An introduction to MATrix LABoratory (MATLAB)

Lecture notes
MATLAB course at HiT 2006.

© Associate Professor, Dr. ing. (PhD)
David Di Ruscio

Telemark University College
Email: david.di.ruscio@hit.no
Porsgrunn, Norway

September 2000, extended and translated to English august 2006.

Revised August 18, 2009

Porsgrunn **August 18, 2009**

Telemark University College
Kjølnes Ring 56
N-3914 Porsgrunn, Norway

Preface

MATLAB is the acronym for MATrix LABoratory. It is an interactive program thats acts as a "Laboratory" for numerical computations that involves matrices. The first version of MATLAB was originally constructed to give easy access to the FORTRAN subroutines in the LINPACK and EISPACK linear algebra packages which was developed in the period 1975-1979. LINPACK and EISPACK was written in FORTRAN and translated from some older ALGOL subroutines developed in the 60'ies.

The subroutines in LINPACK and EISPECK contains several thousand subroutines for all tasks in linear algebra and are also today the state of the art in linear algebra. MATLAB was available for free in university community in the period $1980 - 1987$ and it was written in FORTRAN, both the MATLAB code as well as the executables was available in public domain. The old code are still available.

In the end of the 80'ies MATLAB was commercialized and are developed and sold by the Math Works Inc. MATLAB is today written in the C and C++ language and is much extended from the first version. MATLAB contains today many toolboxes suitable for most topics. The basics commands are still the same as in the original version.

MATLAB is today an important tool for numerical computations in the University and reasearch community all over the world. MATLAB is also used in many industrial applications and concerns.

Contents

List of Figures

Chapter 1

Introduction

These lecture notes are meant to give an introduction to MATLAB as well as giving some tips in using MATLAB. The lecture notes are suitable for the non-experienced as well as the moore experienced user of MATLAB. These lecture notes was first used in a NIF (Norske Sivilingenirers Forening) course at HIT in the spring of 2000. It is also assumed that the lecture notes is suitable to give sufficient basic theory so that MATLAB can be effectively used in many courses at TF. In particular courses that contains simulations, control theory, data analysis and system identification, optimization, linear systems, linear algebra etc. In addition to being lecture notes in basic MATLAB theory, the notes are extended with the theory for many more advanced tasks linear algebra. En example is the numerical linearization of an non linear dynamic state space model.

In connection to the lecture notes there are an catalog/map, **matlab_files**, with MATLAB m-files so that all examples, as well as all figures can be run and reconstructed by the reader. The m-files as well as the latest version of these lecture notes can be obtained from the zip-file, **matlab_files.zip** and the post-scrip file **main_mkurs.ps** from internet. The address is

`http://www-pors.hit.no/tf/fag/a3894/index.htm`

One of the good things with MATLAB is the m-file system. An m-file contains a sequence of MATLAB commands and can result in an efficient programming language, very similarly to the c, pascal and fortran languages. The m-file system will be presented in moore details lather. The moore experienced user of MATLAB will very often learn moore by reading other m-files. In addition to al the m-files which are available from the Math Works Inc. there are existing a lot of free m-files available from the internet page

`http://www.mathworks.com/support/ftp/`
`http://www.mathworks.com/support/ftp/ftpindexv4.shtml`

In these net pages there are also a lot of interesting m-files for the moore experienced user and researcher.

Chapter 2

Matrix Computations and Simulation

2.1 State Space Model

Linear model for a damped spring system.
A model for a damped spring system is as follows:

$$\dot{x} = v, \tag{2.1}$$

$$\dot{v} = -\frac{k}{m}(x - x_0) - \frac{\mu}{m}v + \frac{F}{m} - g. \tag{2.2}$$

We find the equilibrium point for the system by putting $\dot{x} = 0$ and $\dot{v} = 0$. This gives $x = x_0 + \frac{1}{k}(F - mg)$ and $v = 0$. The above state space model is deduced from a force balance on the system, i.e. $m\ddot{x} = \sum F_i$.

Consider now that we are introducing a new variable, $z = x - x_0$, for the position. The state space model can then be written as

$$\dot{z} = v \tag{2.3}$$

$$\dot{v} = -\frac{k}{m}z - \frac{\mu}{m}v + \frac{1}{m}(F - mg). \tag{2.4}$$

This can be written more compact as a matrix/vector state space model as follows

$$\begin{bmatrix} \dot{z} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\frac{k}{m} & -\frac{\mu}{m} \end{bmatrix} \begin{bmatrix} z \\ v \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{m} \end{bmatrix} u, \tag{2.5}$$

where

$$u = F - mg. \tag{2.6}$$

Numerical values for tha variables can be as follows:

$k = 2$, $mu = 0.5$, $m = 5$, $g = 9.81$ og $x0 = 1$. We also use the initial values $x_0 = x(t = 0) = 1.5$ and $v_0 = v(t = 0) = 0$ as initial values for the states.

Putting the numerical values into the equations we are obtaining the following linear state space model

$$\dot{x} = Ax + Bu, \tag{2.7}$$

where the state vector is, $x = \begin{bmatrix} z \\ v \end{bmatrix}$ and the model matrices are

$$A = \begin{bmatrix} 0 & 1 \\ -0.4 & -0.1 \end{bmatrix}, \quad B = \begin{bmatrix} 0 \\ 0.2 \end{bmatrix}. \tag{2.8}$$

The equilibrium position for the system can be computed in MATLAB as follows:

$\gg m = 5; k = 2; g = 9.81; F = 0,\ x0 = 1$
$\gg z = (F - m * g)/k$
$\gg x = z + x_0$

Show that the values for the equilibrium states will be $z = -24.5250$ and $x = -23.525$. This is the position for the mass when time reach infinity. This is also called a stationary state (steady state).

The steady state can also be computed in MATLAB as follows

```
>> A=[0 1;-0.4 -0.1]

A =

         0    1.0000
   -0.4000   -0.1000

>> B=[0;0.2]

B =

         0
    0.2000

>> u=0-5*9.81

u =

  -49.0500

>> x=-inv(A)*B*u

x =

  -24.5250
```

```
         0
>> x=1+x(1)

x =

  -23.5250
```

Non-linear model for a pendulum
A model for a pendulum can be written as

$$\dot{x_1} = x_2, \tag{2.9}$$

$$\dot{x_2} = -\frac{g}{r}\sin(x_1) - \frac{b}{mr^2}x_2, \tag{2.10}$$

where $g = 9.81$, $m = 8$ is the mass, $r = 5$ is the length of the arm of the pendulum, $b = 10$ is the friction coefficient in origo. The states are the angular position $x_1 = \theta$ and the angular velocity $x_2 = \dot{\theta}$. Note that this model is nonlinear due to the term $\sin(x_1)$ leddet. A simpler linearized model can be deduced by noting that for small angles we have that $\sin(x_1) \approx x_1$.

$$\overbrace{\begin{bmatrix} \dot{x_1} \\ \dot{x_2} \end{bmatrix}}^{\dot{x}} = \overbrace{\begin{bmatrix} 0 & 1 \\ -\frac{g}{r} & -\frac{b}{mr^2} \end{bmatrix}}^{A} \overbrace{\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}}^{x} \tag{2.11}$$

Putting the numerical values into the model we get

$$A = \begin{bmatrix} 0 & 1 \\ -1.9620 & -0.0500 \end{bmatrix} \tag{2.12}$$

Note that the above pendulum model is a so called autonomous mode, i.e., a model which only is driven by initial states. The right hand side of this model is impemented in the MATLAB functon **fx_pendel.m**.

2.2 M-file programming

Files that contains MATLAB code are called m-files. Such files have names as, **file_name.m**, i.e. filenames with extension **.m**. m-files can be functions with inn and out arguments, or they can be script files that executes a series of MATLAB commands. You can write m-files in an ASCII text editor. We recommend that you are using the built in m-file editor within MATLAB. After you have stored the m-file it can be executed from the MATLAB workspace or called from other m-files.

We will below illustrate how the numerical values from the daped spring system can be implemented in a m-file. See the m-file **data.m**, which is listed below

```
% data.m
```

```
% m-file for defining the parameters in the damped spring system.
g=9.81;            % acceleration of gravity.
k=2;               % spring constant.
mu=0.5;            % damping coefficient.
x0=1;              % spring position at steady state.
m=5;               % mass.
x_0=1.5;           % initial value for the position, x.
v_0=0;             % initial value for the velocity, v.
F=0;               % force.
```

Write your own **data.m** m-file and execute the script.

You can lock at the content in the m-file by using the MATLAB command **type**. Execute the following in the MATLAB workspace

$\gg$ **type data**

You can now obtain information of which variables who are defined in the workspace by executing the commands

$\gg$ **who**
$\gg$ **whos**

The built in m-file editor can be started from the **File** menu and chosing **new** followed by M-file. Alternatively you can chose **Open** from the **File** menu if the m-file exists. A third alternative is to open an existing m-file from the workspace as

$\gg$ **edit data**

Note that you can use comments within an m-file by starting a line with the % sign. It is a good custom to use comments when you are programming

If you are executing the following command

$\gg$ **help data**

in the workspace, then only the first lines starting with % in the m-file will be listed in the workspace. In this way you can obtain information and help for all built in and existing m-files within MATLAB. You should learn how a MATLAB function is written and documented by locking at other m-files.

Very often we have a lot of m-files within a map. This is called a Toolbox for MATLAB. If you are making a m-file entitled **contents.m** in this map, then the contents will be listed in the workspace when you are executing the MATLAB command

$\gg$ **help** *map*

If the map does not contain a file named **contents.m** then the comments in all the m-files in the map will be listed in the workspace.

It is also possibile to making so called precomiled code from an m-file by the MATLAB command

≫ **pcode data**

The m-file will here be translated to an unreadable file **data.p**. This file can also be executed from the workspace in the same way as m-files. It is one mayor difference. The comments in the m-file is omitted in the p-code file. If you still want help comments for the file then you must have an m-file with the comments. I..e. so that the m-file, **data.m** contains the comments and the p-code file, **data.p** contains the MATLAB commands. There are two advantages with p-code files. One advantage is that the code is hidden from the reader, the other advantage is that the p-code file is executed much faster than m-files.

In connection to this MATLAB tutorial there exist a map with m-files. The map has the name MATLAB_FILES. The contents of this map are listed in the workspace by executing the command **what**.

```
 what matlab_files

M-files in directory m:\tex\fag\matlab_kurs\matlab_files

banana            ex1_id            fx_pendel         obfunc2
contents          ex1_lsim          hmax_euler        obfunc3
d2c_log           ex_euler_lsim     jacobi            plot_ex
dampfjaer         ex_for            maceps            plt_obfunc
data              ex_opc_plt        main_inv_pendel   plt_obfunc2
dread             ex_plt_3d         main_obfunc2 ex1_euler
fig_pendel        obfunc

MAT-files in directory m:\tex\fag\matlab_kurs\matlab_files

xy_data
```

Words like **who**, **whos**, **what** and **which** are also bulit in commands within MATLAB.

