User-Defined Functions

1 Creating a User-Defined Function

A user-defined function is a Matlab program created by the user, saved as a Matlab file, which can be used just like a built-in function.

One technical detail before getting into how to write a function is to make sure that the saved Matlab file that contains the function is in the same directory folder that the user is working in. Otherwise, Matlab will not recognize the function.

A function file consists of 4 parts:

1.1 The Function Definition Line

This must be the first executable line of the function file. (Comments can be written above it but it is not very conventional.) The main syntax for the function definition is as follows:

```
function [out1, out2, ...] = myFunc(in1, in2, ...)
```

- The line should start with the `function` keyword.
- Outputs names are defined and listed in square brackets after the keyword.
- myFunc is the function name. Every user-defined function should have a unique name. This name can contain only alphanumeric characters and an underscore. **Keep in mind that when you save the m-file, the file name should be the same with the function name!**
- Input names are defined and listed in parantheses after the keyword.

The number of inputs and outputs of a user-defined function can be any number the user wants, including zero outputs and/or zero inputs.

1.2 The H1 line

This is the first comment line that comes right after the function definition line. It will be the line of text that appears when a user types `help myFunc` and `lookfor myFunc` in the command line. It is expected to briefly describe what the function does, so it should be as concise and descriptive as possible. The main convention for an H1 line is as follows:

```
%MYFUNC Just an example for this lab
```

1.3 Help Text

You can add more information about the function after the H1 line, giving further details about what the function does, and more importantly defining each input and output.

1.4 Function Body

This is where you include all the calculations and operations. The input names defined in the definition line are used as variable names and the desired task for the function is implemented. A function body can be as long as necessary. **Keep in mind that by the end of the function body, all outputs (if there are any) should be assigned to a variable.**

Let's do an example now. The first function we will define will calculate the total payment after you deposit some money with an interest rate over a given number of years.

The function needs 4 inputs:

- The initial amount of money deposit.
- The interest rate over each time period
- Whether it is a monthly, quarterly or yearly interest
- The number of years after the initial deposit

There will only be one output and it is the total payment at the end of the given number of years.

```
function [total_payment] = invest(deposit, rate, period, num_years)

% INVEST Calculates the total payment over a period of time

% Input Arguments:
% deposit: The initial amount of money that is deposited
% rate: The interest rate over the unit amount of time, should be between 0 and 1.
% period: The unit amount of time for the interest rate to apply, can be a month,
%         a quarter (of a year) or a year. It is a string input, 'M' for monthly,
%         'Q' for quarterly, 'A' for annual. (yearly)
% num_years: Total number of years after the initial deposit

%Output Arguments:
%total_payment: Total money after the interest rate applies for the given amount of time

if period == 'A'
        total_payment = deposit*(1 + rate)^num_years;
elseif period == 'Q'
        total_payment = deposit*(1 + rate/4)^(4*num_years);
elseif period == 'M'
        total_payment = deposit*(1 + rate/12)^(12*num_years);
else
        error('Not a valid time period!')
end
```

There are a few things to note in this example:

- There are 4 inputs, 3 of them are scalar numbers and one is a string input. The inputs to a user-defined function can be in any type; double, float, string, vector, matrix etc.
- There is one single output for this example. You can define as many outputs as you want, as long as you assign them to a variable in the function body.
- The help text after H1 line is very detailed. It is up to the user to decide on the length and details of this comment text but it is usually preferred to be as thorough as possible so everybody can understand how the function works. (That would be helpful particularly with more complex functions.)
- The example has something that is not introduced in this document yet, the error line. In any line in the function body, you can add such an error line that will display the text written inside and stop the function from executing. (More on that in a bit!)

