function into a function handle, simply type:

```
function_handle_id = @external_function_id;
```

To fix all parameters of an external function except one, use anonymous functions:

```
function_handle_id =...
    @(x) external_function_id(par1, ..., park, x, parp1, ..., parpn);
```

where it is assumed the parameters (*par1, ..., park, parp1, ..., parpn*) are already defined (either in the workspace or in the local memory of a function).

Some MATLAB built-in functions output *exitflag*. This flag (an integer) lets the end user know if the function performed normally (e.g. found a root). Refer to the MATLAB help of a specific function for the possible values and the associated meanings of *exitflag*. If the function did not performed normally, it will usually also output a warning message. Warning messages are convenient when working from the command window. When building functions, the exit flag is more relevant, since it allows you to build switch case and conditional statements. For example, if the exit flag is not 1 for *fzero*, the algorithm did not converge and the root found is meaningless. Note that even though an algorithm did not converge, it will still output its last attempt. Be careful and always check the exit flag!

Exercise 12. Using anonymous functions, find the root of $10\exp(x)-5$, by starting from 100. Try to find the root of $\exp(x)+1$ (start from 0). Output both exit flags. What warning message do you get? If the algorithm did not converge for either functions, display an error message with *error*.

Solution 12.
```
func1=@(x) 10*exp(x)-5;
func2=@(x) exp(x)+1;
[x1 fx1,exitflag1]=fzero(func1,100);
[x2,fx2,exitflag2]=fzero(func2,0);

if (exitflag1~=1 || exitflag2~=1)
error('One of the two roots or both were not found');
end
```

Exercise 13. Solve for a root of $1000 * \sin(x) + 3x - 4x^2 + 2x^3$, by creating an external function (*test_func.m*) and importing it in the workspace through a function handle. Initiate *fzero* at 5. *Tip:* Don't forget to put *test_func.m* in your path.

Solution 13. **In test_func.m**

```
function [ out ] = test_func( x )
out=1000*sin(x)+3*x-4*x.^2+2*x.^3;
end
```

In the command window

```
func_hand=@test_func;
[x_test,fx_test]=fzero(func_hand,5);
```

Note how we used the elementwise operator .^ in *test_func* even though *fzero* only needs functions that take a scalar as input. This is simply for further use. For example, we might wish to plot the graph of the function later on (in which case, we will pass it a vector)! *If a function takes a scalar as input and returns a scalar, always try to write it such that it can handle matrices elementwise.*

Remark 3. **Starting values** – The starting value, *x0*, should be set in a range in which you expect to find the root. For example, if you are solving for a function that takes a portfolio weight (no shorting!) as input, you should set *x0* in (0,1). This is particularly relevant for functions having multiple roots. Always plot your functions on the interval of interest to determine if the problem is well-posed (e.g. has only one root).

To add robustness to your root finding algorithms, you could specify *x0* as a vector of length two. MATLAB will then search for a root between the first element and second element of *x0*. The problem has to be well-posed (i.e. there is a single change of sign for f(x) between *x0(1)* and *x0(2)*).

Exercise 14. Advanced: From the function handle generated in the previous exercise (see ex. 13), plot the graph of *test_func* on (-8:0.01:8). Find roots by starting at different *x0*s. Let the starting values be in (-8:8). By holding the plot, mark each solution (root) with a cross.

Solution 14.
```
x=-8:0.01:8;
figure;
hold on;
plot(x,test_func(x));

for x0=-8:8
[x_tmp,fx_tmp]=fzero(func_hand,x0);
plot(x_tmp,fx_tmp,'xr','markersize',10);
end

hold off;
```
The algorithm converges to different values, depending on the starting value!

3.2 Optimization

In MATLAB (and most other high-level languages with built-in optimization routines), when we think about optimization, we think about *minimization*. If you happen to need to *maximize* a certain function, simply define it as its negative (i.e. $-f(x)$) and *minimize*. The two basic built-in routines are *fminunc* and *fmincon* (*fmin* for minimization of f). *fminunc* minimizes problem without constraint (*unc* for unconstrained) and *fmincon* with constraints (*con* for constrained).

Both functions accept function handles taking a vector as the single input (as opposed to fzero *which handles scalars only).*Of course, the function handle must return a scalar for the minimization to make sense! Also, as for *fzero*, you must provide a starting value, *x0*, to the optimization routine.

Remark 4. **Passing vectors to anonymous functions** – As previously mentionned, when creating anonymous functions, inputs are left unspecfied (i.e. they could be scalars, vectors, matrices, ...). If you assume a vector will be passed to your function handle, you must use indices. For example, let's say you want to create a function handle returning $f(x, y) = sin(x + y)$, but by passing a vector of length 2 containing x and y (instead of two scalars). You would simply type:

```
f=@(x) sin(x(1)+x(2));
```

Exercise 15. Using *fminunc*, minimize $f(x) = x^2 + y^2$. Start at (100,100).

Solution 15.
```
f=@(x) x(1)^2+x(2)^2;
[x,fval]=fminunc(f,[100 100]);
```
*Note that the starting vector (*x0*) length tells* fminunc *(and* fmincon*) what is the expected length of the input vector.*

Remark 5. **Local vs. Global maximum** – As for root solving, you might encounter minimization problems that are not well-posed. Indeed, a function could have multiple local minima, in which case *fminunc* and *fmincon* might fail to find the true global minimum. Most basic financial problems (e.g. optimal investment frontiers in modern portfolio theory) will be strictly convex, in which case this is not an issue.

There are four types of constraints you can pass to *fmincon*: linear inequalities, linear equalities, lower & upper bounds and non-linear constraints (both inequalities and equalities). The first three types are passed as matrices or vectors. See the MATLAB help on *fmincon* for how to specify linear constraints. You can have as many constraints as you wish for each constraint type *independently* (e.g. no linear inequality and 3 linear equalities)!

Remark 6. **Specifying lower and upper bounds** – When you have bounds on only a subset of variables, you still have to provide the whole vector of bounds (i.e. vectors of the same length as the input of your function). To disregard the bounds of one element, simply set its lower bound to *-inf* and its upper bound to *inf*.

Exercise 16. Using *fmincon*, minimize $f(x) = x + 3.2 * x^2 + 4.6 * y^2$ such that

- x+y≤1;
- x==2.
- y between -2 and -1.

Start at (2,-1.5). *The starting value provided should at least satisfy your constaints!* Verify that this is the case.

Solution 16.
```
f=@(x) x(1)+3.2*x(1)^2+4.6*x(2)^2;
[x,fval]=fmincon(f,[2 -1.5],[1 1], 1, [1 0], 2, [-inf -2], [inf -1]);
```

To pass non-linear constraints, you must create a function handle (referred as *nonlcon* in the MATLAB help) that returns *c* (first output) and *ceq* (second output). The length of the outputted vectors (*c* and *ceq*) represents the number of respective non-linear contraints. Both *c* and *ceq* are compared to 0. Don't forget to add or subtract a scalar from the left-hand side if your constraint is of the type:

```
c(x)<=constant;
ceq(x)==constant;
```

Remark 7. **Unnecessary linear constraints or bounds** – To discard linear contraints or bounds, set the unwanted constraints to the empty matrix:

```
[x,fval]=fmincon(f,[x1 x2],[],[],Aeq,beq]); %discard linear inequalities
```

Remark 8. **To discard either non-linear equalities or inequalities** – If you are interested only in providing one type of non-linear constraint, set the unwanted constraint to the empty matrix (but your *nonlcon* should always have two outputs!).

