Programming and automated Signal Analysis: Theory

Academic Year 2013-2014
Ilse Jonkers
Contributions by Wouter Aerts, Maarten Afschrift, Josefien Burg en Lianne Zevenbergen

Contents

1 Introduction

1.1 Computers

A computer consists of a dozen components but in general, we can say that it consists of a processor and a memory. There is **memory** that can be read but in the same time be used to write, memory that can only be read and memory that can only be used to store data. Although it might seem a bit strange, in- and output devices - mouse, monitor, keyboard, printer, etc - are also part of the 'memory': we can say the keyboard is a 'read only' memory and the monitor is a 'write only' memory. The **processor** is a different part of the computer. The processor has an executive function: it has to execute commands that will change the memory. With every command, the processor will either inspect or change the memory.

A command is therefore an action that will change the memory. These commands are themselves located in the memory (on a disc + while they are executed also in the internal memory). **A program** consists out of a long list of commands, that - when executed by the processor - have the goal of changing the memory and having a certain effect on it. Because the processor and the memory can only understand code that is written in binary code (bits, bytes using zeros and ones), programming languages are developed to simplify the programming work. Writing a program is building up this program in a specific programming language. Commands that form a program have to be formulated in a specific way, most of the time they are coded in text-form. There are a lot of different notations for coding a program and a set of 'notation-agreements' is called a programming language. A programming language is a formal language designed to communicate instructions to a machine, most often a computer. Programming can be done in a lot of different ways. The last decades, there emerged a multitude of programming languages and every programming style has its own specific programming language, also called the **syntax**. Between programming languages we differentiate imperative, object-oriented, functional and logical languages. How many programming languages exist is hard to say, it all depends on which ones you see as a true programming language: different versions, dialects, etc. An overview of different programming languages can be found on Wikipedia (*en.wikipedia.org/wiki/Listofprogramminglanguages*). It does not make any sense to learn all these languages and that is not needed since there is a lot of similarity between different languages.

1.2 Types of programming language

Programming languages that have several properties in common belong to the same 'programming paradigm'. A large group of programming languages belongs to the **imperative paradigm** (for example 'Assembler'). The computer executes calculations according to a standardized pattern of steps. Every step is a small, simple part of the calculation. This programming language thus uses a series of little steps for the execution of a calculation. Imperative languages are based on commands to change the memory. Imperative programming clearly uses the previous mentioned structure of a processor and a memory. However, other paradigms also use the processor to perform commands but this is not always as visible in the structure of the programming language, as it is in imperative programming.

The **declarative paradigm** contains functional and logical programming languages and is based on functions. Functional programming is not based on giving commands but rather on making functional connections between, for example, diverse cells in a spreadsheet. In this case the empty spreadsheet is the 'program', ready to process actual data. When inputting data, the functional connections in the spreadsheet will result in specific results. Haskell, Lisp and APL are examples of functional programming language. Logical programming uses facts and rules to figure out answers. For example: the facts that are known are 'cats have fur' and 'fur consists out of hair'. The question that can now be answered by the program is 'do cats have hair?'. Examples of the logical programming language are Prolog and SQL.

The **procedural paradigm** combines imperative programming language with the use of methods. The procedures (or methods), simply contain a series of computational steps to be carried out. Any given procedure might be called at any point during the execution of a program, including by other procedures or by itself. The commands within the procedures are written in an imperative language. Procedural programming languages include for example C, Fortran and BASIC.

The **object-oriented programming paradigm** uses procedural programming combined with objects. It is an extension of the procedural paradigm in which not only commands can be grouped to procedures (or methods) but variables can be grouped in objects. This creates a program that is often more clear due to its structure and the fact that it mirrors the reality better. For example: the simulation of an alarm system is easier to understand in terms of 'house, windows, doors and sensors' instead of variables, arrays and function. Furthermore, seen all sensors will be the same, it is easy to expand the system with extra sensors. Programming languages that use this type are Java, C++ and Pascal. Although the background of all programming languages is different, once you learn how to use one specific language, the transfer to other languages is a lot easier. The programming language you need depends on the tasks you want to perform. Each programming language has its advantages and disadvantages in the field of power of expression, availability of the implementations, reliability and efficiency. In the following chapters you will learn to use the software MATLAB. MATLAB (MATrix LABoratory) is a technical software environment that is used for all kinds of mathematical applications as calculation of functions, editing of matrices, statistics, plotting of functions and creating graphical user interfaces. MATLAB uses the programming language 'M-code' (or 'M'). This code is used for the input, editing and executing of data and supports both imperative and declarative programming paradigms.

1.3 Translation

Once a computer program is written, it has to be 'translated' to be used on a specific computer. Dependent on the circumstances, this is done using a **compiler** or an **interpreter**. All programming languages (except for Assembler) can be written, independent of the used computer. To convert the programming language to a machine-code that the specific processor understands, we use a **compiler**. This compiler is computer (or better: processor) specific, since it needs to know the machine-code of that specific computer. This way of translating the programming language (also: source code) is used in most programming languages as for example C and C++. A more direct way to translate source code is by using an **interpreter**. The interpreter is a program that reads the source code and executes the commands it contains. Also this interpreter is machine-specific, but the source code remains machine-independent. The advantage of an interpreter is that there is no need to translate the source code first before it can even start performing the tasks. However, the program will proceed slower and possible mistakes are reported later. Some programming languages combine the use of a compiler and an interpreter, for example Java. Java programs are used to be spread on the internet. However, spreading a compiled version is not useful since this compiled code is machine specific. Spreading a source code is not always desired seen that allows everyone to view and edit the program. Therefore, the source code is translated by the compiler but only to a machine-independent intermediate language, the 'bytecode'. This bytecode is spread on the internet and can be performed by the user of the computer using an interpreter. Matlab is an interpreter, it will translate high level m-source code into low level (binary) machine instructions and executes the program in a sequential manner (line by line).

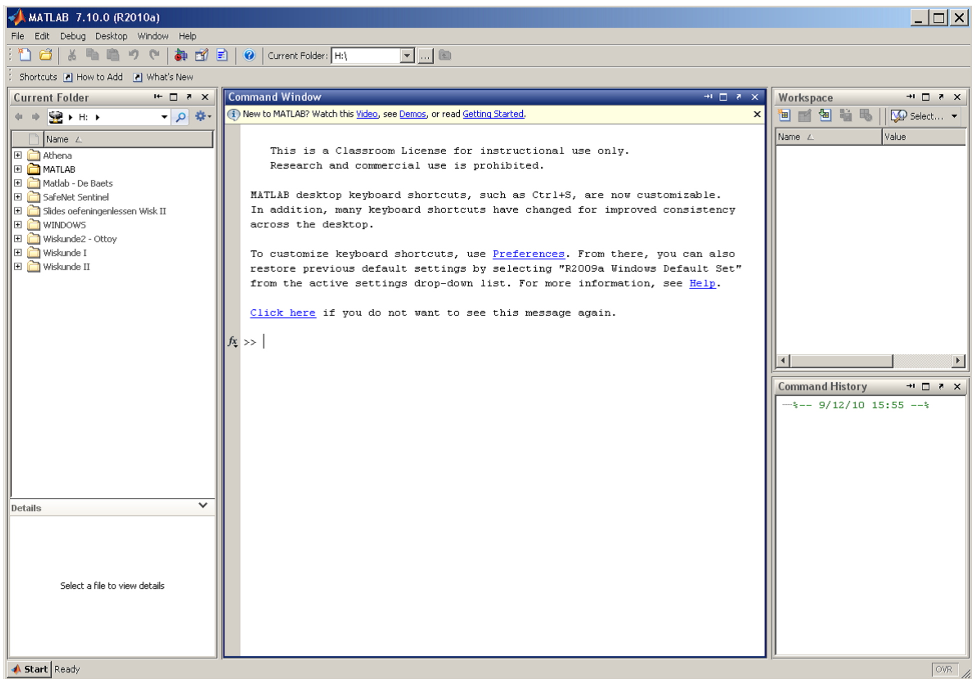
1.4 Programming

Implementing a program starts with writing the program text using the editor. Once the program is complete, the source code is presented to the compiler/interpreter. Now, the source code is first checked on syntax and/or initialization errors and in case the source code does not meet the requirements, an error will be given. In that case, you need to go back to the editor and solve the mistakes that were made. Once these errors are cleared, you will be able to run the program. In case the program does not give you the expected results, there might be a mistake in your thought process and again, you need to solve this in the editor.

Therefore, in the case of larger problems, it is better to start with modeling the problem before starting to implement it. This step allows you to divide a large problem into small sub-problems that can each be tackled separately. Using this approach, you will soon be able to code all possible problems.

2 Matlab

Start MATLAB, the MATLAB desktop appears, containing tools for managing files, variables, and applications associated with MATLAB. The first time MATLAB starts, the desktop appears as shown in the following illustration.



New to MATLAB? Watch this Video, see Demos, or read Getting Started.

```
   This is a Classroom License for instructional use only.
   Research and commercial use is prohibited.

MATLAB desktop keyboard shortcuts, such as Ctrl+S, are now customizable.
In addition, many keyboard shortcuts have changed for improved consistency
across the desktop.

To customize keyboard shortcuts, use Preferences. From there, you can also
restore previous default settings by selecting "R2009a Windows Default Set"
from the active settings drop-down list. For more information, see Help.

Click here if you do not want to see this message again.

fx >> |
```

```
%-- 9/12/10 15:55 --%
```

Figure 2.0.1: Matlab interface

2.1 Command window

The name explains itself, inside the Command Window you can enter commands that are executed by MATLAB and run functions and M-files. Every command can be entered after the prompt ('>>'). For example you can use MATLAB as a sophisticated calculator through the Command Window. Try typing 1+1, or sin(10) and then press ENTER or RETURN. The outcome will be assigned to the variable ans (answer).

```
>> 1+1
ans =
2
```

The *ans* variable is automatically created when you specify no output argument. If you define multiple statements without specifying output arguments *ans* holds the most recent answer. This means that the other answers are overridden. For assigning the value of an expression to a variable in MATLAB the variable name is placed on the left of an equal '=' sign and the expression on the right. The expression is evaluated and the result assigned to the variable name. In MATLAB, there is no need to declare a variable before assigning a value to it. If a variable has previously been assigned a value, the new value overrides the predecessor.

```
>> A = 1+1
A =
2
```

The variable names in MATLAB are case-sensitive. Thus variable C is different from variable c. A variable name can have up to 19 characters, including letters, numbers and underscores. However, variable names might not start with a number. The variables are stored in the Workspace so they can be used in other commands, functions or M-files. If you place a semicolon ';' after an expression in the command line, MATLAB performs the computation but the output is not shown in the Command Window. This is particularly useful when you generate large matrices. The Command History Window shows the instructions that are entered in the Command Window during the current MATLAB session. Previous instructions can be consulted by using the arrow buttons inside the Command Window. By pressing the upward arrow key, the statement you typed is redisplayed. By using the left and right arrow keys you can move the cursor over the statement to change it. Repeated use of the upward arrow key recalls earlier lines. You can also copy previously executed statements from the command history. To save the input and output from a MATLAB session to a file, use the *diary* function.

2.2 Workspace

The Workspace consists of the variables built up during a MATLAB session and stored in memory. Variables could be add to the Workspace by using commands in the Command Window, functions, running M-files, and loading saved workspaces. It is also possible to create new variables or import data from MATLAB supported formats in the workspace using the buttons on the top left side of the Workspace. The workspace browser gives some additional information about the size and data type of the variable. The functions *who* and *whos* could also be used to view information about each variable. Double-click on a variable in the Workspace browser to view and change its contents in the Array editor. To delete variables from the workspace, select the variable and select Delete form the

Edit menu. Alternatively, you can clear your workspace by typing clear *(variablename)* or *clear all* in the command window. The following commands create the variable A and remove it from the workspace afterwards.

```
>> A=10;
>> clear A
```

Note that the workspace is not maintained after you end the MATLAB session.

2.3 M-file

An M-file, or script file, is a simple text file where you can place MATLAB commands. When the file is run, MATLAB reads the commands and executes them exactly as it would if you had typed each command sequentially at the MATLAB command window. This is very useful if the number of commands increases, you want to save your commands or you have to change certain variables or values to find an optimal solution for your problem. All M-file names must end with the extension '.m' (e.g. test1.m). If you create a new m-file with the same name as an existing m-file, MATLAB will choose the one which appears first in the path order (type help path in the command window for more information). To make life easier, choose a name for your m-file which doesn't already exist. To see if a filename.m already exists, type *help filename* at the MATLAB prompt. To create an M-file, go to **File**>**New**>**Script** when you want to create a new M-file. Inside this M-file you can list type the commands, for example type A=1+1. Afterwards, save the M-file (press the save button in the left upper corner of the M-file tab) and execute the M-file by clicking on the green arrow (Save and Run) or press F5. You can also execute the M-file be typing the name of the M-file into the command window. The variable A will now appear in your workspace. It is very important to work in a structured way when you are programming. Therefore you can use comments inside the M-files. You can use the % sign if you want to explain a certain command, or when you write a title for the M-file. You can also use cells, which are two %% signs, to split your m-file into different blocks. To M-file below explains this way of working, be aware that this is only a very simple example.

```
% This code calculates the mean of 5 values(10,15,18,19,23)
%% calculate the sum of the values
A=10;
B=15;
C=18;
D=19;
E=23;
SUM_ABCDE=A+B+C+D+E
%% calculate the mean
MEAN_ABCDE=SUM_ABCDE/5
```

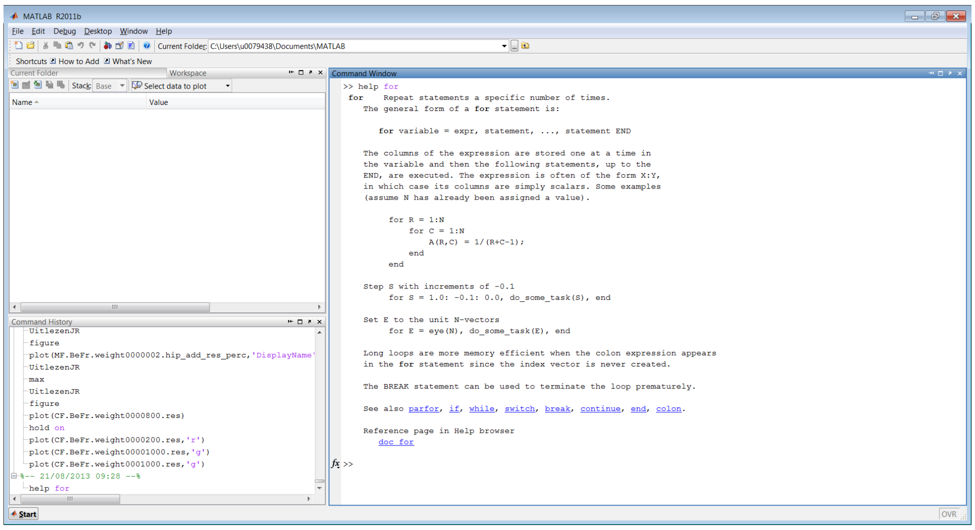
2.4 Folders

MATLAB file operations use the current directory and the search path as reference points. Any file you want to run must either be in the current directory or on the search path. This means that MATLAB is searching inside those paths for M-files, functions and other data types. When we browse to another directory it is not possible to execute the M-files saved in the previous directory because MATLAB will not find it inside the current directory. If you want to access the M-file from another folder we will have to add the folder that contains the M-file to the MATLAB search path with the function *addpath*. You can remove the path again when you use the command *rmpath*. With the command *pwd* you can get the current directory. To change the current directory the command cd is used.

```
>> addpath('path_to_the_M_file')
>> rmpath('path_to_the_M_file')
>> pwd
ans =
................ (the current path)
```

2.5 Help & Documentation

A very useful tool in MATLAB is the **help function**. In the help function you will find supporting documentation that includes examples and describes the function inputs, outputs and calling syntax. There are several ways to access this information. First, you can type *help* + what you are looking for (e.g. 'help for') in the Command Window.