Let's write a similar function but without any outputs. Suppose we are given the same inputs, but instead of getting the total payment as an output, we want to plot how the deposit increases with each year. So, we don't need a numeric output, we just need to display a graph, which will be done inside the function body.

```
function invest_plot(deposit, rate, period, num_years)

% INVEST Plots the amount of money in an account with a given interest rate over time

% Input Arguments:
% deposit: The initial amount of money that is deposited
% rate: The interest rate over the unit amount of time, should be between 0 and 1.
% period: The unit amount of time for the interest rate to apply, can be a month,
%         a quarter (of a year) or a year. It is a string input, 'M' for monthly,
%         'Q' for quarterly, 'A' for annual. (yearly)
% num_years: Total number of years after the initial deposit

%Output Arguments:
% No output, just a graph shown

if period == 'A'
        unit_time = 1;
elseif period == 'Q'
        unit_time = 4;
elseif period == 'M'
        unit_time = 12;
else
        error('Not a valid time period!')
end

total_money = zeros(num_years,1);
total_money(1) = deposit;
counter = 2;

while counter <= num_years
        total_money(counter) = total_money(counter-1)*(1 + rate/unit_time)^unit_time;
        counter = counter + 1;
end

x = 1:num_years;

plot(x,total_money)
xlabel('Number of years passed')
ylabel('Total money')
title('Total Amount of Money Over Time')
```

Note that in this example, the function name comes right after the `function` keyword because there isn't any output The function just displays a plot that is built on all the calculations done with the inputs.

2 Calling the User-Defined Function

After the function is fully defined and ready to be used, it needs to be called in the command line, in a script or even inside another function. **Note that a Matlab function is different than a Matlab script.**

You run a script whereas you call a function. In other words, a script is just a number of lines of code that is run directly whereas a user-defined function is a tool that is called somewhere in a script, entered in a command line or in another function.

Now, let's put the following lines to the command line: (or on a script if you want, but remove the carrots from the beginning of each line)

```
>> clear
>> total_payment = invest(deposit, rate, period, num_years)
```

You should get an error. Why did this happen? This is because you always need to **define the inputs to the function before calling it.** It is the same type of error you would get if you would mistake a function file with a script file and try to run it directly, the function cannot work, because none of the inputs are defined. Now try the following:

```
>> clear
>> deposit = 100
>> rate = 0.15
>> period = 'Q'
>> num_years = 5
>> total_payment = invest(deposit, rate, period, num_years)
>> invest_plot(deposit, rate, period, num_years)
```

These lines of code were able to calculate the total payment after 5 years and plot the amount of money invested by each year. You just successfully defined and called a user-defined function!

There are a few points to note about calling a function before moving on to some details about functions.

Firstly, you can define variables directly while calling the functions, in other words, the following lines of code will run without a problem:

```
>> clear
>> total_payment = invest(100, 0.15, 'Q', 5)
>> invest_plot(100, 0.15, 'Q', 5)
```

You just used the values directly instead of assigning them to a variable first. This is perfectly fine, **assuming that the input order is correct!** For instance, the following code will throw the error that you defined while writing the function:

```
>> clear
>> total_payment = invest(100, 'Q',0.15, 5)
>> invest_plot(100, 'Q',0.15, 5)
```

Take a look at the variable order and then how each variable is used inside the function and try to understand yourself why the function threw that error. **The input order to a function is very important.**

Secondly, the names of the input and output variables of the function are just placeholders, meaning that when you call the function, the variables that are fed into the function or taken as the output can be named whatever you want. For example, the following code runs without any error:

```
>> clear
>> a = 100
>> b = 0.15
>> c = 'Q'
>> d = 5
>> x = invest(a, b, c, d)
```

The code follows the exact same procedure with the previous variable names and assigns its output to variable x this time. One thing to note is, even though this is technically correct, using non-descriptive variable names like a,b,c... can be confusing, especially if you are working for a big project that would require

thousands of lines of code. You certainly cannot keep in mind what all the variables stand for! Therefore, it is highly recommended to keep the variable names related to what they stand for, like in the previous codes above. This goes for the function names as well. Try to use a function name that gives an idea about what the function does, instead of a non-descriptive, generic name like myfunction, function1 etc.

The third and last point to be made before moving on is the **scope** of the variables defined inside the function. Go back and take a look at the `invest_plot` function you have defined. There are variables like `counter` and `x` you have defined inside that function but when you call the function on the command line, you won't see them defined in your workspace. This is because they are created and destroyed within the function, in other words **their scope is within the function.** Make sure you understand this distinction well, because in your script, there might be variables with the same name as the variables inside the function. Take a look at the following code:

```
>> clear
>> counter = 1000
>> invest_plot(100, 0.15, 'Q', 5)
```

After you run the code, the variable `counter` is still as defined in the command window, 1000, even though there was a variable named `counter` inside the `invest_plot` function and its final value is 5. That `counter` inside the function is just gone as the function is terminated.