Exercise 17. Using *fmincon*, minimize $f(x) = x + 3.2 * x^2 + 4.6 * y^2$ such that $x^2 + y^2 \leq 1$.

Start at (sin(0.2),cos(0.2)).

Solution 17. **In nonlcon_test.m**

```
function [c,ceq]=nonlcon_test(x) %two-outputs
c=x(1)^2+x(2)^2 -1 ;
ceq=[]; %discarding
end
```

In the command window

```
f=@(x) x(1)+3.2*x(1)^2+4.6*x(2)^2;
[x,fval]=fmincon(f,[sin(0.2) cos(0.2)],[], [], [], [], [], [], @nonlcon_test);
%don't forget to pass nonlcon_test as a function handle!
```

Remark 9. ***Advanced:*** **Returning the hessian** – Both *fmincon* and *fminunc* return the numerical hessian at the optimal solution. This might be useful when minimizing a negative log-likelihood and estimating the error on fitted parameters.

Remark 10. ***Advanced:*** **Tweaking the optimization routine** – If you feel comfortable with the underlying optimization theory, you could fine-tune MATLAB built-in routines. This is done by providing an object called *options* generated by the function *optimset*. For more information, type *optimset* in the MATLAB help. You could for example change the algorithm used or tweak stopping criteria.

Remark 11. ***Advanced:*** **Casting constrained minimization into unconstrained minimization** – All constrained minimization problems can be casted into a unconstrained versions using changes of variables. For example, let's say you want to set a lower bound, *lb*, on x. You could then define x as $\exp(y) + lb$ and let y be unconstrained. Note how when y is *-inf*, x becomes *lb* and when y is *inf*, x becomes *inf*. Similarly, *tanh* could be useful for bounding variables on $(-1, 1)$. Since *fmincon* is sometimes instable regarding constraints, we strongly recommend casting your problems into unconstrained versions and using *fminunc only*.

3.3 Linear regression

The basic tool to perform linear regressions is *regress*. If you wish to regress y on the factors X, the syntax is:

```
regress(y,X)
```

where each column of X corresponds to a different factor. The length of y is the same as the number of rows in X. In other words, the i^{th} element of y corresponds to the factor observations contained in the i^{th} row of X. Always add a column vector of ones in your factors to allow for a constant loading in the regression.

Exercise 18. Let X=[ones(1000,1) randn(1000,3)] and y=sum(X.*repmat([1.2 0.6 4.2 0.03],[1000,1]),2). If you perform a linear regression of y on X, what loadings do you expect? Perform it!

Solution 18.
```
X=[ones(1000,1) randn(1000,3)];
y=sum(X.*repmat([1.2 0.6 4.2 0.03],[1000,1]),2);
b=regress(y,X);
```
You should have expected b=[1.2 0.6 4.2 0.03].

4 Strings, Cell arrays and Structures

4.1 Strings

MATLAB can handle alphanumeric characters. The variable type holding series of alphanumeric characters is called a *string*. For example, a string could be **this is a string!**, while a character could be **a**, **d** or **1**. In this tutorial, strings & characters are in **bold**. The operator to create strings is the apostrophes ('). The basic syntax is, string_name='type a series of alphanumeric characters here'. You already dealt with this syntax when using the function *disp*.

Exercise 19. Assign 'Hello World' to a variable. Display this variable with *disp*. Assign '1' to a variable. How is it different from assigning 1 to a variable?

Solution 19.
```
hw_str='Hello World';
disp(hw_str);

str1='1'
scal1=1
```
'1' is a string (or character) and 1 is a scalar! In the workspace, the icons associated to *str1* and *scal1* are different.

You might wish to create arrays of strings. For example, it could be useful to store stock names or fund names in some kind of vector. In fact, you will also need this kind of object to store elements of legends on plots! As a first attempt, you would probably try something like this:

```
stock_str1='AAPL';
stock_str2='RIM';
stock_names=[stock_str1 stock_str2]
```

to create such a vector. This is not the correct way of proceeding. Try it! The output is **AAPLRIM**. MATLAB sees each alphanumeric character as one element of a matrix of characters. 'Hello World' is, in fact, a vector of characters of length 11. As any vector, it can be indexed. Thus, the element corresponding to the index 1 would be **H**, and the one corresponding to the index *end*, **d**. The issue here is that while you wish to index conceptual phrases (for example, strings such as 'AAPL'), MATLAB indexes characters (e.g. 'a'). The following two exercices (20 & 21) should make those ideas precise.

Exercise 20. Assign 'Hello World' to a variable. Compute its length. Retreive the index 1,5 and *end*.

Solution 20.
```
hw_str='Hello World';
length(hw_str)
hw_str(1)
hw_str(5)
hw_str(end)
```

Exercise 21. Using the bracket operators ([]), create a 2-by-2 matrix with the following characters:

- at (1,1): **a**
- at (1,2): **b**
- at (2,1): **c**
- at (2,2): **d**

Print the result. Try to horizontally concatenate this matrix of characters with **e**. Why do you get an error message? Try with ['e';'f'].

Solution 21.
```
mat_str=['a' 'b'; 'c' 'd']
[mat_str 'e']
[mat_str ['e';'f']]
```

Remark 12. **Logicals and strings – *switch case* and *strcmp*** – You can test if two strings are equal with the built-in MATLAB function: *strcmp*. It allows for two inputs, namely the two strings to compare. When using the *switch case* syntax with strings, MATLAB will automatically use *strcmp* to conclude on each case.

Exercise 22. Create a switch case on the variable *model* for the following values: **GARCH**, **1D**, **HMM**, **HEAVY**. Assign **GARCH** to model and test the *switch case* by printing the current model used (using *disp*). Using *strcmp*, anwser to the following statement: the model is **HEAVY**.

Solution 22.
```
model='GARCH';
switch model
   case 'GARCH'
   disp('Current model used is GARCH')
   case '1D'
   disp('Current model used is 1D')
   case 'HMM'
   disp('Current model used is HMM')
   case 'HEAVY'
   disp('Current model used is HEAVY')
end

disp('Is the model HEAVY?')
if(strcmp(model, 'HEAVY'))
    disp('Yes!')
else
    disp('No!')
end
```

We still need variables for which indexing returns strings instead of characters. We wish the following command line:

```
stock_names{1}
```

to return **AAPL** and

```
stock_names{2}
```

to return **RIM**. This is the scope of *cell arrays of strings* (see next subsection 4.2).

4.2 Cell arrays

Cell arrays are arrays of higher level than usual arrays (or *numeric arrays*, such as vectors and matrices). Cell arrays can contain numeric arrays. Indeed, each variable or element contained in a cell array can be of any type: string, numeric array and even another cell array. Note that in the same cell array, one element could be a string, while another a matrix. In other words, multiple variable type in a given cell array is acceptable. *Once again, the fundamental difference with numeric arrays is that elements of cell arrays need* not *be scalars!*

The indexing framework is the same as for numeric arrays. Namely, cell arrays contain elements indexed by integers. We will use the keywords *cell vector* and *cell matrix* to refer to cell arrays indexed by one integer and two integers respectively. We could also differentiate between *cell row-vector* and *cell column-vector*.