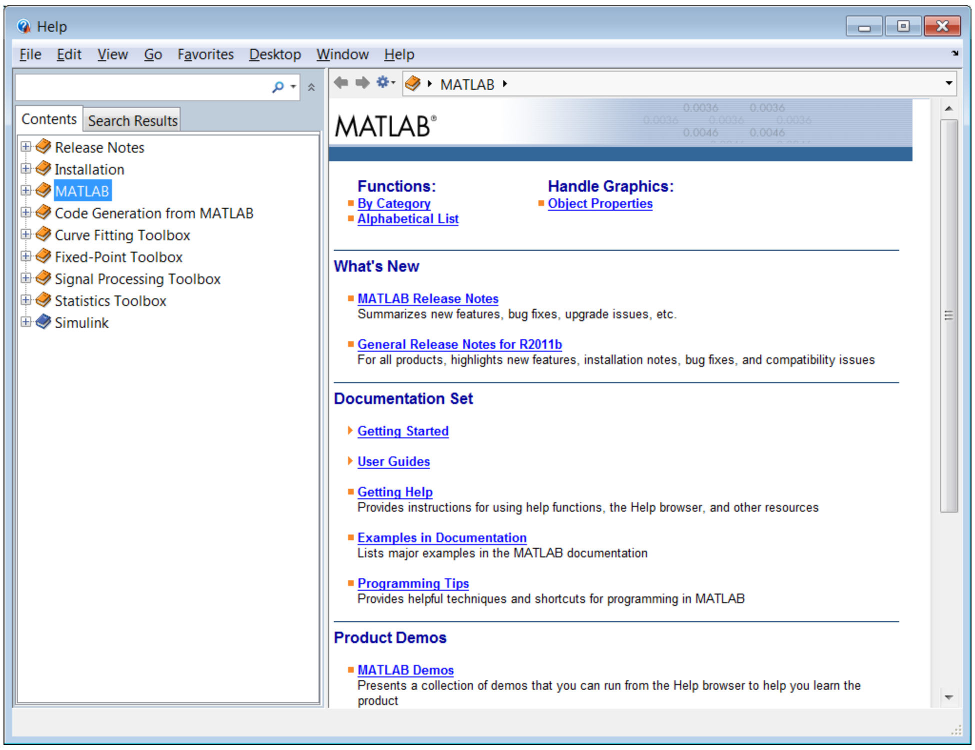

MATLAB R2011b
File Edit Debug Desktop Window Help
Current Folder: C:\Users\u0079438\Documents\MATLAB
Shortcuts How to Add What's New
Current Folder
Workspace
Stack: Base
Select data to plot
Name
Value
Command History
Command Window
Start
OVR

```
UitlezenJR
figure
plot(MF.BeFr.weight0000002.hip_add_res_perc,'DisplayName'
UitlezenJR
max
UitlezenJR
figure
plot(CF.BeFr.weight0000800.res)
hold on
plot(CF.BeFr.weight0000200.res,'r')
plot(CF.BeFr.weight00001000.res,'g')
plot(CF.BeFr.weight0001000.res,'g')
%-- 21/08/2013 09:28 --%
help for
```

```
>> help for
 for    Repeat statements a specific number of times.
    The general form of a for statement is:

       for variable = expr, statement, ..., statement END

    The columns of the expression are stored one at a time in
    the variable and then the following statements, up to the
    END, are executed. The expression is often of the form X:Y,
    in which case its columns are simply scalars. Some examples
    (assume N has already been assigned a value).

         for R = 1:N
             for C = 1:N
                 A(R,C) = 1/(R+C-1);
             end
         end

    Step S with increments of -0.1
         for S = 1.0: -0.1: 0.0, do_some_task(S), end

    Set E to the unit N-vectors
         for E = eye(N), do_some_task(E), end

    Long loops are more memory efficient when the colon expression appears
    in the for statement since the index vector is never created.

    The BREAK statement can be used to terminate the loop prematurely.

    See also parfor, if, while, switch, break, continue, end, colon.

    Reference page in Help browser
       doc for

fx >>
```

Figure 2.5.1: Matlab Help function

This will show a short description of the function. If you want more information, you can click on the hyperlink "doc" at the end of the description to go to the help page with more info or you can type doc + the function where you are looking for in the command window like help. You can also go to the help by going to the **"Help" menu** and choose **"Product Help"** or press F1.



Functions:
- By Category
- Alphabetical List

Handle Graphics:
- Object Properties

What's New

- MATLAB Release Notes
  Summarizes new features, bug fixes, upgrade issues, etc.
- General Release Notes for R2011b
  For all products, highlights new features, installation notes, bug fixes, and compatibility issues

Documentation Set

- Getting Started
- User Guides
- Getting Help
  Provides instructions for using help functions, the Help browser, and other resources
- Examples in Documentation
  Lists major examples in the MATLAB documentation
- Programming Tips
  Provides helpful techniques and shortcuts for programming in MATLAB

Product Demos

- MATLAB Demos
  Presents a collection of demos that you can run from the Help browser to help you learn the product

Figure 2.5.2: Matlab Help function

Apart from the build in help function MATLAB has an excellent user community on the internet; *http : //www.mathworks.nl/matlabcentral/*. On this website you could find all kind of examples, M-files and functions solving your MATLAB problems.

3 Data types

A data type or class is a frequently used term in programming languages. Each variable and expression is associated with a data type. This data type determines the format of the expression and the possible operations. The picture below contains the 15 fundamental data types.

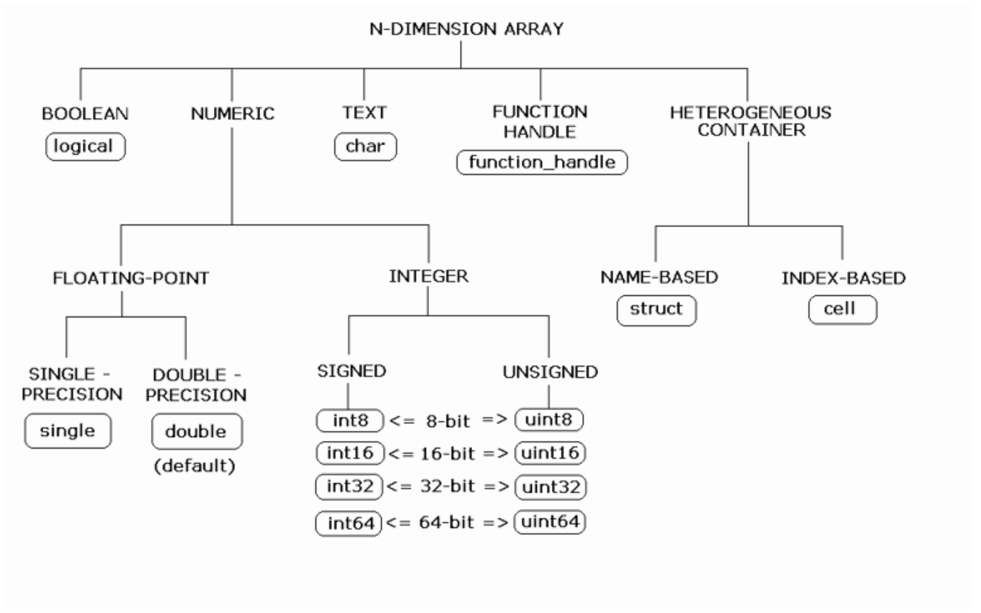


Figure 3.0.1: Data types Matlab

3.1 Integers (Int)

Variables from the data type **int** are integers. An integer is a number that can be written without a fractional or decimal component (dutch translation: 'gehele waarden'). MATLAB makes a distinction between integer types. Firstly, there are two classes; the signed and the unsigned integers. In comparison with the unsigned integers, the signed integers are able to represent positive and negative values. Both signed and unsigned integers can be stored in four different sizes: 8, 16, 32 and 64 bits. The min and max values of the *int 8*, *int16*, *int32* and *int64* can be shown by the command *intmin ('int64')* and *intmax('int64')*.

3.2 Double & single

Double is the default numeric data type in MATLAB. This data type is automatically assigned to every number. A value stored as a double-precision (or double) data type requires -64 bits and is therefore much more precise then the single-precision (or single) data type which required 32 bits.

3.3 Character & string

Characters are stored inside the "char" data type. The variables of the type **char** are recognized by using quotes.

```
>> A = 'b';
A = h
```

A string is a combination of multiple characters. This is stored in Matlab as a row vector.

```
>> B = 'biomechanics';
>> whos  B
  Name       Size                Bytes  Class    Attributes

B          1x12                  24   char
```

3.4 Boolean or logical

A Boolean or logical data type is a data type that can have to values: *'true'* or *'false'*. False is represented by a 0, true is represented by 1. You can change a double data type to a Boolean with the command logical.

```
>> a= 1;            % a is an double
>> b=logical(a)     % b is a logical
```

3.5 Matrices

MATLAB is an abbreviation for ”matrix laboratory”. While other programming languages mostly work with numbers one at a time, MATLAB is designed to operate primarily on whole matrices and arrays. As already mentioned, all MATLAB variables are multidimensional arrays, no matter what type of data. A matrix is a two-dimensional array often used for linear algebra.

3.5.1 Vector

A vector is the simplest kind of matrix. You can enter a vector of any length in MATLAB by typing a list of numbers, separated by commas or spaces, inside square brackets. For example, you can create the vector B.

```
>> B= [1 2 3 4 5 6]
```

Inside you workspace you can see the properties of the vector B. Another way to determine the properties of the variable B is with the commando **whos** (variable name) in the command window.

```
>> whos B
                         Name      Size            Bytes  Class     ...
                            Attributes
                            B         1x6                48  double
```

You can select an element from the vector B with brackets. For example you can select the second element from vector B. Selecting one, or multiple elements from a variable is called indexing.

```
>> B(2)
ans =
4
```

Suppose you want to create a vector of values running from 1 to 9. Here’s how to do to it without typing each number.

```
>> x=1:9
x =
1 2 3 4 5 6 7 8 9
```

The notation 1:9 is used to represent a vector of numbers running from 1 to 9 in increments of 1. The increment is specified as the second of three arguments.

```
>> X = 0 : 2 : 10
X =
0 2 4 6 8 10
```

The vectors above are rows vectors. You can also create column vector with the by using a semicolon ; between the numbers.

```
>> X = [ 2 ; 4; 6 ; 8]
X =
2
4
6
8
```

3.5.2 Matrices

A matrix is a rectangular array of numbers. Row and column vectors, are examples of matrices but with only one dimension. Consider for example the matrix C which is a two dimensional matrix 2 X 3 matrix. You can create this matrix by entering the following command in the command window.

```
>> C = [ 1 2 3 ; 4, 5, 6];
```

Note that the matrix elements in any row are separated by commas or spaces, and the rows are separated by semicolons. You can select an element from a matrix by using brackets.

```
>> C(2,2)
ans =
5
```

If you want the select a row, or a column from a matrix, you can use a colon. In the example below you select the third row of the matrix C.

```
>> C(2,:)
ans =
4 5 6
```

If you want to know the size of a matrix you can use the command size.

```
>> size(C)
ans =
2   3
```

3.6 Cell

A cell array is a collection of containers called cells in which you can store different types of data. This data type is very useful when working with strings. Cell arrays are created just like regular arrays except that curly brackets are used instead of square brackets.

```
>> A = {'example'}
```

The first cell of the cell array A contains now the string example. You can also create a cell array with multiple row and columns. Each cell of the cell array can contain a different data type. For example the cell array B is an 2 X 3 cell array with strings(char) on the first row of cells and numbers (doubles) on the second row of cells.

```
>>B = { ' Monday',' Tuesday',' Wednesday' ;
            15, 18, 20};
```

But you can also use matrices inside cell arrays. If you want the select one cell in the cell array, then you have to use the curly brackets.

```
>> B{1,2}
Ans = Tuesday
```

If you want to select a row or a column, you have to use normal brackets. Be aware that you select then a new cell array, which contains a row or a column of cells.

```
>>  B(:,2)
Ans = 'Tuesday'
[18]
```

Cells are particularly useful for arrays that combine strings and numbers or if you want to store multiple strings.

3.7 Structure

A structure (*struct*) is a data type that groups related data using data containers called fields. Each field can contain data of any type or size. Similar to cell arrays, structures can contain multiple data types. For example we can create the structure student which has the fields: name, theory and exercise. Afterwards you can browse through the fields of the structure Student by typing the name of the structure followed by a point and then press the tab button.

```
>> Student.exercise =15
>>Student.theory=16
>>Student.name= ' Sofie Jacob'
```

The structure as a whole consists of several records, each of which has an optional index. For example we can store multiple students inside the structure Student by using the index.

```
>> Student(2).exercise =14
```

When you want to know all the fieldnames of a structure you can use the command fieldnames(). For example fieldnames(Student) will give us an cell array with the three string (exercise, theory and name). Both the cell array and the structure are more advanced data types that require some exercises to understand them.

4 Basic operations

4.1 Basic operations with numbers

Just like a calculator, matlab is able to execute mathematical operations.

Table 4.1.1: List of basic operations with numbers

+	Addition
-	Subtraction
*	Multiplication
/	Division
∧	Power
Sqrt()	Square root

For example, if you want to add one by one.

```
>> 1+1
ans= 2
```

Now Matlab has computed 1+1 and assigned the result the variable ”ans”. You can also assign the result to another variable than 'ans'. For example:

```
>>  boys =4;
>> girls=10;
>> Students=  boys+girls
```

Now we have assigned the value four to the variable boys and the value 10 to the variable girls. As you can see we have added a semicolon after assigning a value to a variable. This prevents that matlab writes the output from this command to the screen. The '=' sign has another meaning inside matlab than the mathematical meaning. Inside matlab, equal to (=), means set equal to or shorter becomes. For example with x=1; you are telling matlab that the variable x becomes one. This means that there should be always a variable on the left side of the '=' sign.

```
>> a=1+1;    % this will work
>> 2=a;      % this won't work because there is no variable on the ...
    left side of the equation sign
```

Furthermore, this enables us to do the following operations:

```
>> a=2;
>> a=a+1;
```

The second expression is mathematically completely wrong, but is frequently used in multiple programming languages. Please try this with the other basic operations (/,-,*,∧,sqrt()).

4.2 Basic operations with vectors & matrices

In order to be able to do some basic operation with vectors and matrices, it's important to repeat some mathematical definitions for operations with matrices. In the following explanations i stand for the ith row of the matrix and j for the jth column of the matrix.

- Addition of matrices:
  - $C = A + B \rightarrow C_{ij} = a_{ij} + b_{ij}$
  - A and B should have the same dimension and C has the same dimension as A and B.
- Multiply a matrix with a scalar
  - $C = A * k \rightarrow C_{ij} = k * a_{ij}$ with k representing a real number
  - A and C have the same dimension.
- The multiplication of matrices is possible in two ways.
  - Element-wise operations: the dot product, also called scalar product or inner product.
    - $C = A. * B \rightarrow C_{ij} = aijb_{ij}$ note the dot (.) before the *.
    - A, B and C have the same dimension
  - The matrix multiplication, also called the outer product.
    - $C = A * B \rightarrow C_{ij} = \sum_(k = 1)^n a_{ik}b_{kj}$
    - With the dimension of A= (m x n), B=(n x p) and C = (m x p)
  - The Division of matrices is possible in two ways

- $*$ Element-wise operation: Based on the elements of the matrix.
  - $\cdot$ $C = A./B \rightarrow c_{ij} = a_{ij}/b_{ij}$
  - $\cdot$ A, B and C have the same dimension
- $*$ Matrix operation: Based on the inverse of the matrix
  - $\cdot$ $C = A/B$ this means C = A * inv(B)
  - $\cdot$ $D = A\ B$ this mean D= inv(A) * B

The example below explains the difference between the operations with a dot and without a dot.

