Vrije Universiteit Brussel

FACULTEIT INGENIEURSWETENSCHAPPEN

Practica Digital Signal Processing 2015-2016

prof. W. Verhelst, ir. H. Brouckxon, ir. T. Dekens
ir. M. Demol and dr.ir. W. Mattheyses

Vrije Universiteit Brussel
http://www.etro.vub.ac.be/Research/DSSP/

November 3, 2015

Contents

List of Figures

1 Introduction to Matlab; Sampling

1.1 Matlab

1.1.1 General Introduction

Matlab is a frequently used programming language for applied numerical mathematics, engineering design and digital signal processing research and education. In its common use, it operates as an interpreter-based language that works with functions and scripts, but also supports more advanced operations (e.g. object-oriented structures). Especially interesting are its large library of built-in functions that support many common computational problems, its intuitive syntax, and its extensive (graphical) visualisation possibilities. Also, a large number of toolboxes exist that provide extra functionalities for specific technical domains like digital signal processing (filter design, signal processing), image processing and control engineering.

1.1.2 Basic Concepts

In its basic operation, Matlab can be considered as an extensive calculating machine that operates on vectors or (multi-dimensional) matrices of numerical and/or alphanumerical values. Scalars, i.e. single numerical values, can of course also be used, and are considered as $1x1$ matrices. Matrices/vectors can be stored in memory as variables that are identified by meaningful names (chosen by the programmer), and can be used to perform (m)any matrix-based operations.

In the next section we'll briefly explore the basic use of Matlab and its matrix operations. The boxed texts in italics are examples of Matlab commands that can be interactively entered in Matlab's commandline window.

Whenever possible, try to enter these commands to get used to their operation.

Input and output of data; interaction with the program

Data can be entered interactively or imported from a file. The following commands for example create a new scalar variable (1x1 matrix) x1 which contains a numerical value of 5, and a scalar variable y that gets a value 13:

> *x1=5* (spaces are allowed, so this is equivalent to *x1 = 5*).
> *y = x1*2+3* (new scalar variable y, contains value 13).

As you probably noticed, Matlab always prints the name and value of the result of each operation to the command line. In some cases however (e.g. in long programs with large sequences of operations), it would be preferable to suppress this response for most of the intermediate operations, and only print the results that are most interesting. This is possible using a control symbol, designated by a semicolon **;**. By ending the line of an operation by this symbol, the calculation will still be performed, but the output to the command line window will be suppressed. Try for example the following:

```
x1 = x1/x1;
```

To view the (intermediate) value of a given variable at a certain moment, it is sufficient to enter the name of this variable, without the semicolon control symbol:

```
x1
```

By default, the current value of all the variables that were used within the current session or program is maintained in Matlab's memory space. A list of all these variables can be retrieved using the Matlab commands *who* or *whos*. To free memory space from variables and values that will no longer be used, the command *clear* can be used. In itself (without any parameters) this command deletes all variables in the memory space, but it can also be used to delete only specific variables by specifying these as parameters, e.g. *clear x1*

To simplify specific classes of calculations, some variable names are predefined to contain a default value. Examples of this are:

pi equals 3.14159265...

i or j equal the square root of -1, and can be used to represent complex numbers.

It is however important to note that these variables can in theory be given a new value by your program, so be very careful if you want to use them!

```
z = pi + 5 i
```

In the same way as with the scalar values, we can also work with matrices or vectors. The elements in the matrix are placed between square brackets and are entered per row. Each row in the matrix is terminated by a ';'. For example, the Matlab expressions

```
a = [0 1 2 3; 4 5 6 7; 8 9 10 11]
b = [1 2 3]
```

correspond with the following matrices:

$$a = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{pmatrix} \tag{1.1}$$

$$b = \begin{pmatrix} 1 & 2 & 3 \end{pmatrix} \tag{1.2}$$

Matrices can be multiplied (*), summed (+), subtracted (-) or divided (/, \\ ). In this last case, the two operators for division respectively correspond to a *left* or *right* multiplication with the inverse of the second matrix. It is important to note that the matrix operations can only be applied to matrices that comply to the corresponding dimension requirements (e.g. *c=b*a* is possible with the matrices above, but *a*b* will return an error.

