Laboratorieøvelser i Matlab

av

Klara Hveberg

Senter for matematikk for anvendelser
(cma)
Universitetet i Oslo

Forord

I dette heftet finner du laboratorieøvelser i MATLAB til bruk sammen med Lindstrøm og Hvebergs "Flervariabel analyse med lineær algebra". Øvelsene er av varierende lengde og vanskelighetsgrad — noen av dem er ledede oppgavesekvenser, mens andre er rene programmeringsoppgaver som skal teste at du har fått med deg sentrale punkter i teorien. De fleste labøvelsene er delt inn i to eller tre deler: en laboratorie-/oppgavedel, en egen del med løsningsforslag og eventuelt en del som inneholder hint og halvferdige løsninger til oppgavene.

Labøvelsene er egnet for selvstudium, og det er derfor til en viss grad opp til deg selv hvor mye utbytte du får av dem. Du lærer absolutt mest av å prøve deg ordentlig på oppgavene på egen hånd før du kikker i løsningsforslagene. Øvelsene er imidlertid bygget opp sånn at du helst bør ha forstått en oppgave ordentlig før du går løs på neste. Er du usikker på hvordan en oppgave skal løses, kan det derfor være lurt å sjekke at du forstår løsningsforslaget til oppgaven før du går videre.

Labøvelsene forutsetter at du har arbeidet deg gjennom MATLAB-appendikset i "Flervariabel analyse med lineær algebra" på forhånd og kjenner til kommandoene vi gjennomgår der. Andre kommandoer du får bruk for underveis, blir gjennomgått i øvelsene. Etter innholdsfortegnelsen finner du en liten beskrivelse av hver labøvelse hvor det blir oppgitt hvilke matematiske begreper øvelsen dreier seg om, hvilke deler av MATLAB du trenger for å gjennomføre øvelsen, og hvor den naturlig passer inn i forhold til kapitlene i "Flervariabel analyse med lineær algebra".

Jeg har også laget noen enklere og mindre oppvarmingsøvelser som du gjerne kan gjøre mens du leser MATLAB-appendikset. I motsetning til de regulære labøvelsene, forutsetter ikke disse at du vet hvordan man lager m-filer og funksjoner i MATLAB. Du finner oppvarmingsøvelsene i appendiks 1 i dette heftet, rett etter de regulære labøvelsene.

Til noen av øvelsene foreligger det ferdige MATLAB-filer som du kan laste ned fra internettadressen

`http://www.math.uio.no/~klara/matlab/`

Blindern, 28. desember 2006

Klara Hveberg

Innholdsfortegnelse

Beskrivelse av laboratorieøvelsene

Nedenfor finner du kortfattede beskrivelser av hver enkelt labøvelse, med stikkord som antyder hvilke matematiske begreper øvelsen dreier seg om, hvilke deler av MATLAB du trenger for å gjennomføre øvelsen, og hvor den naturlig passer inn i forhold til kapitlene i "Flervariabel analyse med lineær algebra".

Lab 1: Lineæravbildninger og matriser

Innhold: I denne labøvelsen ser vi på sammenhengen mellom lineære transformasjoner og matriser. Vi bruker plottefunksjonen i MATLAB til å tegne opp ulike todimensjonale figurer og ser på hvordan vi kan transformere figurene ved å multiplisere plottepunktene med passende matriser. Til slutt bruker vi `for`-løkker til å lage små "filmsnutter" av figurene i bevegelse.

Begreper: matriserepresentasjon av lineæravbildninger, matrisemultiplikasjon, potenser av matriser.

Matlab: matriser, m-filer (funksjoner og skript), plotting, `for`-løkker.

Tilgjengelige m-filer som kan lastes ned: baat.m, monogramMH.m, klokkeskive.m, klokkepil.m og seilendebaat.m (Se `http://www.math.uio.no/~klara/matlab/`)

Kobling til læreboken: Denne labøvelsen passer godt sammen med seksjon 1.9 i læreboken.

Lab 2: Gauss-eliminasjon

Innhold: I denne labøvelsen ser vi nærmere på hvordan Gauss-eliminasjon kan beskrives på en systematisk måte som lar seg programmere i MATLAB. Vi lager først en MATLAB-funksjon som bringer en matrise over på trappeform ved å bruke Gauss-eliminasjon med delvis pivotering, og deretter utvider vi denne til en funksjon som bringer matrisen på redusert trappeform.

Begreper: elementære radoperasjoner, Gauss-eliminasjon.

Matlab: matriser, kolon-indeksering for rad- og søyleoperasjoner, m-filer, `for`-løkker og `while`-løkker, `if`-tester.

Kobling til læreboken: Gauss-eliminasjon behandles i seksjon 4.1–4.3 i læreboken. Labøvelsen krever at studentene har godt overblikk over metoden, og de bør derfor ha Gauss-eliminasjon i fingrene før de prøver seg på denne øvelsen.

Lab 3: Ligningssystemer og lineær uavhengighet

Innhold: I denne labøvelsen lager vi MATLAB-funksjoner som bruker den reduserte trappeformen til en matrise for å avgjøre om et ligningssystem er konsistent, hvor mange løsninger det eventuelt har, og om søylevektorene i matrisen er lineært uavhengige eller ikke.

Begreper: lineær avhengighet og uavhengighet, konsistente og inkonsistente ligningssystemer.

Matlab: matriser, m-filer, `rref`-kommandoen, `for`-løkker og `while`-løkker, `if`-tester, boolske variabler.

Kobling til læreboken: Lineær uavhengighet behandles i seksjon 4.6 i læreboken. Studentene bør nok ha trent litt på oppgaver om lineær uavhengighet før de prøver seg på denne labøvelsen.

Lab 4: Newtons metode for flere variable

Innhold: I denne labøvelsen lager vi en MATLAB-funksjon som utfører Newtons metode på funksjoner av flere variable. Vi ser også på hva som skjer hvis vi bruker Newtons metode på funksjoner av en kompleks variabel, og hvordan dette leder til eksotiske figurer som kalles fraktaler.

Begreper: Newtons metode, Jacobi-matrise, invers matrise.

Matlab: matriser, m-filer, plotting, `for`-løkker, `if`-tester.

Kobling til læreboken: Newtons metode behandles i seksjon 5.6 i læreboken. Øvelsen er imidlertid selvinstruerende og kan brukes som en introduksjon til metoden. Det siste avsnittet bruker begrepet linearisering som innføres i seksjon 2.8.

Appendiks 1: Oppvarmingsøvelser

Øvelse 1: Matrisemultiplikasjon

Innhold: I denne øvelsen tar vi utgangspunkt i den komponentvise definisjonen av matrisemultiplikasjon, og ser på hvordan multiplikasjonen kan beskrives på alternative måter ved hjelp av radene eller søylene i matrisene.

Begreper: matrisemultiplikasjon, lineærkombinasjoner av vektorer.

Matlab: matriser, kolon-indeksering for rad- og søyleoperasjoner.

Kobling til læreboken: Matrisemultiplikasjon behandles i seksjon 1.5 i læreboken.

Øvelse 2: Egenskaper ved determinanter

Innhold: I denne øvelsen studerer vi egenskaper ved determinanter og hvordan disse påvirkes av elementære radoperasjoner på matrisen. Vi ser også på determinanten til øvre- og nedre-triangulære matriser, identitetsmatriser og permutasjonsmatriser.

Begreper: determinanter, øvre og nedre triangulære matriser, permutasjonsmatriser, identitetsmatriser.

Matlab: matriser, kolon-indeksering for rad- og søyleoperasjoner, `for`-løkker, tilfeldig genererte matriser, triangulære matriser.

Kobling til læreboken: Determinanter av størrelse to-ganger-to og tre-ganger-tre behandles i seksjon 1.8 i læreboken, der det også finnes en uformell beskrivelse av determinanter av høyere orden. En grundigere innføring i determinanter finnes i seksjon 4.9, men øvelsen kan godt brukes som en motivasjon for denne seksjonen.

Lab 1: Lineæravbildninger og matriser

av

Klara Hveberg

I denne laboratorieøvelsen skal vi se nærmere på sammenhengen mellom lineære avbildninger og matriser. Vi skal bruke plottefunksjonen i MATLAB til å tegne opp ulike todimensjonale figurer og se hvordan avbildningene virker på dem. I tillegg til å bli kjent med matriserepresentasjonen til enkle lineæravbildninger, vil du få trening i å lage skript og funksjoner i MATLAB (og bruke `for`-løkker). Flere av MATLAB-funksjonene i denne labøvelsen kan lastes ned fra adressen `http://www.math.uio.no/~klara/matlab/`.

Opptegning av enkle figurer

Hvis vi har to n-dimensjonale vektorer $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ og $\mathbf{y} = (y_1, y_2, \ldots, y_n)$ kan vi (som du sikkert husker) bruke MATLAB-kommandoen

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

til å lage en strektegning som starter i punktet (x_1, y_1) og forbinder dette punktet med punktet (x_2, y_2) ved en rett strek, tegner en ny rett strek fra (x_2, y_2) til (x_3, y_3) osv.

For å tegne opp et kvadrat, kan vi derfor angi koordinatene til de fire hjørnepunktene (i den rekkefølgen vi vil at MATLAB skal forbinde dem) og få MATLAB til å tegne inn rette streker fra ett hjørnepunkt til det neste.

La oss si at vi ønsker å få MATLAB til å tegne opp kvadratet med hjørner i punktene $(0, 0)$, $(0, 4)$, $(4, 4)$ og $(4, 0)$. Vi må da lage en vektor $\mathbf{x}$ som består av førstekomponentene til de fire hjørnepunktene og en vektor $\mathbf{y}$ som består av andrekomponentene (og for å få tegnet opp et fullstendig kvadrat, må vi avslutte med samme hjørnepunkt som vi startet i).

```
>> x=[0 0 4 4 0];
>> y=[0 4 4 0 0];
```

Vi kan nå tegne opp kvadratet i et kvadratisk aksesystem hvor skalaene er satt til [-10 10] i begge retninger ved kommandoene

```
>> plot(x,y)                % trekker linje mellom punktene
>> axis([-10 10 -10 10])
>> axis('square')
```

Når vi etter hvert skal tegne opp mer kompliserte figurer som inneholder flere plottepunkter (la oss si n stykker), og studere hvordan ulike lineæravbildninger virker på disse figurene, er det mer oversiktlig å kunne angi plottepunktene i en $(2 \times n)$-matrise hvor første rad består av førstekoordinatene til punktene og andre rad består av andrekoordinatene. Hver søyle i matrisen skal altså representere et plottepunkt, og matrisen til kvadratet ovenfor vil altså være (2×5)-matrisen

```
>> K = [0 0 4 4 0
        0 4 4 0 0]
```

For å få MATLAB til å plotte kvadratet gir vi nå kommandoene

```
>> x=K(1,:);              % x-vektor skal være første rad i matrisen K
>> y=K(2,:);              % y-vektor skal være andre rad i matrisen K
>> plot(x,y)              % trekker linje mellom punktene
>> axis([-10 10 -10 10])
>> axis('square')
```

For å slippe å gjenta disse plotte-kommandoene hver gang vi skal tegne en ny figur, vil vi lage en MATLAB-funksjon `tegn` som kan kalles med en hvilken som helst $(2 \times n)$-matrise X som inn-parameter, og som tegner opp den tilsvarende figuren. På m-filen `tegn.m` legger vi følgende MATLAB-kode:

```
function tegn(X)
% Denne funksjonen tar som inn-parameter en matrise
% X med 2 rader og et vilkårlig antall søyler.
% Funksjonen tegner hver søyle som et punkt i
% planet og forbinder punktene etter tur med linjestykker.
% Skalaen er satt til [-10, 10] i begge retninger.
x = X(1,:);
y = X(2,:);
plot(x, y, 'b-')        % trekker blå linje mellom punktene
axis([-10 10 -10 10])
axis('square')
```

Etter å ha definert matrisen K som inneholder plottepunktene, kan vi nå få tegnet opp kvadratet med kommandoen

```
>> tegn(K)
```

Istedenfor å definere matrisen K direkte i kommandovinduet, kan vi også lage en liten MATLAB-funksjon `kvadrat` som returnerer matrisen som inneholder plottepunktene for kvadratet. Da legger vi følgende kode på filen `kvadrat.m`:

```
function X = kvadrat
% Returnerer en matrise X med data for tegning av et kvadrat
X = [0 0 4 4 0
     0 4 4 0 0];
```

Vi kan nå bruke funksjonen `kvadrat` som inn-parameter til funksjonen `tegn`, og få MATLAB til å tegne kvadratet ved å gi kommandoen

```
>> tegn(kvadrat)
```

Det kan kanskje virke unødvendig omstendelig å definere slike MATLAB-funksjoner bare for å tegne opp et kvadrat, men fordelen er at vi nå lett kan studere hvordan multiplikasjon med ulike (2×2)-matriser A virker inn på kvadratet. Siden funksjonen `kvadrat` returnerer en (2×5)-matrise, kan vi utføre matrisemultiplikasjonen `A*kvadrat` for en vilkårlig (2×2)-matrise A og få en ny (2×5)-matrise som inneholder "bildene" av plottepunktene i `kvadrat` etter lineærtransformasjonen representert ved matrisen A. For å få MATLAB til å tegne opp et bilde av de transformerte punktene, kan vi altså sende med (2×5)-matrisen `A*kvadrat` som inn-parameter til funksjonen `tegn`. Vi tester ut dette for matrisen

```
>> A=[-2 0
       0 1]
>> tegn(A*kvadrat)
```

Vi ser nå at multiplikasjon med matrisen A har forvandlet kvadratet til et rektangel, og flyttet det mot venstre. Det er imidlertid vanskelig å se nøyaktig hvordan dette har skjedd uten å kunne følge med på hvor de enkelte hjørnepunktene er flyttet. Vi skal derfor lage en alternativ tegne-funksjon som også markerer to fritt valgte plottepunkter i forskjellige farger. Når vi allerede har laget en m-fil `tegn.m`, er det lett å føye til det lille som trengs for å gjøre dette, og lagre koden på en ny m-fil som vi kaller `tegnmedpunkter.m`.

```
function tegnmedpunkter(X,m,n)
% Denne funksjonen fungerer på samme måte som funksjonen tegn,
% bortsett fra at den i tillegg tar to tall m og n som inn-parametre,
% og markerer plottepunkt nr. m og n med hhv rød og grønn sirkel
x = X(1,:);
y = X(2,:);
plot(x,y,'b-',x(m),y(m),'or',x(n),y(n),'og')
axis([-10 10 -10 10])
axis('square')
% Plottekommandoen ovenfor har samme virkning som kommandoene:
% plot(x, y, 'b-')     % trekker blå linje mellom punktene
% hold on              % neste plott kommer nå i samme figur
% plot(x(m),y(m),'or') % markerer punkt nr m med rød sirkel
% plot(x(n),y(n),'og') % markerer punkt nr n med grønn sirkel
% hold off             % ingen flere plott skal inn i samme figur
```

Vi kan nå få markert hjørnene $(0, 4)$ og $(4, 0)$ (plottepunkt nr 2 og nr 4) med henholdsvis en rød og en grønn sirkel ved å gi kommandoen

```
>> tegnmedpunkter(kvadrat,2,4)
```

For å undersøke hva som egentlig skjer med kvadratet når vi multipliserer med matrisen A, gir vi kommandoen

```
>> tegnmedpunkter(A*kvadrat,2,4)
```

Vi ser nå at kvadratet er blitt til et rektangel ved at det er speilet om y-aksen og strukket med en faktor 2 i x-retningen. For å få bekreftet hva som har skjedd med de opprinnelige hjørnepunktene i kvadratet, kan vi se på plottepunktene i den nye matrisen

```
>> A*kvadrat
```

Ved å sammenligne med de opprinnelige plottepunktene fra funksjonen `kvadrat`, ser vi at multiplikasjon med matrisen A har ført til at alle x-koordinatene har skiftet fortegn og blitt dobbelt så store, mens y-koordinatene er de samme som før. Men dette tilsvarer jo nettopp at alle punktene er speilet om y-aksen og flyttet dobbelt så langt i x-retningen.

Oppgave 1

Lag m-filene `kvadrat.m` og `tegn.m` som beskrevet ovenfor (hvis du ikke allerede har gjort det). Bruk kommandoen `tegn(A*kvadrat)` til å studere hva som skjer med kvadratet når matrisen A er henholdsvis

a) $A = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}$ d) $A = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$

b) $A = \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix}$ e) $A = \begin{pmatrix} 1 & 0.5 \\ 0 & 1 \end{pmatrix}$

c) $A = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$ f) $A = \begin{pmatrix} \sqrt{2}/2 & -\sqrt{2}/2 \\ \sqrt{2}/2 & \sqrt{2}/2 \end{pmatrix}$

Beskriv hva som skjer med kvadratet i hvert enkelt tilfelle (du kan eventuelt bruke funksjonen `tegnmedpunkt` hvis du synes det er vanskelig å se hva som skjer), og forsøk å forklare hvorfor multiplikasjon med matrisen gir denne virkningen.

