TSKS10 Signals, Information & Communication
TSKS21 Signals, Information & Images
TSKS11 Networks: Models, Algorithms and Applications
TSDT14 Signal theory

Short Matlab Manual

Mikael Olofsson

INSTITUTE OF TECHNOLOGY
LINKÖPING UNIVERSITY

Department of Electrical Engineering (ISY)
Linköping University, SE-581 83 Linköping

Linköping August 2018

About this document:
This manual is intended for the engineering courses TSKS10 Signals, Information & Communication, TSKS11 Networks: Models, Algorithms and Applications TSKS21 Signals, Information & Images and TSDT14 Signal Theory, given for third and fourth year engineering students at Linköping University.

Differences compared to the August 2017 version:
Close to none. A few misprints have been fixed.

Acknowledgements:
Special thanks to my colleague Lasse Alfredsson, who inspired me to implement the well-known bubblesort algorithm, that I have used as an example here. I would also like to express my gratitude to Professor Erik G. Larsson as well as to Doctors Johannes Lindblom and Mirsad Cirkic for spotting a number of misprints. They have thus helped improving the presentation.

Linköping August 2018
Mikael Olofsson

Short Matlab Manual

Department of Electrical Engineering (ISY)
Linköping University
SE-581 83 Linköping

This document is written in LaTeX2$_\varepsilon$.

Contents

1 Introduction

MatLab is a tool for numerical calculations. The basic data type in MatLab are matrices, that can be both real and complex valued. Column vectors are matrices with only one column and row vectors are matrices with only one row. MatLab offers an interactive command-line user interface where built-in functions can be used, as well as functions that you have developed yourselves.

MatLab has support for presenting matrices and vectors in various ways. Graphs and 3D-surfaces are very simple to generate, and several curves can be plotted in the same graph. There are various toolboxes available for Matlab, that has to be purchased individually. We will use parts of the wide-spread *Signal Processing Toolbox*. Some of the functions in Signal Processing Toolbox are documented under *Functions in Signal Processing Toolbox* on page 5.

This document is far from a complete documentation of the functionality in MatLab and Signal Processing Toolbox, but it should be enough to solve the tasks in the laborations.

2 Help in MatLab

MatLab has on-line-documentation for most functions. There you can get more elaborate explanations than the short explanations given here. You should read that information for the functions that you are interested in. It can be reached in several ways.

MatLab has a traditional help function that can be reached through the menu choice Help/Documentation. The information that you can find there is overwhelming, but the information that is relevant here is under MatLab and Signal Processing Toolbox.

The On-line-help can also be found using the command **help <function name>**. If the help-text is too long for your Matlab window, you can activate a "paging" function with the command **more on**. Using the space bar and the return key, you can step through the text. The key 'q' skips the rest of the text.

An alternative to **help** is **lookfor <word>** which lists all functions associated to **<word>**. It is also possible to get help about other things than functions. E.g. **help signal** lists all functions in Signal Processing Toolbox together with a short description of them. Simply typing **help** results in a list of function groups (including toolboxes) for which there is further help available.

3 Expressions in MatLab

Commands given to MatLab are almost always an assignment or a command such as **print**, **plot** or **hold**. An assignment is expressed as

```
>> a = <expression>;
```

where <**expression**> is a combination of matrices, functions and operands. Notice the semi-colon (;) at the end. It suppresses printing of the result. If the result is a large matrix or vector, you would probably only be annoyed by the printing of the result to the screen. You can always stop a printing using **CTRL-c**. The command has already been performed when the printing starts.

Some functions in MatLab can generate several return values. An assignment to two variables has the following form:

```
>> [a,b] = <expression>;
```

An example of a function that returns two values is **hist**.