To initiate cell arrays (filled with empty matrices), use the built-in MATLAB function *cell*. *cell* should be used for preallocations (for example, preceding *for* loops or at the beginning of scripts). We refer to the numeric arrays or strings contained in cell arrays as *underlying elements* (they are at the *lower level*). The *high level* cell arrays elements are called *cells* (or simply *elements*).

The indexing operator to retrieve underlying elements from cell arrays is the curly brackets ({}). The indexing operator to retrieve cells is the parentheses (()). Indexing cells is a high level operation, whereas retrieving underlying elements with curly brackets is a low level operation.

- *cell_name*(ii): read or write at the higher level (i.e. cells);
- *cell_name*{ii}: read or write at the lower level (i.e. underlying elements: matrices, vectors, strings, ...).

Remark 13. **Preallocating underlying matrices** – When filling underlying matrices contained in cell arrays, don't forget to preallocate them! You should first preallocate cell arrays using *cell*, and then preallocate underlying elements using *zeros*.

Exercise 23. Create a cell column-vector of length two, using *cell*. Fill the first component with [1 2] (a vector) and the second with [1 2; 3 4] (a matrix) (those are lower level operations). Print the cell array. Explore it in the variable editor

(go to the lower lever by double-clicking on an element). Try retreiving the underlying elements of your cell vector (i.e. the vector and the matrix). Test if 1 is contained in the first element of you cell. *Tip:* Once an underlying element is retrieved from a cell (i.e. a vector or matrix), you can perform logical tests or operations on it the usual way. Try multiplying the first element of your cell vector by [2;1].

Solution 23.
```
first_cell=cell([2 1]); %equivalently, cell(2,1);
first_cell{1}=[1 2];
first_cell{2}=[1 2; 3 4];

first_cell

first_cell{1}
first_cell{2}

any(first_cell{1}==1)
first_cell{1}*[2;1]
```

When printing cell arrays containing numeric arrays in the command window, MATLAB will only output the variable type and the size associated to each element (e.g. [1 x 2 double]). Generally speaking, cell arrays of strings have slightly different behaviors. For example, MATLAB will display explicitly all underlying strings. We will come back to cell arrays of strings later on.

When writing data at an index exceeding the cell size, MATLAB will automatically reallocate memory and fill unspecified values with empty matrices (which is why you need to preallocate cell arrays also!). You can concatenate cell arrays using the usual brackets operator ([]). Since you concatenate cell arrays elements (namely cells), this is a high level operation. Clear cell arrays using *clear*.

Exercise 24. Fill the fifth-element of your first cell column-vector (see ex. 23) with an empty matrix. Print the resulting cell vector. Note how, *in a cell array*, the empty matrix can be used to expand a variable when indexing with the curly brackets. Test (using *isempty*) if the third element of your cell array is an empty matrix. Concatenate horizontally your cell array with a new empty cell column-vector of length 5 (so that the concatenating dimensions match!). The result is a cell matrix of size [5 2]. Verify it!

Solution 24.
```
first_cell{5}=[];
first_cell
isempty(first_cell{3})
first_cell=[first_cell cell([5 1])]
size(first_cell)
clear first_cell;
```

Remark 14. **Creating cells with the curly brackets operator** – *When creating cell arrays,* {} *is to cell arrays what is [] to numeric arrays.* Using whites-

paces and semi-colons, you can build cell arrays from matrices and/or strings. Note that as numeric arrays, cell arrays can be transposed.

Exercise 25. Create a cell column-vector containing: [1 2], [3 4], [5 6; 7 8] using curly brackets. Tranpose it to create a cell row-vector.

Solution 25.
```
test_cell={[1 2]; [3 4]; [5 6; 7 8]};
test_cell'
```

Exercise 26. Assign to *stock_id* the following strings: 'SPY', 'QQQ', 'DIA', 'VIX', 'EEM'. You should be able to index each strings by an integer. *Tips:* Use a cell array to hold the strings. Preallocate using *cell*. You created you first *cell array of strings*!

Solution 26.
```
stock_id=cell(5,1);
stock_id{1}='SPY';
stock_id{2}='QQQ';
stock_id{3}='DIA';
stock_id{4}='VIX';
stock_id{5}='EEM';
stock_id

%equivalently
stock_id={'SPY';'QQQ';'DIA';'VIX';'EEM'}
```
When typing *stock_id* in the command window, MATLAB displays the underlying strings.

MATLAB allows for the curly brackets and parentheses operators to be used on a single line of code to index underlying *scalars* directly. Let's say there's a vector at the ii^{th} element of a cell array named *cell_name*. An expression such as:

```
cell_name{ii}(jj)
```

is perfectly acceptable and is equivalent to

```
var_tmp=cell_name{ii}; %this is a numeric array or string
var_tmp(jj)
```

The first {ii} refers to the ii^{th} element of the cell, which is a vector. (jj) refers to the jj^{th} element of that particular vector. You could either read or write data in underlying vectors using that syntax. Of course, this can be extended to underlying matrices, namely:

```
cell_name{ii}(jj,kk)
```

It can also be extended to cell matrices containing underlying matrices (in fact, you could create any combination of indices you wish!):

```
cell_name{ii,jj}(kk,ll)
```

Exercise 27. Using three *for* loops and the function *rand(1)*. Fill a cell column-vector of length 10 with 5-by-5 matrices containing uniform(0,1) random numbers. Preallocate both the cell vector *and the underlying matrices*!

Solution 27.

```
rand_data=cell(10,1); %preallocating the cell.

for cell_id=1:10

rand_data{cell_id}=zeros(5,5); %preallocating the underlying matrices.

    for ii=1:5
        for jj=1:5
        rand_data{cell_id}(ii,jj)=rand(1);
        end
    end

end

rand_data
```

We will now distinguish more precisely high-level operations from low-level operations. We briefly mentionned earlier that the parentheses (()) operator (when applied directly on cell arrays and not after the curly brackets) can be used to index cells (*at the high level*):

```
cell_name(ii)
```

Indexing cell arrays with parentheses allows you to select multiple cells without retrieving the underlying numeric arrays or strings. This is particularly useful when swapping cells, creating sub-cell arrays or deleting cells.

The curly brackets operator *extracts* underlying elements. In a sense, it performs an action. For example, *cell_name*{1:end} or *cell_name*{:} would extract the whole content of a cell array. Thus, expressions such as

```
cell_name{1:2}=[1 2]
```

don't make any sense. Indeed, we are trying to assign a single numeric array to multiple numeric arrays! But

```
cell_name(1:2)={[1 2]}
```

is a correct expression, since we are allocating two cells to the same cell, namely {[1 2]}.

To understand the fundamental difference between the two operators , let's have a look at the following code:

```
cell_tmp=cell(10,1);

cell_tmp(1)=[];
%Here you selected the cell element (high level)
%and you supressed it!

cell_tmp{1}=[];
%Here you extracted the underlying cell element (low level)
% and assigned it to the empty matrix!

cell_tmp(1) %is the same as cell(1)
%i.e. a cell (high level)

cell_tmp{1} %is the empty matrix
%i.e. the underlying first element of the cell array (low level)
```

When extracting multiple elements with the curly brackets operator, *the output will be an unstructured series of numeric arrays (or strings)*. To assign this series, the same number of elements must be provided on the left-hand side. A correct expression would be:

```
[empty1 empty2 empty3 empty4 ] = cell_tmp{1:4}
```

Those are quite subtle issues that will be clarified in the next exercise (28).

Exercise 28. Go in the workspace and open *rand_data* in the variable editor. Are you at the lower or higher level? Now, double-click on any element. Same question?

