

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 November 5, 2009

Porsgrunn **November 5, 2009**

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

Preface

MATLAB is the acronym for MATrix LABoratory. It is an interactive program that 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

1	Introduction	1
2	Matrix Computations and Simulation	2
2.1	State Space Model	2
2.2	M-file programming	4
2.3	Defining vectors and matrices	6
2.4	Format	7
2.5	Diary and printing to files	8
2.6	How to define path	8
2.7	For, while and if sentences	9
2.8	How to make figures and using the plot function	11
2.9	Beregning av egenverdier og egenvektorer	13
2.10	Simulation with the lsim function	14
2.11	Simulation with explicit Euler	15
2.12	Sammenligning av resultat med lsim og Euler	16
2.13	Diskretisering av kontinuerlig modell	17
2.14	Matrisefunksjoner	18
2.15	Simulering med mer avanserte integrasjonsrutiner	19
3	Matrix Methods in Courses	21
3.1	Optimalisering	21
3.2	Numerisk linearisering og Jacobi matrisen	25
3.3	Løsning av ulineære likninger med Newton Raphsons metode	26
3.4	Datamodellering i statiske systemer	27
3.5	Simulering av diskret modell	29
3.6	Identifikasjon av diskret tilstandsrommodell fra data	30

3.7	Plotting av tredimensjonale figurer	31
3.8	Kontur-plott	34
A	The D-SR Toolbox for MATLAB	35
A.1	The DSR function for use with MATLAB	36

List of Figures

2.1	This Figure is generated by the plot_ex.m m-file script.	12
2.2	This figure is generated by the script fig_euler_lsim.m . The figure shows simulation results for the spring system by using explicit Euler and the exact lsim function.	20
3.1	Figuren viser kriteriefunksjonen ?? med tallverdier som i eksempel ?. Figuren er generert av M-filen plt_obfunc.m	23
3.2	Figuren viser den optimale trajektoren $x(t)$ i systemet $\dot{x} = Ax + Bu$ og $u = Gx$ der G minimaliserer ?? og med tallverdier som i eksempel ?. Figuren er generert av scriptet ex_opc_plt.m	24
3.3	Figuren viser kriteriefunksjonen ?? med tallverdier som i eksempel ?. Figuren er generert av M-filen plt_obfunc2.m	25
3.4	Figuren viser den optimale trajektoren $x(t)$ i systemet $\dot{x} = Ax + Bu$ og $u = Gx$ der G minimaliserer ?? og med tallverdier som i eksempel ?. Figuren er generert av scriptet main_obfunc2.m	26
3.5	Den klassiske sombrero figuren. Vi har benyttet funksjonen mesh . Kjøring av filen ex_plt_3d.m med $ie x = 1$ gir denne figuren. . . .	32
3.6	Den klassiske sombrero figuren. Vi har benyttet funksjonen surf . Kjøring av filen ex_plt_3d.m med $ie x = 2$ gir denne figuren. . . .	32
3.7	Kjøring av ex_plt_3d.m med $ie x = 3$ og $ie x = 4$ gir denne figuren. . . .	33

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 more experienced user of MATLAB. These lecture notes were first used in a NIF (Norske Sivilingeniørers Forening) course at HIT in the spring of 2000. It is also assumed that the lecture notes are suitable to give sufficient basic theory so that MATLAB can be effectively used in many courses at TF. In particular courses that contain 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. An example is the numerical linearization of a non-linear dynamic state space model.

In connection to the lecture notes there are a 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-script 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 more details later. The more experienced user of MATLAB will very often learn more by reading other m-files. In addition to all 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 more 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:

$$\dot{x} = v, \quad (2.1)$$

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

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

$$\dot{z} = v \quad (2.3)$$

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

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

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

where

$$u = F - mg. \quad (2.6)$$

Numerical values for the variables can be as follows:

$k = 2$, $\mu = 0.5$, $m = 5$, $g = 9.81$ og $x_0 = 1$. 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

$$\dot{x} = Ax + Bu, \quad (2.7)$$

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

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

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

```
>> m = 5; k = 2; g = 9.81; F = 0, x0 = 1
>> z = (F - m * g)/k
>> x = z + x0
```

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

$$\dot{x}_1 = x_2, \quad (2.9)$$

$$\dot{x}_2 = -\frac{g}{r} \sin(x_1) - \frac{b}{mr^2} x_2, \quad (2.10)$$

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 $\sin(x_1) \approx x_1$.

$$\overbrace{\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix}}^{\dot{x}} = \overbrace{\begin{bmatrix} 0 & 1 \\ -\frac{g}{r} & -\frac{b}{mr^2} \end{bmatrix}}^A \overbrace{\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}}^x \quad (2.11)$$

Putting the numerical values into the model we get

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

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

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

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

» type data

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

» who**» 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

» 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

» 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 looking 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

» 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 possible to making so called precompiled 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 built in commands within MATLAB.