Here are some basic examples and expressions that can be used in MatLab:

i (alternatively **j**)
: The imaginary unit.

a = [1 2;3 4]
: Gives you the matrix $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$.

b = [a a]
c = [a;a]
: Gives you the matrices $\mathbf{b} = \begin{pmatrix} 1 & 2 & 1 & 2 \\ 3 & 4 & 3 & 4 \end{pmatrix}$ and $\mathbf{c} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 1 & 2 \\ 3 & 4 \end{pmatrix}$.

a = eye(n)
b = zeros(m,n)
c = ones(m,n)
: Here, **a** becomes an $\mathbf{n} \times \mathbf{n}$ unit matrix, **b** becomes an $\mathbf{m} \times \mathbf{n}$ matrix with all zeros, and **c** becomes an $\mathbf{m} \times \mathbf{n}$ matrix with all ones.

a = m:n
b = m:k:n
c = [1 2 zeros(1,3)]
: This gives you the vectors $\mathbf{a} = [\mathbf{m}, \mathbf{m{+}1}, \mathbf{m{+}2}, \ldots, \mathbf{n}]$, $\mathbf{b} = [\mathbf{m}, \mathbf{m{+}k}, \mathbf{m{+}2k}, \ldots, \mathbf{n}]$ and $\mathbf{c} = [\mathbf{1}, \mathbf{2}, \mathbf{0}, \mathbf{0}, \mathbf{0}]$.

b = a(m,n), c = a(n), d = a(m:n)
Here **b** will become the element in row **m** and column **n** of the matrix **a**. And **c** will be the **n**-th element in the vector or matrix **a**. In the case where **a** is a matrix, the elements are counted column by column. E.g. if **a** is a 3×4 matrix, then **a(8)** and **a(2,3)** refer to the same element. Finally, **d** is a row vector containing all elements from the **m**-th element up to and including the **n**-th element of the row vector or matrix **a**. If **a** is a column vector, then **a(m,n)** returns a column vector.

b = a(m:k:n)
The resulting **b** is a row vector containing every **k**-th element of the vector or matrix **a**, starting with the **m**-th element. The last element number does not exceed **n**. In case **a** is a column vector, then so is **b**. This can be used to downsample a signal.

b = a(m,:), c = a(:,n), d = a(k:l,m:n)
Result: **b** is row number **m** of the matrix **a**, **c** is column number **n** of the matrix **a** and **d** is a submatrix of **a** formed of the elements in rows **k** through **l** and columns **m** through **n**.

b = a(m1:k1:n1,m2:k2:n2)
Result: **b** is submatrix of the matrix **a**, consisting of every **k1**-th row from row **m1** to row **n1**, and every **k2**-th column from column **m2** to column **n2**.

a([1 3],:) = []
Modifies the matrix **a** by removing rows number 1 and 3.

c = a+b, c = a*b, c = a/b, c = a\b
Matrix addition, multiplication, left division (solves **cb** = **a** if possible) and right division (solves **bc** = **a** if possible).

a′
Hermitian transpose of a vector or matrix **a**, i.e. transpose *and* complex conjugate.

a.′
Ordinary transpose of the vector or matrix **a**, without any complex conjugate.

flipud(a), fliplr(a)
Returns the matrix **a** flipped upside down and left-right, respectively.

size(a)
Returns the vector $[\mathbf{m}, \mathbf{n}]$, where **m** is the number of rows and where **n** is the number of columns in **a**.

c = a.*b, c = a.^2
Element-wise multiplication and squaring, respectively. All arithmetic operations can be made element-wise using this dot notation. Operations that normally operate element-wise (e.g. +, −) do not accept the **.**-notation.

c = a>n
Results in a matrix **c** of the same size as **a**, with ones in the positions where **a**:s element is greater than **n**. Other comparison operations can also be used this way.

a = inv(b)
Matrix inversion. Demands of course that **b** is nonsingular (i.e. invertible).

4 Standard Functions and Commands

sqrt(a), real(a), imag(a), abs(a), angle(a)

sin(a), cos(a), tan(a), cot(a)

log(a), exp(a), round(a)

Those functions operate element-wise on matrices, and they do exactly what you expect them to do. Possibly, it should be noted that angles are interpreted in radians and that **angle(a)** returns the argument of the complex number **a**.

rand(m,n), randn(m,n)

Returns a realization of an $\mathbf{m} \times \mathbf{n}$ matrix of uniformly/Gaussian distributed independent random numbers. The uniform distribution is between 0 and 1, while the Gaussian distribution is N(0,1).

conv(a,b)

Returns the convolution of the vectors **a** and **b**. If **a** has length **n** and **b** has length **m**, then the result has length $\mathbf{n} + \mathbf{m} - \mathbf{1}$. The function accepts both row and column vectors. The returned vector has the same orientation as the second argument. This can be interpreted as time-discrete convolution of signals represented as vectors, but also as polynomial multiplication, where the vectors contain the coefficients of the polynomials in order starting with the constant term.

