



NTNU – Trondheim
Norwegian University of
Science and Technology

AN INTRODUCTION TO
MATLAB AND SIMULINK

Trondheim
February 26, 2014

Contents

1	Introduction	2
1.1	Download and installation of MATLAB	2
1.2	References and Matlab tutorials	2
2	MATLAB basics	3
2.1	The MATLAB workstation	3
2.2	Getting help	4
2.3	Saving and loading data	5
3	Use MATLAB as a calculator	6
4	Matrix calculation	7
4.1	Creating Matrices	7
4.2	Indexing matrices	8
4.3	Array operations	9
4.4	Elementwise operations	9
4.5	Array of function values	10
4.6	Transposition	11
5	Visualization	11
5.1	Plotting several graphs	12
5.2	Graphical screen handling	12
5.3	Other graph types	13
6	Programming in MATLAB	15
6.1	Script m-Files	15
6.2	User defined functions	16
6.3	Comments	17
6.4	If statements	17
6.5	For-loop	19
6.6	While-loop	21
7	Ordinary differential equations	22
7.1	Solving differential equations numerically	22
7.2	Sets of differential equations	23
7.3	Example of numerical solution	24
8	Eigenvalues and eigenvectors	25
9	Sumulink	28
9.1	Constructing a Simulink model	29
9.2	Creating a block diagram	30
9.3	Example of a block diagram	30
9.4	Running a simulation	30

1 Introduction

MATLAB is a computer program designed for technical calculations. Its name is an abbreviation of “MATrix LABoratory”. In this program, matrix computations are implemented in a straightforward manner. However, many other pre-programmed routines are available. In addition, it is possible to implement user-defined calculation routines. MATLAB allows users to implement calculations in relatively short programming time. When these calculations have been performed, they can be visualized by means of several plot-routines.

1.1 Download and installation of MATLAB

You must be directly connected to the eduroam wireless network (at NTNU to be able to follow this installation procedure).

1. Map network drive: `//progdist.ntnu.no/progdist`. If you are having trouble mapping the network drive, please follow the instructions on Innsida.
2. At the network drive, go to: `/campus/Matlab/R2013b/Kilde/Windows_64-og-32-bit` to access the newest version of the MATLAB software.
3. Start setup.exe, and follow the instructions.
4. Read the lesmeg.txt in the “Matlab”-folder to get the right licenses and registration-codes.

1.2 References and Matlab tutorials

This quick-start course is based on the Interactive Matlab Course ([Martens et al., 2012](#)) and A Practical Introduction to Matlab ([Gockenbach, 1999](#)).

Here are some other good MATLAB tutorials:

	<i>URL</i>	<i>Comments</i>
Eindhoven University of Technology	http://www.imc.tue.nl/	The basis for this MATLAB course.
Michigan Tech	http://www.math.mtu.edu/~msgocken/intro/intro.html	Parts of this tutorial is used here.
MathWorks	http://www.mathworks.se/academia/student_center/tutorials/launchpad.html	Many tutorials available including interactive and video tutorials.

Table 1: Useful Matlab links

2 MATLAB basics

MATLAB is a computer program designed for technical calculations. Its name is an abbreviation of “MATrix LABoratory”. In this program, matrix computations are implemented in a straightforward manner. However, many other pre-programmed routines are available. In addition, it is possible to implement user-defined calculation routines. MATLAB allows users to implement calculations in relatively short programming time. When these calculations have been performed, they can be visualized by means of several plot-routines.

MATLAB can be extended with so called toolboxes. In general, these are collections of MATLAB functions that are tailored for special classes of problems. MATLAB performs calculations with the aid of matrices. These are rectangular schemes of numbers, which are also called arrays.

2.1 The MATLAB workstation

A typical layout of the MATLAB workstation can be seen in Figure 1. The main window in MATLAB is the command window, positioned at the center, in which commands can be typed after the MATLAB prompt:

>>

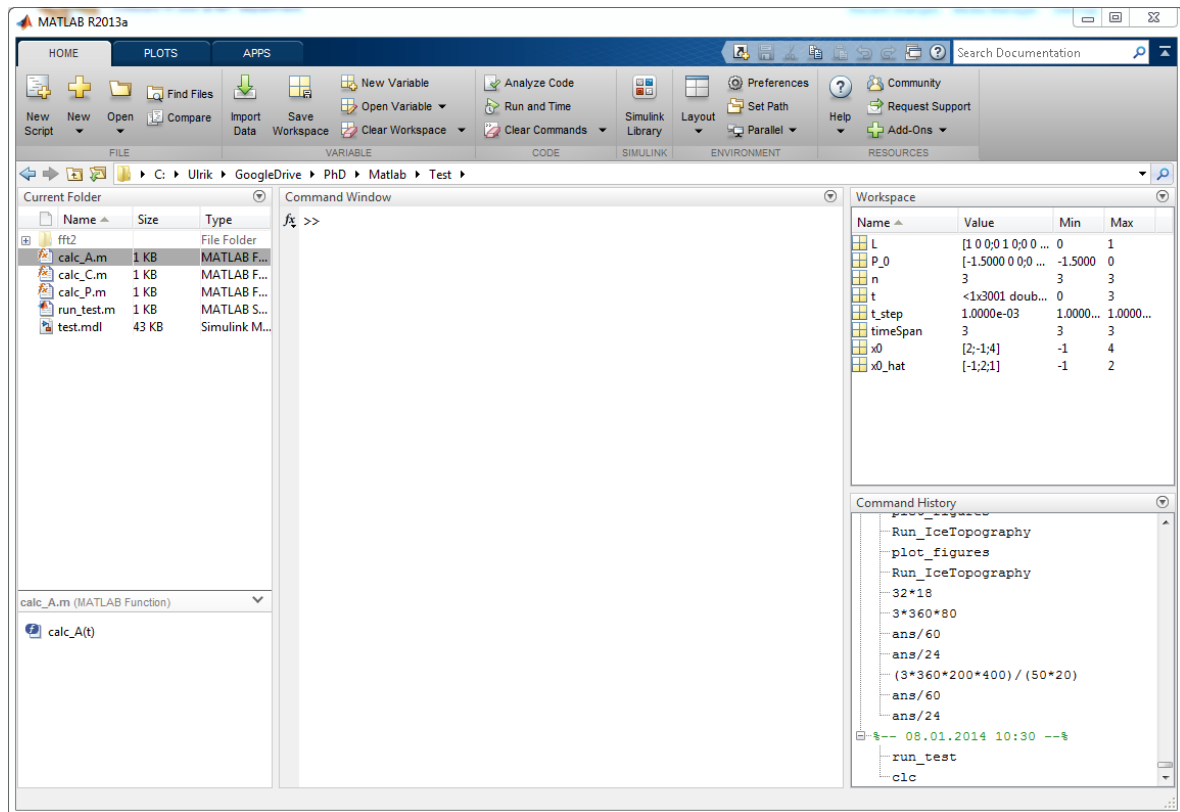


Figure 1: The MATAALB workstation.

Commands are entered with the return key. The command is then executed by MATLAB. If MATLAB is ready, after possible outputs a new prompt appears.

The window on the left is the “Current directory” window. The Current directory window depicts the files in the current directory. This is the first user-governed directory where MATLAB will look for files or functions. On the top right you will find the “Workspace” window, in which all declared variables are shown. The bottom right window is the “Command history”, where you can find all commands you have (recently) entered. By double-clicking on such a command, MATLAB reuses the command.

2.1.1 Interruption

A (long) MATLAB calculation can be interrupted with *Ctrl-c*. After this, a new prompt appears and new commands can be entered. This interruption is especially useful when a programmed loop is running that is corrupt.