For matrices of identical dimensions, it is also possible to apply element-wise operations in which scalar operations are applied between the elements that are in the same position in both matrices. These operations are indicated by the operation, preceded by a dot *(.)*, e.g. *b.*a* and *b./a*.

```
d=2*c
d ./ c
```

If no explicit name is given for the result of an expression, Matlab automatically stores the result of the latest calculation in a default variable called *ans* (a short form for "answer").
Another useful operator is the colon *(:)*. This symbol can be used to produce a matrix, containing all integer numbers from *first* to *last* (including *first* and *last* if they are integer values)

```
first=6;
last=16;
first:last
```

Or more general to produce a matrix containing all values from *first* to *last*, with an increment *step* between successive values:

```
step=3;
first:step:last
```

Variables (matrices) can be saved to disk using the command *save*:

```
save [filename [var1 [var2 [...]]]]
```

The variables stored in *filename* can again be loaded with the command *load*:

```
load [filename [var1 [var2 [...]]]]
```

Example:

```
brol=ans
save tempfile brol;
clear;
whos
load tempfile
whos
```

Creating functions

Up to this point, we entered all commands and calculations directly into Matlab's command window. For practical use this approach however has a number of important drawbacks:

- If we apply complex or long calculations, a large number of intermediate variables will be created in the memory space, making it difficult to oversee what results are finally relevant.
- If we want to perform the same calculations for a new set of input values, the **complete** sequence of operations has to be typed again.
- It becomes very difficult to have an overview of long and complex calculations (no modularity of the source code)
- If a single error is made, sometimes the complete sequence of operations has to be repeated (no editing)

To counteract these problems, Matlab allows the creation of your own functions: On Matlab's menu bar, select *File* → *New* → *M-file*. This opens an editor that can be used to create a new function. Each function starts with the following header syntax:

```
output = functionname(input1,input2,....)
```

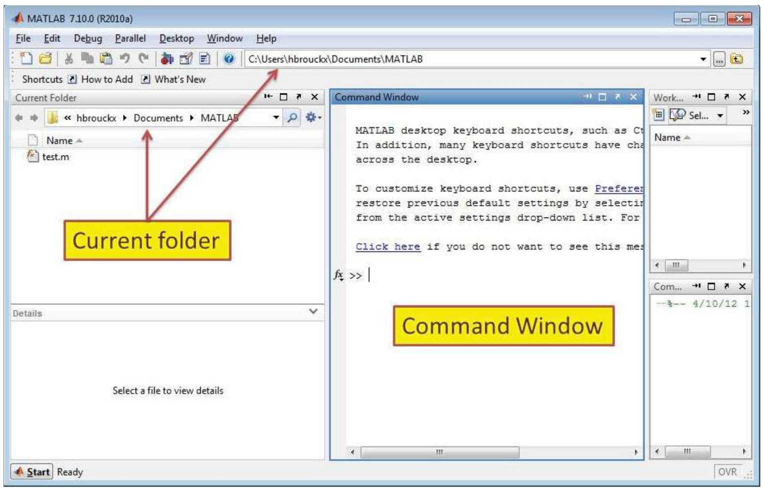
Followed by the sequence of commands that have to be applied to the input parameters. To reduce the amount of output to the command window, these commands are frequently ended by the silencing character (;), and comment lines are added using the comment sign %. The following is an example of a function with comments:

```
function y=supersom(start,step,stop)
% calculate the sum of all integers between start and stop,
% with increment step between the integer values

% row vector containing the integer values
x1=[start:step:stop];

% row vector with the same dimensions as x1, containing only ones
x2=ones(1,length(x1));

% multiply x2 and the transpose vector of x1
y=x2*x1';
```


MATLAB 7.10.0 (R2010a)
File Edit Debug Parallel Desktop Window Help
C:\Users\hbrouckx\Documents\MATLAB
Shortcuts How to Add What's New
Current Folder
« hbrouckx ▸ Documents ▸ MATLAB
Name
test.m
Current folder
Details
Select a file to view details
Command Window
Work...
Sel...
Name
Com...
Command Window
Start Ready
OVR

```
MATLAB desktop keyboard shortcuts, such as Ct
In addition, many keyboard shortcuts have cha
across the desktop.

To customize keyboard shortcuts, use Preferen
restore previous default settings by selectin
from the active settings drop-down list. For

Click here if you do not want to see this mes

>> |
```

```
--%-- 4/10/12 1
```

Figure 1.1: *Matlab's main window*

Note: here we used a new unitary operation, indicated by an ' sign. This operator performs a transposition and complex conjugate on the matrix it is applied to (in this case x1).

