Lab Training on Simulation of Communication Systems Using Matlab®©

Dr.-Ing. Marc Selig

October 2018

*Script by Thomas Edlich, Ziyang Ju and Mario Westmeier, 2007.

Contents

1 Introduction

This lab course is focused on the usage of the simulation tool MATLAB®for occurring problems in wireless communications. The exercises of this lab course deal with the physical layer (PHY) and show basic implementations of different modulation schemes.

1.1 Workstation

The lab course is done in the PC pool of *ComLab* in room 2315. Each PC shall be utilized by not more than two students. Besides the official course, you will have the chance to do some training at these PCs (simply inform an employee during the official working hours). To get access at the login procedure, please use the login "matlab" and the password "comlab" (without quotations).

login: matlab
pw: comlab

Please avoid storing your files on the local hard disk. It is strongly recommended to save all files at the file server (\\141.51.146.10\matlab*Semester*), where *Semester* has to be replaced by the semester the course takes place, e.g. "WinterTerm2018" for the winter term of 2018. Please create a unique folder inside the semester folder to avoid mixing your files with the files of other participants of the course. Thus, your work and training possibilities do not depend on the access (or availability) of a certain PC. In addition to this, please feel free to make copies of your work at removable storage media (usb-stick, etc.).

use the file server

MATLAB®is also available at the ITS-PC pools. See http://www.uni-kassel.de/its for further information. There is a student version offered at the www.mathworks.com (˜$99), thus you can use MATLAB®at your own computer.

1.2 Moodle Server

The training in the course goes along with an e-learning course at the moodle server of the University of Kassel. It can be accessed by any internet browser and the address http://moodle.uni-kassel.de. The login data is your uni account, which is obligatory for every student at the University of Kassel. The course is located at FB16/Nachrichtentechnik. The password to register for this MATLAB®training will be announced at the classroom.

The manuscript is divided into several parts, which will be enabled for download during the semester. From time to time, additional excercises will be available at the moodle server and homework has to be uploaded there by every student. Besides this, the e-learning platform offers easy communication tools, discussion forums, calendars, etc. for every student.

1.3 Exam

The exam will be an oral exam of 30 minutes duration. To register for the exam you have to follow the instructions given in the FAQs on the ECE website. After the successful

registration for the exam via HIS-LSF or OKA you can send a request to me selig@uni-kassel.de to ask for an individual date for the exam.

1.4 Matlab®GUI

Throughout the rest of the course, please start MATLAB®by simply clicking the MATLAB®*for Training* desktop icon. This ensures to set the file server as your default work folder. After MATLAB®has started, you see the graphical user interface (GUI) of the program (see fig. 1.1).

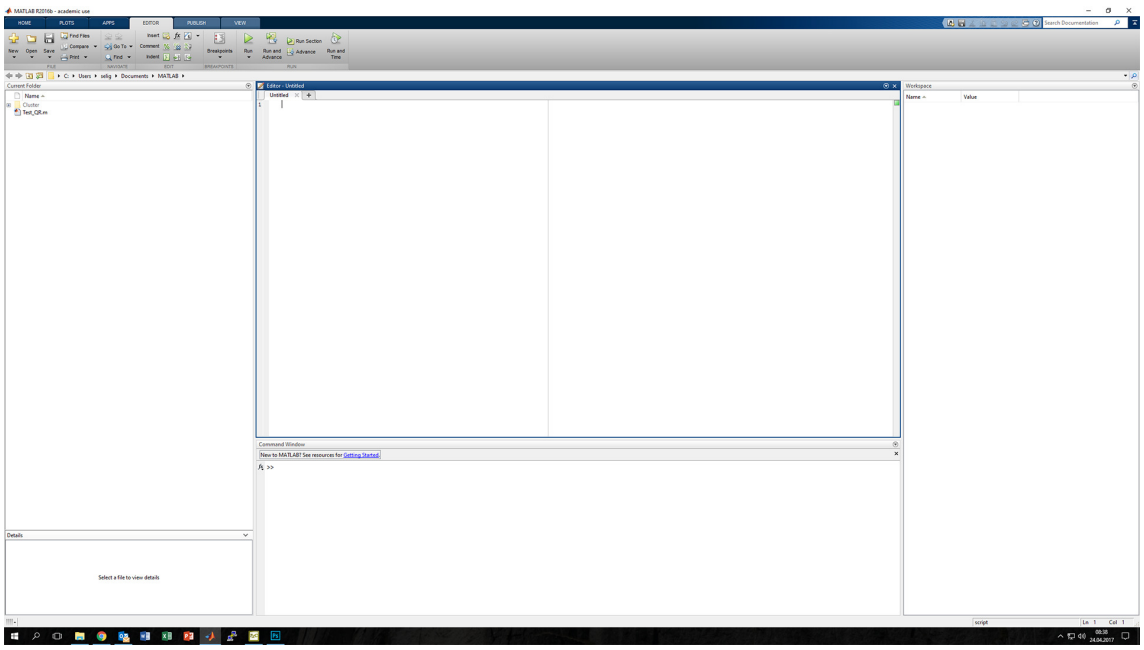

Figure 1.1: MATLAB®graphical user interface.