2.1.2 Variables

The name of a variable in MATLAB has to start with a letter. After that, the name can consist of an arbitrary number of letters, numbers or symbols like “_” and “-”. MATLAB does distinguish between upper and lower case letters.

Some other important commands for the management of variables are given in Table 2.

<i>Command</i>	<i>Explanation</i>
<code>who</code>	Gives a list of the variables in use.
<code>whos</code>	Gives a list of the variables in use as well as some extra information.
<code>clear</code>	Removes all variables.
<code>clear x y</code>	Removes the variables <i>x</i> and <i>y</i> .

Table 2: MATLAB commands

2.2 Getting help

In principle all information about MATLAB can be found in MATLAB’s help facilities. Since the MATLAB commands and functions can often be used in more general situations, the information you will obtain through the help facilities will often be more general than necessary.

Below we list the most important ways to use the MATLAB help facilities.

When you know the name of a MATLAB command you want to use, e.g. the function `plot`, the fastest way to obtain information is to type `>> help 'function name'`, in this case `>> help plot`. This will show you some help in the command window. An alternative is to use the command `doc plot`, that will open the help browser containing a more extensive explanation, sometimes containing examples.

If you do not know the name of a function or do not know whether such a function exists, using the command `>> lookfor 'function'` or the Help Browser would be appropriate.

Help Functions	By typing the command <code>>> help 'function name'</code> in the Command Window, an M-help file appears in the Command Window. This file contains a short description of the function, the syntax, and other closely related help functions. This is MATLAB's most direct help facility, and you will often use this facility to quickly check how a certain function works and which notation should be used. If more extensive results are needed, try the command <code>doc</code> .
Look for	By typing the command <code>>> lookfor 'topic'</code> in the Command Window, you can call up all help possibilities which contain the specific search word. You can use this possibility to look for a function of which you do not remember the function name exactly. In this manner, all related topics appear in the Command Window.

2.3 Saving and loading data

Very often, results of MATLAB calculations need to be reused elsewhere. The values of variables can be saved in a `.mat` file. This can be done with the command

```
>> save filename
```

A saved `.mat` file can be read into MATLAB with the command

```
>> load filename
```

where the extension `.mat` can be omitted. The saved variables and their values are stored in the workspace and can now be used again.

Besides loading data stored in `.mat` files, MATLAB is useful for processing data which is obtained from external sources, e.g., experimental measurements. Typically this data is available as a plain text file organized into columns. MATLAB can easily handle tab or space-delimited text.

There is more than one way to read data into MATLAB from an external data file. The simplest, though least flexible, procedure is to use again the load command to read the entire content of the file in a single step. The load command requires that the data in the file is organized into a rectangular array. No column titles are permitted. One useful form of the load command is:

```
>> load measurements.txt
```

where `measurements.txt` is the name of the file containing the data. The result of this operation is that the data in `measurements.txt` is stored in a variable called 'measurements'. Files with various extension can be used. Note, that any extension except `.mat` indicates to MATLAB that the data is stored as plain ASCII text. A `.mat` extension is reserved for a file which has stored MATLAB variables.

Suppose you had a simple ASCII file named `meas_xy.txt` that contained two columns of numbers. The following MATLAB statements will load this data into the variable `meas_xy`, and then copy it into two vectors, `x` and `y`.

```
>> load meas_xy.txt; % read data into the meas_xy variable
```

```
>> x = meas_xy(:,1); % copy first column of meas_xy into x
>> y = meas_xy(:,2); % and second column into y
```

After applying these commands, the data stored in column 1 of the text file is stored in the MATLAB variable `x`, and the data stored in column 2 of the text file is stored in the MATLAB variable `y`. You are now able to process this data with MATLAB.

Another option to import data in MATLAB is choose the Home-tab and press the Import data-button. In the window opening, you can select your data file. Selecting next in this window, various import options are accessible. For example, you can choose whether MATLAB should split columns between spaces, or between commas. Furthermore, you can choose the number of header rows. On the righthand side of your window, you will see a preview how MATLAB will import the data. If you are satisfied, select finish.

3 Use MATLAB as a calculator

The MATLAB “Command Window” can be used as an advanced calculator and to test parts of your code.

By clicking on the Command Window, after the MATLAB prompt `>>`, a command can be typed. After having pushed the return key, the command is executed, and possibly the result appears on the screen. A command and its result looks like this:

```
>> a = 1 + 2 + 3
a =
    6
```

The result of $1+2+3$ is assigned to the variable `a`. You can now use this variable in successive operations, such as:

```
>> b = 2*a + a/3
b =
   14
```

In MATLAB, variables are introduced by assigning a value. Note, that the value can be numerical values, matrices (called arrays), or other types. You can use the variable, and later possibly assign a new value to the variable. It is not possible to perform calculations with variables that have not been assigned a value. A command does not need to start with an assignment of the form `variable =`. In such a case, the result is automatically assigned to the variable `ans`. You can then use `ans` further.

```
>> 4*a + 1
ans =
    25
>> ans*ans
ans =
   625
```

If you now want to know what the value of `a` is, the following command suffices:

```
>> a
a =
    6
```

Normally, MATLAB displays the output below the command. The output is suppressed by ending the command with a semicolon. After having entered the command `s=1+2`, the screen looks as follows

```
>> s = 1 + 2;
```

The value 3 has been assigned to the variable `s`. If you want to know the value of this variable, you can either look in the “Workspace” window on the top left of your MATLAB window, or request the value of `s` as follows

```
>> s
s =
    3
```

If a command is longer than one line, you can end the line with three dots, and then press the return key. You can then continue on the following line. After completing the command, the screen looks as follows:

```
>> s = 1 + 2 + ...
3 + 4
s =
    10
```

4 Matrix calculation

MATLAB uses matrices (or arrays) as the basic calculation unit. An array is a rectangular scheme of numbers, called elements, arranged in m rows and n columns. The simplest array contains only one column or one row.

4.1 Creating Matrices

An array with one row is also called a row or a row vector, and an array with one column is also called a column or a column vector. If the difference between consecutive elements in a row is always the same, the row can be formed with the command:

```
‘variable’ = ‘initial value’:‘step size’:‘final value’
```

Here ‘initial value’ is the first element of the row, ‘final value’ is the last element of the row, and ‘step size’ is the difference between the consecutive elements. For example:

```
>> x = 1:-0.2:0
```

gives the row

```
x =
    1.0000    0.8000    0.6000    0.4000    0.2000    0
```


If the difference between consecutive elements of the row is +1 , ‘step size’ can be omitted. For example,

```
>> x = 1:5
```

gives the row

```
x =  
    1    2    3    4    5
```

For arrays with m rows and n columns, the numbers or elements a_{ij} are sorted in different rows and columns:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}. \quad (1)$$

Such an array can be entered in different ways:

Between ‘[’ and ‘]’ the elements are separated by spaces or commas, and the rows are separated by a semicolon. The commands are:

```
>> a = [1 2 3;4 5 6;7 8 9]
```

or

```
>> a = [1,2,3;4,5,6;7,8,9]
```

or

```
>> a = [1 2 3  
4 5 6  
7 8 9]
```

In each of the cases above, the result is

```
a =  
    1    2    3  
    4    5    6  
    7    8    9
```

4.2 Indexing matrices

Elements of arrays can be indicated by means of their index. In the case of row and column vectors, one index suffices. This can be used according to Table 3.