Swap the two first elements of the cell vector generated in ex. 27 using the parentheses operator (()) only. Try doing the same with the curly brackets operator ({}) only. Why is this less tractable? Print the first element using the parentheses operator (()) first and then using the curly brackets operator ({}). How do the outputs differ? Now, overwrite the last five matrices with the first five using the parentheses operator (()). Try doing the same with the curly brackets operator. You should not be able, since this is a high level operation! Print the first five matrices using the curly brackets operator ({}). Try to assign the first two matrices to two identifiers on one line of code.

Solution 28.

```
%First way: Higher level
rand_data_cell=rand_data(1); %is a (sub-)cell of size 1
rand_data(1)=rand_data(2);
rand_data(2)=rand_data_cell;
```

```
%Second way: Lower level
rand_data_matrix=rand_data{1}; %is a 5-by-5 matrix
rand_data{1}=rand_data{2};
rand_data{2}=rand_data_matrix;
%Needlessly interact with the lower level!
%Less tractable!
%First way is better!

rand_data(1) %selecting a cell: higher level
rand_data{1} %selecting a matrix: lower level

rand_data(6:10)=rand_data(1:5); %correct!
%copying some cell elements and overwriting others
%high level operations

rand_data{6:10}=rand_data{1:5}; %Error! Does not make sense!
%trying to perform lower level operations to manipulate cell elements!

rand_data{1:5} %extracting underlying elements for display

[rand_data1 rand_data2 ] = rand_data{1:2}
%This makes sense since we assign two elements to two elements
%It is equivalent to
%rand_data1=rand_data{1};
%rand_data2=rand_data{2};
```

In fact, expressions such as *rand_data*{6:10} or *rand_data*{:} are rarely used, except for displaying in the command window.

Remark 15. **Advanced functions for cells** – The following functions could be useful when dealing with cells:

- **cell2mat** – convert cell array to numeric array (*cell array must contain scalars only!*);
- **num2cell** – convert numeric array to cell array;
- **cellfun** – apply function to each cell in cell array.

Exercise 29. Advanced: Using *cellfun* (see help), create a logical vector of the same length as *rand_data* that anwsers to the following statement: one element of the underlying random matrix is greater than 0.95. Print the underlying elements satisfying the statement. *Tip:* use anonymous functions. Let the third element of *rand_data* be the empty matrix. Test if the underlying matrices are empty. Verify that the third element of the resulting logical vector is 1 (and 0 everywhere else!).

Solution 29.
```
g95_indx=cellfun(@(x) any(x(:)>0.95),rand_data)
rand_data{g95_indx}
```

```
rand_data{3}=[]
empty_indx=cellfun(@(x) isempty(x),rand_data)
```

Remark 16. **Cell arrays of strings and *strcmp*** – As briefly mentionned earlier, cell arrays of strings are a specific class of cell array. For example, the function *strcmp* allows you to compare each strings from a cell array to a test string. It outputs a logical array of the same size as the initial cell array of strings.

To convert each rows of a matrix of characters (e.g. ['abc'; 'def'; 'ghi']) into an element of a cell array, you can use *cellstr*. See also *iscellstr* to test if a variable is a cell array of strings. As standard cell arrays, you can create cell arrays of strings using the curly brackets operator directly: *cellstr_name*={'abc' 'def' 'ghi'; 'jkl' 'mno' 'pqr'}.

Exercise 30. Using *strcmp*, create a logical vector indexing the position of **DIA** in *stock_id*. Print **DIA** in the command window using your index. Generate *stock_id2* containing **SDS**,**GLD**,**TZA**. First create a matrix of characters and then convert it into a cell array using *cellstr*. Concatenate vertically *stock_id* with *stock_id2*.

Solution 30.
```
dia_id=strcmp('DIA',stock_id)
stock_id{dia_id}
stock_id2=['SDS';'GLD';'TZA'];
stock_id2=cellstr(stock_id2);
stock_id=[stock_id;stock_id2]
```

Remark 17. ***Advanced:*** **Nested cell arrays** – Sometimes, you will (unwillingly) perform operations that assign a cell array to an element of a cell array. This is usually because you are interchanging higher level operators (e.g. () or []) with lower level operators (e.g. {}) (or vice versa). This could get problematic for *for* loops in which you concatenate cell arrays at each iteration. Indeed, you could end up with thousands of nested cell arrays (see 32). More advanced users might use nested cell arrays to build trees or branching processes (binomial trees for pricing options, for example).

Exercise 31. By using the brackets and curly brackets operators, add the following stocks to *stock_id*: **GENZ**, **BIIB**, **GILD**. Try doing the same with the curly brackets operator *only*.

Solution 31.
```
%Correct way:
[stock_id; {'GENZ'; 'BIIB'; 'GILD'}]
%concatenating two cell arrays with brackets (higher level!)

%Nested cell arrays
{stock_id; 'GENZ'; 'BIIB'; 'GILD'}
%error: concatenating a cell array as an underlying element!

%Curly brackets only? Not possible!
```

```
{stock_id{:}; 'GENZ'; 'BIIB'; 'GILD'}
%error: when using {:}, you lose the structure of the cell array.
```

Exercise 32. Advanced: With a *for* loop, concatenate a cell vector of strings (of length 10000) where each element is its corresponding index (converted into a string). Use the function *num2str* (see help if necessary!). *Tip:* To initiate an empty cell array, simply use *cell*(0) or {}.

Solution 32.
```
%Wrong way: nested cell arrays
embeded_cell=cell(0); %initiate an empty cell array
for ii=1:10000
    embeded_cell={embeded_cell; num2str(ii)};
end


%Correct way
correct_cell=cell(0); %initiate an empty cell array
for ii=1:10000
    correct_cell=[correct_cell; {num2str(ii)}];
end
```

Explore *embeded_cell* in the workspace! Try to understand what caused the cell arrays to be nested in the first case and not in the second.

Cells are a very powerful tool to deal with mixed (integers, scalars, vectors, matrices, three-dimensionnal numeric arrays and/or strings) and complex data sets. Nevertheless, the use of cell arrays limits and complicates vectorization. Try using *cellfun* instead of *for* loops. Be careful not to overuse *for* loops and always try to vectorize first. When confronted with mixed databases, try to break down neatly the alphanumeric part from the stricly numeric part!

Finally, remember that *the smooth parentheses operator (()) allows you to manipulate cells at the high level (namely, its elements), while the curly brackets operator extracts the underlying elements from the lower level (namely, vectors, matrices or strings).*

4.3 Structures

The structure is a very simple and convenient data type. A structure contains different elements, also called *values*, with a string attached to each, called the *field*. This string should highlight the nature of the value (e.g. 'date' if the value is a date). Note that each field could refer to different data type (i.e. both 'date' and 'stock_price' could be fields of the same structure, with the former linked to a string and the latter to a scalar). Structures are used to add a *semantic structure* to your variables.

The basic function to generate structures is *struct*. The inputs are an indefinite number of couples: *field* & *values*, where *field* is necessarily a string. Equivalently, you can add a field by writting data in an undefined field using the dot (.) operator. The basic syntax to read or write from a field is:

```
structure_name.field_name
```

Exercise 33. Create a structure, called *option_data*, containing the following fields and values:

- today_date is '15-may-2010';
- exp_date is '17-may-2010';
- DTE is 2;
- underlying is 'SPX';
- strike is 1200;
- call_price is 10.4;
- put_price is 8.4;

First, use *struct*. Then, clear the structure and re-assign *option_data* to the same fields and values *with the dot operator*. Retrieve **DTE** from *option_data*.

