MATLAB for MAT1120

Erik Bédos, Geir Dahl, Øyvind Ryan

Høst 2013

Innledning

Dette kompendiet inneholder det meste som MAT1120-studenter trenger å lære om MATLAB. Det bygger videre på MATLAB kunnskapene som studentene skal ha tilegnet seg i løpet av MAT1110. Studenter som ikke har tatt MAT1110, eller ikke har tilsvarende MATLAB kunnskaper, bør bruke tid på å lese og jobbe seg gjennom heftet "MATLAB for MAT1110"(som er tilgjengelig på hjemmesiden til MAT1120). En oversikt over andre tilgjengelige MATLAB-ressurser finnes også på hjemmesiden til MAT1120.

I kompendiet finnes koden til de aller fleste MATLAB illustrasjonene som blir vist (eller henvist til) på forelesningene, samt en god del programmeringstips. Vi har lagt vekt på å ta med noen eksempler som illustrerer noen av de viktige teoremene i boka (når det lar seg gjøre) og noen eksempler som krever programmering av en viss lengde. Vi har valgt en noe uformell skrivestil, uten lange omstendelige forklaringer. Det er meningen at studentene skal gå nøye gjennom alle kodene som er i dette kompendiet. Kopier gjerne alle programbitene og kjør disse selv! Hvis det er noe du ikke helt forstår, prøv å forandre på noen av input-verdiene og studer hvilke utslag det gir.

Her uke blir det som regel gitt noen oppgaver som krever bruk av MATLAB. Hvis du er fortrolig med det som står i dette heftet håper vi at du ikke vil ha problemer med å skjønne hvordan disse oppgavene skal løses. Hvis ikke kan du alltid gå på en av de ukentlige datalabene og få hjelp der.

Det viser seg at noen studenter foretrekker å bruke PYTHON (eller andre programpakker), og det kan de godt gjøre, f.eks. under arbeidet med de obligatoriske oppgavene. Øyvind Ryan har skrevet et kompendium (PYTHON for MAT1120") for disse studentene, som kan lastes ned fra hjemmesiden. På forelesningene og i løsningsforslagene vil vi kun forholde oss til MATLAB.

I eksamensoppgavene vil det kunne være punkter som krever kjennskap til det som er omtalt i dette MATLAB-kompendiet. Dette betyr at alle studentene forventes å ha satt seg inn i dette heftet, selv om de velger å bruker et annet program ellers i kurset.

EB / GD / ØR

Innhold

En liten oppvarming **4**

Kapittel 4 **6**
Seksjon 4.2 og 4.3 6
Seksjon 4.4 8
Seksjon 4.6 8
Seksjon 4.8 9
Seksjon 4.9 10

Kapittel 5 **13**
Seksjon 5.1, 5.2 og 5.3 13
Seksjon 5.4 (Notat 2) 15
Seksjon 5.5 og 5.6 16
Seksjon 5.7 21
Seksjon 5.8 25

Kapittel 6 **29**
Seksjon 6.1 29
Seksjon 6.2 29
Seksjon 6.3 31
Seksjon 6.4 32
Seksjon 6.5 35
Seksjon 6.6 36
Seksjon 6.8 39

Kapittel 7 **41**
Seksjon 7.2 41
Seksjon 7.4 44

	Noen nyttige Matlab-kommandoer
Navn	**Forklaring**
`ceil`	Runder opp til nærmeste hele tall
`det`	Regner ut determinanten til en matrise
`diag`	Returnerer en diagonalmatrise med elementene fra input-vektoren på diagonalen
`eig`	Regner ut egenverdiene/egenvektorene til en matrise
`floor`	Runder ned til nærmeste hele tall
`fprintf`	Skriver ut tekst på displayet
`imag`	Returnerer imaginærdelen til et komplekst tall.
`inv`	Returnerer den inverse matrisen
`linspace`	Lager en vektor med et gitt antall punkter i stigende rekkefølge, med startpunkt og sluttpunkt også gitt
`norm`	Regner ut lengden av en vektor/Frobenius-normen til en matrise
`null`	Regner ut en basis for nullrommet til input-matrisen
`orth`	Returnerer en matrise med ortonormale kolonner som er en basis for kol.rommet til input-matrisen
`pause`	Stopper eksekvering av koden for en gitt tid
`plot`	Plotter en funksjon (eller punkter/vektorer)
`poly`	Returnerer $\pm$ det karakteristiske polynomet til input-matrisen (som en vektor)
`polyval`	Beregner verdier av et polynom med gitte koeffisienter
`quiver`	Lager et plott av et vektorfelt
`rand`	Genererer tilfeldige tall mellom 0 og1
`real`	Returnerer realdelen til et komplekst tall
`roots`	Regner ut røttene til et polynom (angitt som en vektor)
`rref`	Bringer en matrise på redusert trappeform
`size`	Regner ut dimensjonene til en matrise
`sort`	Sorterer verdiene i en vektor i stigende rekkefølge
`svd`	Regner ut svd-dekomposisjonen til en matrise.
`tic`	Gir beskjed til Matlab om å starte tidtaking
`toc`	Gir beskjed til Matlab om å avslutte tidtaking

Tabell 1: For en mer utfyllende beskrivelse av disse kommandoene, og en spesifikasjon av parametrene som disse kan fores med, kan du bruke `help` i Matlab.

En liten oppvarming

Vi vil ofte ha bruk for å plotter punkter og vektorer i et plant koordinatsystem. I MATLAB blir ikke koordinataksene tegnet opp automatisk for oss. I koden nedenfor vil aksene bli tegnet inn i figuren som koden produserer, som fire (blåe) linjer fra origo, i det første kallet på funksjonen `plot`. Verdiene av L og U kan forandres etter behov i andre situasjoner.

```
L=2; U=10;

% Tegner akser; b staar for blaatt:
plot([-L U],[0 0],'b',[0 0],[-L U],'b');


axis([-L U -L U]);
axis manual;
grid;                   % soerger for at et rutenett blir tegnet opp

hold on;

% Plotter naa syv tilfeldige punkter:

for k=1:7
  x=(U-1)*rand(1,2);    % genererer en 2-dim  tilfeldig radvektor
                        % med koeff. mellom 0 og 9

  plot(x(1),x(2),'or');            % r staar for roedt
                                   % o staar for moensteret punktene
                                   % blir tegnet med
end

% Nedenfor plottes en tilfeldig vektor og vektoren [5 2]:

plot([0 U*rand],[0 U*rand],'-og'); % g staar for groent.
                                   % o staar for monsteret
                                   % endepunktet blir tegnet med
plot([0 5],[0 2],'-ok');           % k staar for svart.
                                   % o staar for moensteret
                                   % endepunktet blir tegnet med
hold off;
```

Det er viktig å huske på at det å gange en matrise med en vektor svarer til det å lage en lineær kombinasjon av kolonnene i matrisen, med vekter hentet fra selve vektoren. En tilfeldig lineær kombinasjon av kolonnene i en matrise A kan f.eks. lages slik:

```
A=[1 2 3; 4 1 2; 1 4 2]
x=floor(3*rand(3,1))
linkomb=A*x
```

Kommandoen `rref`, som brukes til å beregne den reduserte trappeformen til en matrise, kan brukes til å løse likningssystemer. Det kan også brukes til å sjekke om en vektor er en lineær kombinasjon av kolonnene i en matrise. Dette kan illustreres slik:

```
A=[1 2; 4 1; 1 4]

b=[4;9;6]      % Merk at b = 2 a1 + a2,
               % der a1 og a2 er 1. og 2. kol. i A
               % Det betyr syst Ax = b er konsistent,
               % med losn. x= [2 ; 1]

R=rref([A b]) % Ser da at syst. er konsistent.
               % Legg merke til siste kolonne i R.

b=floor(4*rand(3,1))

             % b er naa en tilfeldig vektor (med naturlige koeff);
             % det er da hoyst sannsynlig at Ax=b ikke er konsistent.


R=rref([A b])         % Se paa siste raden i R.
```

NB: I stedet for å bruke `rref` for å løse et likningssystem av typen $A\mathbf{x} = \mathbf{b}$ som vi på forhånd vet har en *entydig løsning*, kan vi skrive `x=A\b` . (Hvis systemet er inkonsistent, vil MATLAB ofte tilordne en verdi til `x` da også, så denne varianten må brukes med forsiktighet !).

Kapittel 4

Seksjon 4.2 og 4.3

Nullrommet til en matrise A kan finnes ved å skrive

```
null(A)
```

Hvis nullrommet ikke bare består av $\mathbf{0}$-vektoren, vil kolonnene i den returnerte matrisen danne en basis for nullrommet.

Koden nedenfor genererer en tilfeldig 4×4 matrise der alle koeffisientene er 0 eller 1. Hvis nullrommet bare består av $\mathbf{0}$-vektoren (temmelig usannsynlig!) skriver den ut en melding. I motsatt fall tester den at første kolonnevektor i den returnerte matrisen faktisk ligger i nullrommet.

```
n=4; A=floor(1.5*rand(n)) % tilfeldig 4x4-matr. der koeff. er 0 eller 1

V=null(A)           % basis for nullrommet

[r s]=size(V);      % r=ant. rader, s=ant. kolonner i V

if s==0

  fprintf('Nullrommet bestaar bare av 0-vektoren');

else

  v=V(:,1);

  A*v               % sjekker at dette gir 0-vektoren

end
```

En basis for kolonnerommet Col$A$ til matrisen A kan finnes ved hjelp av kommandoen `orth(A)`. Kolonnene i den returnerte matrisen er da en basis for kolonnerommet.

NB: Basisene som produseres ved hjelp av `null` og `orth` er *ortonormale* (dette begrepet diskuteres først i kap. 6).

For å finne en basis for Col(A) som består av kolonner fra A, er det enklest å beregne $R = \texttt{rref}(A)$ og plukke de kolonne i A som svarer til pivotkolonnene i R (jf. Teorem 6 i Kap. 4).

Hvis du i stedet vil ha en basis for radrommet til Row(A) kan du f. eks. bruke at Row(A) er essentielt det samme som Col(A^T); eller så kan du lese det som står under seksjon 4.6 i dette heftet.

Ofte er vi interessert i å finne ut om noen gitte funksjoner $f_1, \cdots, f_n$ er lineært uavhengige, dvs vi må finne ut om det er slik at

$$c_1 f_1(t) + \cdots + c_n f_n(t) = 0$$

for alle t (i def. mengden til f_j-ene), så er eneste mulighet at $c_1 = \cdots = c_n = 0$.

En mye brukt teknikk til dette er å velge ut et sett med punkter (like mange som antall funksjoner), og så skrive ned likningene vi får når vi setter inn for hvert punkt. Finner vi ingen andre løsninger enn null-løsningen vil funksjonene være lineært uavhengige. Som et eksempel, la oss finne ut om funksjonene

$$1, \sin t, \sin^2 t, \sin^3 t, \sin^4 t$$

er lineært uavhengige. Vi velger da ut punkter t_1, t_2, t_3, t_4, t_5 og setter opp likningene vi da får:

$$\begin{array}{rcl} c_1 + c_2 \sin t_1 + c_3 \sin^2 t_1 + c_4 \sin^3 t_1 + c_5 \sin^4 t_1 & = & 0 \\ \vdots & = & \vdots \\ c_1 + c_2 \sin t_5 + c_3 \sin^2 t_5 + c_4 \sin^3 t_5 + c_5 \sin^4 t_5 & = & 0 \end{array}$$

Koeffisientmatrisen til dette likningsystemet er 5×5 matrisen

$$\begin{bmatrix} 1 & \sin t_1 & \sin^2 t_1 & \sin^3 t_1 & \sin^4 t_1 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \sin t_5 & \sin^2 t_5 & \sin^3 t_5 & \sin^4 t_5 \end{bmatrix}.$$

Det er klart at hvis den reduserte trappeformen til denne matrisen er identitetsmatrisen så har likningssystemet ovenfor bare den trivielle løsningen, og da vil funksjonene være lineært uavhengige.