You have worked for 4 weeks, the vector 'hourly_wage' contains your hourly wage for each week. The vector "hours" contains the hours that you have worked during that week.

```
>> hourly_wage = [ 10 11 12 13];
>> hours = [ 30 38 36 41];
>> weekly_wage=hourly\_wage .* hours;   % this works
>> weekly_wage=hourly\_wage *  hours;   % this won't work.
```

Another basic operation with matrices is transposing a matrix. This means that you write the rows of the matrix as the columns of the matrix and the columns of the matrix as the rows. Transposing a matrix is done in matlab with ' . For example:

```
>> A=[1 2 3; 4 5 6];
>>B=A';
    B =      1   4
        2   5
        3   6
```

4.2.1 Matrix multiplication (without dot)

Formula 4.2.1 is used to multiply matrix a and b.

$$c_{ij} = \sum_{k=1}^{n} a_{ik} * b_{kj} \tag{4.2.1}$$

The multiplication of matrices has the following requirements for the dimensions of the matrix.

$$(m \times n) * (n \times p) = (m \times p)$$

$$(2 \times 3) * (3 \times 2) = (2 \times 2)$$

For example the multiplication of 2 matrices will give this result.

$$\begin{pmatrix} 1 & 6 & 4 \\ 7 & 2 & 8 \end{pmatrix} \times \begin{pmatrix} 2 & 6 \\ 5 & 3 \\ 6 & 9 \end{pmatrix} = \begin{pmatrix} 56 & 60 \\ 72 & 120 \end{pmatrix}$$

OR

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{pmatrix} =$$
$$\begin{pmatrix} a_{11} * b_{11} + a_{12} * b_{21} + a_{13} * b_{31} & a_{11} * b_{12} + a_{12} * b_{22} + a_{13} * b_{32} \\ a_{21} * b_{11} + a_{22} * b_{21} + a_{23} * b_{31} & a_{21} * b_{12} + a_{22} * b_{22} + a_{23} * b_{32} \end{pmatrix}$$

The matlab command for this multiplication is

```
>> A = [1 6 4; 7 2 8];
>> B = [2 6; 5 3; 6 9];
>> A * B
ans =
56 60
72 120
```

This means that the number of columns of the first matrix in the multiplication should equal the number of rows in the second matrix. For example the operation B *A will not work. Matlab will give the following error which means that the number of columns of the first matrix is not equal to the number of rows of the second matrix.

```
??? Error using ==> mtimes
Inner matrix dimensions must agree.
```

4.2.2 element-wise matrix multiplication (with dot)

Formula 4.2.2 is used to do element-wise multiplication of two matrices in matlab.

$$c_{ij} = a_{ij} * b_{ij} \tag{4.2.2}$$

This mean that the size the both matrices should be the same.

$$(m \times n) * (m \times n) = (m \times n)$$

$$(2 \times 3) * (2 \times 3) = (2 \times 3)$$

For example the multiplication of 2 matrices will give this result.

$$\begin{pmatrix}1 & 2 & 3\\4 & 5 & 6\end{pmatrix} \times \begin{pmatrix}2 & 3 & 4\\5 & 6 & 7\end{pmatrix} = \begin{pmatrix}2 & 6 & 12\\20 & 30 & 42\end{pmatrix}$$

OR

$$\begin{pmatrix}a_{11} & a_{12} & a_{13}\\a_{21} & a_{22} & a_{23}\end{pmatrix} \times \begin{pmatrix}b_{11} & b_{12} & b_{13}\\b_{21} & b_{22} & b_{23}\end{pmatrix} = \begin{pmatrix}a_{11}*b_{11} & a_{12}*b_{12} & a_{13}*b_{13}\\a_{21}*b_{21} & a_{22}*b_{22} & a_{23}*b_{23}\end{pmatrix}$$

The matlab command for this multiplication is

```
>> A = [1 2 3;  4 5 6];
>> B = [2 3 4 ;  5 6 7];
>> A .* B
ans =
2 6 12
20 30 42
```

4.2.3 other operations

Matrix Operations	
$*$	matrix muliplication ($c_{ij} = \sum_{k=1}^{n} a_{ik} * b_{kj}$)
$/$	matrix division ($c = a * inv(b)$)
$\backslash$	matrix division ($c = inv(a) * b$)
$\wedge$	exponentiation
$sqrtm$	square root

Element-wise operations	
$.*$	multiplication ($c_{ij} = a_{ij} * b_{ij}$)
$./$	division ($c_{ij} = a_{ij}/b_{ij}$)
$.\backslash$	division ($c_{ij} = b_{ij}/c_{ij}$)
$.\wedge$	exponentiation
$sqrt$	square root

4.3 Basic functions

Matlab has a large library of simple and complex functions that are available for use. Several basic functions will be explained in this part. If you want to know more about these functions, feel free to search more information in the Help function. When you search more information about the function size, the matlab help function will give you the following explanation.

```
SIZE   Size of array.
    D = SIZE(X), for M—by—N matrix X, returns the two—element row vector
    D = [M,N] containing the number of rows and columns in the matrix.
    For N—D arrays, SIZE(X) returns a 1—by—N vector of dimension lengths.
    Trailing singleton dimensions are ignored.

    [M,N] = SIZE(X) for matrix X, returns the number of rows and ...
       columns in
    X as separate output variables.
    ......
```

This means that the function size will give you more information about the size of the input variable. For example when we create the following matrix A, size(A) will give the number of rows and numbers of columns as output.

```
    >>A=[2  4 6; 7 8 9];
    >>[X,Y]=size(A)
    X = 2       Y=3
```

This function is useful to repeat the matrix multiplication. The code below creates the matrix C and D. Calculate with and without the function size the size of the matrix C and D.

```
    >> A=[1 2 3];
    >> B= [1;2;3];
    >> C=A*B;
    >> D=B*A;
```

The following table explains some more basic functions. Read the help page of the functions when you want to use them.

Table 4.3.1: Basic matlab functions

Function	**Short explanation**
Mean	Calculates the mean value
Min	Calculates the minimal value
Max	Calculates the maximal value
Median	Calculates the median
Sort	Sort in ascending or descending order
Std	Calculates the standard deviation
......	....

5 Data Import and Export

5.1 Matlab files

Workspace variables can be exported with a simple command ’save(’filename’, ’variables’)’ The ’filename’ is the name of the file, without extension. The output is automatically a .mat-file. If you want to save it in a folder, you need to add the folder name in front of the filename, separated with a slash. The ’variables’ is a comma separated list with the names of the variables that need to be exported. All these variables will be exported in the same mat-file, but when importing this file, the variables will be separated again. When no list of variables is given, all the variables of the workspace will be saved. The following example shows how a variable x and sin(x) were saved together in a file named ’savedData’ in the folder ’Data’:

```
x = 0:.1:2*pi;
sin_x = sin(x);
cos_x = cos(x);

save('Data/Workspace')
save('Data/savedData', 'x', 'sin_x')
```

It is possible to export variables of all kinds into the same mat-file. Following example exports a structured array, a string and a matrix into the same mat-file.

```
matrix = rand(3,4);
structure.x = x;
structure.sin = sin_x;
string = 'Import and Export';
save('Data/Combination', 'matrix', 'structure', 'string')
```

With the same ease as exporting, we can also import the data. Therefore, we use the command ’load(’filename’, ’variables’)’. The ’filename’ is again the name of the file which needs to be imported, with or without extension. The ’variables’ define which variables needs to be imported. When this last input argument is not specified, all variables in the mat-file were imported. Example:

```
load('Data/savedData')
load('Data/Workspace', 'cos_x')
load('Data/Combination', 'string', 'structure')
```

A second option is to load all variables into one structured array, by assigning the 'load' command to a variable, e.g. 'S = load('filename', 'variables')', The following example shows how the variables that are saved in 'Combination.mat' are combined into one structured array:

```
S = load('Data/Combination');
```

5.2 Text files & ASCII data files

Often programs store their data in text files or ASCII files. The columns in the data are separated by use of delimiters, like for example commas, spaces or tabs. A **first option** to import this kind of files is using the 'importdata('filename')' command. This will convert the data into to a structured array, with a first field 'data' containing the numeric data, a second field 'textdata' containing the textual information in form of a cell array and a third field 'colheaders' containing a cell array with a guess of the column headers as specified in the file. The following example shows the import of measured ground reaction forces, saved in an ASCII delimitated file:

```
A = importdata('Data/GRF_example.mot');
```

The function 'importdata' automatically determines the best possible delimiter and the number of header lines in order to read the numeric information. By adding extra output arguments, the values of these variables can be returned as well, for example:

```
[A, delimiterOut,headerlinesOut] =  importdata('Data/GRF_example.mot');
```

'HeaderlinesOut' will give you the number of header lines there are in the file; 'delimiter-Out' will give you the type of delimiter that was used in the file. However, sometimes you know in advance which delimiter is used and how many headerlines were included in the file. You can give this additional information to the 'importdata' command. First the delimiter needs to be specified as a string (comma ',', semicolon (':'), tab '\t' or space (' ')). Secondly the number of headerlines can be specified. For the previous example with seven headerlines and a tab delimiter this becomes:

```
B = importdata('Data/GRF_example.mot', '\t', 7);
```

A **second option** is to import the data interactively. This can be done with the command 'A = uiimport('filename')'. A GUI (Graphical User Interface) dialog will be opened, which allows you to choose the delimiter and the number of headerlines and which immediately gives you a preview of the imported data. This gives you an opportunity to check the used delimiter and number of header lines. By clicking the 'next' button you can choose what variables need to be imported and how they will be saved, in a structured array or in separated vectors using the column headers as name. After clicking the 'Finish' button, the data will be imported into the workspace. It is also possible to generate a MATLAB code (function), that allows later use of the same inputs in other files as well. This will be done by ticking the box at the right side of the 'finish' button. An example of this interactive data import:

```
C = uiimport('Data/GRF_example.mot');
```

5.3 Excel files

Another important and often used data type is an excel file (extension .xls or .xlsx). The standard MATLAB command to import data from such a file is 'num = xlsread ('filename')'. This function reads data from the first worksheet in the Microsoft Excel spreadsheet file, named 'filename' and returns the numeric data in array 'num'. The following example shows how the first worksheet of an excel file called 'Static2foot - 7-25-2013.xls' is imported and stored in the folder 'Data':

```
NUM = xlsread('Data/Static2foot - 7-25-2013.xls');
```

Additional input arguments can be specified if you want to import a specific worksheet. Arguments that can be added are the name of the sheet (string) or the particular range of the excel sheet that needs to be imported. The latter one is indicated by the first element (top left) and the last element (bottom right) of the data matrix. Some examples of additional input arguments:

```
NUM = xlsread('Data/Static2foot - 7-25-2013.xls',   'M13:AD38');
NUM = xlsread('Data/Static2foot - 7-25-2013.xls',   'Static2Feet_2');
NUM = xlsread('Data/Static2foot - 7-25-2013.xls',   ...
    'Static2Feet_2', 'M13:AD38');
```

The first example imports only these data that are within the rectangular range from M13 till AD38 of the first worksheet. The second example imports all data of worksheet 2, named 'Static2Feet_2'. The third example combines the two input arguments and will import the data within range M13:AD38 of the second worksheet. It is also possible to import textual information from the excel sheet. Therefore a second output argument needs to be included, e.g. '[NUM, TXT] = xlsread('filename')', where TXT is a cell array containing text fields. Additionally, is it possible to import the unprocessed data (numbers and text) in a cell array, by adding a third output argument, e.g. '[NUM, TXT, RAW] = xlsread('filename')'. An empty cell in the excel file is replaced by NaN (Not A Number). The next example shows the use of these extra output arguments:

```
[NUM, TXT, RAW] = xlsread('Data/Static2foot2 − 7−25−    2013.xls', ...
    'M13:AD38');
```

The reverse operation, exporting data into an excel file, is very similar to importing the data. To export data to excel we use the command 'xlswrite('filename', data)'. The first input argument is the name of the file where the data need to be stored and the second argument is the data itself, which can be a matrix or a cell array. Via the latter it is possible to also export textual data. Examples of this export command:

```
matrix = [10,2,45;−32,478,50];
cell_array = {92.0,'Yes',45.9,'No'};
xlswrite('Data/ExampleXLS1.xls', matrix);
xlswrite('Data/ExampleXLS2.xls', cell_array);
```

Similar to the import of excel files, it is possible to specify the range and the worksheet to which the data need to be exported. This gives the possibility to export several data sets into one excel file. The next example shows this:

```
xlswrite('Data/ExampleXLS3.xls', matrix, 'WorkSheet1',  'C4:E5');
xlswrite('Data/ExampleXLS3.xls', cell_array,    'WorkSheet2', 'B2');
```

If no worksheet with the specified name exists, it is created. In case you do not specify a worksheet, the range must include both corners and a column character, even for a single cell (such as 'D2:D2'). Otherwise, 'xlswrite' interprets the input as a worksheet name (such as 'D2'). If you specify a worksheet, the range can be specified by only the first cell (such as 'D2'). 'xlswrite' writes the input array beginning at this cell, as shown in previous example.

5.4 Image files

Sometimes it is needed to import images to MATLAB in order to perform some calculations on it. Examples from practice are: digitizing a graph using an image, determining muscle paths via MRI images, division of a foot contour into different areas, and so on. An image can easily be imported using the command 'imread('filename')'. The image, named 'filename', will be loaded into MATLAB and converted into an M-by-N-by-3 array. For each pixel of the M-by-N image, a RGB code will be stored with values between 0 and 255. With other words, the image is converted into three separated images, one for each color (Red, Green and Blue). The next example shows the separation of an image into the different main colors.

```
C = imread('Figures/AdditiveColor.png');
figure; subplot(2,2,1); image(C); axis off; axis image

C_R = C; C_R(:,:,2:3) = 0;
C_G = C; C_G(:,:,[1,3]) = 0;
C_B = C; C_B(:,:,1:2) = 0;

subplot(2,2,2); image(C_R); axis off; axis image
subplot(2,2,3); image(C_G); axis off; axis image
subplot(2,2,4); image(C_B); axis off; axis image
```

In the first line the image will be loaded and in the second line the figure will be plotted. More information about plotting can be found in the next chapter. In the following three lines the images for the red, green and blue color are separated by assigning zero to the other colors. The last three lines contain the plot commands of the separate images. More examples of the 'imread' function can be found in the chapter 6. Equivalent to 'xlsread/xlswrite', there is an 'imwrite' command to export figures. As a first input an M-by-N-by-3 matrix is specified, containing the RGB data of an image. A second argument specifies the filename, while the third argument specifies the format (file extension). The code for exporting the images of our example with the separate colors can be found below. The separate images will be saved in different formats (png, jpg and tiff), to show you the use of the format specifier.

```
imwrite(C_R, 'Figures/AdditiveColorR.png', 'png')
imwrite(C_G, 'Figures/AdditiveColorG.jpg', 'jpg')
imwrite(C_B, 'Figures/AdditiveColorB.tiff', 'tiff')
```

6 Plotting

Scientific results and measures can easily be generated in MATLAB using mathematical operations . In order to report and examine these results, visualization would be very helpful. Not only visualization of the end-result (for use in presentations, papers and theses) but also graphs presenting intermediate steps are useful. The latter ones allow you for example to easily detect mistakes in your processing code. Therefore, it is necessary to know how to plot your results in any format you like and how to export the figures.

The most straightforward way to create a plot is by using the plot command: 'plot(x, y)', where x is a vector defining the independent values and y is the data you want to plot (dependent values). Vector x and vector y need to be of the same length. For example, if you want to plot sin(x) on the interval [0, 2π], you would type:

```
x=0:.1:2*pi;
y=sin(x);
plot(x,y);
```

In the case there is no figure window opened yet, this command will create a figure window, otherwise the plot will be drawn on the last created figure. If you want to create several figures in the same script, a new figure window can be opened using the command 'figure'. You can also plot several sets of data on the same figure but be aware that each vector is of the same length. One way to do this is by using the plot command with extra input arguments: 'plot(x1,y1,x2,y2)'. Another (perhaps better) way is to use the 'hold' command, which holds the plot on the current figure and will plot all subsequent plots on the current figure. The command 'hold off' ends this option and from here on, new plot actions will not appear with the previous ones anymore:

```
figure;
x1=0:.1:2*pi;
x2=pi:.1:3*pi;
y1=sin(x);
y2=cos(x);
plot(x1,y1)
hold on
plot(x2,y2)
hold off
```

When you don't use the hold command, only the last plot will be visible on the figure, the other figures will be erased. Try this yourself.

Only plotting the x and y data is not enough for a self-clarifying figure. Since the beginning, we've been taught to label our plots. MATLAB allows you to do this through the command line. For a label on the x- or y-axis you can use the commands 'xlabel' and 'ylabel', respectively (for a 3D plot you have also 'zlabel'), which has a string containing the label as input. In order to add a title to the plot, you can use the command 'title', with a string as input. Now, we can label and title our sine plot:

```
xlabel('x');
ylabel('sin(x)');
title('My First MATLAB Plot');
```

Sometimes you don't want to display the whole plot, but instead you want to zoom in only within some range. This range can then be characterized by limits on the x- and y-values (and also z-values when plotting in 3D). MATLAB allows you to specify the limits and parts of the plot that lie outside this range will be made invisible. This is done using the command 'xlim' and 'ylim' (and 'zlim'), with as input a 2-element vector containing the lower and upper limit in that direction respectively. For example if we want to limit our plot in the x-direction between π and 2π, we need to add:

```
xlim([pi 2*pi]);
```

So you made your first plot. Despite this plot is very standard, it is a good way to visualize intermediate results in order to check the data processing you are performing for mistakes. However, this basic figure is not really suitable for presenting your end-results properly. In the next section we will therefore go into more detail on how we can modify our figures in order to optimize the layout of the plots. Subsequently, a next section explains the different plot types which are implemented in MATLAB. Next, more information is given on how to handle the figure in an interactive way. The last part of this chapter explains more advanced plot options in more detail.