2.3 Defining vectors and matrices

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

```
>> k = 2; m = 5; mu = 0.5;
>> A = [0, 1; -k/m, -mu/m]
>> 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

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

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

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

```
>> 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

```
>> 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

» $I = \mathbf{eye}(n)$

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

» $I = \mathbf{eye}(n, m)$

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

» $E = \mathbf{randn}(N, m)$

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.

» $A = \mathbf{hilb}(4)$

» $A2 = \mathbf{pascal}(4)$

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

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

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

» $A = \mathbf{pascal}(6)$

» $p = \mathbf{diag}(\mathbf{rot90}(A))$

» $s = \mathbf{roots}(p)$

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.,

» $Ac = \mathbf{compan}(p)$

» $l = \mathbf{eig}(Ac)$

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

» **format short**
» **format long**
» **format bank**

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

With the

» **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 a file. This can be done with the MATLAB command, **diary**.

» **diary filename**

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

» **pwd**
» **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

» **diary off**

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

» **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 **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 element is t_f and that the distance between each time instant is, h . This can simply be defined within MATLAB as follows

```
>> t = t0 : h : tf;
```

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.

```
>> t = 0 : 0.1 : 150;
>> plot(t,t)
```

This gives a straight 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:

```
>> N = length(t)
>> 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. see the m-file **ex_for.m**, i.e.

```
% ex_for
% Eksempel paa for loekke for aa generere en
% tidsserievektor, t=[0 0.1 0.2 ... 150]
t0=0; tf=150; h=0.1;      % Def. skrittlengde,h, og foerste og siste element i t.
N=(tf-t0)/h+1;           % antall elementer i vektoren, t.
t=zeros(N,1);           % Setter av plass til en N x 1 matrise, dvs. vektoren t.

w=t0;                   % "work" variabel for teller.
for i=1:N                % i tillordnes verdiene, 1,2,3,...,N
    t(i)=w;              % Lagrer w i vektoren t.
    w=w+h;              % oppdaterer tellervariabel.
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).

Look 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 useful 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.

```
>> t = 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
% Funksjon for aa beregne maskinpresisjon, eps, slik at eps
% er det minste tallet slik at maskinen forstaar at, eps+1>1.

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

```
>> help colon
```

Note that if you are typing

```
>> help :
```

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

The colon operator 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

```
>> t(1 : 10)
```

ore from element number 1491 to the last element

```
>> 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
% Eksempel paa bruk av plot og relaterte kommandoer.

t=-pi:pi/100:pi;
y=sin(t);
plot(t,y)
```



```
axis([-pi pi -1 1])
xlabel('-\pi \leq t \leq \pi')
ylabel('y=sin(t)')
title('Plot av sinusfunksjon, y=sin(t)')
grid
text(1,1/3,'\it{Merk symmetrien}')
```

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

» **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 obtained by the **xlabel** command. You can use LATEX or 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