Velger vi f.eks. t_1, t_2, t_3, t_4, t_5 til å være $0, 0.1, 0.2, 0.3, 0.4$ kan vi utføre beregningene ved å skrive

```
t = (0:0.1:0.4)';

A = [ (sin(t)).^0 (sin(t)).^1 (sin(t)).^2 (sin(t)).^3 (sin(t)).^4 ];

rref(A)
```

Pass på her at du forstår koden som produserer koeffisientmatrisen A. Her anvendes først funksjonen `sin` på en vektor, deretter opphøyes den resulterende vektoren komponentvis i et sett potenser. Kjører du koden ovenfor, vil du se at den produserer identitetsmatrisen, m.a.o. at det tilhørende homogene systemet har bare den trivielle løsningen, og det følger da at funksjonene er lineært uavhengige.

I koden ovenfor valgte vi punktene $0, 0.1, 0.2, 0.3, 0.4$, og det lyktes oss med dette valget å vise at funksjonene er lineært uavhengige. Det er imidlertid ikke

alltid opplagt hvordan disse punktene skal velges for at denne strategien skal lykkes, og det er ofte slik at flere valg av punkter ikke vil hjelpe (spesielt hvis funksjonene skulle vise seg å være lineært avhengige!). For eksempel vil valget $0, 2\pi, 4\pi, 6\pi, 8\pi$ i koden ovenfor ikke kunne hjelpe oss til å konkludere med lineær uavhengighet. Man må gjerne prøve seg litt frem med forskjellige sett av punkter.

Seksjon 4.4

Koden nedenfor viser hvordan vi setter opp en basisskiftematrise i $\mathbb{R}^2$, og hvordan vi gjør et koordinatskifte.

```
b1=[1;2];  b2=[1; -1];  % basisvektorer for en basis B i R^2

P=[b1 b2]  % gir basisskiftematrisen fra B til stand.basisen
fprintf('det(P)=%f',det(P)) % sjekker om P er invertibel

x=[3; 3]
c=P\x      % gir koordinatvektoren til x mhp B

P*c        % skal gi tilbake opprinnelige koordinater, dvs x
```

Seksjon 4.6

Kall på funksjonen `rank` gir rangen til en matrise. Vi vet at dette er dimensjonen til kolonnerommet til matrisen, og at det også er lik dimensjonen til radrommet til matrisen.

Rangteoremet kan verifiseres slik:

```
% rang til matriser - rang teoremet

m=3; n=5;

A=floor(1.5*rand(m,n)) % en "tilfeldig" matrise

N=null(A)              % kolonnene i N er basis for nullrommet til A

[p q]=size(N);         %  q er da lik dim. til Nul A

r=rank(A);

fprintf('\nrank A = %d , dim Nul A = %d , Ant. kolonner i A n= %d\n'...

        , r, q, n)      % vi kan da se at r + q = n
```

Legg merke til her de tre punktumene på slutten av nest siste linje. Dette betyr at uttrykket går over flere linjer. Dette er lurt å bruke når du har lange uttrykk, som jo fort kan bli mer oversiktlige når man splitter de over flere linjer.

Vi tar med et eksempel til. Koden nedenfor genererer en tilfeldig matrise, og skriver ut en basis for nullrommet og kolonnerommet. Til slutt verifiseres rangteoremet.

```
n=5;

A=floor(2*rand(n))

V=null(A)          % nullrommet

W=orth(A)          % basis for kolonnerommet

[m1,n1] = size(V); % n1 er dimensjonen til nullrommet

[m2,n2] = size(W); % n2 er dimensjonen til kolonnerommet

[n1+n2 n]          % Verifiser rangteoremet: n1 + n2 = n
```

Vi har allerede sett hvordan man kan finne en basis for radrommet til en matrise A. Men det enkleste er egentlig å beregne $R = \texttt{rref}(A)$. Radene i R vil da være en basis for radrommet siden `rref` bare anvender elementære radoperasjoner, og disse forandrer ikke radrommet. Antall rader i R som ikke bare består av 0-ere, m.a.o. antall pivot-elementer, gir rangen til A.

Rangen til en matrise kan også finnes på andre (og bedre) måter. Vi kommer tilbake til dette i kapittel 7. Et problem er at, selv om to matriser ligger veldig nær hverandre, så kan det godt hende at de har forskjellig rang. Dette kan ofte gi numeriske problemer når man skal beregne rangen.

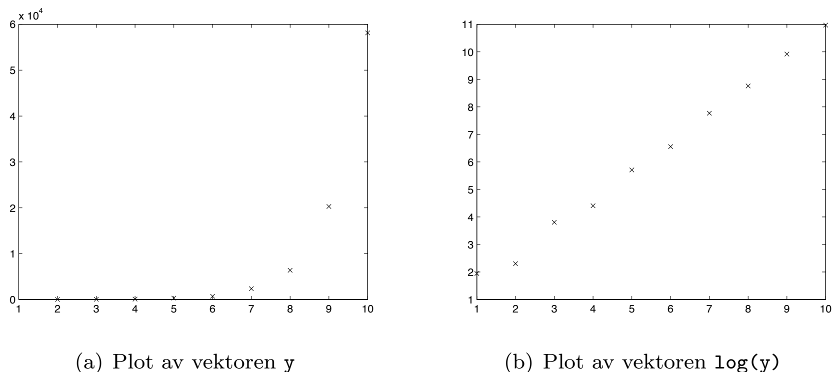
Seksjon 4.8

Koden nedenfor viser hvordan vi kan plotte løsningen av den (ikkehomogene) differenslikningen

$$y_{k+3} - 2y_{k+2} - 5y_{k+1} + 6y_k = -12k - 4. \tag{1}$$

```
n=10;             % proev f.eks. n=40 etterpaa!
y = [7 10 45];    % startverdier

for k=1:n-3
  y(k+3)= 2*y(k+2)+5*y(k+1)-6*y(k)-12*k-4;
end
y

figure(1); plot(y,'kx');

figure(2); plot(log(y),'kx');
```

Koden produserer to plott, begge vist i Figur 1. Grunnen til at vi også plottet



(a) Plot av vektoren `y`

(b) Plot av vektoren `log(y)`

Figur 1: Plott av løsningene til en differenslikning

logaritmen til løsningen av differenslikningen, er at løsningen av denne differenslikningen er

$$y_k = k^2 + 1 - (-2)^k + 3^k. \tag{2}$$

Det er lett å plotte dette for å sjekke at det faller sammen med det vi plottet i Figur 1. Tar vi logaritmen skal vi derfor få noe tilnærmet lineært når k vokser i plottet vårt. Dette forutsetter selvfølgelig at verdiene vi tar logaritmen av er positive (slik de er her). Løsningen (2) ble funnet ved først å finne den generelle løsning av den homogene likningen

$$y_{k+3} - 2y_{k+2} - 5y_{k+1} + 6y_k = 0. \tag{3}$$

Vi vet at dette koker ned til å løse likningen $r^3 - 2r^2 - 5r + 6 = 0$, som vi kan gjøre ved å skrive

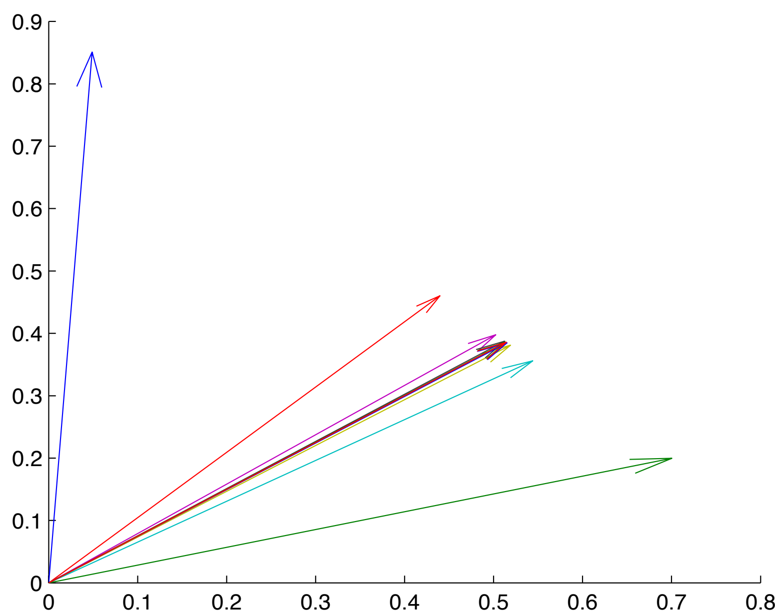
```
roots([1 -2 -5 6])
```

Det gir $r = 1$, $r = -2$, $r = 3$ som løsninger, og disse tallene finner vi igjen i (2).

Seksjon 4.9

Koden nedenfor viser hvordan vi kan generere en Markov-kjede, og plotte dens utviklingen over tid.

```
n=10;    P=[0.4 0.8; 0.6, 0.2]    % P er en stokastisk matrise
r=rand;  x=[r ; 1-r]      % x er en tilfeldig sannsynlighetsvektor
y=zeros(2,n);  hold on
for k=1:n
  y(:,k)=x;
  quiver([0],[0],[y(1,k)],[y(2,k)])
  pause(1);
  x=P*x;
end
y                          % y vil da inneholde de 10 forste leddene i
                           % Markov-kjeden assosiert med P og x.
```



Figur 2: Plott av vektorene i en Markov-kjede

Legg merke til den litt spesielle bruken av funksjonen `quiver` : denne funksjonen brukes vanligvis i MATLAB til å plotte et vektorfelt, men her bruker vi den til å plotte en og en vektor om gangen (inne i løkka). Videre brukes funksjonen `pause`, slik at vi kan se hvordan Markov-kjeden utvikler seg gradvis. Vi har plottet vektorene i Markov-kjeden i Figur 2. Det ser ut som de konvergerer mot en bestemt likevekstsvektor **q**. Denne vektoren kan vi lett finne eksplisitt ved å først regne ut nullrommet til `P - I` (som i Eks. 5 i avsn. 4.9).

Markov-kjeder dukker opp i mange sammenhenger. Et populært eksempel er surfing på nettet, der det å endre fra en tilstand til en annen svarer til å surfe fra en webside til en annen:

```
% En gruppe vid.skole elever "vandrer paa nettet" der bare tre
% nettsider er tilgjengelige (pga datafeil!): uio.no, ntnu.no, og vg.no

% Ved hvert tidskritt bytter noen nettsider med folgende andeler

%                        Fra:

%                 uio      ntnu     vg
%        uio      0.7      0.3      0.25
% Til:   ntnu     0.1      0.5      0.15
%        vg       0.2      0.2      0.6
```

```
% Vi starter med jevn fordeling paa sidene x=(1/3,1/3,1/3).
% Hvordan blir fordelingen etterhvert?

N=14;    % antall tidsskritt

% P=overgangsmatrisen
P= [ 0.7      0.3      0.25;
    0.1      0.5      0.15;
    0.2      0.2      0.6 ]

x=  [1/3;1/3;1/3]; % uniform startfordeling

% alternativt, kan det velges en vilkaarlig startfordeling:
% x =rand(3,1); x=x/sum(x)

F(:,1)=x;
for k=2:N
   x=P*x;
   F(:,k)=x;
end
F

plot(F');  % viser fordelingen som funk. av tiden for hver nettside
```

Plottet vi får gir inntrykk av konvergens mot en likevektsvektor $\mathbf{q}$, og vi kan lese fra plottet at $\mathbf{q}$ er tilnærmet lik $(0.47, 0.19, 0.33)$. (Igjen kunne vi lett ha regnet ut den eksakte verdien av $\mathbf{q}$).