6.1 Plot options

A more general plot command is plot(x, y, 'options'), where options can be one of all different functionalities which can be added to the plot. For a list of all plot options, search in the help tool ('plot'). We will mention here some useful and most used options and their application.

6.1.1 Color

In order to distinguish between different datasets in a figure with multiple datasets, we could give every plot a different color. This can be done by adding a string specifier to the plot command, which indicates the color. There are seven preprogrammed colors in MATLAB, every color is associated with their own code: red ('r'), blue ('b'), magenta ('m'), yellow ('y'), green ('g'), cyan ('c') and black ('k'). For example for our plot becomes this:

```
figure;
plot(x1,y1, 'r')
hold on
plot(x2,y2, 'g')
```

A second method to add color to the plots is by using the 'Color' option specifier, followed by a 3-elements vector, containing the RGB-values of the color you want. The RGB-values determine the relative amount of red(R), green(G) and blue(B) the color contains. With online RGB-generators it is easy to generate the RGB code of the color of your choice. Via this method it is also possible to choose different colors of gray, which is needed when you are e.g. planning to published your results in black and white. In the example two different gray colors are used. Because MATLAB can only work with color values between 0 and 1 we need to normalize the vector by dividing it by 255.

```
figure;
plot(x1,y1, 'Color', [126,126,130]/255)
hold on
plot(x2,y2, 'Color', [193,193,195]/255)
```

6.1.2 Line & Marker style

Another way to distinguish different plots on the same figure is by using different line styles. The default line style is a solid line. In order to change this style, you need to add a line style specifier before the color definition, e.g. for a dashed red line you add '–r'. The most used line styles are the solid line ('-'), the dashed line ('–') and the dotted line (':'). The following example shows how to use the line style option:

```
figure;
plot(x1,y1, '--')
hold on
plot(x2,y2, ':')
```

Instead of plotting a line, connecting all the different data points, it is also possible to plot only the data points, by using a certain marker symbol. This will be specified by replacing the line style specifier by a marker specifier. Most used marker style are the circle ('o'), the asterisk ('*'), the square ('s'), diamond ('d') and a cross ('x'). It is also possible to combine a marker and a line style, as is illustrated in the next example.

```
figure;
plot(x1,y1, '--rs')
hold on
plot(x2,y2, ':gd')
```

Besides changing the color, the line and the marker style, it is also possible to change the visual characteristics of the line and markers. Examples are changing the width of the line, the size of the markers, the color of the markers, etc. These changes are implemented by adding an option specifier and an option value to the plot command, e.g. plot(x, y, 'r', 'Options Specifier', 'Option value'). For example for changing the width of the line the options specifier is 'LineWidth', and the option value specify the width of the line (specified in points).

```
figure;
plot(x1,y1, 'LineWidth', 2)
hold on
plot(x2,y2, 'LineWidth', 4)
```

The process to change the marker color or marker size, is very similar as above but with using different option specifiers ('MarkerEdgeColor', 'MarkerFaceColor', 'MarkerSize'). The next example shows how this is done:

```
x = -pi:pi/10:pi;
y = tan(sin(x)) - sin(tan(x));
figure;
plot(x,y,'--rs','LineWidth',2,  'MarkerEdgeColor','k', ...
    'MarkerFaceColor','g',    'MarkerSize',10)
```

For more information and a complete list of option arguments, you can search in the help to 'LineSpec'.

6.1.3 Legend

In a figure containing multiple plots, it is not only necessary to make a distinction between the different plots, but also to name the plots in a legend. This can be done in MATLAB, using the command 'legend', with the names of the plot as argument in a cell array with strings. Again it is possible to add some options to this command, in the same way as with plot by using an option specifier and an option value. A commonly used option is the specification of the location of the legend, which is indicated by option specifier 'Location', followed by a compass direction: North, South, East, West, NorthEast, NorthWest, SouthEast and SouthWest. If it is needed that the legend should be outside the plot, you can specify this by adding 'Outside', after the compass direction, e.g. NorthWestOutside. It is also possible to let MATLAB determine the best location for the legend, those with the least overlap with the curves. This will be obtained with the option value 'Best' or 'BestOutside'. Run the next example for an illustration of the legend command.

```
figure
x = -pi:pi/20:pi;
plot(x,cos(x),'-ro',x,sin(x),'-.b');
legend({'cos(x)','sin(x)'});

legend({'cos(x)','sin(x)'},'Location', 'SouthWest');

legend({'cos(x)','sin(x)'},'Location', 'Best');
```

For more information about legend, search for 'legend' in the HELP files.

6.1.4 Font options

A figure contains a lot of text, such as the title, the axes label and the legend names and it can be desirable to change the font of this text. This is done by adding again an option specifier, followed by an option value, to the commands of label, title or legend. Examples of text properties are 'FontSize' (in points), 'FontWeight' (normal, bold, light, demi), 'FontName' (name of family, default is Helvetica) and 'FontAngle' (normal, italic, oblique). For example:

```
figure
x = -pi:pi/20:pi;
plot(x,cos(x),'-ro',x,sin(x),'-.b');
legend({'cos(x)','sin(x)'},'Location', 'Best',  'FontSize', 16);
xlabel('x values', 'FontAngle', 'italic',   'FontWeight', 'bold');
ylabel('y values', 'FontAngle', 'oblique',  'FontWeight', 'demi');
title('Cosinus and sinus', 'FontName', 'Courier',   'FontSize', 20)
```

MATLAB also allows you to use LaTeX commands to display equations or symbols very easily, however it require some knowledge of LaTeX. LaTeX is a document preparation system for high-quality typesetting. It is most often used for medium-to-large technical or scientific documents but it can be used for almost any form of publishing. It is characterized by its own programming language (LaTeX), with several commands for e.g. symbols, equations, subscript, superscript, et cetera. Via a LaTeX code it is easy to include a subscript or superscript in your text, by adding _{text in subscript} or ∧{text in superscript} to the string of the text. For example:

```
ylabel('y_{values}');
xlabel('values^{x}');
```

It is also possible to include greek or mathematical symbol via this LaTeX commands. A LaTex command starts with a backslash and arguments are defined within accolades. For de whole list of mathematical symbols, I want to refer you to the Wikibooks site of LaTeX. An example of using LaTeX symbols:

```
figure
alpha = -pi:pi/20:pi;
plot(alpha,cos(alpha));
xlabel('\alpha', 'FontSize', 16);
ylabel('\int{sin(\alpha)}', 'FontSize', 16);
```

6.2 Other plot types

Besides the basic plots where a Y-value will be plotted against an X-value, there is a whole range of different plots implemented in MATLAB. In this section, several commonly used plots will be highlighted in detail. We cannot explain all the different plot sorts, due to the high variability, but remember that you can always find more information in the HELP files. The plot types that will be discussed are the scatter plots, boxplots, bar plots, image plots and 3D plots.

6.2.1 Scatter plot

A first important and often used plot is a scatter plot. A dataset of x- and y-values will be plot against each other, by means of colored markers. In MATLAB, this will be performed with the command scatter(x, y). The inputs x and y are vectors with the same length. An example of the code for a scatter plot can be found here, where the x- and y-values were randomly generated:

```
x = rand(50,1);
y = rand(50,1);
figure;
scatter(x, y)
```

The default marker layout is colored circles with a default size. However the layout can again be changed by adding options to the standard command. As a third argument you can supply the size of the markers, by give the area (specified in point^2). For example we can define the size of the markers as a function of the y-value. In next example the size increases with increasing y-value.

```
figure;
scatter(x, y, 10*(10*y+1))
```

It is also possible to plot filled markers, using the argument 'filled'. The option specifiers related to the marker and line style, as discussed in previous sections, can still be used to modify the layout of the markers. In next example a red square marker is chosen, with a thick black colored edge.

```
figure;
scatter(x, y, 10*(10*y+1), 'sr', 'filled',  'MarkerEdgeColor', 'k', ...
    'LineWidth', 2)
```

6.2.2 Box plot

With the MATLAB command 'boxplot(X)', it is fairly simple to get a boxplot. A matrix X is given as input: all the columns of this matrix represent a data vector for one boxplot. If you give just one vector, only one boxplot will be plotted. On each box, the central mark is the median, the edges of the box are the 25th and 75th percentiles, the whiskers extend to the most extreme data points not considered outliers, and outliers are plotted individually. Here an example code is given for 4 boxplots, with each data vector containing 30 randomly generated samples:

```
X = rand(30,4);
figure;
boxplot(X)
```

Using option specifiers you can change the layout of the boxplot (boxplot(X, 'Name', value)). An example of such an option specifier is 'medianstyle', which lets you choose between drawing a line ('line') or a black dot inside a white circle ('target'). Another possible option specifier is 'orientation', which can be used to change the orientation of the boxplot to 'horizontal' or 'vertical' (default). A last interesting and useful option is to change the label of the boxplots. Instead of the standard enumeration of the boxplots, you can specify a specific name, using the option specifier 'labels', followed by a cell array containing the strings for each boxplots. This example shows the use of those option specifiers:

```
figure;
boxplot(X, 'medianstyle', 'target', 'orientation',  'horizontal', ...
    'labels', {'Box 1', 'Box 2', 'Box 3',      'Box 4'})
```

There are many other option specifiers and the function and use of these specifiers is clearly documented in the HELP on boxplots.

6.2.3 Bar plot

Another visual way to present your data is by using a bar plot. The data will be shown in a bar, with the height of the bar indicating the value. A bar plot will be made using the MATLAB command bar(X), where X is a matrix where each row represent one group with the values in the columns. Without any other specifications, a grouped bar plot will be made, as for example:

```
X = rand(5,3);
figure;
bar(X)
```

which will make a bar plot with five groups, with three bars in each group. It is also possible to go to a stacked plot, where the three bars of each group are stacked together. This will be specified using the option specifier 'BarLayout', followed by 'stacked' (instead of 'grouped'). See the following code:

```
figure;
bar(X, 'BarLayout', 'stacked')
```

Again the layout of the bar plot can be changed using several option specifiers, as for example 'BarWidth' which determines the relative width of the bars. Also some option specifiers as discussed in the section on the general plot options, like 'LineStyle' and 'LineWidth', can be used in this MATLAB command. The next example shows the use of these option specifiers.

```
figure;
bar(X, 'BarWidth', 0.5, 'BarLayout', 'stacked',     'LineStyle', :, ...
    'LineWidth', 2)
```

It is also possible to plot the bars in horizontal direction with the barh(X) command, which has the same option specifiers as for the normal bar(X) command.

```
figure;
barh(X, 'BarWidth', 0.5, 'BarLayout', 'stacked',     'LineStyle', :, ...
    'LineWidth', 2)
```

6.2.4 Image

As discussed in previous chapter, an image can be loaded into MATLAB and will be converted into an M-by-N-by-3 array. For each pixel of the M-by-N image, a RGB code will be stored with values between 0 and 255. With this data it is very simple to plot the image, by using the MATLAB command image(C), where C is the M-by-N-by-3 array of the image. For example:

```
C = imread('Figures/Foot.jpg');
figure;
image(C)
axis off
axis image
```

The last two commands set the visibility of the axes on off and to adapt the size of the axes to those of the image.

6.2.5 3D plot

Previous plot commands are only applicable on 2D data. However it would be possible that we need to work with data in 3D, for example plantar pressure data (x, y, P). With this extra dimension to plot, we need to use slightly different plot commands as the previous one, by just adding a '3' after the command. So the 3D version of plot(x, y) becomes plot3(x, y, z), for scatter(x, y) this is scatter3(x, y, z) and even for a bar plot there is a 3D version, named bar3. The different options for the particular plot command remain the same, only an extra input for the extra third dimension is added. An example of a plot in 3D:

```
alpha = -4*pi:pi/20:4*pi;
x = sin(alpha);
y = cos(alpha);
z = alpha/(2*pi);
figure;
plot3(x, y, z)
```

Despite the large similarities with 2D plots, it is important to look into the viewing aspect of 3D plots. The screen of your laptop or computer is a 2D object, and therefore it can only display 2D image of our 3D plot. Depending on which direction you look to the 3D plot, a different 2D image will be shown. The view direction can be changed with the 'view' command, added after the plot command. The 'view(az, el)' command sets the viewing angle for a 3D plot. The azimuth (az) is the horizontal rotation about the z-axis as measured in degrees from the negative y-axis. Positive values indicate counterclockwise rotation of the viewpoint. The vertical elevation of the viewpoint in degrees is given by el. Positive values of elevation correspond to moving the viewpoint above the object; negative values correspond to moving below the object. Figure 6.2.1 shows an visualization of the azimuth and elevation angle. The default 3D view has and azimuth of -37.5° and an elevation of 30°. You should however be careful when defining the view: For example if we change the view of previous example to an elevation of 90°, we only get a 2D plot, which showed a circle:

```
figure;
plot3(x, y, z)
view(0, 90)
```

The same will happen with an elevation of 0°, so be careful with interpreting the results of a 3D plot.

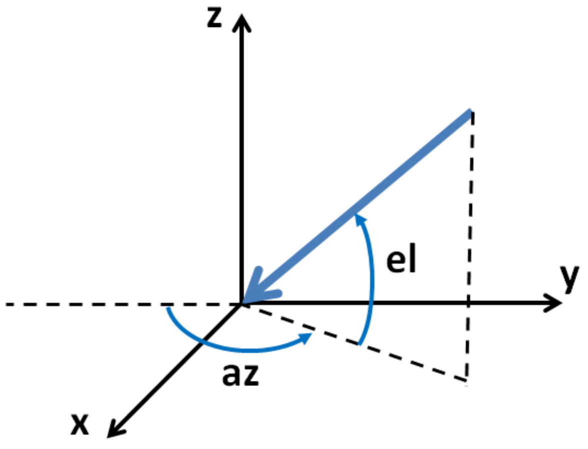


Figure 6.2.1: Visualization of the view direction, determined by the value for the azimuth and elevation angle.

Besides the modified 2D plot functions, there are some plot commands that are specific for 3D application. Two of those commands are the function 'surf(x, y, Z)', which plots a three-dimensional shaded surface and the function 'mesh(x, y, Z)', which plots a wireframe mesh. Both commands have the same input, which is a vector x with length N, a vector y with length M and Z a M-by-N matrix. A practical example is the plantar pressure measured with a pressure plate, were x and y are the position of the pressure sensors to the ground, and Z a matrix is with all the pressure values. In the following example, a Z-matrix will be constructed as a function from x- and y-values, uniformly distributed over the range from [-8, 8]. The 'meshgrid' function transforms the domain specified by two vectors, x and y, into matrices X and Y. These matrices are then used to evaluate functions of two variables: The rows of X are copies of the vector x and the columns of Y are copies of the vector y.

```
x = −8:.5:8;
y = −8:.5:8;
[X,Y] = meshgrid(x, y);
R = sqrt(X.^2 + Y.^2) + eps;
Z = sin(R)./R;
figure
mesh(x,y,Z)

figure
surf(x,y,Z)
```

A variant of the 'surf' and 'mesh' function is the 'surfc' and 'meshc' function, which will be used to plot also a contourplot under the 3D shaded surface or wireframe mesh, respectively.

```
figure
meshc(x,y,Z)

figure
surfc(x,y,Z)
```

The colors of a surface or mesh plot are by default determined by the values of Z. However, it is also possible to choose the color of the surface by adding an extra argument C, which is a matrix with the same size as Z. The values of the C-matrix will be normalized and will define a color that is chosen from a color map. The following example creates a surface plot that is partially colored maximally and partially colored as before:

```
C = Z;
C(1:20, 1:20) = 1;
figure
surf(x,y,Z, C)
```

The colors are chosen from a colormap. Different colormaps are implemented in MATLAB and you can change the colormap by adding the command 'colormap(map)', where 'map' is a string which is a code for a certain colormap. The whole list of colormaps can be found in the help files. The default color map is 'jet'. In the example, we change the colormap to 'hsv' and see that the colors of the plot are really different:

```
figure
surf(x,y,Z)
colormap('hsv')
```

The choice of a colormap can also be very helpful for displaying images, for example for a X-ray image of a human spine, you can use the colormap 'bone', as shown in next example:

```
load spine
image(X)
colormap('bone')
```

6.2.6 Plot combinations

Now we have discussed several different plot functions separately, but nothing prevents you from combining these different plot sorts into one figure. Remember to use 'hold on' between the different plot commands, such that the previous plots will not be deleted. The following code gives an example of a bar plot combined with a normal plot and a scatter plot:

```
figure;
bar(X, 'BarWidth', 0.5, 'LineStyle', :, 'LineWidth',    2);
hold on; plot(X, 'r', 'LineWidth', 3)
hold on; scatter([1:5], X, 100, '^g', 'filled')
```

6.3 Interactive handling

Once a figure is created, you have the ability to inspect the figure interactively as well as to change the figure via the GUI (Graphical user interface). The different tools, allowing the handling of the figure, can be found in the Menu and Tool bar, as shown in figure 6.3.1.

