MATLAB for Communications

a seminar by

Prof. Dr.-Ing. ANDREAS CZYLWIK

Contents

Preface

By Andreas Czylwik and Youssef Dhibi

Matlab (Mat*rix* Lab*oratory*) is a high-performance language for scientific and technological calculations. It integrates computation, visualization and programming in an easy-to-use environment where problems and solutions are expressed in familiar mathematical notation. It is a complete environment for high-level programming, as well as interactive data analysis. Some typical applications are

- system simulations,
- algorithm development,
- data acquisition, analysis, exploration, and visualization, as well as
- modeling, simulation and prototyping.

Matlab was originally designed as a more convenient tool (than BASIC, FORTRAN or C/C++) for the manipulation of matrices. It was originally written to provide easy access to matrix software developed by the LINPACK and EISPACK projects. Afterwards, it gradually became the language of general scientific calculations, visualization and program design. Today, Matlab engines incorporate the LAPACK and BLAS libraries, embedding the state of the art in software for matrix computations. It received more functionalities and it still remains a high-quality tool for scientific computation.
Matlab excels at numerical computations, especially when dealing with vectors or matrices of data. It is a procedural language, combining an efficient programming structure with a bunch of predefined mathematical commands. While simple problems can be solved interactively with Matlab, its real power is its ability to create large program structures which can describe complex technical as well as non-technical systems. Matlab has evolved over a period of years with input from many users. In university environments, it is the standard computational tool for introductory and advanced courses in mathematics, engineering and science. In industry, Matlab is the tool of choice for highly-productive research, development and analysis.

This tutorial script summarizes the tasks and experiments done during the seminar Matlab for Communications offered by the Department of Communication Systems of the university Duisburg-Essen. This seminar gives the students the opportunity to get first in touch with Matlab and further to have a background knowledge about the simulation of communication systems. After a detailed introduction describing the main usage as well as the different definitions in Matlab, some relevant selected topics, like amplitude modulation, fast Fourier transformation or convolution, are treated.

1 Introduction to Matlab

By Youssef Dhibi and Andreas Czylwik

1.1 What is MATLAB?

MATLAB[1] is a high-performance language for technical computing. It integrates computation, programming and visualization in a user-friendly environment where problems and solutions are expressed in an easy-to-understand mathematical notation.
MATLAB is an interactive system whose basic data element is an *array* that does not require dimensioning. This allows the user to solve many technical computing problems, especially those with matrix and vector operations, in less time than it would take to write a program in a scalar noninteractive language such as C or Fortran.
MATLAB features a family of application-specific solutions which are called *toolboxes*. It is very important to most users of MATLAB, that toolboxes allow to learn and apply specialized technology. These toolboxes are comprehensive collections of MATLAB functions, so-called *M-files*, that extend the MATLAB environment to solve particular classes of problems.
MATLAB is a matrix-based programming tool. Although matrices often need not to be dimensioned explicitly, the user has always to look carefully for matrix dimensions. If it is not defined otherwise, the standard matrix exhibits two dimensions $n \times m$. Column vectors and row vectors are represented consistently by $n \times 1$ and $1 \times n$ matrices, respectively.
MATLAB operations can be classified into the following types of operations:

- arithmetic and logical operations,
- mathematical functions,
- graphical functions, and
- input/output operations.

In the following sections, individual elements of MATLAB operations are explained in detail.

1.2 Expressions

Like most other programming languages, MATLAB provides mathematical expressions, but unlike most programming languages, these expressions involve entire matrices. The building blocks of expressions are

[1] MATLAB is a registered trademark of The MathWorks, Inc.

- Variables
- Numbers
- Operators
- Functions

1.2.1 Variables

MATLAB does not require any type declarations or dimension statements. When a new variable name is introduced, it automatically creates the variable and allocates the appropriate amount of memory. If the variable already exists, MATLAB changes its contents and, if necessary, allocates new storage. For example

```
>> books = 10
```

creates a 1-by-1 matrix named `books` and stores the value `10` in its single element. In the expression above, `>>` constitutes the MATLAB prompt, where the commands can be entered.
Variable names consist of a string, which start with a letter, followed by any number of letters, digits, or underscores. MATLAB is case sensitive; it distinguishes between uppercase and lowercase letters. `A` and `a` are not the same variable. To view the matrix assigned to any variable, simply enter the variable name.

1.2.2 Numbers

MATLAB uses the conventional decimal notation. A decimal point and a leading plus or minus sign is optional. Scientific notation uses the letter e to specify a power-of-ten scale factor. Imaginary numbers use either i or j as a suffix. Some examples of legal numbers are:

7 -55 0.0041 9.657838 6.10220e-10 7.03352e21 2i -2.71828j 2e3i 2.5+1.7j.

1.2.3 Operators

Expressions use familiar arithmetic operators and precedence rules. Some examples are:

`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division
`'`	Complex conjugate transpose
`( )`	Brackets to specify the evaluation order.

1.2.4 Functions

MATLAB provides a large number of standard elementary mathematical functions, including `sin`, `sqrt`, `exp` and `abs`. Taking the square root or logarithm of a negative

number does not lead to an error; the appropriate *complex* result is produced automatically. MATLAB also provides a lot of advanced mathematical functions, including Bessel and Gamma functions. Most of these functions accept complex arguments. For a list of the elementary mathematical functions, type

```
>> help elfun
```

Some of the functions, like `sqrt` and `sin` are built-in. They are a fixed part of the MATLAB core so they are very efficient. The drawback is that the computational details are not readily accessible. Other functions, like `gamma` and `sinh`, are implemented in so called *M-files*. You can see the code and even modify it if you want.

1.3 Handling Matrices

MATLAB was mainly designed to deal with matrices. In MATLAB, a matrix is a rectangular array of numbers. So scalars can be interpreted to be 1-by-1 matrices and vectors are matrices with only one row or column. MATLAB has other ways to store both numeric and nonnumeric data, but in the beginning of learning MATLAB, it is usually best to think of everything as a matrix. The operations in MATLAB are designed to be as natural as possible. Where other programming languages work only with single numbers, MATLAB allows to work with entire matrices quickly and easily.

1.3.1 Entering Matrices and Addressing the Elements

The elements of a matrix must be entered one-by-one in a list where the elements of a row must be separated with commas or blank spaces and the rows are divided by semicolons. The whole list must be surrounded with square brackets, e.g.

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

After pressing "Enter" MATLAB displays the numbers entered in the command line

```
A =  1  2  3
     8  6  4
     3  6  9
```

Addressing an element of a matrix is also very easy. The `n`-th element of the `m`-th column in matrix `A` from above is `A(n,m)`. So typing

```
>> A(1,3) + A(2,1) + A(3,2)
```

will compute the *ans*wer

```
ans =  17
```

The k-th to l-th elements of the m-th to n-th columns can be addressed by `A(k:l,m:n)`, e.g.

```
>> A(2:3,1:2)

 ans =  8  6
        3  6
```

Further examples:

```
>> A(1,1:2)
```

addresses the first two elements of the first row.

```
 ans =  1  2
```

```
>> A(:,2)
```

addresses all elements of the second column.

```
 ans =  8
        6
        4
```

1.3.2 Generating Matrices

There are different ways to generate matrices. Assigning elements explicitly was presented in the paragraph above. To create a row vector with 101 equidistant values starting at 0 and ending by π, this method would be very tedious. So two other possibilities are shown below:

```
>> x = linspace(0,pi,101)
```

or

```
>> x = (0:0.01:1)*pi
```

In the first case, the MATLAB *function* `linspace` is used to create `x`. The function's arguments are described by:

```
linspace(first_value, last_value, number_of_values)
```

with the default `number_of_values = 100`.
In the second case, the colon notation `(0:0.01:1)` creates an array that starts at 0, increments by 0.01 and ends at 1. Afterwards each element in this array is multiplied by π to create the desired values in x.

Both of these array creation forms are common in MATLAB. While the colon notation form allows to specify the increment between data elements directly, but not the number of data elements, the MATLAB function `linspace` allows to specify the number of data elements directly, but not the increment value between these data elements.

The colon notation is very often used in MATLAB, therefore a closer look should be taken on it.

`(first_value:increment:last_value)` creates an array starting at `first_value`, ending at `last_value` with an increment which can be negative as well, e.g.

```
>> v = (10:-2:0)

 v =  10  8  6  4  2  0
```

If the increment is 1, then its usage is optional:

```
>> w = (5:10)

 w =  5  6  7  8  9  10
```

MATLAB also provides four functions that generate basic matrices: `zeros`, `ones`, `rand` and `randn`.
Some examples:

```
>> B = zeros(3,4)

 B =  0  0  0  0
      0  0  0  0
      0  0  0  0

>> C = ones(2,5)*6

 C =  6  6  6  6  6
      6  6  6  6  6
```

`>> D = rand(1,5)` generates *uniformly* distributed random elements

```
 D =  0.5028  0.7095  0.4289  0.3046  0.1897
```

`>> E = randn(3,3)` generates *normally* -also called Gaussian- distributed random elements

```
 E =  -0.4326   0.2877   1.1892
     -1.6656  -1.1465  -0.0376
      0.1253   1.1909   0.3273
```

1.3.3 Concatenation

Concatenation is the process of joining small matrices to make bigger ones. In fact, the first matrix A was created by concatenating its individual elements. The pair of square brackets, [ ], is the concatenation operator. For an example, start with the 3-by-3 matrix `A`, and form

```
>> F = [A A+10; A*2 A*4].
```

The result is an 6-by-6 matrix, obtained by joining the four submatrices.

```
F =   1   2   3  11  12  13
      8   6   4  18  16  14
      3   6   9  13  16  19
      2   4   6   4   8  12
     16  12   8  32  24  16
      6  12  19  12  24  38
```

1.3.4 Deleting rows and columns

To delete rows or columns of a matrix, just use a pair of square brackets, e.g.

```
>> A(2,:)  = [ ]
```

deletes the second row of `A`.

```
A =  1  2  3
     3  6  9
```

It is not possible to delete a single element of a matrix, because afterwards it would not still be a matrix. (Exception: vectors, since here deleting an element is the same as deleting a row/column.)

1.3.5 Array Orientation

The orientation of an array can be changed with the MATLAB transpose operator ':

```
>> a = 0:3

 a =  0  1  2  3

>> b = a'

 b =  0
      1
      2
      3
```

1.3.6 Scalar-Array Mathematics

Addition, subtraction, multiplication and division by a *scalar* apply the operation to all elements of the array:

```
>> c = [1 2 3 4;5 6 7 8;9 10 11 12]

 c =  1   2   3   4
      5   6   7   8
      9  10  11  12
```

`>> 2*c-1` multiplies each element in `c` by two and subtracts one from each element of the result.

```
 ans =   1   3   5   7
         9  11  13  15
        17  19  21  23
```

1.3.7 Array-Array Mathematics

When two arrays have the same dimensions, which means that they have the same number of rows and columns, addition, subtraction, multiplication and division apply on an element-by-element basis in MATLAB.

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

 d =  1  2  3
      4  5  6

>> e = [2 2 2; 3 3 3]

 e =  2  2  2
      3  3  3
```

`>> f = d+e` adds `d` to `e` on an element-by-element basis

```
 f =  3  4  5
      7  8  9
```

`>> g = 2*d-e` multiplies `d` by two and subtracts `e` from the result

```
 g =  0  2  4
      5  7  9
```

Element-by-element multiplication and division work similarly, but the notation is slightly different:

```
>> h = d.*e
```

```
h =   2   4   6
     12  15  18
```

The element-by-element multiplication uses the dot multiplication symbol `.*`, the element-by-element array division uses either `./` or `.\`

```
>> d./e

ans =  0.500  1.000  1.500
       1.333  1.666  2.000

>> e.\d

ans =  0.500  1.000  1.500
       1.333  1.666  2.000
```

In both cases, the elements of the array in front of the slash is divided by the elements of the array behind the slash. To compute a matrix multiplication only the asterisk `*` must be used, e.g.

```
>> C = A * B
```

Therefore the *number of columns* of `A` must equal the *number of rows* of `B`.

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

A =  1  2  3
     4  5  6

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

B =  1  2
     3  4
     5  6

>> C = A * B

C =  22  28
     49  64
```

1.4 Graphics

MATLAB offers extensive facilities for displaying vectors and matrices as graphs, as well as annotating and printing these graphs. This section describes some of the most important graphics functions and gives some examples of some typical applications.

