Universitetet
i Stavanger

Introduction to MATLAB

Notes for the introductory course

Trygve Eftestøl

August 15, 2017

Contents

This note provides an introduction to the use of the technical computational These notes provides an introduction to using the technical computational tool , MATLAB. It focus on typical areas of usage: mathematical computations, visualisation, data analysis and programming. Reading these notes, you will get to know MATLAB's user interface, try out vector amd matrix manipulation, use some of the central computational functions, get impressed by MATLAB's capabilities for and reach a level of expertise where you will be able to make your own small MATLAB programs.

1 Introduction

MATLAB is an interactive matrix based system for numeric computations and visualisation. The MATLAB name is acronym for MATrix LABoratory.

MATLAB provides you with the possibility for computations, visualisation and analysis far exxceeding the scope of these notes. Therefore you should get familiar with the use of the help and documentation system integrated in MATLAB. By doing this you will be able to find the proper functionality when you need it.

2 The MATLAB user interface

The procedure for starting MATLAB varies according to your operating system. For *Windows* and *MAC* users the MATLAB application will usually be available from the menu system, represented as an icon. Start MATLAB by double clicking this icon. *UNIX* and *LINUX* users can start MATLAB double clicking an icon or from a terminal window.

Whichever platform MATLAB is started from, a user interface similar to what is shown in figur 1 is launched. The user interface is configurable. You will probably see a configuration slightly diferent from what you see in the figure. The most important available interface windows are the *command window, workspace, editor* and *help* windows.

2.1 The Command Window

In the command window you can declare variables, perform simple operations and call functions. In figure 1 the command window is shown at the bottom right. Commands are given at the prompt, `>>`. If you want to execute a

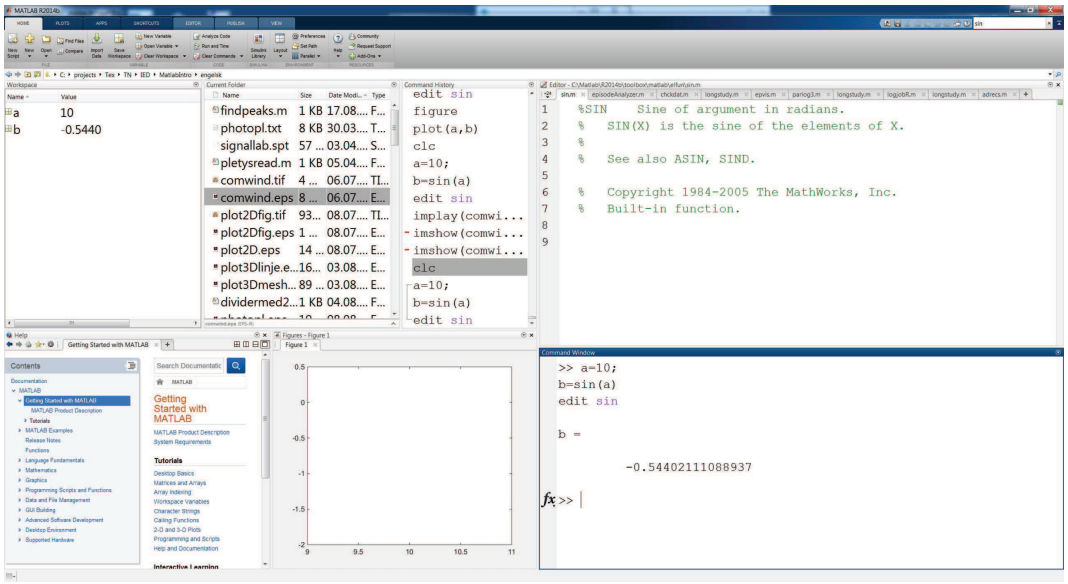

Figure 1: Matlab user interface: Command window (bottom right), Workspace (top left), Editor (top right), Help (bottom left). Also note Figure window (mid bottom) Current folder and Command history (mid top).

command called *function*, this can be done in the command window according to (executes at ENTER):

```
>> function
```

A simple example;

```
>> 10;
```

Notice how the command is ended by a semi colon. If the semi colon is omitted, the result of the operation will be displayed in the command window according to

```
>> 10
```

where MATLAB will respond

```
ans =
    10
```

This applies in general, no matter if one executes simple commands or function calls.

2.2 The workspace

In the workspace window (Workspace) the variables you have created during your MATLAB-session is displayed.vises de variablene du har opprettet i løpet av din pågående MATLAB-sesjon. The variable window is shown in the upper left in figure 1. In the example above, the result from the operation was stored in the variable *ans* which is the standard variable for the most recent result when a variable name is not given explicitly. A variable can be created directly in the command window e.g. when we want to create the variable a and store the value 10 in it. This is done quite simply by giving the command (executes at ENTER)

```
>> a=10;
```

Variables are often used to declare input and output parameters at function call. As an example,

```
>> b=sin(a);
```

means that the command *sin* is called with the variable a as input parameter and the result from the operations executed is sent back through an output parameter and stored in the variable b. At the execution of these commands, the variables a og b will be displayed in the workspace. You can study the content of a variable by double clicking the variable icon in the variable window.