Skaleringsmatriser

Vi skal nå bruke metoden fra forrige avsnitt til å tegne opp en litt mer komplisert figur. På m-fila `hus.m` legger vi følgende plottepunkter for opptegning av et hus (tegn inn punktene på et papir først for å være sikker på at du forstår hvordan opptegningen foregår):

```
function X = hus
% Lager en matrise X som inneholder data for tegning av et hus.
% Kan f.eks kalles med tegn(A*hus) for ulike 2*2-matriser A
X = [0  -7  -6  -6   3   3   0   0   6   6   7   0
     8   1   2  -8  -8  -3  -3  -8  -8   2   1   8];
```

Vi får nå MATLAB til å tegne opp huset ved å gi kommandoen

```
>> tegn(hus)
```

Oppgave 2

Finn en matrise slik at mulitiplikasjon med matrisen

a) gjør huset halvparten så stort

b) gjør huset halvparten så bredt

c) gjør huset halvparten så høyt

Tegn i hvert tilfelle opp det nye huset ved å bruke kommandoen

```
>> tegn(A*hus)
```

Speilingsmatriser

På fila `baat.m` legger vi følgende data for opptegning av en seilbåt (igjen bør du starte med å tegne inn punktene på et papir så du skjønner hvordan figuren fremkommer).

```
function X = baat
% Lager en matrise X som inneholder data for tegning av en båt.
% Kan f.eks kalles med tegn(A*baat) for ulike 2*2matriser A
X = [ 0   7   4  -4  -7   0   0   2   0   5   0
     -4  -4  -7  -7  -4  -4   8   6   6  -2  -2];
```

På samme måte som tidligere kan vi tegne opp båten med kommandoen

```
>> tegn(baat)
```

Oppgave 3

a) Finn en matrise A slik at mulitiplikasjon med A får båten til å seile i motsatt retning.

b) Finn en speilingsmatrise A som får båten til å kullseile (dvs vipper båten oppned).

Tegn opp resultatene ved hjelp av kommandoen

```
>> tegn(A*baat)
```

Speiling, rotasjoner og sammensetning av avbildninger

Vi skal nå leke litt med sammensetning av speilinger og rotasjoner for å lage ulike monogrammer av bokstavene M og H. Vi lager først en MATLAB-funksjon på fila `monogramMH.m` som returnerer en matrise med plottepunktene for å tegne opp monogrammet MH.

```
function X = monogramMH
% Lager en matrise X som inneholder data for tegning av monogrammet MH.
% Kan f.eks kalles med tegn(A*monogram) for ulike 2*2-matriser A
M = [ 0 -1 -1.7 -3.5 -5.3 -6 -7 -6 -5 -3.5 -2 -1  0
     -3 -3  1.5 -1.2  1.5 -3 -3  3  3  0.5  3  3 -3];
H = [ 0 0 1 1   3   3 4  4  3  3    1    1  0
     -3 3 3 0.5 0.5 3 3 -3 -3 -0.5 -0.5 -3 -3];
X=[M,H];       % Skjøter sammen de to matrisene med M etterfulgt av H
% NB! Har egentlig med punktet (0,-3) en gang mer enn nødvendig (både
% i slutten av M og i starten av H) når vi skjøter sammen bokstavene,
% men det er lettere å se hvordan bokstavene M og H fremkommer på
% denne måten.
```

Vi får nå tegnet opp monogrammet MH ved å gi kommandoen

```
>> tegn(monogramMH)
```

Som i de foregående seksjonene skal vi visualisere hva som skjer med plottepunktene når vi multipliserer med ulike (2×2)-matriser A. La oss si at vi ønsker at monogrammet MH skal skrives med skråstilte bokstaver (*kursiv*). For at bokstavene skal helle litt mot høyre, må vi bruke en skjær-matrise (shear matrix) som adderer et lite tillegg til x-komponenten, og dette tillegget bør være proporsjonalt med y-komponenten til punktet (slik at det blir mer forskjøvet mot høyre jo høyere opp i bokstaven punktet ligger). Vi kan for eksempel bruke skjær-matrisen

```
>> A=[1  0.2
      0  1 ]
```

og sjekke resultatet ved kommandoen

```
>> tegn(A*monogramMH)
```

Ønsker vi isteden å gjøre bokstavene smalere ved å krympe dem litt i x-retningen, må vi bruke en skaleringsmatrise som multipliserer alle x-koordinatene med en faktor som er mindre enn 1. Vi kan for eksempel bruke matrisen

```
>> B=[0.8  0
      0    1]
```

og sjekke resultatet ved å skrive

```
>> tegn(B*monogramMH)
```

De to foregående typene matriser har vi vært borti tidligere (i oppgave 1), men hva om vi ønsker å finne en matrise som både skråstiller bokstavene og samtidig gjør dem smalere? Hvis vi tenker oss at dette gjøres i to trinn, kan vi først lage plottematrisen for det kursiverte monogrammet ved å utføre operasjonen

```
>> A*monogramMH
```

og deretter gjøre det kursiverte monogrammet smalere ved operasjonen

```
>> B*(A*monogramMH)
```

Vi tegner opp resultatet ved å gi kommandoen

```
>> tegn(B*(A*monogramMH))
```

La oss se hva som skjer hvis vi isteden danner produktmatrisen

```
>> C=B*A
```

og anvender denne på plottepunktene til monogrammet:

```
>> tegn(C*monogramMH)
```

Bemerkning: Dette tilsvarer altså å gi kommandoen

```
>> tegn((B*A)*monogramMH)
```

Vi får akkurat samme resultat som i stad, og ser altså at sammensetning av avbildninger svarer til matrisemultiplikasjon av matrisene som representerer operasjonene.

Oppgave 4

Ta utgangspunkt i monogrammet MH. I hvert av punktene a)–c) nedenfor skal du finne en matrise A (speiling eller rotasjon) slik at mulitiplikasjon med A omdanner monogrammet MH til monogrammet

a) HM

b) WH

c) HW

Sjekk resultatet i hvert tilfelle ved å bruke kommandoen

```
>> tegn(A*monogramMH)
```

d) Lag tre små MATLAB-funksjoner på filene `speilomx.m`, `speilomy.m` og `roter180.m` som returnerer de speilings- og rotasjonsmatrisene du fant i punkt a)–c). Disse funksjonene kan du benytte i resten av oppgaven.

e) I hvert av tilfellene nedenfor skal du finne en speiling eller rotasjon som

- avbilder MH på HM
- avbilder HM videre på WH
- avbilder WH videre på HW

Her kan det være nyttig å spare på resultatet fra hvert trinn i en matrise som du så kan arbeide videre med i neste trinn. Hvis du i første trinn bruker en matrise A (som returneres av en av de tre funksjonene du laget ovenfor), bør du altså bruke kommandoer av typen

```
>> monogramHM = A * monogramMH
>> tegn(monogramHM)
```

Og i neste trinn

```
>> monogramWH = B * monogramHM
>> tegn(monogramWH)
```

osv...

f) Forklar hvordan resultatene i punkt b) og e) gir oss to måter å avbilde monogrammet MH på monogrammet WH. Bruk disse observasjonene til å finne en sammenheng mellom speiling om x-aksen i forhold til speiling om y-aksen etterfulgt av en rotasjon på 180 grader.

g) Sammenlign matrisen fra funksjonen `speilomx` med matrisen du får ved å bruke kommandoen

```
>> speilomy*roter180
```

Kan du forklare hvordan dette henger sammen med (og underbygger) observasjonen fra punkt f)?

h) Finner du flere sammensetninger som er like?

Mer om sammensetning og generelle rotasjonsmatriser

I denne seksjonen skal vi se nærmere på mer generelle rotasjonsmatriser og sammensetninger av slike. Vi skal bruke MATLAB til å illustrere disse rotasjonene i form av bevegelser til timeviseren på en klokke, og starter derfor med å lage følgende skript på fila `klokkeskive.m` for å tegne opp en klokkeskive med timemarkører:

```
% Skript: klokkeskive
% Tegner opp en sirkulær klokkeskive med tall som markerer timene
t=0:0.05:2*pi;
x=6*cos(t);
y=6*sin(t);
plot(x,y)                       % tegner sirkelskive med radius 6
axis([-10 10 -10 10])
axis('square')
h=0.3;                          % for å høyrejustere posisjon av tall
                                % pga tekstboksens plassering
text(5*cos(pi/2)-h,5*sin(pi/2),'12')  % setter tall som timemarkører
text(5*cos(pi/3),5*sin(pi/3),'1')
text(5*cos(pi/6),5*sin(pi/6),'2')
text(5*cos(0),5*sin(0),'3')
text(5*cos(-pi/6),5*sin(-pi/6),'4')
text(5*cos(-pi/3),5*sin(-pi/3),'5')
text(5*cos(-pi/2)-h,5*sin(-pi/2),'6')
text(5*cos(4*pi/3),5*sin(4*pi/3),'7')
text(5*cos(7*pi/6),5*sin(7*pi/6),'8')
text(5*cos(pi),5*sin(pi),'9')
text(5*cos(5*pi/6)-h,5*sin(5*pi/6),'10')
text(5*cos(2*pi/3)-h,5*sin(2*pi/3),'11')
title('Klokke')
```

```
% Vi kunne også ha brukt en for-løkke til å plassere timemarkørene:
% for t=1:12
%   text(5*cos(pi/2-t*pi/6)-h,5*sin(pi/2-t*pi/6),int2str(t))
% end
```

Vi får nå tegnet opp klokkeskiven med kommandoen

```
>> klokkeskive
```

Det neste vi trenger er en timeviser. På samme måte som i de foregående seksjonene kan vi få MATLAB til å tegne en pil som starter i origo og peker rett opp (mot kl 12) ved å spesifisere plottepunktene i en matrise, og sende denne som inn-parameter til funksjonen `tegn` (som vi laget i oppgave 1). På fila `klokkepil.m` lager vi derfor en funksjon `klokkepil` som returnerer matrisen med plottepunktene for timeviseren.

```
function X = klokkepil
% Lager en matrise som inneholder data for tegning av en pil med
% lengde 4 som starter i origo og peker rett opp (markerer kl 12).
X = [-0.2  -0.2  -0.4   0     0.4   0.2   0.2  -0.2
       0     3     3     3.8   3      3     0    0 ];
```

Vi kan nå få MATLAB til å tegne en klokke der timeviseren peker på 12, ved å gi kommandoene

```
>> klokkeskive
>> hold on              % sørger for at timeviseren kommer i samme figur
>> tegn(klokkepil)
>> hold off             % ingen flere plott skal i samme figur
```

Ideen er nå at vi kan få timeviseren til å peke på ulike klokkeslett ved hjelp av en rotasjonsmatrise A og kommandoen

```
>> tegn(A*klokkepil)
```

La oss si at vi ønsker at timeviseren skal peke på tallet 3. Det betyr at vi må dreie den 90 grader med klokka (dvs en vinkel på $-\pi/2$ radianer). For å finne rotasjonsmatrisen til en dreining på $-\pi/2$ radianer, er det nok å se på hva basisvektorene $\mathbf{e}_1 = (1,0)$ og $\mathbf{e}_2 = (0,1)$ avbildes på, og bruke bildene av disse vektorene som søyler i rotasjonsmatrisen. Dreier vi vektoren $\mathbf{e}_1 = (1,0)$ en vinkel på $-\pi/2$ radianer, havner den på vektoren $(0,-1)$. Dette er altså første søyle i rotasjonsmatrisen. Tilsvarende havner vektoren $\mathbf{e}_2 = (0,1)$ på vektoren $(1,0)$ når vi dreier den en vinkel på $-\pi/2$ radianer, så andre søyle i rotasjonsmatrisen er $(1,0)$. Det betyr at rotasjonsmatrisen for vinkelen $-\pi/2$ er

```
>> A = [ 0  1
        -1  0]
```

Vi kan sjekke at vi har funnet den riktige matrisen ved å bruke kommandoen

```
>> tegn(A*klokkepil)
```

for å se at vi nå får en pil som peker rett mot høyre.

Generelt er det gjerne litt mer arbeid å finne rotasjonsmatrisen til en vinkel, selv om du kan bruke samme fremgangsmåte som ovenfor (men for å finne koordinatene til billedvektorene må du benytte cosinus og sinus til rotasjonsvinkelen).

Oppgave 5

a) Finn matriser A og B slik at mulitiplikasjon med

i) matrisen A dreier viseren en time frem

ii) matrisen B dreier viseren to timer frem

Her skal du selv beregne rotasjonsmatrisene til de aktuelle vinklene ved hjelp av metoden beskrevet ovenfor (se på bildet av basisvektorene under den aktuelle dreiningen).

Sjekk resultatene ved å bruke kommandoene

```
>> klokkeskive
>> hold on
>> tegn(A*klokkepil)          % Bytt ut A med B i punkt ii)
>> hold off
```

b) Forklar hvordan du istedenfor å beregne rotasjonsmatrisen B direkte slik du gjorde ovenfor, kan benytte (en potens av) matrisen A til å dreie viseren to timer frem.

c) Lag en funksjon `roter` som tar en vinkel v (i radianer) som inn-parameter og returnerer rotasjonsmatrisen til vinkelen v. På filen `roter.m` legger du altså en funksjon som starter med

```
function X = roter(v)
```

Test funksjonen ved å bruke den til å tegne opp en klokke hvor viseren peker på ett og en klokke hvor viseren peker på fem. Til dette kan du bruke kommandoen

```
>> tegn(roter(t*(-pi)/6)*klokkepil)
```

for passende verdier av t.

d) Forklar hvordan du isteden kan dreie timeviseren t timer frem ved hjelp av rotasjonsmatrisen A fra punkt a) og utfør dette for de to klokkeslettene i punkt c). Hvilken av de to metodene krever flest regneoperasjoner?

Filmer ved hjelp av for-løkker

Vi skal avslutte denne laboratorieøvelsen med å bruke funksjonene vi allerede har laget til å simulere en klokke som tikker og går. Dette gir en enkel illustrasjon av hvordan vi kan bruke `for`-løkker i MATLAB.

For å illustrere ideen skal vi først se et eksempel på en `for`-løkke som får båten fra oppgave 2 til å seile av sted over figurvinduet fra høyre mot venstre. Båtens forflytning er riktignok ikke en lineæravbildning i planet og kan derfor ikke uttrykkes ved multiplikasjon med en 2×2-matrise (se bemerkningen helt til slutt i dette avsnittet), men eksemplet illustrerer hvordan vi kan bruke `for`-løkker til å generere bilder i bevegelse. På filen `seilendebaat.m` legger vi følgende skript:

```
% Skript: seilendebaat
% Viser en båt som seiler av sted fra høyre billedkant mot venstre.
% Henter inn plottepunktene til båten fra matrisen generert av
% skriptet baat.m
% Tegner opp alt i et aksesystem med skala [-10 10] i begge retninger
% ved hjelp av funksjonen tegn.m
B=baat;                    % leser inn plottepunktene fra baat.m
```

```
X=1/4*[B(1,:)+32; B(2,:)];  % krymper båten og plasserer den i høyre
                            % billedkant
tegn(X);                    % tegner det første bildet av båten
for t=1:160
   X=[X(1,:)-0.1; X(2,:)];  % flytter båten skrittvis mot venstre
   tegn(X);                 % tegner båten i den nye posisjonen
   pause(0.05)              % venter 0.05 sek før neste bilde tegnes
end
```

Denne "filmsnutten" kan nå kjøres ved å gi kommandoen

```
>> seilendebaat
```

Oppgave 6

Bruk fremgangsmåten ovenfor til å lage et skript på fila `tikkendeklokke.m` som tegner opp en en klokke som går en runde med timeviseren. Du kan f.eks la klokka gå med en fart på 1 time per sekund ved å benytte kommandoen

```
>> pause(1)
```

som venter ett sekund før neste bilde tegnes.