» **print -deps filename**

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

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

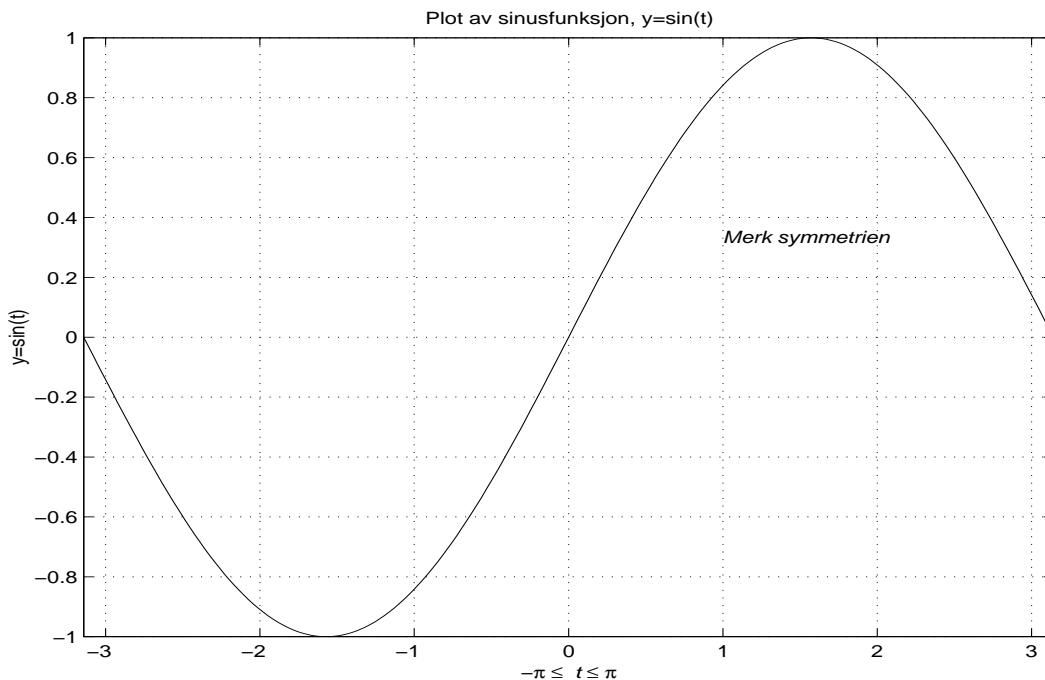


Figure 2.1: This Figure is generated by the **plot_ex.m** m-file script.

2.9 Beregning av egenverdier og egenvektorer

MATLAB er skreddersydd til å utføre de fleste matrise-dekomposisjoner og matrise-faktoriseringer i lineær algebra. Vi vil i dette avsnittet illustrere hvordan vi beregner en egenverdimatrise L og en egenvektormatrise M slik at $A = MLM^{-1}$, ved hjelp av funksjonen **eig**. Vi husker at egenverdiene kan finnes som røttene til den karakteristiske ligning $\det(\lambda_i I - A) = 0$ og egenvektorene som løsning av $Am_i = \lambda m_i$ der $i = 1, \dots, n$. Vi har da at $M = [m_1 \ \dots \ m_n]$

$$\text{og } L = \begin{bmatrix} \lambda_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \lambda_n \end{bmatrix}$$

Vi illustrerer bruken av funksjonene **eig**, **poly** og **roots** i **diary** filen vist under. Merk videre at funksjonen **inv** kan benyttes til å invertere en ikke-singulær matrise.

```
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

```

Vi har her beregnet egenverdiene og egenvektorene til A matrisen ved hjelp av funksjonen **eig**. Vi har funnet koeffisientene til det karakteristiske polynomet til A , d.v.s. $\det(\lambda I - A)$, ved hjelp av funksjonen **poly** og beregnet røttene til den karakteristiske ligningen, d.v.s., $\det(\lambda I - A) = 0$, ved hjelp av funksjonen **roots**. Videre så har vi utført matriseinvertering v.h.a. funksjonen **inv**.

Andre matrise dekomposisjoner og faktoriseringer i lineær algebra er implementert i funksjonene **qr**, **lu**, **chol**, **schur** og **svd**.

2.10 Simulation with the lsim function

The spring system can easily be simulated by using the **lsim.m** function. **lsim** is a part of the Control Toolbox for MATLAB. This function results in an exact solution to a linear dynamic system of the form

$$\dot{x} = Ax + Bu, \quad (2.14)$$

$$y = Dx + Eu, \quad (2.15)$$

where the initial state vector, $x(t = 0) = x_0$, must be known. An m-file script for simulation of the spring system 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

```



```

x(2,1)=v_0;

for i=1:N          % The for-loop, loop over all integers, i=1, i=2, etc.
    y(i,:)=x';    % Store 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 Ordinary Differential Equation (ODE) is implemented in the function `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 global.
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 Sammenligning av resultat med `lsim` og Euler

If we plot the results from `ex1_lsim` and `ex1_euler` in the same window we see that the results from the Euler integration is relatively poor compared to the `lsim` results, especially for a large step length $h = 0.1$. See Figure 2.12. If the step size in the Euler integration is reduced to $h = 0.01$ the results are almost identical. However, the computing time is significantly increased for the Euler integration. This can be checked by the commands

```

>> tic; ex1_lsim; toc
>> tic; ex1_euler; toc

```

The time used is 0.34 for `ex1_lsim` and 4.24 for `ex1_euler` with $h = 0.01$. The time for `lsim` is here $4.24/0.34 = 12.5$ times faster than user `euler`. Note that `lsim` is an exact simulation of a linear dynamic system.