Solution 33.
```
option_data=struct('today_date','15-may-2010',...
'exp_date', '17-may-2010','DTE',2,...
'underlying','SPX', 'strike', 1200,...
'call_price', 10.4, 'put_price', 8.4)

clear option_data;

option_data.today_date='15-may-2010';
option_data.exp_date='17-may-2010';
option_data.DTE=2;
option_data.underlying='SPX';
option_data.strike=1200;
option_data.call_price=10.4;
option_data.put_price=8.4;

option_data

option_data.DTE
```

Remark 18. **...** – In MATLAB, when you have lengthy expressions, you can continue on the next line by breaking the current line with the three-dot operator (...). See the solution of ex. 33 for an example.

You can also create arrays of structures to handle complex databases (e.g. vanilla options data). The indexing operator is the parentheses (()). We will usually only use one index to classify structures (i.e. *structure row-vectors* or *structure column-vectors*), although we could define structure matrices. The fields are always identical throughout the elements of a structure array. If you add a new field to a specific element, this field will be initiated with an empty matrix for all other elements. *The dot operator follows the indexing operators.* For example, you could call the field *field_name* of the ii^{th} element of a structure vector with:

```
structure_name(ii).field_name
```

Exercise 34. Add a second element to *option_data* containing the following values:

- today_date is '15-may-2010';
- exp_date is '17-may-2010';
- DTE is 2;
- underlying is 'SPX';
- strike is 1300;
- call_price is 0.2;
- put_price is 108.2;

Apply *length* to *option_data*. Add a field *exchange* containing **CBOE** for the second element of option_data. What is the value of *exchange* for *option_data*(1)?

Solution 34.
```
option_data(2).today_date='15-may-2010';
option_data(2).exp_date='17-may-2010';
option_data(2).DTE=2;
option_data(2).underlying='SPX';
option_data(2).strike=1300;
option_data(2).call_price= 0.2;
option_data(2).put_price= 108.2;

length(option_data)
option_data(2).exchange='CBOE';

option_data(1).exchange %is the empty matrix!
```

5 Importing data

For this part of the tutorial, you will need Excel (or any spreadsheet that can save under a comma delimited format (.csv) and/or a tab delimited text format (.txt)).

date	price
13-may-2008	10.2
14-may-2008	11.35
15-may-2008	12.01
18-may-2008	8.67
19-may-2008	9.31

Table 1. Basic data set

Save the basic and advanced data sets (see table 1 & 2) as *basdataset* and *advdataset*, respectively, in the following formats:

stock_ID	stock _ticker	date	price	exchange_ID
1123	AAPL	13-may-2008	54.77	G
1123	AAPL	14-may-2008	55.32	G
1123	AAPL	15-may-2008	53.12	G
1123	AAPL	16-may-2008	52.01	G
51864	RIM	13-may-2008	102.03	A
51864	RIM	14-may-2008	102.32	A
51864	RIM	15-may-2008	107.54	A
51864	RIM	16-may-2008	NaN	A
5428	GENZ	13-may-2008	56.43	E
5428	GENZ	14-may-2008	52.53	E
5428	GENZ	15-may-2008	53.21	E
5428	GENZ	16-may-2008	56.01	E
8436	AMZN	13-may-2008	23.43	E
8436	AMZN	14-may-2008	24.53	E
8436	AMZN	15-may-2008	24.07	E
8436	AMZN	16-may-2008	26.21	E
48555	MFST	13-may-2008	21.07	G
48555	MFST	14-may-2008	22.21	G
48555	MFST	15-may-2008		G
48555	MFST	16-may-2008	23.12	G
15689	GOOG	20080513	634.21	F
15689	GOOG	20080514	650.20	F
15689	GOOG	20080515	649.00	F
15689	GOOG	20080516	637.80	F

Table 2. Advanced data set

– Excel workbook (.xlsx);
– comma delimited format (.csv) ;
– tab delimited text format (.txt).

5.1 NaNs

NaN is a scalar representing the *Not-a-Number* state. NaNs naturally appear in corrupted database. Also, when a number is missing in a database, MATLAB will replace the entry by a NaN. When passing arrays containing NaNs to built-in MATLAB functions, the NaN will propagate through the output. Indeed, almost all basic operations are unspecified for NaNs. What is NaN*2? NaN! Try it! Also note that indexing with NaNs will generate an error.

Exercise 35. Write NaN in the command window. Assign NaN to the identifer *data*. Test the NaN state on *data* using *isnan*. Overwrite *data* with a 10-by-10 matrix of NaNs using *nan*. Try 2*a, mean(a), sort(a). Let b=[1;2;3]. Try indexing *b* with NaN.

Solution 35.
```
NaN
a=NaN
isnan(a)
a=nan(10,10)

2*a
mean(a)
sort(a)

b=[1;2;3];
b(NaN) %An index cannot be a NaN! Error!
```

When performing basic operations containing both valid scalars and NaNs, the output will generally be a NaN. This is known as *NaN contamination*.

Exercise 36. Let A=(1:100). Set A(5)=NaN and A(57)=NaN. Apply *sum*, *mean* and *std* to A. Test if A is equal to 6 elementwise.

Solution 36.
```
A=(1:100);
A(5)=NaN;
A(57)=NaN;
sum(A)
mean(A)
std(A)
(A==6)
```

Since NaN is not equal to 6, MATLAB returns 0 for (NaN==6). Most statements will return FALSE when passed a NaN. *Logical statements with NaNs will not generate error messages*!

Exercise 37. With A as previously (ex. 36), compute the correct sum, mean and standard deviation by disregarding NaNs with *isnan*. Use *nanmean* and *nanstd* to perform the same tasks.

Solution 37.
```
sum(A(~isnan(A)))
mean(A(~isnan(A)))
std(A(~isnan(A)))
nanmean(A)
nanstd(A)
```

Remark 19. **Is NaN equal to NaN?** – No! NaN is the only element in MATLAB which is not equal to itself!

Exercise 38. Rewrite *isnan* using Rem. 19. Create an anonymous function to hold your new function (i.e. a function handle). Apply it to A and compare your result with the one from *isnan*.

Solution 38.
```
myisnan=@(x) x~=x;
myisnan(A)
all(myisnan(A)==isnan(A))
```

5.2 Dates

MATLAB has its own date conventions. It basically assigns an integer, the *serial date number*, to each calendar days. Decimals can be used to specify intraday timestamps. MATLAB also supports date as vectors of components with a predefined format:

```
[Year,Month,Day,Hour,Minute,Second]
```

Hour, *minute* and *second are* mandatory, but can be set to zero when dealing with days only. Some useful functions are:

- **datenum:** convert date and time to serial date number;
- **datestr:** convert date and time to string format;
- **datevec:** convert date and time to vector of components.

datenum is a very flexible converting function that takes strings (and cell arrays of strings) as inputs. You can specify any formats by using a combination of **d**, **m**, **y** and delimiting characters (e.g. **-**, whitespace, **,**,...). The formats you will encounter most in financial databases are **dd-mm-yyyy**, **yyyymmdd** and **mm/dd/yyyy**. See the MATLAB help for more information on how to specify more complex formats, if the need arises.