Both the commands and variable display described above is illustrated in figure 1.

2.3 The editor

In the editor (Editor) you can make your own MATLAB-functions by writing your programs in *M-files*. The editor window is shown in the upper right in figure 1. The editor can be used for debugging of the programs you are writing. The two commands described above can be put into an M-file:

```
a=10;
b=sin(a);
```

If the M-file is named *instructions.m*, the two commands can be run by giving the command

```
>> instructions;
```

2.4 Help

The Help window (Help) is placed at the lower left in figure 1 and can be used to search for and display documentation and demonstrations for all the installed MATLAB-products. The help tool can be used in different modi: You can navigate in a *Contents* menu or use the *Search* dialog. In the help window in the figure, content navigation is used to display documentation about how to get started with MATLAB.

In addition to all these tools there is also the *command history* and the *current directory*.

2.5 Exercises

a) Start MATLAB.

b) Study the organisation of the user interface. Use the mouse to move the window to make the configuration of your user interface similar to what is shown in figure 1.

c) Open the editor by giving the command `>> edit`. Place it in the user interface according to the configuration in figure 1 by first clicking the *docking arrow* (curved arrow at the upper right of the window) followed by dragging it into position above the command window.

d) Use the help window in content modus to find documentation for starters (Getting Started). Search for information on *command history* and *Current Directory*.

e) Use the help window in search mode to find information on how the *sine* function is used.

3 Basic functions

Now it is time to get started with the main issue about MATLAB - computations. It will be useful to keep in mind that MATLAB, as the name indicates, is *matrix* based. A matrix is a set of elements organised in a rectangular array. These matrices can represent various types of data:

- Scalars, vectors and matrices which all are diferent types of matrices, but containing one single element, a single row of elements or several rows of elements, respectively.

- Tables which are organised as a list of elements. MATLAB will not distinguish between a matrix and a table, but gives the user the opportunity to handle data as one or the other.

The concept of *elements* might seem a bit vague, but MATLAB makes distinction between elements of different types:

- Numerical numbers which can take part in computations,
- Characters assembled in a vector denoted a *string*,
- Symbols,
- One might even define a matrix as an element.

The first three types of elements can be handled using matrices, while for the last one where the element definition may vary special *cell* structures are applicable. We will return to the various data stuctures of MATLAB.

As the concepts of vectors and matrices might be vague to some, we give some hands-on examples:

- A *sound file* will be a vector. When the sound file is imported into MATLAB, the sound samples will be represented as elements in a vector.
- A *grey tone image* will be a matrix. When the image file is imported into MATLAB, the picture elements (pixels) will be represented as elements in a matrix.

3.1 Creating matrices

Matrices can be assigned in the following ways:

- The elements are explicitly put into a list,
- Generated using built-in functions,
- Created in an M-files called from the command window,
- Read from external data files or applications.

Here are some examples on how we can create matrices directly in the command window.

Assigning scalars:

```
>> a=5
a =
    5
```

Assigning a row vector:

```
>> y=[0 5 10 15 20]
y =
    0    5    10    15    20
```

The elements can be separated by a comma instead of a space. The result will be the same.

A column vector can be assigned correspondingly

```
>> x=[0; 5; 10; 15; 20]
x =
     0
     5
    10
    15
    20
```

A matrix can be assigned as follows

```
>> A=[0 1 2 3 4; 5 6 7 8 9; 10 11 12 13 14; 15 16 17 18 19]
A =
     0  1  2  3  4
     5  6  7  8  9
    10 11 12 13 14
    15 16 17 18 19
```

We can assign the same matrix from an M-file, *Amatrix.m*, where the following instructions are given

```
A=[0 1 2 3 4; 5 6 7 8 9; 10 11 12 13 14; 15 16 17 18 19]
```

A is then assigned by

```
>> Amatrix
A =
     0  1  2  3  4
     5  6  7  8  9
    10 11 12 13 14
    15 16 17 18 19
```

It will be more appropriate and intuitive to write the instructions in the M-file as follows

```
A=[0  1  2  3  4
   5  6  7  8  9
  10 11 12 13 14
  15 16 17 18 19]
```

3.2 Matrix operations

Some of the most fundamental mathematical matrix operators are addition, subtraction, multiplication og transponation. Another useful "house hold"-operator is the colon operator.

+	addition
-	subtraction
*	multiplication
/	division
^	power
'	transponation

It is important to note the difference between *matrix* operations and *element* operations.

Examples of matrix operations can be matrix- vector multiplication well known from linear algebra. If we use the variables from the previous section:

```
>> y+y
ans =
    0   10   20   30   40
```

`x+x` will give an equivalent results, but in column vector format. `x+y` will give an error message as the two vectors have different dimensions.

```
>> b=A*x
b =
     150
     400
     650
     900
```

The other operators will work correspondingly obeying the rules for operations on vectors and matrices from linear algebra. Addition and subtraction works element wise so that the resulting matrix has the same dimension as the operand matrices, and the value in a specific position is the result of the operations performed on the elements in the corresponding positions in the operand matrices. This does not apply to the other operators, men these can me made to work element wise by appending the operator to a period according to

.*	element wise multiplication
./	element wise division
.^	element wise power

Operasjones between a scalar and a vector/matrix will always be element wise.