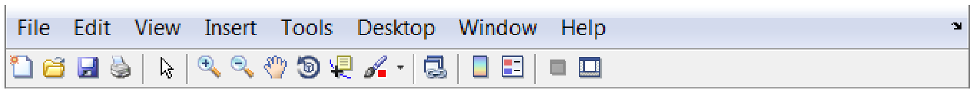


Figure 6.3.1: Menu bar and tool bar of the figure window.

The four first symbols are used to (1) create a new figure, (2) open a saved MATLAB figure (with extension .fig), (3) save the current figure (in different possible extensions like .png, .jpeg and .fig) and (4) print the figure directly. The pointer symbol (fifth symbol), will allow you to select entities of the plot. By double clicking on those entities (e.g. axis label or title) you can edit the text or numbers of it. It is however encouraged to make the edits directly using MATLAB code in the m-file. This is a much faster and more proper way of working. The two magnifier symbols are used to zoom in or zoom out in the plot. Click on the symbol and click on the place where you want to zoom in or out. It is also possible to zoom in within a particular area. Therefore you need to click and drag the mouse while holding the mouse button. A rectangle will be shown and when releasing the mouse button, the rectangular area will be zoomed in. The symbol with the hand is used to pan the view. Click on this symbol and then hold the left mouse button and drag the mouse and the plot will be moved along. The next symbol, the curved arrow, is used to rotate the view. This is particularly useful when inspecting a 3D-plot. The view will be rotated when holding the left mouse button and dragging the mouse. Next button is the data cursor. This tool gives you the ability to determine the value a point of choice in the plot, by just clicking on the point. Then a box will appear which gives the x- and y-value of this point (and z-value in 3D plots).

An equivalent code version of this feature is also implemented in MATLAB, with the command 'ginput(n)', where the value n determine how many points you need to take.

Once this command is running, a large cursor is displayed on the last opened figure and you can click on the points you wanted to know the x- and y-values of. The values, in a format of a matrix, will then be saved to the variable ans or you can assign this matrix to a variable. Example of such a code:

```
figure
alpha = −pi:pi/20:pi;
plot(alpha,cos(alpha));
xy = ginput(5);
hold on; scatter(xy(:,1),xy(:,2), 30, 'rs', 'filled');
```

In the previous example 5 points are selected, and once their x- and y-values were determined, a scatter plot is used to plot these points. By choosing a point, you are not limited to points on the curve but you may choose every point on the figure. One possible way to save or export a figure is by using the interactive method, as described above. Another way is by using a line of code, with the command 'saveas(gcf,'filename.ext')' or 'saveas(gcf,'filename','format'). 'gcf' stands for 'get current figure', meaning that the last opened figure will be saved. The second argument is the filename. If you want to save it in a folder, you need to add the folder name in front of the filename, separated with a slash. The format or extension specifies how the plot will be exported. Two most used formats are .png or .jpg, and it is also possible to save it as a MATLAB figure (.fig). The latter format is very useful when opening the file in MATLAB again, because you don't lose the ability to handle the plot (rotate, zoom in or out, pan view). This export command saves you a lot of time, when it is needed to export a lot of figures at once. An example of such a code can be found here:

```
figure
surf(x,y,Z)
colormap('hsv')
saveas(gcf, 'Figures/SurfacePlot', 'fig')
saveas(gcf, 'Figures/SurfacePlot', 'png')
```

6.4 Advanced plot options

6.4.1 Subplots

So far we have made several plots (graphs) on the same axes; however it is also possible to have multiple axes on the same figure. For example if you want to display several joint angles of different joints (each with its own range) together in one figure, it is mostly done like as shown in Figure 3. The figure is then divided as a sort of matrix, with a number of columns and a number of rows. For figure 6.4.1 this comprises two columns and three rows.

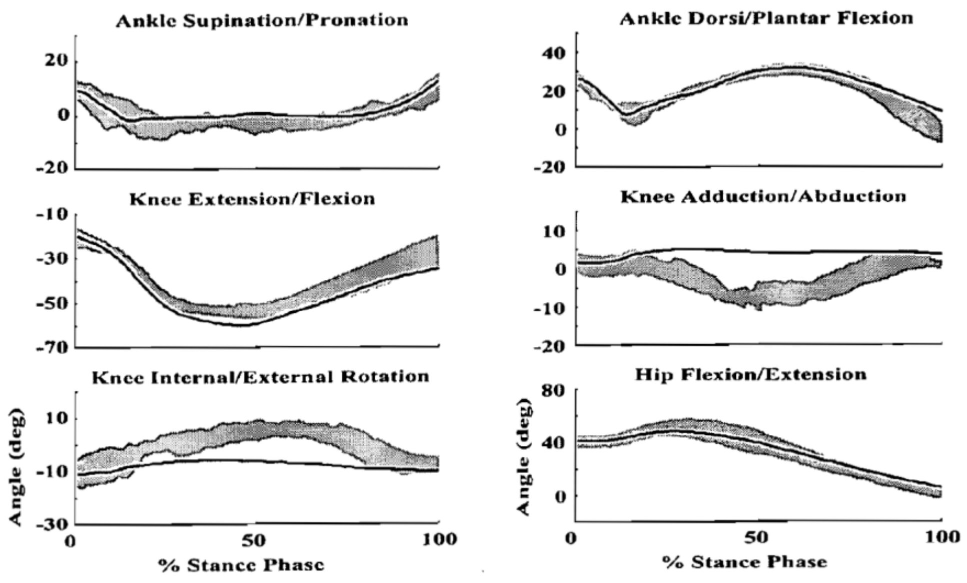


Figure 6.4.1: Example of multiple axes (subplots) on one figure from the literature. Figure taken from reference (Neptune, I.C., & Van den Bogert, 2000)

With the MATLAB command subplot it is very easy to let the figure divide automatically in multiple axes, and for each axes you can define the plot, the axes label and the title. The subplot command requires 3 arguments: (1) the number of rows, (2) the number of columns, (3) the number of subplots. The plots are numbered from left to right, top to bottom. Figure 6.4.2 shows the output of the following code, where the title indicates the code to use the plot. After the subplot command, you can use all the plot related commands as discussed above.

```
alpha = -pi:pi/20:pi;
figure;
subplot(3,2,1);
plot(alpha, sin(alpha))
title('subplot(3,2,1)')
ylabel('sin(\alpha)')

subplot(3,2,2);
plot(alpha, 10*sin(alpha))
title('subplot(3,2,2)')
ylabel('10*sin(\alpha)')

subplot(3,2,3);
plot(alpha, 0.1*sin(alpha))
title('subplot(3,2,3)')
ylabel('0.1*sin(\alpha)')

subplot(3,2,4);
plot(alpha, cos(alpha))
title('subplot(3,2,4)')
ylabel('cos(\alpha)')
```

```
subplot(3,2,5);
plot(alpha, 10*cos(alpha))
title('subplot(3,2,5)')
ylabel('10*cos(\alpha)')
xlabel('\alpha')

subplot(3,2,6);
plot(alpha, 0.1*cos(alpha))
title('subplot(3,2,6)')
xlabel('\alpha')
```

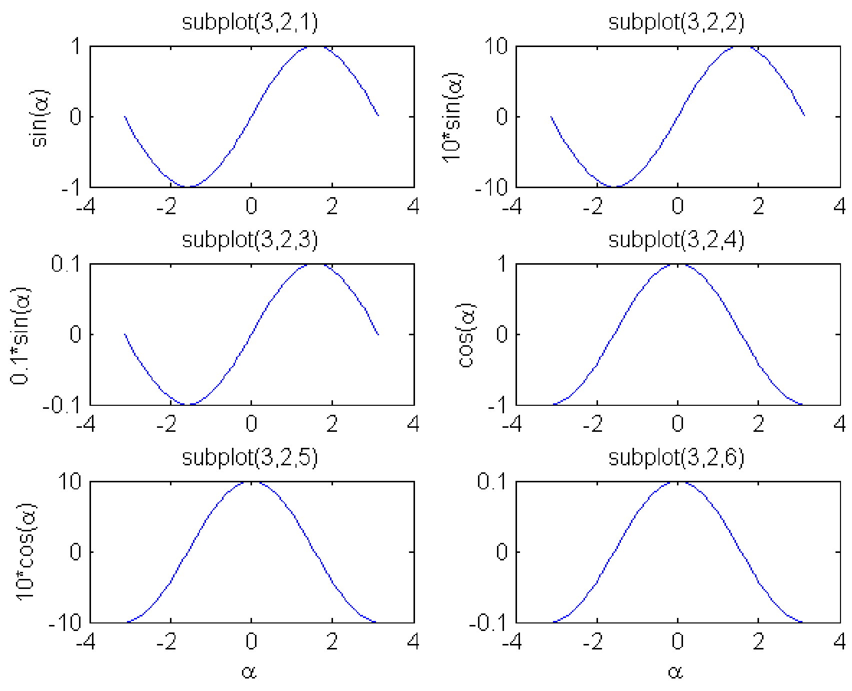


Figure 6.4.2: Example of multiple axes (subplots) on one figure from MATLAB code

7 Functions

MATLAB has a whole series of built-in functions, of which we have explored several in the previous chapters. For example, there are several functions to perform simple operations like finding the minimum and maximum, calculation of the mean, import and export of data, plot commands etc. However, this list of functions is often not enough to meet our requirements. Therefore, it would be useful to implement your own custom-made functions.

A second reason why building your own functions is recommended, is the reduction of code replication, which leads to a decrease of the number of lines in the m-file. For example, if you made your own implementation of a special boxplot, and you want to use the same boxplot to represent the results generated from three different m-files. A possible way to generate the boxplots is by copying your boxplot code and paste it in each of the three m-files. However, this will lead to replication of the code, and also to an increase in number of lines which is certainly not a good practice of programming. Instead of copying the code, you can wrap the implementation in a function and in each m-file, you can call the function using '[output] = myFunctionName(inputs)'. The code of the implementation is not replicated in this case, and in each m-file only one line of code is needed to call the function.

A third reason why functions could be helpful in programming is the ability to exchange these functions with ease among other programmers, without knowing exactly how the function is built. If, for example, some colleague students are interested in using your boxplot layout, you need to distribute only this one function. Your fellow students then only need to know what the input and output of this function is, and not how the function is implemented. So there are a bunch of reasons why functions are needed and in this chapter we want to go more into detail in how the functions needs to be constructed.

7.1 Building functions

A function is nothing else than an m-file with a function-header, which specifies the name of the function and the variable names of the input and the output, using the following syntax:

```
function [output_args] = functionName(input_args)
% FUNCTION description
...
    ...
end
```

The function begins with the code-word 'function', and ends with the word 'end'. In between those words the implementation of the function will be written. Via this implementation a series of output variables will be generated, which are listed in the 'output_args', separated by a comma. In order to calculate the output variables, some information is needed; this information is given via the input arguments 'input_args'. Once the function is made, you save it under the same name as the one you gave the function. When you want to use the function, you need to type the functions' name, with the appropriate input variables in between brackets, in the command window or in an m-file. Remark that function needs to be in the same folder as those of the m-file or the folder containing the function should be added to the search path. The latter method is a more proper way of programming, and will be deepened out in the section on good practice of programming. An example of a function which gives a plot of the mean and the mean plus and minus the standard deviation can be found in the next example:

1) Function to calculate mean and standard deviation:

```
function [mean_data, mean_plus, mean_min] = calc_mean_std(data)
mean_data = mean(data);
std_data = std(data);

mean_plus = mean_data + std_data;
mean_min = mean_data - std_data;

figure;
plot(mean_data);
hold on; plot(mean_plus, 'r')
hold on; plot(mean_min, 'g')
end
```

2) Code, needed to implement this function in another m-file:

```
load('Data/EMG');

[EMG_mean, meanEMG_plus, meanEMG_min] = calc_mean_std(EMG);
title('EMG'); xlabel('Samples'); ylabel('EMG (V)')
```

In the main file of this example a matrix is loaded which contains the EMG data of one muscle for six different trials (measured under the same walking conditions). In order to check the variability between trials, a plot of the mean value and the standard deviation will be made, using the function 'calc_mean_std'. The title and the axes labels can be included after the call to the function. The advantage of this function is that it can now be used for every kind of signals you want, for example also for displaying the mean and standard deviation or the measured ground reaction forces, as shown in the next code:

```
load('Data/GRF');

[GRF_mean, meanGRF_plus, meanGRF_min] = calc_mean_std(GRF);
title('Vertical ground reaction force'); xlabel('Samples'); ...
   ylabel('Vertical force (N)')
```

Instead of replicating the code of the function, we need to implement it once and it is applicable for all kinds of applications, on top of that it can be easily exchanged. A function can also be implemented without input and/or output arguments, by just leaving the list of arguments empty.

7.2 Global and local parameters

All variables in a function M-file are local. This means that they do not appear in the workspace and communication with this workspace occurs only via the input variables. This means that when the function is running, this function only 'sees' the input variables, and thus not the variables which are defined locally in the main file. The next example illustrates this:

1) Function:

```
function [out1, out2] = plus_multiply(in1, in2)
out1 = in1 + in2;
out2 = in1*in2;
end
```

2) Use of function:

```
a = 2;
b = 5;
[c, d] = plus_multiply(a, b);
```

A function named 'plus_multiply' is implemented, which calculates the sum and the multiplication of two input arguments. The local workspace of the function only contains four variables: the input variables 'in1' and 'in2' and the output variables 'out1' and 'out2'. In the main file of the example two variables 'a' and 'b' are initialized and the function is called with 'a' and 'b' as input. The output variables of the function are assigned to variables with names 'c' and 'd'. When going from the local workspace of the main file to the local workspace of the function and vice versa, the value of the variables were transferred, e.g. value of 'a' is stored in 'in1' and value of 'out2' is assigned to 'd'. Besides local variables, it is also possible to define global variables. These global variables are saved in a global workspace and can be accessed within every function. In order to define such a variable, you need to add the code-word 'global' in front of the variable name and you need to define the variable in the function implementation. For example:

1) Function:

```
function [out1, out2] = plus_multiply2(in1, in2)
global count;
out1 = in1 + in2;
out2 = in1*in2;
count = count + 1
end
```

2) Use of function:

```
global count;
count = 0;
a = 2;
b = 5;
[c, d] = plus_multiply2(a, b);
[e, f] = plus_multiply2(c, d);
[g, h] = plus_multiply2(a, f);
```

In this example a global variable 'count' is added to previous 'plus_multiply' function, which keeps track of the number of times the function is called. In the main file, this variable is defined as global and initialized to zero. Every time the function is called, the variable is increased, and the value is stored globally such that it can be accessed for the different calls of the functions.

It is also possible to call another function within a function. For the previous example is possible to separate the two operations and therefore split the function into two separate functions: (1) a function 'plus' and (2) a function 'multiply'.

1) Main function:

```
function [out1, out2] = plus_multiply3(in1, in2)
global count;
out1 = plus(in1, in2);
out2 = multiply(in1, in2);
count = count + 1
end
```

2) Summation function:

```
function out1 = plus(in1, in2)
out1 = in1 + in2;
```

```
end
```

3) Multiplication function:

```
function out1 = multiply(in1, in2)
out1 = in1*in2;
end
```

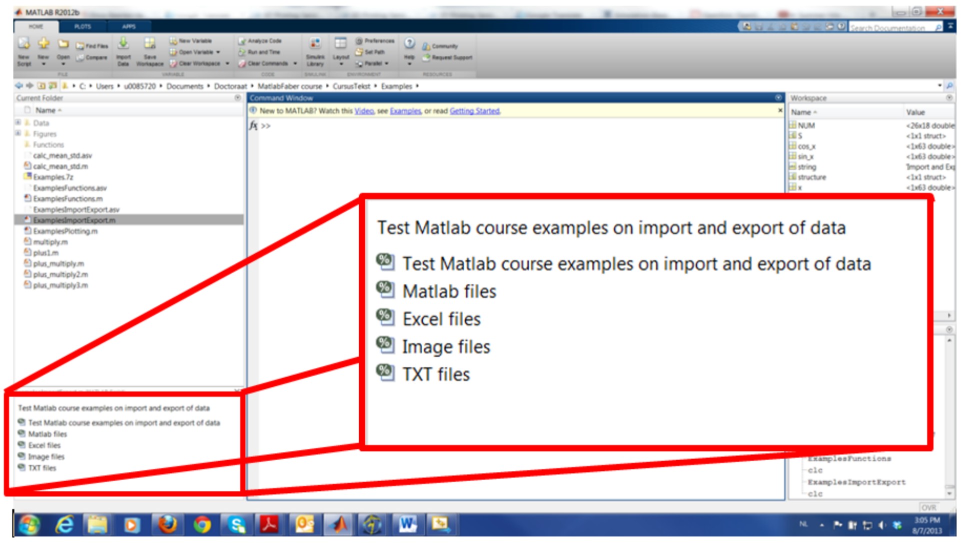
The previous example shows a very unrealistic simple implementation, but off course you are not limited to that. Every function or piece of code from previous chapters can be implemented and used in a function. More realistic and functional examples will be given in the exercise session.