1.4.1 Creating a Plot

The plot function has different forms, depending on the input arguments. If `y` is a vector, `plot(y)` produces a piecewise linear graph of the elements of `y` versus the index

of the elements of `y`. If two vectors are specified as arguments, `plot(x,y)` produces a graph of `y` *versus* `x`. For example to plot the value of the sine function from zero to 2π, use

```
>> x = 0:pi/100:2*pi;
>> y = sin(x);
>> plot(x,y)
```

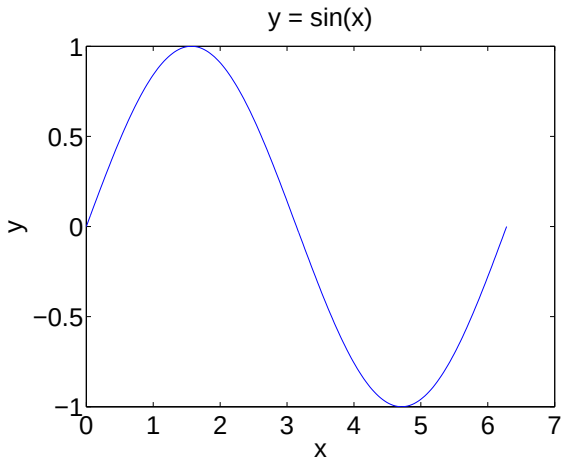


Figure 1.1: Sine plot

The `xlabel`, `ylabel` and `zlabel` functions are useful to add x-, y- and z-axis labels. The `zlabel` function is only necessary for three-dimensional plots. The `title` function adds a title to a graph at the top of the figure and the `text` function inserts a text in a figure. The following commands create the final appearance of figure 1.1 .

```
>> xlabel('x');
>> ylabel('y');
>> title('y = sin(x)')
```

Multiple `x-y` pairs create multiple graphs with a single call to plot. MATLAB automatically cycles through a predefined (but user settable) list of colors to distinguish between different graphs.
For example, these statements plot three related functions of `x1`, each curve in a separate distinguishing color:

```
>> x1 = 0:pi/100:2*pi;
>> y1 = sin(x1);
>> y2 = sin(x1 - 0.25);
>> y3 = sin(x2 - 0.5);
>> plot(x1,y1,x1,y2,x1,y3)
```

The number of points of the individual graphs may be even different. It is possible to specify the color, the line style and the markers, such as plus signs or circles, with: `plot(x,y,'color_style_marker')`

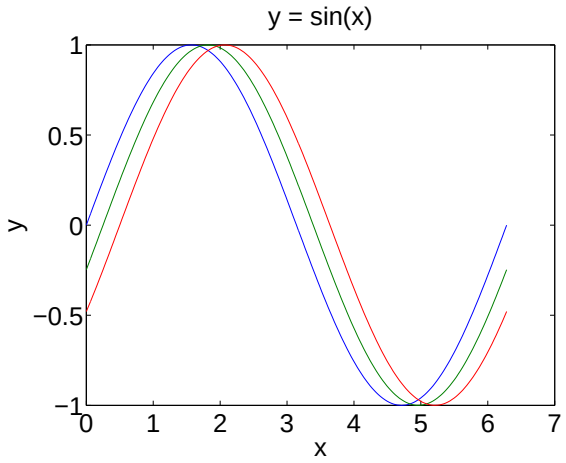


Figure 1.2: Multiple graphs with a single call to plot

A `color_style_marker` is a 1-, 2-, or 3-character string. It may consist of a color type, a line style type, and a marker type:
Color strings are `'c'`, `'m'`, `'y'`, `'r'`, `'g'`, `'b'`, `'w'` and `'k'`. These correspond to cyan, magenta, yellow, red, green, blue, white, and black.
Line style strings are `'-'` for solid, `'--'` for dashed, `':'` for dotted, `'-.'` for dash-dotted and `'none'` for no line.
The most common marker types include `'+'`, `'o'`, `'*'` and `'x'`.
For example, the statement `plot(x1,y1,'b:*')` plots a blue dotted line and places asterisk sign markers at each data point. If only a marker type is specified but not a line style, MATLAB draws only the marker.
The `plot` function automatically opens a figure window to plot the graphic. If there is already an existing figure window, this windows will be used for the new plot. The command `figure` can be used to keep an existing figure window and open a new one, which will be used for the next plot. To make an existing window the current window, type `figure(n)` where `n` is the number in the title bar of the window to be selected. The next graphic will be plotted in this selected window.
To add further plots to an existing graph, the `hold` command is useful. The `hold on` command keeps the content of the figure and plots can be added. So the above example could be done with three single `plot` commands and the `hold on` command. `hold off` ends the `hold on` status of a figure window. `hold` can be used to toggle between on and off.

1.4.2 Controlling Axes

Usually, MATLAB finds the maxima and minima of the data to be plotted by itself and uses them to create an appropriate plot box and axes labeling. The `axis` function overwrites this default by setting custom axis limits,

```
>> axis([xmin xmax ymin ymax]).
```

The following example illustrates the use of the functions presented above.

```
>> t = -pi:pi/100:pi;
>> s = cos(t);
>> plot(t,s)
>> axis([-pi pi -1 1])
>> xlabel('-\pi \leq t \leq \pi')
>> ylabel('cos(t)')
>> title('The cosine function')
>> text(-2, -0.5,'This is a note at position (-2, -0.5)')
```

`\leq` is used to generate the less-equal sign.

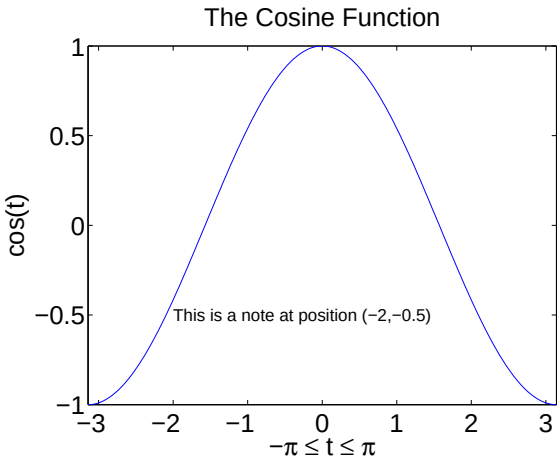


Figure 1.3: Example for controlling the axes

To take a closer look at an interesting part of a plot, the `zoom` command can be used. Afterwards it is possible to zoom by marking this part with the mouse. The `grid` command is used to turn a grid on and off.

1.5 Flow control

Computer programming languages offer possibilities to allow the programmer to control the flow of command execution. This flow control is based on decision making structures. Some of the most important structures are the `for`-loop, the `while`-loop and the `if-else-end`-structure. Since the constructions often affect several MATLAB commands, they are mostly used in *M-files* (see also 1.6). The `for`-loop repeats a group of statements a fixed, predetermined number of times. The general form of a `for`-loop is

```
>> for x = array
       commands...
   end
```

The `commands...` between the `for` and the `end` statements are executed one time for every column in `array`. For example

```
>> for k = 1:10
       z(k) = 2 * i;
   end
>> z
```

z = [2 4 6 8 10 12 14 16 18 20]
It is a good idea to indent the loops for readability, especially when they are nested.

```
>> for l = 1:5
     for m = 1:8
       H(l,m) = 1/(l+m);
     end
   end
```

While the `for`-loop evaluates a group of commands a fixed number of times, a `while`-loop evaluates a group of statements an indefinite number of times. The general form of a `while`-loop is

```
>> while expression
       commands...
   end
```

The `commands...` between the `while` and the `end` statements are executed as **all** elements in `expression` are **true** (nonzero). For example

```
>> a = 1; b = 10;
>> x = 1:10;
>> while (a =< b)
       z(a) = x(b-a);
       a = a+1;
   end
>> z

z = [ 10 9 8 7 6 5 4 3 2 1 ]
```

Many times, sequences of commands must be conditionally evaluated. In MATLAB this is provided by the `if-else-end` construction.

```
>> if expression
       commands1...
   else
       commands2...
   end
```

The `commands1...` between the `if` and the `else` statements are evaluated if **all** elements in `expression` are **true** (nonzero). Alternatively the `commands2...` between the `else` and the `end` statements are executed. For example

```
>> a = 5;
>> if a > 0
      c = 2*a;
   else
      c = -2*a;
   end
>> c

c = 10
```

1.6 Working with MATLAB

For simple problems, entering requests at the MATLAB prompt in the command window is fast and efficient. However, as the number of commands increases, or whenever a change of value of one or more variables with a reevaluation is desired, typing at the MATLAB prompt becomes tedious. MATLAB allows to place MATLAB commands in a simple text file, and by telling MATLAB to open this file, the stored commands are evaluated one-by-one as if they were just typed in. Those files are called *M-files*. There are two kinds of M-files:

- Scripts, which do not accept input arguments or return output arguments.
- Functions, which can accept input arguments and return output arguments. Internal variables are local to the function. The only difference in the syntax of a Script-file and a Function-file is the first line. The first line in a Function-file starts with the keyword `function` followed by the list of output arguments, an equals sign, the name of the function and ending with the list of input variables enclosed in parentheses and separated by commas. If the function has multiple output arguments, the output argument list must be enclosed in square brackets, e.g.:
  `function [x,y,z] = cosytrans(theta, phi, rho)`
  In a Script-file there is no predefined syntax for the first line.

To create or edit an M-file in the environment of *Linux* the command `nedit` can be used to start a text editor. The command `nedit` must be typed at a Linux command window. The use of this editor is very simple and the most important commands can be found in top-down-menus.
Another important aspect concerning the work with MATLAB should be mentioned here. The commands presented in this paper were introduced *without* a concluding semicolon. Therefore, a response to the commands occurs at the command prompt. So entering a new variable causes a repetition of the variable name and its values. Sometimes it is much better to avoid this repetition especially in large *M-files* since a load of information would appear on the screen and the really interesting data might get lost within this load. To suppress this response a concluding semicolon must be entered after the command, e.g.:

```
>> a = 0:5
```

leads to the response

```
a =  0  1  2  3  4  5
```

while

```
>> a = 0:5;
```

generates the same array, but does not display it.

Whenever there is a question about MATLAB, the best way to solve it, is to use the MATLAB help command which is a powerful tool searching within a huge data base. Just type `help` at the command prompt. A list of the main topics will be listed. By typing `help topic` the help can be specified. Additionally the cross-references help to find the interesting command with its options.

2 Vector Manipulations

By Rainer Siebel

2.1 Introduction

In Matlab applications it is often necessary to create vectors, to delete elements of given vectors, to substitute elements in given vectors, to append vectors or to insert a given vector into another vector. All these methods are discussed in this short tutorial.
The corresponding m-files are available.

2.2 Examples

2.2.1 Basic Operations

With the aim to visualize generated vectors, the first thing to do is to define figures and perhaps subfigures. This is done as follows:

```
%This m-file demonstrates basic MATLAB line vector manipulations

clear all      %=Clear all variables
close all      %=close all windows
figure(1);

subplot(211);  %upper subfigure
grid on;
box on
hold on;
subplot(212);  %lower subfigure
grid on;
box on
hold on;
```

In the first two lines all variables are cleared and all figures are closed.
Then figure 1 is defined and displayed with an upper and a lower sub-screen, a bounding box and a grid. With "hold on" all plots are kept on the screen.
The command subplot(m,n,p), or subplot(mnp), breaks the Figure window into an m-by-n matrix of small axes and selects the p-th axes for the current plot. Thus, subplot(211) creates two rows and one column of subplots in the current figure and selects the upper subfigure for plotting. Correspondingly, a command subplot(224) for example would create 2 rows and 2 columns of subplots in the current figure and selects the lower right

subfigure for plotting.
Notice, that all textstrings preceded by a % character or all lines starting with a % character are comment lines. Thus, the following strings are not interpreted as MATLAB commands

In the next lines some line vectors are defined and plotted:

```
%After we have defined some general screen settings we start here
%with the demonstration program.
%_________________________________________________________________
%Methods to define a line vector
%define a line vector with 10 zero elements
a0=zeros(1,10)
%define a line vector with 10 one elements
a1=ones(1,10)
%Defintion of a line vector element by element
a=[0 1 2 3 4 5 6 7 8 9 10];
%same vector using the linspace definition
b=linspace(0,10,11)
%same vector using another definition
b=[0:1:10] or b=[0:10]
```

In the lines above, several methods to define a line vector are given, i.e. defining a vector with all zero elements or a vector with all 1 elements or the definition of a vector element by element. The "`linespace(`x, y, z`)`" method defines linearly spaced vector elements with x the first element, with y the last element and with z the number of elements. Whereas the `[first:incr:last]` method defines a row vector with first element the increment value and the last vector element. If no increment value is given, 1 is assumed as increment.

```
%visualization of the vector a
subplot(211);
plot(a);
title('Plot of vector a');
```

with the lines above, the vector a is plotted to the upper sub-window. Notice, that the horizontal axes of the `plot(a)` command is automatically scaled by the index of the vector a, which starts always with 1 for the first element and ends with the N-th element, i.e. from 1 to N=11 in this case.
In the following lines the command "`fliplr`" is used, which arranges all vector elements in reverse order and the corresponding vector is plotted to the lower sub-window.

```
%arrange the vector elements in reverse order
c=fliplr(a);
%visualization of the vector c
subplot(212);
```

```
plot(c);
title('Plot of vector c');
pause
```

How to join two line vectors into a new one (appending vectors) is shown in the following few lines. Moreover, a title line is added to the sketch in the lower sub-window. Hold off is used to overwrite the previous plot in the lower sub-window.

```
hold off;
%appending vectors
d=[a c];
plot(d);
title('Plot of vector d');
grid on;
%which is a trapezoid function
pause;
```

How to delete single or multiple subsequent elements of a line vector is shown in the following lines. Before and after carrying out this operation give the following command to the command window: "length(c)", which confirms, that the length of the vector is reduced by one.

```
%Delete the first element of vector c
c(1)=[];
%accordingly we can delete elements 3..6 of vector c using
%c(3:6)=[];
%if we now redefine vector d we get a triangle
d=[a c];
plot(d);
title('Plot of vector d');
grid on;
pause;

%Now we delete 2 elements starting at the center of vector d
d(11:12)=[];
plot(d);
title('Plot of vector d');
grid on;
pause;
%Now we delete more elements from pos 7 to pos 14 and again get a triangle
d(7:14)=[];
subplot(211);
hold off;
plot(d);
title('Plot of vector d');
grid on;
```