3.3 Matrix functions

Some basic mathematical matrix functions allow us to sum, diagonalise, find determinants etc.

sum	sums the elements in a vector or along the columns in a matrix
diag	diagonalises a matrix
det	computes the determinant of a matrix

There are also some functions useful for "building" matrices.

eye	generates an identity matrix
ones	generates a matrix of ones
rand	generates a matrix of random numbers

length	computes the length of a vector
size	determines the dimension of a matrix
:	automatic generation

As an example, a $2X2$-matrix can be generated by

```
>> I=eye(2)
I =
     1  0
     0  1
```

The colon operator is very powerful and can be used to increase the efficiency of.

```
>> n=[1:5]
n =
    1   2   3   4   5
```

As we can see, the operator fills in all the integers between 1 and 5. The default increment value is 1, but we can decide its size by giving its size. For example we can set the increment value in the previous example to 2.

```
>> n=[1:2:5]
n =
    1   3   5
```

As an example, some of the vectors and matrices we generated previously can be described more efficiently

```
>> y=[0:5:20]
y =
    0   5   10   15   20
```

3.4 Indexing matrices

By indexing one might work with excerpts of matrices. The colon operator is important in this context. The element in row i, column j in a matrix, A, can be retrieved by giving the command `A(i,j)`. The first element indicates the row while the second element gives the column. As an example

```
>> A(2,3)
ans =
    7
```

retrives the element in row number 2, column number 3 in the matrix A.

We do not need to limit our attention to single elements, but can use the colon operator to indicate which part of the matrix we want to access.

```
>> A(2,1:end)
ans =
    5   6   7   8   9
```

retrives the second row from A. *end* is used to indicate te last element. This command can be given an even more convenient short form

```
>> A(2,:)
ans =
    5   6   7   8   9
```

Finally, it is useful to be able to expand a matrix by connecting it to another matrix.

```
>> [a y]
ans =
    5   0   5   10   15   20
```

3.5 Logical operators

MATLAB has several logical operators which are applied on matrices element wise

<	less than
>	larger than
<=	less than or equal to
>=	less than or equal to
==	equal
~=	not equal

Note that == is used as a logical operator wile = is used for assignment.

In addidion we have the well known logical operators

&	logical AND
|	logical OR
~	logical NOT

We can determine which elements in y are larger than or equal to 10 ved

```
>> y>=10
ans =
    0    0    1    1    1
```

which gives a boolean result where 0 and 1 indicates the outcome of the test being FALSE or TRUE respectively.

Furthermore, we can use the function *find* to identify the elements in the vector fulfilling the test:

```
>> idx=find(y>=10)
idx =
    3    4    5
```

idx indexes the positions of the elements fulfilling the test.

3.6 Exercises

Some of the operations that you will performwill return an error message. Why? a) Generate the variables *a,y,x* and A corresponding to how it was shown earlier in this section. Use the colon operator to generate the vektor x and the matrix A efficiently.[1]

b) Execute the operations `a+a, a+y, a*y, a*A, x*A, x*x, x.*x, A.*A`.

c) Execute the operation `det(A), diag(A), diag(diag(A))`.

d) Establish a 3X3 identity matrix.

e) Generate a vector corresponding to $\mathbf{x} = (1\ 2\ \ldots 100)$.

[1] Hint: Use transponation for efficient generation of x.

f) Fetch every second element from column number 2 in A.

g) Fetch 6 elements from the middle of A.

h) Expand A by appending y as an extra row.

i) Write the elements of y larger than or equal to 15 to the command window.

j) Crudely put, genes are the "cook book" of an organism. A DNA-string contains genes, areas that regulate the expression of genes, and areas that do not have any function that we know of.

DNA is coded with four exchangeable "building blocks", called *bases*, abbreviated to A, T, C eller G, corresponding to their chemical names: Adenine, Tymine, Guanine og Cytosine. The variable *nucseq*[2] represents a DNA-string. Upon analysis of DNA-strings it is convenient to remove the areas without function. To do this we can take advantage of the periodicity of 3 of the areas of interest. These areas can thus be identified using frequency analysis. To do this, the sequence has to be reformated to a analysable sequence. One way to do this is to make a binary sequence for each of the bases. For instance, the binary sequence for 'a' for the DNA string 'atcgacgta' will be '100010001' while the binary sequence for 'c' will be '001001000'. Make a binary sequence for each of the four bases for the DNA string in nucseq.

4 Visualisation

MATLAB has an extensive library of routines for visualisation of vectors and matrices. We will concentrate on a small selection of the most important functions for generating 2D and 3D graphics. In addidion, there are good opportunity for putting text to and printing graphics.