The created function can be saved to a file by selecting *File* → *Save As...* in the Matlab editor's menu bar. The name of the file should be the same as the name of the function (if this is not the case, Matlab will give the function the name of the file), followed by the extension .m.(here: "supersom.m")

To run a saved function, the directory where the corresponding file is located should also be known by Matlab. This can be done by setting this directory as the "Current Folder" (see figure 1.1) or by adding it to the Matlab path (a list of active directories that contain functions) with the command:

```
path(path, 'c:\path to supersom');
```

Once the function is saved to a file, and its location is known by Matlab, the function can be run by typing the following command in the command window:

```
testje=supersom(1,2,7)
```

or shorter (using the implicit return variable "ans"):

```
supersom (1,2,7)
```

Some additional comments:

- The scope of variables that are declared/used inside a function is limited to that function. Once the function terminates, these variables are no longer available (e.g. use *whos* to check if the variables $x1$ and $x2$ are still visible)
- Because a description of the function was added in the comments of the file, Matlab can now help users that want to use our *supersom* function (check this with the command *help supersom*). More generally, all comment lines that directly follow the function header, up to the first line that doesn't start with the comment sign %, are used to generate the help description of the function.
- If the function name and filename don't correspond, Matlab will only use the filename to identify the function (using the function name will not work).
- A saved function can always be modified (to correct errors or improve functionality) by opening it in the Matlab editor (*File* → *Open m-file...*)
- Within a Matlab function, it is possible to call other matlab functions (or even itself, i.e. recursion)
- Input and/or output variables are not always required in a function. Input and output of data can alternatively be performed by direct interaction with the user (through the command window, or even through graphical user interfaces). Illustrations of this possibility will be given in the practical exercises in section 1.2.

Some frequently used control flow statements are:

```
if expression
    statements
end
```

```
if expression
    statements
else
    statements
end
```

```
for variable = expression (e.g. i = 1:n)
    statements
end
```

```
while expression
    statements
end
```

In these control flow statements, frequently conditional expressions are used that are based on the comparison operators == (true if the compared variables have the same value), ∼= (true if the compared variables are different),& (logical AND),| (logical OR), and the logical NOT operator ∼. The order in which complicated expressions are calculated, can be influenced by using brackets, e.g. *x1 == 3 ∗ (x2 − 1) + 2*.

Some frequently used interactive input/output statements are:

```
variable with user's choice = input('prompt for the user')
```

```
disp('message string to display')
```

1.1.3 Signals in Matlab

An important consideration that has to be made is that signals can only be represented in Matlab as a set of numerical values of finite length (i.e. finite-length sampled signals). As an example, we'll look at a sinusoidal signal, generated using the Matlab *sin*() function, which calculates the sine values corresponding to a matrix of angles, expressed in radians.

Exercise

Use the sin() function to construct a finite-length sampled version $s_s(n)$ of an analogue sine signal $s_a(t)$ and store it in the vector s:

$$s_a(t) = Asin(2\pi f_0 t + \phi) \tag{1.3}$$

with $A = 1, f_0 = 400Hz, \phi = 0$ and sampling frequency $f_s = 3200Hz$. Start the first sample at time $t = 1ms$ and stop at $t = 6ms$.

Plot *s* using the command *stem(s)*. What time-scale does Matlab use for the horizontal axis of this plot?