The following lines show how to substitute vector elements by other vector elements. To avoid calculations we can use either the "length()" of the vector to be substituted or the "size()" definition. Notice, that the size() operation refers always to the matrix dimensions of variables, because all variables are in principle considered to be matrices. Hence, a line vector is a 1 by N matrix. Thus, "size(d,1)" is the number of rows = 1 here, "size(d,2)" is the number of columns (elements of the vector).

```
%inserting vectors into other vectors
%first we generate a zero line vector with 30 elements
e=zeros(1,30);
%and insert the previously defined vector d at position 10
%all of the following 3 lines carry out the same operation
e(1,10:1:20)=d
e(1,10:1:10+length(d)-1)=d
e(1,10:1:10+size(d,2)-1)=d
plot(e);
title('Plot of vector e');
grid on;
pause;
```

2.2.2 Mathematical Operations on Vectors

The operations discussed in the following are usually carried out on all elements of the vector(s).

```
%Now calculate the sum of all elements of the vector e
fprintf('Sum of e is: %3.2f\n',sum(e));

%Now determine the norm of the vector e
fprintf('norm of e is sum(e.*e): %3.2f or %3.2f\n',sum(e.*e),e*e');
```

Notice, that sum(e.*e) is the same operation as e*e'. Whereas the point multiplication e.*e carries out the multiplication element by element of the vector e, the e*e' operation is in principle a matrix multiplication. If e is a line vector, then e' is the transposed e-vector, i.e. a column vector and e*e' is the scalar quantity sum(e.*e) $= \sum_{i=1}^{N} e_i^2$, with e_i the vector elements of vector e.
The fprintf() command used above creates a formatted string. The principle construction rule of this command is well known from standard programming languages like C, FORTRAN etc.

```
t=fprintf('text %format string(s)\n', variable(s));
```

where t is a string variable (which may be omitted), the text string for output and the format string(s) and perhaps a line break must be enclosed in apostrophes, the format string(s) always start with a % character followed by the total number of characters to be printed, the . separator which separates the integer values from the fractions and the number of characters to be printed after the . separator. The character f, which follows

in the format string defines the floating point format. \n carries out one line break. Separated by a comma, the variable(s) follow(s) in the same order and with as many variables (or expressions) as format strings are defined.
Greek characters within the text string of fprintf() are generated by the LATEX-notation starting with a \followed by the name of the Greek character; like for example \Omega, \omega or \Gamma, \gamma for the corresponding upper- and lower case characters.

```
%square the elements of vector e or build any power of the
%vector elements of e.
f=e.*e
f=e.^2
subplot(212);
plot(f);
title('Plot of vector f');
grid on;
```

e.*e and e.^ 2 show the same results but e.^ x is more universal because x may be any power of the elements of vector e and x can be any real number.

2.2.2.1 Natural Functions of Vectors

```
%Natural functions of vectors
a
plot(sin(a));
%The plot shows a polygon, which does not resemble
%a sine function very much, because the horizontal
%spacing is not fine enough.
%The simple plot(sin(a)) command always connects the precise values
%of the vector components of sin(a) with straight lines. Thus, the plot()
%command always creates a polygon. However, if we create a vector with much
%narrower spacing the corresponding plot result resembles a sine function.
clear all;
a=linspace(0,2*pi,101);
plot(sin(a));     %This is still plotted against the vector index
grid;
pause;
plot(a,sin(a));  %Now sin(a) is plotted versus vector a
pause;
set(gca,'Xlim',[0,2*pi]);
grid;
```

The plot of natural functions requires usually a much more narrow spacing of the horizontal axis. If we use for example our original vector a with eleven elements and plot sin(a), then we get a polygon which does not very much resemble the sine function.
If we clear all vectors, redefine vector a with 100 intervals between 0 and 2π and plot it we get a much better resolution.
The notation plot(a,sin(a)) plots sin(a) versus vector a, which is now a properly scaled

function.
"set(gca,'Xlim',[0,2*pi]);" sets the entire horizontal range of the plot.
"set(gca,'Ylim',[-2,2]);" would set the vertical range.
Alternatively, we could use the axis() command to fix the vertical and horizontal scale of the plot, like for example:

```
axis([0,2*pi,-2,2])   i.e.  axis([xmin,xmax,ymin,ymax])
```

MATLAB provides all basic mathematical functions like all trigonometric and exponential functions and even more specific functions like Bessel, Gamma, Beta functions etc.
Type "help elfun" or "help specfun" to the command window to find the notation for all the build in functions.

2.3 Problems:

1. Define a linearly spaced line vector a in the range -2 .. +2 with 101 elements and plot this vector to the upper sub-screen with grid and proper horizontal and vertical scale.

2. Plot the square and other powers (3, 4) of this linear function to the lower sub-screen.
   Plot the square root of this linear function to the lower sub-screen and observe how MATLAB handles negative values of vector a.
   Add a proper title line.

3. Plot two periods of the function $\sin(\omega_0 t)$ over t with $f_0 = 10Hz$ with a proper horizontal scaling.

4. Plot two periods of the functions $\sin(\omega_0 t)$ and plot the corresponding $\sin^2(\omega_0 t)$ over t with $f_0 = 10Hz$ into the same diagram with blue, red lines and insert the legend.

5. Calculate the integral over 2 periods of the $\sin^2(\omega_0 t)$ function with $f_0 = 10Hz$ and check whether the result is reliable.

6. Delete 2 periods of the $\sin^2(\omega_0 t)$ function defined above, plot it over t and calculate the integral over 2 periods of it.

7. Generate one period of the function $x(t) = \sin(\omega_0 t)$ over t with $f_0 = 10Hz$ and plot the convolution of x(t)*x(t) using the conv-method.

8. Plot the convolution result of x(t)*x(-t) and check the horizontal range of the result.

9. Calculate the factorial of n by integration using the Gamma-function.
   $\int_0^\infty e^{-t} \cdot t^{n-1}\, \mathrm{d}t = \Gamma(n) = (n-1)!$

10. Do similar things with other natural functions.

3 Normalization of Physical Quantities

By Peter Laws

3.1 Introduction

Sometimes in order to simplify calculations, we have different methods to deal with variables which will join the calculations. Normalization is one of the methods. In today's computer mathematics, there are a few normalization methods. In this chapter, the general normalization method is used to normalize physical quantities as the preparation of calculation and programming. The principle of normalization is to reduce the large data range into smaller range. If the reduced data are not normalized, the calculation will overweight those features that have on an average larger values. One of advantages that normalization has is that the normalized data within small range,unified unit or numeral values can bring best results and simplified calculation.

3.2 Example: Normalization of Signals

The normalization of physical quantities can be done following the steps shown in the example of a voltage signal defined by

$$u(t) = U_0 \ \sin(2\pi f_0 t) \ \text{rect}\left(\frac{t}{T_v} - \frac{1}{2}\right), \tag{3.1}$$

where $U_0 = 1.5\text{V}$, $f_0 = 10\text{kHz}$, $T_v = 10T_0$, and $T_0 = 1/f_0$. The normalization of the signal is done step by step:

1. Determine the period T_0 of the sinusoidal part $\sin(2\pi f_0 t)$ of $u(t)$.
2. Determine the length T_v of the voltage signal $u(t)$.
3. Sketch $u(t)$ as a function of t with all significant values on abscissa and ordinate.
4. Plot $u(t)$ within the time interval $[t_l = -0.5\text{ms}, \ \ t_u = 1.5\text{ms}]$ using Matlab.

In order to complete normalization in the value/definition domain of u(t) we should first determine the normalizing quantities. Those are $U_N = 1V$, $f_N = 1Hz$, and $t_N = 1s$. Extending the normalizing quantities into equation (3.1) yields

$$u(t) = \frac{U_0}{U_N} U_N \sin\left(2\pi \cdot \frac{f_0}{f_N} \cdot f_N \cdot \frac{t}{t_N} \cdot t_N\right) \text{rect}\left(\frac{\frac{t}{t_N} \cdot t_N - \frac{\frac{T_v}{t_N} \cdot t_N}{2}}{\frac{T_v}{t_N} \cdot t_N}\right). \tag{3.2}$$

The normalized quantities are therefore $U_{0_n} = \frac{U_0}{U_N}$, $f_{0_n} = \frac{f_0}{f_N}$, and $t_n = \frac{t}{t_N}$, where the relation $f_N \cdot t_N = 1$ has to be fulfilled. The resulting normalized signal is

$$\frac{u(t)}{U_N} = U_{0_n} \sin\left(2\pi \cdot f_{0_n} \cdot t_n\right) \operatorname{rect}\left(\frac{t_n - \frac{t_{vn}}{2}}{T_{vn}}\right) = u_n(t_n). \tag{3.3}$$

After finishing the normalization process we can proceed with MATLAB programming. The next step is to try to write the conversion of data input/ouput handing and signal algorithm into a MATLAB script. Such a script is given in the following.

```
disp('');
Uon=input('Uon = normalized amplitude U0/V = ');
fon=input('fon = normalized frequency f0/Hz = ');
Tvn=input('Tvn=normalized pulse width Tv/s = ');
tln=input('tin = lower normalized time tl/s = ');
tun=input('tun = upper normalized time tu/s = ')
N=input('N =number of normalized time increments tin = ');
disp('');
clear t u 10. tin=(tun-tln)/N;
%normalized time increment tin
for n=1:1:N+1 12. tn=tln+(n-1)*tin;
      %next normalized time point
      x=(tn-Tvn/2)/Tvn;
        if abs(x)>0.5
               rect=0;
        else
               rect=1;
         end
      t(n)=tn;
      %normalized time vector t
      u(n)=Uon*sin(2*pi*fon*tn)*rect;
      %normalized signal vector u
 end
 plot(t,u,'b-')
%function plot
grid
xlabel('time t/s \rightarrow')
ylabel('u(t)/V \rightarrow')
title('Signal Plot')
```

At the beginning of the program, an empty line will be displayed by using the command `disp`. This command is usually used to display text or array. `disp(X)` displays an array, without printing the array name. If `X` contains a string, the string is displayed. In this case, `X` should be an empty string. The output will display an empty line at the very beginning of the outputs. Notice, `disp` does not display empty arrays.
From *line*2 to *line*7, some values of input variables are defined:

- Normalized amplitude U_{0_n}

- Normalized frequency f_{0_n}
- Normalized pulse width T_{v_n}
- Normalized lower time t_{l_n}
- Normalized upper time t_{u_n}
- Number of normalized time increments t_{i_n}.

The used rect-function is defined by

$$\text{rect}(x) = \begin{cases} 1 & \text{for} \quad |x| \leq 1/2 \\ 0 & \text{else.} \end{cases} \tag{3.4}$$

4 Basic Plotting

By Lars Häring

4.1 Plots with basic functionality

Plot two sine functions with frequency $f_1 = 200$ Hz and $f_2 = 50$ Hz and amplitudes $A_1 = 1$ V and $A_2 = 1.5$ V within the time interval t=[0,50] ms in the same figure. Use the sampling period $T = 10\mu s$.

`f1=200; f2=50; T=1e-5;`	Parameter settings...
`Tst=0; Te=5e-2; A1=1; A2=1.5`	
`t=[Tst:T:Te];`	A vector is defined containing needed sampling time instants.
`y1=A1*sin(2*pi*f1*t);`	The first output vector is calculated.
`y2=A2*sin(2*pi*f2*t);`	The second output vector is calculated.
`fig1=figure;`	A figure is opened and its handle is named 'fig1'.
`plot(y1);`	Plots the first output vector versus its indices and connects subsequent points with lines.
`plot1=plot(t,y1);`	Plots the first output vector versus input vector. Since we want to change the appearance of this plot later, we need to define the handle 'plot1'.
`xlab=xlabel('t (in s)');`	Displays label on x-axis.
`ylab=ylabel('y (in V)');`	Displays label on y-axis.
`ti=title('Sine Functions');`	Displays title above the figure.
`hold on;`	The active figure will not be overwritten by the next plot command. *Note*: 'hold off' will deactivate the effect; 'hold' toggles between 'hold on' and 'hold off'.
`plot2=plot(t,y2,'r--');`	Plots the second output vector versus input vector. *Note*: 'plot(t,y2,'g–')' defines color (here: green) and line style (here: dashdot) manually; for further information type 'help plot'.
`grid on;`	Activates grid of both x- and y-axis (syntax like 'hold').
`axis([Tst,Te,-1.7,1.7]);`	axis([xmin xmax ymin ymax]) controls axis scaling.
`set(plot1,'linewidth','2');`	Changes linewidth of first curve. *Note:* The same syntax can be used to change other parameters like color, fonttype and fontsize of labels etc.; a list of parameters can found in Matlab help.
`set(plot2,'linewidth','2');`	Changes linewidth of second curve.

```
leg=legend([plot1,plot2],'A1=1 V, f1=200 Hz','A2=1.5 V, f2=50 Hz');
```
Displays a legend for the curves with handles 'plot1' and 'plot2'.

`set(ti,'Fontsize',13);` Changes the font size of the title.

4.2 Plots using 'subplot'

Plot the same sine functions into two subfigures by using the command 'subplot'.