2.3 Defining vectors and matrices

The matrices in the above section can be defined within MATLAB as follows:

$\gg k = 2; m = 5; mu = 0.5;$
$\gg A = [0, 1; -k/m, -mu/m]$
$\gg B = [0; 1/m]$

Here, the coefficients k, m and mu must of course be defined in advance. The matrices vcan also be defined directly from numerical values

$\gg A = [0, 1; -0.4, -0.1]$

$\gg B = [0; 0.2]$

Semicolon, ;, are used to denote line shift or carriage return (CR). Note that space also can be used to separate numbers, i.e.,

$\gg A = [0\ \ 1; -0.4\ \ -0.1]$

or you can define the matrix by using carriage return as follows

$\gg A = [0, 1$
$-0.4, -0.1]$

There are a number of built in functions to define common types of matrices and elementary matrices. These functions belongs to the Elmat Toolbox. You can get an overview of the contents of the Elmat Toolbox by typing

$\gg$ **help elmat**

Someone of the functions which are often used are the **eye**, **zeros**, **ones**, **rand** and **randn** functions.

As an example the $n \times n$ identity matrix are defined as follows

$\gg I = \mathbf{eye}(n)$

eye can also be used to define the $n \times m$ matrix

$\gg I = \mathbf{eye}(n, m)$

A matrix E of randomly uniformed numerical values can be generated by the **randn.m** function, i.e.,

$\gg E = \mathbf{randn}(N, m)$

There are a number of built in functions which can be used for testing purposes in linear algebra. Some famous test matrices are the **hilb.m** and **pascal.m** functions for generating $n \times n$ test matrices.

$\gg A = \mathbf{hilb}(4)$
$\gg A2 = \mathbf{pascal}(4)$

The Hilbert-matrix is an example of a matrix which is numerically difficoult to invert. The Pascal matrix can be used to define the coefficients in the polynomial

$$(s+1)^5 = p_6 s^5 + p_5 s^4 + p_4 s^3 + p_3 s^2 + p_2 s + p_1 \tag{2.13}$$

The six coefficients in this polynomial can be found from the Pascal triangle, i.e., from the Pascal matrix. The roots of this polynomial can be computed numerically from the **roots.m** function. This can be done as follows

$\gg A = \mathbf{pascal}(6)$
$\gg p = \mathbf{diag}(\mathbf{rot90}(A))$
$\gg s = \mathbf{roots}(p)$

Note that the vector p is found as the anti-diagonal to the Pascal matrix. Compare the roots computed by the **roots** function and the exact roots of the polynomial.

The function **rot90** is used to rotate a matrix 90°. The functions **rot90**, **hilb** and **pascal** is a part of the ELMAT Toolbox.

Another method to compute eigenvalues (ore roots) is from a canonical form ore companion matrix, i.e.,

$\gg Ac = \mathbf{compan}(p)$
$\gg l = \mathbf{eig}(Ac)$

Here, l, is a vector of eigenvalues which normally should be equal as the vector, s, computed from the **roots.m** function above.

2.4 Format

Numerical computations are computed in double precision within MATLAB. The machine epsilon are a built in constant and denoted, **eps**. The default format used in the workspace and for file printing is the **short** format. You can switch between different formats by using the format commands

$\gg$ **format short**
$\gg$ **format long**
$\gg$ **format bank**

as well as a number of more special formats. A particular useful format for money calculations is the **bank** format.

With the

$\gg$ **format rat**

you can calculate exactly with fractions as $\frac{1}{2}$, $\frac{2}{3}$ etc.

2.5 Diary and printing to files

It is common to work a lot in the MATLAB command window. It can therefore be useful to write the workspace command history and the computed results to to a file. This can be done with the MATLAB command, **diary**.

$\gg$ **diary** *filnavn*

The contents in the MATLAB command window (workspace) will then be written to the file, *filename*. You can check in which map the file are stored by typing

$\gg$ **pwd**
$\gg$ **dir**

An example of such a diary file will be given in the next section. Note that the **pwd** command is a UNIX command which stands for Present Work Directory. In order to finish and turn of the writing to the diary file, the command **diary off** are used

$\gg$ **diary off**

If you want to start writing to this diary file later yoy turn it on by the command

$\gg$ **diary on**

2.6 How to define path

A new path is most simply defined by the built in path-browser. A new path can also be defined in the command window as follows

```
path(path,'i:\laerer\davidr\matlab_kurs\matlab_files')
```

Note that MATLAB always first are looking for m-files at the underlying directory (map). You can check which directory which is your primary (underlying) directory with the Present Work Directory, **pwd**, command.

You can change directory with the **cd** (Change Directory) command (which also is a UNIX command). An example is as follows

```
pwd
cd c:\temp
```

2.7 For, while and if sentences

Assume that we want to define a time-series vector, t, where the first element is t_0 and the last (final) element is t_f and that the distance between each time instant is, h. This can simply be defined within MATLAB as follows

$\gg t = t_0 : h : t_f;$

We are putting into some numerical values as follows $t_0 = 0$, $h = 0.1$ and $t_f = 150$. In order to illustrate we can plot the vector as follows.

$\gg t = 0 : 0.1 : 150;$
$\gg$ **plot(t,t)**

This gives a stright line in the x, y plane. If you want to check the number of elements in the t vector you can use the *length* command. In order to check the size of matrices within MATLAB we are using the **size** command. Try the following:

$\gg N =$ **length**(t)
$\gg$ **size(t)**

N is here the number of elements in the t vector. The same time series vector t can also be built up via a **for** loop as follows, e.e. se the m-file **ex_for.m**, i.e.

```
% ex_for
```

```
% Example for using a for-loop for generating a time series vector
% t=[0 0.1 0.2 ... 150]
t0=0; tf=150; h=0.1;      % Define step length,h, and first and final element in, t.
N=(tf-t0)/h+1;            % Number of elements in vector, t.
t=zeros(N,1);             % define space for an (N x 1) matrix, i.e., space for, t.

w=t0;                     % a "work" variable for the counter.
for i=1:N                 % i is denoted the values, 1,2,3,...,N
   t(i)=w;                % Store w in the vector, t.
   w=w+h;                 % Update the work variable.
end
```

Try to execute and run the m-file script, **ex_for.m**. You can also list the content of the file by typing

```
>> type ex_for
```

in the workspace (command window).

Lock at all the other variables which are generated by running the script by simply typing

```
>> who
```

in the workspace (command window).

In connection with for loops it is often usefull and practically to use the **if** and **while** sentences. If you want to jump out of a loop it is useful to use the **break** command.

Note that the time-series defined above can be directly defined by the built in MATLAB command **linspace**, i.e.

$\gg t = \mathbf{linspace}(0, 150, 1501)$

An example of how to use the **while** sentence is given in the m-file function **maceps** which can be used to compute the machine epsilon, **eps**. The machine epsilon, **eps**, is the smallest number such that the computer understand that $eps + 1 > eps$ This number may vary from computer architectures. Note that **eps** is a built in constant within MATLAB and you therefore do not have to compute it. Constants like $pi = \pi = 3.14..$ and the imaginary number $i = \sqrt{-1}$ eller $j = \sqrt{-1}$ are also predefined within MATLAB.

```
function meps=maceps
% maceps
% meps=maceps
% Function to compute the machine precision, eps, i.e. eps is
```

```
% the smallest number such that the machine detect that, eps+1>1.
% Usually the machine precision is about eps=1.0*e^(-16)
meps=1;
while meps+1 > 1
   meps=meps/2;
end
meps=meps*2;
```

This function has no input arguments. In order to check the number of input arguments to a function you can use the **nargin** function within a function.

The colon operator is a very useful in order to effectively programming in MATLAB. You can obtain help and info on operators like colon by simply typing

$\gg$ **help colon**

Note that if you are typing

$\gg$ **help** :

then you obtain a list of all the operators within MATLAB.

The colon oparator can e.g. be used to pick out a part of a vector or a matrix, i.e. the ten first elements in, t, are obtained by

$\gg t(1:10)$

ore from element number 1491 to the last element

$\gg t(1491:end)$

The colon operator is very useful to pick out submatrices from a larger matrix. The following example are illustrating this

```
 XY=ceil(randn(5,6))

XY =

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

 Y=XY(:,4:6)

Y =

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

```
0     0     1
1     1     2
```

The **ceil** function rounds the elements of a matrix to the nearest integers towards infinity. Other related functions are **fix** and the **floor** functions.

2.8 How to make figures and using the plot function

A central function used in order to plot vectors and matrices is the **plot** function. This function used to plot the function $y = \sin(t)$ is illustrated in the **plot_ex.m** m-file.

```
% plot_ex.m
% Example for using the plot function and related functions.

t=-pi:pi/100:pi;
y=sin(t);
plot(t,y)
axis([-pi pi -1 1])
xlabel('-\pi \leq \it t \leq \pi')
ylabel('y=sin(t)')
title('Plot of the sinus function, y=sin(t)')
grid
text(1,1/3,'\it{Note the symmetry}')
```

If you are executing the script, i.e. typing the command

$\gg$ **plot_ex**

then you obtain a plot or figure with $y = \sin t$ in the interval $-\pi \leq t \leq \pi$ in a separate figure window.

Note that the text along the x-axis is obtaines by the **xlabel** command. You can use LATEX ore TEX command in order to obtain fancy text strings and mathematical symbols like π. The content of the figure window can be written to the printer or a file with the **print** command. If the figure are to be used in a document or report it is practical to print the figure to a "encapsulated post-script" eps-file. This can be done by

$\gg$ **print -deps** *filname*

where the name **filename** is user specified. This figure are illustrated in Figure 2.1.

Other related plot functions are **subplot** and **stairs**.

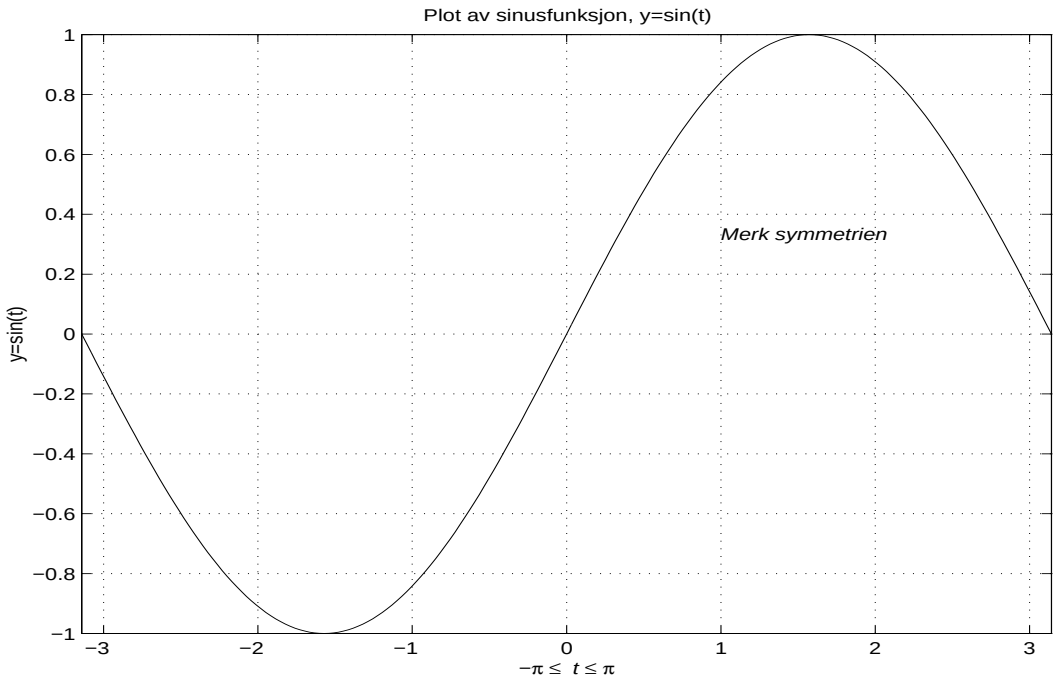

Figure 2.1: This Figure is generated by the **plot_ex.m** m-file script.

2.9 Computing eigenvalues and eigenvectors

MATLAB is designed to be able to perform most matrix factorizations and matrix decompositions in the field of linear algebra. Assume given a square matrix A, real valued or complex. We will in this section illustrate how we can compute an eigenvalue matrix L and an eigenvector matrix M such that $A = MLM^{-1}$, by using the MATLAB function **eig**.

Remember that the eigenvalues of a square matrix A may be computed as the roots of the characteristic equation $\det(\lambda_i I - A) = 0$ and the eigenvectors as the solution of the linear systems $Am_i = \lambda m_i$ where $i = 1, \ldots, n$. We then have that $M = \left[\begin{array}{ccc} m_1 & \cdots\cdots & m_n \end{array} \right]$ and $L = \left[\begin{array}{ccc} \lambda_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \lambda_n \end{array} \right]$

We are illustrating the use of the functions **eig**, **poly** and **roots** in the **diary** file shown below. Also note that the function **inv** can be used to invert an non-singular matrix.