The MATLAB®GUI is divided into several subwindows, namely the *Workspace*, the *Command Window*, the *Editor* and the *Command History*. The *Workspace* contains all variables, which are declared at the time, and their corresponding values or properties (e.g. size of matrices). The *Command Window* includes a prompt (like DOS or Shell) to enter commands and to put out results. All commands entered in the *Command Window* are listed in the *Command History*, so you can easily remind the further entered commands. By double-clicking an entry of the *Command History* the appropriate command is executed in the *Command Window*.

Workspace

Command Window

Command History

Since the main tool is the *Command Window*, several keyboard shortcuts exist to simplify the work. Tab. 1.1 shows a cutout of important shortcuts, some more can be found in the MATLAB®Help (press F1).

You can use the *Command Window* to define variables and to enter functions or commands. Each output of a function is related to a variable. Different from other programming languages (like C++), you can not declare a variable, but you have to define it. Try out the following example:

TASK:
At the *Command Window* type in "`1+1`". If you confirm your input with the

↑	Recall previous line.
↓	Recall next line.
Home	Move to beginning of current statement.
End	Move to end of current statement.
Esc	Clear the command line when cursor is at the command line. Otherwise, move cursor to command line.
Enter in selection	Append selection to statement at command line and execute it.

Table 1.1: Keyboard Shortcuts in the Command Window

"Enter" key, MATLAB®generates a variable called "`ans`" (for answer) that is defined by the result of your input (that is "2", of course). You can see "`ans`" now in the *Workspace*. Now use the input "`clear`" to remove all variables from the *Workspace*. If you now enter "`x=1+1`", no variable "`ans`" will be generated, since the result of your input is already related to the variable "`x`". clear

Confirming an input with the enter key usually leads to an immediate output. Sometimes it is practical to avoid this output (e.g. all elements of a 100x100 matrix), since more than one input (calculation) is needed for the result of interest. Thus, there is an advantage of clarity by avoiding all intermediate outputs. Simply adding a semicolon (`;`) at the end of the line will calculate the input and display the output at the *Workspace*, but not in the *Command Window*. semicolon

In the example above, the "`clear`" function is used to delete all variables from the *Workspace*. But this function has got additional options, that can be discovered by typing "`help clear`" in the *Command Window*. Using "`help`" in front of a function name results in a help text that describes the usage of the appropriate function. You will get to know more functions later on. help

Pressing the F1 button, the "help browser" will open. In the left part of the window you see the "Help Navigator", which shows a list of all installed MATLAB®components. In the right window you see the content of the choosen topic. With the register "search" in the left window the documentation can be searched for a term.

1.5 Vectors and Matrices

The name MATLAB®stands for matrix laboratory. The program was developed to work with just one single type of data - the matrix. Values in the matrix can be real-valued or complex-valued. For this reason a matrix is defined by

- its name
- the number of rows
- the number of columns
- the values of all matrix elements

Up to version 4.2c only this one data type exists. Beginning with version 5.0 other data types as arrays, structures, lists, etc. are also used. In this lab we will mainly work with

the matrix data type. As you will see during the exercises a lot of problems in modern communication systems can be understood and solved using matrix models. A scalar is represented by a 1x1 matrix. A matrix with a row or column number equal 1 is called a vector.

1.5.1 Generating Matrices

Because MATLAB®only performs matrix and vector operations the assignment of values to a variable is an essential task.

In general, such an assignment has the following syntax:

$$\text{variable} = \text{expression}$$

Each time a variable is assigned a new expression or value, the old value is overwritten. An expression can be a complex formula, another matrix or a vector. For the name of the variable any sequence of alphanumeric characters not starting with a number is allowed. Including the underscore _, but excluding characters like: $+ - */., ()\% =$. For identification the first 63 characters of the variable name are considered. Older versions (< 7) only consider the first 31 characters. variables

There are also some predefined variables like i, j or pi. If you reuse them with different values, the original value will be overwritten. `i, j, pi`

Example for the creation of a 3x3 matrix:

The input `A = [1 2 3; 4 5 6; 7 8 9]` results in the output

$$\mathtt{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}.$$

The content of a variable is written into brackets [ ]. Elements in one row are separated by comma (,) or a blank. Different rows are separated by semicolons (;).

> **TASK:**
> Create a 4x4 matrix.