4.1 Simple 2D plotting

In this context we want to plot function values to their corresponding argument values. Consider the function $y = f(x)$. We would like to plot values of y against corresponding values of x.

To do this one might generate a vector, $\mathbf{x}$ with argument values, or *points of computation*, for the function. The next step is to perform the computation corresponding to $y = f(x)$ for each single point of computation in $\mathbf{x}$.

[2] *nucseq.mat* which can be downloaded from www.ux.uis.no/~trygve-e

As a simple introductory example of plotting we consider the function $y = f(x)$ der $f(x) = x^2$. We want to plot the function in the interval $x \in [-2, 2]$.

The first thing we need to do is to generate points of computation in the interval $x \in [-2, 2]$.

```
>> x=-2:0.01:2;
```

Notice how we employ an increment step of 0.01 so that we get 401 points of computation in total. Furthermore, we compute the functional values for each point of computation:

```
>> y=x.^2;
```

Here we make use of the power operator in an element wise manner so that the operation is executed on each single element in **x**.

Finally we generate the plot:

```
>> plot(x,y);
```

MATLAB responds by generating a figure window as shown in figure 2.

It is also possible to plot several functions simultaneously, define line types and colour, put text on the axes, make titles and print the graphics, just to mention some of the most useful functions.

```
>> y2=2*abs(sin(x));
>> plot(x,y,'b',x,y2,'r:');
>> xlabel('x');
>> ylabel('y');
>> title('Demonstration of simple 2D plotting');
>> print -depsc plot2D
```

Note that ' can not be copied to the command window. The results can be seen in figure 3. You should be able to find out the effect of the various commands by studying figure 3 which actually is an en EPS-fil, *plot2D.eps*, generated by the last command.

4.2 3D plotting

In the previous section we demonstrated how to plot functional values of y against values of x. For 3D plots we want to plot values of z as a function of