```
A=[0,1;-0.4,-0.1]

A =

         0    1.0000
   -0.4000   -0.1000
```

```
eig(A)

ans =

  -0.0500 + 0.6305i
  -0.0500 - 0.6305i

[M,L]=eig(A)

M =

   0.8425 - 0.0668i   0.8425 + 0.0668i
        0 + 0.5345i        0 - 0.5345i


L =

  -0.0500 + 0.6305i        0
        0            -0.0500 - 0.6305i

M*L*inv(M)

ans =

   0.0000 - 0.0000i   1.0000 - 0.0000i
  -0.4000 + 0.0000i  -0.1000 + 0.0000i

p=poly(A)

p =

    1.0000    0.1000    0.4000

l=roots(p)

l =

  -0.0500 + 0.6305i
  -0.0500 - 0.6305i

diary off
```

We have in this example computed the eigenvalues and the eigenvectors of a matrix A with the function **eig**.

We have found the coefficients in the characteristic polynomial of matrix A,

i.e., $\det(\lambda(I - A)$, by use of the function **poly** and computed the roots of the characteristic equation, i.e. the solution of, $\det(\lambda I - A) = 0$, by using the function **roots**. Furthermore, matrix inversion have been done by using the function **inv**.

Some other important matrix factorizations and decompositions in linear algebra are implemented in the functions **qr**, **lu**, **chol**, **schur** and **svd**.

2.10 Simulation of linear dynamic systems with lsim

The damped spring system can simply be simulated by using the function **lsim.m**. This function gives an exact solution of the linear continuous dynamic system of the form

$$\dot{x} = Ax + Bu, \tag{2.14}$$
$$y = Dx + Eu, \tag{2.15}$$

where the initial value of the state vector, $x(t = 0) = x_0$ is known.

A MATLAB script (m-file) which simulates the damped spring system with the function **lsim** is implemented in the file **ex1_lsim.m**, i.e.,

```
% ex1_lsim.m
% Example of simulating a linear continuous dynamic state space model
% with the Control System Toolbox function, lsim.
%
data;                       % Define the constant parameters in the file, data.m

A=[0,1;-k/m,-mu/m];         % Matrices in the state space model.
B=[0;1/m];
D=eye(2);
E=[0;0];

h=0.1;                      % Specified step length.
T0=0; Tf=150;               % Start time, T0, and final time, Tf.
t=T0:h:Tf;                  % Time series vector, i.e. defining the time instants.
N=length(t);                % Number of parameters in the time vector.

f=F*zeros(N,1);
f=prbs1(N,75,300)+2;        % Make a force experiments on the system.
u=f-m*g*ones(N,1);          % Force input (influence) on the mass.
y=lsim(A,B,D,E,u,t,[0;0]); % Simulate model with lsim.
x=y(:,1)+x0;                % posisjon

plot(t,[x y(:,2)]), grid, xlabel('Continuous time [sec]')
```

Do this example by writing your own script or simply run the demo by the MATLAB command

≫ **ex1_lsim**

in the MATLAB command window.

Note that **lsim** is a function in the Control System Toolbox.

If you know the name of a function and wish to know which toolbox this function belongs to then you may got the answer by using the command **which**. Try to write

≫ **which lsim**

in the MATLAB command window.

2.11 Simulation by the explicit Euler method

Another useful and simple method for simulation of dynamic systems is to use the so called explicit EULERS's method. An example of this is shown and implemented in the file **ex1_euler.m**.

```
% ex1_euler.m
% Simulating the damped spring system with the explicit Euler method.
clear                        % Clear old definitions from the memory.
global k m x0 mu g F         % Define some global parameters in order to use them
data;                        % in the right hand side function, dampfjaer.m

h=0.1;                       % Step length for the Euler integration.
h=dread('Skrittlende ',h);   % Possibly change of parameter, h.
t=0:h:150;                   % Define the time horizon.
N=length(t);                 % Number of elements, N, in vector, t.
y=zeros(N,2);                % Define space for the simulation results.

x(1,1)=x_0;                  % Initial values for the states.
x(2,1)=v_0;

for i=1:N                    % The for-loop, loop over all integers, i=1, i=2, etc.
   y(i,:)=x';                % Sore the state vector, x, at time, t, in vector, y
   fx=dampfjaer(t(i),x);     % Define the right hand side in the differential equation.
   x=x+h*fx;                 % Integrate with explicit Euler.
end

plot(t,y), grid              % Plot the solution.
title('Simulation results of the damped spring system with explicit Euler')
xlabel('Continuous time [sec]')
```

The right-hand-side of the differential equation is defined in the function **dampfjaer**. Se the file **dampfjaer.m**, i.e.,

```
function fy=dampfjaer(t,y)
% fy=dampfjaer(t,x)
% Description: Function used to define the right hand side in damped spring model.
% Input variable: x  - the state vector at time, t.
% output variable:  fy - the right hand side in the differential eq. dot(x)=fy.

%global k m x0 mu g F                 % May define variables as globel.
k=2; m=5; x0=1; mu=0.5; g=9.81; F=0;

x=y(1);
v=y(2);
fx=v;
fy=-k*(x-x0)/m - mu*v/m -g + F/m;
fy=[fx;fy];
```

2.12 Comparing the results by using lsim and explicit Euler

Plotting the results by using the matlab Control Toolbox function **ex1_lsim** with the results by using the explicit **ex1_euler** method we se that the results from the Euler integration is relatively inaccurate when using step-length $h = 0.1$. This is illustrated in Figure 2.2. However, by reducing the step-length used in **ex1_euler** to e.g. $h = 0.01$ then the results are approximately the same from the two options. Note however that the computing time is considerably increased by reducing the step-length. Computing tome may be checked for by using

$\gg$ **tic; ex1_lsim; toc**
$\gg$ **tic; ex1_euler; toc**

The time consumed is here 0.34 for **ex1_lsim** and 4.24 for **ex1_euler** with $h = 0.01$. Using **lsim** is here $4.24/0.34 = 12.5$ times faster than using **euler**. Furthermore note that **lsim** gives exact simulation results of a linear continuous time dynamic system.

2.13 Discretization of continuous models

Using explicit Euler, **lsim** or other methods for simulating continuous dynamic models is the same as obtaining a discrete time model and simulate this discrete model. A discrete state space model equivalent of a linear continuous time model

$$\dot{x} = A_c x + B_c u, \tag{2.16}$$

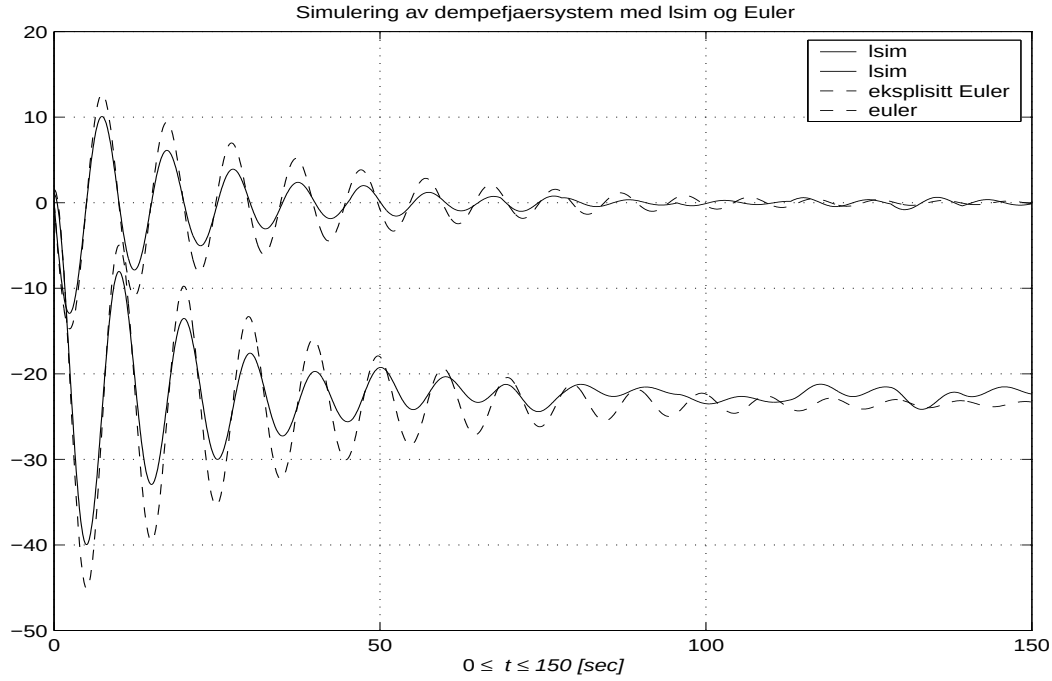

Figure 2.2: This Figure is generated by the m-file script **fig_euler_lsim.m**. The figure shows results by simulating the damped spring system by using explicit Euler and the function **lsim**.

$$y = D_c x + E_c u, \tag{2.17}$$

with initial state $x(t=0) = x_0$ may be written as

$$x_{k+1} = Ax_k + Bu_k, \tag{2.18}$$
$$y_k = Dx_k + Eu_k, \tag{2.19}$$

with initial state $x_{k=0} = x_0$ given. By using the explicit Euler method we have that

$$A = I + hA_c, \tag{2.20}$$
$$B = hB_c, \tag{2.21}$$
$$D = D_c, \tag{2.22}$$
$$E = E_c, \tag{2.23}$$

where $h > 0$ is the step-length. **lsim** is using an exact solution of the continuous time model and we have,

$$A = e^{A_c h}, \tag{2.24}$$
$$B = \int_0^h e^{A_c \tau} B d\tau = A_c^{-1}(e^{A_c h} - I)B_c, \tag{2.25}$$

when the square matrix A_c is non-singular (invertible).

Remember that the explicit Euler method is inaccurate for large step-length parameters h. The largest possible step-length which may be used with explicit Euler may be computed by require that all eigenvalues of the transition matrix $A = I + hA_c$ should be less than one, i.e.,

$$\mid \lambda(I + hA_c) \mid < 1. \tag{2.26}$$

The largest step-size which may be used in connection with the explicit Euler method is then

$$0 < h_{\max} < \frac{2}{\max \mid \lambda(A_c) \mid} \tag{2.27}$$

if the wigenvalues of A are real.

If the eigenvalues of A are complex conjugate, e.g., $\lambda = \alpha \pm j\beta$, then the largest step-length would be

$$0 < h_{\max} < \frac{2 \mid \alpha \mid}{\alpha^2 + \beta^2} \tag{2.28}$$

This formula may be deduced from the matrix

$$A = \begin{bmatrix} \alpha & -\beta \\ \beta & \alpha \end{bmatrix} \tag{2.29}$$

which have eigenvalue given by $\lambda = \alpha \pm j\beta$. We may compute this step-length in MATLAB by

$\gg l = \mathbf{eig}(A)$
$\gg hmax = 2 * \mathbf{abs}(\mathbf{real}(l(1)))/(\mathbf{real}(l(1))^2 + \mathbf{imag}(l(1))^2)$

This may be implemented in a MATLAB m-file function as in the file **hmax_euler.m**.