2.13 Diskretisering av kontinuerlig modell

Det som foregår når vi benytter eksplisitt Euler, **lsim** eller andre metoder for simulering av dynamiske systemer er at man setter opp en diskret modell for den kontinuerlige modellen. En diskret tilstandsrommodell for en lineær tilstandsrommodell

$$\dot{x} = A_c x + B_c u, \quad (2.16)$$

$$y = D_c x + E_c u, \quad (2.17)$$

med initialtilstand $x(t=0) = x_0$ kan skrives

$$x_{k+1} = A x_k + B u_k, \quad (2.18)$$

$$y_k = D x_k + E u_k, \quad (2.19)$$

med initialverdi $x_{k=0} = x_0$ gitt. Dersom vi benytter eksplisitt Eulers metode så har vi at

$$A = I + h A_c, \quad (2.20)$$

$$B = h B_c, \quad (2.21)$$

$$D = D_c, \quad (2.22)$$

$$E = E_c, \quad (2.23)$$

der $h > 0$ er skrittlengden. **lsim** benytter en eksakt løsning av den kontinuerlige modellen, d.v.s.,

$$A = e^{A_c h}, \quad (2.24)$$

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

dersom A_c er inverterbar.

Husk at eksplisitt Eulers metode er unøyaktig for store skrittlengder. Den største skrittlengden det er mening i å benytte kan bestemmes ved å kreve at egenverdiene til $A = I + h A_c$ skal ha tallverdi mindre enn en, d.v.s.,

$$|\lambda(I + h A_c)| < 1. \quad (2.26)$$

Den største skrittlengden vi kan benytte med eksplisitt Eulers metode er da

$$0 < h_{\max} < \frac{2}{\max |\lambda(A_c)|} \quad (2.27)$$

dersom egenverdiene til A er reelle. Dersom egenverdiene er komplekse, f. eks., $\lambda = \alpha \pm j\beta$, så finner vi den største skrittlengden fra formelen

$$0 < h_{\max} < \frac{2 |\alpha|}{\alpha^2 + \beta^2} \quad (2.28)$$

Dette kan vi utlede fra matrisen

$$A = \begin{bmatrix} \alpha & -\beta \\ \beta & \alpha \end{bmatrix} \quad (2.29)$$

som har egenverdier gitt ved $\lambda = \alpha \pm j\beta$. I MATLAB kan dette beregnes ved

```
>> l = eig(A)
>> hmax = 2 * abs(real(l(1)))/(real(l(1))^2 + imag(l(1))^2)
```

Dette kan med fordel implementeres som i funksjonen **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 dempefjærsystemet finner vi at den maksimale skrittlengden man kan benytte med Eulers metode er $h \leq h_{\max} = 0.25$. For pendelmodellen har vi $h \leq h_{\max} = 0.0255$. I praksis må vi, p.g.a. krav til nøyaktighet, benytte en skrittlengde som er noe mindre.

Dette viser at egenverdiberegninger er sentralt i forbindelse med simulering. Eksplisitt Eulers metode er mye benyttet i praksis. Grunnen til dette er at den er enkel og at den gir et svar som er nøyaktig nok sammenlignet med den virkelige prosess. Eksplisitt Eulers metode er særlig aktuell dersom den kontinuerlige tilstandsrommodellen er ulineær.

Dersom den dynamiske modellen er lineær er det bedre å benytte en eksakt diskretisering. I MATLAB finner vi en eksakt diskrete modell ved kommandoen

```
>> [A, B] = c2d(Ac, Bc, h)
```

De resulterende diskrete tilstandsrommodellmatrisene A, B samt $D = D_c$ og $E = E_c$ gir da en komplett diskret dynamisk modell som gitt ved ligningene (2.18) og (2.19). Relaterte funksjoner er her **c2dm**, **d2c** og **d2cm**.