Bemerkning om translasjoner for spesielt interesserte

Som vi var inne på da vi skulle generere filmsnutten med den seilende båten, er en translasjon (forflytning i x-retningen og/eller y-retningen) i planet ikke en lineærtransformasjon (alle lineære avbildninger sender origo på origo) og kan derfor ikke uttrykkes ved multiplikasjon med en (2×2)-matrise. Det finnes imidlertid et nyttig triks som gjør at vi kan uttrykke slike translasjoner ved hjelp av matriser likevel. Ved å gå opp en dimensjon og benytte (3×3)-matriser, kan vi finne en matrise som fikser jobben. Vi benytter det som kalles homogene koordinater, som grovt sagt betyr at vi istedenfor å se på punktene (x, y) i xy-planet, løfter planet ett hakk opp og ser på de tilsvarende punktene $(x, y, 1)$ i det parallelle planet $z = 1$. Enhver lineæravbildning av punktet (x, y) i det opprinnelige planet kunne oppnås ved multiplikasjon med en (2×2)-matrise `A=[a b; c d]`, og den tilsvarende avbildningen av det korresponderende punktet $(x, y, 1)$ i det løftede planet $z = 1$ oppnås nå ved multiplikasjon med (3×3)-matrisen `B=[a b 0; c d 0; 0 0 1]`. Men ved å gå opp en dimensjon har vi i tillegg fått mulighet til å translatere punktet $(x, y, 1)$: For å flytte punktet h enheter bortover i x-retningen, multipliserer vi bare med matrisen `X=[1 0 h; 0 1 0; 0 0 1]`, og for å flytte punktet k enheter bortover i y-retningen, multipliserer vi med matrisen `Y=[1 0 0; 0 1 k; 0 0 1]`.

Litt om animasjoner

Hvis du ønsker å lage en filmsnutt som skal spilles av flere ganger, kan det være lurt å lage en animasjon. Dette gjøres ved at vi etter hvert plott lagrer plottet i en struktur F ved kommandoen

```
>> F(t)=getframe;                  % plott nummer t lagres i F(t)
```

Etterpå kan vi da spille av animasjonen så mange ganger vi vil med kommandoen

```
>> movie(F,k,m)
```

hvor k angir angir antall ganger filmen skal kjøres, og m angir antall figurer som skal vises per sekund.

For å lage en animasjon av den seilende båten i det forrige eksemplet, bruker vi da kommandoene:

```
>> B=baat;
>> X=1/4*[B(1,:)+32; B(2,:)];
>> for t=1:160
      X=[X(1,:)-0.1; X(2,:)];
      tegn(X)
      F(t)=getframe;              % lagrer hvert plott i strukturen F
   end
```

Vi kan nå spille av animasjonen i passe tempo med kommandoen

```
>> movie(F,1,20)
```

Oppgave 7 (ekstraoppgave)

Lag en animasjon av den tikkende klokken.

Lab 2: Gauss-eliminasjon

av

Klara Hveberg

I denne laboratorieøvelsen skal vi se på hvordan vi kan lage MATLAB-funksjoner som utfører *Gauss-eliminasjon* på matriser, dvs som bringer dem på trappeform ved hjelp av elementære radoperasjoner. I tillegg til at du må tenke gjennom hvordan Gauss-eliminasjon kan beskrives på en systematisk måte som kan programmeres, vil du få trening i å bruke MATLAB til å utføre radoperasjoner på en matrise, og i å lage funksjoner med `for`-løkker eller `while`-løkker.

Hvis du føler at du har god kontroll på Gauss-eliminasjon og ikke vil ha hjelp til programmeringen, kan du godt hoppe over innledningen og gå rett løs på oppgave 3 og 4, hvor du først skal lage en funksjon som bringer en vilkårlig matrise over på trappeform, og deretter en som bringer matrisen på redusert trappeform (uten å bruke den innebygde MATLAB-funksjonen `rref`).

Gauss-eliminasjon (enkel versjon uten ombytting av rader)

Vi husker at Gauss-eliminasjon bruker radvise operasjoner til å bringe en matrise over på såkalt trappeform, dvs en form hvor

i) det ledende elementet (dvs. det første elementet som ikke er null) i en rad alltid står lenger til høyre enn det ledende elementet i raden ovenfor.

ii) eventuelle nullrader er samlet i bunnen av matrisen

La oss starte med å repetere hvordan vi utfører Gauss-eliminasjon på en konkret matrise

$$A = \begin{pmatrix} 6 & -2 & 0 \\ 9 & -1 & 1 \\ 3 & 7 & 5 \end{pmatrix}$$

Vår første oppgave er å sørge for at vi har et ikke-trivielt pivot-element (altså et element forskjellig fra null) i posisjon $(1,1)$ i første søyle og skaffe oss nuller under dette elementet ved å trekke passende multiplum av første rad ifra radene nedenfor. Siden $A(1,1) = 6$, betyr det at vi må trekke $A(2,1)/A(1,1) = 9/6 = 3/2$ av første rad fra andre rad for å få null i posisjonen under pivot-elementet, så vi utfører kommandoene

```
>> k=A(2,1)/A(1,1);
>> A(2,:)=A(2,:) - k*A(1,:)
```

Deretter sørger vi for å få null i tredje rad ved å trekke $A(3,1)/A(1,1) = 3/6 = 1/2$ av første rad fra tredje rad

```
>> k=A(3,1)/A(1,1);
>> A(3,:)=A(3,:) - k*A(1,:)
```

Vi står nå igjen med matrisen

$$A = \begin{pmatrix} 6 & -2 & 0 \\ 0 & 2 & 1 \\ 0 & 8 & 5 \end{pmatrix}$$

og er altså ferdig med å skaffe oss nuller under pivot-elementet i første søyle. Vi går nå videre til andre søyle hvor pivot-elementet er $A(2,2) = 2$. For å skaffe oss nuller i

søylen under pivot-elementet, må vi nå bare trekke $A(3,2)/A(2,2) = 8/2 = 4$ ganger andre rad fra tredje rad

```
>> k=A(3,2)/A(2,2);
>> A(3,:)=A(3,:) - k*A(2,:)
```

Vi står nå igjen med matrisen

$$A = \begin{pmatrix} 6 & -2 & 0 \\ 0 & 2 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

og er altså ferdig med å bringe matrisen over på trappeform.

Algoritmen vi fulgte ovenfor kan oppsummeres i følgende programskisse:

```
FOR <hver søyle, bortsett fra den siste som godt kan droppes>
   FOR <hver rad nedenfor pivot-raden>
     <beregn hvilket multiplum av pivot-raden vi må trekke fra raden
      vi er i for å nulle ut elementet i søylen under pivot-posisjonen>
     <trekk fra det riktige multiplumet av pivot-raden>
   END
END
```

Oppgave 1

Lag et skript `gaussA` som utfører Gauss-eliminasjon på matrisen

$$A = \begin{pmatrix} 6 & -2 & 0 \\ 9 & -1 & 1 \\ 3 & 7 & 5 \end{pmatrix}$$

ved hjelp av `for`-løkker (bruk to `for`-løkker nøstet inni hverandre). MATLAB skal altså i hvert trinn selv regne ut de riktige multiplumene av pivotraden som skal trekkes fra radene nedenfor.

Oppgave 2

Bruk ideen fra skriptet i forrige oppgave til å lage en MATLAB-funksjon `gauss` som tar en matrise A som inn-parameter, utfører Gauss-eliminasjon og returnerer matrisen på trappeform. Du kan forutsette at matrisen A kan bringes på trappeform uten at du støter på null-elementer i pivot-posisjoner underveis. Det betyr at du finner pivot-elementene på hoveddiagonalen i posisjonene (j, j). Du kan også anta at matrisen er kvadratisk, hvis du synes det gjør oppgaven lettere. Du trenger ikke å lage en robust funksjon som sjekker at disse antagelsene er oppfylt for matrisen A. Kommandoen

```
>> [m,n]=size(A)
```

kan brukes til å lese inn dimensjonen til matrisen A.

Gauss-eliminasjon med ombytting av rader

I eksemplet i forrige avsnitt var vi "heldige" og unngikk å støte på null-elementer i pivot-posisjoner underveis i prosessen. Generelt må vi imidlertid regne med å støte på slike null-elementer, og må da lete etter et ikke-trivielt element lenger nede i samme søyle. Hvis vi finner et slikt ikke-trivielt element, flytter vi dette opp i pivot-posisjonen

ved å bytte om rader i matrisen. Hvis det bare står nuller under nullelementet i pivot-posisjonen, går vi videre til neste søyle, men begynner da å lete etter pivot-elementet i samme rad som vi lette i forrige søyle. Det betyr at pivot-posisjonen i søyle nr j ikke nødvendigvis lenger er posisjonen (j, j) på hoveddiagonalen (trappetrinnene har ikke nødvendigvis dybde 1), så vi må føre regnskap med radindeksen og søyleindeksen hver for seg.

La oss starte med å se på et eksempel hvor det dukker opp et null-element i en pivot-posisjon, og hvor alle elementene i søylen nedenfor er null. Vi ser på matrien

$$A = \begin{pmatrix} 0 & 0 & 7 \\ 5 & -5 & 10 \\ 1 & -1 & 6 \end{pmatrix}$$

Vår første oppgave er å prøve å skaffe et ikke-trivielt pivot-element (altså et element forskjellig fra null) i posisjon $(1, 1)$ i første søyle. Vi leter da nedover i første søyle etter et ikke-trivielt element, og finner dette i rad nr 2. Vi kan derfor skaffe oss et ikke-trivielt pivot-element ved å bytte om første og andre rad:

```
>> A([1 2],:)=A([2 1],:)
```

Matrisen vår er nå på formen

$$A = \begin{pmatrix} 5 & -5 & 10 \\ 0 & 0 & 7 \\ 1 & -1 & 6 \end{pmatrix}$$

og vi skaffer oss nuller under pivot-elementet i første søyle ved å trekke passende multiplum av første rad ifra radene nedenfor. Vi har allerede null i posisjonen rett under pivot-elementet, og trenger derfor ikke å gjøre noe med rad nr 2. For å få null i tredje rad, trekker vi $A(3, 1)/A(1, 1) = 1/5$ av første rad fra tredje rad

```
>> k=A(3,1)/A(1,1);
>> A(3,:)=A(3,:) - k*A(1,:)
```

Vi står nå igjen med matrisen

$$A = \begin{pmatrix} 5 & -5 & 10 \\ 0 & 0 & 7 \\ 0 & 0 & 4 \end{pmatrix}$$

og er altså ferdig med å skaffe oss nuller under pivot-elementet i første søyle. Vi går nå videre til andre søyle og leter etter et pivot-element i posisjon $(2, 2)$. Siden elementet $A(2, 2)$ er null, leter vi nedover i søylen etter et ikke-trivielt element å flytte opp. Men alle elementene nedenfor er også null, så vi har ikke noe å utrette i denne søylen. Vi går derfor videre til søyle nr 3 og leter denne gangen etter et pivot-element i posisjon $(2, 3)$ (vi befinner oss altså ikke lenger på diagonalen, men leter videre i samme rad som vi startet på i forrige søyle). Her finner vi et ikke-trivielt element $A(2, 3) = 7$, og nuller ut elementet nedenfor i samme søyle ved å trekke $A(3, 3)/A(2, 3) = 4/7$ ganger andre rad fra tredje rad

```
>> k=A(3,3)/A(2,3);
>> A(3,:)=A(3,:) - k*A(2,:)
```

Vi står nå igjen med matrisen

$$A = \begin{pmatrix} 5 & -5 & 10 \\ 0 & 0 & 7 \\ 0 & 0 & 0 \end{pmatrix}$$

og er altså ferdig med å bringe matrisen over på trappeform. Eksemplet illustrerer at vi gjennomløper søylene etter tur på samme måte som i forrige avsnitt, men at vi denne gangen også må holde styr på radene: vi skal øke radindeksen med 1 dersom pivot-elementet vi finner (etter eventuell ombytting av rader) er forskjellig fra null, men la radindeksen forbli den samme dersom vi bare får et null-element i pivot-posisjonen.

Vi skal nå se hvordan vi kan modifisere funksjonen `gauss` (fra oppgave 2) til å takle matriser hvor vi også kan støte på null-elementer i pivot-posisjoner og må bytte om rader underveis. Ideen er altså at hver gang vi støter på et slikt null-element $A(i,j) = 0$ i en pivot-posisjon, leter vi etter ikke-trivielle elementer nedenfor dette nullelementet i søyle nr j. Dersom vi finner et slikt ikke-trivielt element $A(p,j) \neq 0$, bytter vi om radene p og i, og fortsetter som i den opprinnelige funksjonen.

Ved vanlig Gauss-eliminasjon står vi fritt til å bytte om rad j og en hvilken som helst rad nedenfor som har et ikke-trivielt element i søyle nr j (ved programmering kan vi f.eks bestemme oss for å bytte med den første aktuelle raden vi finner). For å redusere avrundingsfeilene lønner det seg imidlertid å velge rad nr p på en slik måte at $A(p,j)$ er størst mulig i absoluttverdi. Dersom vi alltid velger p på denne måten (uansett om det opprinnelige elementet i pivot-posisjonen er null eller ikke) kalles prosedyren *Gauss-eliminasjon med delvis pivotering*. Denne prosedyren er enkel å implementere i MATLAB ved hjelp av en ferdig innebygd funksjon `max` som finner det største elementet i en søylevektor. Kommandoen

```
>> [maksverdi,p]=max(y)
```

returnerer det maksimale elementet `maksverdi` i søylevektoren y og nummeret p til raden hvor dette elementet befinner seg.

Gauss-eliminajon med delvis pivotering

Vi kan nå skissere en algoritme for å utføre Gauss-eliminasjon med delvis pivotering. For å legge oss nærmest mulig opp til den enkle algoritmen vi skisserte før oppgave 1, skal vi basere oss på `for`-løkker. Dette krever imidlertid at vi i visse situasjoner må avbryte en løkke ved hjelp av kommandoen `break`, og noen synes kanskje at dette er lite elegant programmering. I løsningsforslaget vil du derfor også finne et alternativ som baserer seg på en `while`-løkke.

Programskisse

```
FOR <hver søyle>
   <finn elementet med størst absoluttverdi i søylen under pivoten>
   <beregn nummeret q til raden som inneholder dette største elementet>
   <bytt om den opprinnelige pivot-raden med rad nr. q>
   HVIS <pivot-elementet ikke er null>
      FOR <hver rad nedenfor pivot-raden>
         <beregn hvilket multiplum av pivot-raden vi må trekke fra raden
          vi er i for å nulle ut elementet i søylen under pivoten>
         <trekk fra det riktige multiplumet av pivot-raden>
```

```
      END  % FOR
   <øk radindeksen med 1 for å lete etter neste pivot i raden nedenfor.>
   <avbryt løkken dersom radindeksen blir større enn antall rader>
   END  % HVIS
% (Hvis elementet i pivot-posisjonen er null, vet vi (siden dette er
%  elementet med maksimal absoluttverdi i søylen) at alle elementer i
%  søylen er null, så vi fortsetter bare til neste søyle uten å
%  oppdatere radindeksen)
END  % FOR
```

Oppgave 3

Lag en MATLAB-funksjon `gauss_delpiv` som tar en vilkårlig $(m \times n)$-matrise A som inn-parameter, utfører Gauss-eliminasjon med delvis pivotering på den og returnerer matrisen på trappeform. Husk at du kan ha nytte av kommandoen

```
>> [maksverdi,p] = max(abs(A(j:n,j)));
```

Du kan følge programskissen ovenfor, men kan gjerne bruke en `while`-løkke isteden hvis du synes det gir penere programmering.

Gauss-Jordan-eliminasjon

Hittil har vi bare laget funksjoner som bringer en matrise over på trappeform ved hjelp av Gauss-eliminasjon. I dette avsnittet skal vi se hvordan vi kan utvide disse til en funksjon som bringer matrisen på *redusert trappeform*, dvs en trappeform hvor vi (i tillegg til krav i) og ii) til en matrise på trappeform, har følgende krav oppfylt:

iii) alle ledende elementer er 1-ere

iv) i hver pivot-søyle er alle elementer null bortsett fra pivot-elementet

Dette oppnås gjerne ved såkalt bakoverreduksjon, dvs at vi etter å ha fått matrisen på trappeform starter i nederste rad og skaffer oss nuller over pivot-elementet (hvis det fins et) i denne raden. Deretter gjør vi det samme over pivot-elementet i nest nederste rad osv. oppover. Fordelen med denne metoden er at vi skaffer nuller som vi drar med oss og har nytte av i senere utregninger etter hvert som vi jobber oss oppover (og dette gjør jobben mye lettere når vi utfører regningene for hånd). Når vi skal programmere, er det imidlertid enklere å gjøre disse operasjonene underveis i Gauss-eliminasjonen. Da trenger vi bare et par linjer ekstra med programkode for å gjøre funksjonen `gauss_delpiv` om til en funksjon som utfører Gauss-Jordan-eliminasjon og bringer matrisen over på redusert trappeform.

Oppgave 4

Utvid funksjonen `gauss_delpiv` til en funksjon `gauss_jordan` som tar en matrise A som inn-parameter, utfører Gauss-Jordan-eliminasjon på den og returnerer matrisen på redusert trappeform.

Hint: Du må sørge for to ting:

i) Gjør alle pivot-elementene til 1-ere ved å dele pivot-raden med pivot-elementet.

ii) Skaff nuller i søylen over hvert pivot-element ved å trekke et passende multiplum av pivot-raden fra radene ovenfor.