```
function hmax=hmax_euler(A)
% hmax_euler
% Function used to compute the maximum step length when using
% the explicit Euler method for simulation of a system
% dot(x)=Ax der A =[alpha,-beta;beta,alpha]
% This system have complex conjugate eigenvalues,
% l=alpha +- j beta.
l=eig(A);
hmax=2*abs(real(l(1)))/(real(l(1))^2+imag(l(1))^2);
```

For the damped spring system we find that the maximum step-length which may be used when using the explicit Euler method is $h \leq h_{\max} = 0.25$. For the pendulum model we have $h \leq h_{\max} = 0.0255$. In order to achieve acceptable accuracy when using the explicit Euler method, the step-length have to be chosen smaller, e.g. about $h = \frac{h_{\max}}{10}$.

This shows that the problem of computing eigenvalues of matrices is a central problem in connection with simulation of dynamic systems. The explicit Euler method is much used in practice due to its simplicity. Using the explicit Euler method gives a simple solution and results which usually is accurate enough compared to the uncertainty in the mathematical model of the real world system. The explicit Euler method is in particular a very simple solution in case of non-linear continuous time dynamic models.

In case that the model is linear then it may be better to use an exact solution as e.g. given by the **lsim** function. Note that an exact discrete time model may be found directly in MATLAB by using the function

$\gg [A, B] = \mathbf{c2d}(A_c, B_c, h)$

The resulting discrete state space model matrices A, B and $D = D_c$ and $E = E_c$ then gives a complete discrete time dynamic model as given by the equations (2.18) og (2.19). Related functions with this is given in **c2dm**, **d2c** and **d2cm**.

2.14 Matrix functions

There exist lot of built in matrix functions in MATLAB. Most of these functions is implemented in the MATFUN toolbox. An overview of these functions is obtained by eriting

$\gg$ **help matfun**

in the command window.

We will here give an example of using the function **expm** which may be used for compputing the matrix exponential $A = e^{A_c h}$ which mwas used in the section above.

$\gg A = \mathbf{expm}(A_c * h)$

The corresponding B matrix in the discrete time model may then be computed as

$\gg B = \mathbf{inv}(A_c) * (A - \mathbf{eye}(2)) * B_c$

2.15 Simulation by using more accurate integration methods

In the MATLAB toolbox FUNFUN there is a number of integration methods which are very accurate and efficient fo0r the simulation of non-linear continuous time dynamic models. A non-linear Odinary Differential Equation (ODE) may be written as

$$\dot{x} = f(x) \tag{2.30}$$

with known initial state vector $x(t = t_0) = x_0$. This ODE equation may be discretized by a so called θ-method, d.v.s.,

$$\frac{x_{k+1} - x_k}{h} = (1 - \theta)f(x_{k+1}) + \theta f(x_k). \tag{2.31}$$

As we se by choosing $\theta = 1$ then we obtain the explicit Euler method., $\theta = 0$ gives the implicit Euler method and $\theta = \frac{1}{2}$ gives the trapezoid method.

As we se when $\theta \neq 1$ then we have, when $f(x)$ is non-linear, solve a non-linear equation with respect to the new state på x_{k+1} at each new time step k. This may be solved by using e.g. the Newton-Raphsons method for solving non-linear equations.

The trapezoid method, i.e., the θ-method with $\theta = \frac{1}{2}$, is implemented in the FUNFUN function **ode23t**. Experience shows that when using $\theta = 0.52$ we obtain a method efficient for so called stiff dynamic systems. A similar method is implemented in the FUNFUN function **ode15s**

By using these functions we have to implement the right hand side in the differential equation, i.e. the function $f(x)$ in a m-file function with the same syntax as demonstrated in **dampfjaer.m**.

As an example we simulate the damped spring system by using the **ode45** function, d.v.s,

$\gg [t, x] = \textbf{ode45}('dampfjaer', [0\ 150], [1.5; 0]);$

We may also use the stiff-solver **ode15s** as follows,

$\gg [t, x] = \textbf{ode15s}('dampfjaer', [0\ 150], [1.5; 0]);$

The non-linear pendulum model may be simulated as follows

$\gg [t, x] = \textbf{ode15s}('fx\_pendel', [0\ 150], [pi/2; 0]);$

A stiff ODE system is a system where the ratio between the largest time constant and the smallest time constant is large. This is equivalent to a system where the ratio between the largest eigenvalue and the smallest eigenvalue is a large number. By large we mean a ratio in the range 100 to 10000 or larger. The relationship between a time constant T and an eigenvalue λ is,

$$T = \frac{1}{\lambda} \tag{2.32}$$

Note that the eigenvalues have to be computed from a linearization of the non-linear function $f(x)$.

Note that some of the ODE solvers are working with a varable step-length h. This means that the step length h and the number of computed solutions for x in the simulation time horizon $0 \leq t \leq 150$ in the above examples is not constant. Small values for h are required when there are much variations in the

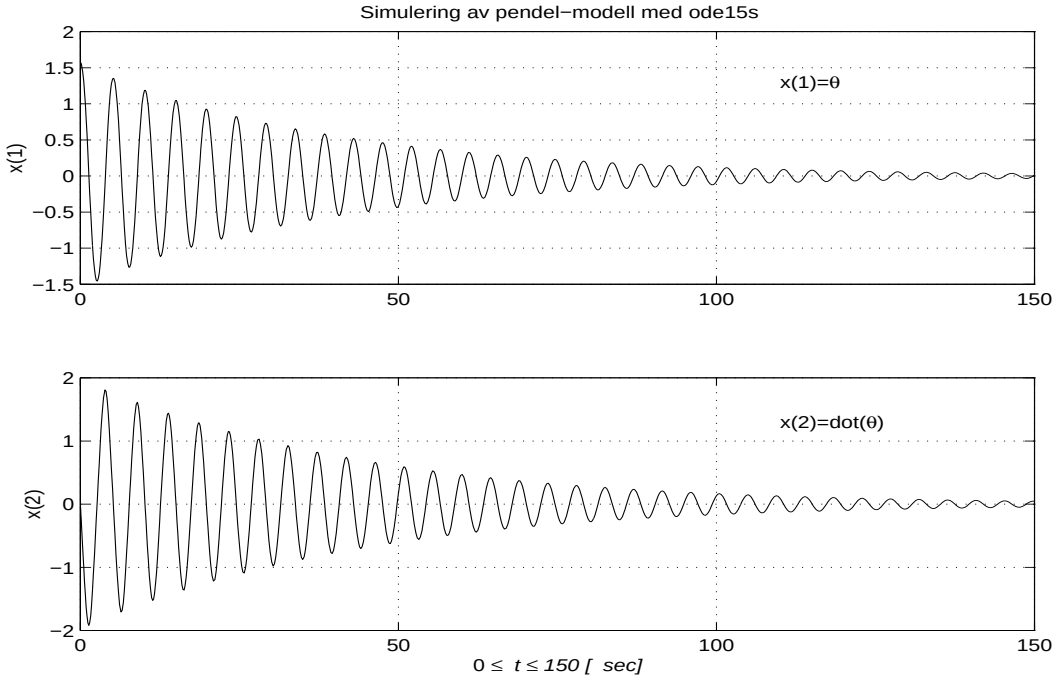

Figure 2.3: This Figure is generated by the m-file script **fig_pendel.m**. We have here simulated the pendulum model by using the function **ode15s**.

x profile and larger step-lenght h may be chosen when x is more constant. We may however force the ODE solver to compute a solution at uniform specified time instants. If we want solutions at each specified time instants as, e.g., $t = 0$,$t = h$, $t = 2h$ and so on we may modify the ODE function call as follows

$\gg [t, x] = \mathbf{ode15s}('fx\_pendel', 0 : h : 150, [pi/2; 0]);$

where h is a specified step-length parameter, e.g., $h = 0.1$.

Chapter 3

Matrix Methods in Courses

3.1 Optimalisering

Vi skal i dette avsnittet illustrere hvordan vi kan finne minimum av funksjoner v.h.a. standardfunksjonene **fminbnd** og **fminsearch** i FUNFUN toolboksen. Merk at disse funksjonene erstatter de tidligere funksjonene **fmin** og **fmins**.

Følgende eksempler illustrerer den enkleste bruken av disse funksjonene

```
 x=fminbnd('x^2+2*x+1',-10,10)

x =

   -1.0000
```

Vi har her funnet at minimum av funksjonen $J(x) = x^2 + 2x + 1$ er gitt for $x = -1$. Et optimaliseringsproblem i to variable løses som i følgende eksempel

```
 x=fminsearch('2*(x(1)-1)^2+x(1)-2+(x(2)-2)^2+x(2)',[1;1])

Optimization terminated successfully:
 the current x satisfies the termination criteria using OPTIONS.TolX of 1.000000e-004
 and F(X) satisfies the convergence criteria using OPTIONS.TolFun of 1.000000e-004


x =

    0.7500
    1.5000
```

Vi har her funnet at minimum av funksjonen $J(x_1, x_2) = 2(x_1 - 1)^2 + x_1 - 2 + (x_2 - 2)^2 + x_2$ er gitt ved $x_1 = 0.75$ og $x_2 = 1.5$.

Vi skal nå løse et mer avansert optimaliseringsproblem. Slike avanserte optimaliseringsproblem studeres b.l.a. i faget Avansert reguleringsteknikk ved TF. Følgende funksjon

$$J(G) = \int_0^7 (x^T Q x + u^T P u) dt \tag{3.1}$$

der

$$\dot{x} = Ax + Bu, \tag{3.2}$$
$$u = Gx, \tag{3.3}$$

skal minimaliseres m.h.t. G. Initialtilstanden $x(t = 0) = x_0$ er gitt.

Eksempel 3.1 *Tallverdiene for dette eksemplet er* $A = -0.1$, $B = 1$, $Q = 1$, $P = 1$ *og* $x_0 = 5$.

For å benytte **fminbnd** og **fminsearch** er det hensiktsmessig å definere funksjonen J i en M-fil funksjon. Denne kan være som definert i funksjonen **obfunc.m**, d.v.s.,

```
function J=obfunc(g)
% OBFUNC
% Obfunc beregner kriterieverdien
% J=int_0^7 (x'*Qx+u'*P*u) dt
% der
% dot(x)=ax+bu, u=gx, x(t=0)=x0.
% Parametre: a=-0.1, b=1, q=1, p=1, x0=5.
a=-0.1; b=1; x0=5;          % systemparametre.
q=10; p=1;                   % vektparametre i kriteriet.

imet=2;

if imet == 1                % Numerisk kriteriefunksjon.
h=0.005;                    % Skrittlengde.
t=0:h:7;                    % Oppdeling av tidsaksen.
N=length(t);                % Lengden av vektoren t.