Arrays (both matrices and vectors) can be concatenated into new arrays. If the arrays are arranged in a row, the number of rows in the arrays have to be equal. If the arrays are arranged in column, the number of columns in the arrays have to be equal. The concatenation of the arrays $\mathbf{a}=[2\ 3\ 4]$ and $\mathbf{b}=[6\ 5]$ proceeds as follows:

<i>Command</i>	<i>Result</i>
<code>a(1,2)</code>	Gives the element on the first row and second column of a .
<code>a(1,:)</code>	Gives the first row of a .
<code>a(:,3)</code>	Gives the third column of a .
<code>a(:,2)=[10;10;10]</code>	Changes the second column of a into a column with 10's only. Only works if a is a $n \times 3$ array, $n \geq 2$.
<code>b(1)</code>	The first element of b in the cases when b contains either one row or one column.
<code>a(1,[3,1])</code>	An array consisting of the 3rd and 1st elements of the first row of a .

Table 3: MATALB matrix indexes.

```
>> [a,b]
ans =
     2     3     4     6     5
```

4.3 Array operations

An ‘array operation’ is an operation (like addition or subtraction) that is applied to corresponding elements of two arrays of the same form. When an operation is applied to all elements of one array, one also speaks of an ‘array operation’.

The operations addition and subtraction are automatically interpreted as array operations. After the assignments

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

addition of the arrays results in

```
>> a+b
ans =
     7     7     7
```

and subtraction of the arrays results in

```
>> a-b
ans =
    -5    -3    -1
```

Note that the arrays a and b should be of the same size.

4.4 Elementwise operations

Array operations, such as multiplication ‘*’, division ‘/’ and raising to a power ‘^’, are indicated by putting a dot ‘.’ in front of the operation sign. In these array operations, the

operation is performed elementwise, e.g., the (1,1)-element of `a.*b`, yields `a(1,1)*b(1,1)` when `a` and `b` are arrays of appropriate sizes. Hence

```
>> a.*b
ans =
     6    10    12
>> a./b
ans =
    0.1667    0.4000    0.7500
>> a.^b
ans =
     1    32    81
```

For all these operations, the following conventions hold:

- If in an array operation one of the arrays is a number, this number is interpreted as an array of which each element equals this number, and of which the size is equal to the size of the other array.
- If each element of an array is to be multiplied by a scalar, one does not need to put a dot in front of the multiplication sign.

Examples:

```
>> 2.^b
ans =
    64    32    16
>> [2 2 2].^b
ans =
    64    32    16
>> a./2
ans =
    0.5000    1.0000    1.5000
>> 3*a
ans =
     3     6     9
```

Detailed information about these operations can be obtained with the command `help arith`.

4.5 Array of function values

Consider the function $f(x) = \frac{x^3}{1+x^2}$. We want to assign the row vector `[f(0), f(0.2), ..., f(1.8), f(2)]` to the variable `y`. First we make an array with the arguments `[0, 0.2, ..., 1.8, 2]`.

```
>> x = 0:0.2:2
x =
Columns 1 through 7
     0    0.2000    0.4000    0.6000    0.8000    1.0000    1.2000
```

```

Columns 8 through 11
1.4000  1.6000  1.8000  2.0000

```

For making the row of function values, we use array operations. Here, using dots is necessary, since the function $f(x)$ should be evaluated per element of x .

```

>> y = x.^3./(1+x.^2)
y =
Columns 1 through 7
0  0.0077  0.0552  0.1588  0.3122  0.5000  0.7082
Columns 8 through 11
0.9270  1.1506  1.3755  1.6000

```

4.6 Transposition

In an array (matrix), the rows and columns can be interchanged. This is called transposition. The result is called the transposed array. In MATLAB, transposition can be performed with the function `transpose`, or simply the accent (`'`). For example

```

>> A = [1 2;3 4;5 6]
A =
     1     2
     3     4
     5     6
>> transpose(a)
or
>> A'
ans =
     1     3     5
     2     4     6

```

The commands `transpose(A)` and `A'` give the same result for arrays containing real valued elements, but not for matrices having complex elements.

5 Visualization

To draw a graph, you first need to open a graphic screen. This is done with the command:

```
>> figure
```

Graphs are drawn with the command `plot`. The command

```
>> plot(v)
```

where v is a row vector containing the numbers v_1 until v_n , generates a plot where the points $(1, v_1)$, $(2, v_2)$ until (n, v_n) are connected by straight lines. The command

```
>> plot(v,w)
```

where v and w are two row vectors with the same number of elements n , plots the points $(v_i, w_i), i \in \{1, \dots, n\}$, and connects them by straight lines.

5.1 Plotting several graphs

The command

```
>> plot(v,w,x,y)
```

plots the points (v_i, w_i) and (x_i, y_i) in the same figure.

To determine the appearance of your graph, after plotting a vector you can add the color and linestyle of the graph. For example, the command

```
>> plot(x,y,'r--')
```

will plot y versus x , with a red dashed line. Type `help plot` to see options for other lines, markers and colors.

Suppose that we want to draw the graph of the function $f(x) = \frac{\sin(x)+2x}{1+x^2}$ on the interval $[0, 2\pi]$ consisting of 100 linearly spaced points. Note that the smoothness of the graph is of course determined by the number of point. The graph can be plotted by using the following commands:

```
>> x = 0:2*pi/99:2*pi;  
>> y = (sin(x)+2*x)./(1+x.^2);  
>> figure  
>> plot(x,y)
```

When drawing a graph, the axes are chosen automatically. You can choose the lengths of the axes yourself with

```
>> axis([XMIN XMAX YMIN YMAX])
```

which takes care that the graph is drawn in the rectangle $XMIN \leq x \leq XMAX$ and $YMIN \leq y \leq YMAX$.

After each new plot command, the old plot will in general disappear. If you want to draw different plots with different plot commands in the same figure, you have to give the following command after the first plot command

```
>> hold on
```

MATLAB then holds the graphs. The command

```
>> hold off
```

restores the state in which every new plot command makes the previous plot disappear.

5.2 Graphical screen handling

The commands in Table 4 are also important for handling the graphical screen.

<i>Command</i>	<i>Result</i>
<code>shg</code>	Show current graph window.
<code>clf</code>	Clear the current figure.
<code>grid</code>	Add a grid to the figure.
<code>title('title_text')</code>	Gives the figure a title called 'title_text'.
<code>text(x,y,'string')</code>	puts the text 'string' at the point (x,y) of the last plot.
<code>xlabel('text')</code>	writes information along the x-axis.
<code>ylabel('text')</code>	writes information along the y-axis.
<code>legend('graph1','graph2',...)</code>	show a legend-window in your figure.

Table 4: Graph commands.

5.2.1 An illustrative example

To illustrate the above methods, we plot the functions $f_1(t) = 2\sin(2\pi t)$ and $f_2(t) = \cos(\pi t)$ on the interval $[0, 1]$ in one figure. The sine-function is marked with dashed yellow line, while the cosine-function is dot-dashed in black. Moreover, we give the figure a suitable title, and x - and y -labels. Finally, we add legends to the figure and specify the y -axis to be between -2.2 and 2.2 . The result is seen in Figure 2 and the code is seen below.

```
>> t = 0:0.05:1;
>> f_1 = 2*sin(2*pi*t);
>> f_2 = cos(pi*t);
>> figure
>> plot(t,f_1,'y--')
>> hold on
>> plot(t,f_2,'k.-')
>> hold off
>> xlabel('t')
>> ylabel('f(t)')
>> title('Sine and cosine functions')
>> legend('f_1(t)=2sin(2\pit)','f_2(t)=cos(\pit)')
>> axis([0 1 -2.2 2.2])
```

5.3 Other graph types

You can use other plot functions in MATLAB as well. The most important are shown in Figure 3 and listed in Table 5.

You can save a figure by selecting *File* \rightarrow *Save as* and select the required file directory, filename and file type. An other option is to use the `print` command. Type `help print` for additional information. When you want to be able to open and modify your figures again in MATLAB, you should save your figure as a `.fig` file.