Lab 3: Lineær uavhengighet

av

Klara Hveberg

I denne laboratorieøvelsen skal vi se på hvordan vi kan bruke den reduserte trappeformen til en matrise for å undersøke om søylene i matrisen er lineært uavhengige, om matriseligninger av typen $A\mathbf{x} = \mathbf{b}$ er konsistente, og om de i så fall har en eller flere løsninger. I tillegg til å gi deg en test på om du har fått med deg sammenhengen mellom disse begrepene, vil labøvelsen gi deg trening i å lage MATLAB-funksjoner som er litt mer omfattende enn de du har laget tidligere.

La oss starte med å repetere et par grunnleggende begreper som du vil få bruk for i denne øvelsen. Vi sier at vektorene $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$ er *lineært uavhengige* dersom ligningen $x_1\mathbf{v}_1 + x_2\mathbf{v}_2 + \cdots + x_n\mathbf{v}_n = \mathbf{0}$ bare har den trivielle løsningen $x_1 = x_2 = \cdots = x_n = 0$. I motsatt fall sier vi at vektorene er *lineært avhengige*, og en av vektorene kan da uttrykkes som en lineær kombinasjon av de øvrige. Et ligningssystem er *konsistent* dersom det har minst én løsning og *inkonsistent* dersom det ikke har noen løsning.

I flere av oppgavene kan du ha nytte av boolske variabler som antar en sannhetsverdi (`true` eller `false`). Slike variabler er ikke omtalt i MATLAB-appendikset i "Flervariabel analyse med lineær algebra", men du kan finne ut mer om dem ved å gi kommandoen

```
>> help logical
```

Ønsker du en boolsk variabel som heter `avhengig` og som skal tilordnes verdien `true`, skriver du bare

```
>> avhengig = true % her bruker vi enkelt likhetstegn for tilordning
```

Da kan du senere teste sannhetsverdien til denne variabelen i en if-setning ved å skrive

```
>> if avhengig==true % her bruker vi dobbelt likhetstegn for testing
```

eller ved å bruke den kortere varianten

```
>> if avhengig
```

Hvis du ikke er så glad i å programmere, finner du halvferdige programbiter i seksjonen med hint, hvor du bare trenger å fylle ut noen vesentlige linjer som tester den matematiske forståelsen av problemet. Du lærer imidlertid mye mer av å prøve å sette opp strukturen i programmet på egen hånd først.

Oppgave 1

Lag en MATLAB-funksjon `linavh` som tar en vilkårlig matrise A som inn-parameter, og som skriver ut den reduserte trappeformen til matrisen sammen med beskjed om hvorvidt søylene i matrisen er lineært avhengige eller lineært uavhengige. Her er poenget at du bare skal bruke informasjon fra trappematrisen (og du har derfor f.eks ikke lov til å bruke den innebygde MATLAB-funksjonen `rank` selv om du skulle kjenne til begrepet "rangen til en matrise"). Du kan bruke den innebygde MATLAB-funksjonen `rref(A)` for å bringe en matrise på redusert trappeform. For å finne dimensjonen til en matrise, kan du bruke kommandoen `[m,n]=size(A)`.

Oppgave 2

Lag en MATLAB-funksjon `inkons` som tar en vilkårlig matrise A og en vektor $\mathbf{b}$ som inn-parametre, og som skriver ut den reduserte trappeformen til den utvidete matrisen $[A, \mathbf{b}]$ sammen med beskjed om hvorvidt matriseligningen $A\mathbf{x} = \mathbf{b}$ er konsistent eller inkonsistent. (Hvis systemet er konsistent, trenger ikke denne funksjonen å avgjøre hvor mange løsninger systemet har).

Oppgave 3

Utvid MATLAB-funksjonen `inkons` fra forrige oppgave til en funksjon `antlosn` som tar en vilkårlig matrise A og en vektor $\mathbf{b}$ som inn-parametre, og som i tillegg til å avgjøre om matriseligningen $A\mathbf{x} = \mathbf{b}$ er konsistent eller inkonsistent, også gir beskjed om hvor mange løsninger (en eller uendelig mange) systemet har dersom det er konsistent.

Oppgave 4

I denne oppgaven får du bruk for følgende matriser og søylevektorer:

$$A = \begin{pmatrix} 0 & 0 & 1 \\ 5 & 0 & 10 \\ 1 & 0 & 6 \end{pmatrix} \qquad B = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 3 & 4 & 6 \\ 3 & 1 & 4 & 2 \\ 5 & 5 & 7 & 10 \end{pmatrix} \qquad C = \begin{pmatrix} 1 & 2 & 3 & 4 & 1 \\ 5 & 3 & 4 & 6 & 2 \\ 3 & 1 & 4 & 2 & 3 \\ 5 & 5 & 7 & 10 & 4 \end{pmatrix}$$

$$D = \begin{pmatrix} 5 & 2 & 7 & 5 \\ 7 & 7 & 4 & 9 \\ 4 & 3 & 9 & 8 \\ 3 & 5 & 9 & 6 \\ 2 & 2 & 6 & 8 \end{pmatrix} \qquad E = \begin{pmatrix} 1 & 7 & 6 & 2 & 2 & 6 \\ 0 & 3 & 4 & 6 & 4 & 8 \\ 9 & 5 & 5 & 8 & 8 & 1 \\ 2 & 1 & 3 & 5 & 7 & 6 \\ 3 & 10 & 4 & 6 & 5 & 1 \end{pmatrix}$$

$$\mathbf{a} = \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix} \qquad \mathbf{b} = \begin{pmatrix} 1 \\ 3 \\ 2 \\ 1 \end{pmatrix} \qquad \mathbf{d} = \begin{pmatrix} 4 \\ 14 \\ 6 \\ 10 \\ 4 \end{pmatrix}$$

a) Test funksjonen `linavh` på matrisene A, B, C, D og E og sjekk om svaret er riktig.

b) Test funksjonen `inkons` med følgende inn-parametre og sjekk om svaret er riktig: $(A, \mathbf{a})$, $(B, \mathbf{b})$, $(C, \mathbf{b})$, $(D, \mathbf{d})$ og $(E, \mathbf{d})$. Dersom svaret du får er at et ligningssystem er konsistent, skal du bruke funksjonen `antlosn` til å avgjøre hvor mange løsninger systemet har.

Lab 4: Newtons metode for flere variable

av

Klara Hveberg

I denne laboratorieøvelsen skal vi studere Newtons metode for å finne tilnærmede løsninger til systemer av ikke-lineære ligninger. Vi starter med å repetere Newtons metode for funksjoner av en variabel, og ser deretter på hva som skjer hvis vi bruker samme metode på en funksjon hvor variabelen er et komplekst tall. Her skal vi spesielt studere hvordan ulike valg av startverdier påvirker konvergensen, og se at dette leder til noen fascinerende geometriske figurer som kalles fraktaler. Til slutt ser vi på hvordan Newtons metode kan generaliseres til å fungere for systemer av ligninger i flere variable.

Hvis du føler at du har god kontroll på Newtons metode for funksjoner av en variabel (og kanskje til og med har programmert metoden i PYTHON eller lignende språk tidligere), kan du godt hoppe over innledningen, gjøre oppgave 2, og deretter gå videre til avsnittet om Newtons metode for funksjoner av en kompleks variabel.

Newtons metode for funksjoner av en variabel

La oss begynne med en kort repetisjon av ideen bak Newtons metode. Fra teorien for funksjoner av en variabel, vet vi at når vi bruker Newtons metode til å finne tilnærmede løsninger til et nullpunkt for en funksjon $f(x)$, starter vi med å gjette på et nullpunkt x_0 (som ofte må velges med litt omhu for å få metoden til å fungere), og deretter danner vi oss en følge $x_1, x_2, x_3, \ldots$ av (forhåpentligvis) stadig bedre tilnærminger til nullpunktet ved å bruke følgende strategi: Hver gang vi har en tilnærmingsverdi x_n, lar vi den neste tilnærmingsverdien x_{n+1} være skjæringspunktet mellom x-aksen og tangenten til grafen i punktet $(x_n, f(x_n))$. Eller sagt på en annen måte: Hver gang vi har en tilnærmingsverdi x_n, finner vi den neste tilnærmingsverdien x_{n+1} ved å erstatte den opprinnelige funksjonen med den beste lineære tilnærmingen til funksjonen over punktet x_n (dvs tangenten) og bruke nullpunktet til denne lineariseringen som en tilnærmet verdi for nullpunktet til funksjonen.

Tangenten til funksjonen i punktet $(x_n, f(x_n))$ har ligningen

$$y = f(x_n) + f'(x_n)(x - x_n)$$

For å finne ut hvor denne tangenten skjærer x-aksen, setter vi inn $y = 0$ og får dermed ligningen

$$0 = f(x_n) + f'(x_n)(x - x_n)$$

Løser vi denne ligningen med hensyn på x, og kaller løsningen x_{n+1}, får vi følgende rekursjonsformel for hvordan vi kan regne ut x_{n+1}:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

La oss først se et enkelt eksempel på hvordan vi kan utnytte denne formelen for å få MATLAB til å finne en tilnærmet verdi for nullpunktet til funksjonen $f(x) = x^2 - 2$ (eller

med andre ord finne en tilnærmet verdi for kvadratroten av 2). Siden den deriverte av funksjonen vår er $f'(x) = 2x$, blir rekursjonsformelen vi skal bruke i dette tilfellet

$$x_{n+1} = x_n - \frac{x_n^2 - 2}{2x_n}$$

For å få MATLAB til å regne ut de 5 første tilnærmingsverdiene fra Newtons metode med startverdi $x_0 = 1$, kan vi derfor bruke følgende `for`-løkke:

```
>> x=1;
>> for t=1:5
      xny=x-(x^2-2)/(2*x); % Beregner neste tilnærmingsverdi
      x=xny    % utelater semikolon for å få se alle tilnærmingsverdiene
   end
```

Det kan nå være lurt å sjekke om den siste verdien MATLAB beregnet (som nå ligger i variabelen x) faktisk er en god tilnærming til nullpunktet for funksjonen, ved å regne ut funksjonsverdien i dette punktet med kommandoen

```
>> x^2-2
```

Merknad: I MATLAB er indekseringene av x-ene strengt tatt unødvendig, siden likhetstegnet betyr "tilordning" og ikke "likhet". Istedenfor å gi kommandoene

```
>> xny=x-(x^2-2)/(2*x)
>> x=xny;
```

kan vi simpelthen gi kommandoen

```
>> x=x-(x^2-2)/(2*x)
```

Her spiller x-en på venstre side rollen som `xny`, mens den på høyre side er vår gamle `x`. I denne laboratorieøvelsen kommer vi imidlertid stort sett til å benytte det første alternativet i håp om at det vil gjøre det enklere å forstå programmene (men hvis du føler deg trygg på det andre alternativet, må du gjerne benytte dette).

Oppgave 1

a) Bruk en `for`-løkke for å få MATLAB til å regne ut de 5 første tilnærmingsverdiene Newtons metode gir til nullpunktet som funksjonen $f(x) = x^3 + 2x^2 - 2$ har i intervallet mellom 0 og 1. Du kan bruke startverdien $x_0 = 1$. (Husk at du først må finne rekursjonsformelen for denne funksjonen ved å regne ut $f'(x)$ og sette inn i den generelle formelen for Newtons metode.) Ser det ut som om tilnærmingene konvergerer mot en bestemt verdi? Undersøk i så fall om denne verdien faktisk er et nullpunkt for funksjonen ved å la MATLAB regne ut funksjonsverdien i dette punktet.

b) Hvor mange iterasjoner må gjennomføres før de fire første desimalene i tilnærmingsverdien ikke endrer seg? Hvor mange iterasjoner som må gjennomføres før de tolv første desimalene ikke endrer seg.

   MATLAB-tips: For å få MATLAB til å skrive ut flere desimaler på skjermen, kan du gi kommandoen

   ```
   >> format long
   ```

Litt om konvergens

Det er ikke sikkert at følgen av approksimasjoner vi får ved Newtons metode konvergerer mot nullpunktet vi er på jakt etter, men sjansen for at metoden fungerer blir større hvis startpunktet x_0 ligger i nærheten av det faktiske nullpunktet. En god (men ikke sikker) indikasjon på at approksimasjonene konvergerer, er at forskjellen mellom to påfølgende ledd x_n og x_{n+1} i følgen blir liten. Når man skal programmere Newtons metode, er det derfor vanlig å spesifisere en toleranse (f.eks 10^{-12}) og stoppe iterasjonene dersom forskjellen mellom x_n og x_{n+1} (i absoluttverdi) blir mindre enn denne toleransen. Man antar da at x_{n+1} er et godt estimat for nullpunktet (men man bør egentlig sjekke at funksjonsverdien i dette punktet faktisk er nær null). Dersom approksimasjonene ikke konvergerer, risikerer man at differensen mellom x_{n+1} og x_n aldri blir mindre enn den oppgitte toleransen, så for å unngå å bli gående i en evig løkke av iterasjoner, bør man også spesifisere et maksimalt antall iterasjoner som skal utføres.

La oss vende tilbake til eksempelet med funksjonen $f(x) = x^2 - 2$ og se på hvordan vi kan modifisere `for`-løkken slik at MATLAB utfører iterasjonene av Newtons metode (med startverdi $x_0 = 1$) inntil differensen mellom x_{n+1} og x_n er mindre enn en toleranse på 0.0001, men likevel slik at den ikke utfører mer enn maksimalt 20 iterasjoner. Vi bruker da kommandoen `break` til å avbryte løkken hvis differansen blir mindre enn den valgte toleransen:

```
>> tol=0.0001;      %angir den ønskede toleransen
>> maksit=20;       %maksimalt antall iterasjoner som skal utføres
>> x=1;             %startverdi
>> diff=tol+1;      %initialiserer diff til å være større enn toleransen
>> for t=1:maksit
     xny=x-(x^2-2)/(2*x);
     diff=abs(xny-x);    % beregner absoluttverdien av differensen
     x=xny               % utelater semikolon for å få se alle verdiene
       if diff<tol       % tester om vi skal avbryte løkken
          break;
       end % if
   end % for
```

Du kan selvfølgelig benytte en `while`-løkke isteden, hvis du foretrekker det:

```
>> tol=0.0001;      %angir den ønskede toleransen
>> maksit=20;       %maksimalt antall iterasjoner som skal utføres;
>> diff=tol+1;      %initialiserer diff til å være større enn toleransen
>> x=1;
>> t=0;
>> while (diff>tol) & (t<maksit)
      xny=x-(x^2-2)/(2*x);
      diff=abs(xny-x);   % beregner absoluttverdien av differensen
      x=xny              % utelater semikolon for å få se alle verdiene
      t=t+1;
   end
```

Når vi skal lete etter nullpunktene til en funksjon ved hjelp av Newtons metode, vil det ofte være aktuelt å prøve med litt forskjellige startverdier. Det kan derfor være kjekt å lage en MATLAB-funksjon `newton` hvor vi kan spesifisere startverdien som innparameter.

For å kunne benytte denne på forskjellige funksjoner uten å måtte gå inn i hovedfilen `newton.m` og endre rekursjonsformelen hver gang, kan det også være greit å bruke den generelle rekursjonsformelen $x_{n+1} = x_n - \frac{f(x)}{f'(x)}$ i `newton` og la den hente informasjon om den aktuelle funksjonen og dens deriverte fra en egen m-fil `fogder.m`. Da trenger vi bare å gjøre endringer lokalt i den sistnevnte filen når vi bytter funksjon. Hvis vi vil studere flere funksjoner, er det ofte praktisk å kunne ha dem liggende på m-filer med forskjellige navn istedenfor å måtte endre på filen `fogder.m` hele tiden. Dette kan vi få til ved å sende med navnet på m-filen som innparameter til MATLAB-funksjonen `newton`. Vi lar i så fall denne få følgende signatur:

```
function x=newton(start,fogdf)
% MATLAB-funksjon som prøver å finne nullpunkt for en funksjon ved hjelp
% av Newtons metode. Den leser inn definisjonen av funksjonen og dens
% deriverte fra m-filen med navn angitt av innparameteren fogdf
%Innparametre:
% start: startverdi for iterasjonene
% fogdf: navn på fil som inneholder den aktuelle funksjonen og dens
% deriverte
%Utparameter:
% x: den beregnede tilnærmingsverdien til nullpunktet
<Programkode: Denne skal du lage i oppgave 2>
```

Ønsker vi f.eks å bruke MATLAB-funksjonen `newton` på funksjonen $f(x) = x^3 + 2x^2 - 2$ (fra oppgave 1), spesifiserer vi bare $f(x)$ og dens deriverte på m-filen `fogder.m` på følgende måte:

```
function [y,d]=fogder(x)
y=x^3+2*x^2-2;
d=3*x^2+4*x;
```

For å bruke MATLAB-funksjonen `newton` med startpunkt $x_0 = 2$ på funksjonen $f(x)$, sender vi nå navnet på funksjonsfilen `fogder.m` med som innparameter ved å gi kommandoen

```
>> x=newton(2,'fogder');
```

Legg spesielt merke til at vi må ha apostrofer rundt navnet på filen som vi sender med som innparameter, men at vi utelater etternavnet `.m`.