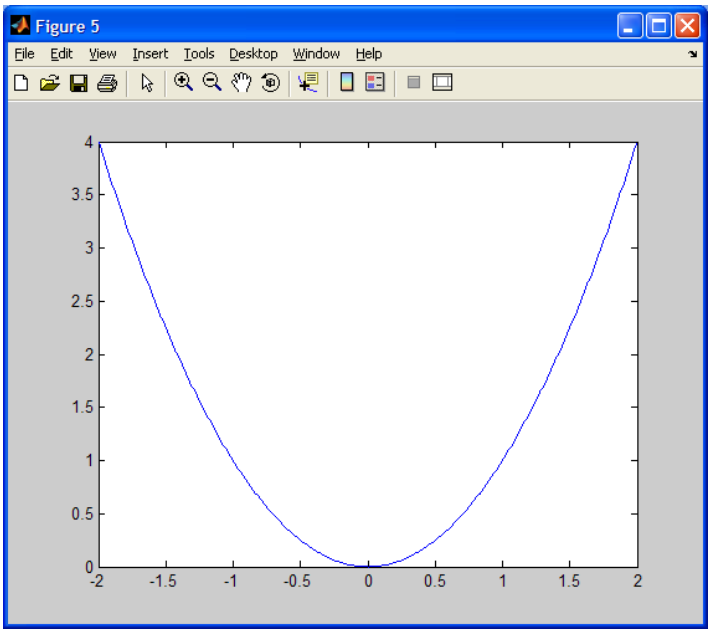

Figure 2: Figure window with plot of $y = x^2$

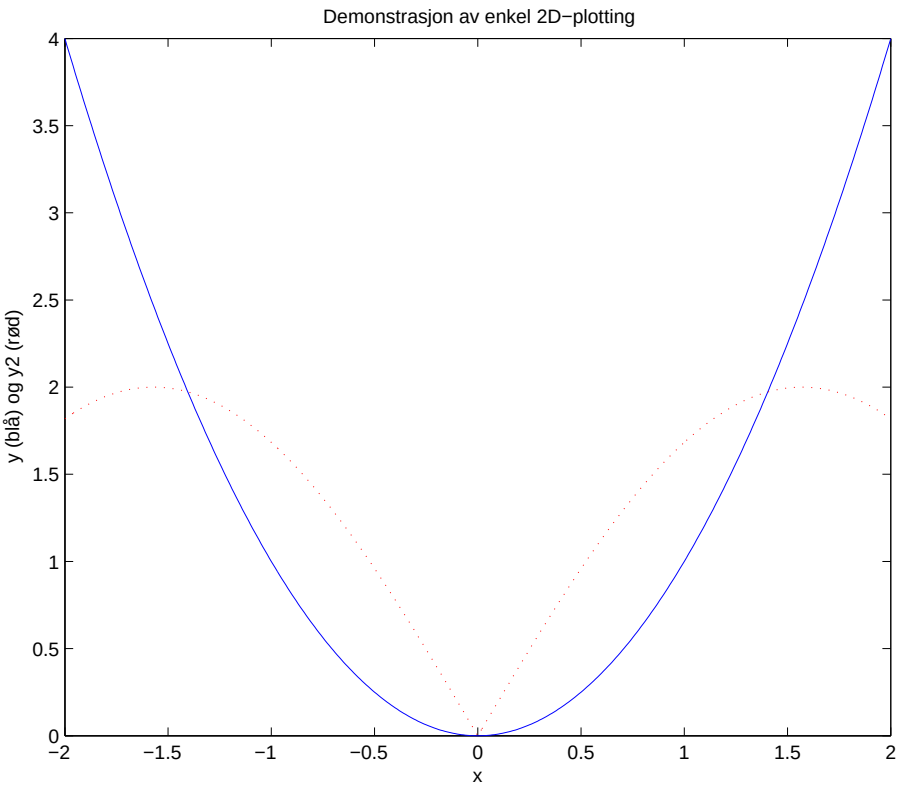

Figure 3: Figure window with plot of $y = x^2$

x og y, $z = f(x, y)$.

3D line plot We can plot a line through (x, y, z) points defined parametrically using the following instructions:

```
>> t=0.01:.01:20*pi;
>> x=cos(t);
>> y=sin(t);
>> z=t.^3;
>> plot3(x,y,z);
>> xlabel('x');
>> ylabel('y');
>> zlabel('z');
>> title('Example of a 3D line plot');
```

The result can be seen in figure 4.

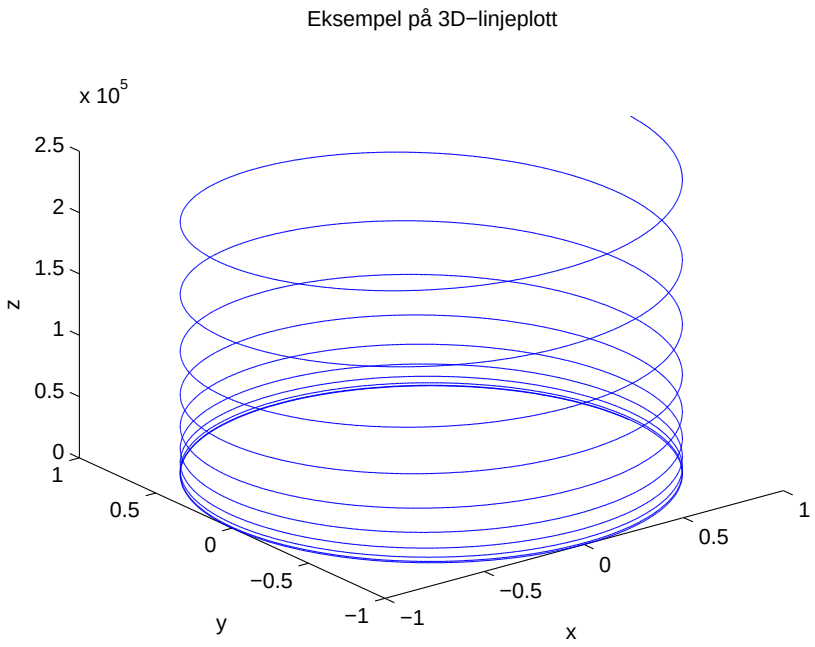

Figure 4: Figure window with 3D line plot

3D surface plot We can plot surface functions as for example the surface given as $z = e^{-x^2-y^2}$. We want to plot this function over the square $[-2, 2] \times [-2, 2]$. To do this we have to generate a grid of point of computations (computational grid) (x, y) within this square. This is done by first

defining the computational grid along each axis and then to span the grid by using the function *meshgrid*.

```
>> x=-2:.2:2;
>> y=-2:.2:2;
>> [X,Y]=meshgrid(x,y);
```

The computation of the function values and plotting of the surface using the function *mesh* renders the plot shown in figure 5.

```
>> z=exp(-X.^2-Y.^2);
>> mesh(z)
>> xlabel('x');
>> ylabel('y');
>> zlabel('z');
>> title('Example of a 3D mesh plot');
```