J=0;                        % Integrerer kriteriet med eksplisitt Eulers metode.
for i=1:N
   x=exp((a+b*g)*t(i))*x0; % Tilstanden, x(t), dvs. x ved tidspunkt t.
   L=x'*(q+g'*p*g)*x;       % Funksjon under integral-tegnet (integranden).
   J=J+h*L;                 % eksplisitt Euler step.
end

elseif imet == 2            % Analytisk kriteriefunksjon.
   tf=7;
```

```
   Wo=(q+p*g^2)*(exp(2*(a+b*g)*tf)-1)/(2*(a+b*g));
   J=x0'*Wo*x0;
end
```

Vi kan nå beregne et estimat av minimumspunktet ved hjelp av **fminbnd** i MATLAB med kommandoene

```
  flops(0)
  g=fminbnd('obfunc',-5,0)

Optimization terminated successfully:
 the current x satisfies the termination criteria using OPTIONS.TolX of 1.000000e-004


g =

   -0.9005

 flops

ans =

      170249
```

Benytter vi funksjonen **fminsearch** får vi

```
 g=fminsearch('obfunc',-0.5)

Optimization terminated successfully:
 the current x satisfies the termination criteria using OPTIONS.TolX of 1.000000e-004
 and F(X) satisfies the convergence criteria using OPTIONS.TolFun of 1.000000e-004


g =

   -0.9005

 flops

ans =

      506796
```

Vi har her benyttet funksjonen **flops** for å beregne antall multiplikasjoner, subtraksjoner og addisjoner (d.v.s. antall flyttallsoperasjoner) som er benyttet med metodene. Som vi ser så gir begge metodene samme svar men **fminbnd** er ca. 3 ganger så rask som **fminsearch**. Det er ofte en god ide å plotte

funksjonen som skal minimaliseres, dersom dette er mulig. Man kan dermed lese av et estimat for minimumsverdien. Kriteriefunksjonen er vis i figur 3.1

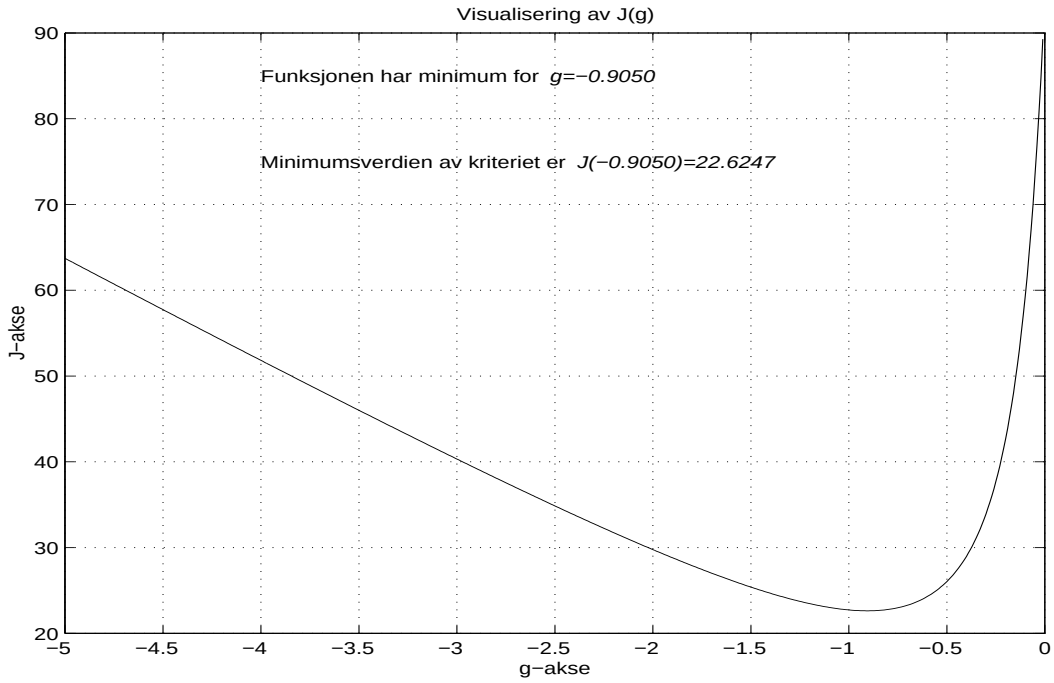

Figure 3.1: Figuren viser kriteriefunksjonen 3.1 med tallverdier som i eksempel 3.1. Figuren er generert av M-filen **plt_obfunc.m**.

Det kan fra teorien om optimalregulering vises at den eksakte løsningen på dette problemet er $G = -0.9050$. Grunnen til at vi ikke får eksakt løsning er at vi integrerer med eksplisitt Eulers metode. En mer nøyaktig løsning kan vi få ved å redusere skrittlengden h i funksjonen **obfunc**. En M-fil for plotting av resultatene er gitt i **ex_opc_plt.m**. Kjøring av denne resulterer i Figur 3.2.

En mer generell implementering av denne løsningen er gitt i funksjonene **main_obfunc2.m** og **obfunc2.m**. **main_obfunc2.m** er et hoved-script som benytter funksjonen **obfunc2.m**. Uheldigvis så er **obfunc2.m** meget regnekrevende, men siden problemet er lineært så eksisterer det en langt raskere implementering. Se **obfunc3.m**. Vi har i denne sammenheng implementert et eksempel der vi ønsker å finne en optimal PI-regulator for et system.

Eksempel 3.2 *I dette eksemplet benytter vi*

$$A = \begin{bmatrix} -0.1 & 0 \\ -1 & 0 \end{bmatrix}, \quad B = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, Q = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}, \quad P = 1, \; x_0 = \begin{bmatrix} 5 \\ 2 \end{bmatrix}, \quad (3.4)$$

En kjøring gir

```
g=fminsearch('obfunc3',[0.5,-0.5])
```

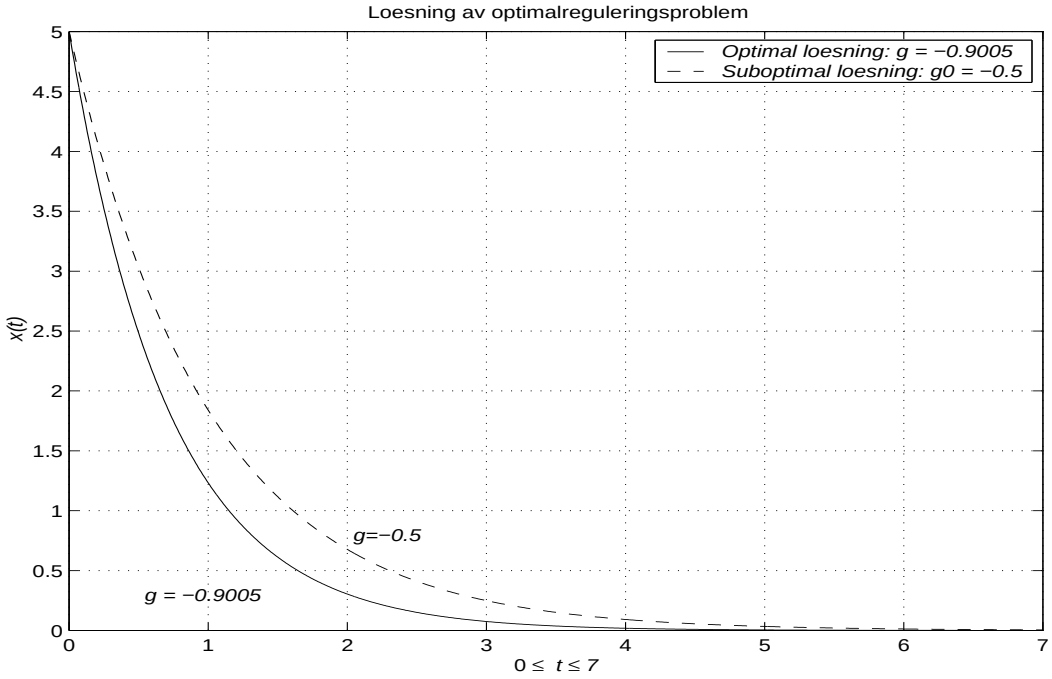

Figure 3.2: Figuren viser den optimale trajektoren $x(t)$ i systemet $\dot{x} = Ax + Bu$ og $u = Gx$ der G minimaliserer 3.1 og med tallverdier som i eksempel 3.1. Figuren er generert av scriptet **ex_opc_plt.m**.