save filename var1 var2...

load filename

delete filename

Save the variables **var1**, **var2**,... in a file, load variables from a file, and delete a file, respectively. The command **save filename** without a variable list saves all variables presently in your workspace.

who, whos

List all variables in your workspace. **whos** prints out more information about the variables.

cd, pwd, ls

Works as the UNIX-commands with the same name, i.e. change directory, print working directory, and list working directory, respectively.

clear all

Clear memory, i.e. remove all variables from workspace.

soundsc(x,fs)

Send vector **x** to speakers, using sample rate **fs**, where **fs** should be between 5000 and 48000.

5 Functions in Signal Processing Toolbox

X = fft(x,n), y = ifft(Y,n)
Calculates an **n** point DFT and inverse DFT, respectively. Since the indexing in MatLab starts with 1, **X(1)** corresponds to the constant term. If **n** is less than the sequence length, then the sequence is truncated. If **n** is greater than the sequence length, then the sequence is padded with zeros. If **n** is omitted, then the (I)DFT is calculated using the actual sequence length.

y = fftshift(x), y = ifftshift(x)
Cyclic shifts of vectors. **fftshift** rearranges the outputs of **fft** by moving the zero-frequency component to the center of the vector. Notice that **fftshift** is not its own inverse. Instead, **ifftshift** is the inverse of **fftshift**. So, **ifftshift(fftshift(x))** returns the vector **x**, while **fftshift(fftshift(x))** does not.

xcorr(a,b)
Returns the cross-correlation of the vectors **a** and **b**. The function accepts both row and column vectors. The returned vector has the same orientation as the first argument. This function can also be called with only one argument, in which case the return value is the auto correlation of the vector. *Notice that no normalization takes place here. This is not the Bartlett or Blackman-Tukey estimate of the ACF.*

y = filter(b,a,x)
Filters the signal **x** through the filter

$$H[z] = \frac{\mathbf{b(1)} + \mathbf{b(2)}z^{-1} + \ldots + \mathbf{b(n)}z^{-\mathbf{n}-1}}{\mathbf{a(1)} + \mathbf{a(2)}z^{-1} + \ldots + \mathbf{a(m)}z^{-\mathbf{m}-1}},$$

where **n** is the length of **b** and **m** is the length of **a**. The result **y** is truncated to the same length as **x**. You can make the result longer by padding **x** with zeros.

[b,a] = butter(n,W)

[b,a] = cheby1(n,R,W)

[b,a] = cheby2(n,W)

[b,a] = fir1(n,W)
Generates filter coefficients for a filter of degree **n** and normalized cut-off frequency **W**, and in relevant cases ripple **R** in dB. The resulting vectors **a** and **b** can be used in the command **filter** above.
Observe that MatLabs opinion about normalized frequencies is that 1 corresponds to half the sample frequency, and not the sample frequency as we are used to.

hamming(n), hanning(n)

blackman(n), boxcar(n)
Creates window functions. The last one creates a rectangular window.

6 Graphs and Figures

plot(y), plot(x,y), plot(x1,y1,'-',x2,y2,'.',...)
Basic plot function. For more information, see the online-help.

stem(y), stem(x,y)
Make stem plots, i.e. plot time-discrete signals as we usually do. For more information, see the online-help.

hist(a,n), [f,d] = hist(a,n)
Calculates and plots a histogram for columns in **a**. The parameter **n** is the number of intervals to use. The first form plots the histogram, while the second form calculates the histogram and returns the vectors **f** and **d**. **f(k)** is the number of hits in interval number **k**, while **d(k)** is the center of interval number **k**. You can also let **n** be a sorted vector that indicates the centers of the intervals. The two outer intervals will then be infinitely large.

clf
Clear the current figure.

hold on, hold off
Plotting commands like **plot**, **stem** and **hist** normally erase the current graph before a new graph is drawn. The command **hold on** disables this default behaviour and makes the current graph stay. This makes it possible to create complicated graphs using several plot commands. The command **hold off** reinstates the default behaviour.