Det er enkelt å generere tilfeldige regulære stokastiske matriser. Vi kan f.eks. bruke følgende funksjon

```
function A=randstocmatrix(n)

  % Returnerer en regulaer stokastisk matrise

  A = rand(n,n)+ 0.01;

  for (k = 1:n)

    A(:,k) = A(:,k)/sum(A(:,k));

  end
```

Legg merke til at vi inne i koden legger til 0.01 for å være sikre på at ingen koeffisient i den stokastiske matrisen som produseres er lik 0; dermed er det opplagt at denne matrisen blir regulær.

Kapittel 5

Seksjon 5.1, 5.2 og 5.3

Koden nedenfor plotter en tilfeldig enhetsvektor `x` og vektoren `A*x` mot hverandre (for matrisen `A = [2 4; 4 1]`).

```
figure(1);  A=[2 4; 4 1];
x=10*rand(2,1);  x=x/norm(x)

y=A*x

plot([0 x(1)],[0 x(2)]);  hold on;
plot([0 y(1)],[0 y(2)],'r');
xlim([0 5]); ylim([0 5]);
hold off
```

Plottet viser om `x` er en egenvektor for `A` eller ikke. I så fall er de to vektorene parallelle. Kjøres koden flere ganger vil man se at det er lite sannsynlig at det skjer.

For å be MATLAB om å regne ut egenverdier og egenverdier til en gitt kvadratisk matrise `A` kan man bruke funksjonen `eig`. Skriver du

```
[V,D]=eig(A)
```

vil `D` være en diagonalmatrise med egenverdiene til `A` på diagonalen, mens matrisen `V` består av tilhørende egenvektorer i tilsvarende kolonne. Eksempelvis, hvis `A` er en 2×2 matrise, vil den andre egenverdien være `D(2,2)`, med tilhørende egenvektor `V(:,2)`. Koden nedenfor verifiserer at disse er et egenverdi/egenvektor par.

```
A=[1 2; 4 3]

[V,D]=eig(A)

lambda=D(2,2)
x=V(:,2)

A*x - lambda*x % skal gi nullvektoren !
```

Hvis du ikke lister opp returparametrene `[V,D]` i kallet på `eig(A)` vil funksjonen returnere en kolonnevektor der komponentene er egenverdiene til `A` med eventuelle gjentakelser.

Gitt at vi vet at `lambda` er en egenverdi for en $n \times n$ matrise `A`, så kan vi finne tilhørende egenvektorer ved å løse systemet `(A-lambda*eye(n))*x=0`, det vil si ved å bestemme nullrommet til `A-lambda*eye(n)`. Koden nedenfor viser hvordan dette kan gjøres. For ordens skyld verifiseres det til slutt at resultatet faktisk er en egenvektor.

```
A=[1 2; 4 3]; lambda=5;  B=A-lambda*eye(2);

V=null(B)          % gir ortonormal basis for nullrommet

x=V(:,1)
A*x-lambda*x  % Skal gi nullvektoren !
```

Egenverdier og egenvektorer hjelper oss til å forstå likevektsvektorer for dynamiske systemer. En likevektsvektor er jo ikke noe annet enn en egenvektor tilhørende egenverdien 1, som samtidig er en sannsynlighetsvektor. Betrakt koden

```
P=[0.6  0.3; 0.4 0.7]; % stokastisk matrise
x=[0.72; 0.28]; x'   % f.eks. prosentvis andel av en startpopulasjon
pause(2);

for k=1:10
  x=P*x;
  x'
  pause(1);
end
```

Kjører du koden vil du se at systemet ser ut til å konvergere mot en bestemt likevektsvektor i det lange løp. Forklaringen er at den stokastiske matrisen P er regulær, og dermed vil enhver assosiert Markov kjede konvergere mot det entydige bestemte likevektsvektoren (jf. Teorem 18 i avsn. 4.9).

Koden nedenfor viser først hvordan MATLAB kan beregne verdien av et polynom p i et punkt. Polynomet må oppgies ved at koeffisientene til polynomet er listet i en vektor. Deretter brukes kommandoen `roots` til å finne (approksimative verdier av) røttene til p. Til slutt viser koden hvordan man kan finne det karakteristiske polynomet til en matrise.

```
p=[1 -6 5];    % svarer til polynomet p(t)= t^2 - 6t + 5
polyval(p,2.5)   % evaluerer p i t=2.5
roots(p)'       % gir roettene til p

A=[1 2 3; 4 5 6; 7 8 9]
q=poly(A)       % -q er da det karakteristiske polynomet til A
roots(q)'       % gir egenverdiene til A
```

NB: Legg merke til at `poly(A)`, regner ut $\det(\lambda I - A)$ (det er slik det karakteristiske polynomet ble definert i MAT1110), mens boka til Lay definerer det karakteristiske polynomet til A som $p_A(\lambda) = \det(A - \lambda I)$.

Hvis A er $n \times n$, så er

$$\det(\lambda I - A) = \det(-(A - \lambda I)) = (-1)^n \det(A - \lambda I) = (-1)^n p_A(\lambda) .$$

Dette betyr at `poly(A)` regner ut p_A hvis n er et liketall, mens det regner ut $-p_A$ hvis n er et oddetall.

Følgende kode bruker kommandoen `eig` og undersøker om to gitte matriser er diagonaliserbare ved å sjekke om egenvektorene som blir returnert faktisk kan brukes til å diagonalisere matrisen.

```
A=[1 2 3; 4 5 6; 7 8 9];     % A er diagonaliserbar!
[P,D]=eig(A)

rank(P)       % Hvis A er diag.bar er rang P = ant. kol. i A
[A*P  P*D]    % Skal ha A*P = P*D naar A er diagonaliserbar

A=[1 2; 0 1]            % A er ikke diagonaliserbar!
[P,D]=eig(A)    % Legg merke til hva P og D blir !

rank(P)        % Her er rang P < ant. kolonner i A
```

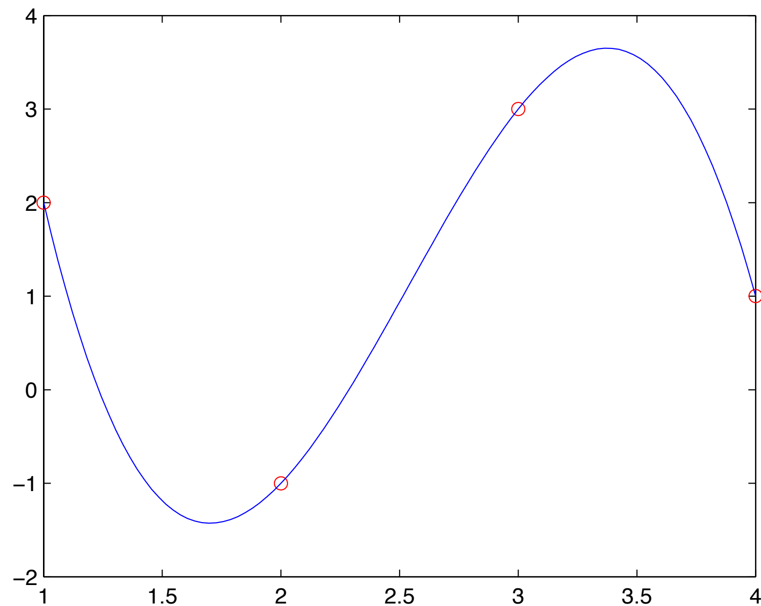
Seksjon 5.4 (Notat 2)

I Notat 2 så vi som en anvendelse av teorien at dersom vi starter med et gitt antall punkter i planet, la oss si $(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)$ der alle a_j-ene er forskjellige, så finnes det nøyaktig et *interpolerende* polynom p av grad $n-1$ som går gjennom disse $n+1$ punktene. En måte å beregne dette polynomet ble samtidig beskrevet.

Velger vi f.eks. de fire punktene $(1, 2), (2, -1), (3, 3), (4, 1)$, kan vi gå frem slik for å beregne p og plotte dets graf sammen med punktene:

```
a=[1; 2; 3; 4];
b=[2; -1; 3; 1];

M=[a.^0 a.^1 a.^2 a.^3];
q=M\b      % q gir koeff. til det interpolerende polynomet

plot(a, b,'or')
hold on;
t=linspace(1,4);
p=q(1) + q(2)*t + q(3)*t.^2+ q(4)*t.^3;
plot(t,p,'b')
```

Plottet denne koden produserer er vist i Figur 3. Det er fin øvelse å sette seg ned og programmere en funksjon som tar to vektorer `a` og `b` av samme lengde (la oss si n) som parametre, og som plotter polynomet av grad $n-1$



Figur 3: Plott av et interpolerende polynom

som går gjennom punktene $(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)$. Koden bør returnere en feilmelding hvis det ikke lar seg gjøre å finne et slikt polynom (tenk da over når det ikke mulig å finne et slikt polynom!).

Seksjon 5.5 og 5.6

Det er enkelt å bruke MATLAB for å tegne baner til diskrete dynamiske systemer av typen $\mathbf{x}_{k+1} = A\,\mathbf{x}_k$ i planet. Hvordan banene vil se ut avhenger av egenverdiene (reelle eller komplekse) til den reelle 2×2 matrisen A og av startvektoren.

Noen illustrasjoner når A har to forskjellige reelle egenverdier vil du få frem hvis du kjører de tre første kodene vi presenterer her.

```
figure(1);
L=60; U=60;
plot([-L U],[0 0],'b',[0 0],[-L U],'b'); % Tegner akser
axis([0 U 0 U]);
axis manual;
grid;
hold on;
```

```
A=[0.5 0.4; -0.14 1.1]

[P D] = eig(A)

x=[15;30];

for k=1:30
plot(x(1),x(2),'or');
pause(1)
x=A*x;                 % regner ut neste tilstand i systemet
end
```

I eksemplet ovenfor vil du se at origo er en såkalt *attraktor*, fordi begge egenverdiene har absoluttverdi mindre enn 1. Når du kjører denne koden (og de neste) vil du se at plotterutinen tar en pause før den plotter det neste punktet. Dette skyldes kallet på funksjonen `pause`.

I neste eksemplet er origo det som kalles en *repellor* i Lays bok, fordi begge egenverdiene har absoluttverdi størrre enn 1.

```
figure(2);
L=60; U=60;
plot([-L U],[0 0],'b',[0 0],[-L U],'b'); % Tegner akser
axis([0 U 0 U]);
axis manual;
grid;
hold on;

A=[0.5 0.4; -0.104 1.1]

[P D] = eig(A)

x=[15;30];

for k=1:20
plot(x(1),x(2),'or');
pause(1)
x=A*x; % regner ut neste tilstand i systemet
end
```

I det tredje eksemplet er origo et *sadelpunkt*, fordi en av egenverdiene har absolutt verdi mindre enn 1 mens den andre har absoluttverdi større enn 1.