```
g =

   -1.9016    0.9961
```

Bruk av **obfunc2** gir her samme resultat men er altså langt mer beregningskrevende. Det er imidlertid viktig å merke seg at **obfunc2** er generell i den forstand at den enkelt kan modifiseres til et kriterium som er ikke-kvadratisk. Det kan vises at den eksakte optimale løsningen er uavhengig av x_0. Dersom vi definerer kriteriet 3.1 over en uendelig tidshorisont, d.v.s. integrerer fra null til uendelig, så kan vi vise at den optimale løsningen er gitt ved $G = \left[\begin{array}{cc} g_1 & g_2 \end{array}\right] = \left[\begin{array}{cc} -1.9025 & 1.0000 \end{array}\right]$ Denne kan finnes ved **lqr** funksjonen i Control Systems Toolbox'en. Vi gjenngir funksjonene **main_obfunc2.m** og **obfunc2.m**.denne nedenfor,

```
% main_obfunc2
% Script for loesning av optimalreguleringsproblem.
clear global
global a b q p x0 h t N
p_text=['iex=1 : skalart optimaliserings-problem (P-regulator)';
        'iex=2 : todimensionalt problem (optimal PI-regulator)';
        'iex=3 : En langt raskere implementering av iex=2     '];
disp(p_text)
iex=2;
```

```
iex=dread('Example number: 1-3',iex);
if iex == 1
   a=-0.1; b=1; x0=5;          % systemparametre.
   q=1; p=1;                   % vektparametre i kriteriet.
   h=0.005;                    % Skrittlengde.
   t=0:h:7;                    % Oppdeling av tidsaksen.
   N=length(t);
   g0=-0.5;
   g=fminsearch('obfunc2',g0)
elseif iex == 2
   a=[-0.1 0;-1,0]; b=[1;0]; x0=[5;2];
   q=[2,0;0,1]; p=1;
   h=0.001;                    % Skrittlengde.
   t=0:h:7;                    % Oppdeling av tidsaksen.
   N=length(t);
   g0=[-1 1];
   g=fminsearch('obfunc2',g0) % NB: raskere med obfunc3.
elseif iex == 3
   a=[-0.1 0;-1,0]; b=[1;0]; x0=[5;2];
   q=[2,0;0,1]; p=1;
   h=0.001;                    % Skrittlengde.
   t=0:h:7;                    % Oppdeling av tidsaksen.
   N=length(t);
   g0=[-1 1];
   g=fminsearch('obfunc3',g0)

end


n=length(a);
x=lsim(a+b*g,zeros(n,1),eye(n),zeros(n,1),t,t,x0);
if iex == 1
   plot(t,x), grid, title('iex=1: optimal P-regulator')
elseif iex == 2
   subplot(2,1,1), plot(t,x(:,1)), grid,
   title('iex=2: optimal PI-regulator')
   text(2.2,3,'x_1 - systemtilstand')
   subplot(2,1,2), plot(t,x(:,2)), grid
   text(2.2,0.5,'x_2 - regulatortilstand')
   xlabel('0 \leq \it t \leq 7 [\it sec]')

end
% END MAIN_OBFUNC2

function J=obfunc2(g)
% OBFUNC2
% Obfunc beregner kriterieverdien
```

```
% J=int_0^7 (x'*Qx+u'*P*u) dt
% der
% dot(x)=Ax+Bu, u=Gx, x(t=0)=x0.
% Parametre: global a, b, q, p, x0, h, t, N
% Merk: dette er en mer generell versjon av OBFUNC.M
% Se tilhoerende hovedscript, MAIN_OBFUNC2.M

global a b q p x0 h t N
J=0;                        % Integrerer kriteriet med eksplisitt Eulers metode.
for i=1:N
   x=expm((a+b*g)*t(i))*x0; % Tilstanden, x(t), dvs. x ved tidspunkt t.
   L=x'*(q+g'*p*g)*x;       % Funksjon under integral-tegnet (integranden).
   J=J+h*L;                 % eksplisitt Euler step.
end
% END OBFUNC2
```

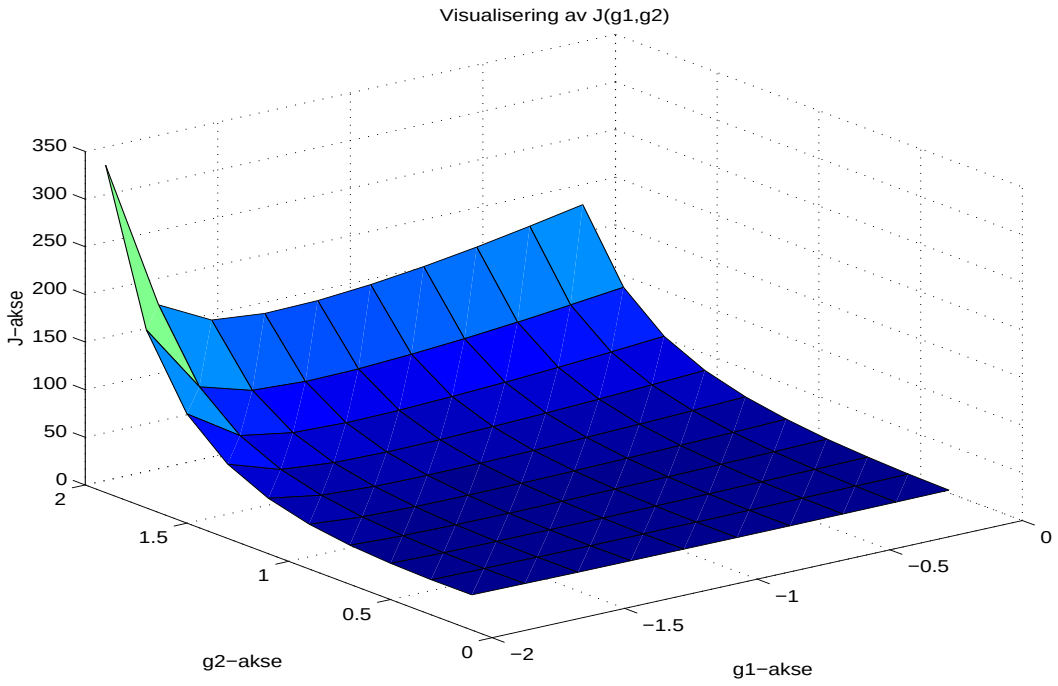

Figure 3.3: Figuren viser kriteriefunksjonen 3.1 med tallverdier som i eksempel 3.2. Figuren er generert av M-filen **plt_obfunc2.m**.

3.2 Numerisk linearisering og Jacobi matrisen

Mange av metodene vi har benyttet for simulering og optimalisering benytter seg av numerisk linearisering og gradientberegning. Anta at vi har gitt en funksjon $f(x,t) \in \mathbb{R}^m$ som ønskes linearisert om punktene t og $x \in \mathbb{R}^m$. Vi har

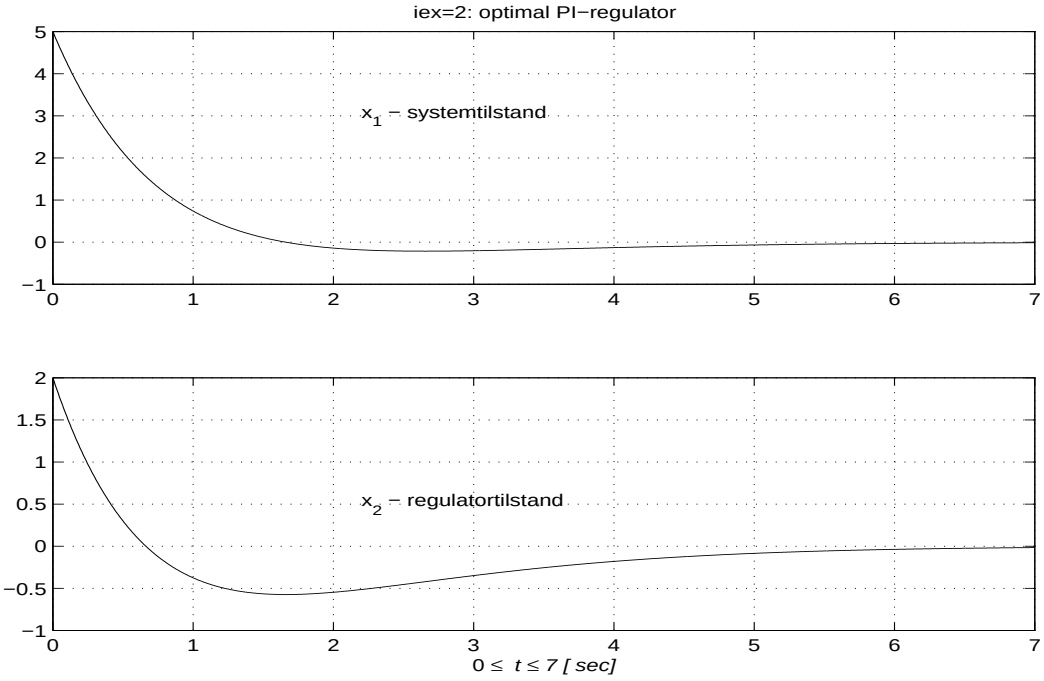

Figure 3.4: Figuren viser den optimale trajektoren $x(t)$ i systemet $\dot{x} = Ax + Bu$ og $u = Gx$ der G minimaliserer 3.1 og med tallverdier som i eksempel 3.2. Figuren er generert av scriptet **main_obfunc2.m**.

da at Jacobimatrisen er definert som

$$\frac{\partial f}{\partial x^T} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \in \mathbb{R}^{m \times n} \tag{3.5}$$

En algoritme for å implementere en metode for numerisk beregning av Jacobimatrisen er implementert i funksjonen **jacobi.m**. Denne er gjenngitt her

```
function A=jacobi(fx_fil,t,x)
% JACOBI
% Funksjon for aa beregne Jacobi-matrisen. A=df/dx for funksjonen
% fx=f(t,x), dvs. en linearisering av fx=f(t,x) om verdiene t og x.
% Syntaks:
% A=jacobi('fx_fil',t,x)
% Inngangsparametre:
% fx_fil - fil for definering av fx-funksjonen.
%          Syntax: function fx=fx_fil(t,x)
% t      - tidspunkt
% x      - vektor av samme dimensjon som i M-filen fx_fil.m
% Utgangsparametre:
```

```
% A      - Jacobimatrisen

h=1e-5;
n=length(x);
xh=x;
fx=feval(fx_fil,t,x);
m=length(fx);
A=zeros(m,n);
for i=1:n
   xh(i)=x(i)+h;
   fxh=feval(fx_fil,t,xh);
   xh(i)=xh(i)-h;
   A(:,i)=(fxh-fx)/h;
end
```

MATLAB gjør det mulig å implementere Jacobimatrisen med kun en for løkke. Vi kan nå beregne den lineariserte matrisen for pendel-modellen ved kommandoen

$\gg A = \mathbf{jacobi}('\mathbf{fx\_pendel}', [], [0; 0])$

Funksjonen kan selvsagt også benyttes dersom funksjonen $f(x, t)$ er lineær. Et eksempel på dette er

$\gg A = \mathbf{jacobi}('\mathbf{dampfjaer}', [], [0; 0])$

3.3 Løsning av ulineære likninger med Newton Raphsons metode

Gitt en funksjon $f(x) \in \mathbf{R}^m$ som er avhengig av en vektor $x \in \mathbf{R}^n$. Anta at vi ønsker å finne løsningen x^* slik at

$$f(x^*) = 0. \tag{3.6}$$

En Taylor-rekkeutvikling av $f(x)$ om løsningen, x^*, gir

$$f(x^*) = f(x) + \left.\frac{\partial f}{\partial x}\right|_x (x^* - x) = 0. \tag{3.7}$$

Vi løser m.h.t. x^* og får

$$x^* = x - \left(\left.\frac{\partial f}{\partial x}\right|_x\right)^{-1} f(x). \tag{3.8}$$

Dette kan benyttes for å lage følgende iterasjons-ligning

$$x_{i+1} = x_i - \left(\left.\frac{\partial f}{\partial x}\right|_{x_i}\right)^{-1} f(x_i). \tag{3.9}$$

Iterasjonsprosessen (3.9) refereres til som Newton-Raphsons metode. Iterasjonsprosessen starter ved at vi spesifiserer en startverdi, x_0, og itererer (3.9) inntil

$f(x_i) \leq$ tol der tol er en spesifisert toleranse for hva som er null. En mye benyttet numerisk toleranse er tol $= \sqrt{\text{meps}}$ der meps er maskinpresisjonen. Maskinpresisjonen er som vi har funnet tidligere gitt ved meps $= 1/2^{52} \approx 2.22 \cdot 10^{-16}$. Dette gir tol $= 1.5 \cdot 10^{-8}$. Det som gjør Newton-Raphsons metode så effektiv er den raske konvergenshastigheten nær løsningen (2. ordens konvergens). Anta at dersom x er nær løsningen, x^*, d.v.s., slik at $x^* = x + \varepsilon$. Dette gir

$$f(x+\varepsilon) = f(x) + \left.\frac{\partial f}{\partial x}\right|_x \varepsilon \approx 0, \tag{3.10}$$

fordi koeffisientene foran de andre (høyere ordens) leddene i Taylor-rekken er proporsjonal med ε^2 som er tilnærmet lik null for små ε.

En implementering av Newton-Raphsons metode for numerisk løsning av $f(x) = 0$ er gitt i funksjonen **newrap.m**. Denne er gjenngitt her