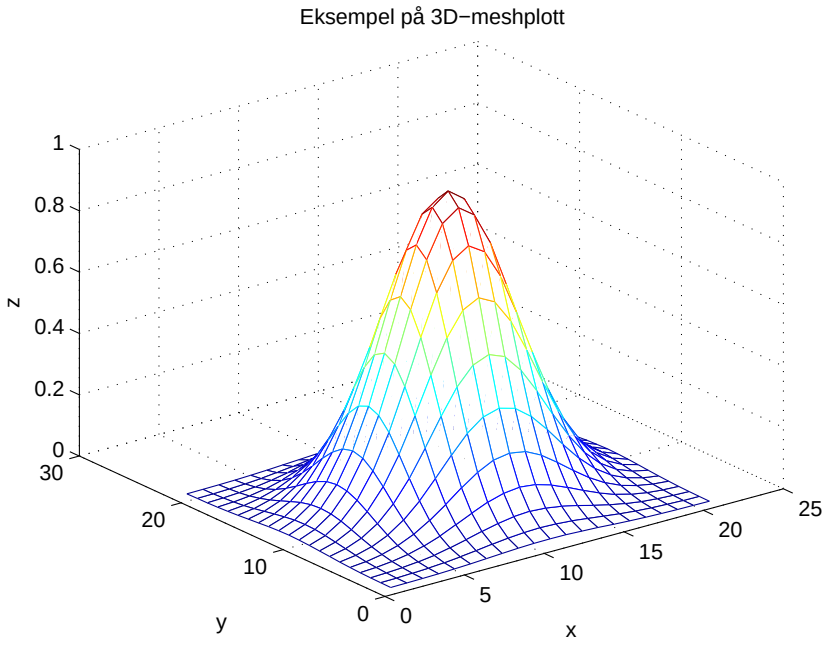

Figure 5: Figure window with 3D mesh plot

4.3 Exercises

a) Plot the signal $x = Acos(\omega_0 t + \theta_0)$, in the interval $t \in [0, 3\pi]$ for $\omega_0 = 2/3$, $A = 2$ and $\theta_0 = 0$. [3]

b) Generate the 3D mesh plot shown in figure 5. Closer inspection of the figur vil show that the x og y axes does not display correct values. Correct this so that the displayed values correspond to the square $[-2, 2] \times [-2, 2]$.

c) Display the function as a surface instead of mesh by using the function *surf*.

d) Generate a 3D mesh plot of the two dimensional *sinc* function $z = sin(r)/r$, where $r = x^2 + y^2$ over the square $[-8, 8] \times [-8, 8]$. [4]

5 Programming

As mentioned previously, you can make your own MATLAB functions by writing programs in so called *M files*. The example in section 2.3 demonstrates how a set of indtructions can be run by calling an M file containing these instructions. This type of M files only containing a set of instructions are called *script* files.

In this section on programming, we shall focus on another type of M files that carries the term *function* files. This enables you to create your own MATLAB functions with input and output parameters such as the sine function used as example of one of MATLAB's built in functions in section 2.2.

5.1 Function files

As a simple example we will make a function for computing the normal probability density function,

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} \cdot e^{\frac{-(x-\mu)^2}{2\sigma^2}}. \tag{1}$$

Here we will use the variable x and the function parameters μ and σ as input parameters. p will be the output parameter. We will call the function *pdens.m*. The contents of the file might look like følger:

[3] Remember that $w_0 = 2\pi f_0$ and that the period is $T = 1/f_0$, so this corresponds to plotting one period of x.

[4] Remember that there will be a problem if you try to compute the function for $x = y = 0$. This can be handled by adding a value *eps* close to 0 to all points of computation.

```
function p=pdens(x,m,s)

% PDENS Computes the probaaility density values
%   P=PDENS(X,M,S) computes the density value P for
%   X for a gaussian density function with
%   mean value M og standard deviation S.

p=1/(sqrt(2*pi)*s)*exp(-1/2*(x-m)^2/(2*s^2));
```

Notice the help text where each comment line starts with a `%` so that MATLAB won't interpret the line as an instruction to execute.

5.2 Control structures

MATLAB employs many of the control structures well known from other programming languages. We will look at the most important ones.

5.2.1 Conditional testing with *if*

One of the two important control strucures is *if*. In describing an algorithm, one often needs to choose between two different activities depending upon some test condition being true or false. In MATLAB this can be expressed as

```
>> if condition; activity1; else activity2; end;
```

For example if one needs to make a procedure, *absolute*, that computes the absolute value $y = |x|$ of a number, x, so that $y = x$ when x is positive and $y = -x$ otherwise.

```
function y=absolute(x)

% ABSOLUTE Computes the absolute value
%   Y=ABSOLUTE(X) computes the absolute value, Y, of X

if x < 0
  y=-x;
else
```

```
  y=x;
end
```

Note the *key words if*, *else* and *end*. These key words are reserved for this purpose and may not be used for other purposes.

5.2.2 Iterative structures using *while* or *for*

while The second of the two important control structures is *while*. This makes it possible to repeat the execution of an activity as long as som condition is fullfilled. Combined with conditional testing this gives the opportunity to describe algorithms in a powerful way. In MATLAB this may be expressed as

```
>> while condition; activity; end
```

As long as the condition is true, execute the activity. Return and execute the activity again. When the condition is false, the iterations terminate and the execution continues at the first instructions after the while structure.

As an example one might want to make a function, *divideby2*, which divides an integer n with 2 as many times as possible. The commands `fix` and `rem` are used to compute the integer quotient and the remainder respectively.

```
function q=divideby2(n)

% DIVIDEBY2 Divide by 2 as long as the remainder is zero
%   Q=DIVIDEBY2(N) computes the quotient, Q,
%   after the maximum number of divisions by 2

q=n;
while rem(q,2) == 0
  q=fix(q/2);
end
```

As we can see `q` is a local variable that changes and is a part of the test condidion. This is an essential part of the control part of the while structure which consists of the components

- Initialisation: q is set equal to the integer to be divided by 2.
- Test : it is tested if q can be divided by 2.