The letter e is also used for defining the power of 10.

```
2e-3 = 0.002
```

A special matrix is the empty matrix. It has 0 rows and 0 columns. It is defined by

```
>>empty = []
```

1.5.2 Basic Matrix Operations

- $+$ addition
- $-$ subtraction
- $*$ matrix multiplication
- $.*$ element-wise multiplication

- ^ power
- ' complex conjugate transpose
- .' transpose
- \ left division[1]
- / right division
- ./ element-wise division

If the sizes of the matrices do not fit for the operation, an error message will occur. Exceptions are scalar-matrix-operations, where each entry of the matrix is operated by the scalar.

TASK:
Make some experiments testing the operators mentioned above.

1.5.3 Relations and Binary Combinings

In MATLAB®the following relations and combinings can be used.

- < less
- > greater
- <= less-equal
- >= greater-equal
- == equal
- ~= unequal
- & and
- | or
- ~ not
- `any` returns 1 if at least one element of a matrix is unequal 0.
- `all` returns 1 if all elements of a matrix are unequal 0.

1.5.4 Submatrices and Colon Notation

In MATLAB®matrices and vectors can be used to realize complex data manipulations. Smart usage of submatrices and colon notation allows the user to reduce the number of necessary loops and keeps the code simple and readable. Being able to handle these features is important.

The *colon notation* is a practical way to create vectors, where the distance to the adjacent element is one. For this, only the start and the end value, seperated by a colon has to be defined.

colon notation

[1] A linear equation system is solved e.g. $x = A^{-1}B \Rightarrow x = A\backslash B$
MATLAB®does not directly calculate the inverse of A, but uses a more efficient algorithm to implement this.

Example:

```
>>1:4

  ans =
       1 2 3 4
```

The *extended colon notation* is utilized to create sequences, where the distance to adjacent elements is not equal to 1. The following example creates a sequence of numbers that are spaced by a value of 3:

```
>>1:3:8

  ans =
       1 4 7
```

The interval is defined with the outer values while the distance of two adjacent elements is defined by the value in the middle of the expression. The example also shows, it is not necessary that the upper border is one element of the output sequence. The colon notation yields the same result as using the command `linspace`, e.g. `linspace(1,7,3)`. For the definition of logarithmic distances the command `logspace` can be used. linspace

Parts of matrices or vectors can be extracted using braces ( ) to specify the subscripted elements. MATLAB®uses three different methods for subscripting submatrices

- Indexed subscripting, where rows and columns are defined to address a submatrix, e.g. A(1,2) addresses the element in the first row and the second column of A.

  The colon notation can be used to access submatrices, e.g. `A(1:4,3)` returns a column vector consisting of the first four entries of the third column of A.

  A colon by itself denotes a whole column or row (like a wildcard), e.g.
  `A(:,3)` adresses the third column of A.

  `A(2,:)` returns all elements in the second row of A.

  `A(1:4,:)` returns the first four rows of A.

  `A(1,[2 4])` extracts the second and fourth element in row one.

  `A(:,[2 3]) = A(:,[2 3])*[1 2;3 4]`. The Columns 2 and 3 of A are multiplied by a 2x2 matrix.

  `A(:,[1 3 5]) = B(:,1:3)` replaces columns 1, 3, 5 of A with the first three columns of B.

- Indexed subscripting, counting

  In this method a single index ranges from 1 to the total number of elements in the matrix. Counting is done column by column, starting with column 1, row 1. Please note, that the programming language C/C++ is counting row by row.

  `A(4)` returns the 4th element of A.

`A([1 3 5])` extracts elements 1, 3 and 5 and returns them as a row vector.

`A(:)` converts all elements of A into a single column vector. The matrix A is read out column-wise (column by column). If A is a NxM matrix, the result is then [A(1,1) A(2,1) ... A(N,1) A(1,2) A(2,2) ... A(N,M)].

- Binary subscripting

Binary subscripting is mostly done using a matrix of identical size with all elements being either 1 or 0. In the subscripting matrix a "1" addresses an element, a "0" does not. The argument of binary subscripting must be of logical data type, so the command logical has to be used for conversion.

`A(logical([0 1; 1 0 ]))` returns the secondary diagonal of A.

1.6 Functions

1.6.1 Matrix Building Functions

MATLAB®offers many commands to create special matrices. Some convenient matrix building functions are

- `eye(x)` produces an x by x identity matrix
- `zeros(x,y)` produces an x by y matrix of zeros
- `ones(x,y)` produces an x by y matrix of ones
- `diag(x)` produces a diagonal matrix from vector x
- `rand(x,y)` produces an x by y matrix with elements that are uniformly distributed in the interval (0,1)
- `randn(x,y)` produces an x by y matrix with elements that are Normally distributed with mean zero and variance one
- `repmat(A,x,y)` replicates and tiles the matrix A to produce an x by y block matrix.