Exercise 39. Convert '13-may-2008' into a serial date number using *datenum*. Convert the result into a vector.

Solution 39.
```
date_matlab=datenum('13-may-2008','dd-mmm-yyyy')
date_vect=datevec(date_matlab)
```

Serial date numbers and vectors of components are simply integers and arrays of integers. As such, you can perform usual logical statements or operations on them. Never perform logical test on strings of dates. Converting them to *serial date numbers* first is more convenient. To perform operations on years, months or days, use vector of components. MATLAB will *automaticaly* make the necessary adjustments when passed a vector of components for which the *number of months is greater than 12* or *the number of days is greater than the number of days in the current month*. When MATLAB adjusts for faulty *month* or *day*, it first resolves the conflict for *month* and, then, for *day*.

Exercise 40. Assign *date1* to '13-may-2008' and *date2* to '05-jan-2001'. Test if the two dates are equal. Add 640 days to *date1*. Add two years, 14 months and 63 days to *date2*. Print the results in strings.

Solution 40.
```
date1='13-may-2008';
date2='05-jan-2001';

date1_mat=datenum(date1,'dd-mmm-yyyy');
date2_mat=datenum(date2,'dd-mmm-yyyy');
%where _mat stands for matlab format
% not matrix!

(date1_mat==date2_mat)

datestr(date1_mat+640)
%adding days directly to the matlab format (serial date numbers)

date2_vec=datevec(date2_mat);

datestr([date2_vec(1)+2 date2_vec(2)+14 date2_vec(3)+63 0 0 0])
```

5.3 Excel: Basic

To import from Excel workbooks, use the function *xlsread*.

Remark 20. **Specifying paths to data files** – For MATLAB to know which file to open, you will often have to specify paths in string variables. If a file is in the MATLAB path, you can only specify its name and extension. Otherwise, you have to specify the whole path. For example,

- Name and extension *only*:

  ```
  fileid = 'testdataset.xlsx'
  ```

- Whole path:

  ```
  fileid = 'C:\MATLAB\tutorial\PART2\testdataset\testdataset.xlsx'
  ```

Exercise 41. Make sure *basdataset.xlsx* is in your path. Import *'basdataset.xlsx'* using *xlsread*. Assign the data set to *bas_xlsx_num*. Remove *basdataset.xlsx* from your path. Try again! What is the error? Call *xlsread* by passing the whole path to *basdataset.xlsx* as input (this will be system-dependent!). Print the result. Don't forget to add back *basdataset.xlsx* to your path.

Solution 41.
```
bas_xlsx_num=xlsread('basdataset.xlsx');
```

xlsread offers three outputs: numeric only, alphanumeric only and mixed. The first one contains numeric values only. It disregards columns containing only alphanumeric characters. If a column is not disregarded, but still contains alphanumeric characters, the non-numeric entries will be replaced by NaNs. The second output (*alphanumeric only*) contains all the text entries from the imported file, with numeric values disregarded (converted into empty strings). *This behavior was modified in MATLAB R2012b, please refer to the help.* Finally, the mixed output contains the raw data in a cell array. In other words, it contains every data point regardless of the (variable) type.

For advanced programmers, we strongly suggest using the raw output when confronted with faulty or complex databases since it maintains the structure of the database, no matter the type of each entries. For newcomers, we suggest correcting, or pre-processing, databases in a more friendly environment, such as Excel. You could then import *only* numeric arrays through the first output of *xlsread*.

Remark 21. **Disregarding outputs** – You will encounter situations where you are only interested in the third or fifth output of a function. If the previous outputs are not needed, you should not waste computational resources assigning them to identifiers. To disregard an output, replace the identifier by a tilde ($\sim$) operator. Don't forget to separate outputs with commas! For example,

```
[~,out2] = function_name(inputs);
%disregarding first output
```

Exercise 42. Import *basdataset.xlsx* using *xlsread*. Retrieve all outputs in the following identifiers *[bas_xlsx_num, bas_xlsx_txt, bas_xlsx_raw]*. Compare the three outputs. Try importing only the raw data by using the tilde operator ($\sim$).

Solution 42.
```
[xlsx_num, xlsx_txt, xlsx_raw]=xlsread('basdataset.xlsx');
[~, ~, xlsx_raw]=xlsread('basdataset.xlsx');
```

Exercise 43. Compute the log returns of the time series contained in the basic data set using the first (numeric) output of *xlsread*. Are the timestamps (dates) increasing or decreasing?

Solution 43.
```
bas_xlsx_num=xlsread('basdataset.xlsx');
timeseries=bas_xlsx_num; %no header in the numeric output
logret=diff(log(timeseries)); %increasing timestamps
```

Exercise 44. Convert dates from the basic data set in serial date numbers. *Tip:* Use the second output (alphanumeric only) of *xlsread*. Verify that the conversion was accurate.

Solution 44.
```
[bas_xlsx_num, bas_xlsx_txt]=xlsread('basdataset.xlsx');
date_str_tmp=bas_xlsx_txt(2:end,1); %discarding header
date_mat=datenum(date_str_tmp,'mm/dd/yyyy');
datestr(date_mat) %verification
```

5.4 Excel: Advanced

Remark 22. ***unique*** – When importing databases, you will often be confronted with columns containing multiple instances of the same value. For example, the column *stock_ID* indexes the different stocks. Since we have multiple prices for each stocks, the same indices appear multiple times! What if you are interested in the number of stocks contained in the data set. Use the built-in MATLAB function *unique* on this column to retrieve one instance of each index. You can then compute the length of the resulting vector to determine the number of stocks in the database.

Remark 23. ***Alphanumeric NaNs*** – Our advanced data set contains a **NaN**. MATLAB sees this entry as a string of length 3! When importing numeric entries only, *xlsread* will replace every **NaN** by a numeric NaN (as any string entry!). Nevertheless, if you import through the raw output, you will have to convert **NaN**s (the strings) into NaNs (the *Not-a-Number*s).

Exercise 45. Process the data from *adv_xlsx_raw*. First, discard headers. Secondly, replace alphanumeric **NaN**s by numeric NaNs. Discard **stock ticker** and **exchange ID**. Classify the data by stocks in a cell vector by using *unique*. Create an array of string containing stock tickers indexing your cell vector of data.

Solution 45.
```
[~, ~, adv_xlsx_raw]=xlsread('advdataset.xlsx');
headers=adv_xlsx_raw(1,:);
adv_xlsx_raw=adv_xlsx_raw(2:end,:);

stockID=cell2mat(adv_xlsx_raw(:,1));
stockID_u=unique(stockID); %adding _u to the identifier.

bystock=cell(length(stockID_u),1); %preallocating!
bystock_index=cell(length(stockID_u),1); %preallocating!

for id=1:length(stockID_u)

   selected_data_tmp=adv_xlsx_raw(stockID_u(id)==stockID,:);
   bystock{id}=selected_data_tmp(:,[ 1 3 4 ]);
   bystock_index(id)=selected_data_tmp(1,2)
```

```
%creating an index with the stock tickers
%why are we using parentheses to index a cell here?

%Eliminating alphanumeric NaNs
%logical indicating strings in the price column
   alpha_nan_id=cellfun(@(x) ~isnumeric(x),bystock{id}(:,3));
%if any string in price column
       if any(alpha_nan_id)
       bystock{id}{alpha_nan_id,3}=NaN; %replace by NaN
   end

end
```

Verify that *bystock_index* correctly index *bystock*.