As can be noticed from the example above, Matlab indexes the position of the numerical values in its vectors and matrices using **integers**, starting with index 1 (contrary to e.g. the C programming language, where the first index of a vector is 0). This means that, if the exact time index of each of the samples is needed, we need to keep track of these indices ourselves (e.g. in a separate matrix, or reconstruced through the known start time and sampling frequency). **Remark:** in most theory books and courses on sampled signals, the first index of a sampled signal $s_s(n)$ is chosen to be 0. This means that in our case, the following correspondence exists between the first sample of the sampled signal, the analog signal and the matlab vector:

$$s_s(0) = s_a(0.001) = s[1] \tag{1.4}$$

Or more general:

$$s_s(n) = s_a(0.001 + \frac{1}{3200} * n) = s[n+1] \tag{1.5}$$

It is thus very important to know the sampling times and/or the sampling frequency of the sampled signal to be able to evaluate the frequency content of the signals. Now calculate the vector $s2$, corresponding to the sampled signal $s_{s2}(n)$ that is obtained by sampling a sine signal with amplitude $A = 1$, frequency $f_0 = 800Hz$, phase $\phi = 0$ at a sampling frequency $f_s = 6400Hz$. The first sample should start at time $t = 0.5ms$ and the last sample stops at $t = 3ms$. Plot the sampled signal in a new figure:

```
figure;
stem(s2); title('s2')
```

What do you see when you compare the plots of $s1$ and $s2$? Calculate and plot the difference between $s(n)$ and $s2(n)$.

Now close the figures, and plot the two signals in two subplots, one located above the other, and use the "analog" (continuous) time as an index for the abscis. Do this by filling two new matries $t1$ an $t2$ with the times corresponding to the samples, and inserting the following code:

```
close all;
subplot(2,1,1), stem(t1,s);
subplot(2,1,2), stem(t2,s2);
```

Now close this window, and plot the following sampled sine signal:

$$x_4(n) = cos(\frac{\pi}{\sqrt{23}}n) \text{ for } 0 \leq \text{n} \leq 50 \tag{1.6}$$

Even though it is based on a periodic cosine, the signal x_4 isn't periodic (no repetition of a set of samples can be observed). Explain why?

1.1.4 Some frequently used standard commands and functions

plot(x,y) plots a ”continuous” graph of the samples in variable y (vertical axis) versus the values in vriable x (horizontal axis). The graph is linearly interpolated between successive points.

While learning Matlab, some very useful commands are *’help’* and *’help topic’* (e.g. *help plot*). In its basic form, *help* returns a list of matlab’s known directories and functions (the Matlab path). When combined with the name of a known dirctory, *’help directory’* returns the content of this directory (e.g. *help datafun*). Finally, *’help functionname’* returns the manual for the requested function (e.g. *help fft*).

In most cases, the help function described above will suffice to find the most frequently used Matlab commands. A more thorough (but significantly slower) search for functions with a specific functionality can also be performed by using the *’lookfor’* command. This command searches the complete documentation of all functions for a specified keyword. As an example, search for a function that computes the group delay of a filter:

```
lookfor delay
```

1.1.5 Using figures and sounds in Matlab

As we’ve seen in the previous text, Matlab can plot on-screen graphs and figures. For practical use; it is however also possible to print these figures to paper (menu bar: File → Print...) or to copy-paste the figure to other Windows programs like MS Word (menu bar: Edit → Copy). To make the graphs more clear, it is frequently useful to format the graphs with specific colors and linestyles, as in the following example:

```
set(gcf, 'color', 'w'); % Background color (white)
% gcf = get current figure
p=plot(x, y, 'k'); % plot y(x) using a black ('k') line
ylabel('y');
xlabel('x')
set(gca, 'xlim', [x1 x2], 'ylim', [y1 y2]); % rescale
% x-axis limits from x1 to x2; y-axis from y1 to y2
set(gca, 'Xcolor', 'k', 'Ycolor', 'k'); % set the color of the X and Y
axis to black
% gca: get current axes
set(p, 'color', 'r'); % changle the color of plot p to red ('r')
```

As an alternative possibility of data presentation, Matlab also allows playing sounds that are represented as a sampled signal in a vector (single channel audio) or matrix (multichannel audio). This is done through the use of the command *soundsc(x,fs)* with fs representing the sampling frequency and x the sampled signal. The amplitude of the signal is automatically adjusted to a suitable range ([-1,1]) prior to D/A conversion. A prerequisite for sound presentation is of course the presence of a suitable sound device, supported by Windows.