These first two sections should give you a complete picture on how to define a function and how to call it on a script/command line/another function. The last section will go into details of a user-defined function and show how you can hone your user-defined function even further, so that it won't be any different than a built-in function in Matlab. **Writing good functions is an important first step for any specialization that requires software development!**

3 Improving the User-Defined Function

3.1 Number of Input and Output Arguments

A useful tool before you start improving your functions is the number of input and output arguments to your function. In Matlab, they are associated with two words: `nargin` and `nargout`. Suppose you want to know how many inputs and outputs a function has. One way is always opening the file and look it up but there is a more practical way with these keywords. Making sure that the `invest` and `invest_plot` functions are still in your directory file, try the following code:

```
>> num_inputs = nargin(@invest)
>> num_outputs = nargout(@invest)
```

With nargin and nargout, you can find the number of input and output arguments to any function. In the code, you see that a @ character is used before the function name. In Matlab, @ stands for a **function handle**, a tool that allows a defined function name to be assigned to a variable name or to be used as direct input to another function, like an integer or string. The full usage of a function handle is beyond the scope of this course but for now, remember it as a piece of necessary notation for using the function `invest` as an input to `nargin` and `nargout`.

You can try the same lines with `invest_plot`.

3.2 Default Input Values

You just learned about the keyword nargin, now you will use it to assign default values to the function inputs.

Take a look at the following lines of code first:

```
>> A = [1, 2, 3; 4, 5, 6; 7, 8, 9];
>> s1 = sum(A)
>> s2 = sum(A,2)
```

The built-in sum function works both with one and two inputs. Try using the functions you have defined with 3 or 2 inputs. You will get the error `"Not enough input arguments"`. Furthermore, the s1 and s2 are different when the code is run. The reason for both of these is that **Matlab assigned a default value of 1 to the second input argument.** This means that whenever that second input is missing, the function assumes that it is 1. If it is entered by the user, the function uses the value that the user has entered.

Now, let's implement this in the `invest` function. We want the default number of years to be 5 and the default period of time to be a quarter. In order to add these defaults, the keyword `nargin` should be used, but in a slightly different way:

```
function [total_payment] = invest(deposit, rate, period, num_years)

% INVEST Calculates the total payment over a period of time

% Input Arguments:
% deposit: The initial amount of money that is deposited
% rate: The interest rate over the unit amount of time, should be between 0 and 1.
% period: The unit amount of time for the interest rate to apply, can be a month,
%         a quarter (of a year) or a year. It is a string input, 'M' for monthly,
%         'Q' for quarterly, 'A' for annual. (yearly)
% num_years: Total number of years after the initial deposit

%Output Arguments:
%total_payment: Total money after the interest rate applies for the given amount of time

#Adding the default values
if nargin < 4
        num_years = 5;
end
if nargin < 3
        period = 'Q';
end

if period == 'A'
        total_payment = deposit*(1 + rate)^num_years
elseif period == 'Q'
        total_payment = deposit*(1 + rate/4)^(4*num_years)
elseif period == 'M'
        total_payment = deposit*(1 + rate/12)^(12*num_years)
else
        error('Not a valid time period!')
end
```

When a function is defined, the number of inputs of that function is automatically stored in the keyword nargin **within the scope of that function** and that keyword is used to assign the default values.

Take a look at what you did to add the default input values. You first checked if the number of inputs is less than 4, if it is, that means the last input is missing, so you assigned the default value of 5 to it. Then, you checked if the number of inputs is less than 3, if it is, that means that the second last input is missing too, so you assigned the default value of 'Q' to it. You can assign any default value to as many input values as you want.

Now take a look at it the following function calls:

```
>> clear
>> total_payment1 = invest(100, 0.15, 'Q', 5)
```

```
>> total_payment2 = invest(100, 0.15, 'Q')
>> total_payment3 = invest(100, 0.15)
```

All three output values should be the same.

3.3 Adding Error and Warning Lines

Any user that uses a function (built-in or user-defined) eventually makes mistakes while calling them. They may use a wrong variable type (a string for a numeric input) or a value that does not make any sense for the task of the function. In that case, the function should throw either an error or a warning.