Example:

```
>> A = [3 2 0; 2 1 -1]

A =

     3     2     0
     2     1    -1

>> B = repmat(A,2,3)
```

```
B =

     3     2     0     3     2     0     3     2     0
     2     1    -1     2     1    -1     2     1    -1
     3     2     0     3     2     0     3     2     0
     2     1    -1     2     1    -1     2     1    -1
```

TASK:
Matrices can be built from submatrices. Create an arbitrary matrix A of the size 3x3. Now, try the command
`B= [A, zeros(3,2); zeros(2,3),eye(2)]` to build a new matrix B of the size 5x5.

1.6.2 Math Functions

Some MATLAB®functions operate essentially on scalars, but operate element-wise when applied to a matrix.

- `sin`: sine
- `asin`: inverse sine
- `exp`: exponential
- `sqrt(x)`: square root $\sqrt{x}$
- `abs(x)`: absolute value of elements of x
- `conj(x)`: complex conjugate x^*
- `floor(x)`: round elements of x to nearest integer towards minus infinity
- `ceil(x)`: round elements of x to nearest integer towards plus infinity.

Some MATLAB®functions operate essentially on a vector (row or column), but act on a matrix in a column-by-column fashion to produce a row vector containing the results of their application to each column. Row-by-row action can be obtained by using the transpose (see 1.5.2). We could use the 'help' command in MATLAB®to see what kind of operation the following functions execute.

`max  sum  median  min  mean`

Example:

`median(y)` first sorts the elements of vector y according to their values in non-decreasing order. If `length(y)` is odd, `median(y)` is the equal to the element indexed by `(length(y)+1)/2`. If `length(y)` is even, `median(y)` is given by calculating the mean value of the two middlemost elements of `y`.

```
>> median ([ 1 2 200 5 6 300])

ans =

    5.5000
```

Example:

`max(y)` returns the largest element of y, if y is a vector. If it is a matrix, `max(y)` returns a row vector containing the largest element of each column of the matrix.

```
>> A = [3 2 0; 2 1 -1; 5 8 -1]

A =

     3     2     0
     2     1    -1
     5     8    -1

>> M = max(A)

M =

     5     8     0
```

TASK:
Try to get the minimum element in a matrix A. (This is a scalar.)

1.6.3 Matrix Functions

Matrix functions are applied to matrices. They perform certain operations on matrices, while their output could be a scalar, a vector or a matrix. MATLAB®is really powerful and efficient in these operations, what often yields to prefer matrix operations instead of loops to save computing time. Some useful matrix functions are:

- `eig`: eigenvalues and eigenvectors
- `svd`: singular value decomposition
- `inv`: inverse
- `sqrtm`: matrix square root
- `det`: determinant
- `size`: size of the matrix.

TASK:
Let's find the inverse of matrix A ...

```
>> X = inv(A)

X =

    0.4667    0.1333   -0.1333
   -0.2000   -0.2000    0.2000
    0.7333   -0.9333   -0.0667
```

How to check, if the result is correct?

```
I =

    1.0000    0.0000         0
    0.0000    1.0000   -0.0000
   -0.0000   -0.0000    1.0000
```

Where does the -0.0000 come from?

TASK:
Get to know the functions in the following list by using the '`help`' command in MATLAB®.

fix rem sign round prod sort std norm rank expm poly

1.6.4 Representation of Polynominals

Polynominals are described by a vector of their coefficents. The coefficent of the highest order is the first one in the vector. Therefore, the polynom $p(s) = s^2 + 3s + 2$ can be expressed as

```
>>p = [1 3 2]
```

The roots of the polynom are calculated using the command `roots` roots

```
>>roots(p)
  ans =
        -2
        -1
```

If the roots are known, the corresponding polynom can be calculated with `poly` poly

```
>>poly([-2 -1])
  ans =
        1  3  2
```

1.7 Building m-Files

You have learnt to include functions that are already implemented in MATLAB®. But it is also possible to program and include own functions. The programming language to realize own MATLAB®functions is very native. In fact, the code, which will be stored as so called "m-file", is a collection of commands and functions as they are used in the *Command Window*. These "m-files" can be executed by calling them from the *Command Window*, as it is done for built-in functions.

Writing a new m-file starts with opening the MATLAB®text editor. To do so, enter the command "`edit`" in the *Command Window*. This command can be utilized also for editing an existing file by simply adding the filename ("`edit` *`filename`*"). If the filename edit

does not exist, MATLAB®will ask for creating a new file with this name.