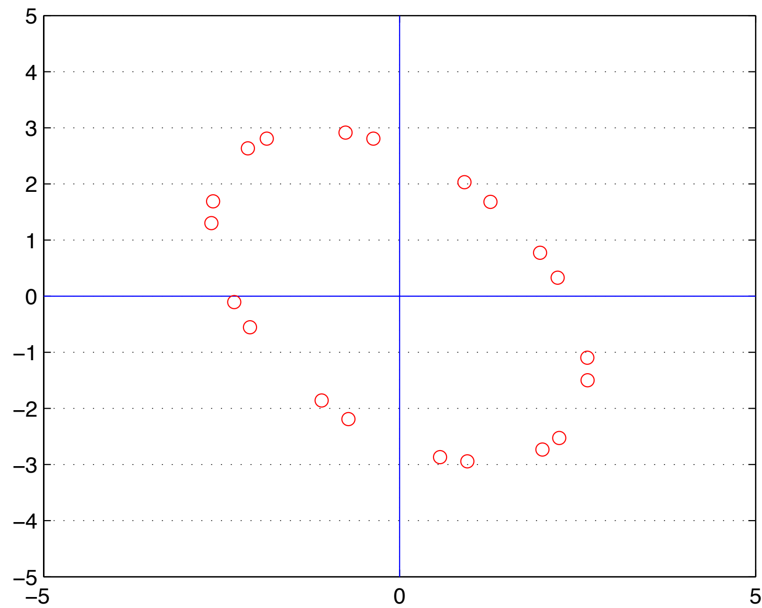
```
figure(3);
L=60; U=60;
plot([-L U],[0 0],'b',[0 0],[-L U],'b'); % Tegner akser
axis([-L U -L U]);
axis manual;
grid;
hold on;
```

```
A=[0.6 0; 0 1.5];

x=[50;3];

for k=1:8
plot(x(1),x(2),'or');
pause(1)
x=A*x;        % regner ut neste tilstand i systemet
end

x=[-20;3];    % velger annen startvektor

for k=1:8
plot(x(1),x(2),'ob');
pause(1)
x=A*x;        % regner ut neste tilstand i systemet
end


x=[20;0];       % velger  annen startvektor

for k=1:10
plot(x(1),x(2),'og');
pause(1)
x=A*x;         % regner ut neste tilstand i systemet
end
```

MATLAB kommandoen `eig` regner ikke bare ut reelle egenverdier, men også ut eventuelle komplekse egenverdier med tilhørende egenvektorer. Her er noen eksempler der A har to komplekse (konjugerte) egenverdier.

```
figure(4);
L=5; U=5;
plot([-L U],[0 0],'b',[0 0],[-L U],'b'); % Tegner akser
axis([-L U -L U]);
axis manual;
grid;
hold on;

A=[0.5 -0.6; 0.75 1.1]
                 % A har komplekse egenverdier med modulus r=1
x=ones(2,1)+rand(2,1);
for k=1:20
  plot(x(1),x(2),'or');
  pause(1)
  x=A*x;             % regner ut neste tilstand i systemet
end
```

Når du kjører koden ovenfor vil du se at systemet hverken beveger seg mot

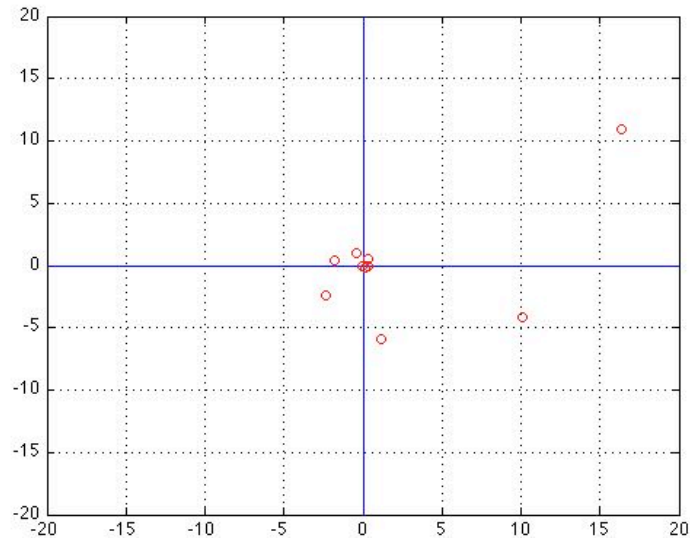


Figur 4: Plott av utviklingen i et diskret dynamisk system

origo, eller mot uendelig. Dette skyldes at alle egenverdiene til systemet har absoluttverdi lik 1. Det resulterende plottet er vist i Figur 4.

Et system der begge egenverdiene har absoluttverdi større enn 1 (så origo er en repellor) finner du her:

```
figure(5);
L=500; U=500;
plot([-L U],[0 0],'b',[0 0],[-L U],'b');
axis([-L U -L U]);
axis manual;
grid;
hold on;

A=[1 -1.5; 1.5 1]              % r = sqrt(13)/2 > 1  !!!

x=ones(2,1)+rand(2,1);

for k=1:10
  plot(x(1),x(2),'or');
  pause(1)
  x=A*x;
end
```



Figur 5: Plott av utviklingen i et annet diskret dynamisk system

Et system der begge egenverdiene har absoluttverdi mindre enn 1 (så origo er en attraktor) finner du her:

```
figure(6);
L=20; U=20;
plot([-L U],[0 0],'b',[0 0],[-L U],'b');
axis([-L U -L U]);
axis manual;
grid;
hold on;

A=[1 -1.5; 1.5 1];

B=inv(A)                          % r= 2/sqrt(13)  < 1  !!!
x=10*ones(2,1)+10*rand(2,1);

for k=1:10
  plot(x(1),x(2),'or');
  pause(1)
  x=B*x;
end
```

Det resulterende plottet er vist i Figur 5.

Seksjon 5.7

Denne seksjonen handler om systemer av første ordens diff.likninger på formen

$$\begin{bmatrix} x_1'(t) \\ \vdots \\ x_n'(t) \end{bmatrix} = A \begin{bmatrix} x_1(t) \\ \vdots \\ x_n(t) \end{bmatrix}. \tag{4}$$

der A er en reell $n \times n$ matrise. Slike systemer kan enkelt løses når A er diagonaliserbar (reelt eller komplekst) ved å se på egenverdiene og egenvektorene til matrisen A.

Lays bok konsentrerer seg stort sett om tilfellet der $n = 2$.

I Lays bok er det angitt formler for basiser for løsningsrommet når egenverdiene er komplekse. Nedenfor har vi tatt med koden for å plotte disse basisfunksjonene for matrisen

$$A = \begin{bmatrix} -2 & -2.5 \\ 10 & -2 \end{bmatrix}, \tag{5}$$

som har to (konjugerte) komplekse egenverdier med negativ realdel.

```
A=[-2 -2.5; 10 -2]

[U,D] = eig(A)
a = real(D(1,1));     % Merk at a < 0
b = imag(D(1,1));

y1 = []; y2 = [];
for t=-2:0.05:3;
  y1 = [ y1 (real(U(:,1))*cos(b*t) - imag(U(:,1))*sin(b*t))*exp(a*t) ];
  y2 = [ y2 (real(U(:,1))*sin(b*t) + imag(U(:,1))*cos(b*t))*exp(a*t) ];
end

figure(1)
plot(y1(1,:),y1(2,:),'g');
figure(2)
plot(y2(1,:),y2(2,:),'r');
figure(3)
hold on;

axis([-5 5 -10 10])
plot(y1(1,:),y1(2,:),'g');
plot(y2(1,:),y2(2,:),'r');

r=-5:0.5:5;
s=-10:0.5:10;
[x,y]=meshgrid(r,s);
u = -2 * x -2.5 * y;
v = 10*x-2*y;
quiver(x,y,u,v);
```

I denne koden har vi brukt to nye funksjoner:

- `real`, som returnerer realdelen til et komplekst tall,
- `imag`, som returnerer imaginærdelen til et komplekst tall.

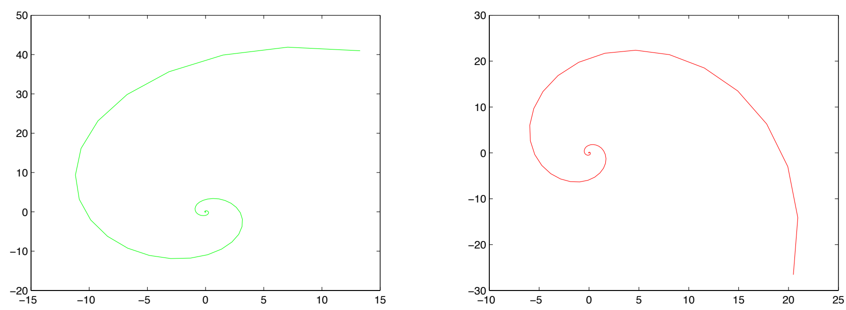
Et tilsvarende eksempel for matrisen

$$A = \begin{bmatrix} 0.5 & 1 \\ -1 & 0 \end{bmatrix}, \tag{6}$$

der egenverdiene istedet er komplekse med positiv realdel, er som følger:

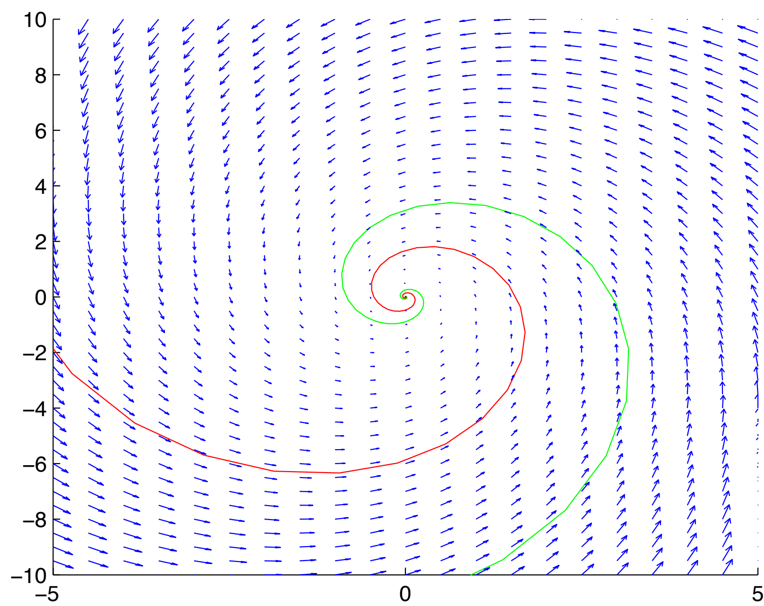
```
A=[0.5 1; -1 0]

[U,D] = eig(A)
a = real(D(1,1)); % a > 0
b = imag(D(1,1));

y1 = []; y2 = [];
for t=0:0.05:10;
  y1 = [ y1 (real(U(:,1))*cos(b*t) - imag(U(:,1))*sin(b*t))*exp(a*t) ];
  y2 = [ y2 (real(U(:,1))*sin(b*t) + imag(U(:,1))*cos(b*t))*exp(a*t) ];
end

figure(1)
axis([-3 6 -10 4])
plot(y1(1,:),y1(2,:),'g');
figure(2)
axis([-3 6 -10 4])
plot(y2(1,:),y2(2,:),'r');
figure(3)
hold on;

axis([-3 6 -10 4])
plot(y1(1,:),y1(2,:),'g');
plot(y2(1,:),y2(2,:),'r');
r=-3:0.4:6;
s=-10:0.4:4;
[x,y]=meshgrid(r,s);
u = 0.5 * x + y;
v = -x;
quiver(x,y,u,v);
```

Figurene 6 og 7 viser løsningskurvene som disse kodene produserer. Skriver man systemet (4) om på vektorform , sier det at tangentvektoren $\mathbf{x}'(t)$ er gitt ved vektoren $A\,\mathbf{x}(t)$. I plottene har vi derfor også tegnet inn vektorfeltet $\mathbf{F}(\mathbf{x}) = A\,\mathbf{x}$. I MAT1110 lærer man å tegne vektorfelt ved hjelp av funksjonen `quiver`, og den har vi brukt her. Det er nyttig å plotte vektorfeltet i slike illustrasjoner, for da kan vi faktisk se om løsningene beveger seg mot origo, eller om de beveger seg bort fra origo.