`f1=200; ... fig1=figure;` same as in 1.1

`sub1=subplot(211);` 'subplot(nr,nc,counter)' plots a subfigure within the active figure. The figure is divided into 'nr' rows and 'nc' columns and the last parameters determines the position of the subfigure (the numbering is row-wise and 1≤counter≤nr*nc).

`plot1=plot(t,y1);` same as in 1.1 ...

```
xlab1=xlabel('t (in s)');
ylab1=ylabel('y (in V)');
set(plot1,'linewidth',2);
grid on;
axis([Tst,Te,-1.7,1.7]);
```

`sub2=subplot(212);` Activates the second subfigure (below the first one).

... same as for first subfigure

```
leg1    = legend([plot1],'A1=1 V, f1=200 Hz');
leg2    = legend([plot2],'A2=2 V, f2=50 Hz');
```
Adds legend for the first and second subplot

4.3 Problems

4.3.1 Sampling of 'continuous' waveform

Given is the Matlab routine '[s]=waveform(t)' which generates the periodical signal $s(t)$ with period $T = 20$ ms. It is sampled in the time interval t=[0,65]ms with four different sampling frequencies $f1 = 100$Hz, $f2 = 250$Hz, $f3 = 500$Hz and $f4 = 1000$Hz. Display in four subfigures the 'continuous' waveform and the corresponding sampled waveform, respectively. Use for the sampled waveform the command 'stem'. Moreover, consider following properties of the figures:

General FontName of subplots:	'Times'
General FontSize of subplots:	8
x/y-Label:	Time (in s) / Amplitude (in V)
FontName of labels:	'Times'
FontSize of labels:	11
Linestyle/-color/-width of 'continuous' waveform:	dotted / RGB=[0.4,0.9,0.3] / 2
Linestyle/-color/-width of sampled waveforms:	solid,'.' / blue / 1

Axis scaling:	x=[0,65]*1e-3, y=[-1.1,1.1]
Legend:	'continuous', 'sampled'
Grid:	on

4.3.2 Simple 3D-Plots

Plot a two-dimensional Gaussian probability density function $f_{\mathbf{x},\mathbf{y}}(x, y)$ with zero mean and variance 1 within the range $x \in [-5, 5], y \in [-5, 5]$:

$$f_{\mathbf{x},\mathbf{y}}(x, y) = \frac{1}{2\pi} \mathrm{e}^{-\frac{1}{2}(x^2+y^2)} \tag{4.1}$$

Try out the commands `mesh`, `surf` and `contour` to plot the graph!

4.4 Basic Plot Routines

4.4.1 2D-Routines

- `plot(X,Y)`:
  plots vector `Y` versus vector `X`. If `X` or `Y` is a matrix, then the vector is plotted versus the rows or columns of the matrix, whichever line up. If `X` is a scalar and `Y` is a vector, `length(Y)` disconnected points are plotted. If `X` is left out, vector `Y` is plotted versus its indices.

- `bar(X,Y)`:
  draws the columns of the `M`-by-`N` matrix `Y` as `M` groups of `N` vertical bars. The vector `X` must be monotonically increasing or decreasing.

- `stairs(X,Y)`:
  draws a stairstep graph of the elements of vector `Y`. `stairs(X,Y)` draws a stairstep graph of the elements in vector `Y` at the locations specified in `X`. The `X`-values must be in ascending order and evenly spaced.

- `stem(X,Y)`:
  plots the data sequence `Y` as stems from the x-axis terminated with circles for the data value. `stem(X,Y)` plots the data sequence `Y` at the values specified in `X`.

- `semilogx(...), semilogy(...), loglog(...)`:
  is the same as `plot(...)`, except a logarithmic (base 10) scale is used for the x-axis, y-axis or both.

- `polar(THETA, RHO)`:
  makes a plot using polar coordinates of the angle `THETA`, in radians, versus the radius `RHO`.

4.4.2 3D-Routines

- `mesh(X,Y,Z,C)`:
  plots the colored parametric mesh defined by four matrix arguments. The view point is specified by `view`. The axis labels are determined by the range of `X`, `Y` and `Z`, or by the current setting of `axis`. The color scaling is determined by the range of `C`, or by the current setting of `caxis`. The scaled color values are used as indices into the current `colormap`.

- `surf(X,Y,Z,C)`:
  is the same as `mesh`. The shading model is set by `shading`.

- `surfl(X,Y,Z,C)`:
  is the same as `surf(...)` except that it draws the surface with highlights from a light source.

- `contour(Z)`:
  is a contour plot of matrix `Z` treating the values in `Z` as heights above a plane. A contour plot are the level curves of `Z` for some values `V`. The values `V` are chosen automatically. `contour(X,Y,Z)`: `X` and Y specify the (x,y) coordinates of the surface as for `surf`.

5 Random Functions

By Lars Häring

5.1 Continuous Uniform Distribution

Uniformly distributed, continuous random numbers are to be generated and their histograms are to be discussed.

`a=rand(1,100)`	rand(x,y) generates a matrix with x rows and y columns, whose elements are uniformly distributed within the range [0,1].
`hist(a);`	hist(a) plots the histogram. The value range of a is divided into 10 intervals. The number of elements in these intervals is plotted.
`a=rand(1,1e5);`	
`hist(a);`	
`hist(a,100);`	hist(a,N) plots the histogram. The value range of a is divided into N intervals. The number of elements in these intervals is plotted.

Mostly, random numbers not uniformly distributed within [0,1] but within an arbitrary range [a,b] are needed:

`a=rand(1,1e5);`	
`hist(a,100);`	a $\longrightarrow$ a is RV $\in$ [0, 1]
`a=rand(1,1e5)+0.5;`	
`hist(a,100);`	a $\longrightarrow$ a is RV $\in$ [0.5, 1.5]
`a=rand(1,1e5)+n;`	a $\longrightarrow$ a is RV $\in$ [n, n+1]
`a=2*rand(1,1e5);`	
`hist(a,100);`	a $\longrightarrow$ a is RV $\in$ [0, 2]
`a=A*rand(1,1e5);`	a $\longrightarrow$ a is RV $\in$ [A·0, A·1]
`a=A*rand(1,1e5)+N;`	a $\longrightarrow$ a is RV $\in$ [N, A+N]

Problem: Generate 100.000 random numbers that are uniformly distributed within the range [-10,20] and plot the histogram!

5.2 Discrete Uniform Distribution

Now, *discrete* uniformly distributed random numbers shall be generated.

`a=floor(N*rand(x,y)+1);`	This command generates a matrix with x rows and y columns, whose elements are uniformly distributed integer numbers within the range [1..N].
`a=floor(20*rand(1,1e5)+1);`	
`hist(a)`	In order to see the functionality of this command, the standard resolution of hist is not sufficient.
`hist(a,100)`	Now you can see that values can only take integer values from 1 to 20.

Problem: When tossing a fair dice, all events 1,2,3,4,5,6 are equally likely. Generate 100.000 random numbers that represent the results of a dice. Calculate the average value, the standard deviation and the variance of these results.

5.3 Gaussian Distribution

In the following, continuous, gaussian distributed random numbers are generated and their histograms plotted.

`b=randn(1,1e5);`	randn(x,y) generates a matrix with x rows and y columns, whose elements are normally distributed (Gaussian, zero mean, variance 1).
`hist(b,100)`	

Problem: Calculate the mean value and the variance of b.

`x=sqrt(5)*randn(1,1e5)+1;`	Mean and variance is changed. Therefore, x is Gaussian distributed now.
`hist(x,100)`	

Problem: Calculate the mean and the variance of x.

5.4 Mean and Variance

Zero mean, uniformly distributed noise is to be generated whose variance is equal to the variance of normally distributed noise.

`x1=randn(1,1e5);`	x1 contains 100.000 samples of normally distributed noise.
`x2=rand(1,1e5);`	x2 contains 100.000 samples of uniformly distributed noise.
`mean(x2)`	x2 has mean 0.5.
`x2=x2-mean(x2);`	Now, x2 is zero mean.
`var(x2)`	Zero mean x2 has generally a variance of $\frac{\text{width}^2}{12}$.
`x2=1/sqrt(var(x2))*x2;`	Now, x2 is zero mean AND has the variance 1.

Problem: Plot the first 1.000 samples of x1 and x2 one on top of the other with the same axis scaling.

5.5 Central Limit Theorem

Now, it will be shown that the (infinite) sum of statistically independent random variables exhibits a Gaussian distribution.

Problem: Represent four uniformly distributed random variables a,b,c,d with 100.000 samples, respectively. Plot the histograms of the sums of random variables as follows:

a	a+b
a+b+c	a+b+c+d

5.6 Transformation of Random Variables

Finally, the transformation of a random variable through a nonlinear characteristic $y = f(x)$ is investigated.

Problem: Use the given Matlab function "y=NonLinearCharacteristic(x)" and plot the characteristic in the range -10 < x < 10. Transform a zero mean, uniformly distributed random variable with 100.000 samples through the nonlinear characteristic and plot the histogram.

6 Programming Structures

By Stefan Bieder

6.1 Using Functions

In previous dates it is shown how to use script-Files by putting together many commands into a single command block. By defining *functions* it is possible to give parameters to the command blocks and to get back parameters as the result of calculations of the command block.
To use a function named `NameOfFunction` a script file must be created named `NameOfFunction.m`. A general structure of a this script file is given as follows:

```
function [Out1, Out2, ...] = NameOfFunction(In1, In2,...)
Command block
```

Thereby, the variables `Out1, Out2` are the output of the function named `NameOfFunction` while the variables `In1, In2` are input variables of the function. The structure above is written in a script file and saved as `NameOfFunction.m`.
Then, at the command window or at other script files (functions) the function `NameOfFunction` is called by

```
[Var1, Var2, ...] = NameOfFunction(Var3, Var4,...);
```

Example 1:
The function `DateInDayName` stored in the script file `DateInDayName.m` converts a given date with the format DD.MM.YYYY into its corresponding week day:

```
function [DayName] = DateInDayName(Day,Month,Year);
% Author Stefan Bieder
% Version 1.0
% Date 13.05.2004
%
% Converts a Date into the name of the corresponding day
% valid range 01.03.1900 - 28.02.2100
% Input parameters: Date
% Day - Number 1..31
% Month - Number 1..12
% Year - 1900..2100
% Algorithm found at http://www.math.uni-bonn.de/people/gw/wochentag.html

%%%%%%%% Command Block
%
```

```
% Prelimery settings
MonthTable=[ 1 4 3 6 1 4 6 2 5 0 3 5;
             1 1 0 0 0 0 0 0 0 0 0 0 ];
DayTable=cell(1,7);
DayTable(1) = {'Sunday'};
DayTable(2) = {'Monday'};
DayTable(3) = {'Tuesday'};
DayTable(4) = {'Wednesday'};
DayTable(5) = {'Thursday'};
DayTable(6) = {'Friday'};
DayTable(7) = {'Saturday'};

% Calculation of the DayName using the algorithm proposed at
% http://www.math.uni-bonn.de/people/gw/wochentag.html
%
Year=Year-MonthTable(2,Month);
Year=Year-1900;
a=floor(Year/4)+Year+Day+MonthTable(1,Month);
DayIndex=mod(a,7)+1;

% DayIndex gives the Index of the DayNames within the DayTable
DayName=DayTable(DayIndex); % The return variable is set with its value
```

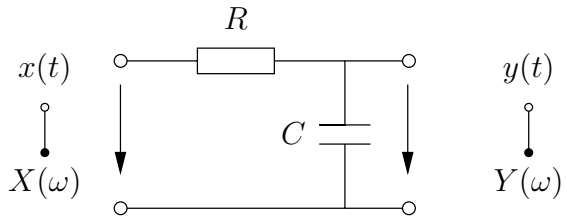
To get the week day of the date 01.01.2000 and store it in the variable `name` the function is called at the command line with `name=DateInDayName(01,01,2004);`.

Task 1:

Determine the week day of your own day of birth and check for the next 10 years whether your birthday is at a weekend or not.

Task 2:

Given is the general transfer function of a RC-lowpass filter:



RC lowpass filter

The transfer function of the RC-lowpass filter is $H_{\mathrm{LP}}(\omega) = \dfrac{1}{1 + \mathrm{j}\omega RC}$ and the phase of the RC-lowpass filter is $\varphi_{\mathrm{LP}}(\omega) = -\arctan(\omega RC)$

Program a function named `[mag,phase]= function RC_Lowpass(omega,R,C)` which returns the values of $|H_{\mathrm{LP}}(\omega)|$ and $\varphi_{\mathrm{LP}}(\omega)$ for given R and C and a vector of specified frequencies ω! Use your programmed function to plot the values of $H_{\mathrm{LP}}(\omega)$ for $\omega = \{0, (1), 100\}$ for $R = 1 \cdot 10^6$ and $C = 10 \cdot 10^{-9}$.

6.2 Programming Language Constructs

The programming language constructs of MATLAB can be used in script files and at the command line also. You can get an overview of all constructs using `help lang`. In this seminar we want to describe just the main important commands for the control flow.

6.2.1 if..., else..., end

The general form of a simple `if` construct is

```
if relation (logical condition)
 commands
end
```

The commands will be executed only if the logical condition is true. By using the `elseif` and `else` construct, it is possible to build up a branch of tests of logical conditions.
Example 2:

```
function []=TestVar1(n);
% Tests whether n is positive, negative or zero.
% Author Stefan Bieder
% Version 1.0
% Date 05.07.2004

if n < 0
    disp('n is negative');
elseif n > 0
    disp('n is positive');
else
    disp('n is zero');
end
```

Task 3:

> Test how the function `TestVar1` works with different `n`, e.g. `n=10`, `n=-10`, `n=0`, `n=[-1 3]`...
> Extend the function `TestVar1` such that it outputs `n is a matrix` if `n` is not a scalar.