As mentioned before, programming an m-file is simply writing down the commands which shall be executed. Keep in mind to add semicolons (;) at the end of each line, if the appropriate result should not be displayed in the *Command Window*. Comments are marked with a "%" in front of the line. The first comment lines before a blank line arises will be plotted in the *Command Window* if the command "`help`" is applied to the m-file. % comments

Example:

m-file "test":

```
%This is the function "test"
%It calculates 1+1 and give it to
%the variable x.

x=1+1;
```

Command Window:

```
>> help test
 This is the function "test"
 It calculates 1+1 and give it to
 the variable x.

>>
```

Usually, these comment lines shall introduce the user in the function of the m-file.

The usage of input and output variables is done by using the command "function". The following examples shall demonstrate the usage of "function", see MATLAB®Help (F1 or "help function") for additional information: function

```
function ovar = fname(ivar)

function [ovar1,ovar2] = fname(ivar1,ivar2).
```

In the first example, one output variable "ovar" and one input variable "ivar" is used in the function "fname". Regard the squared brackets in the second example, if two or more output variables are needed. The variables (input/output) are separated with commas. Including the "`function`" command in the first line of an m-file leads to the declaration of the output and input variables. The value of each variable is either defined in the function call (defining input variables) or calculated in the m-file (output values).

1.8 Loops and Conditional Jumps

Commands stored in an m-file will be consecutively executed, but it is also possible to repeat several code lines (loop) or to jump to a certain code line if an argument is true

(conditional jump). Defining the conditions for a loop or jump is done by using logic *expressions* which can be either true or false. These are normally of the form '*expression* ***op*** *expression*', where ***op*** is ==, <, >, <=, >= or ~=. The following examples will explain the usage of the most important commands to include a conditional structure.

1.8.1 While

The included *statements* are executed an indefinite number of times while the real part of the *expression* has all nonzero elements.

Syntax

```
while expression
   statements
end
```

Example:

```
ii = 2;
x = 1;
while ii > 0
  x = x + 1;
  ii = ii - 1;
end
```

1.8.2 For

The included *statements* are executed a specific number of times. *expression* can be a vector or a scalar. Each column of *expression* is stored one at a time in the *variable* while the statements are executed. In most cases, the *expression* is of the form 'scalar:scalar' (see sec. 1.5.4).

Syntax

```
for variable = expression
   statements
   ...
   statements
end
```

Example:

```
x = 0;
for ii = 1:3
  x = x + ii;
end
```

Task:
Use a loop to multiply all even numbers of the interval [1 10]. What is the result?

1.8.3 If - Else - Elseif

Evaluating the conditions (*expression*) yields to a conditional execution of the appropriate *statements*. If *expression1* is true (nonzero), *statements1* is executed, otherwise *expres-*

sion2 is evaluated. The command *elseif* is utilized to implement a nested if-evaluation. It simplifies the coding in the case of a linear sequence of conditional statements with only one terminating end, since each if-command has to be terminated by a separate end-command.

Syntax

```
if expression1
   statements1
elseif expression2
   statements2
else
   statements3
end
```

The example shows two methods producing identical results.

Example:

```
if A                        if A
   x = a;                      x = a;
else                        elseif B
   if B                        x = b;
      x = b;                elseif C
   else                        x = c;
      if C                  else
         x = c;                x = d;
      else                  end
         x = d;
      end
   end
end
```

Task:
Write an m-file "'valproof.m"', that proofs if the input number is positive or negative or equal to zero. Use an if-expression for this task and output a "'-1"' for negative, "'+1"' for positive and "'0"' for zero.

1.8.4 Switch - Case - Otherwise

'Switch' executes one set of *statements* selected from an arbitrary number of alternatives (called *cases*). The case *statements* are executed, if the value of *switch_expr* is equal to the *case_expr*. If one alternative shall be executed by more than one *case_expr*, these can be grouped by using the form {*case_expr1, case_expr2, ...*}. If no *case_expr* fits the value of *switch_expr*, the *statements* following the *otherwise* command will be executed.

Syntax

```
switch switch_expr
   case case_expr1
     statements1
   case {case_expr2, case_expr3}
     statements2
   otherwise
```

```
    statements3
end
```

Example:

```
switch X
   case A
      x = a;
   case B
      x = b;
   case {C, D, E}
      x = e;
   otherwise
      x = f;
end
```

TASK:
Write an m-file "evenodd.m" to proof the input integer is even or odd. Use switch-case expression for this. The output is "2" for even input and "1" for odd input. What about the input of zero?