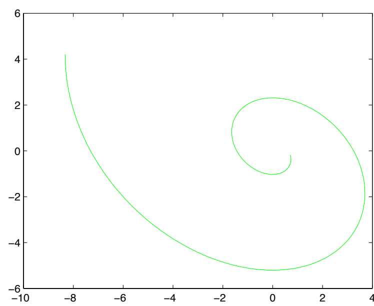
(a) Løsningen $y1$

(b) Løsningen $y2$

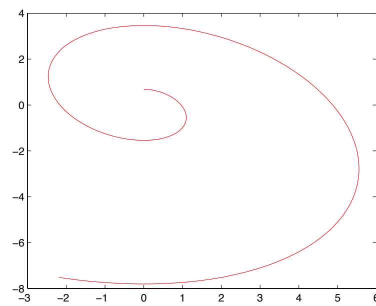


(c) Løsningene $y1$ og $y2$ sammen med vektorfeltet

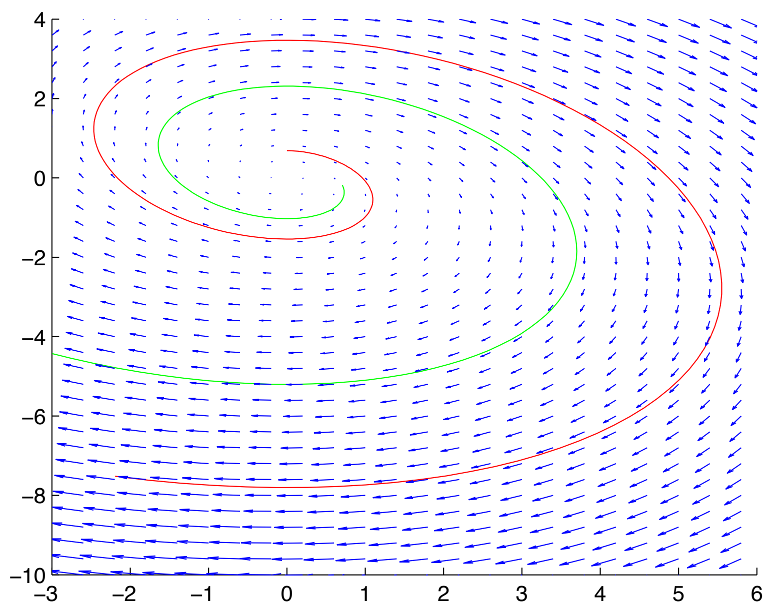
Figur 6: Løsninger av et førsteordens lineært differensiallikningssystem med komplekse egenverdier. Realdelen til egenverdiene er mindre enn 0.



(a) Løsningen $y1$



(b) Løsningen $y2$



(c) Løsningene $y1$ og $y2$ sammen med vektorfeltet

Figur 7: Løsninger av et førsteordens lineært differensiallikningssystem med komplekse egenverdier. Realdelen til egenverdiene er større enn 0.

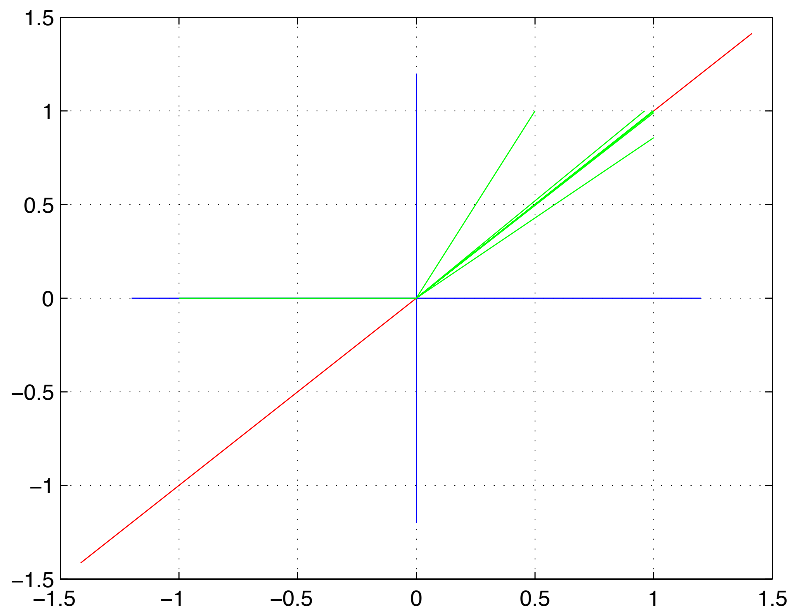
Seksjon 5.8

Et viktig ledd i potensmetoden er å skalere vektoren ved hver iterasjon. Hvis det ikke gjøres vil vektoren kunne vokse mot uendelig. Dette inntreffer hvis du f.eks. kjøre denne koden:

```
L=150; U=150;
plot([-L U],[0 0],'b',[0 0],[-L U],'b'); % Tegner akser
axis([-L U -L U]);
axis manual;  grid;
hold on;

A=[0.5,1.5; 1, 1]
x=[-10;10]                  % x = startvektor
[V D]=eig(A)            % Egenverdiene til A er -0.5 og 2

plot(200*V(1,2)*[-1,1],200*V(2,2)*[-1,1],'r');
for j=1:7
  pause(1)
  plot([0 x(1)],[0 x(2)],'g');
  x=A*x;
end
```

Hvis alle egenverdiene har absoluttverdi 1, trenger man ikke å skalere vektoren ved hver iterasjon. Vektoren vil jo da ikke vokse mot uendelig. Et slikt eksempel er vist i koden nedenfor. Her oppdager vi et annet problem, nemlig at vektoren ikke konvergerer, men varierer mellom fire tilstander. Dette fenomenet kan skje når vi ikke har en egenverdi som dominerer alle de andre egenverdiene. I dette tilfellet har vi to egenverdier med lik absoluttverdi.

```
L=12; U=12;
plot([-L U],[0 0],'b',[0 0],[-L U],'b');   % Tegner akser
axis([-L U -L U]);
axis manual;  grid;
hold on;

A=[1,2; -1, -1]   % A har ingen reelle egenverdier
x=[-2 ;4]         % x = startvektor
[V D]=eig(A)      % Egenverdiene til A er i og -i,
                          % saa A har ikke en strengt dom. egenverdi

for j=1:9
  pause(1)
  plot([0 x(1)],[0 x(2)],'g');
  x=A*x;
end        % x-ene  tar  bare fire verdier som det byttes imellom.
```



Figur 8: 7 iterasjoner i potensmetoden

Det er enkelt å regne ut at matrisen

$$A = \begin{bmatrix} 0.5 & 1.5 \\ 1 & 1 \end{bmatrix}$$

har egenverdiene -0.5 og 2. Vektoren `[1; 1]` er da en egenvektor som tilhører den dominante egenverdien til A, som i dette tilfellet er 2. Vi kan gjenta potensmetoden med skalering 7 ganger og få frem plottet vist ovenfor i Figur 8 ved å kjøre følgende kode:

```
L=1.2; U=1.2;
plot([-L U],[0 0],'b',[0 0],[-L U],'b'); % Tegner akser
axis([-L U -L U]); axis manual; grid;
hold on;

A=[0.5,1.5; 1, 1]
x=[-1;0]                          % x = startvektor
[V D]=eig(A)              % Egenverdiene til A er -0.5 og 2

% plotter egenrommet til egnverdien 2 (som er dominant)
plot(2*V(1,2)*[-1,1],2*V(2,2)*[-1,1],'r');

plot([0 x(1)],[0 x(2)],'g'); pause(1)
for j=1:7
```

```
  y=A*x;
  [t, r] = max(abs(y));        % r er komponenten som gir maks
  mu=y(r), x=y/mu              % skalerer med mu
  error=  max(abs(mu*x-A*x))   % angir et maal for feilen
  plot([0 x(1)],[0 x(2)],'g');
  pause(1)
end
```

Selve potensmetoden med skalering kan implementeres på funksjonsform slik:

```
function [xvals,muvals]=powermethod(A,x,numtimes,tol)
  xvals=[];
  muvals=[];

  for r=1:numtimes
    x = A*x;
    [maxval,maxnr]=max(abs(x));
    mu = x(maxnr);
    muvals=[muvals mu];
    xvals=[xvals (1/mu)*x];
    % R = x'*A*x/(x'*x)
    error=max(abs(A*x-mu*x));
    if error<tol
      break;
    end
  end
```

Funksjonen returnerer en vektor med estimatene på egenvektorene, og en annen vektor med estimatene på egenverdiene. Koden kjører potensmetoden gjentatte ganger, inntil avviket fra å være en egenvektor er mindre enn en angitt toleranse. I koden har vi brukt en variant av funksjonen `max`, som returnerer to verdier: både selve maks-verdien, og den indeksen hvor maksimum inntreffer. Legg merke til at en linje er kommentert ut i koden. På denne linjen regnes ut det som kalles Rayleigh-kvotienten; den gir også et estimat på den største egenverdien (faktisk et enda bedre estimat enn μ).

En enkel implementasjon av den inverse potensmetoden for matrisen A ovenfor er som følger:

```
A=[0.5,1.5; 1, 1]

x=[-1;0]          % x = startvektor
k=7               % k= antall iterasjoner

[V D]=eig(A)      % Egenverdiene til A er -0.5 og 2

% skal "finne" egenverdien -0.5; "tipper" denne verdien:

a= -0.2;
```

```
B=A-a*eye(2);


for j=1:k
  y=B\x;
  [maxval , maxnr] = max(abs(y));
  mu = y(maxnr);
  nu=a + 1/mu;           % nu er tilnaermet lik egenverdien
  x=y/mu;                % x er tilnaermet tilh. egenvektoren
  error =  max(abs(nu*x-A*x))    % angir maal for  feilen
 end
```

Her kjøres metoden 7 ganger, og i hver iterasjon skrives feilen ut.

Selve den inverse potensometoden kan implementeres på funksjonsform slik:

```
function [xvals,nuvals]=inversepowermethod(alpha,A,x,numtimes,tol)
  xvals=[];
  nuvals=[];
  n= length(x);

  for r=1:numtimes
    reduced = rref( [(A-alpha*eye(n)) x]);
    y = reduced(:,n+1);
    [maxval,maxnr]=max(abs(y));
    mu = y(maxnr);
    xvals=[xvals (1/mu)*y];
    nuvals=[nuvals alpha+(1/mu)];
    error=max(abs(A*x-nu*x));
    if error<tol
      break;
    end
  end
```

Parameterne for denne funksjonen er: en verdi `alpha` rundt hvilken det skal letes etter eventuell egenverdi, input-matrisen, maksimalt antall iterasjoner og feiltoleranse.

Kapittel 6

Seksjon 6.1

Det euklidske indreproduktet (prikkproduktet) av to vektorer `x` og `y` i $\mathbb{R}^n$ regnes enkelt i MATLAB ved

```
x'* y
```

Normen (eller lengden) til `x` regnes ved

```
norm(x)
```

Man kan bruke indreproduktet til å finne vinkelen mellom to vektorer i $\mathbb{R}^2$ eller i $\mathbb{R}^3$ (se Lay s. 335). Det er ingenting i veien for å bruke samme formel til å definere vinkelen mellom vektorer i $\mathbb{R}^n$ for $n \geq 4$. Her er et eksempel på hvordan vinkelen mellom to vektorer i $\mathbb{R}^4$ beregnes:

```
x = [1; 1; 1; 1]
y = x + 2*rand(4,1)

c = x'*y/(norm(x)*norm(y)) % dette gir cosinus til
                           % mellomliggende vinkel

rad = acos(c);             % gir vinkelen i radianer;
                           % acos er omvendt funksjon til cosinus
angle = 180*rad/pi         % omregner vinkelen til grader
```

Seksjon 6.2

Vi er ofte interessert i å finne ut om kolonene (eller radene) til en $m \times n$ matrise `A` er ortogonale eller ortonormale på hverandre. Vi kan da be MATLAB regne ut

```
A'*A
```

Kolonene til `A` er ortogonale (resp. ortonormale) hvis og bare hvis svaret her er en diagonalmatrise (resp. identitetsmatrisen) (jf. Teorem 6 i kap. 6). Det samme gjelder for radene til `A`.