For following operators can be used for the relation (logical condition).

`<`	less than
`>`	greater than
`<=`	less than or equal
`>=`	greater than or equal
`==`	equal
`~=`	not equal

Note that "=" is used in an assignment statement while "==" is used in a relation. Relations may be connected or quantified by the logical operators:

`&` and
`|` or
`~` not

When applied to scalars, a relation is actually the scalar 1 or 0 depending on whether the relation is true or false.

Task 4:

Try entering `3 < 5`, `3 > 5`, `3 == 5`, `3 == 3` ...

When applied to matrices of the same size, a relation is a matrix of 0's and 1's giving the value of the relation between corresponding entries.

Task 5:

Try entering

```
>> a = rand(3)
>> b = rand(3)
>> a == b
>> a == a
>> a(1,1) =  b(1,1)
>> a == b
>> a ~= b
```

A relation between matrices is to be true if each entry of the relation matrix is nonzero. Hence, if you wish to execute statement when matrices A and B are equal (which means that all entries are equal) you have to type

```
if all(all(a == b))
   statement
end
```

but if you wish to execute statement when only a single entry of A and B is equal, you have to type

```
if any(any(a == a))
   statement
end
```

Task 6:

Try entering

```
>> a = rand(3)
>> b = rand(3)
>> a(1,1) =  b(1,1)
>> a ~= b
>> if all(all(a~=b)) disp('yes'); else disp('no'); end
>> if any(any(a~=b)) disp('yes'); else disp('no'); end
```

Note that the seemingly obvious `if A ~= B, statement, end` will not give what is intended since statement would execute only if each of the corresponding entries of A and B differ. The functions `any` and `all` can be creatively used to reduce matrix relations to scalars. Two `any`'s are required above since `any` is a vector operator.

6.2.2 for, while loops

The structure of a `for` loop is as follows:

```
for VAR=START:INCREMENT:STOPP
    Command Block
end
```

The for loop repeats the command block as long as the value of the variable VAR is STOPP. At the first run of the loop the variable VAR has the value START, the command block is run then the variable VAR is incremented with INCREMENT, the command block is run...

Example 3:
The following example calculates the sum of the numbers 1...100.

```
S=0;
for i=1:1:100
    S=S+i;
end
```

Example 4:
The following example shows a (slow) realization of a step function

```
function [Out]=Step(In);
% Slow relization of the step function
% (In(x) >=0 => Out(x)=1, In(x)<0 => Out(x)=0)
% Author Stefan Bieder
% Version 1.0
% Date 05.07.2004

for x=1:length(In)
    if In(x)<0
        Out(x)=0;
    elseif In(x)>=0
        Out(x)=1;
    end
end
```

Task 7:

Program a function `ToggleChar` that toggles the characters of an inputs string `S` from lower case to upper case and vice versa. Use the commands:

`double`	converts ASCII string into according numbers
`char`	converts numbers into according ASCII string

Notice: `for` loops are used if the number of iterations is known in advance, because this number is set by `VAR=START:INCREMENT:STOPP`. Without this knowledge, `while` loops have to be used. The structure of a `while` loop is as follows:

```
while relation (logical condition)
 command block
end
```

Example 5: The following program adds all numbers input at the keyboard. It is not known in advance how many numbers will be input by the user.

```
function [Out]=AddNum();
% Adds all numbers input at the keyboard
% 'X' for end
% Author Stefan Bieder
% Version 1.0
% Date 05.07.2004

dat=input('Type a number "0" ends program ');
S=0
while dat~=0
   if isnumeric(dat)
       S=S+dat;
   else
       disp('This is not a number, "0" for cancel')
   end
   disp(['Sum is: ' num2str(S)])
   dat=input('Type a number "0" ends program ');
end
```

7 Amplitude Modulation

By Youssef Dhibi

7.1 Transmission Process

A general model of transmission is shown in the following figure.

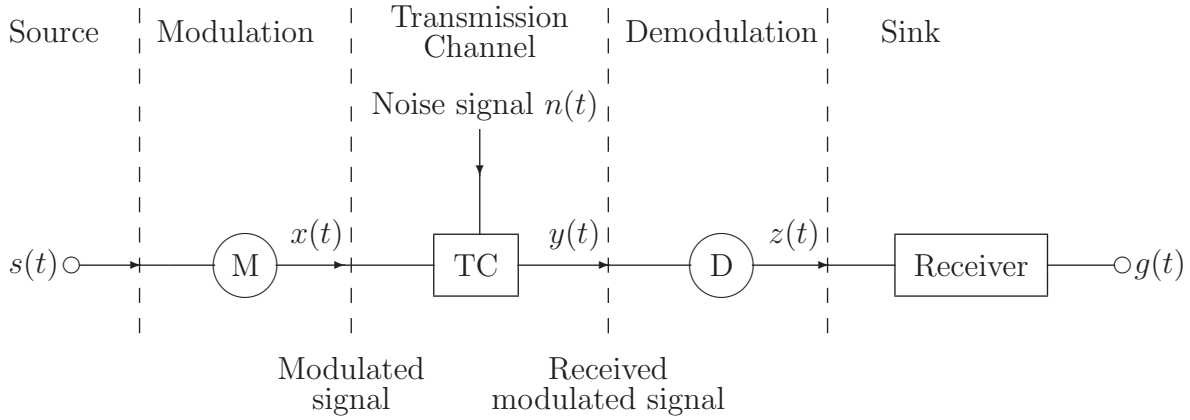


Fig. 1: Model of transmission process

The *baseband* signal $s(t)$ has to be adapted to the properties of the channel. Therefore, the low baseband frequency region is usually shifted to a higher frequency region. That means that transmitting the signal $x(t)$ through the transmission channel (TC) involves translating the baseband message signal $s(t)$ to a bandpass signal $x(t)$ at frequencies that are usually much greater than the baseband frequency. This process is called *modulation*.

After transmitting $x(t)$ through the TC, the modulated signal $y(t)$ is received, which contains both signals, the *source message* and the *carrier*. The modulating signal $m(t)$, as we will see later, *carries* the source signal $s(t)$ as its envelope and therefore, the notation carrier gets self-evident. The next step is to extract the baseband source message $s(t)$ from the carrier so that it may be processed and interpreted at the receiver (*sink*). This process is called *demodulation* and it will be closer examined later.

7.2 Ideal Modulator

To achieve the modulation, a basic element, the *modulator*, is needed. Fig. 2 illustrates the so-called *ideal modulator*, which simply multiplies the source signal $s(t)$ with the

modulating signal $m(t)$ to generate the *modulated signal* $x(t)$

$$x(t) = s(t)\, m(t) \quad \circ\!\!-\!\!\bullet^{\mathcal{F}} \quad X(\omega) = \frac{1}{2\pi} S(\omega) * M(\omega).$$

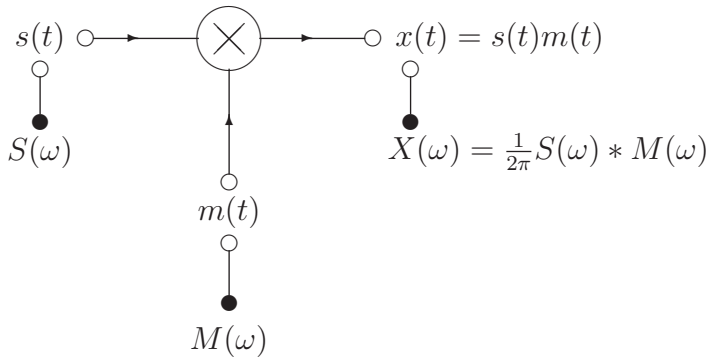


Fig. 2: Ideal Modulator

The ideal modulator translates the source spectrum $S(\omega)$ from the baseband frequency to the high frequency region around the carrier frequency ω_c. Such a shifting is impossible with an *LTI*-system. It can only be achieved either with a *non-linear* and/or *time-variant* system. The ideal modulator is a *linear time-variant* system and therefore usable for modulating baseband signals. When the modulating signal is given by

$$m(t) = A\cos(\omega_c t) \quad \circ\!\!-\!\!\bullet^{\mathcal{F}} \quad M(\omega) = A\pi[\delta(\omega - \omega_c) + \delta(\omega + \omega_c)],$$

for any signal $s(t)$ with a limited spectrum $S(\omega)$, the spectrum of the modulated signal $X(\omega)$ can be calculated as follows

$$\begin{aligned} x(t) = As(t)\cos(\omega_c t) \quad \circ\!\!-\!\!\bullet^{\mathcal{F}} \quad X(\omega) &= \frac{A}{2} S(\omega) * [\delta(\omega - \omega_c) + \delta(\omega + \omega_c)] \\ &= \frac{A}{2} S(\omega - \omega_c) + \frac{A}{2} S(\omega + \omega_c). \end{aligned}$$

The modulation may be done by varying the amplitude, phase or frequency of the carrier in accordance with the amplitude of the message signal. In practice, sinusoidal signals are mainly used as modulating signals. Therefore, this type of signals will be closer examined.

7.2.1 Modulated Signal

In AM, the amplitude of a high frequency carrier signal is varied in accordance to the instantaneous amplitude of the modulating message signal. The principle of AM will be explained first by a sinusoidal source signal

$$s(t) = \cos(\omega_s t),$$

where ω_s is the *cut-off* frequency of $s(t)$ and should be much lower than the carrier frequency ω_c. As mentioned earlier, the AM is a linear modulation method and the AM modulated signal (denoted as $x_{AM}(t)$) can be written as

$$x_{AM}(t) = A\,(1 + m_{AM}\cos(\omega_s t))\ \cos(\omega_c t + \varphi),$$

with A is an *arbitrary constant* and m_{AM} is called *degree of modulation* (also called *modulation index*) of the AM. There are two interesting cases of m_{AM}:

- Case 1: $|m_{AM}| \leq 1$. In this case, $x_{AM}(t)$ (solid line) is oscillating between the two *envelopes* $A\,(1 + m_{AM}\cos(\omega_s t))$ (dashed line) and $-A\,(1 + m_{AM}\cos(\omega_s t))$ (dotted line) as shown in the following figure.

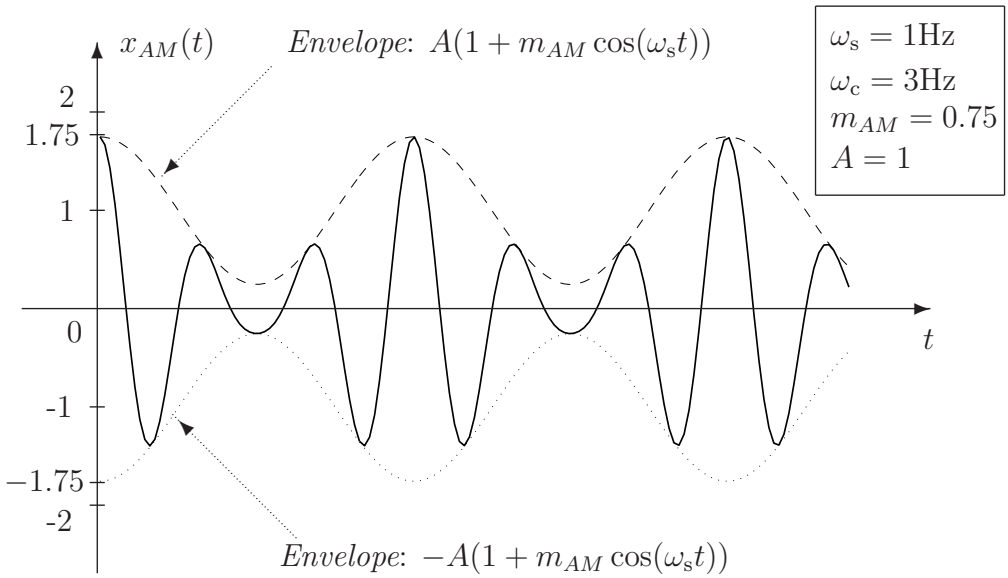


Fig. 3: AM Signal $x_{AM}(t)$ with $|m_{AM}| \leq 1$

- Case 2: $|m_{AM}| \geq 1$. Here, the AM signal $x_{AM}(t)$ (solid line) is oscillating between the two *envelopes* $|A\,(1 + m_{AM}\cos(\omega_s t))|$ (dashed line) and $-|A\,(1 + m_{AM}\cos(\omega_s t))|$ (dotted line) as shown below.