1.8.5 Break - Continue - Return - Error

There are additional commands which can be used in `for` or `while` loops. The `break` function terminates the execution of such a loop. Statements in the loop that appear after the `break` statement are not executed. The following example shows the creation of a random vector x, that will be stopped if the random element is equal to 3. The variable y is equal to the index of the last element before the value 3 arises, since this statement is not executed if the break is done.

break

Example:

```
for ii = 1:20
   x(ii) = randi(6,1);
   if x(ii) == 3
      break;
   end
   y = ii;
end
```

The `continue` statement passes control to the next iteration of the `for` or `while` loop. In contrast to the `break` statement, the loop will be executed the definite number of times, but statements following the `continue` command will be ignored. The following example shows the creation of a random vector x containing elements of either zero or one, where only the number of ones are counted by the variable y.

continue

Example:

```
y = 0;
for ii = 1:10
   x(ii) = randi(0:1,1);
   if ~x(ii)
```

```
      continue;
   end
   y = y + 1;
end
```

The `return` statement is utilized to return to the invoking function. In practical, `return` is often used to step back from an m-file in case of a wrong input parameter. The following example (from MATLAB®help) shows a function call to calculate the determinant of a matrix. If the assigned matrix is empty, a `return` statement causes a return to the invoking function. return

Example:

```
function d = det(A)
if isempty(A)
   d = 1;
   return
else
   ...
end
```

The `error` command displays an error message, quits the running program and returns control to the keyboard. In practice, the error command will often be used in combination with the commands `nargin, nargout`. With `nargin` the number of input arguments can be proofed, `nargout` proofs the number of output arguments. error nargin,nargout

Example:

```
function foo(x,y)
if nargin ~= 2
     error('Wrong number of input arguments')
end
```

1.9 2D- and 3D-Visualizations

In many cases, the calculated results shall be visualized on a graph, e.g. the signal magnitude over time. This section shall demonstrate the basics to create two- and three-dimensional graphs in MATLAB®.

1.9.1 Planar Plots

The easiest way to produce a planar (2D) plot is to use the command `plot`.

Syntax

```
plot(Y)
plot(X1, Y1, X2, Y2, ...)
plot(X1, Y1, LineSpec, ...)
```

`plot(Y)` The columns of Y are plotted versus its index. This is the simplest form of using the plot command. If Y is a complex-valued vector, the real part of Y is plotted versus its imaginary part.

`plot(X1, Y1, X2, Y2, ...)` The columns of Y1 and Y2 are plotted versus the columns of X1 or X2, respectively. By default, Y1 and Y2 are displayed in different colours.

`plot(X1, Y1, LineSpec, ...)` The specification of the plotted line in colour and type is included by defining a string. The string is of the form '<colour><plotsymbol><linetype>', where <colour> is one latter (abbrev. for colour), <plotsymbol> is one sign to define each tab and <linetype> consists of one or two signs defining the line type (see table 1.2).

<colour>		<plotsymbol>		<linetype>	
b	blue	.	point	-	solid
g	green	o	circle	:	dotted
r	red	x	x-mark	-.	dashdot
c	cyan	+	plus	–	dashed
m	magenta	∗	star	(none)	no line
y	yellow	s	square		
k	black	d	diamond		
		v	triangle (down)		
		ˆ	triangle (up)		
		<	triangle (left)		
		>	triangle (right)		
		p	pentagram		
		h	hexagram		

Table 1.2: Plot options to define line specifications

TASK:
Write the following code to the m-file "'sine.m"' and let the m-file run.

```
x = 0:pi/10:2*pi;
y = sin(x);
plot(x,y,'gh:')
```

The task result shows the plot of a sine wave. The line specification defines a green dotted line, where the vector elements of y are marked with hexagrams.

A plot is always displayed in a separate window, called a *figure*. If no *figure* is active, i.e. no plot window is open, a new *figure* is created. A new plot is always drawn in the active *figure*. If you want to create two separate plots, you have to open a new *figure* by typing `figure()`. Each *figure* is identified by a unique number that is increased by a counter every time a new *figure* is created. The command `figure(<num>)`, where <num> is the unique figure number, is utilized to address a previous opened *figure*. figure

Planar plots can be edited, e.g. by inserting a title, labels and a legend. This is very useful if you want to use the created plot in a document.

TASK:
Explore the functions `title`, `xlabel`, `ylabel`, `legend` and `grid` by using the MATLAB®Help and get to know the usage of these commands. Utilize these commands at the sine-plot created above.

1.9.2 Subplots

The command `subplot()` creates and controls multiple axes in one figure. This feature makes the comparison of two plots very easy, since both are displayed in the same figure. subplot

Syntax
`subplot(m,n,p)`,

where `m` denotes the number of rows and `n` the number of columns of an array that divides the figure in `n*m` axes. `p` is the number of the actually activated axes, which is counted line by line.

Example:

To enhance the m-file "sine.m" with the new subplot commands type:

```
x = 0:pi/10:2*pi;
y1 = cos(x);
y2 = sin(x);
subplot(2,1,1); plot(x,y1);
subplot(2,1,2); plot(x,y2);
```

The example shows the plots of a sine- and a cosine wave drawn in two axes in the same figure as it is shown in fig. 1.2.