You have already seen the error line in this document, inside the `invest` function. The function calculates the total payment for annual, quarterly or monthly interest rate depending on the third input. However, whatif the user enters something else than 'A', 'Q' or 'M'? It can be another letter or it can even be an integer. This is why the function has an `else` line that throws an error for anything other than 'A', 'Q' or 'M'. Try the following lines and see the function throwing the error:

```
>> clear
>> total_payment1 = invest(100, 0.15, 'R', 5)
>> total_payment2 = invest(100, 0.15, 'fbdwf', 5)
>> total_payment3 = invest(100, 0.15, 154, 5)
```

Now, comment out the `else` statement and the error line. Try running the lines above again. What do you see? Take a look at the entire if statement with the else commented out. Why is the output not assigned to anything?

Error lines are important in order to cover all the possibilities of inputs a user can enter while calling a function. For the undesired inputs, the function should just throw an error and stop.

Another option is to add a warning line inside the function. A warning line displays the text written in it but unlike an error line, it does not stop the function from running. The function runs as implemented but the warning text is displayed on the command window to warn the user something might be wrong.

Let's do a modification to the invest function:

```
function [total_payment] = invest(deposit, rate, period, num_years)

% INVEST Calculates the total payment over a period of time

% Input Arguments:
% deposit: The initial amount of money that is deposited
% rate: The interest rate over the unit amount of time, should be between 0 and 1.
% period: The unit amount of time for the interest rate to apply, can be a month,
%         a quarter (of a year) or a year. It is a string input, 'M' for monthly,
%         'Q' for quarterly, 'A' for annual. (yearly)
% num_years: Total number of years after the initial deposit

%Output Arguments:
%total_payment: Total money after the interest rate applies for the given amount of time

#Adding the default values
if nargin < 4
num_years = 5;
end
if nargin < 3
period = 'Q';
end
```

```
%Warning the user
if rate >= 1
        warning('Interest rate is not within a reasonable range!')
end

if period == 'A'
total_payment = deposit*(1 + rate)^num_years
elseif period == 'Q'
total_payment = deposit*(1 + rate/4)^(4*num_years)
elseif period == 'M'
total_payment = deposit*(1 + rate/12)^(12*num_years)
else
error('Not a valid time period!')
end
```

Take a look at the if statement with the warning line. If the input interest rate is bigger than 1, the function still performs the calculations and gives the output, but meanwhile warns the user about that output might not make much sense. (In fact, it won't make any sense because it is highly unlikely to find an investment with an interest rate higher than 100%.)

To see the warning line in action, try the following code:

```
>> total_payment = invest(100, 3, 'Q', 5)
```

4 Exercises

1. You have read the differences between a script file and a function file. A very common mistake people make when they first learn about user-defined functions is putting input statements inside the function body. Explain why this doesn't make any sense and actually defeats the whole purpose of creating a user-defined function. (**And please don't make this mistake in the second midterm or in the final!**)

2. For the `invest` function, you already assigned the default parameters for the third and fourth input values. Add default values to the first two as well. (Hint: Use another two if statements.) After you are done, the following line should be executed without any error:

   ```
   >> total_payment = invest
   ```

3. You have seen how to call a function on the command line and in a script. Now, let's call them inside another function.

   Define a new function called `invest_num_plot` that takes two inputs, deposit and number of years, and returns the total money as the output. It randomly picks a quarterly interest rate. While returning the output, it should also plot the amount of money by each year. **You can use five executable lines in the function body at most. Any default values or error checking is not necessary for this exercise.**

   In order to implement this function, you need to use the `invest` and `invest_plot` functions inside. This is a very common approach while writing a complicated program. You should start with writing simpler functions and then create more and more complex functions using the simpler ones as building blocks.

   Test your function with the following line. Since the rate will be random, it will be a different total payment value each time you call the function.

   ```
   >> total_payment = invest_num_plot(100, 5)
   ```

4. This is more of a conceptual challenge. Create a `random_points` function with no inputs and no outputs that randomly creates five points in the 2D plane and plots them.

   (Inside the function, you need to pick 5 random integers for the x-coordinates, 5 random integers for y-coordinates and then use Matlab's scatter function to plot these points.)

   The function should be called with the following line:

   ```
   >> random_points
   ```