7.3 Good practice of programming

When writing a large program, consisting of many functions, it could be difficult to keep an overview on the whole program, especially when you are working with different people over a long period of time. To make it easier, you should adopt a good practice of programming when writing code. One rule is to include comments in your code where needed. In MATLAB this is supported by using the symbol '%'. The whole line behind this symbol, colored in green, will be ignored by MATLAB when running the file or function. Using this symbol, it is possible to put extra information after operations and in the beginning of a command line. Using comments it is also possible to section your code, by typing the symbol twice '%%'. The text after this symbol will be shown in the preview window in MATLAB, to give an overview over the sections in the M-file. For example, the M-file with the examples of the chapter about import and export of data, could be structured as follows:

```
%% Test Matlab course examples on import and export of data
...
        ...
%% Matlab files
...
    ...
%% Excel files
...
    ...
%% Image files
...
    ...
%% TXT files
...
   ...
```

Figure 7.3.1 shows how the sectioning is made visible in the preview window, when you click on the M-file in the current folder window.



Test Matlab course examples on import and export of data
- Test Matlab course examples on import and export of data
- Matlab files
- Excel files
- Image files
- TXT files

Figure 7.3.1: Preview window showing the sectioning of an M-file by using %% comments.

Difficult parts of code should be accompanied by comments explaining the code broadly. Function M-files should contain header lines, explaining shortly the function, the input(s) and the output(s) of the function. The next sample of code shows an example of possible header lines for the function 'calc_mean_std', used in previous examples:

```
function [mean_data, mean_plus, mean_min] = calc_mean_std(data)
%% CALC_MEAN_STD Function calculating the mean, mean + standard
%   deviation and mean - standard deviation. It makes a plot of these
%   signals
%
%       * Input:    - data : Matrix containing the data which needs
%       to processed
%
%       * Output:   - mean_data : the mean of data, along the rows
%                   - mean_plus : mean + standard deviation
%                   - mean_min : mean - standard deviation
%
% Author: Wouter Aerts (07/08/2013)
...
    ...
end
```

Adding comments to the code is one way to keep the overview, another way is by saving the M-files in a structured way in a directory. When writing large programs, you will end up with a large amount of functions. Instead of saving these function M-files in the same directory as the main M-file, we could save these files in for example a directory called 'Functions'. In order to be able to run the functions, MATLAB must know where these M-files were situated, and therefore you need to add the path of the directory to the search path of MATLAB. This can be done with the function 'addpath(genpath('path_name_string'))', where 'path_name_string' is a string containing the path name to the directory 'Functions', e.g. 'C:\Users\u0085720\Documents\MATLAB\Examples\Functions'. It is however difficult to know what the exact path to your functions directory is, and if you want to share your code with colleagues, this pathname needs to be adapted to the new path name of colleagues' computer. However, there is a solution for this problem. MATLAB is able to automatically determine what the directory of the main file is. The following code shows an example of how this is implemented for a main file called 'ExamplesFunctions.m':

```
file_name = 'ExamplesFunctions.m';
function_dir = which(file_name);
function_dir = function_dir(1:end-length(file_name));
addpath(genpath([function_dir, 'Functions']))
```

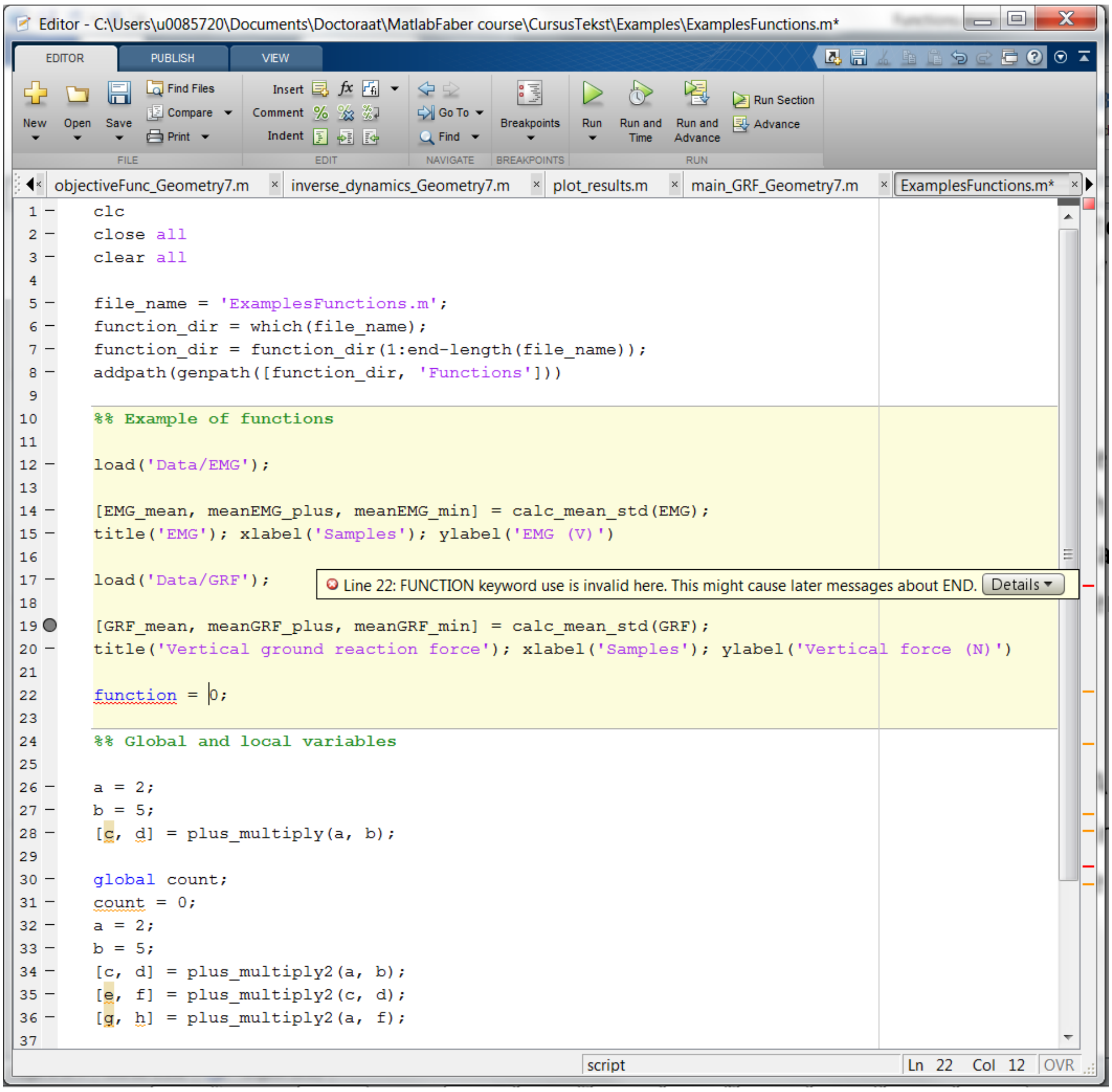
The first step in the example above is the specification of the main file. Next, the pathname of this file is determined with the command 'which'. The string is a full path name and after removing the name of the file and adding the directory name 'Functions' to the string, this serves as input to the command 'addpath(genpath())'. With this part of code, the directory 'Functions' and all of its subdirectories are added to the search path of MATLAB. For a further increase of the overview, it is even possible to subdivide the functions directory into different subdirectories which bundle different M-files with the same kind of function. For example if you are writing a program to process biomedical data, possible subdirectories are 'Import', 'Preprocessing', 'Calculations', 'Plotting', etc.

7.4 Debug

Mistakes happen and errors occur, even the most experienced programmers are not able to program without any errors. Debugging is a method to find the mistakes in order to correct them. Debugging is needed because when you are working with several functions, the errors could be hidden in different layers of the code. There are different kinds of errors, which will be detect and solved differently. The most common errors are the syntax errors, the initialization errors and the logical errors. Each of these errors will be handled in detail in the next sections.

7.4.1 Syntax errors

A first kind of error is an error to the syntax of the MATLAB code, or with other words, violating the MATLAB programming language. Examples are: missing ')', '}', or ']'; invalid use of keywords like 'function', 'while', 'for'; missing a keyword 'end'; etc. These errors will be automatically detected by MATLAB, even before you have ran the function or main file. MATLAB will underline the error and give a red error sign in the bar at the right side of the script window. When you place your cursor on the red line in the side bar, it will give more information about the error: where it is situated (line number) and a short description of the error. The line numbers are indicated in the left side bar of the script window. Figure 7.4.1 shows an example of such an error detection and description. In the example the use of the keyword 'function' is invalid. The bar at the right side give also some warnings, indicated in orange. These are no errors and so the program will run perfectly, but these are parts of the code that could be optimized, mostly to increase execution speed. MATLAB gives a suggestion on how to optimize your code when placing your cursor on the orange symbol in the bar.



```
1  clc
2  close all
3  clear all
4
5  file_name = 'ExamplesFunctions.m';
6  function_dir = which(file_name);
7  function_dir = function_dir(1:end-length(file_name));
8  addpath(genpath([function_dir, 'Functions']))
9
10 %% Example of functions
11
12 load('Data/EMG');
13
14 [EMG_mean, meanEMG_plus, meanEMG_min] = calc_mean_std(EMG);
15 title('EMG'); xlabel('Samples'); ylabel('EMG (V)')
16
17 load('Data/GRF');
18
19 [GRF_mean, meanGRF_plus, meanGRF_min] = calc_mean_std(GRF);
20 title('Vertical ground reaction force'); xlabel('Samples'); ylabel('Vertical force (N)')
21
22 function = 0;
23
24 %% Global and local variables
25
26 a = 2;
27 b = 5;
28 [c, d] = plus_multiply(a, b);
29
30 global count;
31 count = 0;
32 a = 2;
33 b = 5;
34 [c, d] = plus_multiply2(a, b);
35 [e, f] = plus_multiply2(c, d);
36 [g, h] = plus_multiply2(a, f);
37
```

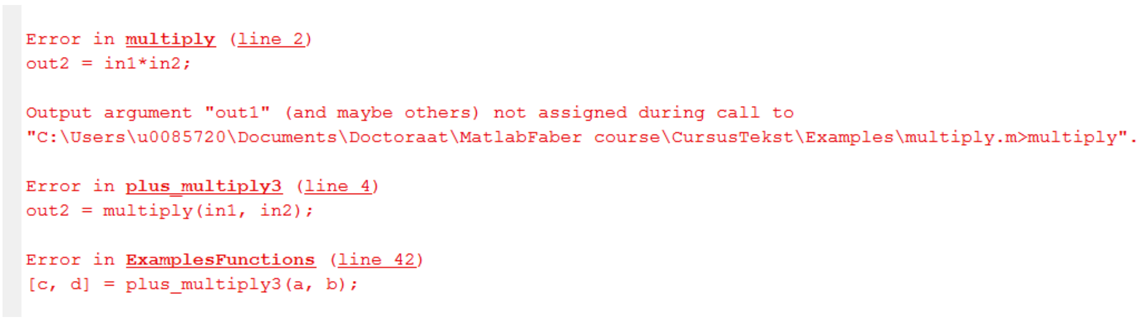
Line 22: FUNCTION keyword use is invalid here. This might cause later messages about END.

Figure 7.4.1: Example of a syntax error. MATLAB automatically detects this kind of error and will indicate it with a red line underneath the error and a red sign in the side bar at the right.

7.4.2 Initialization errors

A second kind of error is the initialization error. This is an error which occurs when a variable, that is not yet initialized, is used in a certain function or operation, or with other words, a value is not yet assigned to the variable. This is an error which is not detected before the function is called, but once running MATLAB will detect the error automatically. The program will stop running and an error message will appear in the command window stating what the error is and where this error is situated. If the error occurs inside a function, the whole stack of the called functions will be displayed. For example if for the 'multiply and plus' example the variable 'out1' is not initialized in the 'multiply' function, an error description is given as shown in figure 7.4.2.

```
function out1 = multiply(in1, in2)
    out2 = in1*in2;
    end
```



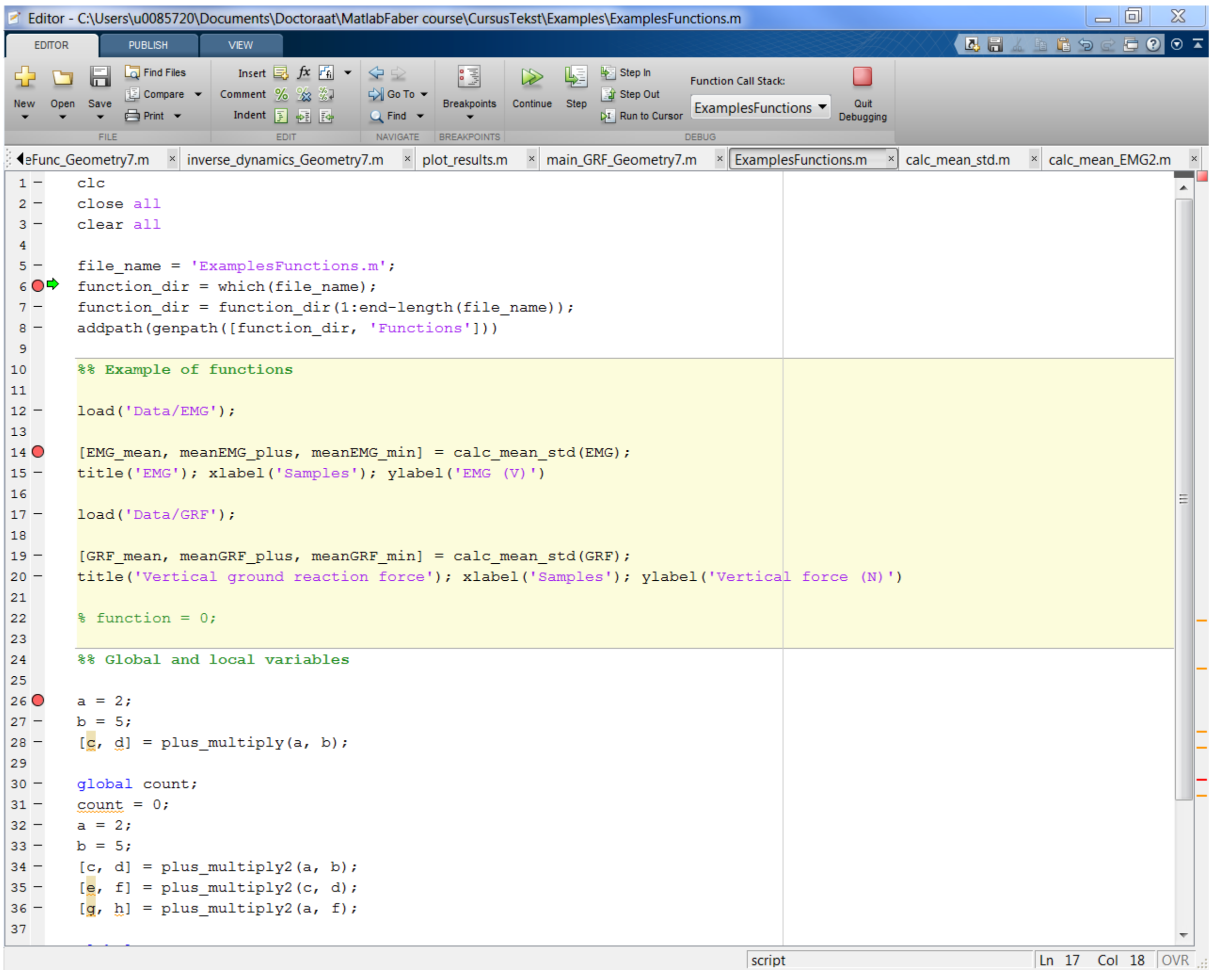
```
Error in multiply (line 2)
out2 = in1*in2;

Output argument "out1" (and maybe others) not assigned during call to
"C:\Users\u0085720\Documents\Doctoraat\MatlabFaber course\CursusTekst\Examples\multiply.m>multiply".

Error in plus_multiply3 (line 4)
out2 = multiply(in1, in2);

Error in ExamplesFunctions (line 42)
[c, d] = plus_multiply3(a, b);
```

Figure 7.4.2: Example of an error description as given by MATLAB. The initialization error is situated in the function 'multiply', which is a function called in function 'plus_multiply3', which in turn is called in the main file ExamplesFunctions.