```
function x=newrap(fname,t,x0)
% newrap
% Purpose: Solve f(t,x)=0 for x using Newton Raphsons method.
% x=newrap('fname',t,x0)
% On input:
% fname  - Filname for definition of function, f=f(t,x).
% x0     - Initial guess for the solution
% t      - Dummy parameter.
% On output:
% x      - The solution x, for f(t,x)=0.

% Author: David Di Ruscio, october 2000.
max_iter=500;              % Maximum number of iterations.
x=x0;
tol=sqrt(eps);             % Tolerance for f=f(t,x) = 0.
er_f=100; it=1;            % Initialization of some parameters.
while er_f>tol
  f=feval(fname,t,x);      % Compute functional value.
  A=jacobi(fname,t,x);     % Compute the Jacobian df/dx=A numerically.
  x=x-pinv(A)*f;           % x_k+1=x_k - inv(df/dx)*f(x)
  er_f=norm(f);
  it=it+1;
  if it>=max_iter; break; end
end
```

Vi skal nå illustrerer bruken ved å finne det stasjonære konsentrasjons-profilet, x, i en destillasjonskolonne med ett trinn, koker og akkumulator. Først må vi definere funksjonen $f(x)$ i en fil med syntaks $f =$ fcol3_a$(0, x)$. Du kan selvsagt velge ett annet filnavn enn fcol3_a. Vi har

```
x0=[0.1;0.2;0.3];
```

```
x=newrap('fcol3\_a',0,x0)
x =

   0.0500
   0.4591
   0.9500
```

3.4 Datamodellering i statiske systemer

En vanlig problemstilling i mange fagdisipliner er å beregne et estimat (eller løsning) for parametervektoren B i et statisk system

$$Y = XB + E \tag{3.11}$$

og der vi bare kjenner datamatrisene X og Y. E er en matrise med støy. Vi antar her at sammenhengen mellom datamatrisene er gitt ved

$$\overbrace{\begin{bmatrix} 0.35 \\ 0.34 \\ 4.32 \\ 0.97 \end{bmatrix}}^{Y} = \overbrace{\begin{bmatrix} 0.16 & 0.10 & 0.08 \\ 0.17 & 0.11 & 0.09 \\ 2.02 & 1.29 & 1.0 \\ 0.5 & 0.4 & 0.1 \end{bmatrix}}^{X} \overbrace{\begin{bmatrix} 1.00 \\ 1.00 \\ 1.00 \end{bmatrix}}^{B} + \overbrace{\begin{bmatrix} 0.01 \\ -0.03 \\ 0.02 \\ -0.03 \end{bmatrix}}^{E}. \tag{3.12}$$

Problemet vi skal studere er å beregne løsninger for B med utgangspunkt i kjente datamatriser X og Y. Datamatrisene X og Y er lagret på filen **xy_data.mat**. Matrisene lan lastes inn i matlab ved kommandoen

```
load xy_data
```

Dersom vi benytter minste kvadraters metode (Ordinary Least Squares (OLS) får vi OLS løsningen

```
>>B_ols=pinv(X'*X)*X'*Y

B_ols =

    7.3310
   -6.1018
   -2.5452

norm(B_ols)

ans =

    9.8718
```

Vi ser at dette estimatet er svært forskjellig fra de virkelige parametrene som er gitt ved $B = \begin{bmatrix} 1.00 & 1.00 & 1.00 \end{bmatrix}^T$. Vi legger og merke til at normen til løsningen er $\| B \| = \sqrt{3} = 1.7321$. Dette er vesentlig forskjellig fra normen til OLS løsningen som er $\| B_{OLS} \| = 9.8718$.

Dersom vi benytter Partial Least Squares (PLS) metoden får vi dersom vi benytter en komponent

```
 K1=X'*Y

K1 =

    9.2820
    6.0332
    4.4756

 B1_pls=K1*inv(K1'*X'*X*K1)*K1'*X'*Y

B1_pls =

    1.2912
    0.8393
    0.6226

 norm(B1_pls)

ans =

    1.6611
```

Vi ser at denne løsningen er vesentlig bedre enn OLS løsningen. Benytter vi PLS metoden med to komponenter får vi

```
 K2=[X'*Y X'*X*X'*Y]

K2 =

    9.2820   66.7316
    6.0332   43.4169
    4.4756   32.0991

 B2_pls=K2*inv(K2'*X'*X*K2)*K2'*X'*Y

B2_pls =

    1.2600
    0.5984
```

```
    1.0132

 norm(B2_pls)

ans =

    1.7240
```

3.5 Simulering av diskret modell

Vi kan lage en diskret modell med utgangspunkt i en kontinuerlig modell som vist i avsnitt 2.13. En eksakt diskret modell for dempefjærsystemet er da

$\gg [a, b] = \mathbf{c2d}(A, B, 0.5)$
$\gg d = \mathbf{eye}(2)$
$\gg e = \mathbf{zeros}(2, 1)$

Vi har her benyttet en skrittlengde (eller samplingsintervall) $h = 0.5$. Vi benytter for enkelthetsskyld små bokstaver som symboler for matrisene i den diskrete modellen. Utførelse av kommandoene over i MATLAB vil gi resultatene

```
[a,b]=c2d(A,B,0.5)

a =

    0.9512    0.4796
   -0.1918    0.9033


b =

    0.0244
    0.0959

d=eye(2)

d =

     1     0
     0     1

e=zeros(2,1)

e =

     0
     0
```

```
diary off
```

Vi kan nå studere responsene etter at systemet er påtrykt et enhetssprang og en impuls ved

$\gg$ **figure**(1)
$\gg$ **dstep**(a, b, d, e)
$\gg$ **figure**(2)
$\gg$ **dimpulse**(a, b, d, e)

Dersom vi har en diskret lineær tilstandsrommodell så kan denne enkelt simuleres v.h.a. funksjonen **dlsim**. Dette gjøres ved kommandoen

$\gg [Y, X] =$ **dlsim**(a, b, d, e, U, x_0)

Merk at diskrete data kan plottes med funksjonen **stairs**, men ofte benyttes bare **plot**.

3.6 Identifikasjon av diskret tilstandsrommodell fra data

Dersom vi kjenner inngangsdata, U, og utgangsdata, Y, for et system så er det enkelt å identifisere en dynamisk modell for systemet. Dette kan vi f.eks. enkelt gjøre v.h.a. **dsr** funksjonen, d.v.s.

$\gg [A, B, D, E] =$ **dsr**(Y, U, L)

der L er et heltall som spesifiseres av brukeren. Se appendiks A. **dsr** er en funksjon i D-SR Toolbox for MATLAB. Man kan få tilgang til denne ved å henvende seg til forfatteren.

3.7 Plotting av tredimensjonale figurer

Vi gjengir her et klassisk "sombrero" eksempel.

```
[x,y]=meshgrid(-8:0.5:8);
r=sqrt(x.^2+y.^2)+eps;
z=sin(r)./r;
mesh(z)
```

z er her en 33×33 matrise av tall. Denne danner grunnlaget for $x - y$ aksen i horisontalplanet. z aksen skaleres automatisk av **mesh**. Merk at maks og min verdiene av z kan finnes ved

```
ans =
```

```
     1

 min(z(:))

ans =

   -0.2172
```

Sombreroen er vist i Figur 3.5.

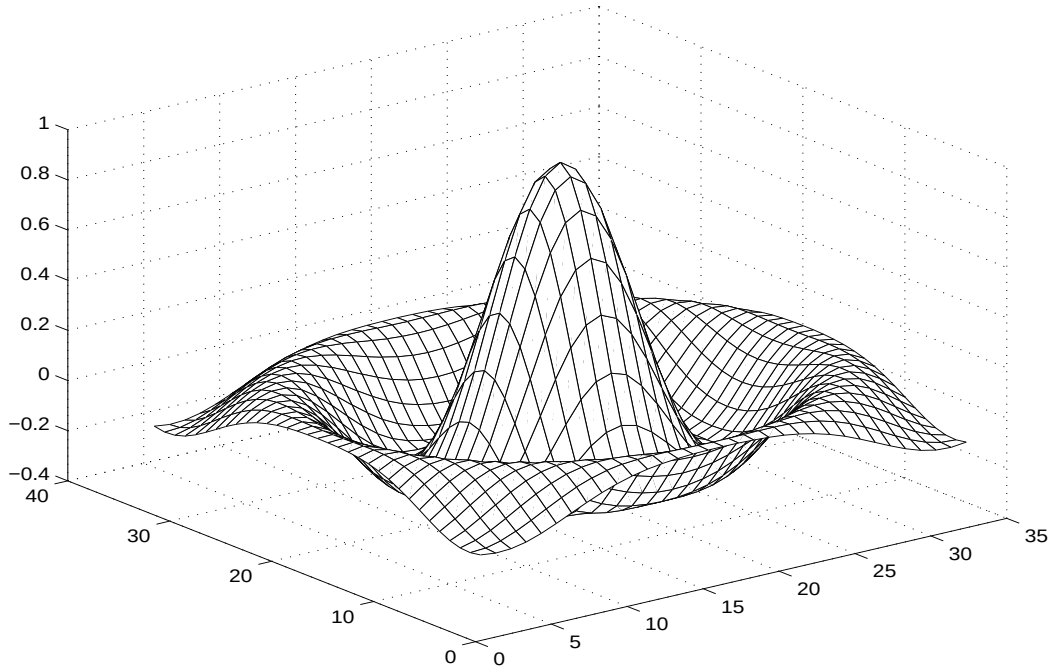

Figure 3.5: Den klassiske sombrero figuren. Vi har benyttet funksjonen **mesh**. Kjøring av filen **ex_plt_3d.m** med $iex = 1$ gir denne figuren.

Merk at sombreroeksemplet også kan plottes med funksjonen **surf**. Et alternativ, som er vel så forklarende, er