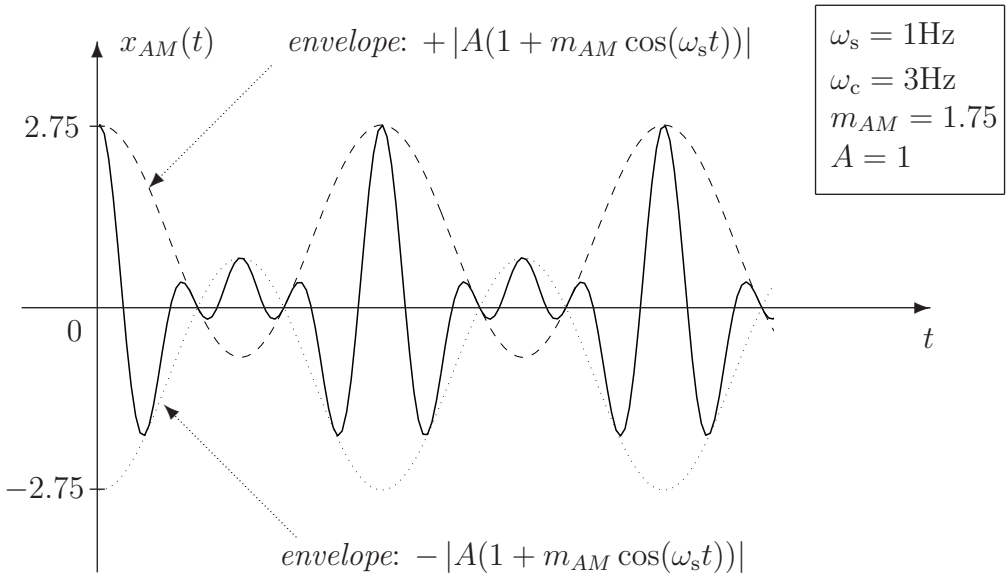


Fig. 4: AM Signal $x_{AM}(t)$ with $|m_{AM}| \geq 1$

In contrast to the case $|m_{AM}| \leq 1$, an m_{AM} greater than 1 will distort the message if an *envelope detector* is used as demodulator.

To obtain the modulated signal $x_{AM}(t)$, a modulation circuit is needed. Generally, AM signals $x_{AM}(t)$ could be generated with the system shown below.

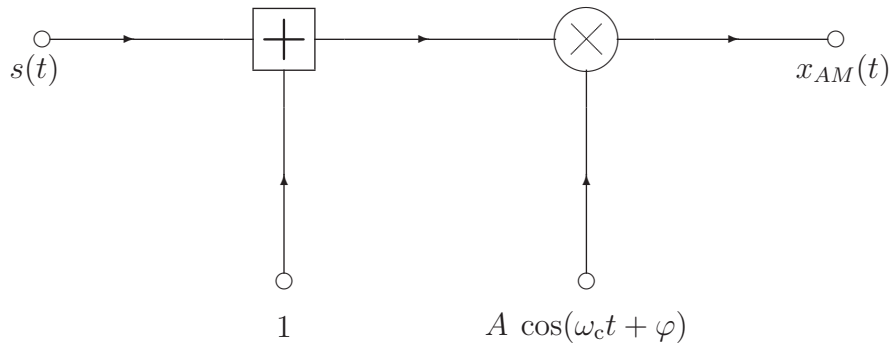


Fig. 5: System for Generating AM-modulated Signals

7.2.2 Generating AM Signals with MATLAB

The generation of AM signals using MATLAB is done by solving the following task.

Problem:

Open the MATLAB "m-file editor" and write a script (*m-file*) consisting of the commands answering the following steps. Call the m-file "`AMmod.m`". Use $A = 1$, $m(t) = \cos(\omega_c t)$, and $\omega_c = 3\omega_s$, where ω_s is the cut-off frequency of the input signal.

1. Define a time vector t starting from -10, ending by 10 in steps of 0.01.

2. Generate the input signal $s(t) = \cos(\omega_s t)$ with $\omega_s = 2\pi$.

3. Create a figure showing $s(t)$ vs. t. Limit the axis as follows:

   a) x-axis: -5 to 5 and

   b) y-axis: -2 to 2.

   Use the command `help axis` to get information how to do axis limitation with MATLAB.

4. Create a second figure. Subdivide it into 2 subplots (type `help subplot` to get help).

5. Plot the output signal $x_{AM}(t)$ for $m_{AM} = 0.75$ in the first subplot. Plot the envelopes of $x_{AM}(t)$ into the same subplot. Use different colors.

6. Plot the output signal $x_AM(t)$ for $m_{AM} = 1.75$ in the second subplot. Plot the envelopes of $x_{AM}(t)$ into the same subplot. Use different colors.

7.3 Transmission Channel

For sake of simplicity we consider an ideal transmission channel. Its impulse response is given by

$$h(t) = \delta(t). \tag{7.1}$$

Hence, its output signal $y(t)$ can be written as

$$y_{AM}(t) = x_{AM}(t) * h(t) = x_{AM}(t) * \delta(t) = x_{AM}(t). \tag{7.2}$$

7.4 Additive Noise

The additive noise is usually modeled by a GAUSSian process. Its probability density function (pdf) $f_N(n)$ is therefore given by

$$f_N(n) = \frac{1}{\sqrt{2\pi\sigma_N^2}} \mathrm{e}^{-\frac{(n-\mu_N)^2}{2\sigma_N^2}}. \tag{7.3}$$

μ_N and σ_N^2 are the mean value and the variance, respectively. MATLAB uses an internal generator to generate random variables resulting from such a process. Use the command "`help randn`" to learn more about it.
The impact of the additive noise is usually described by the so-called *signal-to-noise ratio* (SNR) defined as

$$\mathrm{SNR}|_{\mathrm{dB}} = 10 \log_{10} \left(\frac{E_s}{E_n} \right), \tag{7.4}$$

where E_s and E_n are the energy of the input signal and the noise component, respectively, which can be determined from the relations

$$E_s = \int_{-\infty}^{+\infty} s^2(t) \mathrm{d}t \quad \text{and} \quad E_n = \int_{-\infty}^{+\infty} n^2(t) \mathrm{d}t. \tag{7.5}$$

The effect of the noise is considered during the demodulation process.

7.5 Demodulation

The objective of the demodulation is, as mentioned earlier, to extract the source message $s(t)$ from the modulated bandpass signal $x(t)$. Generally, the AM demodulation techniques can be divided into two categories:

- *Coherent* demodulation: Requires the knowledge of the frequency and phase of the transmitted carrier at the receiver.
- *Non-coherent* demodulation: The phase information of the transmitted carrier is not required at the receiver. In contrast, the carrier frequency has only to be known quite imprecisely.

Because of the noise $n(t)$ (see Fig. 1), it is impossible to reconstruct exactly the source signal $s(t)$ and therefore, each demodulator delivers just an estimate $g(t) \sim s(t)$ for the signal $s(t)$. In this Seminar study, two different circuits of demodulation are considered: the so-called DAM- and QADM-demodulators (*Demodulator for AM* signals and *Quadrature Amplitude Demodulaor*).

7.5.1 DAM demodulator

The DAM (Fig. 6) forms a coherent demodulator for AM signals and requires the knowledge of the carrier frequency ω_c as well as its phase φ.

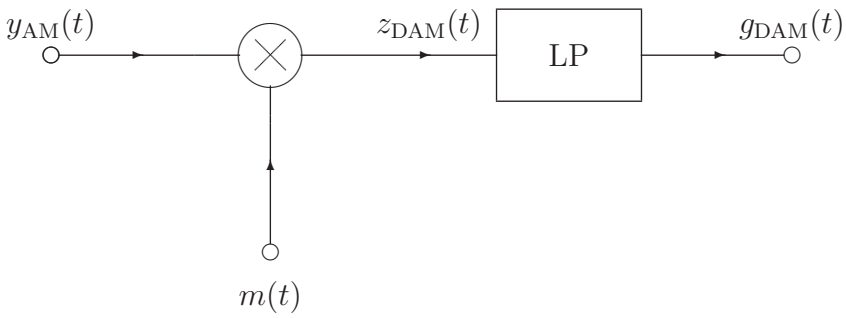


Fig. 6: DAM-demodulator

The DAM can be demonstrated for the case $m(t) = \cos(\omega_c t + \varphi')$. We consider the modulated signal $y_{\mathrm{AM}}(t) = s(t)\ \cos(\omega_c t + \varphi)$ as the input signal. The output signal can be written as

$$\begin{aligned} z_{\mathrm{DAM}}(t) &= y_{\mathrm{AM}}(t)\, m(t) \\ &= s(t)\ \cos(\omega_c t + \varphi)\ \cos(\omega_c t + \varphi'). \end{aligned}$$

After filtering by the lowpass device and some easy calculation steps, the final output signal is determined to

$$g_{\mathrm{DAM}}(t) = \frac{1}{2} s(t)\ \cos(\varphi - \varphi'). \tag{7.6}$$

Problem:

Under considering an ideal transmission channel do the following steps using MATLAB. Extend the m-file "`AMmod.m`" by the commands implementing the DAM-demodulator.

- No phase error, $n(t) = 0$

  1. Generate the signal $m_{\mathrm{dem}}(t) = m(t)$.
  2. Determine the impulse response of the lowpass filter. Assume an ideal lowpass device with the cut-off frequency ω_{LP} and rectangular transfer function of magnitude 1.
  3. Determine the range of ω_{LP} yielding to a correct demodulation.

4. Choose now a suitable value for ω_{LP} and plot the output signal $g_{\mathrm{DAM}}(t)$. What can you conclude?

- Impact of noise and phase error.

1. do the same steps by considering first a phase error $\varphi_{\mathrm{error}} = \varphi - \varphi' = \frac{\pi}{2}$ and plot the output signal $g_{\mathrm{DAM}}(t)$. Interpret the result.
2. Consider now an additive noise component. Study the effect of SNR= -30dB, SNR= 0dB, and SNR= 30dB.

7.5.2 QADM demodulator

The *quadrature AM demodulator* is drawn in Fig. 7.

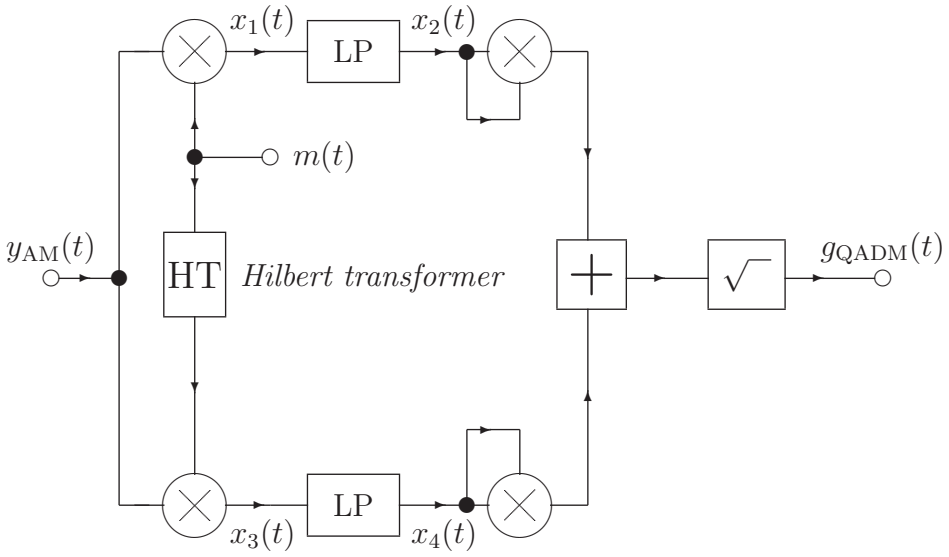


Fig. 7: QADM-demodulator.

Problem:

1. Since the carrier phase φ is *not* required. Show this mathematically by calculating the output signal of the demodulator. Is the QADM a coherent or non-coherent receiver?

2. Extend the file "`AMmod.m`" by the commands implementing the QADM-demodulator.

8 Fourier Transformation

By Batu Chalise

8.1 Introduction

A given signal can be viewed either in time domain or in frequency domain. Time domain representation of a signal is like the trace on an oscilliscope where the vertical deflection is the signal amplitude, and the horizontal deflection is the time variable. Frequency domain representation of a signal is like the trace on a spectrum analyser, where the horizontal deflection is the frequency variable and the vertical deflection is the signal amplitude at that frequency. Fourier Transform (FT) can be used to enter from one domain to another. Depending on the type of processing we want to perform on signals, time or frequency domain representation is used. For example frequency domain representation gives the frequency characteristics like bandwidth of the signal which cannot be exactly known from time domain representation. Moreover, in many cases, processing signals in frequency domain makes the mathematical manipulations much simpler reducing the computational tasks required by hardware.

FT and its discrete derivations have a lot of applications in the fields of communications, speech-processing, astronomy, optics, seismic and many others. It would be near impossible to give examples of all the areas where the FT is involved, because any field of physical science that utilises sinusoidal signals in it's theory, such as engineering, physics, applied mathematics, and chemistry, will make use of Fourier theory and transforms. Now adays, Digital Signal Processing (DSP) is a backbone for almost all types of technology. As most computations are carried out in digital form like by using computers or microprocessors, the discrete version of FT has a significant role.

In this MATLAB seminar, we deal with discrete version of FT which is known as Discrete Fourier Transform (DFT). The computational complexity of DFT is reduced by carrying out Fast Fourier Transform (FFT). Although, there are many methods for performing FFT, the purpose of this seminar is not to deal with their details but to use the MATLAB 'FFT' and 'IFFT' functions for solving a particular problem.