Mange funksjoner i MATLAB returnerer ortonormale mengder, f.eks. `null` og `orth`, som gir matriser der kolonnene danner en ortonormal basis for henholdsvis nullrommet og kolonnerommet til input-matrisen. Følgende kode bruker `orth` til å lage en matrise med ortonormale kolonner. Deretter endrer vi lengden av kolonnene i matrisen; den nye matrisen har da fremdeles ortogonale kolonner.

```
n=5; p=4;  A = rand(n,p) % tilfeldig 5 x 4 matrise

B=A'*A      % Sjekker om A har  ortogonale/ortonormale kolonner.
            % Hoyst usannsynlig at det er tilfelle !

U=orth(A)   % kolonnene i U gir ortonormal basis for Col A
S=U'*U      % verifiser ortonormale kolonner i U

 % Lager naa en matrise V med ortogonale kolonner
 % der kolonnene har lengde mellom 0 og 10
for j=1:4
       V(:,j)=10*rand(1)*U(:,j);
end

V
T=V'*V      % Sjekker at V har ortogonale kolonner
```

I neste kode bruker vi `orth` til å finne en ortonormal basis $\mathcal{U}$ for kolonnerommet til en tilfeldig valgt $n \times n$ matrise A (med $n = 4$): vi tenker her at $\mathcal{U}$ består av kolonnene til `U=orth(A)` Med sannsynlighet 1 vil A ha full rang, dvs A vil være invertibel, så $\mathcal{U}$ vil være en ortonormal basis for $\mathbb{R}^n$. Deretter velger vi en tilfeldig vektor `v` og beregner de ortogonale projeksjonene av `v` langs vektorene i $\mathcal{U}$. Så verifiserer vi at Teorem 5 i avsnitt 6.2 holder. Til slutt sjekker vi at Pythagoras læresetning gjelder også her.

```
n=4; A=rand(n);      % tilfeldig 4 x 4 matrise
rank(A)              % Sjekker at A har full rang.

U=orth(A)            % Kol.til U skal danne ort.n. basis for Col A = R4
M=U'*U               % verifiser ortonormale kol.

v=floor(10*rand(n,1)) % tilf. vektor med heltallige komp. mellom 0 og 9

for j=1:n
    c(j)=v'*U(:,j);             % koef. for U(:,j)
    P(:,j)=c(j)*U(:,j);         % ort. proj. langs U(:,j)
end
P                               %viser alle de ort. proj. som kolonner

s=sum(P,2);                     % gir summen av alle ort. projeksjonene
v_og_sum_av_proj=[v s]          % bor vaere like i henhold til Teorem 5!

pythagoras=[norm(v)^2  norm(c)^2] % bor vaere like i henhold til Pyth.!
```

Seksjon 6.3

Den første koden viser hvordan vi kan beregne den ortogonale projeksjonen av en vektor `v` på et underrom `L` utspent av to vektorer i $\mathbb{R}^3$. Deretter beregnes avstanden fra `v` til `L`. Til slutt plukker vi ut 1000 tilfeldige vektorer i `L` og beregner avstanden fra disse til `v`. Minimum av alle disse avstandene vil da være større (eller lik) avstanden fra `v` til `L`.

```
u1=[2; 1; 4]; u2=[1; -3; 1/4]; % er ortogonale;utspenner et underrom L

v = [1; 1; 1];
proj =((v'*u1)/(u1'*u1))*u1 + ...
      ((v'*u2)/(u2'*u2))*u2       % Gir proj. av v paa L (jf. Teorem 5)

proj'*(v-proj)                   % skal gi 0

dist_v_L = norm(v-proj)            % gir avstanden fra v til L

% trekker 1000 vektorer i L og beregner deres avstand til v:
for j=1:1000
    x = [u1 u2]*rand(2,1);
    dist(j) = norm(v-x);
end

% beregner minste avstand m_d fra v til de utplukkede vektorene i L;
% dist_v_L skal vaere mindre eller lik m_d !

m_d = min(dist)
```

Koden nedenfor beregner den ortogonale projeksjonen `Proj` av en vektor `y` ned på kolonnerommet til en matrise `A` på to forskjellige måter, og verifiserer at disse er like.

Den ene måten baserer seg på Teorem 8 i avsnitt 6.3: først beregnes projeksjon av `y` på hver vektor i en ortogonal basis for kolonnerommet; deretter legges alle disse sammen, og det gir oss den ønskede projeksjonen `Proj`. Vi sjekker da at vektoren `y - Proj` står ortogonalt på `Col A`, slik det skal i henhold til Teorem 8.

Den andre måten bruker Teorem 10 i avsnitt 6.3: projeksjonen `Proj` beregnes ved hjelp av matrisen UU^T, der matrisen U er valgt slik at dens kolonner utgjør en ortonormal basis for kolonnerommet til `A`.

```
n=4; p=3; A=rand(n,p)     % tilfeldig 4 x 3 matrise
U=orth(A)          % kolonnene i U gir ortonormal basis for W=Col A
M=U'*U               % verifiserer ortonormale kolonner i U

y=floor(10*rand(n,1)) % en vektor med tilfeldige koeff. mellom 0 og 10
for j=1:p
   c(j)= y'*U(:,j);     % gir koeff. langs U(:,j)
   P(:,j)= c(j)*U(:,j); % gir proj.av y  paa U(:,j)
end
```

```
P                % kolonnene til P gir proj. av y langs kol. til U

Proj = sum(P,2)  % gir projeksjonen av y paa W = Col A

z = y - Proj_y    % gir komponenten til y langs det ort. kompl. til W
z'*A             % Sjekker at z er ortogonal paa W

Q=U*U'           % Q er stand. matrisen til proj. paa W
Q*y              % skal gi det samme som Proj
Proj-Q*y       % skal gi nullvektoren (tilnaermet)
```

Seksjon 6.4

Gram-Schmidt prosessen kan kodes som en MATLAB-funksjon. Vi kan gjøre det rett frem ved å følge oppskriften fra Teorem 11 i avsnitt 6.4:

```
function V=gramschmidtdirect(A)
  n=size(A,2);
  V=A;
  for j=1:n
    v0=V(:,j);
    for i=1:j-1
      v1=V(:,i);
      V(:,j)=V(:,j)-((v0'*v1)/(v1'*v1))*v1;
    end
  end
```

Kolonnene til output-matrisen V vil utgjøre en ortogonal basis for kolonnerommet til input-matrisen A etter kallet på funksjonen `gramschmidtdirect`. Hvis vektorene i A skulle være lineært avhengige vil man merke det i Gram-Schmidt prosessen ved at noen av de kolonnene man får i V blir null-kolonner. I praksis vil dette nesten aldri inntreffe fordi avrundingsfeil gjør at man sjelden får noe som er eksakt lik null.

Det som trekkes fra i den innerste løkka i koden for `gramschmidtdirect` kan sees på som et matriseprodukt (jf. Teorem 10 i avsnitt 6.4 for beregning av projeksjoner). Bruker vi dette blir koden noe kortere:

```
function V=gramschmidt(A)
  n=size(A,2);
  V = A;
  for j=1:n
    V(:,j) = V(:,j) - V(:,1:(j-1))*V(:,1:(j-1))'*V(:,j);
  % V(:,j) = V(:,j)/norm(V(:,j));  % for aa normalisere vektorene
  end
```

Det er lett å modifisere denne koden slik at alle de ortogonale vektorene får lengde 1, m.a.o. blir ortonormale: du trenger bare ta bort kommentartegnet til venstre inn i for-løkka.

Følgende kode tester om funksjonen `gramschmidt` faktisk returnerer en ortogonal basis:

```
A=rand(10,4);    % Tilfeldig matrise

V=gramschmidt(A)

V'*V              % Er diagonal hvis kol.i V er ortogonale
```

Koden i `gramschmidt`-funksjonen vi har laget er numerisk ustabil: på grunn av avrundingsfeil vil koden kunne produsere et fullt sett med ortogonale vektorer, selv om vektorene vi startet med var lineært avhengige. Funksjonen `orth` tar høyde for slike avrundingseffekter, og er derfor å foretrekke; `orth` sørger i tillegg for at vektorene får lengde 1.

Hvis vi kjenner en ortogonal mengde av vektorer i $\mathbb{R}^m$ kan den utvides til en ortogonal basis for hele $\mathbb{R}^m$ ved hjelp av følgende kode:

```
function V=gramschmidtutvid(U)
  [m,n]=size(U);
  U = [U eye(m)];
  [R,piv]=rref(U);
  V=U(:,piv);
  for j=(n+1):m
    V(:,j) = V(:,j) - V(:,1:(j-1))*V(:,1:(j-1))'*V(:,j);
  % V(:,j) = V(:,j)/norm(V(:,j)); % for aa normalisere vektorene
  end
```

Her hekter vi først på identitetsmatrisen til høyre for input-matrisen `U` (som antaes å ha ortogonale (evt ortornormale) kolonner). For å finne hvilke kolonner fra denne utvidede matrise som utgjør en basis for $\mathbb{R}^m$, bruker vi en variant av `rref` som returnerer hvilke kolonner som er pivotkolonner. Disse kolonnene blir deretter input til Gram-Schmidt prosessen. Vi skal anvende denne metoden når vi senere programmerer utregning av SVD'en til en matrise i kapittel 7.

Boka nevner at Gram-Schmidt prosessen på kolonnene til en matrise $A = [\mathbf{x}_1\ \mathbf{x}_2\ \cdots\ \mathbf{x}_n]$ gir essentielt det samme som QR-faktoriseringen av matrisen, uten at dette er utdypet helt. For å se dette bedre, skriver vi først Teorem 11 på formen

$$\mathbf{x}_j = \mathbf{v}_1 \frac{\mathbf{x}_j \cdot \mathbf{v}_1}{\mathbf{v}_1 \cdot \mathbf{v}_1} + \cdots + \mathbf{v}_{j-1} \frac{\mathbf{x}_j \cdot \mathbf{v}_{j-1}}{\mathbf{v}_{j-1} \cdot \mathbf{v}_{j-1}} + \mathbf{v}_j. \tag{7}$$

Setter vi $Q_1 = [\mathbf{v}_1\ \mathbf{v}_2\ \cdots\ \mathbf{v}_n]$ og lar R_1 være den øvretriangulære matrisen med koeffisienter

$$(R_1)_{ij} = \begin{cases} \frac{\mathbf{x}_j \cdot \mathbf{v}_i}{\mathbf{v}_i \cdot \mathbf{v}_i} & \text{når } i < j \\ 1 & \text{når } i = j \\ 0 & \text{når } i > j, \end{cases}$$

ser vi at (7) sier at $A = Q_1 R_1$ (beregnet kolonne for kolonne). Denne faktoriseringen oppfyller alle egenskapene til en QR-faktorisering, med unntak av at kolonnevektorene i Q_1 ikke nødvendigvis har lengde 1. Dette kan vi oppnå ved å skrive $A = Q_1 D^{-1} D R_1$, der D er diagonalmatrisen med $D_{ii} = \|\mathbf{v}_i\|$.

Regner vi ut $Q = Q_1 D^{-1}$ får vi at kolonne nr. j i Q er $\mathbf{u}_j = \frac{\mathbf{v}_j}{\|\mathbf{v}_j\|}$, som svarer til vektorene fra Gram-Schmidt prosessen i normalisert form.

Regner vi ut $R = DR_1$ får vi at

$$R_{ij} = \begin{cases} \|\mathbf{v}_i\| \frac{\mathbf{x}_j \cdot \mathbf{v}_i}{\mathbf{v}_i \cdot \mathbf{v}_i} = \mathbf{x}_j \cdot \left(\frac{\mathbf{v}_i}{\|\mathbf{v}_i\|} \right) = \mathbf{x}_j \cdot \mathbf{u}_i & \text{når } i < j \\ \|\mathbf{v}_i\| & \text{når } i = j \\ 0 & \text{når } i > j, \end{cases}$$

Vi ser nå at $A = QR$ oppfyller alle egenskaper til QR-faktoriseringen.