La oss til slutt se på hvordan vi i MATLAB-funksjonen `newton` kan lese inn funksjonsverdien og den deriverte fra m-filen med navn gitt ved innparameteren `fogdf`. For å beregne funksjonsverdien og den deriverte i et punkt x og ta vare på disse verdiene i to variable vi kaller `fx` og `dfx`, kan vi bruke den innebygde MATLAB-kommandoen `feval` på følgende måte:

```
[fx,dfx]=feval(fogdf,x);
```

MATLAB-funksjonen `feval` tar altså et funksjonsnavn og et punkt som innparametre, og returnerer funksjonen evaluert i det oppgitte punktet.

Merknad: Grunnen til at vi må bruke MATLAB-funksjonen `feval` er at innparameteren `fogdf` er et navn på en funksjonsfil (altså en tekststreng), så det gir ikke mening å prøve å kalle funksjonen med innparameter `x` ved å gi kommandoen `[fx,dfx]=fogdf(x)`. Hvis du er fortrolig med anonyme funksjoner i MATLAB, kan du gjerne la `newton` ta slike funksjoner direkte som innparametre istedenfor å bruke en m-fil og sende med filnavnet.

Oppgave 2

a) Lag en MATLAB-funksjon på m-filen `newton.m` som leser inn verdiene til en funksjon og dens deriverte fra en egen m-fil, og beregner tilnærmingsverdier for nullpunktet ved hjelp av disse verdiene. Som innparametre skal MATLAB-funksjonen ta et startpunkt `start` og navnet `fogdf` på m-filen hvor funksjonen f og dens deriverte er definert. MATLAB-funksjonen skal deretter utføre Newtons metode inntil differensen mellom x_{n+1} og x_n er mindre enn en toleranse på 0.0000001, eller den har utført det maksimale antall iterasjoner som du tillater (du kan f.eks la dette være `maksit=30`). Til slutt skal MATLAB-funksjonen returnere den sist beregnede tilnærmingsverdien som utparameter.

MATLAB-tips: Dersom iterasjonsindeksen din heter `t`, kan du legge til følgende kommandolinje i iterasjonsløkka for å få en pen utskrift på skjermen av iterasjonsnummeret, den tilhørende tilnærmingsverdien `x` som blir beregnet for nullpunktet i hvert trinn, og tilhørende funksjonsverdi `fx` i dette punktet:

```
fprintf('itnr=%3d  x=%15.12f  f(x)=%15.12f\n',t,x,fx)
```

Du kan lese mer om denne kommandoen ved å skrive

```
>> help fprintf
```

i MATLAB-vinduet.

b) Test MATLAB-funksjon du lagde i punktet ovenfor på funksjonen $f(x) = x^3 + 2x^2 - 2$ (fra oppgave 1) for å finne tilnærmingsverdier for nullpunktet funksjonen har i intervallet $[0, 1]$.

c) Lag et plott av funksjonen $f(x) = x^3 + 2x^2 - 30x - 5$ over intervallet $[-8, 6]$ for å danne deg et bilde av hvor nullpunktene ligger. Bruk deretter MATLAB-funksjonen du lagde i punkt a) til å finne tilnærmingsverdier for hvert av de tre nullpunktene. Du kan for eksempel bruke startverdiene $x_0 = 1$, 4 og -4.

d) Vi skal nå se på noen tilfeller hvor Newtons metode virker dårlig. Vi skal først ta for oss funksjonen $g(x) = x^3 - 5x$. Lag et plott av denne funksjonen over intervallet $[-3, 3]$ og kjør deretter Newtons metode med startpunktet $x_0 = 1$. Hva skjer med iterasjonene? Kan du se hvorfor dette skjer utifra grafen du plottet? Undersøk om det samme problemet oppstår hvis du kjører Newtons metode på nytt med startpunktet $x_0 = 2$ isteden.

La oss deretter prøve Newtons metode på funksjonen $h(x) = x^{\frac{1}{3}}$. Her er $x = 0$ opplagt det eneste nullpunktet. Lag et plott av funksjonen over intervallet $[-4, 4]$ for å danne deg et bilde av hvordan grafen oppfører seg i nærheten av nullpunktet. Kjør deretter Newtons metode med startpunkt $x_0 = 1$. Nærmer tilnærmingsverdiene seg nullpunktet $x = 0$? Prøv å forklare hva som skjer utifra grafen.

MATLAB-tips: For å unngå at MATLAB returnerer komplekse røtter av negative tall, kan det være lurt å representere funksjonen $h(x) = x^{1/3}$ på følgende måte i MATLAB:

```
>> y=sign(x).*abs(x).^(1/3);
```

Newtons metode for funksjoner av en kompleks variabel

I forrige seksjon studerte vi Newtons metode for å finne tilnæringsverdier for nullpunkter til funksjoner av en reell variabel, men det viser seg at metoden fungerer like godt for funksjoner av en kompleks variabel. Du har sannsynligvis ikke vært borti komplekse funksjoner tidligere, men hvis du foreløpig bare godtar påstanden om at de kan deriveres

på samme måte som reelle funksjoner, vil du kunne plugge dem inn i rekursjonsformelen for Newtons metode og se hva som skjer.

Som vanlig begynner vi da med å gjette på et startpunkt x_0, som denne gangen er et komplekst tall på formen $a + ib$ og altså kan tolkes som et punkt (a, b) i planet. Deretter bruker vi rekursjonsformelen til å danne oss en følge av komplekse tall som (forhåpentligvis) er stadig bedre tilnærminger til et nullpunkt for funksjonen vi studerer. Da vi studerte Newtons metode på funksjoner av en reell variabel, så vi at startpunktet vi valgte kunne være avgjørende for om følgen av tilnærminger konvergerte, hvilket nullpunkt de eventuelt konvergerte mot og hvor fort de konvergerte mot dette. Generelt økte sjansen for konvergens hvis startpunktet vi valgte lå nær det virkelige nullpunktet, men fasongen på grafen var også avgjørende.

For funksjoner av komplekse variable, er situasjonen enda mer innviklet — så innviklet at det å studere hvilke nullpunkter de forskjellige startpunktene i planet gir oss konvergens mot, leder til kompliserte og fascinerende figurer som kalles fraktaler. Dette skal du få en liten smakebit på i denne seksjonen hvor vi skal studere Newtons metode på komplekse funksjoner av typen $f(z) = z^3 + az + b$ for ulike verdier av konstantene a og b. Som nevnt kan vi derivere disse funksjonene på vanlig måte og få at $f'(z) = 3z^2 + a$, så rekursjonsformelen for Newtons metode blir i dette tilfellet $z_{n+1} = z_n - \frac{z^3+az+b}{3z^2+a}$.

En funksjon på formen $f(z) = z^3 + az + b$ har tre (komplekse) nullpunkter, og det vi ønsker å gjøre, er å fargelegge (et område av) planet slik at vi kan se hvilke startpunkter som gir konvergens mot samme nullpunkt når vi bruker Newtons metode. For hvert av de tre nullpunktene til funksjonen velger vi altså ut en farge og deretter fargelegger vi alle startpunkter som gir konvergens mot samme nullpunkt, i den fargen som vi valgte for nullpunktet.

La oss se hvordan vi kan få MATLAB til å lage et slikt bilde. Først må vi definere hvilke komplekse tall vi ønsker å studere som startpunkter for Newtons metode (dvs hvilket området i planet vi ønsker å studere). La oss velge å se på de komplekse tallene hvor både realdelen og imaginærdelen ligger mellom -2 og 2. Vi kan ikke studere alle disse punktene siden det er uendelig mange av dem, men vi lager oss et tett gitter av punkter i dette området ved kommandoene:

```
>> x=[-2:0.01:2];
>> y=[-2:0.01:2];
>> [X,Y]=meshgrid(x,y);
>> [m,n]=size(X);        %tar vare på størrelsen for senere bruk i løkke
```

Deretter vil vi la MATLAB finne numeriske tilnærmingsverdier for nullpunktene til funksjonen vi skal studere (da blir det enkelt å sjekke hvilket nullpunkt tilnærmingsverdiene konvergerer mot). Til dette kan vi bruke den innebygde MATLAB-kommandoen `roots(p)` som returnerer en vektor med nullpunktene til et polynom p. Polynomet skal spesifiseres som en radvektor bestående av koeffisientene foran de ulike potensene av variabelen, slik at funksjonen $f(z) = z^3+az+b$ får polynomspesifikasjonen $p = [1, 0, a, b]$ (husk å ta med koeffisienten null foran "det usynlige" leddet z^2). Aller først må vi imidlertid spesifisere hvilken funksjon vi vil studere ved å angi verdiene av konstantene a og b. Vi vil starte med å studere funksjonen $f(z) = z^3 - 1$ og velger derfor $a = 0$ og $b = -1$:

```
>> a=0;
>> b=-1;
```

```
>> p=[1 0 a b];
>> rts=roots(p);
```

Nå er vi klare til å gjennomløpe ett og ett punkt i gitteret og utføre Newtons metode med dette punktet som startverdi. Som tidligere velger vi først et maksimalt antall iterasjoner som skal utføres, og en toleranse som avgjør når tilnærmingsverdiene er så nær et nullpunkt at vi vil avbryte iterasjonene. Deretter bruker vi en dobbelt `for`-løkke til å gjennomløpe punktene i gitteret. For hvert punkt bruker vi Newtons metode med punktet som startverdi og kjører gjentatte iterasjoner inntil tilnærmingsverdiene eventuelt kommer innenfor den oppgitte toleransen fra ett av nullpunktene. I så fall avbryter vi løkka og gir punktet den fargeverdien vi har tilegnet nullpunktet vi fikk konvergens mot. I dette tilfellet vil vi bruke fargeverdien 1 (blå) for det første nullpunktet `rts(1)`, fargeverdien 44 (gul) for det andre nullpunktet `rts(2)`, og fargeverdien 60 (rød) for det tredje nullpunktet `rts(3)`. Vi lagrer fargeverdiene til punktene i en matrise T slik at punkt nr (j,k) i gitteret får fargeverdien angitt av elementet $T(j,k)$ i matrisen. Før vi starter løkkene, initialiserer vi alle elementene i T til fargeverdien 27 (turkis), så de startpunktene som eventuelt ikke gir oss konvergens mot noe nullpunkt i løpet av det maksimale antallet iterasjoner vi utfører, vil beholde denne fargen.

```
>> maxit=30;
>> tol=0.001;
>> T=27*ones(m,n);
for j=1:m
    for k=1:n                 %gjennomløper punktene i gitteret
        z=X(j,k)+i*Y(j,k);    %omgjør punkt i gitter til komplekst tall
    %Så kjøres iterasjoner av Newtons metode med punktet som startverdi
        for t=1:maxit
            zny=z-(z^3+a*z+b)/(3*z^2+a);
            z=zny;
            if abs(z-rts(1))<tol
                T(j,k)=1;
               break;
            end
            if abs(z-rts(2))<tol
               T(j,k)=44;
               break;
            end
            if abs(z-rts(3))<tol
                T(j,k)=60;
                break;
            end
        end %for t
    end %for k
end %for j
```

Nå kan vi få fargelagt punktene i det aktuelle området i planet med fargeverdien bestemt i matrisen T ved å gi følgende kommandoer:

```
>> image(x,y,T) %fargelegger punktet (x(j),y(k)) i planet med fargen
                %angitt av verdien av matrise-elementet T(j,k)
>> axis xy      %setter aksesystemet i vanlig xy-orientering
```

```
>> colorbar     %tegner opp søyle som angir fargeverdiene ved plottet
```

Oppgave 3

a) Lag et skript på m-filen `kompleks.m` som genererer et bilde av hvilke punkter i planet som konvergerer mot de forskjellige nullpunktene til funksjonen $f(z) = z^3 - 1$.

b) Gjør de endringene som skal til i skriptet fra punkt a) for å få tilsvarende bilder for funksjonene $f(z) = z^3 + 2z$ (dvs at du setter parametrene til å være $a = 2$ og $b = 0$) og $f(z) = z^3 - z + 2$ (dvs at $a = -1$ og $b = 2$).

Hittil har vi laget bilder som forteller hvilket nullpunkt de forskjellige startpunktene gir konvergens mot, men bildene gir ingen informasjon om hvor raskt konvergensen går. For å få et bilde som også inneholder informasjon om konvergenshastigheten til de forskjellige startpunktene, kan vi justere litt på fargeverdiene slik at punktene får lysere fargevalør jo langsommere konvergensen går. Istedenfor bare å sette fargeverdiene til 1, 44 eller 60, kan vi justere fargeverdiene etter hvor mange iterasjoner t som er utført, f.eks på følgende måte:

```
if abs(z-rts(1))<tol
    T(j,k)=1+t*(23/maxit);
    break;
end
if abs(z-rts(2))<tol
   T(j,k)=48-t*(23/maxit);
    break;
end
if abs(z-rts(3))<tol
    T(j,k)=64-t*(23/maxit);
    break;
end
```

Merknad: Ikke blir forvirret over at vi gir et lite tillegg i det første tilfellet, men gjør et lite fradrag i de to neste. Det kommer bare av at fargeskalaen vi bruker er slik at blåfargen blir lysere for høyere verdier, mens gul og rød blir lysere for lavere verdier.

Oppgave 4

a) Modifiser skriptet fra forrige oppgave slik at bildet gir informasjon om konvergenshastigheten til de ulike startpunktene i tillegg til hvilket nullpunkt de gir konvergens mot. Du kan gjerne gjøre om skriptet til en MATLAB-funksjon som tar a og b som innparametre, slik at du lett kan eksperimentere med ulike funksjoner.

b) Generer et bilde for funksjonen $f(z) = z^3 - 1$ og beskriv hva bildet viser. Prøv å gjette på hvor nullpunktene ligger utifra bildet.

c) Generer tilsvarende bilder for funksjonene $f(z) = z^3 + 2z$ og $f(z) = z^3 - z + 2$.

Oppgave 5

Modifiser MATLAB-funksjonen fra forrige oppgave slik at du fritt kan velge sentrum og forstørrelse. Bruk den på funksjonen $f(z) = z^3 - 1$, og lag en sekvens av bilder som zoomer inn på origo, og deretter en tilsvarende sekvens som zoomer inn på punktet $(-0.8, 0)$.

Hint: Første del av skriptet vi har brukt tidligere, kan f.eks omskrives slik:

```
sentrumx=0;
sentrumy=0;
lengde=2;
x=linspace(sentrumx-lengde, sentrumx+lengde,400);
y=linspace(sentrumy-lengde, sentrumy+lengde,400);
```

Her har du nå mulighet for å velge andre koordinater for sentrumspunktet og andre intervallengder. Du kan for eksempel zoome inn mot origo ved først å sette $lengde = 1$, deretter $lengde = 0.5$, og til slutt $lengde = 0.25$. Flytt deretter sentrum litt mot venstre ved å velge $sentrumx = -0.8$, og foreta en tilsvarende innzooming der.

Ekstraoppgave: Hvis du har gjennomført den frivillige slutten på laboratorieøvelse nr 1, hvor du lærte å lage animasjoner, kan du prøve å lage en animasjon som viser innzooming mot et punkt på fraktalen.

Newtons metode for systemer av to ligninger i to ukjente

La oss se på hvordan Newtons metode kan generaliseres til å løse ligningssystemer av typen:

$$f(x, y) = 0$$
$$g(x, y) = 0$$

Siden vi nå arbeider med funksjoner av to variable, blir grafene flater i rommet, og nullpunktene vi skal tilnærme vil ligge i xy-planet. Strategien er imidlertid den samme som før: Vi starter med å gjette på et nullpunkt (x_0, y_0), og deretter danner vi oss en følge $(x_1, y_1), (x_2, y_2), (x_3, y_3) \ldots$ av (forhåpentligvis) stadig bedre tilnærminger til nullpunktet ved å benytte nullpunktene til lineære approksimasjoner (tangentplan) av de opprinnelige funksjonene: Hver gang vi har en tilnærmingsverdi (x_n, y_n), finner vi den neste tilnærmingsverdien (x_{n+1}, y_{n+1}) ved å erstatte de opprinnelige funksjonene $f(x, y)$ og $g(x, y)$ med deres beste lineære tilnærminger (dvs tangentplanene) over punktet (x_n, y_n), og bruke skjæringspunktet mellom xy-planet og disse lineariseringene som en tilnærmet verdi for nullpunktet til de opprinnelige funksjonene.