2.14 Matrisefunksjoner

Det finnes en rekke innebygde matrisefunksjoner i MATLAB. De fleste av disse ligger i toolbox'en MATFUN. En oversikt får man ved å skrive

```
>> help matfun
```

Vi vil her gi et eksempel på bruk av funksjonen **expm** for beregning av matriseeksponenten $A = e^{A_c h}$ som ble presentert i avsnittet over.

» $A = \mathbf{expm}(A_c * h)$

Den tilhørende B matrisen i den diskrete modellen kan nå enkelt beregnes ved

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

2.15 Simulering med mer avanserte integrasjonsrutiner

I toolbox'en FUNFUN ligger det en rekke integrasjonsrutiner som er meget gode for ulineære og mer avanserte simuleringsoppgaver. En ordinær differensialligning (ODE) kan skrives

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

med kjent initialverdi-vektor $x(t = t_0) = x_0$. Denne ODE ligningen kan diskretiseres med en såkalt θ -metode, d.v.s.,

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

Vi ser at dersom vi velger $\theta = 1$ så får vi eksplisitt Eulers metode, $\theta = 0$ gir implisitt Eulers metode og $\theta = \frac{1}{2}$ gir trapesmetoden. Vi ser at når $\theta \neq 1$ så må vi, når $f(x)$ er ulineær, løse en ulineær ligning med hensyn på x_{k+1} for hvert tidskritt. Dette kan gjøres v.h.a. Newton-Raphsons metode.

Trapesmetoden, d.v.s. θ -metoden med $\theta = \frac{1}{2}$, er implementert i funksjonen **ode23t**. Dersom vi velger $\theta = 0.52$ får vi en metode som er egnet for stive systemer. En lignende metode er implementert i funksjonen **ode15s**

Bruken av disse integrasjonsrutinene krever at høyresiden $f(x)$ implementeres i en funksjon med samme syntaks som **dampfjaer.m**. Vi kan nå som et eksempel simulere dempefjærsystemet med funksjonen **ode45**, d.v.s.,

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

Vi kan også benytte en stiv løser, f. eks.

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

Den ulineære pendelmodellen kan simuleres med f. eks.

» $[t, x] = \mathbf{ode15s}('fx_pendel', [0 \ 150], [pi/2; 0]);$

Et stift ODE system er et system der forholdet mellom den største og minste egenverdien til den lineariserte systemmatrisen er et stort tall. Dette er det samme som at forholdet mellom den største og den minste tidskonstanten i systemet er et stort tall. Med stort menes her i området 100-10000. Sammenhengen mellom en egenverdi λ og en tidskonstant T er gitt ved $T = \frac{1}{\lambda}$.

Legg merke til at flere av ODE løserene arbeider med variabel skritt lengde. Det betyr at skritt lengden og antall beregnede løsninger i tidsintervallet $0 \leq t \leq 150$


```
figure=m:/tex/fag/matlab_kurs/matlab_files/fig/ex1sim_euler.eps,height =  
9.0cm,width = 14cm
```

Figure 2.2: This figure is generated by the script **fig_euler_lsim.m**. The figure shows simulation results for the spring system by using explicit Euler and the exact **lsim** function.

i eksemplene over ikke er konstante. Man kan imidlertid tvinge løserne til å beregne løsningen ved uniforme eller spesifiserte diskrete tidskritt. Ønsker vi løsninger ved uniforme tidskritt $t = 0, t = h, t = 2h$ osv. så kan eksemplet over modifieres slik

```
>> [t, x] = ode15s('fx_pendel', 0 : h : 150, [pi/2; 0]);
```

der h er en spesifisert skrittlengde, for eksempel $h = 0.1$.

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 toolboxen. 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 optimaliseringsproblemer studeres b.l.a. i faget Avansert reguleringsteknikk ved TF. Følgende funksjon

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

der

$$\dot{x} = Ax + Bu, \quad (3.2)$$

$$u = Gx, \quad (3.3)$$

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.,

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 plote 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 skritt lengden 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.

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**.