Dette betyr at hvis vi normaliserer vektorene etter hvert steg i Gram-Schmidt prosessen, så vil $\mathbf{x}_j \cdot \mathbf{u}_i$ (som vi regner ut underveis i Gram-Schmidt prosessen) bli elementene i matrisen R, mens de ortonormale vektorene som fremkommer i Gram-Schmidt prosessen blir elementene i matrisen Q. Koden for QR-faktorisering kan dermed skrives slik:

```
function [Q,R]=qrfact(A)

  [m,n]=size(A);
  Q = A;
  R = zeros(n,n);

  for j=1:n
    R(1:(j-1),j) = Q(:,1:(j-1))'*Q(:,j);
    Q(:,j) = Q(:,j) - Q(:,1:(j-1))*R(1:(j-1),j);
    R(j,j) = norm(Q(:,j));
    Q(:,j) = Q(:,j)/R(j,j);
  end
```

Her har matrisen V endret navn til Q, i tråd med notasjonen i boka. I løkka i koden ovenfor regnes først ut alle indreproduktene vi trenger i en iterasjon av Gram-Schmidt prosessen (svarer til neste kolonne i matrisen R) ved hjelp av et matriseprodukt. I neste linje regnes deretter ut neste vektor (svarer til neste kolonne i matrisen Q), der vi igjen bruker at (7) har form som et matriseprodukt. Med andre ord, hver iterasjon i løkka fyller ut en ny kolonne i matrisene R og Q. Til slutt normaliseres vektoren.

Følgende kode tester at funksjonen `qrfact` faktisk returnerer matriser Q, R som er slik at kolonnene i Q er ortonormale og $A = QR$. Dessuten skriver den ut R slik at vi kan se at den er øvretriangulær med positive elementer på diagonalen.

```
A=rand(10,4)     % tilfeldig matrise

[Q,R]=qrfact(A)  % R skal v?re ovretriangulaer med pos. elem. paa diag.
Q'*Q             % Blir lik I hvis kol. til Q er ortonormale
Q*R              % Skal gi tilbake A
```

I boka står det at QR-faktoriseringen kan brukes til å approksimere egenverdiene til en matrise. For symmetriske matriser er det enkelt å få til (men det er verre å begrunne hvorfor det virker !). Se f.eks. på følgende kode:

```
B = rand(6,6); % tilfeldig matrise

A = B'+B       % A blir symmetrisk (dvs A'=A)
Egenv = eig(A)

for k=1:100
  [Q,R]=qrfact(A);
  A = R*Q;
end

A_100 = A     % Sammenlikn egenverdiene til A
              % og diagonalelementene til A_100
```

Kjører du denne koden vil du se at matrisen A_{100} er tilnærmet lik en diagonalmatrise, og at dens diagonalelementer gir approksimasjoner av egenverdiene til A. Konvergensen kan være noe treg og det vil kunne hende at 100 iterasjoner ikke er nok (du må da endre koden og velge et høyere tall). En av grunnideene i koden er at hvis $A = QR$, så er

$$RQ = Q^T Q\, RQ = Q^T AQ = Q^{-1}AQ$$

noe som viser at $A = QR$ og RQ er similære matriser; dermed er egenverdiene til RQ de samme som egenverdiene til A.

Seksjon 6.5

I dette avsnittet har vi sett at vi kan finne minstekvadraters løsninger av lineære likningssystemer på to måter:

1. ved å regne ut projeksjonen ned på kolonnerommet og løse likningssystemet med denne som ny høyreside,
2. ved å sette opp normallikningene direkte og løse disse.

Koden nedenfor viser eksempler på begge delene.

```
A = [3 5 1; 1 1 1; -1 5 -2; 3 -7 8]
b = [1;0;0;0]
R = rref([A b])   % Ser fra R at systemet Ax=b er inkonsistent

% Metode 1
fprintf('** ortonormal basis\n');
W = orth(A)
fprintf('** projeksjon av b paa Col A\n');
proj_b = (b'*W(:,1))*W(:,1)+(b'*W(:,2))*W(:,2)+(b'*W(:,3))*W(:,3)

x_mk1 = A \ proj_b           % Skal gi minstekvad. loesning

Feilestimat = norm(A*x_mk1-b) % gir et maal paa feilen
```

```
% Metode 2
fprintf('** loeser normalikningene\n');
x_mk2 = A'*A \ A'*b         % Skal ogsaa gi minstekvad. loesning
```

Seksjon 6.6

Vi gir her koden til flere eksempler som illustrerer henholdsvis lineær regresjon, approksimasjon med 2. grads polynomer, approksimasjon med harmonisk svingning og multippel lineær regresjon.

```
% Eksempel 1 -- Lineaer regresjon

% x og y koordinatene til punktene (0,1),(1,2),(2,1),(3,5):
x = [0; 1; 2; 3];  y = [1; 2; 1; 5];
m = 4;        % antall punkter

X = [ones(m,1) x]        % X er designmatrisen
beta = X'*X \ X'*y        %gir minste kvadraters losning

t = linspace(-1, 4);
f = beta(1)+ beta(2)*t;  %gir minste kvadraters linje
figure(1)
plot(x, y,'or')         % plotter datapunktene
hold on
plot(t,f,'b')           % plotter minste.kvad.-linje

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Eksempel 2 -- Lineaer regresjon

% Datapuktene fremkommer her ved aa velge punkter paa linjen
% y=1+2*x og forstyrre y-koordinatene en smule (tenker at
% dette skyldes maaleusikkerhet eller stoey)

x = (0:0.2:4)'; m = length(x);     % m er antall punkter
y1 = ones(m,1) + 2*x;
y = y1 + 0.3*(0.5*ones(m,1)-rand(m,1));

X =[ones(m,1) x];       % X er designmatrisen
beta = X'*X \ X'*y     % gir minste kvadraters losning
t = linspace(-1,5);
f = beta(1)+ beta(2)*t;  % gir minste kvadraters linje

figure(2)
plot(x, y,'or')     % plotter datapunktene
hold on
plot(t,f,'b')       % plotter minste kvad.-linjen
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% Eksempel 3 -- Lineaer modell med 2. grads polynomer

% x og y koordinatene til  punktene:
x = [0; 1; 2; 3]; y = [1; 2; 1; 5]; m = 4;

X = [ones(m,1) x x.^2]          % X er designmatrisen
beta = X'*X \ X'*y              % gir minste kvadraters losning

t = linspace(-1, 4);
p = beta(1)+ beta(2)*t + beta(3)*t.^2;  % gir minste kvad. 2.gradspol.

figure(4)
plot(x, y,'or')     % plotter datapunktene
hold on
plot(t,p,'b')       % plotter minste.kvad. polynomet p

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Eksempel 4 -- Lineaer modell med 2.grads pol.

x=(1:0.2:2.4)'; m = length(x);          % m er antall punkter

y1 = ones(m,1)+ 3.75*x - x.^2;
y =y1+ 0.3*(0.5*ones(m,1)-rand(m,1));

X  =[ones(m,1) x x.^2];             % X er designmatrisen

beta = X'*X \ X'*y                 % gir minste kvadraters losning

t = linspace(0,5);
p = beta(1)+ beta(2)*t + beta(3)*t.^2;  % gir minste kvad. 2.grads pol.

figure(4)
plot(x, y,'or')     % plotter punktene
hold on
plot(t,p,'b')       % plotter p

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Eksempel 5 -- Lineaer modell harmonisk svingning

% Skal finne en harmonisk svingning med periode 4 som gaar gjennom fire
% oppgitte punker.

% x og y koordinatene til punktene:
x = [0; 1; 2; 3]; y = [-3; 3; 5; -2];  m = 4;

X = [ones(m,1) sin((0.5)*pi*x) cos((0.5)*pi*x)]   % X er designmatrisen

beta = X'*X \ X'*y                          % gir minste kvadraters losning
```

```
t=linspace(-2, 6);
f = beta(1)+ beta(2).*sin((0.5)*pi*t) + beta(3).*cos((0.5)*pi*t);

figure(5)
plot(x, y,'or')
hold on;
plot(t,f,'b')

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Eksempel 6 -- Lineaer modell harmonisk svingning

% Som i Eks.5, men med flere punkter; disse kommer fra en "forstyrret"
% harmonisk svingning

x = (0:0.2:4)'; m = length(x); % m er antall punkter

y1 = ones(m,1)+2.*sin((0.5)*pi*x)-cos((0.5)*pi*x);
y = y1+ 0.4*(0.5*ones(m,1)-rand(m,1));

X = [ones(m,1) sin((0.5)*pi*x) cos((0.5)*pi*x)]   % X er designmatrisen

beta = X'*X \ X'*y                      % gir minste kvadraters losning

t =linspace(-2, 6);
f = beta(1)+ beta(2).*sin((0.5)*pi*t) + beta(3).*cos((0.5)*pi*t);

figure(6)
plot(x, y,'or')
hold on
plot(t,f,'b')

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Eksempel 7 -- Multippel lineaer regresjon

%u, v, w  gir koordinatene til 5  punkter i rommet:

u=[0; 1; 2; 3; 4]; v=[1; 0; 1; 2; 2]; w=[3; 2; 0; 1; -1];
m = 5;                  % antall punkter

X = [ones(m,1) u  v]     % X er designmatrisen

beta = X'*X \ X'*w

% z = beta(1) + beta(2)*x + beta(3)*y  vil da gi likningen for planet
% som best approksimerer de gitte punktene
```

Seksjon 6.8

En kompakt form for å løse normallikningene ved vektet minstekvadraters metode ser slik ut:

```
(W*X)'*W*X \ (W*X)'*W*y
```

Her er W vektmatrisen, X designmatrisen, og y observasjonsvektoren, i henhold til notasjonen som brukes i avsnitt 6.6.

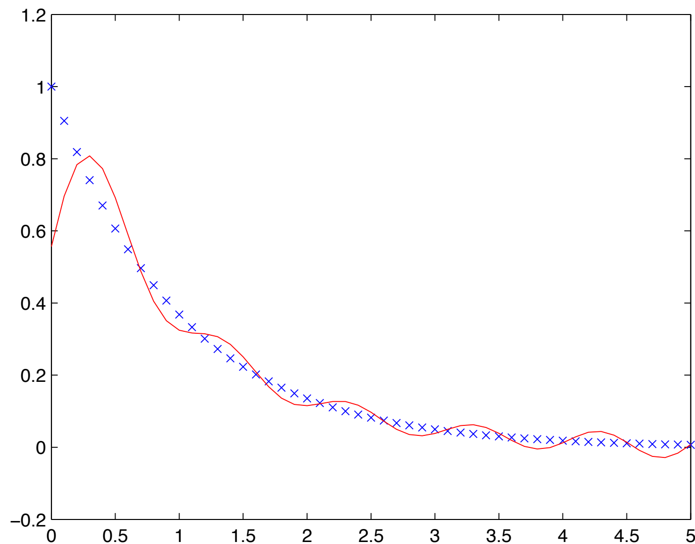
La oss si at vi ønsker å finne den vektede minstekvadraters linje for noen målepunkter, f.eks. punktene $(2,3), (3,2), (5,1), (6,0)$, og la oss anta at det andre og tredje punktet ansees som dobbelt så sikre som det første og det siste. Koden for å regne ut minstekvadraters linje, vektet minstekvadraters linje, og plotte disse sammen med punktene, blir som følger.

```
W = diag([ 1 2 2 1 ]);
A = [ 1 2 ; 1 3 ; 1 5 ; 1 6 ];
y = [ 3 2 1 0 ]';

utenvekting = A'*A \ A'*y;
medvekting = (W*A)'*W*A \ (W*A)'*W*y;

plot(A(:,2),y,'kx')     % Plotter punktene
hold on

t=linspace(0,7,100);
plot(t,utenvekting(1)+utenvekting(2)*t,'g'); % Uten vekting
plot(t,medvekting(1)+medvekting(2)*t,'b');   % Med vekting
legend('punkter','Uten vekting','Med vekting')
```

Pass på her å observere hvordan den større usikkerheten i første og fjerde punkt gir utslag i grafen for den vektede minstekvadraters linjen.