Analogien til det éndimensjonale tilfellet blir klarere om vi benytter vektornotasjonen

$$\mathbf{x} = \begin{pmatrix} x \\ y \end{pmatrix}, \ \mathbf{F} = \begin{pmatrix} f \\ g \end{pmatrix} \ \text{og} \ \ \mathbf{F}'(\mathbf{x}) = \begin{pmatrix} \frac{\partial f}{\partial x}(x, y) & \frac{\partial f}{\partial y}(x, y) \\ \frac{\partial g}{\partial x}(x, y) & \frac{\partial g}{\partial y}(x, y) \end{pmatrix}$$

der $\mathbf{F}'(x)$ er Jacobi-matrisen til $\mathbf{F}$ i punktet $\mathbf{x}$. Skjæringslinjen mellom de to tangentplanene er da gitt ved lineariseringen

$$T_{\mathbf{x}_n}\mathbf{F}(\mathbf{x}) = \mathbf{F}(\mathbf{x}_n) + \mathbf{F}'(\mathbf{x}_n)(\mathbf{x} - \mathbf{x}_n)$$

til vektorfunksjonen $\mathbf{F}$ i punktet $\mathbf{x}_n = (x_n, y_n)$, og denne linjen skjærer xy-planet i nullpunktet til lineariseringen $T_{\mathbf{x}_n}\mathbf{F}$, det vil si i det punktet $\mathbf{x}$ som oppfyller den lineære ligningen

$$\mathbf{F}(\mathbf{x}_n) + \mathbf{F}'(\mathbf{x}_n)(\mathbf{x} - \mathbf{x}_n) = 0$$

Dersom Jacobi-matrisen $\mathbf{F}'(\mathbf{x}_n)$ er invertibel, får vi løsningen

$$\mathbf{x} = \mathbf{x}_n - \mathbf{F}'(\mathbf{x}_n)^{-1}\mathbf{F}(\mathbf{x}_n)$$

som vi bruker som neste tilnærming $\mathbf{x}_{n+1}$ til nullpunktet for $\mathbf{F}$. Denne prosedyren gir oss altså rekursjonsformelen

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \mathbf{F}'(\mathbf{x}_n)^{-1}\mathbf{F}(\mathbf{x}_n)$$

som vi kan bruke til å bestemme følgen av tilnærminger til nullpunktet. På komponentform blir formelen

$$\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \begin{pmatrix} x_n \\ y_n \end{pmatrix} - \begin{pmatrix} \frac{\partial f}{\partial x}(x_n, y_n) & \frac{\partial f}{\partial y}(x_n, y_n) \\ \frac{\partial g}{\partial x}(x_n, y_n) & \frac{\partial g}{\partial y}(x_n, y_n) \end{pmatrix}^{-1} \begin{pmatrix} f(x_n, y_n) \\ g(x_n, y_n) \end{pmatrix}$$

Når vi skal implementere løsningen i MATLAB, er det lite effektivt å løse ligningssystemet ved å regne ut den inverse av Jacobi-matrisen og multiplisere med denne. Det går mye raskere å løse ligningssystemet ved å bruke MATLABs innebygde venstredivisjonsoperator \. Istedenfor å gi en kommando av typen

```
>> xny=x-inv(J)*f(x)
```

bruker vi altså heller en kommando av typen

```
>> xny=x-J\f(x)
```

hvor J er Jacobi-matrisen. Legg spesielt merke til rekkefølgen her: Når vi skal bruke venstredivisjonsoperatoren til å "dele $f(x)$ på J" (dvs venstremultiplisere $f(x)$ med J^{-1}), skal J stå til venstre for divisjonsoperatoren.

Oppgave 6

a) Lag en MATLAB-funksjon på filen `'newtonfler.m'` som leser inn verdiene til en vektorfunksjon F og dens Jacobi-matrise J fra en egen m-fil, og bruker disse verdiene til å beregne tilnærmingsverdier til nullpunktet ved hjelp av Newtons metode. Som innparametre skal MATLAB-funksjonen ta et startpunkt `start` (angitt som en søylevektor) og navnet på en MATLAB-funksjon `FogJ` som returnerer en vektor [F,J], hvor F er en vektorfunksjon (angitt som en søylevektor med komponenter f og g) og J er dens Jacobi-matrise med de partiellderiverte. MATLAB-funksjonen skal deretter utføre Newtons metode inntil normen til $x_{n+1} - x_n$ er mindre enn en toleranse på 0.0000001, eller den har utført det maksimale antallet iterasjoner du tillater (du kan f.eks velge dette til å være `maksit`= 30). Til slutt skal den returnere den sist beregnede tilnærmingsverdien som utparameter.

MATLAB-tips: Du kan bruke MATLAB-funksjonen `norm(xny-x)` for å regne ut den aktuelle normen. Dersom iterasjonsindeksen din heter 't', kan du legge til følgende kommandolinje i iterasjonsløkka for å få en pen utskrift på skjermen av iterasjonsnummeret, alle tilnærmingsverdiene x som blir beregnet for nullpunktet og tilhørende funksjonsverdi $F(x)$ for vektorfunksjonen i disse punktene:

```
fprintf('itnr=%2d x=[%13.10f,%13.10f] F(x)=[%13.10f,%13.10f]\n',...
t,x(1),x(2),F(1),F(2))
```

b) I resten av denne oppgaven skal vi studere ligningssystemet

$$x^2 + y^2 - 48 = 0$$

$$x + y + 6 = 0$$

Bruk MATLAB til å lage et implisitt plott av de to ligningene i samme figurvindu (du kan for eksempel lage et kontur-plott med en enkelt nivåkurve for funksjonsverdien 0). Bruk figuren til å bestemme hvor mange løsninger dette ligningssystemet har.

MATLAB-tips: Hvis du har definert en funksjon f over et område gitt ved kommandoen `[x,y]=meshgrid(-10:0.1:10)`, kan du lage et kontur-plott for funksjonen f med bare én nivåkurve i høyden 0 ved å gi kommandoen

```
>> contour(x,y,f,[0 0])
```

(Generelt vil kommandoen `contour(x,y,f,v)` gi et plott av nivåkurvene til funksjonen f i høydene spesifisert i vektoren v).

c) Bruk figuren du lagde i punkt b) til å finne fornuftige startverdier for Newtons metode, og kjør deretter MATLAB-funksjonen `newtonfler` med disse startverdiene for å finne tilnærmede verdier for nullpunktene.

Oppgave 7

Dersom du har fulgt den generelle løsningsmetoden vi antydet i teksten da du lagde MATLAB-funksjonen `newtonfler` i forrige oppgave, burde den også fungere for systemer av mer enn to variable. Du kan undersøke dette ved å legge inn vektorfunksjoner med flere komponenter i funksjonsfilen som sendes som innparameter. Vil du for eksempel løse et ligningssystem på formen

$$f(x, y, z) = 0$$

$$g(x, y, z) = 0$$

$$h(x, y, z) = 0$$

definerer du bare en vektorfunksjon $F = [f; g; h]$ som en søylevektor med tre komponenter, og dens tilhørende Jacobi-matrise $J = [dfx, dfy, dfz; dgx, dgy, dgz; dhx, dhy, dhz]$ på funksjonsfilen du sender med som innparameter til Newton. Prøv ut dette for å finne en tilnærmet løsning på ligningssystemet

$$y^2 + z^2 - 3 = 0$$

$$x^2 + z^2 - 2 = 0$$

$$x^2 - z = 0$$

ved hjelp av Newtons metode.

Merknad: MATLAB-funksjonen `newtonfler` fra forrige oppgave skal ikke trenge noen modifikasjon for å beregne tilnærmingsverdiene for systemer av mer enn to variable. Men i kommandoen `fprintf` som vi benyttet for å få en pen utskrift på skjermen av alle de beregnede tilnærmingsverdiene, ba vi bare om å få skrevet ut de to første komponentene til x og $F(x)$. For å få en utskrift av alle de tre komponentene (med litt færre desimaler enn sist, slik at vi får plass på skjermen), kan du f.eks modifisere kommandoen slik:

```
fprintf('itnr=%2d x=[%9.6f,%9.6f,%9.6f] F(x)=[%9.6f,%9.6f,%9.6f]\n',...
t,x(1),x(2),x(3),F(1),F(2),F(3))
```

Appendiks 1

Oppvarmingsøvelser

I denne seksjonen finner du noen enkle oppvarmingsøvelser som du kan gjøre for å trene inn MATLAB-kommandoene som brukes til å operere på rader og søyler i en matrise. Disse øvelsene krever ikke kjennskap til bruk av m-filer og funksjoner i MATLAB, og kan derfor gjennomføres mens du leser første del av MATLAB-appendikset i "Flervariabel analyse med lineær algebra".

Øvelse 1: Matrisemultiplikasjon

av

Klara Hveberg

I denne enkle oppvarmingsøvelsen skal vi ta utgangspunkt i den komponentvise definisjonen av matrisemultiplikasjon og se hvordan denne leder til alternative måter å beskrive multiplikasjonen på ved hjelp av søylene og radene i matrisene.

La oss begynne med å repetere den komponentvise definisjonen av matrisemultiplikasjon: Produktet av to matriser A og B er definert ved at elementet i posisjon (i, k) i produktmatrisen AB er skalarproduktet mellom i-te rad i A og k-te søyle i B, det vil si summen av produktene av korresponderende elementer i rad nr. i fra A og søyle nr. k fra B:

$$(AB)(i,k) = \sum_{j=1}^{n} A(i,j)B(j,k)$$

Produktet av en matrise og en søylevektor

Lag en matrise

```
>> C=[1 5 2; 9 6 8; 8 7 4];
```

og en søylevektor

```
>> x=[2; 5; 8];
```

Siden søylevektoren x er en (3×1)-matrise, kan vi bruke den komponentvise definisjonen av matrisemultiplikasjon og la MATLAB regne ut matriseproduktet

```
>> C*x
```

Bruk deretter søylene i C som vektorer

```
>> c1=C(:,1);
>> c2=C(:,2);
>> c3=C(:,3);
```

og la MATLAB beregne lineærkombinasjonen av disse vektorene hvor vi bruker elementene i søylevektoren $\mathbf{x}$ som koeffisienter:

```
>> x(1)*c1+x(2)*c2+x(3)*c3
```

Sammenlign med matriseproduktet du beregnet ovenfor.

Oppgave 1

a) Hva forteller dette resultatet om sammenhengen mellom matriseproduktet $C\mathbf{x}$ og søylene i C? Formuler denne sammenhengen med ord.

b) Er dette en generell sammenheng som også gjelder for andre matriser C og vektorer $\mathbf{x}$? Prøv i så fall å forklare denne sammenhengen ut ifra den komponentvise definisjonen av matrisemultiplikasjon.

Søylevis matrisemultiplikasjon

Definer to matriser

```
>> A=[2 4 6; 1 3 5; 7 8 9];
>> B=[1 4; 2 5; 3 6];
```

og la MATLAB regne ut matriseproduktet

```
>> A*B
```

Bruk deretter søylene i B som vektorer

```
>> b1=B(:,1);
>> b2=B(:,2);
```

og la MATLAB regne ut produktene

```
>> A*b1
>> A*b2
```

Oppgave 2

a) Hva er sammenhengen mellom AB, $A\mathbf{b}_1$ og $A\mathbf{b}_2$? Prøv å formulere med ord en regel for hvordan vi kan regne ut hver av søylene i matrisen AB.

b) Er dette en generell sammenheng som også gjelder for andre matriser A og B? Prøv i så fall å forklare denne sammenhengen ut ifra den komponentvise definisjonen av matrisemultiplikasjon.

Oppgave 3

Bruk resultatene fra oppgave 1 og 2 til å forklare hvordan man kan regne ut første søyle i matriseproduktet AB ved hjelp av søylene i A. Gjør det samme for andre søyle i AB.

Bruk til slutt MATLAB til å sjekke at resultatet stemmer for matrisene A og B som vi definerte før oppgave 2.

Produktet av en radvektor og en matrise

Lag en matrise

```
>> R=[1 3 8; 4 6 2; 5 7 9];
```

og en radvektor

```
>> y=[3 -4 6];
```

Siden radvektoren $\mathbf{y}$ er en (1×3)-matrise, kan vi bruke den komponentvise definisjonen av matrisemultiplikasjon og la MATLAB regne ut matriseproduktet

```
>> y*R
```

Bruk deretter radene i R som vektorer

```
>> r1=R(1,:);
>> r2=R(2,:);
>> r3=R(3,:);
```

og la MATLAB beregne lineærkombinasjonen av disse vektorene hvor vi bruker elementene i vektoren $\mathbf{y}$ som koeffisienter:

```
>> y(1)*r1-y(2)*r2+y(3)*r3
```

Sammenlign med matriseproduktet du beregnet ovenfor.

Oppgave 4

a) Hva forteller dette resultatet om sammenhengen mellom matriseproduktet $\mathbf{y}R$ og radene i R? Formuler denne sammenhengen med ord.

b) Er dette en generell sammenheng som også gjelder for andre vektorer $\mathbf{y}$ og matriser C? Prøv i så fall å forklare denne sammenhengen ut ifra den komponentvise definisjonen av matrisemultiplikasjon.

Radvis matrisemultiplikasjon

Definer to matriser

```
>> D=[1 2 3; 4 5 6; 7 8 9];
>> E=[2 4; 1 3; 7 8];
```

og la MATLAB regne ut matriseproduktet

```
>> D*E
```

Bruk deretter radene i D som vektorer

```
>> d1=D(1,:);
>> d2=D(2,:);
>> d3=D(3,:);
```

og la MATLAB regne ut produktene

```
>> d1*E
>> d2*E
>> d3*E
```

Oppgave 5

a) Hva er sammenhengen mellom DE, $\mathbf{d}_1E$, $\mathbf{d}_2E$ og $\mathbf{d}_3E$?

b) Er dette en generell sammenheng som også gjelder for andre matriser D og E? Prøv i så fall å forklare denne sammenhengen ut ifra den komponentvise definisjonen av matrisemultiplikasjon.

Oppgave 6

Bruk resultatene fra oppgave 4 og 5 til å forklare hvordan man kan regne ut første rad i matriseproduktet DE ved hjelp av radene i E. Gjør det samme for andre og tredje rad i DE.

Bruk til slutt MATLAB til å sjekke at resultatet stemmer for matrisene D og E som vi definerte før oppgave 5.

Øvelse 2: Egenskaper ved determinanter

av

Klara Hveberg

I denne øvelsen skal vi bruke MATLAB til å studere hva elementære radoperasjoner gjør med determinanten til en matrise. Deretter skal vi se på determinanten til triangulære matriser, identitetsmatriser og permutasjonsmatriser.

Multiplisere en rad med en skalar

Lag en (4×4)-matrise

```
>> A=[2 4 6 8; 1 3 5 7; 9 8 7 6; 5 4 3 2]
```

og la MATLAB beregne determinanten

```
>> det(A)
```

Vi skal nå se hva som skjer med determinanten hvis vi erstatter første rad i A med 3 ganger seg selv:

```
>> A(1,:)=3*A(1,:)
>> det(A)
```

Hva skjedde med determinanten?

For å sjekke at dette resultatet ikke er en tilfeldighet knyttet til den spesielle matrisen A vi har valgt, skal vi la MATLAB generere tilfeldige (4×4)-matriser med heltallige komponenter mellom -10 og 10 ved kommandoen

```
>> A=round(20*rand(4)-10)
```

(Forklaring til kommandoen: `rand(4)` genererer en tilfeldig (4×4)-matrise hvor elementene er desimaltall mellom 0 og 1. Ved å multiplisere med 20, får vi elementer med heltallsdel mellom 0 og 20. Vi trekker deretter ifra 10 for å få elementer med heltallsdel mellom 10 og -10, og kommandoen `round` avrunder tallene til nærmeste heltall).

Oppgave 1

Bruk MATLAB til å generere en tilfeldig (4×4)-matrise, beregne determinanten og sammenligne denne med determinanten til matrisen som fremkommer ved å erstatte første rad med 3 ganger seg selv. Gjenta eksperimentet et par ganger.

a) Hva skjedde med determinanten til matrisene når du multipliserte første rad med 3? Gjelder dette også dersom du multipliserer en annen rad med 3? Hva skjer med determinanten hvis du isteden multipliserer raden med skalaren 8? Bruk disse observasjonene til å gjette på og formulere en regel for hva som skjer med determinanten til en generell kvadratisk matrise når vi multipliserer en vilkårlig rad med en skalar.

b) Hva tror du resultatet blir hvis du isteden multipliserer en søyle i A med en skalar? Du kan for eksempel multiplisere tredje søyle i A med 3 ved kommandoen

```
>> A(:,3)=3*A(:,3)
```

og beregne determinanten

```
>> det(A)
```

Bytte om to rader

La MATLAB generere en tilfeldig (4×4)-matrise med heltallige komponenter mellom 10 og -10 ved kommandoen

```
>> A=round(20*rand(4)-10)
```

og beregne determinanten

```
>> det(A)
```

Vi skal undersøke hva som skjer med determinanten hvis vi bytter om rad 2 og rad 4 i matrisen A

```
>> A=A([1,4,3,2],:) % Her vektorindekserer vi radene i den rekkefølgen
                    % vi vil ha dem (rad 2 og 4 har byttet plass)
>> det(A)
```

Hva skjedde med determinanten?