7.4.3 Logical errors

The third kind of error is the logical error, which is also the most difficult error to detect. A logical error is an error to the outcome of the program. Everything seems to work, but after running the program, the results are not those you would have expected. For example you expect that the 'multiply' function multiply the two input variables, but after testing the program, it seems that the input variables where divided. This is a logical error. For this simple example it is quite easy to detect the error, but for larger programs, existing out of hundreds of functions, it is quite challenging to detect those problems. One tool to help detecting these errors is the debug tool. This tool allows you to run the code line by line and to inspect the local work space. In order to stop running when reaching a specific line of code, you can set a breakpoint, just by clicking on the small line next to the line number. A red circle will appear. When running the program (by clicking on the 'run'-button or pressing F5) the program will stop at the breakpoint.

It is also possible to add multiple breakpoints, and when running the program, it will stop at each breakpoint. Figure 7.4.3 shows a screen view of an example where breakpoints were added at line 6, line 14 and line 26. The green arrow indicates the line the program reached. For further progressing, several options are possible, all indicated by buttons in the debug toolbar. This is the toolbar which appears immediately after pushing the 'Run'-button in the run toolbar, which evolves to the debug toolbar, as shown in figure 7.4.4.



```
1   clc
2   close all
3   clear all
4
5   file_name = 'ExamplesFunctions.m';
6   function_dir = which(file_name);
7   function_dir = function_dir(1:end-length(file_name));
8   addpath(genpath([function_dir, 'Functions']))
9
10  %% Example of functions
11
12  load('Data/EMG');
13
14  [EMG_mean, meanEMG_plus, meanEMG_min] = calc_mean_std(EMG);
15  title('EMG'); xlabel('Samples'); ylabel('EMG (V)')
16
17  load('Data/GRF');
18
19  [GRF_mean, meanGRF_plus, meanGRF_min] = calc_mean_std(GRF);
20  title('Vertical ground reaction force'); xlabel('Samples'); ylabel('Vertical force (N)')
21
22  % function = 0;
23
24  %% Global and local variables
25
26  a = 2;
27  b = 5;
28  [c, d] = plus_multiply(a, b);
29
30  global count;
31  count = 0;
32  a = 2;
33  b = 5;
34  [c, d] = plus_multiply2(a, b);
35  [e, f] = plus_multiply2(c, d);
36  [g, h] = plus_multiply2(a, f);
37
```

Figure 7.4.3: Example of debugging a piece of code. The red circles after the line numbers indicate the breakpoints. The green arrow indicates the line which will be executed next.

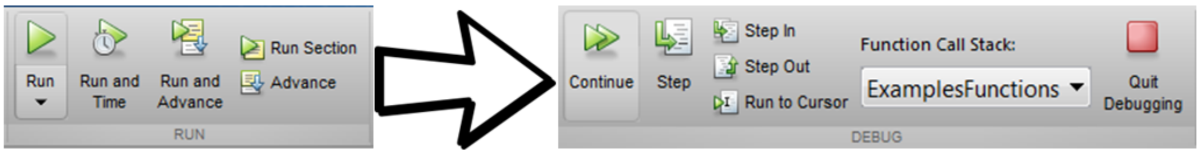


Figure 7.4.4: After pushing the run-button (or pressing F5) the run toolbar will evolve to a debug toolbar.

The debug toolbar has buttons which allow you to progress through the code. The first button is the 'Continue'-button. When pressing this button (or pressing F5) the program will continue until it hits the next breakpoint. This could be a breakpoint in the main file, as well as a breakpoint in one of the function files. The next button is the 'Step'-button, which allows you to go line by line through the code. Each line (a function or just an operation) will be executed, without going inside the function. The next button 'Step In', will allow you to go into the function. The workspace will then change to the local workspace of this function. In that case, the variables of the main file are not visible, until you leave the function. This step-in button not only works for own build functions, but also works for MATLAB built-in functions (e.g. mean(), max(),etc.). Be careful that you don't change anything in these files when you dive into these functions. This could lead to a malfunctioning of MATLAB. With the 'Step Out'-button, you can go out of the function and go one function higher in the function call stack. This function call stack is shown next to the buttons and gives the sequence of functions which have been called. With the 'Run to Cursor'-button it is possible to let the program run till the line of code where your cursor is situated. The 'Quit Debugging'-button at the outer right will stop the debugging and change the debug toolbar back to the normal run toolbar.

The Run toolbar also has several buttons related to several ways of running a program. The most important button, as you already know, is the 'Run'-button (also F5), which is used to run the program. The 'Run and Time'-button will run the program and will additionally give the execution time of every function. This could be very helpful to determine which function is the slowest in your program in order to improve this functions' speed if possible. The button 'Run and Advance' allows you to run one section (divided by the double comment sign (%%) and marked in yellow) and stop at the beginning of the next section. The function of this button is also split in the two last buttons 'Run Section' and 'Advance'.

8 Flow control

8.1 Conditional statements

Very often, it is necessary to perform operations only when certain conditions are met. To define these specific conditions, relational and logical operations are used. Simple flow control in MATLAB is performed with the *if* and *switch* statements. The selection of the statements executed by if statement is based on whether the condition is true or false. While the execution of switch statements is based on a number of possible options depending on the value of the switch expression.

8.1.1 If-elseif-else construct

In computer programming, situations exist in which we want to apply different statements depending on different conditions. In those situations the *if* statement is frequently used. The basic structure for an *if* statement is the following:

```
if conditions #1
statements #1
elseif conditions #2
statements #2
:
else
final statements
end
```

The if statement evaluates a logical expression, and executes a group of statements when the expression is true. The optional elseif and else keywords provide for the execution of alternate groups of statements only when the previous expressions in the if-block are false. An if-block can include multiple elseif expressions. The else statements are only evaluated if the previous (else)if expressions are false. Note that the else statement itself has no logical condition. When one expression is verified, the statements associated with it are executed and the if-block does not continue verifying the rest of the expressions. The end keyword terminates the last group of statements to conclude the if-block. The use of end is mandatory after an if statement, regardless there is an else(if) as well. A graphical method of designing an if-statement is shown in the figure below.

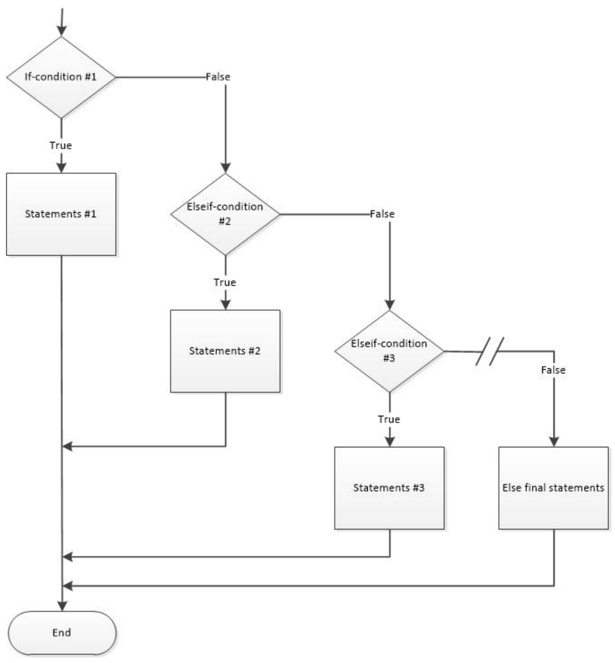


Figure 8.1.1: Flow chart of if/elseif /else statement.

An evaluated expression is true when the result is nonempty and contains nonzero elements (logical or real numeric). Otherwise, the expression is false. Expressions can include any combination of arithmetic, relational, and logical operators. MATLAB evaluates the compound expressions from left to right, meeting the operator precedence rules. The use of relational and logical operators is explained in the previous chapter. Within an if expression, all logical operators, including | and &, short-circuit. With short-circuiting, the evaluation of an expression is stopped as soon as the results of the entire expression is known. In other words, the second operand is only evaluated when the result is not fully determined by the first operand. The simplest form of an if-statement contains only the if and end keywords, this can be used to evaluate one condition. The first line starts with if, followed by the condition you want to test for. The lines between the if and end keywords describe the statements which are executed if the condition is true.

```
B = −10;
if B < 0
disp('B is less than 0');
end
```

Instead of using two if statements, you can use an if ... else ... end construction instead. The statements followed by the else are only executed if the previous if-statements are false. The else statement has no logical condition. Here is a code that checks the age of a person:

```
age = 17;
if age ≤ 18;
        disp('Person is 18 or younger');
else
        disp('Person is older than 18');
end
```

The above example has only two conditions: either the person is 18 or younger, or the user is older than that. Try to adapt the above example by changing the age to 20 and run the code again. The message in the else statements should display in the output window. You can test for more than two choices by introducing the elseif keyword followed by a logical condition that it evaluates if the preceding if (and possible elseif condition) is false. For example if you want to test for more age ranges; e.g. 18 or younger, 19 to 21, and 22 and over.

```
age = 17;
if age ≤ 18;
        disp('Person is 18 or younger');
elseif age < 22;
        disp('Person is between 19 and 21');
else
        disp('Person is older than 21');
end
```

The first if tests for condition number one, in the above example 18 or under. Next the elseif statement is tested. Anything not caught by the first two conditions will be caught by the final else. You can have multiple elseif statements within an if block. The previous if blocks have shown examples using relational operators to define the conditions to test. However, the conditions can include any combination of arithmetic, relational, and logical operators as long as the output will give a true or false output (logical). The following example combines relational and logical operators to solve the problem. If we want to find the smallest of three given numbers, we know that if A is the smallest, then it must be smaller than the other two. Moreover, the condition for A number being the smallest is mutually exclusive. This will lead to the following definition of the if construct:

```
if (A < B) && (A < C)
        Result = A;
elseif (B < A) && (B < C)
        Result = B;
else
        Result = C;
end
```

Test the previous example with different values for A, B and C.

8.1.2 Switch-case-otherwise construct

The switch statement executes groups of statements depending on the value of a variable or expression. The basic structure of a switch construct consists of four keywords; switch, case, otherwise and end.

```
switch switch_expression
case case_expression #1
        statements #1
case case_expression #2
    statements #2
:
otherwise
    final statements
end
```

The keyword switch is followed by an expression to evaluate, resulting in a scalar or string. The keywords case and otherwise delineate the groups of possible statements for the switch expression. The end keyword terminates the last group of statements to conclude the switch-block. A graphical representation of a switch statement is shown in the flow chart diagram in the figure below.

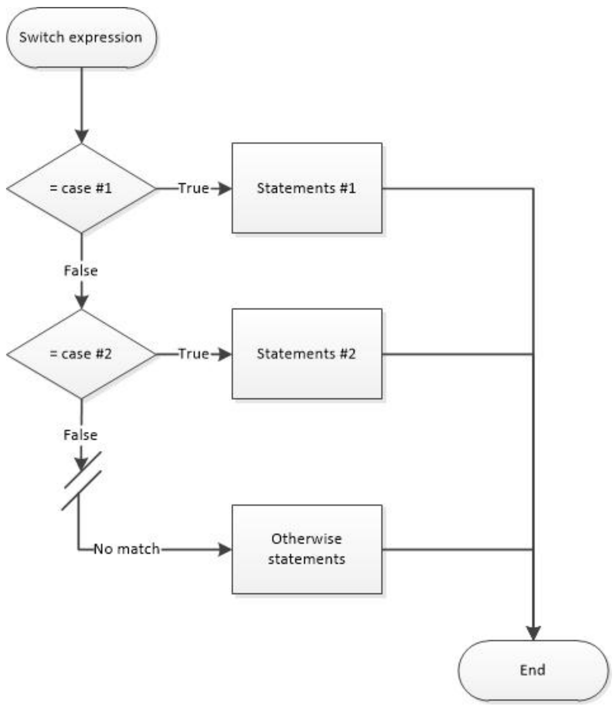


Figure 8.1.2: Flow chart of switch statement.

A switch-block can contain any number of case groups. Each case keyword is followed by a possible value for the switch expression. This value could be a scalar, string, or a cell array of scalars or strings. The cell array can be used to handle multiple conditions in a single case statement. The switch expression is compared to each case value until one of the cases is true. When a case is true, MATLAB executes the corresponding statements, and then exits the switch block. So, only the first matching case is executed. If no match is found among the case statements, then MATLAB skips to the (optional) otherwise statements, or else to the end statement. Note that a case expression cannot include any relational operators to compare against the switch expression. To test for inequality, use the if-statements instead.

For numeric expressions, a case statement is true if the case expression is equal to the switch expression (eq(case _expression,switch _expression)). Depending on the value of the switch expression, signal number, one of the three cases will be executed. The otherwise statement will only be executed if no match is found, in this case if the signal number is not equal to 1, 2 or 3.

```
time = 0:0.1:2*pi;
signal = [sin(time);cos(time);2*sin(time)];
n_signal = 1;   % signal number
switch n_signal
        case 1
            out_sig = signal(1,:);
        case 2
            out_sig = signal(2,:);
        case 3
            out_sig = signal(3,:);
        otherwise
            out_sig = [];
end
```

For string expressions, a case statement is true if the string of the case expression is equal to the string of the switch expression (strcmp(case _expression,switch _expression)). The switch structure can for example be used to decide which plot needs to be created based on the value of the string plottype:

```
X = rand(30,4);
plottype = 'bar';
switch plottype
        case '2Dline'
            plot(X)
            title('2D line graph')
        case 'box'
            boxplot(X)
            title('Box plot')
        case 'bar'
            bar(X)
            title('Bar graph')
        otherwise
            warning('Unexpected plot type');
end
```

For a cell array case expression, at least one of the elements of the cell array matches switch expression, as defined above for numeric and string expressions. This can be used to use the switch statement to handle multiple conditions in a single case statement by enclosing the case expression in a cell array. Instead of defining 12 cases for every month to decide the season a cell array is used to define the months of one season leaving 4 cases.

```
month = 10;
switch month
    case {12, 1, 2}
        disp([num2str(month) ' is a winter month'])
    case {3, 4, 5}
        disp([num2str(month) ' is a spring month'])
    case {6, 7 , 8}
        disp([num2str(month) ' is a summer month'])
    case {9, 10, 11}
        disp([num2str(month) ' is a fall month'])
    otherwise
        disp([num2str(month) ' is not a valid month'])
end
```

8.2 Loop control statements

In programming there may be situations in which we need to execute the same block of code several times. For example if you need to create a plot using different data inputs or if you want to check the average grade of every student in the class. In such cases we will use a loop. A loop is a sequence of statements which is specified once but which may be carried out several times in succession. The function of the loop is to repeat the block of code a given number of times or until it meets a certain condition. In MATLAB we can distinguish two types of loops; for and while loops. 'For statements' loop a specific number of times. Whereas, 'while statements' loop as long as a condition remains true.

8.2.1 For

Most programming languages have constructions for repeating a loop a fixed, predefined number of times. These loops are known as count-controlled loops or for loops. The basic syntax for a for loop is the following:

```
for index = values
program statements
    :
end
```

A graphical method to design a for loop is shown in the figure below.

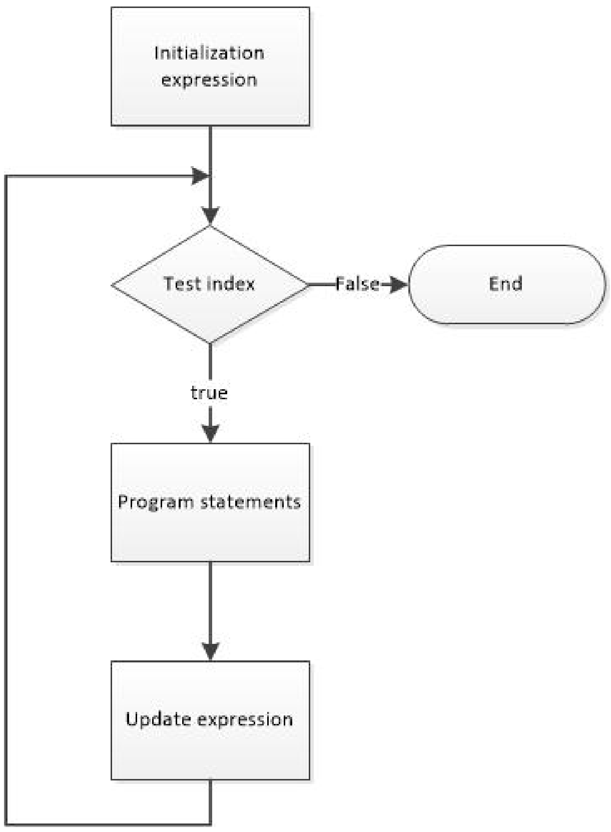


Figure 8.2.1: Flow chart diagram of a for-loop.