I dette avsnittet har vi også sett hvordan Fourier approksimasjoner til funksjoner er definert. I praksis er funksjonene som regel gitt ved et sett datapunkter, slik at vi bare kan regne ut tilnærmede verdier av integralene som gir oss Fourier koeffisientene. Her er et eksempel på hvordan en Fourier approksimasjon til en funksjon kan beregnes når dette er tilfelle. Vi betrakter her funksjonen $f(t) = e^{-t}$ på intervallet $[0,5]$ og antar at $f(t) = 0$ når $t \in (5, 2\pi])$. Vi tenker at f er et signal (og t angir tiden i sekunder) og at vi bare kjenner signalets verdier for hvert tiendedels sekund i intervallet $[0,5]$.

```
t=0:0.1:5; f = exp(-t);
N = length(t);          % antall komp. i vektoren t
kmax = 6;               % antall ledd (frekvenser) i approksimasjonen

% Beregner Fourier koeffisientene opptil kmax
% tilnaermer da integralene fra formel (7) i avsnitt 6.8

x=0:0.1:4.9; y=exp(-x); % 0.1 er bredden paa partisjonen
```



Figur 9: Plottet generert av koden for å regne ut en Fourier approksimasjon av $exp(-t)$

```
A0 = sum(y)*0.1/(2*pi);
for k=1:kmax
  A(k) = sum(y.*cos(k*x))*0.1/pi;
  B(k) = sum(y.*sin(k*x))*0.1/pi;
end

% Beregner Fourier approksimasjonen av orden kmax
for n=1:N
  fnew(n) = A0 + ...
      sum(A(1:kmax).*cos((1:kmax)*t(n))+B(1:kmax).*sin((1:kmax)*t(n)));
end

plot(t,f,'x')    % Plotter datapunktene
hold on
plot(t,fnew,'r') % Plotter Fourier approksimasjonen (rekonstruksjonen)
```

Approksimasjonen som fremkommer ved å kjøre koden ovenfor er tegnet opp i Figur 9.

Man kan beregne bedre tilnærminger av Fourier koeffisientene enn det vi har gjort her. Man f.eks. bruke Simpsons formel for å finne en mer nøyaktig approksimasjon av integralene som inngår. Ellers anbefales det å eksperimentere med forskjellige verdier for `kmax` (som bestemmer antall ledd i Fourier-approksimasjonen). Test også gjerne andre funksjoner, for eksempel $f(t) = t$.

Kapittel 7

Seksjon 7.2

For å tegne grafen til noen kvadratiske former som er positiv definitt, negativ definitt, og indefinitt, bruker vi noen kommandoer som bl.a. er omtalt i Matlab-heftet for MAT1110. Den positiv definite formen $z = 3x^2 + 7y^2$ kan vi plotte slik:

```
r = -1:0.05:1;

s = -1:0.05:1;

[x,y] = meshgrid(r,s);

z = 3*x.^2 + 7*y.^2;

mesh(x,y,z)
```

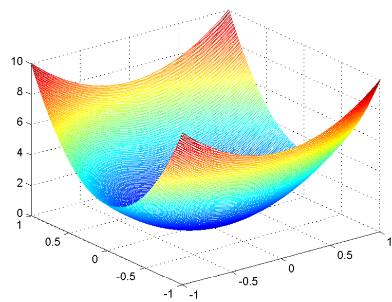
Plottet er vist i Figur 10. Der vi også har tatt med den negativ definite formen $z = -3x^2 - 7y^2$, og den indefinite formen $z = 3x^2 - 7y^2$. For å plotte disse trenger du kun å forandre den nest siste linjen i koden over.

For å plotte løsningsmengden til $3x^2 + 7y^2 = 3$ kan vi erstatte siste linje i koden ovenfor med

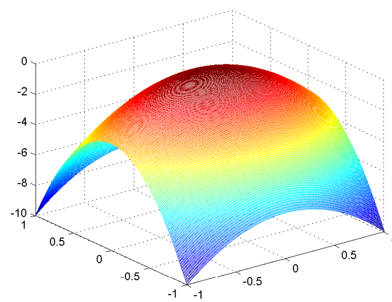
```
contour(x,y,z,[3 3])
```

Vektoren `[3 3]` ovenfor indikerer at det bare skal plottes en nivåkurve, nemlig den for $z = 3$. Det tilsvarende plottet finner du i Figur 11.

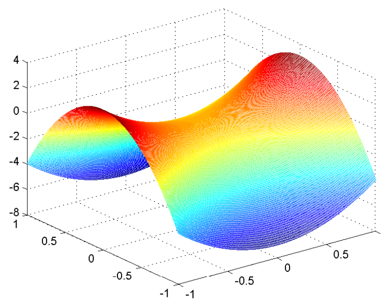
Siste parameteren til `contour` kan brukes til å spesifisere hvor mange nivåkurver det skal plottes. Hvis man utelater den, vil antall nivåkurver som plottes bli valgt automatisk. Nivåkurvene kan hjelpe oss med å avgjøre om formen er indefinitt eller ikke. I Figur 12 har vi plottet nivåkurvene til den positiv definite formen og til den indefinite formen som vi betraktet ovenfor. Som vi ser er nivåkurvene til den positive definite formen ellipser, mens de er hyperbler for den indefinite formen.



(a) Den positiv definite formen $z = 3x^2 + 7y^2$

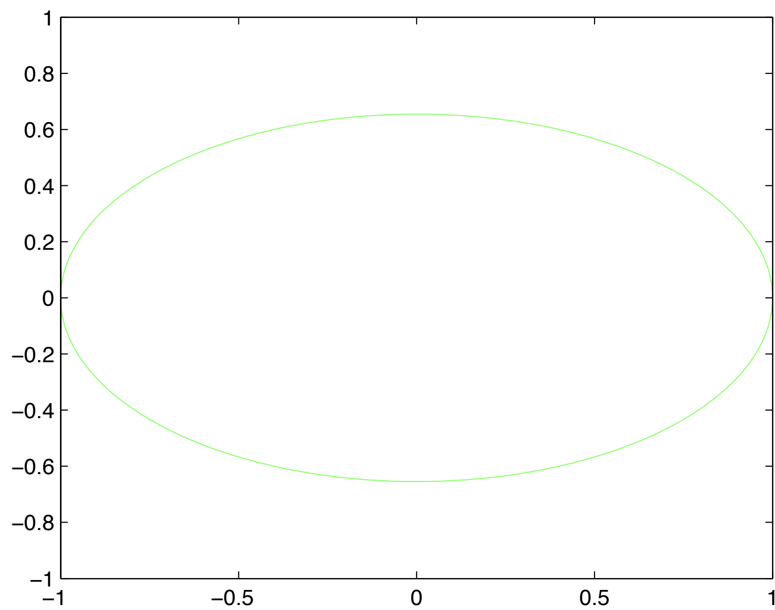


(b) Den neg. definite formen $z = -3x^2 - 7y^2$

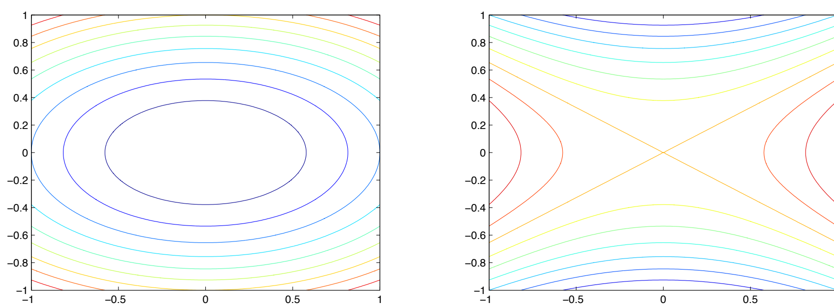


(c) Den indefinite formen $z = 3x^2 - 7y^2$

Figur 10: Plott av positiv definitt, negativ definitt, og indefinitt kvadratiske former



Figur 11: Plott av nivåkurven $3x^2 + 7y^2 = 3$



(a) Nivåkurver til den positiv definite formen $z = 3x^2 + 7y^2$

(b) Nivåkurver til den indefinite formen $z = 3x^2 - 7y^2$

Figur 12: Plott av nivåkurver for en positiv definitt og en indefinitt kvadratisk form

Seksjon 7.4

For å få Matlab til å beregne singulærverdidekomposisjonen til en matrise `A` kan vi bruke følgende kommando:

```
[U,Sigma,V] = svd(A)
```

Matrisene `U`, `Sigma` og `V` som returneres vil da være som angitt i Teorem 10 i avsnitt 7.4, (`Sigma` svarer til Σ). Skriver man bare `svd(A)` returneres singulærverdiene til `A` (i en vektor). Alternativt kan disse produseres ved å skrive `sqrt(eig(A'*A))`.

Man kan også regne ut singulærverdidekomposisjonen ved hjelp av resultatene i avsnitt 7.4. Anta f.eks. at `A` er en 4×4-matrise med singulærverdier $40, 20, 10, 0$. Vi kan da produsere singulærverdidekomposisjonen til `A` slik:

```
% Beregner o.n.basis for egenrommene til A'*A:

v1 = null(A'*A-1600*eye(4,4));
v2 = null(A'*A-400*eye(4,4));
v3 = null(A'*A-100*eye(4,4));
v4 = null(A'*A);

V = [ v1 v2 v3 v4 ] % blir en ortogonal matrise

U = [ (1/40)*A*v1 (1/20)*A*v2 (1/10)*A*v3 ]
% U har ortonormale kolonner, men er ikke kvadratisk !

u4 = null( U' ); % gir en vektor ortogonal paa kolonnene til U

U = [ U u4 ]     % utvider U til en ortogonal matrise

Sigma = diag([40,20,10,0])
```

Selv en ørliten perturbasjon av noen av koeffisientene i en matrise vil kunne forandre rangen til matrisen. Derimot vil forandringen i singulærverdiene være liten. Dette kan illustreres ved å kjøre følgende kode:

```
A = magic(4)    % Legg merke til at A(4,4) er lik 1

r=rank(A)
R=rref(A)
s=svd(A)       % gir singulaere verdiene

% gjoer en liten perturbasjon av A(4,4):

A(4,4) = 1.00000000001

r=rank(A)
R=rref(A)
s=svd(A)
```

Teorem 10 angir en oppskrift på hvordan U, V, Σ i singulærverdidekomposisjonen kan regnes ut. Alle stegene kan programmeres nokså direkte, men det hjelper å ta i bruk noen nye funksjoner. Vi bruker f.eks. funksjonen **find** til å finne singulærverdiene som er (litt) større enn 0.

En MATLAB-funksjon basert på oppskriften fra Teorem 10 blir som følger:

```
function [U,Sigma,V]=svdfact(A)

  [m,n] = size(A);
  [W,D] = eig(A'*A);

  singvals = sqrt(diag(D)); % gir vektor med singulaerverdiene

  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  % Lager forst V ved aa omordne paa kol. til W:

  [sortsingvals, sortcolW] = sort(singvals,'descend');

  % sorterer forst sing. verdiene til A i avtagende rekkefolge
  % og tar vare paa hvilke kolonnenummer i W som gir denne rekkefolgen

  V = W(:,sortcolW);

  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  % Lager Sigma:

  % kommandoen find(sortsingvals ~= 0) skulle teoretiskt gi hvilke komp.
  % i sortsingvals som er storre enn 0; men pga avrundingsfeil kan noen
  % av sing. verdiene som er lik 0 bli beregnet til noe ulik 0 !!!
  % Bruker derfor istedet:

  possingvals = find(sortsingvals >= 1.0e-06);

  r=length(possingvals); % r gir da antall sing.verdi. "ulik" 0,
                         % m.a.o. r er den "effektive" rangen til A

  Sigma = zeros(m,n);
  Sigma(1:r,1:r) = diag(sortsingvals(1:r));

  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  % Lager til slutt U:

  U_r = A*V(:,1:r)*inv(diag(sortsingvals(1:r)));
  U = gramschmidtutvid(U_r);     % dette er funksjonen vi laget i 6.4,
                                 % bruker da varianten med normalisering

  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Kjør `svdfact(A)` for noen `A` og sjekk at `U*Sigma*V'` blir (nesten) lik `A` !