```
[x,y]=meshgrid(-8:0.5:8);
r=sqrt(x.^2+y.^2)+eps;
z=sin(r)./r;
surf(x,y,z)
```

Resultatet av dette er vist i figur 3.6.

Et annet eksempel er

```
[x,y]=meshgrid(-2:0.2:3);
z=x.*exp(-x.^2-y.^2);
surf(x,y,z)
```

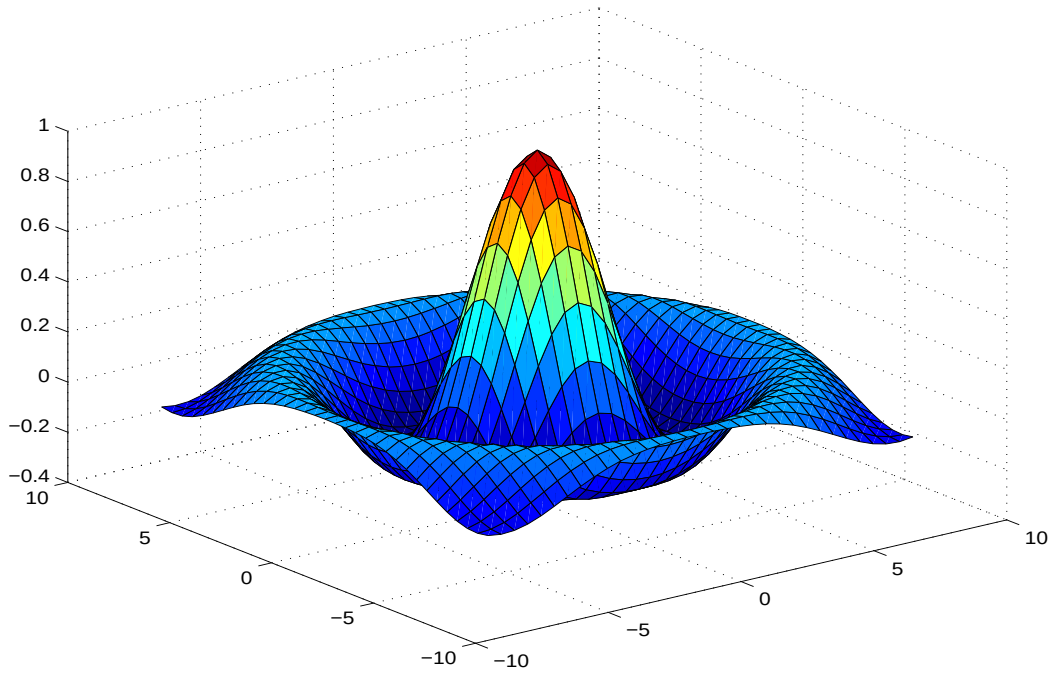

Figure 3.6: Den klassiske sombrero figuren. Vi har benyttet funksjonen **surf**. Kjøring av filen **ex_plt_3d.m** med $iex = 2$ gir denne figuren.

Vi refererer til dette som topp-dal eksemplet. Etter kjøring av dette får vi resultater som i Figur 3.7.

Når man benytter **surf** så er det viktig å passe på dimensjonene til matrisene x, y og z. Dersom z er en $m \times n$ matrise må x ha lengde n og y ha lengde m. D.v.s. at x kan være en n-dimesnjonal vektor og y en m-dimensjonal vektor. I eksemplene over er x og y matriser.

Nedenfor viser vi hvordan det kan gjøres uten bruk av **meshgrid**.

```
x=-2:0.2:3; n=length(x);
y=-2:0.2:4; m=length(y);
z=zeros(m,n);
for j=1:m
   for i=1:n
      z(j,i)=x(i)*exp(-x(i)^2-y(j)^2);
   end
end
surf(x,y,z);
```

De fire eksemplene over er implementert i scriptet **ex_plt_3d.m**. Dette gjenngis her

```
% ex_plt_3d.m
```

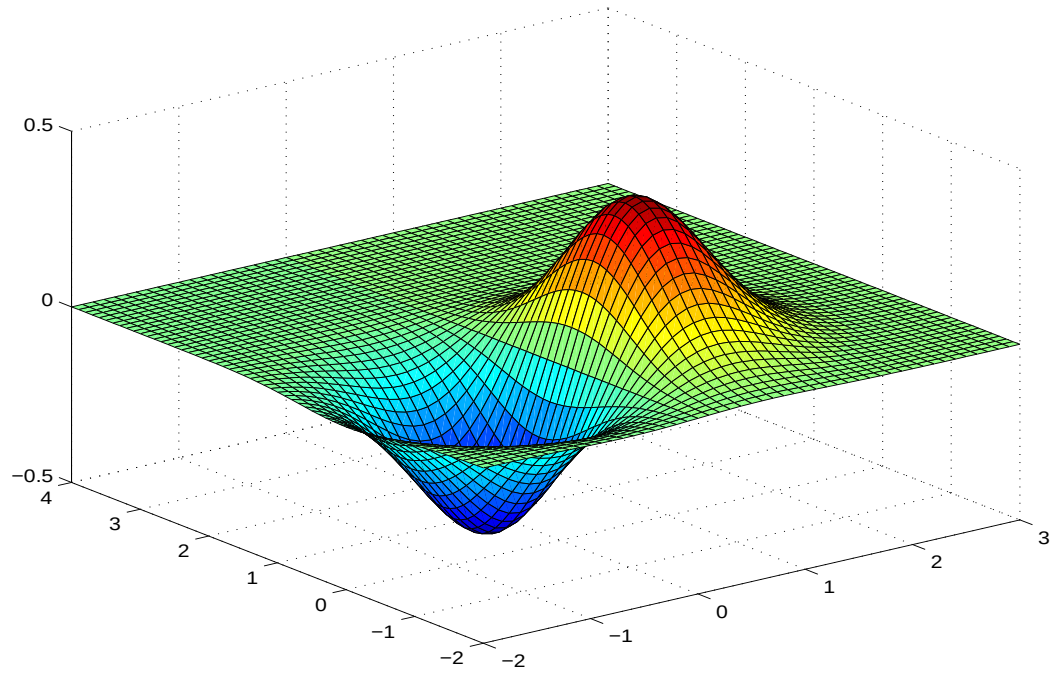

Figure 3.7: Kjøring av **ex_plt_3d.m** med $iex = 3$ og $iex = 4$ gir denne figuren.

```
% Eksempler for plotting av tre-dimensjonale figurer

ex_text=['iex=1: klassisk sombrero eksempel med MESHGRID og MESH.'
         'iex=2: klassisk sombrero eksempel med MESHGRID OG SURF.'
         'iex=3: Top-dal eksempel med MESGRID og SURF.           '
         'iex=4: Top-dal eksempel med FOR loekker og SURF        '
         'iex=5: Banana-shaped valley funksjon med FOR loekker   '];
disp(ex_text)

iex=2;
iex=dread('Velg eksempel nr. 1-2-3-4-5',iex);
if iex == 1
   [x,y]=meshgrid(-8:0.5:8);  % Klassisk sombrero eksempel, med MESH.
   r=sqrt(x.^2+y.^2)+eps;
   z=sin(r)./r;
   mesh(x,y,z)                % eller bare, mesh(z) !
elseif iex == 2
   [x,y]=meshgrid(-8:0.5:8);  % Klassisk sombrero eksempel, med SURF.
   r=sqrt(x.^2+y.^2)+eps;
   z=sin(r)./r;
   surf(x,y,z)
elseif iex==3                 % Topp-dal eksempel med SURF og MESGRID.
   [x,y]=meshgrid(-2:0.2:3);
   z=x.*exp(-x.^2-y.^2);
```

```
   surf(x,y,z)
elseif iex ==4                % Som iex=3 men kanskje mer forklarende.
   x=-2:0.2:3; n=length(x);
   y=-2:0.2:4; m=length(y);
   z=zeros(m,n);
   for j=1:m
      for i=1:n
         z(j,i)=x(i)*exp(-x(i)^2-y(j)^2);
      end
   end
   surf(x,y,z);
elseif iex ==5                % Banana-shaped valey
   x=-2:0.2:3; n=length(x);
   y=-2:0.2:4; m=length(y);
   z=zeros(m,n);
   for j=1:m
      for i=1:n
         X=[x(i);y(j)];
         J=banana([],X);
         z(j,i)=J;
      end
   end
   surf(x,y,z);


end
```

3.8 Kontur-plott

Med funksjonen **contour** kan man enkelt lage et kote-kart (kontur-plott) av en to-dimensjonal funksjon $z = f(x, y)$. Et kontur-plott av matrisene i som ga de tre-dimensjonale figurene i forrige eksempel kan enkelt plottes ved kommandoen

```
contour(x,y,z)
```

Appendix A

The D-SR Toolbox for MATLAB

A.1 The DSR function for use with MATLAB

THE ACRONYM:
Deterministic and Stochastic system identification and Realization (DSR).

PURPOSE:
Given the output data matrix Y and the input data matrix U. DSR estimate the system order n, the matrices A, B, D, E, CF, F and the initial state vector $x0$ in the discrete time, combined deterministic and stochastic dynamic model, on innovations form:

$$x_{t+1} = Ax_t + Bu_t + Ce_t, \quad x_{t=0} = x0, \tag{A.1}$$
$$y_t = Dx_t + Eu_t + e_t, \tag{A.2}$$

where t is discrete time, $C = CF \cdot F^{-1}$ is the Kalman filter gain matrix and $\mathrm{E}(e_t e_t^T) = FF^T$ is the innovations noise covariance matrix.

SYNOPSIS:

$$[A, B, D, E, CF, F, x0] = \mathrm{dsr}(Y, U, L)$$
$$[A, B, D, E, CF, F, x0] = \mathrm{dsr}(Y, U, L, g)$$
$$[A, B, D, E, CF, F, x0] = \mathrm{dsr}(Y, U, L, g, J, M, n)$$

PARAMETERS ON INPUT:

- Y – An $(N \times m)$ matrix with output data/observations. N is the number of observations and m is the number of output variables.
- U – An $(N \times r)$ matrix with input data/observations. r is the number of input variables.
- L – Integer. Number specifying the future horizon used for predicting the system order. Choose $L > 0$ such that the assumed system order satisfy $n \leq Lm$. $L = 1$ is default.

OPTIONAL INPUT PARAMETERS:
DSR has four optional integer parameters. These parameters is for advanced use. A description is given below.

- g – Integer. Optional model structure parameter. $g = 1$ is default. If $g = 0$ then a model with $E = 0_{m \times r}$ is estimated. If $g = 1$ then E is estimated.
- J – Integer. Number defining the past horizon used to define the instrumental variables used to remove future noise. $J = L$ is default and recommended.
- M – Integer. With $M = 1$ (default) a simple method for computing CF and F is used. A more computational expensive method is used when $M \neq 1$.
- n – Integer. Optional specification of model order, $0 < n \leq Lm$.