subplot(m,n,p)
This command splits the current figure window into a rectangular grid of **m** rows and **n** columns, and indicates that the next **plot** command will be drawn in the p-th rectangle, counted row by row.

grid on, grid off
Turns on and off the grid in the current graph.

title('text'), xlabel('text'), ylabel('text')
Add title and labels on the axes in the current graph.

axis([xmin xmax ymin ymax]), axis('auto')
Control the grading of the axes in the current graph. The first version assigns minimum and maximum values for the axes, while the second version turns on automatic scaling. The deault behaviour is automatic scaling of axes.

plottools on, plottools off
Turn on and off editing tools for the current graph. When you have created a graph, you may want to adjust how it is presented. Do this before printing. See below.

print -deps filename.eps, print -depsc filename.eps
Print the current graph to the EPS file filename.eps. The first version is a gray-scale image, while the second is a color image.

7 Generating Noise

In the laborations in TSDT14 Signal Theory, you will have to generate needed signals yourselves. It is then suitable to start with noise and filter that. MatLab has a built-in function **rand** that returns a realization of a matrix of uniformly distributed random variables. There is also a corresponding function **randn** that generates Gaussian distributed matrices. For example,

x=randn(2^16,1);

creates a realization of a column vector consisting of 2^{16} samples of independent Gaussian variables with mean 0 and variance 1. Note that the last one in the command does not indicate the variance, but that the number of columns in the matrix is one.

Unfortunately, it is hard to generate true randomness in a computer. Instead, pseudonoise is used, which is often generated using binary feedback shift registers. Those shift registers are generally very long, and can easily be implemented in software. Sequences from such shift registers look very much like realizations from a uniformly distributed noise-process.

8 Working with Graphs

In the most simple case, when you are plotting a curve in Matlab, you simply get the indices of the samples along the horizontal axis. To get a nice scale on the horizontal axis, when for instance plotting a spectrum, you can plot the vector against another vector. When the horizontal axis is supposed to correspond to normalized frequencies, that vector can be **(0:n-1)/n**. The variable **n** is here assumed to be the number of points in the spectrum. E.g.: The command **plot((0:99)/100,cos((0:99)/100*pi).^2)** plots a graph that could be the power spectral density of the output from a low-pass filter.

Remember that the Fourier transform generally produces a complex valued result. A power spectral density is a special case, which is real and non-negative. However, since numbers have a finite representation in Matlab, there can still be small numeric residues in the imaginary part, that we know should not be there. Then the functions **abs** and **real** can come in handy to remove any numeric residues in the imaginary part.

9 Writing Scripts and Functions

You will most certainly want to do several similar calculations, that finally result in graphs. Therefore, it is a good idea to start writing scripts and functions early. Scripts for the easy cases and functions for the more complex cases, where you call the functions with different arguments to achieve whatever it is that you want to achieve. This will especially hold for the spectral estimation in TSDT14.

A script is simply a file containing a number of Matlab commands. It must have extension **.m**, live in your current directory, and it is executed by typing its name – without the extension. The commands in the file are then executed as if you had typed them in at the Matlab prompt. All variables used in the script are global variables in your workspace.

Assume that we want to write a simple script that calculates the area of a rectangle, given its length and width, and we want to call that script **rectangle**. Then the filename should be **rectangle.m**, and the contents of that file could be

```
% RECTANGLE    Area of a rectangle.
%
%    This script calculates the area of a rectangle with sides A and B,
%    where A and B are assumed to be global variables, i.e. A*B. The
%    resulting area is placed in the global variable Area.

Area = A*B;
```

Rows starting with `%` are comments, and initial rows that are comments also serve as documentation of the script. Those rows are displayed if you type **help rectangle**. The last line calculates the area and places the result in the global variable **Area**.

To use this script to calculate the area of a rectangle with sides 5 and 7, we type the following at the Matlab prompt.

```
A = 5;
B = 7;
rectangle
```

After that, the global variable **Area** exists and has the value 35.

A function is also a file with extension **.m**, containing Matlab commands. There are at least three differences between a script and a function. The function has a local workspace, where it has its own (local) variables. The function can take arguments as input, and it can return values as its output. A script does not have or do any of that. The visual difference is that a file that specifies a function starts with a special line. The following lines of that file then calculate the return value(s).