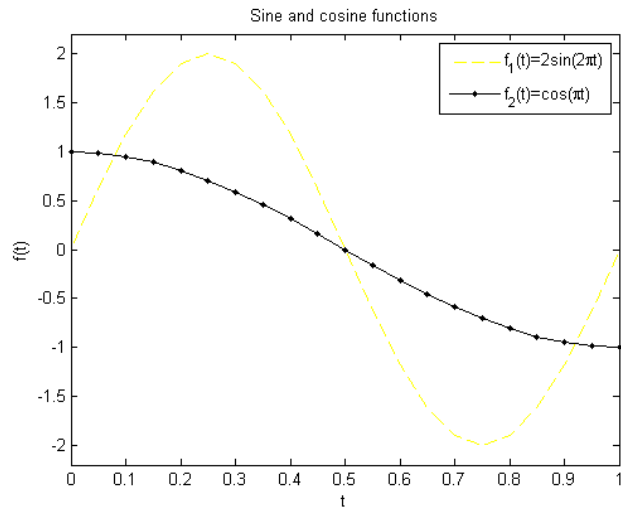


Figure 2: An illustrative example of a MATLAB plot.

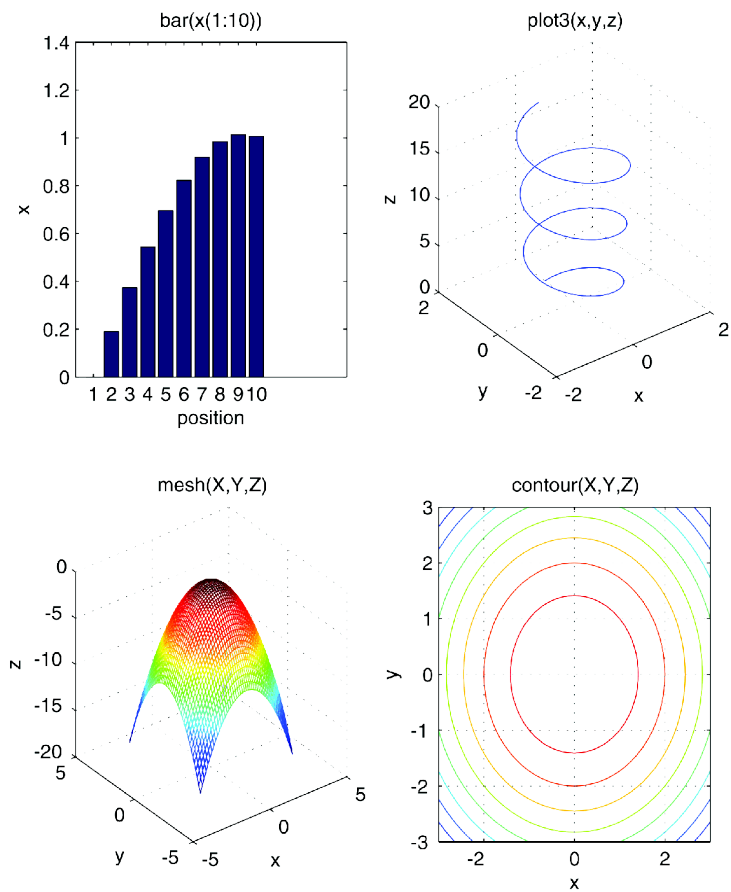


Figure 3: Examples of other graphs in MATLAB.

<i>Command</i>	<i>Result</i>
<code>bar(x)</code>	Produces a bar-plot of vector x .
<code>plot3(x,y,z)</code>	Produces a three-dimensional curve between the points, whose coordinates are given in the vectors x , y , and z .
<code>mesh(X,Y,Z)</code>	Produces a three-dimensional surface between the grid points, whose coordinates are given in the matrices X , Y , and Z .
<code>contour(X,Y,Z)</code>	Produces level curves in (x,y)-plane, such that all points on a line have the same value for z . Since the inputs in this functions are gridpoints, the inputs X , Y , and Z should be matrices.

Table 5: Other MATLAB graphs.

6 Programming in MATLAB

6.1 Script m-Files

If you have to execute a series of commands repeatedly, it might be better to write the commands in a so called *script file*. The file should have a name with the extension `.m`. As an example, let us consider the drawing of the function $f(x) = |x \sin(2\pi x)|$ on the interval $[0, 2]$ by using a script file: Under the 'Home'-tab, press the New script-button and the MATLAB Editor/Debugger is opened. Type the following lines:

```
x = 0:0.1:2;
y = abs(x.*sin(2*pi*x));
plot(x,y)
title('f(x)= |x sin(2 pi x)| on the interval [0,2]')
xlabel('x')
ylabel('f(x)')
axis([0 2 0 2])
```

Save the file by choosing the 'Editor'-tab and press the Save-button. Save it with the name *picture.m*. If you now give the command

```
>> picture
```

in the MATLAB prompt, then the graph of $f(x)$ on the interval $[0, 2]$ will appear. An other option to save and run a file written in your editor window is to press the 'play'-button when in the 'Editor'-tab, or simply pressing the short-key (F5). Be careful, that this will overwrite a possible previous file.

Note that it is better in general to save the commands that generate a figure than to save the figure itself.

6.2 User defined functions

6.2.1 Function m-files

In MATLAB you can also define your own functions. MATLAB assumes by default that all functions act on arrays. Therefore, you must keep in mind the rules for array operations when writing your own functions. You can then combine your own functions with MATLAB functions. A function definition has to be saved in a *function file*, which is a file with the extension *.m*. The name of the file has to be the same as the name of the function, and should have the extension *.m*.

Consider the function $f(x) = x^2 + e^x$. In MATLAB we will write a function with the same name. The definition of the function has to be saved in the file *f.m*. Choose the Home-tab and press the *New* → *Function* and the 'MATLAB Editor/Debugger' is opened. If you now type the lines:

```
function y = f(x)
y = x.^2 + exp(x);
```

and you save the file under the name *f.m*, then within MATLAB the function $f(x)$ is available.

- The first line of the file has to contain the word 'function'.
- The variables used are local; they will not be available in your 'Workspace'.
- If x is an array, then y becomes an array of function values.
- The semicolon at the end prevents that at every function evaluation unnecessary output appears on your screen.
- The name of the function, i.e., f , must be the same as the file name, i.e., *f.m*.

You can change the definition of $f(x)$ by typing the command `edit f` in the command prompt.

Warning: m-files should not be given the name of existing variables or MATLAB functions. Conversely, after you have defined a function yourself, you should not give variables the same name as your function, otherwise the function will not work any more.

6.2.2 Anonymous functions

Sometimes it is more convenient to define your function at the command line. MATLAB functions produced at the command line are called anonymous functions. For example, consider again the function $f(x) = x^2 + e^x$. We can create the anonymous function as follows:

```
>> f = @(x)(x.^2+exp(x))
```

Here, `f` is the name of the function, `@` is the function handle, x is the input argument and `x.^2+exp(x)` is the function.