- Modification: q is changed to the next quotient which will be attempted to be divided by 2. This modification ensures that the terminal condidion will be reached.

for It is worth noting that the *while* loop repeats an undetermined number of times. If you want to repeat a number of instructions a fixed, predetermined number of times, it is convenient to use *for* loops.

As an example we want to modify the function *pdens* so that you can compute the density values for more than one point of computation at a time. A possible way to do this:

```
function p=pdens(x,m,s)

% PDENS Computes the probaaility density values
%   P=PDENS(X,M,S) computes the density value P for
%   the values in a vector X for a gaussian density function
%   with mean value M og standard deviation S.

for n=1:length(x)
  p(n)=1/(sqrt(2*pi)*s)*exp(-1/2*(x(n)-m)^2/(2*s^2));
end
```

As you can see very few modifications are needed to make the function handle vectors. The number of iterations are determined usint the *length* function, and the loop counter `n` is used for vector indexing of both `x` og `p`.

5.3 Exercises

a) Make a function to determine whether a number is odd or even. Return 1 if the number is odd, otherwise return 0.

b) Make a function that generates a vector of random numbers (integers in the range $[0, 9]$ which can be generated by use of *randint*). Further to this, the function shall distribute the numbers to two vectors, one with the odd numbers and the other with the even mumbers.

6 Analysis

To finish this MATLAB introduction, we present an example where MATLAB is used for analysis of realistic data. This example will illustrate *reading* data from a file, *analysis* of the signals decoded from the data. This analysis will involve *spectral analysis*, a central concept in signal analysis, and furthermore *filtering* of the signal. This filtering will make it possible for you to solve the *problem*. In the example the step by step procedure for reaching the will be illustrated. In the final part of the exercise you will have to do the calculations leading to the solution of the problem.

6.1 The problem - computing the heart rate

In this exercise you will be working with a photoplethysmographic dataset measured on one of the empoyees at the institute's laboratory of *medical engineering*. The data file is named *photopl.txt*. You will also need the function *findpeaks.m*[5]. This signal carries information on the pulse beats measured from the employee.

Using the signal, you shall compute the number of beats per minute (the heart rate) and how much the time between the individual beats vary (the heart rate variability). In brief, the method for doing this will be based on finding the time instants for each individual beat.

6.2 Solving the problem

When working with this kind of problem, it will obviously be useful to study the measured signal to consider a possible method for solving the given problem. The first step will be to read the signal from the data file.

Reading the input If the data file is opened in an editor e.g. *Notepad* you will see that the data are ASCII-coded and structured into one channel. The four first lines in the file give information on what kind of measurements the file contains in addition to date of recording, sample frequency and the number of recorded samples. To be able to analyse and perform arithmetic operations on the samples, the data has to be read into MATLAB where the meaured sampled are organised in a vector. This can be done by giving the following commands:

[5]The files *photopl.txt* and *findpeaks.m* can be downloaded from www.ux.uis.no/~trygve-e

```
>> x=csvread('photopl.txt',4,0);
>> x=x/std(x);
```

The function *csvread* is used to decode the ASCII coded data from line 5 and further on, while dividing by the standard deviation is done to scale the signal to unity variance which will be convenient in the further analysis.

Plotting the signal When we want to plot the signal, it will also be convenient to generate a time vector for the signal. Considering the four first lines in the file, we know that the sampling frequency is 40 Hz. The commands for plotting the necessary information can be like

```
>> t=((0:length(x)-1)/40)';
>> plot(t,x);
>> xlabel('t [sek]');
>> ylabel('x(t)');
```

The blue curve shown in 6 show the recorded signal, and the individual pulse beats are clearly displayed as a periodic signal with approximately one beat per minute. One immediate idea is that the pulse rate can be determined by

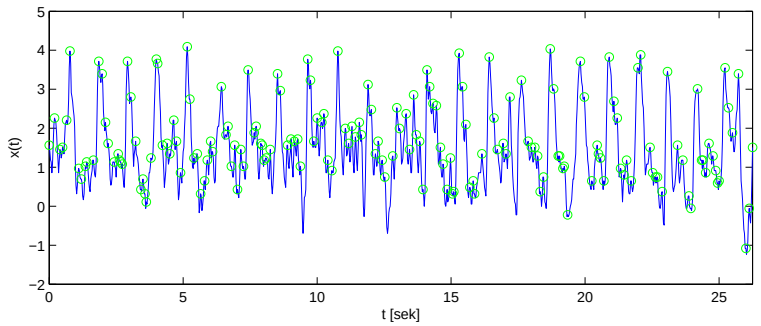

Figure 6: Pulse signal (blue curve) and detected peaks (green circles)

first detecting the beats representing the individual pulse beats.

The detections can be done by using the function *findpeaks*. The folowing commans illustrate the usage of this function and the visualisation of the detected peaks.

```
>> n=findpeaks(x);
>> hold on
>> plot(t(n),x(n),'go')
>> axis([t(1) t(end) -2 5])
```

The variable `n` contains the sample number of the local maxima detected in `x` using the *findpeaks* function. The command *hold on* makes it possible to plot the detected tops superimposed on the pulse curve plotted previously. The plotting of the peaks is doe by indexing the `t` and `x` and specifying their representation as green circles ('go'). The command *axis* is used to delimit the plot along the axes.