1.2 Illustration of theoretical concepts

1.2.1 Sampling theorem

The sampling theorem can be illustrated by sampling (co)sines of different frequencies, and plotting the resulting spectrum. Because, for digital systems, the ratio between the signal's frequency and the sampling frequency is most important.

Two remarks however have to be made:

- Computers are inherently digital systems. This means that all signals that are generated and handled by matlab are digital. Ideally therefore we would need an analog signal generator and an analog-digital convertor (ADC) to obtain the sampled signals. This would require a relatively large setup, which we will avoid by considering sinusoidal signals, for which we know an accurate mathematical description of the analog signal at all times. In this way we can simulate the ADC conversion in Matlab, and generate the sampled signals:
  - analog signal (cosine with frequency f_0):
  $$c_a(t) = cos(2\pi f_0 t) \tag{1.7}$$
  - sampled signal (sampling frequency f_s):
  $$c(n) = cos(2\pi f_0 \frac{n}{f_s}) \tag{1.8}$$
- Also, since we work with a digital system, it is impossible to calculate the complete Fourier transform in the continuous frequency domain ω. The DFT (discrete fourier transform) however does allow the calculation of a "sampled" Fourier transform, corresponding to an

infinite signal that periodically repeats the finite input sequence [1]. The smart choice of our (sinusoidal) input signal however allows us to use this property of the DFT, and make sure that the results of the DFT correspond to the wanted Fourier transform. By choosing the finite sequence to correspond to an integer number of periods of the sinusoidal signal, the time signal obtained by repeating the sequence corresponds perfectly to the original sinusoidal signal, and the analog and discrete Fourier transforms indicate the same frequency domain representation. One very important remark is however that the periodicity of the signal has to be considered in the sampled sequence. As we've seen before, the fact that the signal is sampled can change the periodicity of the signal itself (an extreme example of this is that, if we sample a sinewave signal at a sampling frequency that is an irrational multiple of the sine's frequency, the resulting sequence is non-periodic). We therefore choose the ratio of the frequency of the analog signal and the sampling frequency in such a way that the resulting sampled signal is periodic, by ensuring that this ratio is a rational number:

$$\frac{f_0}{fs} = \frac{k}{nfft} \tag{1.9}$$

We see that if our input sequence has a length of $nfft$ samples (which will be used as input for the DFT), this sequence contains exactly k periods of the signal, since $nfft\,T_s = kT_0$. In this case we have that:

$$c(n) = cos(2\pi\frac{k}{nfft}n) \tag{1.10}$$

In matlab, our strategy can be implemented as follows:

```
function sampling
% sample cosines of different frequencies
% , and plot the corresponding magnitude
% spectra.
%
% use: sampling

nfft=1024;
% close all open graphs.
close
% main loop: generate sampled cosines with normalized frequency
% i/nfft
% (i from 0 to nfft) in steps of 56 (remark: if i ≥ nfft/2
% something strange will happen. Why?).
step=56;
for i=0:step:nfft
    % cconstruct matrix x as a row matrix, containing an
    % integer number of periods from the sampled cosine
    x=cos(2*pi*(i/nfft)*(0:nfft-1));
    plot((-1/2:1/nfft:1/2-1/nfft), abs(fftshift(fft(x,nfft))));
    title(['Amplitude spectrum (f0=', num2str(i/nfft),' & fs=1)'])
    pause(1)
end
```

Now try to modify this program to work with block waves instead of sinusoidal waves by applying the "*sign*" function on the sinusoidal signal. Use smaller steps in frequency (e.g. stepsize 20), and limit the period of the block waves to a range corresponding to frequencies smaller than $fs/2$ (don't include frequency 0). To speed up the function, the pause between plots can be shortened (leave a long pause only for the first plot).