Anonymous functions can have multiple input arguments. The general structure of anonymous functions is

```
name = @(input arguments) (functions).
```

The anonymous function will be evaluated in the same way as the function m-files. Thus the value of `f(1.2)` will be obtained by the command

```
>> f(1.2)
ans =
    4.7601
```

6.3 Comments

The most important way to make a program understandable to others is the use of comments. In MATLAB, this can be done in three ways, all including the `%`-sign:

- Single line comment: Place the `%`-sign on a line in your program, and everything after the `%`-sign will not be executed. The shortcut for commenting out a line in MATLAB is `Ctrl+r` while `Ctrl+t` can be used to remove the comment.
- Cell mode: You can subdivide a MATLAB program into cells. These are similar to sections in a normal texts. To start a cell, one should type two `%`-signs followed by the 'cellheader', i.e., `%% 'cellheader'`, where 'cellheader' can be replaced with a caption. For example, one can divide a program in a cell 'Initialization', a cell 'Calculations' and a cell 'Output Processing'. When you are editing an m-file, you can run the active cell (i.e. the cell containing your cursor) by using `Ctrl+Enter`.
- Multiple line comment: To comment several lines, use the syntax:

```
%{
First commented line
Second commented line
...
Last commented line
%}
```

6.4 If statements

One of the most commonly used programming language constructs is the if-construct. With this construct it is possible to decide whether or not to execute certain program lines, based on a relational test of logical variables. The general form of this construct is:

```
if logical expression
    program lines
elseif logical expression
    program lines
else
    program lines
end
```

The `elseif` and `else` statements are optional, so they can also be omitted. A logical expression is either true or false. In MATLAB, the value 1 is given to a true expression, and the

value 0 is given to a false expression. When evaluating logical expressions, we use relational and logical operators given in Table 6.

<i>Relation</i>	<i>Explanation</i>
<	Less than
<=	Less or equal than
>	Greater than
>=	Greater or equal than
==	Equal to
~=	Not equal to
<i>Logical</i>	
&	and
	or
~	not

Table 6: MATLAB logicals.

Examples:

- $1 < 2$ is true,
- $1 == 1$ is true,
- $1 == 2$ is false,
- $1 \sim= 2$ is true.

Example 1. Write a MATLAB program that simulates the sign function. Use the MATLAB help to see what this function means.

```
% make a row with time steps
t = -10:0.1:10

% determine the number of time steps in array t with the command size
n = max(size(t))

% initialize the values of f at zero
f = zeros(1,n)

% determine the function values in a FOR loop
for k = 1:n
    if t(k) < 0
        f(k) = -1;
    elseif t(k) == 0
        f(k) = 0;
    elseif t(k) > 0
        f(k) = 1;
    end
end
end
```

```
% plot the function
plot(t,f)
```

Check that, due to the fact that we have initialized f at 0, this if-loop can be simplified.

Example 2. Using a for-loop and conditional tests to calculate the inner product of two vectors.

To this end, we write a new function m-file, i.e., we create our own MATLAB command to calculate the inner product. We call our new command `inprod`. To realize this, we write a new function m-file `inprod.m`. We will call the two vectors we want to multiply in this file a and b . We will call the output argument result.

```
function result = inprod(a,b)
% test whether the vectors a and b have the same length
%(otherwise it is not possible to determine the inner product.)
[ra ca] = size(a); % the command size gives the size of the matrix a by
assigning the number of rows of a to ra and the number of columns of a to ca.
[rb cb] = size(b);

if ca~=1 % Note, that '~' represents 'not'
    error('first argument is not a vector')
end
if cb~=1
    error('second argument is not a vector')
end
if ra~=rb
    error('vectors cannot be multiplied: they do not have the same length')
end
% initialization of result (a number)
result = 0;
for p = 1:ra
    result = result+a(p)*b(p);
end
```

6.5 For-loop

MATLAB knows two repeating constructs. The first is the for-loop, and the second one is the while-loop. The for-loop construct offers you the possibility to execute certain pieces of a program several times, while the value of certain variables is changing. A so-called loop variable keeps track of how often the program lines have to be repeated.

The general form of a for-loop is:

```
for loopvariable = start:step:end
    command lines
end
```

When step is omitted, a step size of 1 is used by default.

Example 3. A simple for-loop

```
% definition of constant n
n = 10;
% producing an array with n rows
% and 1 column, filled with zeroes
A = zeros(n,1);
% fill this column with values from 1 up to
% n in a FOR loop
for k = 1:n
    A(k,1) = k;
end
A % print the result on the screen.
```

This loop works as follows: after the for command, the loop variable k is given the value 1, and the program lines until the end are executed. Thus, $A(1,1)$ is given the value 1. After the end command, MATLAB jumps back to the for line, and the loop variable k is given the value 2. After this, the program lines are executed again: $A(2,1)$ is given the value 2. This is repeated until the last step, where the loop variable k is given the value 10. After this, the program continues with the lines after end, where A is printed on the screen.

for $k = 1:n$ means ‘for k is 1 up to n with steps of 1’. You can also use larger steps to go from 1 to n , or start with another value: for example, for $k = 2:2:10$ means ‘for k is 2 up to 10 with steps of 2’ (i.e., 2,4,6,8,10). Of course you are free to choose other loop variables like, for instance, a and/or b .

Example 4.

```
% initialization of the variable p
p = 0;
% run loop n times
for k = 1:n
    p = p+1; % increase counter variable p
end
p
```

In this loop the value of p is raised by one every time the program goes through the loop. When the program goes through the loop for the first time, the new value of p becomes equal to the old value of p (which was 0) plus 1, so p becomes 1. So this is an assignment statement and not an equation. This kind of assignment construction is convenient to use to keep track of a counter or a summation within a loop construction.

It is also possible to nest one or more for loops. By using this, you can easily fill an $n \times m$ matrix by going through the matrix element by element:

Example 5. Give each element of a matrix a random value between 0 and 1

```
% definition of the size of the matrix
nr = 4; % number of rows
nc = 3; % number of columns
% producing an n x m matrix with nr rows and nc columns, which is filled
```

with zeroes. It is not really necessary to make the matrix beforehand and fill it with zeroes, but it is neater, it prevents errors, and sometimes it may be useful. Especially for large matrices, as it is much faster.

```
A = zeros(nr,nc);
% fill the matrix with random values between 0 and 1 in a nested FOR loop
for r = 1:nr
    for c = 1:nc
        A(r,c) = rand(1);
    end
end
A
```

6.6 While-loop

The second repeating construct in MATLAB is the while-loop. With the for-loop you have to specify beforehand how many times you want to execute the loop. With the while-loop this number of executions is not specified beforehand, but rather depends on whether or not a certain expression is true or false. The general form of a while-loop is:

```
while logical expressions
    command lines
end
```

Example 6. A ‘fair’ lottery. Here we use the while-loop. Write a script m-file with the name `lottery.m` and the following contents:

```
% let MATLAB determine a random value between 1 and 10
n = round(rand(1)*10+0.5);
% initialization of stopping variable
stop = 0;
while stop == 0
    choice = input('Enter an integer between 1 and 10')
    if choice == n
        stop = 1;
        disp('*** ***)
        disp('You have won!!!!')
        disp('*** ***)
    else
        disp('Sorry, incorrect: next player please')
    end
end
end
```

7 Ordinary differential equations

In this section we will solve ordinary differential equation (ODE) such as

$$\dot{x} = f(x). \quad (2)$$

When the analytic solution is not available one has to use numerical solution techniques to approximate a solution.

The following section explains the use of numerical solvers, and how to interpret the results.

7.1 Solving differential equations numerically

For the numerical calculation (or rather, approximation) of the solution of a differential equation, one uses algorithms that are related to the Euler algorithm. Briefly, the Euler algorithm boils down to the following. Even though one does not know the solution of the differential equation explicitly, one does know the derivative of the solution in every point (this is given by the right hand side of the differential equation), or, in other words, one knows what the direction field of the differential equation is. This direction field is now used to approximate the solution. From a given initial value, the next point of the solution is calculated by moving from the initial value in the direction of the direction field. From the point obtained in this way, one does the same again, etc. So the quality of the approximation of the solution depends on the initial value, the direction field, and the step size.

In MATLAB, differential equations can be solved numerically with the commands such as `ode45`, `ode23` or `ode15s`. The underlying algorithms of `ode45` and `ode23` make use of Runge-Kutta-Fehlberg integration with variable step size, i.e., the algorithm increases the step size when the solution varies less. `ode23` uses the second and third order formulas, while `ode45` uses the fourth and fifth order formulas. The routine `ode15s` uses another integration routine. This routine is specialized in problems, that the previous routines have problems to deal with: so-called stiff differential equations.