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*

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

En kjøring gir

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

```
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 modifieres 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 $G = \begin{bmatrix} g_1 & g_2 \end{bmatrix} = \begin{bmatrix} -1.9025 & 1.0000 \end{bmatrix}$. Denne kan finnes ved **lqr** funksjonen i Control Systems Toolbox'en. Vi gjenngir funksjonene **main_obfunc2.m** og **obfunc2.m** denne nedenfor,

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 $x \in \mathbb{R}^n$. Vi har da at Jacobimatrisen er definert som

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

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

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**.

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

```
>> A = jacobi('fx_pendel', [], [0; 0])
```

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

```
>> A = jacobi('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

$$f(x^*) = 0. \quad (3.6)$$

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

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

Vi løser m.h.t. x^* og får

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

Dette kan benyttes for å lage følgende iterasjons-ligning

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

Iterasjonsprosessen (3.9) refereres til som Newton-Raphsons metode. Iterasjonsprosessen starter ved at vi spesifiserer en startverdi, x_0 , og itererer (3.9) inntil $f(x_i) \leq \text{tol}$ der tol er en spesifisert toleranse for hva som er null. En mye benyttet numerisk toleranse er $\text{tol} = \sqrt{\text{meps}}$ der meps er maskinpresisjonen. Maskinpresisjonen er som vi har funnet tidligere gitt ved $\text{meps} = 1/2^{52} \approx 2.22 \cdot 10^{-16}$. Dette gir $\text{tol} = 1.5 \cdot 10^{-8}$. Det som gjør Newton-Raphsons metode så effektiv er den raske konvergenstakten nær løsningen (2. ordens konvergens). Anta at dersom x er nær løsningen, x^* , d.v.s., slik at $x^* = x + \varepsilon$. Dette gir

$$f(x + \varepsilon) = f(x) + \frac{\partial f}{\partial x} \Big|_x \varepsilon \approx 0, \quad (3.10)$$

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 gjengitt her

Vi skal nå illustrere bruken ved å finne det stasjonære konsentrasjons-profilen, x , i en destillasjonskolonne med ett trinn, koker og akkumulator. Først må vi definere funksjonen $f(x)$ i en fil med syntaks $f = \text{fcol3_a}(0, x)$. 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

$$Y = XB + E \quad (3.11)$$

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

$$\begin{array}{c} \overbrace{\left[\begin{array}{c} 0.35 \\ 0.34 \\ 4.32 \\ 0.97 \end{array} \right]}^Y = \overbrace{\left[\begin{array}{ccc} 0.16 & 0.10 & 0.08 \\ 0.17 & 0.11 & 0.09 \\ 2.02 & 1.29 & 1.0 \\ 0.5 & 0.4 & 0.1 \end{array} \right]}^X \overbrace{\left[\begin{array}{c} 1.00 \\ 1.00 \\ 1.00 \end{array} \right]}^B + \overbrace{\left[\begin{array}{c} 0.01 \\ -0.03 \\ 0.02 \\ -0.03 \end{array} \right]}^E. \end{array} \quad (3.12)$$

Problemet vi skal studere er å beregne løsninger for B med utgangspunkt i kjente datamatriser X og Y . Datamatrissene X og Y er lagret på filen `xy_data.mat`. Matrisene kan 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 $B = [1.00 \ 1.00 \ 1.00]^T$. 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

```
>> [a,b] = c2d(A,B,0.5)
>> d = eye(2)
>> e = zeros(2,1)
```

Vi har her benyttet en skritt lengde (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

```
>> figure(1)
>> dstep(a, b, d, e)
>> figure(2)
>> 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

```
>> [Y, X] = dlsim(a, b, d, e, U, x0)
```

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.

```
» [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 ”sombbrero” 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 $x - y$ aksene i horisontalplanet. z aksene skaleres automatisk av **mesh**. Merk at maks og min verdiene av z kan finnes ved

```
ans =  
  
1  
  
min(z(:))  
  
ans =  
  
-0.2172
```

Sombbreroen er vist i Figur 3.5.

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

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

Figure 3.7: Kjøring av `ex_plt_3d.m` med `icx = 3` og `icy = 4` 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

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 x_0 in the discrete time, combined deterministic and stochastic dynamic model, on innovations form:

$$x_{t+1} = Ax_t + Bu_t + Ce_t, \quad x_{t=0} = x_0, \quad (\text{A.1})$$

$$y_t = Dx_t + Eu_t + e_t, \quad (\text{A.2})$$

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

SYNOPSIS:

$$\begin{aligned} [A, B, D, E, CF, F, x_0] &= \text{dsr}(Y, U, L) \\ [A, B, D, E, CF, F, x_0] &= \text{dsr}(Y, U, L, g) \\ [A, B, D, E, CF, F, x_0] &= \text{dsr}(Y, U, L, g, J, M, n) \end{aligned}$$

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, $0 < n \leq Lm$.