8.2 Background

The Fourier transform of a continuous, aperiodic signal $x(t)$ is given by

$$X(\omega) = \int_{-\infty}^{\infty} x(t)\mathrm{e}^{-j\omega t}\,dt \tag{8.1}$$

and its inverse is

$$x(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} X(\omega)\mathrm{e}^{j\omega t}\, d\omega. \tag{8.2}$$

The continuous signal $x(t)$ is now discretized with sampling period T and the discrete signal $x(nT) \equiv x(n)$ is mathematically expressed as

$$x(n) = \sum_{n=-\infty}^{\infty} x(t)\delta(t - nT). \tag{8.3}$$

The Fourier transform of this discrete signal is

$$F\{x(n)\} = \frac{2\pi}{T}\sum_{n=-\infty}^{\infty} X(\omega - n\frac{2\pi}{T}). \tag{8.4}$$

It can be observed from (8.4), that the FT of $x(nT)$ consists of a periodic repetition of $X(\omega)$ with $\omega_0 = \frac{2\pi}{T}$ as a period in frequency. For example, if $X(\omega)$ is bandlimited with a bandwidth of W and $\omega_0 \geq 2W$, then $X(\omega)$ for $\omega_0 - W \leq \omega \leq \omega_0 + W$ is same as for $2\omega_0 - W \leq \omega \leq 2\omega_0 + W$. Thus the FT of a discrete signal is continuous and is completely described by its $X(\omega)$ defined in a frequency interval of ω_0. The DFT is the discrete version of this $X(\omega)$. If there are N frequency samples within period of $\omega_0 = \frac{2\pi}{T}$, the discrete frequency values are given by

$$\begin{aligned} \omega_k &= k\frac{\omega_0}{N} \\ &= k\frac{2\pi}{NT}, \quad \{k = 0, 1, 2, \cdots N-1\}. \end{aligned} \tag{8.5}$$

Thus with the help of (8.5), the discrete form of (8.1) can be written as

$$\begin{aligned} X(\omega_k) &= \sum_{n=0}^{N-1} x(n)\mathrm{e}^{-j\omega_k nT} \\ X(k\frac{2\pi}{NT}) &= \sum_{n=0}^{N-1} x(n)\mathrm{e}^{-j\frac{2\pi kn}{N}}, \quad \{k = 0, 1, 2, \cdots, N-1\}, \end{aligned} \tag{8.6}$$

whose inverse (IDFT) is given by

$$x(n) = \frac{1}{N}\sum_{k=0}^{N-1} X(k)\mathrm{e}^{j\frac{2\pi kn}{N}}, \quad \{n = 0, 1, 2, \cdots, N-1\}. \tag{8.7}$$

We can see from (8.6) that for the computation of DFT of each of the N samples, we need $(N-1)$ complex additions and N complex multiplications. This means we need a total of $N(N-1)$ complex additions and N^2 complex multiplications. The Fast Fourier Transform (FFT) algorithm provides an efficient procedure for computing the discrete Fourier transform of a finte-duration sequence. To compute the discrete Fourier transform of a sequence of N samples using FFT algorithm, we need, in general, only $N\log_2 N$ complex additions and $(\frac{N}{2})\log_2 N$ complex multiplications. Therefore by using FFT algorithm, the number of arithmetic operations is reduced by a factor of $\frac{N}{\log_2 N}$, which is a considerable saving in computational effort for large N.

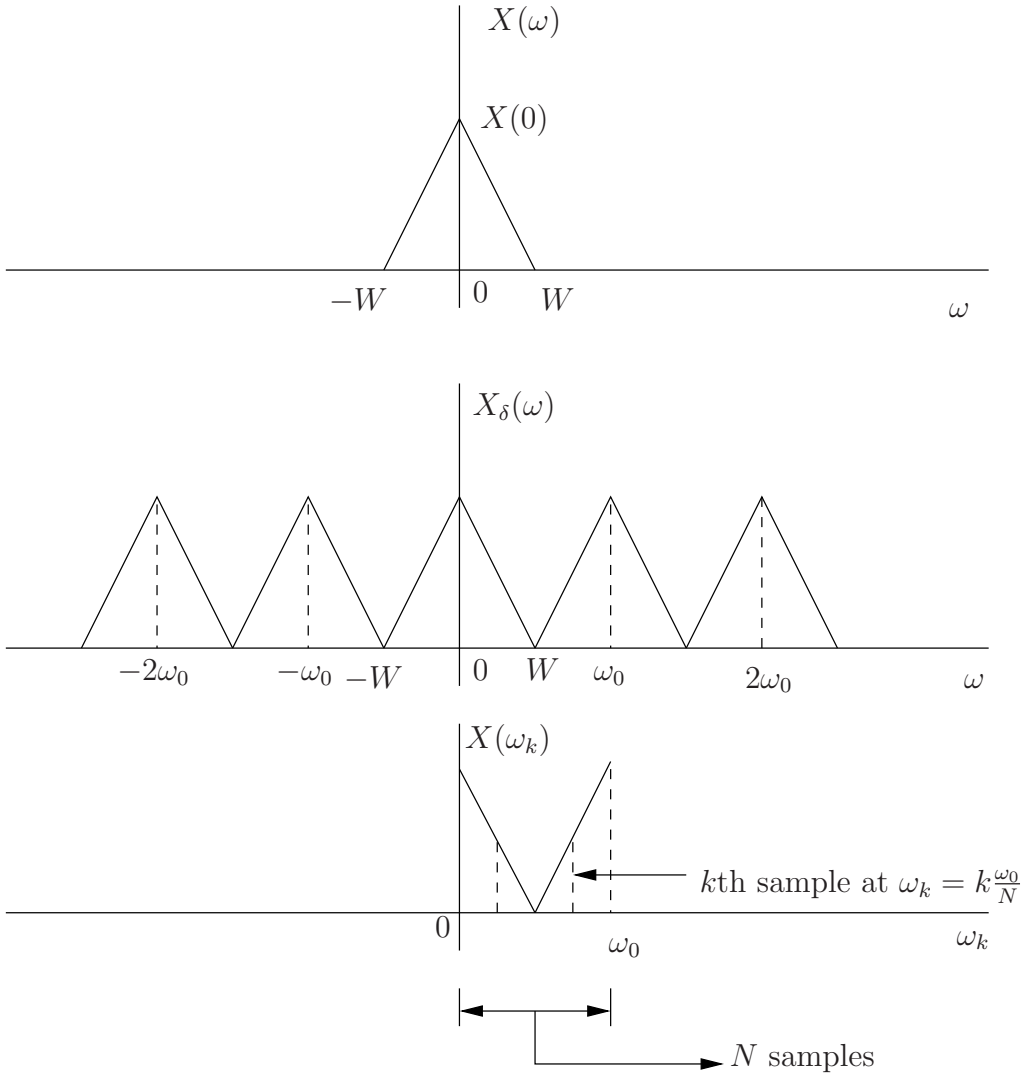


Figure 8.1: Discrete Fourier Transform

8.3 Sample Program

```
%%%%%%%%%%Create a time domain signal%%%%%%%%%%%%

N = 1024;    % Number of samples
ta = 10e-4;  % Sampling time in secs
t = linspace(0, N*ta, N);  %Create a time vector of duration N*ta
w=16;   %Scaling factor for choosing width of the pulse
cp=0;   %Center of the pulse
r=rect(t,N*ta/w,cp); %Generate a rectangular pulse

%%%%%%%%%%%%%%%%%%Plot the rectangular pulse%%%%%%%%%%%%%%
figure(1);       %Create a figure
plot(t,r);        %Plot it
xlabel('t/s -->'); %Label x-axis
```

```
ylabel('r(t) -->'); %Label y-axis
axis([0, N*ta, -0.1 1.1]); %Adjust the range of axes
grid on;                   %Create grids

%%%%%%%Compute FFT of rectangular pulse %%%%%%

R = fft(r); %Fast fourier transform

%%%%%%%%Generate the frequency axis%%%%%%%%%%

fa = 1/ta; % Maximum computed frequency
f = linspace(0, fa, N); %Generate frequency vector

%%%%%%%% Plot Output of Fourier Transform %%%%%%%%

figure(2);          %Figure number 2
plot(f, abs(R));    %Plot amplitude versus frequency
xlabel('f in [Hz] -->');  %x-axis labeling
ylabel('R(f) -->');       %y-axis labeling
grid on;                  %Create grids

%%%%%%%%%% Shifting of the spectrum in Frequency domain%%%%%%%%%%

fs = f - fa/2;  %Subtracting each element of vector f by fa/2
Rs = fftshift(R); %Function of MATLAB that shifts the zero-frequency
                  %component to centre of spectrum

%%%%%%%%%%%%%%%%%%%Output of shifted Fourier Transform%%%%%%%%%%%%%%

figure(3);  %Figure No. 3
plot(fs, abs(Rs)); %Plot amplitude of shifted FT against frequency
xlabel('f in [Hz] -->'); %x-axis label
ylabel('X(f)-->');       %y-axis label
grid on;

%%%%%%%%Return from Frequency Domain to Time Domain%%%%%%%%%%

df = fa / N; %Maximum frequency resolution
T = 1/df; % Maximum Time
t = linspace(0, T, N); %Create a time vector N samples
r = ifft(R);    %Computes the inverse fast Fourier transform

%%%%%%%%Output of inverse FT%%%%%%%%%%%%%%%%%

figure(4); %Figure No. 4
plot(t(1:N), real(r(1:N))); %Plot real part of vector r
```

```
xlabel('t in [s] -->'); %x-axis label
ylabel('r(t) -->');     %y-axis label
axis([0, N*ta, -0.1 1.1]) %Adjust the axes
```

8.4 Assignment

8.4.1 Step 1

Generate the following sinusoid signals using 1024 samples taken at a sampling period of 10 μs.

$$x = \sin(2\pi f_1 t)$$
$$y = \sin(2\pi f_2 t) \tag{8.8}$$

where $f_1 = 1050Hz$ and $f_2 = 2000Hz$. Plot the two signals in a same figure using an appropriate time axis.

8.4.2 Step 2

Compute FFT of the two sinusoids generated in step 1 using MATLAB **fft** function. Using an appropriate frequency axis, plot the amplitude spectrum of the signals in the same figure. Compare the result with the spectrum one can get from theoretical computation of the Fourier transform of the given sinusoids.

8.4.3 Step 3

Use MATLAB **fftshift** function to shift the amplitude spectrum of the signals to origin. Compare the spectrum with that obtained from theoretical Fourier Transform.

8.4.4 Step 4

What is the maximum frequency resolution in (2)? Use this information to find the maximum duration of signal in time. Generate an appropriate time axis and plot the time domain signals using **ifft** function of the MATLAB. Compare the results with that of signals of (1).

9 Convolution

By Rainer Siebel

9.1 Introduction

Convolution is one of the most important operations in linear systems theory and other subjects of modern communication theory. Thus, every student must be familiar with the convolution procedure and must be in the position to carry out the convolution procedure for all types of functions. Particularly for functions, which are non-zero only within a limited definition range and zero elsewhere, the convolution procedure requires to distinguish different validity ranges (with different upper/lower limits) for the convolution integral.
Although MATLAB provides the "conv" procedure which carries out the convolution of two functions with one command it is advantageous to visualize the whole convolution procedure.
Visualization as usual helps to understand, in particular how to find the different validity ranges and corresponding limits for the convolution integral.
The MATLAB demonstration program given with this script serves for this purpose and visualizes the convolution procedure for each time step for two time limited functions.
However, the aim of this MATLAB-seminar is to learn the MATLAB-syntax and to do things ourself. Hence, we shall start with a few problems related to the convolution integral

$$y(t) \;=\; x(t) * h(t) = \int_{-\infty}^{\infty} x(t-\tau)\cdot h(\tau)\,\mathrm{d}\tau = \int_{-\infty}^{\infty} x(\tau)\cdot h(t-\tau)\,\mathrm{d}\tau.$$

9.2 Problems to be solved

The convolution of two functions $x(t),\ h(t)$ as shown in the above integral requires the interpretation of the functions (either $x(t-\tau),\ h(\tau)$ or $x(\tau),\ h(t-\tau)$) in terms of the integration variable τ. This is no problem for $x(\tau)$ or $h(\tau)$ because these functions look like $x(t)$ and $h(t)$ except that they are plotted versus τ. The sketch of $x(t-\tau)$ versus τ (or $h(t-\tau)$ versus τ) is slightly more complicated. Since t is an independent variable in the convolution integral these functions look different for every time instance t.

1. Generate a row vector x with 100 samples which represents the non-zero part of the function $x(t)$ as for example a linear increasing ramp starting at (0,0) and ending at (99,1).

2. Generate an extended all zero vector x_e which finally shall represent the functions $x(t)$ or $x(\tau)$ or $x(-\tau)$ or $x(t-\tau)$ within a time range from -200 $\leq t,\ \tau \leq$ +300.

3. Generate a linearly increasing vector tau with values from -200 to +299, which later serves in the plot command for the representation of the horizontal time axes t or τ.

4. Create 3 vertically arranged subplots within one figure.

5. Insert the vector x into the vector x_e in such a way, that the result represents either $x(t)$ or $x(\tau)$ in the range from -200 $\leq t,\ \tau \leq$ +300 and plot x_e into the upper subplot as $x(t)$ with all axes scales and labels.

6. Plot into the second diagram the following functions: $x(\tau)$, $x(-\tau)$ as functions of τ with different colors, a legend which explains the meaning of the different functions and correctly labeled axes and scales in the same range as chosen above from -200 $\leq \tau \leq$ +300.