The mentioned routines can be applied to sets of first order differential equations of the form:

$$\frac{dy}{dt} = f(t, y). \quad (3)$$

Here y is the state vector and f the vector valued function that gives the time-derivative of the state vector as a function of t and y .

Throughout this section, extensive use is made of function-files to facilitate numerical solving of differential equation. For an explanation of these, see “User defined functions” in Section 6.2.

The form in which `ode23` is used, is:

```
>> [t,y] = ode23('filename',[t0,t1],y0)
```

Here 'filename' is the name of the `.m` file in which the differential equation is defined, $[t_0, t_1]$ the time-interval over which the differential equation is solved, and y_0 the vector with initial values. The algorithm in the program `ode23` first determines in a point the derivative $\frac{dy}{dt}$ of the

function defined in the function file 'filename'. With this, the next point is determined, etc. The algorithm itself determines with which time steps it goes through the interval $[t_0, t_1]$. Since `ode45` works with higher order formulas, less integration steps are needed with this command. As a consequence of this, `ode23` in general gives less smooth figures than `ode45`. The output of `ode` gives a vector t with the time instances at which the solution x has been calculated, and a row vector y with associated solution values.

7.1.1 Choice of solver

For most differential equations, both `ode23` and `ode45` are suited. The difference between `ode23` and `ode45` is rather subtle: with the same accuracy, `ode23` will need more time steps, though each time step is calculated faster.

However, for a certain class of differential equations, the stiff differential equations, the previously mentioned solvers will not be able to find an accurate solution, or may need excessive computation times for taking very small time steps. In that case, the problem might be of a class called "Stiff differential equations". For these differential equations, the `ode15s` is a better choice.

7.2 Sets of differential equations

The commands `ode23` and `ode45` can also be applied for sets of differential equations. For example, consider the set of differential equations

$$\dot{x}_1 = 5x_1 - 2x_2 \quad (4)$$

$$\dot{x}_2 = 7x_1 - 4x_2 \quad (5)$$

The initial values $x_1(0) = 2$ and $x_2(0) = 8$ define exactly one solution.

With the commands `ode23` and `ode45` this type of differential equations can be tackled. To start, we write the function file `deq.m` in the "MATLAB Editor/Debugger", with the contents:

```
function xdot = deq(t,x)
xdot = [5*x(1)-2*x(2);7*x(1)-4*x(2)];
```

Note, that `xdot` has to be a column vector. The command:

```
>> [t,x] = ode23('deq', [0,1], [2,8])
```

now calculates on the time interval $[0, 1]$ the values of $x_1(t)$ and $x_2(t)$ with initial conditions $x_1(0) = 2$ and $x_2(0) = 8$. The variable `x` becomes an array with two columns, where the first column contains the values of $x_1(t)$ and the second column contains the values of $x_2(t)$. The variable `t` is a vector consisting of all time-instances where the solution is calculated. You can plot the solutions with the command:

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

Since `x` consists of two columns, two graphs are drawn in the same figure. If you only enter the command:


```
>> ode23('deq',[0,1],[2,8])
```

the graphs of $x_1(t)$ and $x_2(t)$ are drawn, since by default the “OutputFcn” `odeplot` is used. Note that you could also have written the file `deq.m` in the following way:

```
function xdot = deq(t,x)
A = [5,-2;7,-4];
xdot = A*x;
```

7.3 Example of numerical solution

The Duffing equation is a model for mechanical oscillations with a nonlinear spring. The dynamics of this system, at a certain parameter is described by:

$$\ddot{x} + 0.1\dot{x} - x + x^3 = 0. \quad (6)$$

To use MATLAB to compute the direction field, trajectories and solutions of this system, we rewrite this system in the state space form $\dot{q} = f(t, q)$, where $q = [q_1, q_2]^\top = [x, \dot{x}]^\top$. The nonlinear function $f(t, q)$ becomes

$$\dot{q} = f(t, q) = \begin{bmatrix} q_2 \\ -0.1q_2 + q_1 - q_1^3 \end{bmatrix}. \quad (7)$$

This equation is implemented in the MATLAB routine `duffing.m`, containing

```
function qdot=duffing(t,q);
qdot=[q(2); -0.1*q(2)+q(1)-q(1)^3];
```

In the following script file, the above mentioned function file is used, to plot first the direction field, afterwards the trajectory over time, and finally the solution curves in phase plane, together with the direction field. Since $f(t, q)$ is not explicitly dependent on t , the script uses `duffing(0,q)` to compute the direction field, this is valid at all times.

```
%% script file to analyse duffing equation.
clear all;
close all;

%% create grid for direction plot
[Q1,Q2]=meshgrid(-1.5:.3:1.5,-0.7:.14:0.7);

%% create matrices with elements of vector Qd
Qd1=zeros(size(Q1));
Qd2=zeros(size(Q2));
for i=1:11;
    for j=1:11;
        Qdot=duffing(0,[Q1(i,j);Q2(i,j)]);
        Qd1(i,j)=Qdot(1);
        Qd2(i,j)=Qdot(2);
```

```

        end
end

%% plot direction plot
figure(1);
quiver(Q1,Q2,Qd1,Qd2);
xlabel('q_1');
ylabel('q_2')
title('Direction field of Duffing equation');

%% create time vector and numerical solution with ode45
[t, Q]=ode45('duffing',[0,20],[1.5;0]);

%% plot trajectory in time
figure(2)
plot(t,Q(:,1))
xlabel('t');
ylabel('q_1');

figure(3)
plot(t,Q(:,2))
xlabel('t');
ylabel('q_2');

%% plot trajectory in phase space
figure(4);
quiver(Q1,Q2,Qd1,Qd2);
xlabel('q_1=x');
ylabel('q_2=dotx')
title('Solution curve in phase plane');
hold on;
plot(Q(:,1),Q(:,2),'k');
hold off;

```

Running these files yields the plots in Figure 4–6.

8 Eigenvalues and eigenvectors

In addition to solving linear systems (with the backslash operator), Matlab performs many other matrix computations. Among the most useful is the computation of eigenvalues and eigenvectors with the `eig` command. If A is a square matrix, then

```
ev = eig(A)
```

returns the eigenvalues of A in a vector, while

```
[V,D] = eig(A)
```

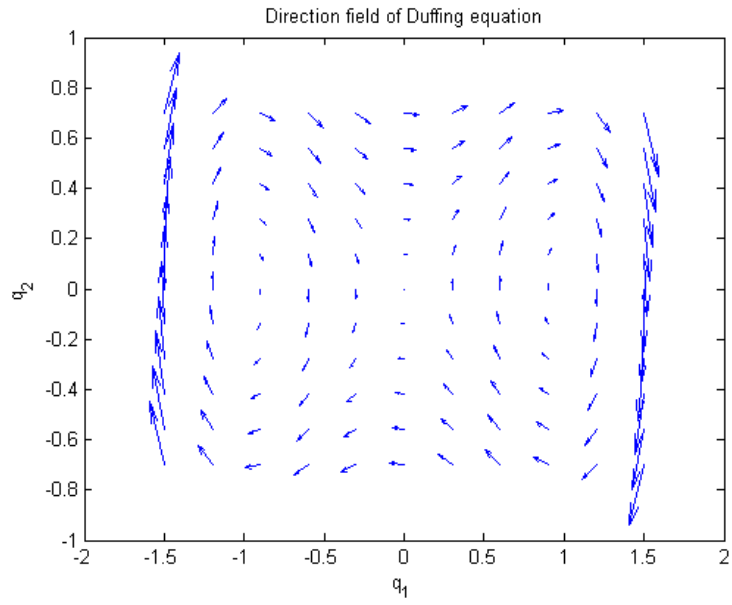


Figure 4: Direction field of Duffing equation.