Vi skal nå sjekke om dette resultatet er en tilfeldighet som bare er knyttet til denne matrisen A.

Oppgave 2

Gjenta eksperimentet ovenfor et par ganger med nye tilfeldig genererte (4×4)-matriser.

a) Hva skjedde med determinanten til matrisene når du byttet om rad 2 og rad 4? Gjelder det samme hvis du isteden bytter om rad 1 og rad 3? Bruk disse observasjonene til å gjette på og formulere en regel for hva som skjer med determinanten til en generell kvadratisk matrise når vi bytter om på to vilkårlige rader i matrisen.

b) Hva tror du resultatet blir hvis du isteden bytter om to søyler i A? Du kan for eksempel bytte om søyle nr. 2 og 3 med kommandoen

```
>> A=A(:,[1,3,2,4])
```

og beregne determinanten

```
>> det(A)
```

Addere et skalarmultiplum av en rad til en annen rad

La MATLAB generere en tilfeldig (4×4)-matrise med heltallige komponenter mellom 10 og -10 ved kommandoen

```
>> A=round(20*rand(4)-10)
```

og beregne determinanten

```
>> det(A)
```

Vi skal nå se hva som skjer med determinanten hvis vi adderer 3 ganger andre rad til første rad (det vil si at vi erstatter første rad med summen av seg selv og 3 ganger andre rad)

```
>> A(1,:)=A(1,:)+3*A(2,:)
>> det(A)
```

Hva skjedde med determinanten?

Vil vil igjen sjekke at dette resultatet ikke er en tilfeldighet som bare er knyttet til denne matrisen A.

Oppgave 3

Gjenta eksperimentet ovenfor et par ganger med andre tilfeldig genererte (4×4)-matriser.

a) Hva skjedde med determinanten til matrisene når du adderte 3 ganger andre rad til første rad? Gjelder det samme også hvis du isteden adderer 5 ganger tredje rad til andre rad? Bruk disse observasjonene til å gjette på og formulere en regel for hva som skjer med determinanten til en generell kvadratisk matrise når vi adderer et vilkårlig skalarmultiplum av en rad til en annen rad.

b) Hva tror du resultatet blir hvis vi isteden adderer et skalarmultiplum av en søyle til en annen søyle i A? Du kan for eksempel addere 3 ganger fjerde søyle til andre søyle ved kommandoen

```
>> A(:,2)=A(:,2)+3*A(:,4)
```

og beregne determinanten

```
>> det(A)
```

Øvre triangulære matriser

Vi kaller en matrise *øvre triangulær* dersom alle elementene under hoveddiagonalen er null.

Vi kan generere en tilfeldig øvre triangulær (5×5)-matrise med heltallige komponenter i MATLAB ved kommandoen

```
>> U=triu(round(10*rand(5)+1)
```

og beregne determinanten

```
>> det(U)
```

(U står for "Upper diagonal matrix")

Deretter kan vi ta produktet av alle elementene på hoveddiagonalen til matrisen U ved hjelp av en `for`-løkke:

```
diag=U(1,1)
for i=2:5
  diag=diag*U(i,i)
end
```

Hva er sammenhengen mellom de to resultatene?

Oppgave 4

Gjenta eksperimentet ovenfor et par ganger med nye tilfeldig genererte øvre triangulære matriser. Bruk disse observasjonene til å gjette på og formulere en regel for hvordan vi lett kan beregne determinanten til en øvre triangulær matrise.

Nedre triangulære matriser

Vi kaller en matrise nedre triangulær dersom alle elementene over hoveddiagonalen er null.

Vi kan generere en tilfeldig nedre triangulær (5×5)-matrise med heltallige komponenter i MATLAB ved kommandoen

```
>> L=tril(round(10*rand(5)+1)
```

og beregne determinanten

```
>> det(L)
```

(L står for "Lower diagonal matrix")

Deretter kan vi ta produktet av alle elementene på hoveddiagonalen til matrisen L ved hjelp av en `for`-løkke:

```
diag=L(1,1)
for i=2:5
  diag=diag*L(i,i)
end
```

Hva er sammenhengen mellom de to resultatene?

Oppgave 5

Gjenta eksperimentet ovenfor et par ganger med nye tilfeldig genererte nedre triangulære matriser. Bruk disse observasjonene til å gjette på og formulere en regel for hvordan vi lett kan beregne determinanten til en nedre triangulær matrise.

Identitetsmatriser

En identitetsmatrise har 1-ere på hoveddiagonalen og nuller overalt ellers. I MATLAB kan vi generere en (5×5) identitetsmatrise ved kommandoen

```
>> A=eye(5)
```

og beregne determinanten

```
>> det(A)
```

Oppgave 6

a) Beregn determinanten til identitetsmatrisene av dimensjon (4×4) og (6×6). Bruk dette til å gjette på og formulere en regel for hva determinanten til en generell $(n \times n)$ identitetsmatrise er.

b) Forklar dette resultatet ved hjelp av resultatene vi kom frem til om determinanten til øvre og nedre triangulære matriser i forrige seksjon.

Permutasjonsmatriser

En permutasjonsmatrise er en kvadratisk matrise som inneholder nøyaktig en 1-er i hver rad og i hver søyle. Den fremkommer ved en ombytting (permutasjon) av radene i identitetsmatrisen.

Når vi skal lage en (5×5) permutasjonsmatrise i MATLAB, lager vi først en identitetsmatrise av samme størrelse

```
>> I=eye(5)
```

Deretter velger vi en permutasjon av radnumrene i denne matrisen (husk at en permutasjon av tallene fra 1 til 5 bare er en omstokking av disse tallene). Ønsker vi for eksempel å lage permutasjonsmatrisen hvor andre og tredje rad har byttet plass, stokker vi om på disse radene ved å skrive

```
>> P=I([1 3 2 4],:)
```

Vi kan få MATLAB til å generere en tilfeldig permutasjon av tallene fra 1 til 5 ved å skrive

```
>> randperm(5)
```

Vi kan altså generere en tilfeldig (5×5) permutasjonsmatrise P ved å bruke kommandoen

```
>> P=I(randperm(5),:)
```

og deretter regne ut determinanten

```
>> det(P)
```

Oppgave 7

Gjenta eksperimentet ovenfor flere ganger med nye tilfeldig genererte permutasjonsmatriser. Bruk disse observasjonene til å gjette på og prøve å formulere en regel for hva determinanten til en permutasjonsmatrise er.

Hint: Merk at P fremkommer fra identitetsmatrisen ved å bytte om to og to rader et visst antall ganger. Undersøk hva som skjer når antall slike ombyttinger er et partall og hva som skjer når det er et oddetall.

Bonusavsnitt for spesielt interesserte

Egenskapene vi kom frem i begynnelsen av denne labøvelsen, kan brukes til å definere hva determinanten $\det(A) = |A|$ til en kvadratisk $(n \times n)$-matrise $A = (a_{ij})$ er. Det er imidlertid også vanlig å definere determinanten ved hjelp av såkalte elementære produkter. Et elementært produkt $\prod_i a_{i\sigma_i}$ av elementer fra matrisen inneholder nøyaktig én faktor $a_{i\sigma_i}$ fra hver rad og hver søyle. Det betyr at søyleindeksene $\sigma_1, \sigma_2, \ldots, \sigma_n$ er en permutasjon ("omstokking") av linjeindeksene $1, 2, \ldots, n$. Hvert elementært produkt kan utstyres med et fortegn $\text{sign}(\sigma) = (-1)^{\text{inv}(\sigma)}$ som er bestemt av antall inversjoner $\text{inv}(\sigma)$ i permutasjonen σ, det vil si av antall par (σ_i, σ_j) av søyleindekser som opptrer i invertert ("omvendt") rekkefølge. Determinanten kan nå defineres som summen av alle elementære produkter forsynt med fortegn, det vil si at

$$\det(A) = \sum_{\sigma} \text{sign}(\sigma) \prod_{i=1}^{n} a_{i\sigma_i}$$

Oppgave 8 (ekstraoppgave)

Prøv å for forklare følgende egenskaper ved determinanten ut ifra definisjonen av determinanten som en sum av elementærprodukter forsynt med riktig fortegn:

a) regelen du kom frem til i oppgave 1a)

b) regelen du kom frem til i oppgave 2a)

c) regelen du kom frem til i oppgave 3a)

d) regelen du kom frem til i oppgave 4a)

e) regelen du kom frem til i oppgave 5a)

Appendiks 2

Løsningsforslag og hint

I denne seksjonen finner du løsningsforslag til labøvelsene og oppvarmingsøvelsene. Foran løsningsforslaget til labøvelse nr 3, finner du hint til denne labøvelsen som inneholder halvferdige programsnutter hvor du bare må fylle inn noen vesentlige linjer som tester den matematiske forståelsen av problemene.

Løsningsforslag til lab 1

Løsning til oppgave 1

a) Kvadratet blir dobbelt så stort. Multiplikasjon med A fordobler både x-koordinatene og y-koordinatene til punktene i `kvadrat`.

b) Kvadratet blir dobbelt så langt i x-retningen (så det blir et rektangel). Multiplikasjon med A fordobler alle x-koordinatene og lar y-koordinatene forbli uendret.

c) Kvadratet speiles om x-aksen. Multiplikasjon med A bytter fortegn på y-koordinaten til hvert punkt, mens x-koordinaten forblir uendret.

d) Kvadratet kollapser til et linjestykke (nedre sidekant) langs x-aksen. Multiplikasjon med A setter alle y-koordinatene til 0, mens x-koordinatene forblir uendret. Dette tilsvarer å projisere alle punktene ned på x-aksen.

e) Kvadratet blir deformert som om toppen dyttes mot høyre mens bunnen ligger i ro. Multiplikasjon med A forandrer ikke på y-koordinaten, men x-koordinaten får et tillegg som er halvparten av y-koordinaten (dvs at x-koordinaten får et større tillegg jo høyere opp i kvadratet punktet ligger). Dette er et eksempel på en såkalt skjær-avbildning.

f) Kvadratet roteres 45 grader (mot klokka). Multiplikasjon med A avbilder vektoren $(1, 0)$ på vektoren $(\sqrt{2}/2, \sqrt{2}/2)$ som også fremkommer ved å dreie vektoren $(1, 0)$ en vinkel på 45 grader mot klokka. Tilsvarende avbildes vektoren $(0, 1)$ på vektoren $(-\sqrt{2}/2, \sqrt{2}/2)$ som fremkommer ved å dreie vektoren $(0, 1)$ en vinkel på 45 grader mot klokka. Vi skal senere se at dette er et spesialtilfelle av en generell rotasjonsmatrise.

Løsning til oppgave 2

a) Vi må skrumpe figuren i begge retninger (halvere koordinatene)

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

b) Vi må skrumpe bare i x-retningen (halvere alle x-koordinater)

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

c) Vi må skrumpe bare i y-retningen (halvere alle y-koordinater)

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

Løsning til oppgave 3

a) Alle x-koordinatene må skifte fortegn (vi speiler om y-aksen)

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

b) Alle y-koordinatene må skifte fortegn (vi speiler om x-aksen)

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

Løsning til oppgave 4

a) Alle x-koordinatene må skifte fortegn (speiler om y-aksen)

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

b) Alle y-koordinatene må skifte fortegn (speiler om x-aksen)

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

c) Roter π radianer (180 grader)

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

d) På fila `speilomx.m` legger vi følgende kode:

```
function A = speilomx
% Returnerer matrise for speiling om x-aksen.
A = [ 1  0
      0 -1];
```

På fila `speilomy.m` legger vi følgende kode:

```
function A = speilomy
% Returnerer matrise for speiling om y-aksen.
A = [-1  0
      0  1];
```

På fila `roter180.m` legger vi følgende kode:

```
function A = roter180
% Returnerer rotasjonsmatrise for dreining 180 grader.
A= [-1  0
     0 -1];
```

e) Speiler vi MH om y-aksen får vi HM:

```
>> monogramHM = speilomy * monogramMH;
>> tegn(monogramHM)
```

Roterer vi HM 180 grader får vi WH:

```
>> monogramWH = roter180 * monogramHM;
>> tegn(monogramHM)
```

Speiler vi WH om y-aksen får vi HW:

```
>> monogramHW = speilomy * monogramWH;
>> tegn(monogramHW)
```

Roterer vi HW 180 grader får vi MH:

```
>> monogramMH = roter180 * monogramHW;
>> tegn(monogramMH)
```

f) Fra punkt b) vet vi at vi kan gå direkte fra MH til WH ved å speile om x-aksen. Og fra punkt e) vet vi at vi oppnår det samme ved først å speile MH om y-aksen og deretter rotere resultatet HM en vinkel på 180 grader. Disse observasjonene tyder på at det å speile om x-aksen gir samme resultat som det å først speile om y-aksen og deretter rotere 180 grader.

g) Matrisene vi får ved kommandoene

```
>> speilomx
```

og

```
>> speilomy*roter180
```

er like.
Forklaring: Sammensetning av avbildninger tilsvarer multiplikasjon av matrisene som representerer avbildningene.

h) Vi ser for eksempel at vi kan komme fra MH til HW på to forskjellige måter: Vi kan enten gå direkte fra MH til HW ved å rotere 180 grader, eller vi kan først speile MH om x-aksen slik at vi får WH, og deretter speile WH og y-aksen og få HW. Dette tyder på at det å rotere 180 grader gir samme resultat som først å speile om x-aksen og deretter speile om y-aksen. Bruk MATLAB til å sjekke at matrisene `speilomy*speilomx` og `roter180` er like.

Løsning til oppgave 5

a) i) Vi dreier pila 30 grader med klokka

```
A = [ cos(-pi/6)  -sin(-pi/6)
      sin(-pi/6)   cos(-pi/6)]
```

ii) Vi dreier pila 60 grader med klokka

```
B = [ cos(-pi/3)  -sin(-pi/3)
      sin(-pi/3)   cos(-pi/3)]
```

b) Å dreie viseren to timer frem er det samme som å dreie viseren en time frem to ganger etter hverandre. Siden vi vet at sammensetning av avbildninger tilsvarer matrisemultiplikasjon, kan vi altså benytte matrisen A^2 for å dreie viseren to timer frem.

c) På fila `roter.m` legger vi følgende funksjon:

```
function X = roter(v)
% Denne funksjonen tar som input en vinkel v
% og returnerer matrisen som roterer figuren en vinkel v.
X = [ cos(v)  -sin(v)
      sin(v)   cos(v)];
```

Vi tester denne ved å bruke kommandoene

```
>> tegn(roter((-pi)/6)*klokkepil)      % klokka viser ett
>> tegn(roter(5*(-pi)/6)*klokkepil)    % klokka viser fem
```

d) Som i punkt b) vet vi at det å dreie viseren t timer frem er det samme som å dreie viseren en time frem t ganger, og at sammensetning av avbildninger tilsvarer matrisemultiplikasjon. Vi kan derfor bruke matrisen A^t til å dreie klokkeviseren t timer frem ved å gi kommandoen

```
>> tegn(A^t*klokkepil)
```

Metoden med bruk av potenser krever flere regneoperasjoner enn metoden som baserer seg direkte på rotasjonsmatrisene, og forskjellen blir større jo større t er, siden antall regneoperasjoner som kreves for å beregne A^t vokser eksponentielt med t, mens det å beregne $\sin(t)$ og $\cos(t)$ ikke blir noe mer komplisert når t vokser.