The program statements are repeatedly executed as defined in the for (index) statement. The block of statements is always concluded with the end keyword to close the for-loop. The for-loop will continue to do the specified statements for a certain amount of time depending on the definition of the index. The index is in every loop updated until the last index is evaluated, at that point the loop will stop. The most simple form to define the index is by index = initval:endval, this loop increments the index variable from initval to endval by 1. The standard increment size in MATLAB is 1. It is important to note that the index can have any name just like any other variable in MATLAB. More generally, the increment step can be defined; index = initval:step:endval. The statements are executed repeatedly starting from initval with index incremented by step if the value is positive, or decremented when step is negative for each iteration until the endval is reached. The number of iterations for this index statement is equal to ((endval-initval)/step)+1, if this is not an integer then it will be rounded down to the next integer. By means of good programming practice, avoid assigning a value to the index variable within the body of a loop. The for statement overrides any changes made to the index within the loop.

As an example of a for loop with an increment size of 1, consider the computation of the following series.

$T_n = \sum_{(j=1)}^{n} (-1)^{(j+1)} \frac{1}{j} (= +\frac{1}{1} - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + ... + (-1)^{n+1} \frac{1}{n})$

This is implemented in a script file containing the following statements

```
n = 100000;
sign = -1;
a = 0;
for j = 1:n
        sign = -sign;
        a = a + sign.*1./j;
end
Tn = a
```

The next example creates a sinus plot with three different frequencies, with an increment size of 3:

```
x = 2*pi*(0:0.01:1);
figure, hold on
colors = {'r','g','b','y','m','c','k'}; % line colors
i = 1;  % start index color
for a = 1:3:9
        plot(x, sin(a*x),'color',colors{rem(i,7)})
        i = i+1;    % update index color
end
```

As an exercise try to change the increment size and check what is happening to the plot.

Even more generally the index can be defined by an array. MATLAB creates a column vector index from subsequent columns of the defined array. The statements inside the loop are repeatedly executed with the index set to each of the element of the vector in turn. The input array can be of any MATLAB data type, including a string, cell array, or struct. In many programming languages including MATLAB, only real positive integers can be used to define the subscript indices in the for-loop. Since, an array cannot have $0^t h$ or negative elements. The use of subscript indices is useful if you want to construct a recursive row, such as

$A_{(k+1)} = \frac{1}{3}A_k + 4, k = 1, 3, 4, ...and A_1 = 1$

```
A(1) = 1;
for k = 1:4
        A(k+1) = A(k)./3 + 4
end
```

Another example in which the use of subscript indices is necessary is for assigning matrices. In those cases nested loops are used for matrix assignment. A nested loop is a loop within a loop, an inner loop within the body of another one. During the first pass of the outer loop, the inner loop is triggered, which executes to completion. During the second pass of the outer loop triggered again and executed to completion. This repeats until the outer loop finishes. It is a good idea to indent the loops for readability, especially when they are nested. When using nested loops be sure that each loop is closed with the end keyword. The following script creates a 4 by 3 matrix. Copy this scrip in a MATLAB m-file to analyze the result.

```
nr = 4; % number of rows
nc = 3; % number of columns
H = zeros(nr,nc);
for i = 1:nr
    for j = 1:nc
        H(i,j) = i+j
    end % inner loop
end % outer loop
```

It is also possible to combine the for loop with if and/or switch statements. In the following example we create based on the vector values three groups.

```
C = [7,12,9,18,14,6];
for i = 1:length(C)
        if C(i) ≤ 10
                D(i) = 1;
        elseif C(i) ≤ 15;
                D(i) = 2;
        else
                D(i) = 3
        end % if statement
end % for loop
```

8.2.2 while

Most programming languages have constructions for repeating a loop until some condition changes. These loops are known as condition controlled loops or while loops. Note that some variations place the conditional test before the statements are repeated, while others have the test at the end of the loop. In the former case the body may be skipped completely, while in the latter case the body executed at least once. In MATLAB the conditional expression of the while statement is tested before the statements are reported. The keyword end terminate the block of code.

```
while conditions
program statements
:
end
```

As long as the expression remains true, the while loop will repeatedly execute the defined statements, as shown in the flowchart of Figure 7. An evaluated expression is true when the result is nonempty and contains all nonzero elements (logical or real numeric).

Otherwise, the expression is false. Expressions can include relational operators and logical operators. MATLAB evaluates compound expressions from left to right, adhering to operator precedence rules. Within a while expression, all logical operators, including — and &, short-circuit. With short-circuiting, the second operand is only evaluated when the result is not fully determined by the first operand.

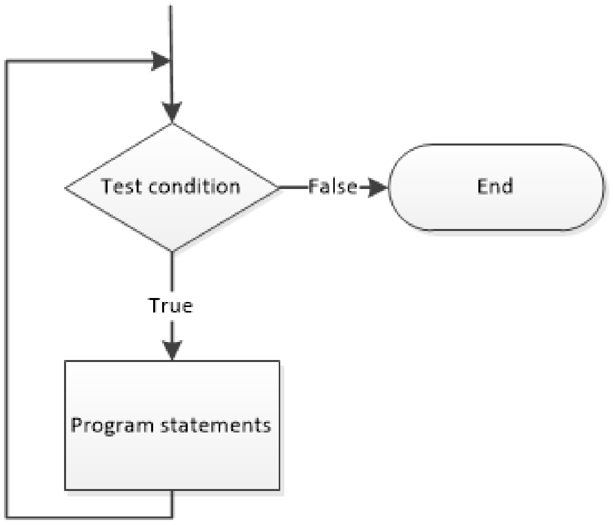


Figure 8.2.2: Flow chart of while loop.

The while-loop is frequently used when the number of iterations before the loop starts is unknown. However, it must eventually become false after a finite number of steps, or the program will never terminate. If you accidentally create an infinite loop, a loop that never ends on its own, the loop can be executed by pressing Ctrl+C.

A simple example of a while loop is to find the first integer n for which its factorial is a 100-digit number. The factorial, denoted by n!, of a non-negative integer n is the product of all positive integers less than or equal to n.(e.g. 5!=5x4x3x2x1) The first step is to initialize the start integer in our case n = 1. While the factorial is less than a 100-digit number the loop will continue. The start integer is updated in every loop, but you have to create the definition yourself.

```
n = 1;
nFactorial = 1;
while nFactorial < 1e100
    nFactorial = nFactorial * n;
    n = n + 1; % update n value
end
```

You can use the while loop also in a nested form.

8.2.3 Continue

The continue statement passes control to the next iteration of the for or while loop in which it appears, skipping any remaining statements in the body of the loop. The same holds for continue statements in nested loops. The execution continues at the beginning of the loop in which the continue statement was encountered.

The following example computes the product of an array of elements. In case an element of the input array equals to zero the program will skip that particular element in the computation of the product.

```
B = [3, 5, 7, 0, 20];
ProductB = 1;
for i = 1:length(B)
    if B(i) == 0 % When B(i) equals to zero, it skips the statement ...
       ProductB = ProductB * B(i)
            continue;
    end
    ProductB = ProductB * B(i);
end
```

Compare the result of the following computation with the result of prod(B) command in MATLAB. prod is a standard MATLAB function which computes the product of array elements. However, the continue statement is rarely needed and its use should be avoided. For example the previous code example could be written as

```
B = [3,5, 7, 0, 20];
for i = 1:length(B)
    if B(i) ≠ 0
        ProductB = ProductB * B(i);
    end
end
```

8.2.4 Break

The break statement terminates the execution of a for or while loop before all predefined conditions are met. The remaining statements in the loop that appear after the break statement are not executed. This can be useful if you want to stop running a loop because a condition has been met other than the loop end condition. In nested loops, break exits only from the loop in which it occurs. Control passes to the statement that follows the end of that loop. The following example shows a while loop which is always true, and therefore will never stops to run without another statement. In this example we introduce the break command, the while loop will be terminated when b is greater than 5.

```
b = 0;
while 1
        b = b + 1;
        if b > 5
            break;
        end
end
```

If you inadvertently create an infinite loop (a loop that never ends on its own), stop execution of the loop by pressing Ctrl+C.

9 Fourier Analysis

In signal analysis, 'frequency' is a central concept. This is not only so for auditory signals. The concept of frequency is most obvious here: hearing performance is often evaluated using an audiogram i.e. audible threshold for standardized frequencies. However, frequencies come to play when analyzing most if not all signals.

9.1 Sine and cosine as corner stone for signal analysis

Al signals can be expressed as a summation of a series of sine and cosine functions with variable amplitude and frequency. The example below might help in clarifying this concept. Each sine or cosine wave is specified by an amplitude and frequency. Therefore a common notation is

$F(x) = b \times sin(cx + d)$ or $f(x) = b \times cos(cx + d)$

with b being the amplitude and c the period. The period of the signal is characterized as 1/frequency. In signal analysis, often the angular frequency ? is used. It is a scalar measure of rotation rate. Angular frequency (or angular speed) is the magnitude of the vector quantity angular velocity. One revolution is equal to $2\pi radians, hence?{=}2\pi/T = 2$ where: w is the angular frequency or angular speed (measured in radians per second), T is the period (measured in seconds), f is the ordinary frequency (measured in hertz) (sometimes symbolised with ?). In figure 1 and 2 you see three sine and cosine waves with varying period (T= 1s, 2/3s, 4s) but amplitude 1. You see that in these notations, the angular frequency is used with a frequency of 1 being equal to 2?

10 Appendix

10.1 Arithmetic operators

Operators		**Description**
$+$	plus, uplus	Addition, unary plus
$-$	minus, uminus	Subtraction, unary minus
$.*$	times	Array multiplication
$*$	mtimes	Matrix multiplication
$./$	rdivide	Array right division
$.\backslash$	ldivide	Array left division
$/$	mrdivide	Matrix right division. Solves the system of linear equations for x.
$\backslash$	mldivide	Matrix left division. Solves the system of linear equations for x.
.^	power	Array raised to a power
^	mpower	Matrix or scalar raised to a power (exponent)
$kron$		Kronecker tensor product
$.'$		Real transpose
$'$		Complex conjugate transpose
$:$		Create vectors, array subscripting, and for loop iterations.
$=$		Assignment

10.2 Relational operators

Operators		**Description**
$<$	lt	Less than
$<=$	le	Less than or equal to
$>$	gt	Greater than
$>=$	ge	Greater than or equal to
$==$	eq	Equal to
$\sim=$	ne	Not equal to
isequal		Test arrays for equality
isequaln		Test arrays of equality, treating NaN values as equal

10.3 Logical operators

Operators		**Description**
~	not	Find logical NOT of array or scalar inputs.
&	and	Find logical AND of array or scalar inputs.
|	or	Find logical OR of array or scalar inputs.
xor		Find logical exclusive-OR of array or scalar inputs. True if only one input is true, but not both.
&&		Short-circuit AND of scalar inputs.
||		Short-circuit OR of scalar inputs.

10.4 Logical functions

Command	**Description**
all	Determine if all array elements are nonzero or true
any	Determine if any array elements are nonzero
exist	Check if a variable or file exists
*is**	Detect state
ischar	Determine if input is an character array
isempty	Determine if input is an empty array
isinf	Determine if input is value an infinite number (Inf)
islogical	Determine if input is logical array
isnumeric	Determine if input is numerical array
ismember	Array elements that are members of set array
isnan	Determine if input value is not-a-number (NaN)

10.5 Flow control

break	Terminates the execution of a for or while loop before all predefined conditions are met
case	Case switch
continue	Passes control to the next iteration of the for or while loop
else	Conditionally execute statements, used in if statements
elseif	Conditionally executes statements, used in if statements
end	Terminate for, while, switch and if statements or indicates the last element of an array
for	Repeat statements a specific number of times
if	Conditionally execute statements
otherwise	Optional part of a switch statement
switch	Switch among several cases based on expression
while	Repeat statements an indefinite number of times until all predefined conditions are met

10.6 Creating matrices

eye	Identity matrix
linspace	Generate linearly spaced vectors
logspace	Generate logarthmically spaced vectors
ones	Create an array of all ones
rand	Uniformly distributed random numbers and arrays
randn	Normally distributed random numbers and arrays
$repmat(x, m, n)$	Defines an m-by-n matrix in which each element is x
*zero*s	Create an array of all zeros
: (*colon*)	Regularly spaced vector

10.7 Special Values

ans	The most recent variable that has not been assigned to another variable.
eps	Smallest incremental number, the value of this variable is approximately relative accuracy of MATLAB
i	The complex number i with the value of $\sqrt{-1}$
inf	Infinity
j	The complex number j with the value of $\sqrt{-1}$
NaN	Not a number, and invalid numerical value e.g. 0/0. A calculation with NaN results in NaN.
pi	Value of $\pi = 3.1415$
$realmin$	Smallest floating-point number, $2.2251e - 308$
$realmax$	Largest floating-point number, $1.7977e + 308$
$nargin, nargout$	Number of function arguments
$varargin, varargout$	Pass of return variable numbers of arguments

10.8 Elementary math functions

Trigonometry	
sin	$\sin(x)$; sine of argument in radians
sind	$\sin(x)$; sine of argument in degrees
asin	$\sin^{-1}(x)$; inverse sine in radians
cos	$\cos(x)$; cosine of argument in radians
cosd	$\cos(x)$; cosine of argument in degrees
acos	$\cos(x)^{-1}$; inverse cosine in radians
tan	$\tan(x)$; tangent of argument in radians
tand	$\tan(x)$; tangent of argument in degrees
atan	$\tan(x)^{-1}$; inverse tangent in radians
$atan2$	Inverse tangent function with two arguments
Exponents and logarithms	
exp	e^x; exponential
log	$\ln(x)$; natural logarithm
$log2$	$\log_2(x)$; base 2 logarthm
$log10$	$\log_{10}(x)$; base 10 logarithm
sqrt	$\sqrt{x}$; square root
abs	Absolute value and complex magnitude
Special arithmetic operators	
ceil	Round towards positive infinity
fix	Round towards zero
floor	Round towards negative infinity
mod	Modulus after division
rem	Remainder after division
round	Round towards nearest integer
imag	Imaginary part of a complex number
real	Real part of a complex number

10.9 Statistics

$corrcoef$	Correlation coefficients
cov	Covariance matrix
max	Largest elements in array
$mean$	Average or mean value of array
$median$	Median value of array
min	Smallest elements in array
$mode$	Most frequent values in array
std	Standard deviation
var	Variance

10.10 Plotting commands

$figure$	Creating new figure window
$plot$	Plot x-y graph
$hold\ on$	Keep current plots
$xlim, ylim, zlim$	Setting limits to the axes
$xlabel, ylabel, zlabel$	Adding labels to the axes
$title$	Adding a title to the plot
$Legend$	Adding a legend to the plot
$subplot$	Dividing plot in subplots
$Scatter, scatter3$	Scatter plot in 2D and 3D
$boxplot$	Boxplot
$bar, bar3$	Bar plot in 2D and 3D
$barh, barh3$	Horizontal bar plot in 2D and 3D
$plot3$	Plot x-y-z graph
$image$	Image
$axis\ off$	Hide axes
$meshgrid$	Conversion of vectors into a grid (matrix)
$Mesh, meshc$	Wireframe mesh, without and with contour plot
$Surf, surfc$	Three-dimensional shaded surface, without and with contour plot
$Colormap$	Conversion matrix for implemented colors

10.11 Import and export functions

load	Loading mat-files
save	Saving mat-files
saveas	Saving figures and plots
importdata	Import of ASCII files
uiimport	Graphical User Interface version of importdata
xlsread	Reading Excel files
xlswrite	Writing Excel files
imread	Reading image
imwrite	Writing images

10.12 Good practice of programming

%	Comment
%%	Sectioning of the code
$CTRL + R$	Comment selected lines
$CTRL + T$	Uncomment selected lines
function	Keyword to construct a function
clc	Clear command window
closeall	Close all figure windows
clearall	Clear all variables of the workspace
$CTRL + C$	Interrupt the program
...	Continue to next line