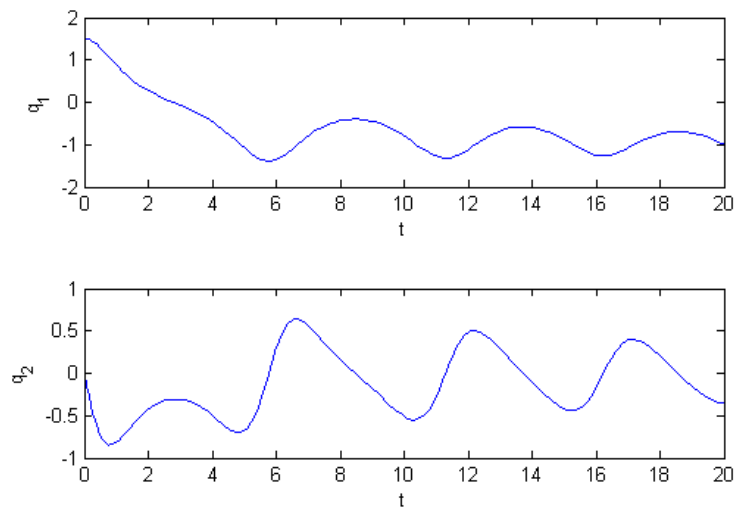


Figure 5: Time trajectory of Duffing equation.

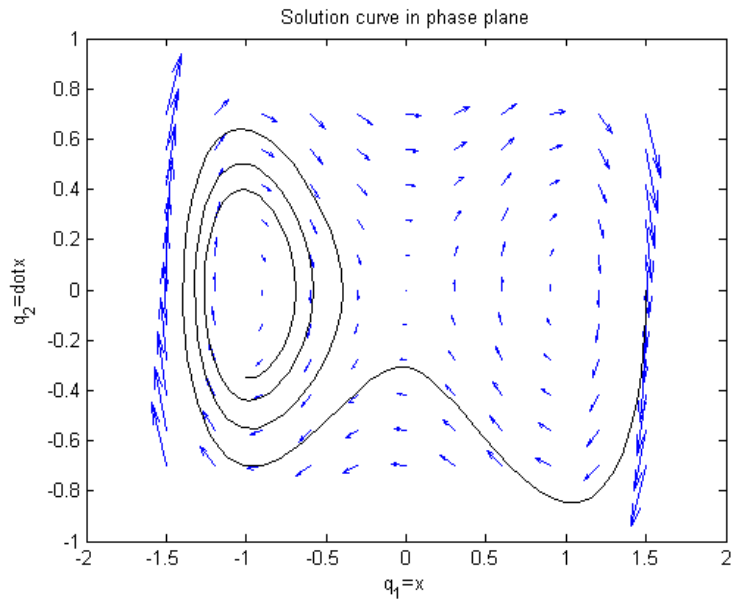


Figure 6: Trajectory of Duffing equation in Phase plane.

returns the spectral decomposition of A : V is a matrix whose columns are eigenvectors of A , while D is a diagonal matrix whose diagonal entries are eigenvalues. The equation $AV = VD$ holds. If A is diagonalizable, then V is invertible, while if A is symmetric, then V is orthogonal.

Example 7.

```
>> A = [1 3 2;4 5 6;7 8 9]
```

A =

```
1 3 2
4 5 6
7 8 9
```

```
>> eig(A)
```

ans =

```
15.9743 + 0.0000i
-0.4871 + 0.5711i
-0.4871 - 0.5711i
```

```
>> [V,D] = eig(A)
```

V =

```
-0.2155 0.0683 + 0.7215i 0.0683 - 0.7215i
-0.5277 -0.3613 - 0.0027i -0.3613 + 0.0027i
-0.8216 0.2851 - 0.5129i 0.2851 + 0.5129i
```

D =

```
15.9743 + 0.0000i 0.0000 + 0.0000i 0.0000 + 0.0000i
0.0000 + 0.0000i -0.4871 + 0.5711i 0.0000 + 0.0000i
0.0000 + 0.0000i 0.0000 + 0.0000i -0.4871 - 0.5711i
```

```
>> A*V-V*D
```

ans =

```

1.0e-14 *
-0.0888 + 0.0000i   0.0777 - 0.1998i   0.0777 + 0.1998i
0.0000 + 0.0000i   -0.0583 + 0.0666i   -0.0583 - 0.0666i
0.0000 + 0.0000i   -0.0555 + 0.2387i   -0.0555 - 0.2387i

```

There are many other matrix functions in Matlab, many of them related to matrix factorizations. Some of the most useful are:

- `lu` computes the LU factorization of a matrix;
- `chol` computes the Cholesky factorization of a symmetric positive definite matrix;
- `qr` computes the QR factorization of a matrix;
- `vd` computes the singular values or singular value decomposition of a matrix;
- `cond`, `condest`, `rcond` computes or estimates various condition numbers;
- `norm` computes various matrix or vector norms.

9 Sumulink

Simulink is a toolbox of MATLAB that can be used for modeling, analyzing and simulating dynamical systems. Here, we focus on the use of Simulink on the use with ordinary differential equations.

Simulating the dynamical behavior of a system is important in engineering. Many systems can be written as a differential equation, such as:

$$m\ddot{x}(t) + b\dot{x}(t) + kx(t) = u(t), \quad (8)$$


describing the movement of a single mass, on which an actuation force u is applied. For some simple differential equations, trajectories $x(t)$ of the system can be computed analytically. Many of these analytical solutions have been implemented in MATLAB and can be obtained with the command `dsolve`.

However, many systems found in engineering or physics are described by more complex differential equations, for which no analytical solution is known or even exists. Trajectories of these systems can be approximated numerically. Hereto, solvers are developed that approximate the change of the state variables over a small time step. Herewith, step by step, an approximation of the trajectory is found. For example, the MATLAB functions `ode23` and `ode45` are numerical solvers. Simulink is an other tool in MATLAB using numerical solvers.

Use of simulink has some advantages over the use of the functions `ode23` and `ode45`. Simulink has a graphical interface, such that the structure of the system can be clearly visible. Inputs, such as a discontinuous signal $u(t)$ for system (8) can be easily applied. Furthermore, real time applications are possible.

9.1 Constructing a Simulink model

To compute trajectories with MATLAB by means of block diagrams, the tool Simulink should be used. To open Simulink, type `simulink` in your command window or press the icon when in the Home-tab in the MATLAB window. Now, the Simulink Library Browser should appear.

A new file can be opened by selecting *File* → *New* → *Model* or selecting the icon  on the toolbar. In the window appearing, one should build the model. Basically, you will have to draw the block diagram. Hereto, blocks from the Library Browser should be dragged into the model while holding your left mouse button.

In the library browser, you will find the integrator block in the category Continuous, the gain and sum block are part of the category Math Operations. When a block is inserted in the model, one can flip the block (press the right mouse button on block, select *Format* → *Flip*). To connect two blocks, click on the `>>` exiting the first block, hold your right-mouse-button and drag a line towards the `>>` entering the second block. You can give a name to a signal by double-clicking on the connection.

Blocks and arrows can be moved in your model by using either the arrows on your keyboard or by dragging them with your mouse.

Many blocks in Simulink have certain parameters. To edit these parameters, double click on the block and change the value. The particular gain used is a parameter of the gain block. The initial value of a signal exiting an integrator block is a parameter of this block. In the parameters of the sum-block, one can make the block subtract signals instead of add them, by changing the list of signs from `|++` to `|+-`.