1.9.3 Hold

The command `hold` has the parameter `on` or `off`. If `hold on` is set, then MATLAB®holds the current plot and all axis properties so that the next graphic commands are added

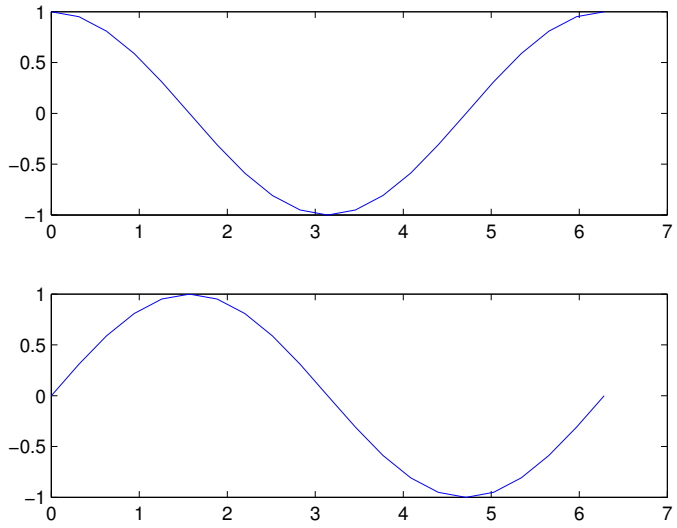

Figure 1.2: Example of subplots.

to the existing figure. With the command `hold off`, MATLAB®returns to default mode where new plot commands erase existing drawings and reset all axis before drawing the new plot.

1.9.4 Semilog-Plots

If the plotted data comprises a large scale of values, it is recommended to display the data in logarithmic scale. A very common application in communications is the graph of the bit error rate (BER) versus the signal-to-noise ratio (SNR). Both variables, BER and SNR, comprise large scales of values, hence they are usually displayed in a logarithmic form. Values of BER are typically of the form $5 * 10^{-4}$ (which is still linear!) and the SNR is given in dB.

Plotting BER versus SNR using the command `plot(SNR,BER)` will result in a linear scale of the y-axis, since the values of BER are of a linear form. The command `semilogy(SNR,BER)` can be used to display the values of BER on a logarithmic scaled y-axis and thus to avoid a transformation of these values to a logarithmic form (e.g. dB). Except the scaling of the y-axis, this command has the same functionality as the `plot()` function. The different results of the following example are shown in fig. 1.3.

semilogy

Example:

```
BER=[0.5 0.3 0.08 0.002 1e-5 5e-8 7e-9];
subplot(1,2,1); plot(BER);xlim([1 7])
subplot(1,2,2); semilogy(BER);xlim([1 7])
```

1.9.5 3D Plots

There are many cases, where the plotted function depends on more than one variable. In the field of communications the time-variant transfer function depends on frequency and time. Usually, it is drawn as a 3-dimensional (3D) figure with the absolute value versus

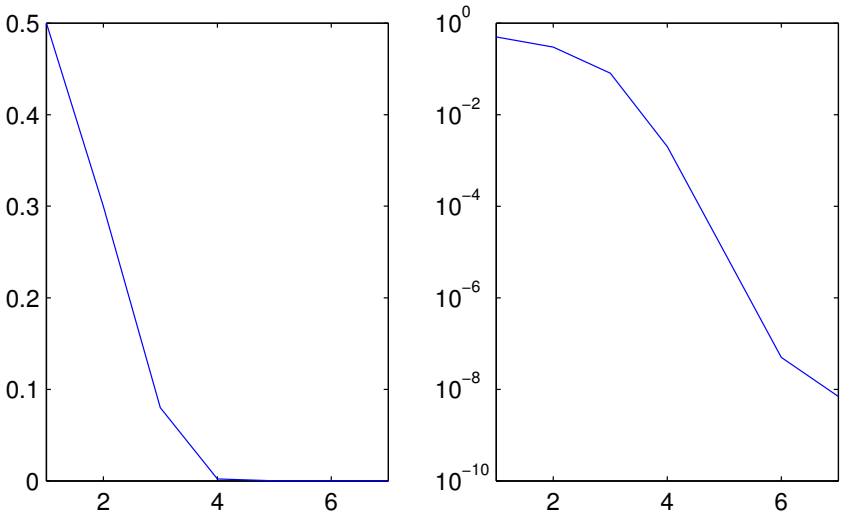

Figure 1.3: Display BER using 'plot()' (left) or 'semilogy()' (right).

frequency and time. In MATLAB®, there are two basic commands to produce such a 3D plot, namely `mesh` and `surf`. The usage of these functions is the same, but the result differs in its appearance. The command `mesh` creates a wireline model, where only the lines between adjacent values are coloured. Graphs created with `surf` have coloured surfaces between the defining points.

mesh
surf

Syntax

```
mesh(Z)
mesh(Z,C)
mesh(X,Y,Z)
mesh(X,Y,Z,C)

surf(Z)
surf(Z,C)
surf(X,Y,Z)
surf(X,Y,Z,C)
```

According to the syntax of the `plot`-command, `mesh/surf` is called in the same way. Z denotes the values that shall be plotted versus the X- and Y-axis. The variable C defines the colour for each point. In most practical cases, colour depends on the value of the point, hence $C = Z$ and C can be omitted.