Now run the program with the block waves. In the first plot, it can be clearly seen that contrary to the sinewave, the block wave shows multiple harmonics in its spectrum (in theory, the harmonics of an analog block wave are not bandlimited). Run the program a first time and follow the evolution of the first harmonic (fundamental frequency) and check what happens to it. Repeat running the program, but now follow the evolution of the second, third,... harmonics. What happens to the different harmonics? Which ones exhibit aliasing, and when does this aliasing occur?

1.2.2 Oversampling or upsampling

In this example we'll look at oversampling, a technique which is for example used in Digital-to-Analog conversion systems. Write the following program in Matlab, and run it as *oversamp(i)*, with *i* the desired oversampling factor. The plots show the time and frequency domain representations of a sampled sine signal, and of an i-times oversampled version of this signal. Can you see what happens to the spectrum when a signal is oversampled?

```
function oversamp(o)
% use: oversamp(o)

nsin=16; nfft=1024;
close; % close all open plots.

if o > (nfft/nsin)
    disp(['oversampling factor too high. '...
    'limit to ' num2str(nfft/nsin)])
elseif o < 1
    disp('Error! Choose o > 1.')
else
    disp('frequency axis 0-> 1 (digital => periodic)');
    disp('Triangle shapes are caused by low graphical resolution '...
    'and intepolation by Matlab');

    % construct x as a row matrix containing nfft/nsin periods
    % of the sampled cosine
    x=cos(2*pi*(1/nsin)*(0:nfft-1));

    % draw the sampled cosine...
    subplot(2,2,1);
    plot((0:1:nsin-1), x(1:nsin) ,'.'); title('1 period cosine')

    % and its spectrum (same frequency resolution as we'll
    % later use for the oversampled case
    subplot(2,2,2);
    plot((-1/2:1/(nsin*o):1/2-1/(nsin*o)),abs(fftshift(fft(x,(nsin*o)))));
    title('Amplitude spectrum')

    % construct the oversampled signal; The buffer now contains
    % (nfft/nsin)/o periods.The original cosine had a period nsin.
    xo=zeros(1,nsin*o);
    for i=0:nsin-1
    xo(i*o+1)=x(i+1);
    end

    % plot the oversampled signal...
    subplot(2,2,3), plot(xo,'y.');
    title([num2str(o),'-fold oversampling']);

    % ... and its spectrum
    subplot(2,2,4);
    plot((-1/2:1/(nsin*o):1/2-1/(nsin*o)), abs(fftshift(fft(xo,nsin*o))));
    title('Spectrum after oversampling');
end
```

1.3 Exercises

The goal of these exercises (and the exercise sessions in the following weeks) is to gain a better insight in the theoretical concepts that were introduced during the theoretical part of the Digital Signal Processing course. It is therefore highly encouraged to experiment with the wide range of simulation possibilities, made available by Matlab, to solve and elaborate on the proposed problems.

All solutions should be saved in ".m" files, with sufficient documentation (use %) to explain why you implemented the solution in that way, and what is happening. Also, an answer to the questions should be present in the comments of the function files.

1.3.1 Exercise 1: Frequency Domain Sampling

Design your own program to illustrate the fact that **frequency domain** sampling can lead to **time domain** aliasing. This can for example be done by generating a signal $x(n)$ (e.g. a signal consisting of one triangle) in the time domain, and performing the following operations:

- transform the signal to the frequency domain, using the fft function.
- subsample the frequency domain representation by a factor N (2,3,...)
- Transform the (re)sampled signal to the time domain, using the $ifft$ function

Questions:

- What happens to the time domain signal for different values of N? What happens when the ratio $\frac{Nfft}{N}$ is not an integer number? Why?
- What happens to the timescale of the time domain signals? What about the sampling frequency?
- How can time domain aliasing be avoided? Try to illustrate this.

1.3.2 Exercise 2: Sampling the product of two cosines

Sample the time domain representation of a product of two cosines, one with frequency $500Hz$ and one with frequency $1000Hz$. What is the theoretical minimal sampling frequency, required to unamiguously represent this signal?

What do you notice when you sample the signal at different sampling frequencies, ranging from $750Hz$ to $4000Hz$? Also try to listen to the different sampled signals if possible.

Bibliography

[1] W. Verhelst, Digitale signaalprocessing I, VUB uitgaven, 1994-95.