The integrator block has its initial condition as parameter. Here, one specifies the initial output of the integrator block. By changing these initial conditions, a Simulink model can be used to compute trajectories from predefined initial conditions.

Inputs in Simulink have to be defined as a function of time, for example $u(t) = 3 \sin(2t)$. You will find many possible inputs in the Sources-category of your Library browser. Most used are the sine wave, step or constant blocks. A sine wave block has the properties amplitude, frequency and phase, the latter describing the initial phase of the signal. A step block has parameters initial time, initial value and final value. A constant source has the constant value as a parameter.

To handle outputs in Simulink, one needs to use the blocks in the category Sinks in the Library browser. Most used are the Scope, To File, and To Workspace blocks. A scope will visualize the signals entering it. When you have finished a simulation, double-click on the scope to see the results. The To File block saves the entering signal to a `.mat` file. When you select the block properties, you can change the file name, and the name of the stored variable. The block To Workspace saves the signals to the workspace, such that one can use them later in the command window. In this block's properties, one can change the variable name, and choose whether the data should be saved as structure, structure with time or array.

It is allowed to call variables that are present in your workspace in block parameters. For example, a gain-block with gain `k` will work properly, when a scalar `k` is available in the MATLAB workspace.

To save your model, press the save button in the toolbar of your model window. Simulink models will be saved with the extension *.mdl*. Later, you can open these models by selecting the file saved.

Please note, that the Library browser contains many more blocks as mentioned in this section. When you double click on a certain block in the library browser, you will see the purpose and parameters of the block.

9.2 Creating a block diagram

Most dynamical systems can be represented in a block diagram. A differential equation is splitted in simple blocks. Arrows connecting blocks are signals. The most used blocks are given in the following table:

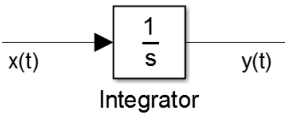
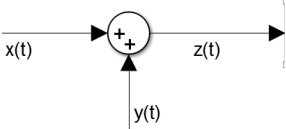
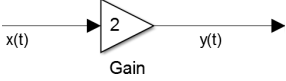
Integrator		$y(t) = \int_0^t x(s) ds$
Sum		$z(t) = x(t) + y(t)$
Gain		$y(t) = 2x(t)$

Table 7: Popular Simulink blocks

9.3 Example of a block diagram

To clarify, a block diagram is constructed for the system (8), i.e.

$$m\ddot{x}(t) + b\dot{x}(t) + kx(t) = u(t).$$

The highest state derivatives is \ddot{x} . Integrating this state once, we obtain \dot{x} . Integration of \dot{x} yields x . \ddot{x} is represented by ddx and \dot{x} by dx in Figure 7. Assume that $m = 1$, $b = 2$ and $k = 3$ and rewriting (8), yields an expression for \ddot{x} :

$$\ddot{x} = u - 2\dot{x} - 3x \quad (9)$$

and we get the Simulink model shown in Figure 7.

9.4 Running a simulation

To run a simulation of your Simulink model, you can select the button “Start Simulation” on the toolbar of the model. Take care, that the correct initial conditions are used for the

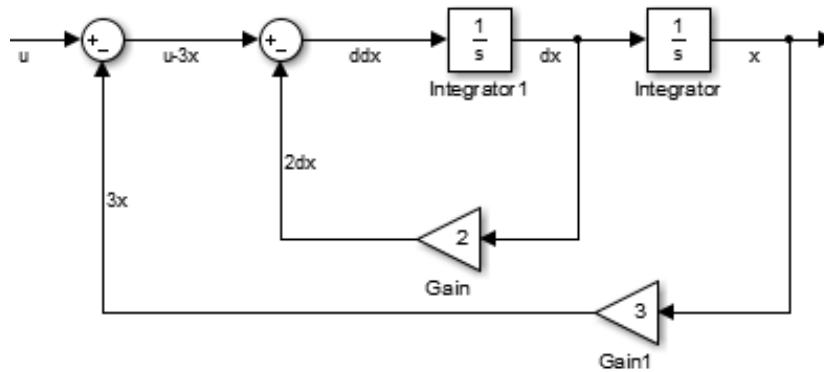


Figure 7: The Simulink block diagram.

integrator blocks. The simulation will finish when the maximum simulation time is reached. If this takes to long, you can terminate your simulation in the meantime by pressing the “Stop Simulation” button on the toolbar.

When the simulation is stopped, you can double-click on a scope to see the signal over time. Signals send to “To Workspace” or “To file” blocks are created in your workspace, or in the file in the current directory, respectively.

9.4.1 Simulation parameters

To edit the parameters used by Simulink to simulate the behavior of the system, click on *Simulation* → *Configuration Simulation Parameters*. A screen similar to the one in Figure 8 will be shown.

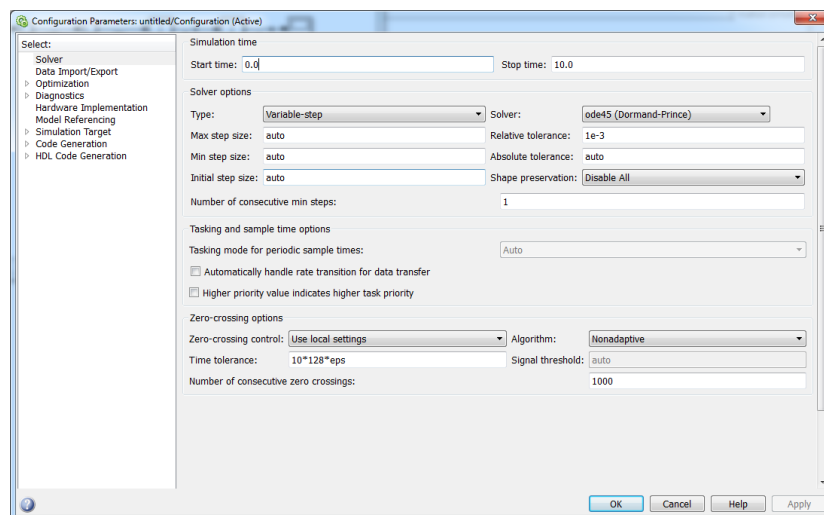


Figure 8: Simulink setup window

In this window, you will be able to set a start and stop time for your simulation. Note, that these times represent the simulated time, which usually differs from the computation time.

9.4.2 Type of solver

Many solver options can be changed. The most important one is the solver type. The solver type determines the time steps Simulink is taking. Though very small time steps are usually more accurate, they require longer computation time.

One can choose Variable-Step or Fixed-Step solvers. In a fixed step solver, the solver takes fixed time steps and computes at each time step, how the states should be updated. The most important option for this solver is the fixed-step size.

More complex solvers are variable-step solvers. They try to take their step size as large as possible, but will reduce the step size when the computation becomes less accurate. Usually, variable-step solvers are faster. By default, one should select the `ode45` solver. However, for certain problems, `ode45` will need very long computation times. In that case, one can choose a different solver, such as `ode15s`.

Important parameters that should be chosen for a variable step solver are the Relative Tolerance and the Absolute Tolerance. The relative tolerance measures the error relative to the size of each state. The relative tolerance represents a percentage of the state's value. The default, $1e-3$, means that the computed state will be accurate to within 0.1%. The Absolute Tolerance is a threshold error value. This tolerance represents the acceptable error as the value of the measured state approaches zero. A variable step solver chooses its time steps such, that both absolute and relative tolerance are satisfied.

References

- F. B. L. Martens, P. van Zutven, J. C. v. d. Meer, H. A. v. Essen, and J. J. B. Biemond, “Interactive matlab course,” Eindhoven University of Technology, Tech. Rep., 2012.
- M. S. Gockenbach, “A practical introduction to matlab,” Michigan Tech, Tech. Rep., 1999.