TASK:
Create two vectors x and y, each containing at least 40 elements inside the interval $[-6\pi \;\; 6\pi]$. Perform a coordinate transformation from cartesian to polar coordinates by calculating the radius $r(ix, iy) = \sqrt{x^2(ix) + y^2(iy)}$. r is a matrix of the size `lenght(x)` x `length(y)`. For each element of this matrix, calculate $z = \frac{sin(r)}{r}$. Hint: Use two encapsulated `for` loops to create and access each element of r like it is done in the following

```
for ix=1:length(x)
   for iy=1:length(y)
     ...
```

Plot z versus x and y by using `surf()`. In the next step extend the existing code to avoid the "hole" at the peak of the plot.

The task draws a 3-dimensional sinc-function by using the command `surf`. The output is printed in fig. 1.4.

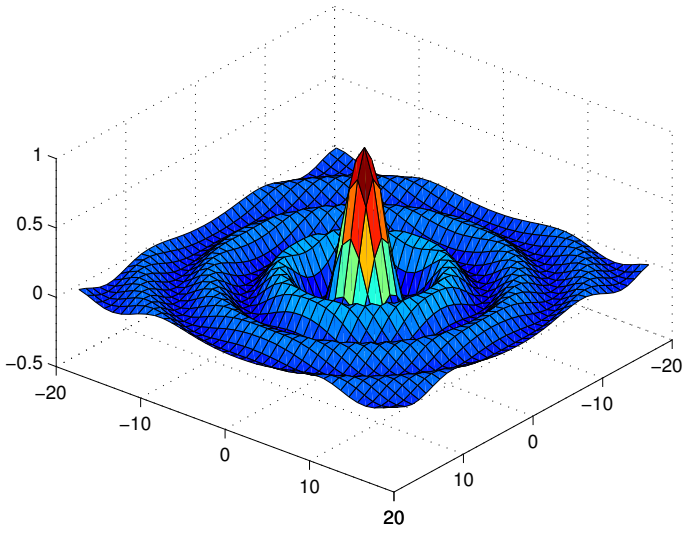

Figure 1.4: 3-dimensional sinc-function created with `surf`.

Another possibility to create a graph which depends on two variables is given by the command `plot3`. This function is utilized to print lines and points in 3-D space. Analogous to `plot`, the colour and line type can be varied by defining a line description. plot3

Syntax

```
plot3(x,y,z)
plot3(x1,y1,z1,x2,y2,z2,...)
plot3(X,Y,Z)
plot3(x,y,z,s)
```

The syntax of `plot3` is similar to the usage of `plot`, but `plot3` always needs at least three input arguments of the same size. x, y and z are vectors of the same length. Two or more lines can be plotted by using explicit vectors (x1, y1, z1) and (x2, y2, z2) or by defining matrices X, Y and Z of the same size which plot several lines obtained by the columns of X, Y and Z. The line description string (see `plot` in sec. 1.9.1) is denoted by the variable s.

Example:

```
t = 0:pi/50:10*pi;
plot3(sin(t),cos(t),t);
```

The example draws a helix in 3-D space.

1.10 Saving and Loading Data

1.10.1 Create a .MAT File with 'save'

A MATLAB®MAT-file is a binary-formatted file that is specific to MATLAB®and is used for efficient storage, i.e. loading and saving operations. MAT files literally save a MATLAB®workspace - variable names as well as values. These files are designed to be read by all platforms that run MATLAB®and are capable of storing the current state of a workspace.

MAT files are useful for holding results when you are in the middle of a simulation and have to leave, if you don't want to throw away your intermediate results. Issuing the command

```
>> save results.mat
```

stores all the variables in the workspace into the file results.mat.

You can also save specific variables to a MAT file. Assuming your workspace contains the variables x and y, then the command

```
>> save results.mat x y
```

saves only those two variables to the file results.mat.

Saving specific variables to a MAT file is helpful if you want to save some particular results as part of a larger or on-going simulation.

1.10.2 Use 'load' to Read a MAT File

The `load` command is used to retrieve the workspace. The syntax for using `load` is simple:

```
>> load results
```

causes the contents of the file results.mat to be loaded into the workspace (note that the .mat extension is not needed).

BEWARE: Because MAT files contain variable names as well as values, it is possible that a variable name in the MAT files is common to a variable name in the current workspace. In such conflicts, the existing variables are over-written by the variable being loaded.