Let us turn our example script into a function, and let us keep the name **rectangle**. Then the filename should still be **rectangle.m**, and the contents of that file could be

```
function Area = rectangle(a,b)
% RECTANGLE    Area of a rectangle.
%    RECTANGLE(A,B) returns the area of a rectangle with sides A and B.
%
%    This function returns the area of a rectangle with sides A and B,
%    i.e. A*B.

Area = a*b;
```

This example shows a few things that are found in most function files. The first line indicates that the file defines a function, that the name of that function is **rectangle**, and that it takes (at most) two arguments with local names **a** and **b**, and returns one value, which has local name **Area**. The following five comment lines is the documentation of the function. The last line is the calculation of the return value. Note that all variables in the function are local variables, and do not affect the globals workspace.

The documentation follows the standard that is recommended for documenting Matlab functions. First the name of the function with a very short description of what it does. Then a row that specifies its arguments and its return values. Last a more detailed description of what the function does. In this simple case, all those rows may look a bit silly, since they more or less repeat the same thing over and over again, and we could very well skip the last part of the documentation. Real-life functions are normally a lot more complex, and then this standard makes a lot more sense.

To use this function to calculate the area of a rectangle with sides 5 and 7, we type the following at the Matlab prompt.

```
A = rectangle(5,7)
```

After that, the global variable **A** exists and has the value 35.

Let us adjust our function so that it calculates not only the area of the rectangle, but also the circumference of it. Let us also allow the function to accept both one and two arguments, where one argument is to be interpreted as the side of a square.

```
function [Area,Circ] = rectangle(a,b)
% RECTANGLE    The area and circumference of a rectangle or a square.
%    RECTANGLE(A) returns the area and circumference of a square with
%    side A.
%    RECTANGLE(A,B) returns the area and circumference of a rectangle
%    with sides A and B.
%
%    This function returns the area and circumference of a rectangle
%    with sides A and B, i.e., A*B and 2*(A+B). The argument B can be
%    omitted, and is then set to A, i.e. we are then calculating the
%    area and circumference of a square with side A.

if nargin==1
    b = a;
end

Area = a*b
Circ = 2*(a+b)
```

This example shows a few more things. First, we can note the syntax for more than one return value in the first row. Secondly, we let the documentation reflect the new behaviour, and especially we indicate both allowed ways to call the function. Then – in the code – there is the handling of a variable number of arguments. The variable **nargin** is an integer that indicates how many arguments are actually passed to the function when it is invoked. Last we can note that the different return values can be calculated separately in the function.

To use this function to calculate the area and circumference of a rectangle with sides 5 and 7, and for a square with side 6, we type the following at the Matlab prompt.

```
[A1,C1] = rectangle(5,7)
[A2,C2] = rectangle(6)
```

After that, the global variables **A1**, **A2**, **C1** and **C2** exist and have the values **A1=35**, **A2=36**, **C1=24** and **C2=24**.

Finally, a more complex example of a Matlab function:

```
function S = bubblesort(V,d)
% BUBBLESORT    Sort a vector.
%    BUBBLESORT(V) returns a sorted version of the vector V.
%    BUBBLESORT(V,D) returns a sorted version of the vector V.
%
%    This function sorts the vector V using the well-known sorting
%    algorithm called bubblesort. It also displays the propagation of
%    the sorting in a bar-chart. The optional argument D indicates a
%    delay in seconds between each step in the algorithm, which can
%    be used to affect your experience of the graphics. Suitable
%    values of D are probably between zero and one. If D is omitted,
%    it is set to zero.

if nargin==1
    d=0;
end
S = V;
N = length(V);
for pos = N:-1:2
    for k = 1:pos-1
        S(k:k+1)=[min(S(k:k+1)) max(S(k:k+1))];
        S2=[S;zeros(2,N)];
        if pos<N
            S2(1,pos+1:N)=0;
            S2(2,pos+1:N)=S(pos+1:N);
        end
        S2(1,k:k+1)=[0 0];
        S2(3,k:k+1)=S(k:k+1);
        bar(1:N,S2','stack'), drawnow
        pause(d)
    end
end
S2(3,1:2)=[0 0];
S2(2,1:2)=S(1:2);
bar(1:N,S2','stack')
```