Remark 24. ***num2str*** – When importing dates in formats that do not contain delimiters (such as **yyyymmdd**), MATLAB will store them as *numeric* scalars. To convert them to serial date numbers, *datenum* asks for a string as input. First, convert the numbers to strings using *num2str*. Then, pass the resulting strings as inputs to *datenum* to generate the corresponding serial date numbers.

Exercise 46. From the processed data (*bystock*), create a cell column-vector, *date_bystock*, containing the date time series for each stocks. Your dates should be in a serial date number format.

Solution 46.
```
date_bystock=cell(size(bystock));
for id=1:length(bystock)

    date_tmp=cell2mat(bystock{id}(:,2));

    if isnumeric(date_tmp(1)) %format yyyymmdd
         date_bystock{id}=datenum(num2str(date_tmp),'yyyymmdd');
    else %format mm/dd/yyyy
        date_bystock{id}=datenum(date_tmp,'mm/dd/yyyy');
    end

end
```

Exercise 47. From the processed data (*bystock*), create a cell column-vector, *ret_bystock*, containing the log-return time series for each stocks.

Solution 47.
```
ret_bystock=cell(size(bystock));

for id=1:length(bystock)
    ret_tmp=cell2mat(bystock{id}(:,3));

    ret_bystock{id}=diff(log(ret_tmp));
end
```

Exercise 48. As you might have already noticed, the dates are the same for each stock (this is not the case for all databases, be careful!). Create a matrix that contains the data from the advanced data set. Fill the first column with the dates. Each subsequent columns should contain the stock log-returns, where the second column correspond to the first index in *bystock_index* (i.e. keep your indexing structure). *Tip:* Assume the prices were *close prices*, so that you can disregard the first date.

Solution 48.
```
processed_data=zeros(length(ret_bystock{1}),length(ret_bystock)+1);
processed_data(:,1)=date_bystock{1}(2:end); %filling dates

for id=1:length(bystock)
   processed_data(:,id+1)=ret_bystock{id}; %filling returns
end
```

Note how it is difficult to vectorize when manipulating cell arrays. Here, you have no choice but to use *for* loops. Congratulations, you pre-processed your first database in MATLAB!

5.5 Text files (comma or tab delimited)

When importing from text files containing both numeric values and strings, you can use the generic function *importdata* (in fact, you could even use it on Excel workbooks). This is a 'jack of all trades, master of none' function. It can usually handle simple data sets, but will be instable for more complex data sets. It first *tries* to detect if the data has column or row headers. It then returns a structure with at least two-fields. The first field, *data*, contains the numeric data only (as the first output of *xlsread*). The second field, *textdata*, contains the text data only (as the second output of *xlsread*). If *importdata* detects headers, it will output them in separate fields.

Exercise 49. Import the basic data set from the comma-delimited file (.csv), from the tab delimited file (.txt) and from the Excel workbook (.xlsx) using *importdata*. Assign the respective results to *bas_csv*, *bas_txt* and *bas_xls*. What is the output variable type? Explore the data.

Solution 49.
```
bas_csv=importdata('basdataset.csv');
bas_csv
bas_csv.data
%did not detect any headers

bas_txt=importdata('basdataset.txt');
bas_txt
bas_txt.data
bas_txt.colheaders
%retrieved the wrong column headers!
```

```
bas_xls=importdata('basdataset.xlsx');
bas_xls
bas_xls.data.basdataset
%importdata outputs a nested structure!
```

Note how the output from *importdata* differs across file formats. It is very unstable!

importdata might not detect headers correctly for mixed data (such as the advanced data set). Worst, it might not recognize the correct delimiter and output whole lines of data. Under those circumstances, consider pre-processing the data, either by using low-level built-in MATLAB functions or through Excel.

Exercise 50. Try importing the advanced data set with *importdata*. Is it the output you expected? Probably not!

Solution 50.
```
adv_csv=importdata('advdataset.csv');
adv_txt=importdata('advdataset.txt');
```

For files containing *numeric values only*, you can also use *csvread* to import comma delimited files. This is the safest and more efficient way to import data in MATLAB. Consider pre-processing your data in Excel to eliminate any strings. Save the pre-processed data as a comma delimited file and import it through *csvread*.

Exercise 51. In Excel, create a workbook containing two columns of numbers (without headers!). Save it as *valueonly.csv*. Import the file using *csvread*. Now try doing the same for 'basdataset.csv' (which contain strings!).

Solution 51.
```
csvread('valueonly.csv')

csvread('basdataset.csv') %strings not supported!
```

Remark 25. ***Advanced:*** **low-level reading functions** – You could read data from a single or multiple lines with low-level built-in MATLAB functions:

- Open a file (*fopen*) with reading permissions (set *permission* to *'r'*). Assign the output to the identifier of your file (e.g. *fileID*);
- Use *textscan* to scan a line or multiple lines from *fileID*. Specify under which format to import;
- Once you are done, close your file with *fclose*.

You can also navigate through files with *fseek*, *ftell* and *frewind*. To test if the *end-of-the-file* was reached, use *feof*.

Remark 26. ***Advanced:*** **specifying formats** – When specifying importing (or exporting) formats, you have to replace any expected variable type by the following:

- **%f**: for numbers;

– **%s**: for strings.

and separate them by the correct delimiter (if the function has not already a built-in *delimiter* input, e.g. *textscan*). Basic delimiters are:

– **whitespace**;
– **\t** (the tab delimiter);
– **,**.

Every line should end by the *end-of-line* character *\n*.

Let's say you expect the following database structure:

```
APPL 658.32, 100
```

You would then specify the format as

```
'%s %f, %f\n'
```

Exercise 52. Let str_model='The current model is ' and model_ID=3. Print in the command window using fprintf (see help): **The current model is 3**.

Solution 52.
```
str_model='The current model is ';
model_ID=3;
fprintf('%s %f \n',str_model, model_ID);
```

Remark 27. ***Advanced: textscan* and delimiters** – MATLAB will assume a *whitespace* as the default delimiter when using *textscan*. If you have a comma-delimited or tab delimited file, set *delimiter* to ',' or '\t' respectively:

```
line=textscan(fID,'%s %f %s %f %s\n',1,'delimiter',',');
line=textscan(fID,'%s %f %s %f %s\n',1,'delimiter','\t');
```

When specifying a *delimiter* to *textscan*, separate %s and %f with *whitespaces* ('e.g. %s %f %s %f %s\n'), not the delimiter!

Exercise 53. Advanced: From the advanced data set (in comma-delimited format), read the first line and assign it to *headers*. Read the remaining lines line-by-line and assign the result to *data*. Explore *headers* and *data* in the variable editor.

Solution 53.
```
adv_fID=fopen('advdataset.csv');
frewind(adv_fID); %not mandatory
headers=textscan(adv_fID,'%s %s %s %s %s\n',1,'delimiter', ',');

data={}; %the output from textscan is a cell array
while(~feof(adv_fID))
line=textscan(adv_fID,'%f %s %s %f %s\n',1,'delimiter', ',') %reading one line
data=[data;line]; %concatenating cells
end
```

6 Exporting data

Exporting from MATLAB is more straightforward than importing. Indeed, formats in MATLAB are uniformized (matrices, vectors, cells, ...). This is in stark contrast with external databases that can get quite messy. To export data in an Excel workbook, use the function *xlswrite*.

Exercise 54. Generate a 1000-by-3 matrix of random returns and save it in an Excel workbook.