The detected tops are thus shown as green circles superimposed on the pulse curve as shown in figure 6. We can see that the pulse beats are detected according to our specification, but that the result includes a large number of false detections as well.

Spectral analysis of the signal When we study the signal, we can see that the pulse beats have a periodic nature. Using *filtering* techinques will possibly enable us to emphasise the periodic component corresponding to the pulse beats we want to detect. At the same time the filter should suppress the components corresponding to the false detections.

To design such a filter we need to know the exact frequency the pulse beats correspond to.

The spectral analysis is done by estimating the spectrum of the signal and then visualising it. This can be done by the following commands:

```
>> [Pxx,f]=pwelch(x,hanning(256),0,1024,40);
>> subplot(2,1,1)
>> plot(f,10*log10(Pxx))
>> xlabel('f [Hz]');
>> ylabel('Magnitude [dB]');
>> xlabel('t [sek]');
>> grid
>> axis([0 20 -40 20])
```

The spectral estimate is computed by the function *pwelch* which is one of a number of possible methods. The spectrum is shown in the upper part of figure 7. The command *subplot* is used to make a split figure window.

As can be seen from studying the spectrum, the signal has a dominant frequency component at 1 Hz. This corresponds to the periodic component of the pulse beats which we previously roughly estimated to be around 1 beat per second.

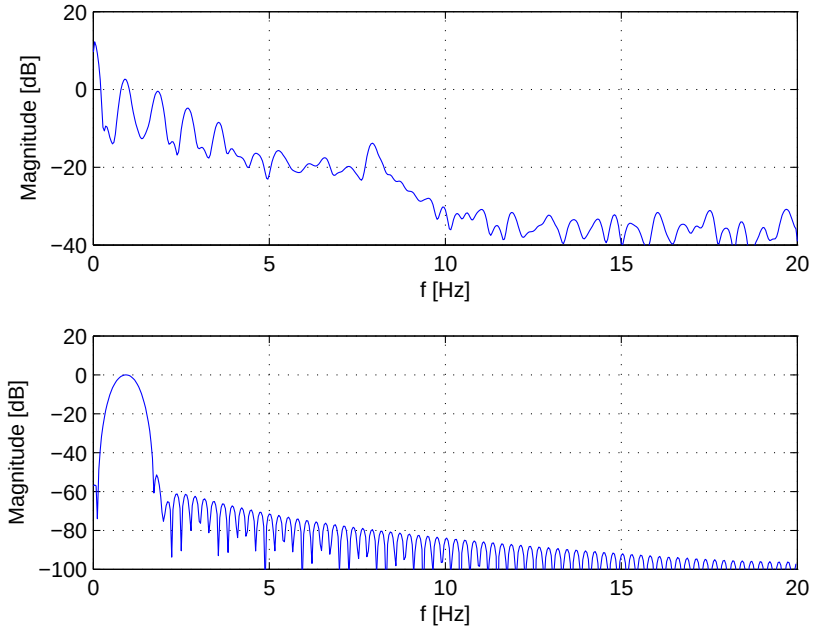

Figure 7: Spectrum (top) and designed band pass filter (bottom)

Filter design We want to design a band pass filter emphasising the frequency comonent at 1 Hz while suppressing other components.

```
>> b=fir1(150,[0.58,1.27]/20);
>> [h,w]=freqz(b);
>> subplot(2,1,2)
>> plot(w/(2*pi)*40,20*log10(abs(hb)))
>> xlabel('f [Hz]');
>> ylabel('Magnitude [dB]');
>> grid
>> axis([0 20 -100 20])
```

The command *fir1* designs the filter where the first parameter states the order of the filter (the higher order, the steeper transition area). The other parameter indicates the lower and upper cutoff frequency for the pass band 0.58 og 1.27 Hz respectively (division by 20 corresponds to normalising to half the sampling frequency). In the frequency domain, the result of filtering will correspond to multiplication of the spectrum at the top with th efrequency response of the filter at the bottom of the figure.

Now we have come to the part where you perform the last steps to reach the solution of the problem. You will have to filter the signal and make a new detection attempt.a er vi kommet dithen at signalet kan filtreres og ny

deteksjon kan utføres. But you will be on your own now (See the exercises given below).

6.3 Exercises

a) Use the function *filtfilt* to filter the pulse signal x.[6]

b) Plot the spectrum of the filtered signal (red curve) in the same plot as the original signal (blue curve). What has been the effect of the filter?

c) Do the peak detection once more, but this time on the filtered signal.

d) Visualise the filtered signal and the recently detected peaks together in the same plot.

e) Determine the heart rate[7].

f) Determine the heart rate variability[8].

[6] In the function documentation the syntax is given as y=filtfilt(b,a,x). Set a=1;

[7] It will be useful to apply the function *mean* to compute the average value.

[8] Express the variability using the standard deviation which can be computed by using the function *std*.