7. Now we want to show the function $x(t-\tau)$ plotted versus τ in the range from -200 $\leq\ \tau \leq$ +300 for all time instances t in the range from -100 $\leq t \leq$ +200.
   Write a "for" loop that illustrates how $x(t-\tau)$ changes with the independent variable t and update for each t in the above mentioned range the sketch in the lower subplot.

8. Mark the actual time instance t using an up-arrow below the horizontal axis.

   Hint: The required command to place a symbol at a certain location in the plot is :

   ```
   text(xposition,yposition,'\uparrow','Fontsize',12, ...
        'HorizontalAlignment','center');
   ```

9. Generate a row vector h with 100 samples and constant height 1 and moreover an extended all-zero row vector h_e in the same range as for the extended row vector x_e.
   Insert the h vector into h_e such that h_e represents a rect-function of length 100 samples starting at $\tau = 0$ in the range from -200 $\leq\ \tau \leq$ +300.
   Insert additionally the sketch of h_e into the loop that displays $x(t-\tau)$ in the lower diagram in such a way that $h(\tau)$ as well as the moving $x(t-\tau)$ is visualized.

10. The convolution integral $y(t) = x(t) * h(t) = \int_{-\infty}^{\infty} x(t-\tau) \cdot h(\tau)\ \mathrm{d}\tau$ calculates the "area" under the product $x(t-\tau) \cdot h(\tau)$.

    Calculate the product of x_e and h_e, display the product vector additionally in the lower diagram and fill the "area" of the product with yellow color.

    Hint: The required "fill with color" command is: fill(τ-vector, product-vector, 'y');

11. Calculate the "area" under the product for each time instance t and insert the result at the correct position for each t into an initially all-zero row vector y of length 500 which represents finally the convolution result $y(t)$ in the range from -200 $\leq\ t \leq$ +300.

12. Insert the convolution the result y into the loop and plot additionally the vector y to the lower diagram with red color.

So far some tasks which are required to illustrate the convolution procedure. The complete m-file is given and commented on the following pages. Note, that there are many different possibilities to carry out these tasks. Maybe you find even more effective methods to solve the problems. However, the results should be the same.

9.3 MATLAB demonstration program visualizing the convolution procedure

This MATLAB program demonstrates the convolution procedure. MATLAB provides one build in command: conv(a,b), which carries out the convolution of two row vectors a,b. The m-file below however visualizes the convolution procedure step by step.
First, all variables are cleared and all plot windows are closed.
The command **scrsz = get(0,'ScreenSize');** command grabs the screen coordinates from the system and the command
figure('Position',[0.5*scrsz(3) 0.5*scrsz(4) 0.5*scrsz(3) 0.9*scrsz(4)]);
adjusts the figure in the right half plane of the screen. (Mind, that this command may give different results for a windows operating system than for a LINUX operating system. Thereafter, the figure is created with two sub figures vertically arranged, the background colors of the figure and the sub figures are adjusted and a grid for each sub figure is created as shown with the following m-file commands.

```
%This program demonstrates the convolution procedure
%progammed by Rainer Siebel
%_____________________________________________________________________
clear all                                   %=Clear all variables
close all                                   %=close all windows
scrsz = get(0,'ScreenSize');                %get sreensize
figure('Position',[0.5*scrsz(3) 0.5*scrsz(4) 0.5*scrsz(3) 0.9*scrsz(4)]);
%Posititon of the window on the screen
figure(1);
grid on;
clear all;
subplot(211);
set(gca,'Xcolor','k')%[1 1 1])
set(gca,'Ycolor','k')%[1 1 1])
set(gca,'Color',[0.886 0.886 0.922])%[0 0 0])
% This is the background color of the upper sub-window
grid on;
box on
hold on;
subplot(212);
set(gca,'Xcolor','k')%[1 1 1])
```

```
set(gca,'Ycolor','k')%[1 1 1])
set(gca,'Color',[0.886 0.886 0.922])%[0 0 0])
%This is Background color of the lower sub-window
grid on;
box on
hold on;
```

With the next m-file commands we prepare the information lines for the selection of some predefined input (ifun) and impulse response (p_resp) functions. **ifun=...** defines a column vector array of characters. A call of **ifun** displays this array in the command window.
The command **sel_x=input('choose the number for input function: ');** waits for a number to be typed in and assigns the number to the variable **sel_x**.

```
%After we have defined some general screen settings we start here
%with the demonstration program.
%________________________________________________________________
%print this messages to the command window
ifun=['Predefined input-functions                                ';
      '1) linear ramp (100 samples)                              ';
      '2) one period of a sine-function                          ';
      '3) rect function (50 samples)                             ';
      '4) one sided decreasing exponential                       ';
      '5) double rect-function (positive followed by negative)   ';
      '6 linear increasing from -1 to +1                        '];
ifun
sel_x=input('choose the number for input function: ');
```

Similarly, we select the number of the predefined impulse response vector to be used in the convolution procedure in the following lines of the m-file.

```
p_resp=['Predefined impulse responses                              ';
        '1) rect function (70 samples)                            ';
        '2) delayed rect function (25 samples delay 125 duration';
        '3) linear increasing from -1 to +1                       ';
        '4) one period of a cosine-function                      ';
        '5) triangle (duration 100 samples)                       ';
        '6) matched filter for chosen input function             '];
p_resp
sel_h=input('choose the number for impulse response: ');
```

Then we define the corresponding vectors for the input functions and impulse responses. With the if command only those vectors are created, which are selected according to the chosen numbers sel_x and sel_h.

```
subplot(211);       %first select upper sub-window
```

```
%define vector with 100 elements for the independent variable
t=(0:1:99);

%define input function of ~100 samples length
%a few predefined functions are given below

if sel_x==1; x=t/99; end;    %linearly increasing function from 0 ..1
if sel_x==2; x=sin(2*pi*t/99); end; %one period of a sine function
if sel_x==3; x=[linspace(1,1,50)]; end; %const. vector with 50 elements
if sel_x==4; x=exp(-0.04*t);  end;
 %exponentially decreasing vector with 100 elements
if sel_x==5; x=[linspace(1,1,50),linspace(-1,-1,50)]; end;
      %two rects with amplidude 1 and -1
if sel_x==6; x=linspace(-1,1,100); end;
      %linearly increasing vector from -1 to +1 with 100 elements

%define impulse response of max 150 samples length
%a few predefined functions are given below
%uncomment your choice or define some more impulse responses
if sel_h==1; h=linspace(1,1,70); end;
if sel_h==2; h=[linspace(0,0,25),linspace(1,1,125)];  end;
if sel_h==3; h=linspace(-1,1,100); end;
if sel_h==4; h=cos(2*pi*t/99); end;
if sel_h==5; h=[linspace(0,1,50),linspace(1,0,50)]; end;
if sel_h==6; h=fliplr(x); end;
%_____________________________________________________________
```

Item 6 for the impulse response just flips the input vector. Thus, the convolution result would show the response of a "matched filter" with impulse response $h(t) = x(T - t)$, with T the total length of the input signal. Hence, if we want to visualize the output of a matched filter for any of the predefined input signals $x(t)$, we just choose one of the input signals and choose for the impulse response option 6.
With the aim to show a longer time interval than only the length of the created input and impulse response vectors we extend the "time axis" from -200 to +299 time instances, i.e. a total of 500 time instances. For the extended vector x_e we always start with 200 zeros, append the input vector and fill the remaining time instances with zeros as shown below with the command: **x_e=[linspace(0,0,200),x,linspace(0,0,300-size(x,2))];**
Then we plot the input vector versus the horizontal time axis in red color, create a title and a legend and label the horizontal axis.

```
%from here downward don't change any line

%define the scale tau
tau=(-200:1:299);

%extend the x-vector length by zero-valued intervals
```

```
x_e=[linspace(0,0,200),x,linspace(0,0,300-size(x,2))];
plot(tau,x_e,'r','LineWidth',2);
 %plot the extended x-vector over tau in red color
grid on;
axis([-200 300 -1.5 2]);
 %define the limits of the horizontal and vertical range
title('input signal x(t)','FontWeight','Bold');  %add a title line
ht=legend('x(t)');                    %include the legend
set(ht,'Color',[1 1 1])               % and its color
xlabel('t\rightarrow');               % set xlabel
display('press any key to continue');
 %print this message to the command window
pause
hold on;
```

The same procedure is used to plot the impulse response $h(t)$ to the upper subteen in blue color.

```
%extend the h-vector length by zero-valued intervals
h_e=[linspace(0,0,201),h,linspace(0,0,299-size(h,2))];
plot(tau,h_e,'b','LineWidth',2);
title('input signal x(t) (red) and impulse response h(t) (blue)',...
 ...'FontWeight','Bold');
ht=legend('x(t)','h(t)');
set(ht,'Color',[1 1 1])
xlabel('t\rightarrow');
pause
```

Finally, we plot the function $x(-t)$ in black color and dashed lines using the fliplr() command to reverse the order of elements in the vector x. The title line and legend is always updated.

```
%plot x(-t)
x_inv=[linspace(0,0,201-size(x,2)),fliplr(x),linspace(0,0,299)];
plot(tau,x_inv,'--k');
title('input signal x(t) (red) and impulse response h(t) (blue) and ...
                  x(-t)=mirrored x(t) (black)','FontWeight','Bold');
ht=legend('x(t)','h(t)','x(-t)');
set(ht,'Color',[1 1 1])
xlabel('t\rightarrow');
pause;
%hold off;
```

Now we have displayed in the upper sub screen all functions to be used in the convolution procedure $x(t) * h(t) = \int_{-\infty}^{\infty} x(t-\tau) \cdot h(\tau) \mathrm{d}\tau$.
In the next m-file lines we interpret the convolution integral.
First we define a zero vector y, which finally holds the convolution result and is updated

by one new value for each run through the loop. $x(t-\tau)$ and $h(\tau)$ must be plotted versus the integration variable τ, which means that $h(\tau)$ looks like $h(t)$ except that we plot it versus τ and $h(t-\tau)$ looks like a shifted replica of $h(-t)$ when plotted versus τ. We start with a negative shift t of 100 time units and shift t by one time instance to the right in each run through the loop **for k=-100:1:298;**. The x_shift vector is created new for each time shift k in the loop by inserting the flipped x-vector one time instance to the right for each run through the loop.
With the command **y(201+k)=0.04*x_shift*h_e';** the $\sum_{i=1}^{500} \text{x\_shift}(i) \cdot \text{h\_e}(i)$ of the two vectors x_shift and h_e is calculated and multiplied by an arbitrarily chosen factor 0.04 to fit into the vertical range of the lower sub screen. Note, this is done by matrix multiplication of the row-vector x_shift with the column-vector h_e'. Then the extended vector h_e is plotted in blue color, the x_shift vector is plotted to the same sub screen with black dashed lines. With the command **fill(tau,x_shift.*h_e,'y');**, the "point product" of h_e and x_shift is build, displayed with a thin black line and filled everywhere with yellow color where the "point product" is nonzero. Since $x(t) * h(t) = \int_{-\infty}^{\infty} x(t-\tau) \cdot h(\tau)\mathrm{d}\tau$ equals the area between the horizontal axis and the function given by the product $x(t-\tau) \cdot h(\tau)$, the total yellow area is proportional to the convolution product. Note that those parts of the "area" are are counted as negative contributions, if they are below the horizontal axis!
Further the actual time $t = \tau$, i.e. the actual t which corresponds to the value on the τ-axis is displayed and the legend is updated. For predetermined time instances k the procedure is stopped.

```
%subplot(212);
%define output-vector
y=linspace(0,0,500);

for k=-100:1:298;
%k=50;
x_shift=[linspace(0,0,201+k-size(x,2)),fliplr(x),linspace(0,0,299-k)];
y(201+k)=0.04*x_shift*h_e';
subplot(212);
plot(tau,h_e,'b','LineWidth',2);
axis([-200 300  -1.5 2.5]);
hold on;
plot(tau,x_shift,'--k','LineWidth',2);
fill(tau,x_shift.*h_e,'y');
plot(tau,y,'r','LineWidth',2);
title('Convolution process for increasing t','FontWeight','bold');
text(-180,-1.3,'yellow filled area is proportional to the output signal');
grid;
text(k,-0.2,'\uparrow','Fontsize',12,'HorizontalAlignment','center');
text(k,-0.2,' t=\tau','Fontsize',12);

legend('h(\tau)','x(t-\tau)','   0.04\cdot y(t)', ...
          '\int_{-\infty}^{\infty} x(t-\tau)\cdot h(\tau) d\tau=y(t)',2);
```

```
xlabel('\tau\rightarrow');
set(gca,'Xcolor','k')%[1 1 1])
set(gca,'Ycolor','k')%[1 1 1])
set(gca,'Color',[0.886 0.886 0.922])%[0 0 0])
grid on;
pause(0.05);
if (k==-50|k==0|k==130|k==50|k==110|k==160|k==230)
   pause;
end;
%pause;
hold off;
end;
pause;
clf;
```

The whole procedure was programmed with the aim to visualize the convolution procedure step by step. However, if we are just interested in the convolution result, the only thing we have to do is to use the build in MATLAB function **conv(h,x)** as shown below.

```
%And here the short Matlab version with the same result.
h=[linspace(0,0,1),h,linspace(0,0,20)];
 %here we just add some zeros to the h-vector
y=0.04*conv(h,x);
plot(y,'m','LineWidth',2);
axis([-200 300 1.2*min(y) 1.2*max(y)]);
grid;
pause;
clf;
```