Solution 54.
```
ret=randn(1000,3)*0.20/252+0.06/252;
xlswrite('random_xls',ret);
```

MATLAB also offers *.mat* files to easily store data generated from the workspace. For example, you could wish to save a matrix of results for further references. To do so, use the function *save*.

Exercise 55. Save the matrix of returns from the previous exercise in a *.mat* file.

Solution 55.
```
save('random_mat','ret');
```

You can also save the whole workspace at once (see *save* in the help for more information).

Remark 28. ***Advanced:* low-level exporting functions** – To write data in a text file:

- Open a file (*fopen*) with writing permissions (set *permission* to 'w'). Assign the output to an identifier (e.g. *fileID*);
- Use *fprintf* to either write a single line or multiple lines at once. Specify the desired format of the lines;
- Once you are done, close your file with *fclose*.

Remark 29. ***Advanced: fprintf*** – If no file identifier is passed, *fprintf* will print in the command window. Otherwise, it will print in the desired file.

Exercise 56. Advanced: Using low-level functions, write the matrix of returns, *ret*, in a text file.

Solution 56.
```
fileID=fopen('random_txt','w');
fprintf(fileID,'%f %f %f\n',ret);
fclose(fileID);
```

7 Multidimensional arrays

Multidimensional array refers to any variable indexed by more than one integer. Thus, matrices can be seen as multidimensional arrays with two indices. Under specific circumstances, you might be interested in creating variables indexed by three integers (rarely by four or more). The first dimension refers to the rows

(vertical), the second to the columns (horizontal) and the third to the *pages*. You could visualize such a variable as a *cube*. For example, let's say you have yearly matrices of cross sectional data (perhaps, countries and sectors). It is natural to maintain the matrix structure of the data. Thus, we would assign the countries to the first index (rows) and sectors to the second index (columns) (or vice versa). We would then assign years to pages such that we can browse through time with the third index.

Exercise 57. Let's say you have data on exports (million USD) generated by three countries for four specific sectors for two years (yearly data). The first year, the corresponding exports are [14 22 42 65 ; 34 43 23 12; 15 56 98 32] and, the second year, [52 66 94 64; 73 82 43 12; 85 24 57 61] (where rows are countries and columns sectors). Create a three-dimensional array with two pages to hold this data. Print it in the command window and explore it in the variable editor. Is the array presented in the command window in the same format as in the variable editor? Retrieve the following data:

- exports of all sectors for the first country (let's say Canada) for both years;
- exports for the industrial sector (second columns) for all countries for the first year;
- all exports for the second year;

What is the expected size of each output?

Solution 57.
```
exportUSD(:,:,1)=[14 22 42 65 ; 34 43 23 12; 15 56 98 32];
exportUSD(:,:,2)=[52 66 94 64; 73 82 43 12; 85 24 57 61];

exportUSD(1,:,:)  %exports of Canada
%a row-vector for each pages

exportUSD(:,2,1) %first year of data for the industrial sector
%a column-vector

exportUSD(:,:,2) %all exports for the second year
%a matrix
```
Yes, they are presented in the same format. MATLAB will always break down three-dimensional arrays into pages when displaying them. *To preserve the matrix structure of the data, we must request whole pages, as in exportUSD(:,:,2)* .

When passing multidimensional arrays to functions operating on vectors, such as *sum*, *cumsum*, *diff*, *sort* and *max/min*, you must specify on which dimension to operate through *dim* (which is usually the second input of built-in MATLAB functions). Setting *dim* to 3 for a three-dimensional array will apply the function on pages. The size of the resulting output will be the size of one page. The following exercise (58) should make this more concrete.

Remark 30. **Specifying dimensions** – You will often initiate matrices using functions such as *ones*, *zeros*, *randn*, *rand*, etc... When specifying the desired size, both of the following syntaxes will generate a m-by-n-by-p-by-... multidimensional array:

```
ones(m,n,p,...);
ones([m n p ...]);
```

Since *size* returns vectors of the type [m n p ...], we usually prefer to use the latter format (namely *ones([m n p ...])*).

Exercise 58. Create 10 pages of 2-by-2 matrices containing normalized gaussian random variables, using *randn*. Retreive the fifth page. Compute the sum along all three dimensions, using *sum*. What is the expected size of the resulting variable when *dim* is 1, 2 and 3?

Solution 58.
```
gaussian_rv=randn([2 2 10]);
gaussian_rv(:,:,5)

sum(gaussian_rv,1) %cumulative sum on the 2 rows (vertically)
sum(gaussian_rv,2) %cumulative sum on the 2 columns (horizontally)
sum(gaussian_rv,3) %cumulative sum on the 10 pages
```
We expect the following sizes for the output of *sum*:

- *dim* is 1: 10 pages of row-vectors (of length 2);
- *dim* is 2: 10 pages of column-vectors (of length 2);
- *dim* is 3: a 2-by-2 matrix.

8 Appendix A: Debugging and Profiler

When MATLAB encounters a faulty line in scripts or functions, it will display an error message indicating the faulty line number. Sometimes, the error might not happen precisely at the line proposed by MATLAB. Debugging tools might be useful in those circumstances. To enter the debbuging mode, first click on the left margin of the editor to add a breakpoint. You should add breakpoints at lines preceding an expected faulty block of lines. When you execute the code, MATLAB will reach your breakpoint and will 'break' the execution. You then automatically enter the debbuging mode (see the K before >> in the command window). In this mode, you can use debugging functions (type debugging in the MATLAB help). For example, use *dbstep* to execute your code line by line or *dbcont* to go to the next breakpoint.

To speed up your code, you can use the profiler. Type a function or script in the profiler to perform an analysis of the computational burden (in time units, usually seconds) of each function called by your code. You will then know where to put your efforts to increase performance!

9 Appendix B: Input and Output checks

Your functions should be robust to faulty inputs. You might code a function assuming that the input is a column-vector. What if someone else use your function and assume it is row-vector or, worst, a matrix? Errors will almost certainly by generated. It might not be obvious to the new user what is going wrong! You should always check inputs using if-statements and display an error message explaining the issue. See the MATLAB built-in function *error* to display an error message and terminate a function.

It is also part of good coding practices to include a header of comments explaining the nature of each input. What is the expected type (row-vector, column-vector, matrix)? What is the expected length/size (if any) ?

Remark 31. ***nargin, nargout*** – *nargin* and *nargout* are automatically assigned by MATLAB in all functions. *nargin* represents the number of inputs and *nargout* the number of outputs. Use *nargin* to test if the user has entered the correct number of inputs. You could also define default values if inputs are missing.

Exercise 59. Add robustness to your function scalar_product.

Solution 59.
```
function [var3] = scalar_product(var1, var2)
%var1 and var 2 should be two scalars

if nargin<2
error('too few inputs.')
end

if ~isscalar(var1) || ~isscalar(var2)
error('var1 and var2 must be scalars.')
end

var3=var1*var2;

end
```

Exercise 60. Set the default value of *var2* in *scalar_product* to 1. Try calling *scalar_product*(10).

Solution 60.
```
function [var3] = scalar_product(var1, var2)
%var1 and var 2 should be two scalars

if nargin<2
var2=1;
end

if nargin<1
```

```
error('too few inputs.')
end

if ~isscalar(var1) || ~isscalar(var2)
error('var1 and var2 must be scalars.')
end

var3=var1*var2;

end
```