Løsning til oppgave 6

På fila tikkendeklokke.m legger vi følgende skript:

```
% Skript: tikkendeklokke
% Viser en timeklokke som går en runde (1 time per sekund).
% Henter inn en ferdig opptegnet klokkeskive generert av skriptet
% klokkeskive.m
% Plottepunktene for en timeviser som peker rett opp (kl 12) er
% definert i en matrise som genereres av skriptet klokkepil.m
% Denne timeviseren roteres vha en rotasjonsmatrise og tegnes opp
% i samme figur som klokkeskiven ved hjelp av funksjonen tegn.m
for i=1:13
   klokkeskive                % skript som tegner opp klokkeskiven
   hold on
   tegn(roter((i-1)*(-pi/6))*klokkepil) % tegner den roterte viseren
   hold off
   pause(1)                   % venter 1 sek før neste figur tegnes
end
```

Løsning til oppgave 7 (ekstraoppgave)

Vi kan lage en klokkeanimasjon ved å bruke kommandoene

```
>> for i=1:13
      klokkeskive
      hold on
      tegn(roter((i-1)*(-pi)/6)*klokkepil)
      F(i)=getframe;
      hold off
   end
```

Vi kan nå spille av klokkeanimasjonen (hvor timeviseren beveger seg med en fart på en time i sekundet) tre ganger på rad med kommandoen

```
>> movie(F,3,1)
```

Løsningsforslag til lab 2

Løsning til oppgave 1

På filen `gaussA.m` legger vi følgende skript:

```
A=[6 -2 0; 9 -1 1; 3 7 5];
  for j=1:2              % gjennomgår en og en søyle (unntatt siste)
    for i=j+1:3          % gjennomgår en og en rad nedenfor pivoten
      k = A(i,j)/A(j,j); % finner riktig multiplum å trekke fra
      A(i,:) = A(i,:) - k*A(j,:); % nuller ut elementene under pivoten
    end % for i
  end % for j
```

Vi sjekker at vi får samme resultat som da vi utførte skrittene manuelt:

```
>> gaussA
>> A
```

Løsning til oppgave 2

På filen `gauss.m` legger vi følgende kode:

```
function U = gauss(A)
% Fil: gauss.m
% Funksjon som tar en kvadratisk matrise A som inn-parameter, utfører
% Gauss-eliminasjon uten pivotering på denne matrisen og
% returnerer en matrise U på trappeform.
% Funksjonen fungerer bare på matriser som kan bringes på trappeform
% uten at vi støter på null-elementer i pivot-posisjoner underveis.
[m,n]=(size(A));
% Antar at matrisen er kvadratisk slik at m=n.
% Antar også at vi aldri støter på null-elementer i pivot-posisjonene,
% så pivot-elementene er A(j,j) =0 på hoveddiagonalen.
  for j=1:n-1            % gjennomgår en og en søyle
    for i=j+1:m          % gjennomgår en og en rad nedenfor pivoten
      k = A(i,j)/A(j,j); % finner riktig multiplum å trekke fra
      A(i,:) = A(i,:) - k*A(j,:); % nuller ut elementene under pivoten
    end % for i
  end % for j
  U=A;
```

Løsning til oppgave 3

På filen `gauss_delpiv.m` legger vi f.eks. følgende programkode:

Alternativ 1: Funksjon som bruker `for`-løkke og `break`

```
function U = gauss_delpiv(A);
% Fil: gauss_delpiv.m
% Funksjon som tar en matrise A som inn-parameter, utfører
% Gauss-eliminasjon med delvis pivotering på denne matrisen og
% returnerer matrisen på trappeform U.
 [m,n]=size(A)
  i=1;
```

```
  for j=1:n
    [maksverdi,p] = max(abs(A(i:m,j)));
    q = p+i-1;
    A([i q],:) = A([q i],:);
    if A(i,j)  = 0
      for r=i+1:m
        k = A(r,j)/A(i,j);
        A(r,:) = A(r,:) - k*A(i,:);
      end  % for r
      i=i+1;
      if i>m
        break
      end % if i
    end % if
  end % for j
  U=A;
```

Alternativ 2: Funksjon som bruker `while`-løkke.

På filen `gauss_delpiv.m` legger vi følgende kode:

```
function U = gauss_delpiv(A);
% Fil: gauss_delpiv.m
% Funksjon som tar en matrise A som inn-parameter, utfører
% Gauss-eliminasjon med delvis pivotering på denne matrisen og
% returnerer matrisen på trappeform U.
% Rutinen ligner Gauss-eliminasjon uten pivotering, bortsett fra at vi
% i hvert steg først velger det maksimale elementet i søylen som pivot.
  [m,n]=size(A)
  i=1;
  j=1;
  while (i<=m) & (j<=n)
    [maksverdi,p] = max(abs(A(i:m,j)));
    q = p+i-1;
    A([i q],:) = A([q i],:);
    if A(i,j)  = 0
      for r=i+1:m
        k = A(r,j)/A(i,j);
        A(r,:) = A(r,:) - k*A(i,:);
      end  % for r
      i=i+1;
      j=j+1;
    else % her er A(i,j)=0
      j=j+1;
    end % if
  end % while
  U=A;
```

Bemerkning: I MATLAB er det generelt tidsbesparende å unngå `for`-løkker dersom man kan bruke vektor-indeksering isteden. I rutinene ovenfor kunne vi erstattet den innerste `for`-løkken (4 linjer) med følgende to linjer:

```
    k(i+1:m,j) = A(i+1:m,j)/A(i,j);
    A(i+1:m,j:n) = A(i+1:m,j:n) - k(i+1:m,j)*A(i,j:n);
```

Løsning til oppgave 4

På fila `gauss_jordan.m` legger vi f.eks følgende kode (vi modifiserer her funksjonen med `while`-løkke fra oppgave 3, men kunne like godt brukt funksjonen med `for`-løkke):

```
function U = gauss_jordan(A);
% Fil: gauss_jordan.m
% Funksjon som tar en matrise A som inn-parameter, utfører
% Gauss-Jordan-eliminasjon med delvis pivotering på denne matrisen og
% returnerer matrisen på redusert trappeform U.
  [m,n]=size(A)
  i=1;
  j=1;
  while (i<=m) & (j<=n)
    [maksverdi,p] = max(abs(A(i:m,j)));
    q = p+i-1;
    A([i q],:) = A([q i],:);
    if A(i,j)  = 0
    A(i,:) = A(i,:)/A(i,j);  % skaffer ledende 1-er i pivot-raden
      for r=1:m              % går gjennom alle radene
        if r =i              % bortsett fra rad nr i (pivot-raden)
          k = A(r,j);
          A(r,:) = A(r,:) - k*A(i,:); % nuller ut elementet i søyle j
        end %if
      end % for r
      i=i+1;
      j=j+1;
    else % her er A(i,j)=0
      j=j+1;
    end % if
  end % while
  U=A;
```

Utvidet løsningsforslag til oppgave 3 med flere kommentarer

```
function U = gauss_delvpiv(A);
[m,n]=size(A) % leser inn antall rader m og antall søyler n i matrisen A
  i=1;  % Vi starter i første rad og setter derfor radindeksen 'i' lik 1
% Vi skal nå gjennomgå en og en søyle:
  for j=1:n
% For hver søyle j skal vi finne elementet med størst absoluttverdi
% i denne søylen fra rad nr i og nedover til rad m, dvs blant elementene
% i A(i:m,j). Verdien av det maksimale elementet returneres i variabelen
% maksverdi, mens antall rader vi måtte gå nedover fra rad nr i for å
% finne det maksimale elementet, returneres i variabelen p.
% (Dersom det fins flere like store maksimale elementer i en søyle,
% velger MATLAB den første raden som inneholder et slikt maksimalt
% element.)
    [maksverdi,p] = max(abs(A(i:m,j)));
% Nå kan vi beregne radnummeret q til raden hvor det maksimale elementet
% i søyle nr j befinner seg ved å justere indeksen 'p' (vi må legge til
% (i-1) rader siden letingen etter maksimalelementet begynte i rad nr i)
    q = p+i-1;
% Så flytter vi rad nr q med det maksimale elementet opp ved å bytte om
% rad q og rad i (den øverste raden vi holder på med i dette trinnet)
    A([i q],:) = A([q i],:);
% Vi har nå fått det maksimale elementet i søyle nr j som vårt nye
% pivot-element A(i,j). Resten av prosedyren er den samme som før:
% Vi ønsker å nulle ut alle elementene under pivot-elementet i søyle j.
% Hvis pivot-elementet A(i,j) vi nå har fått er forskjellig fra null,
% skal vi først dele alle rader nedenfor med dette pivot-elementet for
% å finne det riktige multiplumet av rad nr i som vi må trekke fra den
% aktuelle raden for å nulle ut elementet under pivot-elementet i søyle
% nr j. Deretter trekker vi fra det riktige multiplumet av rad nr i.
    if A(i,j)  = 0
      for r=i+1:m
        k = A(r,j)/A(i,j);
        A(r,:) = A(r,:) - k*A(i,:);
      end % for r
% Vi skal nå gå videre til neste søyle og lete etter et pivot-element i
% neste rad. Vi øker derfor radindeksen 'i' med 1. Hvis radindeksen blir
% større enn totalt antall rader i matrisen, avbryter vi prosessen.
      i=i+1;
      if i>m
         break
      end
    end % if A(i,j)
% Hvis elementet i pivot-posisjonen (i,j) er null, vet vi (siden det
% var det maksimale elementet i søylen) at alle elementer under også er
% null.
% Vi skal da bare gå videre til neste søyle uten å øke radindeksen.
  end % for j
U=A;
```

Hint til lab 3

Hint til oppgave 1

Du kan godt løse oppgaven ved å bruke en `for`-løkke og `break`, men vi velger å gi en programskisse som benytter en `while`-løkke. Her trenger du bare å erstatte innmaten i de to parantesene <gi brukeren beskjed om søylene er lineært avhengige eller uavhengige> og <betingelse oppfylt> med riktig MATLAB-kode.

```
function linavh(A)
% Fil: linavh.m
% Funksjon som tar en matrise som inn-parameter og sjekker om
% søylene i matrisen er lineært avhengige eller uavhengige.
[m,n]=size(A);
if m<n
  <gi brukeren beskjed om søylene er lineært avhengige eller uavhengige>
else
  U=rref(A)
    uavh=true;
    j=1;
    while uavh==true & j<=n
      if <betingelse oppfylt>
        disp('Søylevektorene er lineært avhengige')
        uavh=false;
      end
      j=j+1;
    end
  if uavh==true
    disp('Søylevektorene er lineært uavhengige')
  end
end
```

Hint til oppgave 2

Du kan godt løse oppgaven ved å bruke en `for`-løkke og `break`, men vi velger å gi en programskisse som benytter en `while`-løkke. Her trenger du bare å erstatte innmaten i parantesen <betingelser oppfylt> med riktig MATLAB-kode.

```
function inkons(A,b)
% Fil: inkons.m
% Funksjon som tar en matrise A og en vektor b som inn-parameter, og
% sjekker om ligningssystemet A*x=b er konsistent eller inkonsistent.
[m,n]=size(A);
if length(b) =m
  disp('Vektoren du tastet inn har feil dimensjon')
else
  Aug=[A,b]; % utvidet matrise med dimensjon m*(n+1)
  U=rref(Aug)
  konsistent=true;
  i=1;
  while konsistent==true & i<=m
```

```
    if <betingelser oppfylt>
      disp('Ligningssystemet er inkonsistent')
      konsistent=false;
    end
    i=i+1;
  end
  if konsistent==true
    disp('Ligningssystemet er konsistent')
  end
end
```

Hint til oppgave 3

Du kan godt løse oppgaven ved å bruke en `for`-løkke og `break`, men vi velger å gi en programskisse som benytter en `while`-løkke. Her trenger du bare å erstatte innmaten i de fire parantesene <betingelse oppfylt>, <betingelser oppfylt>, <gi brukeren beskjed om antall løsninger> og <gi variabelen `entydig` riktig sannhetsverdi>, med riktig MATLAB-kode.

```
function antlosn(A,b)
% Fil: antlosn.m
% Funksjon som tar en matrise A og en vektor b som inn-parameter, og
% sjekker om ligningssystemet A*x=b er inkonsistent eller konsistent.
% Dersom ligningssystemet er konsistent, sjekker funksjonen om det
% har uendelig mange løsninger eller en entydig løsning.
[m,n]=size(A);
if length(b) =m
  disp('Vektoren du oppga har feil dimensjon i forhold til matrisen')
else
  Aug=[A,b]; % utvidet matrise med dimensjon m*(n+1)
  U=rref(Aug)
  i=1;
  konsistent=true;
  while konsistent == true & i<=m
    if <betingelser oppfylt>
      disp('Ligningssystemet er inkonsistent')
      konsistent=false;
    end
    i=i+1;
  end
  if konsistent==true
    disp('Ligningssystemet er konsistent')
     entydig=true;
     if m<n
       <gi brukeren beskjed om antall løsninger>
       <gi variabelen entydig riktig sannhetsverdi>
     else
       j=1;
       while entydig==true & (j<=n) % trenger ingen radindeks siden n<=m
         if <betingelse oppfylt>
           disp('Systemet har uendelig mange løsninger')
```

```
          entydig=false;
        end
        j=j+1;
      end
      if entydig==true
        disp('Ligningssystemet har nøyaktig en løsning')
      end
    end % if m<n
  end % if konsistent
end % if length
```

Løsningsforslag til lab 3

Løsning til oppgave 1

Søylene i en matrise A er lineært avhengige hvis og bare hvis det tilsvarende homogene ligningssystemet $A\mathbf{x} = \mathbf{0}$ har uendelig mange løsninger. Dette vil være tilfelle dersom det finnes frie variable, dvs dersom det finnes pivot-frie søyler. Dette vil opplagt finnes hvis matrisen har flere søyler enn rader, dvs hvis $m < n$. Hvis $m \geq n$ kan vi teste om det finnes pivot-frie søyler ved å undersøke om det finnes elementer på diagonalen som er null i den reduserte trappeformen til matrisen A. Det første null-elementet vi finner hvis vi leter nedover diagonalen, må stå i en pivot-fri søyle. (Vær sikker på at du skjønner hvorfor! Dette betyr ikke at vi kan finne alle pivot-frie søyler bare ved å lete etter nuller på diagonalen! Men siden vi bare er interessert i om det finnes noen pivot-fri søyle, er det nok å finne den første.)

Vi gir først et løsningsalternativ som bruker en `while`-løkke:

```
function linavh(A)
% Fil: linavh.m
% Funksjon som tar en matrise som inn-parameter og sjekker om
% søylene i matrisen er lineært avhengige eller uavhengige.
% Søylene er lineært avhengige hvis det tilsvarende homogene
% ligningssystemet har uendelig mange løsninger, dvs dersom det
% finnes frie variabler. Dette tilsvarer at det finnes pivot-frie
% søyler, noe som vil være tilfelle hvis m<n eller hvis det finnes
% elementer på diagonalen som er null i den reduserte trappeformen.
[m,n]=size(A);
if m<n
  disp('Søylevektorene er lineært avhengige pga flere søyler enn rader')
else
  U=rref(A)
    uavh=true;
    j=1;
    while uavh==true & j<=n
      if U(j,j)==0 % Sjekker om det fins et null-element på diagonalen
        disp('Søylevektorene er lineært avhengige')
        uavh=false;
      end
      j=j+1;
    end
  if uavh==true
    disp('Søylevektorene er lineært uavhengige')
  end
end
```

Alternativ løsning med `for`-løkke og `break`

```
function linavh(A)
% Fil: linavh.m
% Funksjon som tar en matrise som inn-parameter og sjekker om
% søylene i matrisen er lineært avhengige eller uavhengige.
```

```
[m,n]=size(A);
if m<n
  disp('Søylevektorene er lineært avhengige pga flere søyler enn rader')
else
  U=rref(A)
    uavh=true;
    for j=1:n
      if U(j,j)==0 % Sjekker om det fins et null-element på diagonalen
        disp('Søylevektorene er lineært avhengige')
        uavh=false;
        break;
      end
    end
  if uavh==true
    disp('Søylevektorene er lineært uavhengige')
  end
end
```

Løsning til oppgave 2

Et ligningssystem av typen $A\mathbf{x} = \mathbf{b}$ er konsistent hvis og bare hvis den reduserte trappeformen til den utvidete matrisen (augmented matrix) $[A, \mathbf{b}]$ ikke har en rad på formen $[\mathbf{0}, \mathbf{0}, \ldots, \mathbf{0}, \mathbf{c}]$ hvor $\mathbf{c}$ er forskjellig fra null.

Vi gir først et løsningsalternativ som bruker en `while`-løkke:

```
function inkons(A,b)
% Fil: inkons.m
% Funksjon som tar en matrise A og en vektor b som inn-parameter, og
% sjekker om ligningssystemet A*x=b er konsistent eller inkonsistent.
% Husk at et slikt ligningssystem er konsistent hvis og bare hvis den
% reduserte trappeformen til den utvidete matrisen ikke har en rad på
% formen [0 0 ...0 c] hvor c er forskjellig fra null.
[m,n]=size(A);
if length(b) =m
  disp('Vektoren du tastet inn har feil dimensjon')
else
  Aug=[A,b]; % utvidet matrise med dimensjon m*(n+1)
  U=rref(Aug)
  konsistent=true;
  i=1;
  while konsistent==true & i<=m
    if (U(i,1:n)==zeros(1,n)) & (U(i,n+1) =0)
      disp('Ligningssystemet er inkonsistent')
      konsistent=false;
    end
    i=i+1;
  end
  if konsistent==true
    disp('Ligningssystemet er konsistent')
  end
end
```

Alternativ løsning med `for`-løkke og `break`

```
function inkons(A,b)
% Fil: inkons.m
% Funksjon som tar en matrise A og en vektor b som inn-parameter, og
% sjekker om ligningssystemet A*x=b er konsistent eller inkonsistent.
[m,n]=size(A);
if length(b) =m
  disp('Vektoren du tastet inn har feil dimensjon')
else
  Aug=[A,b]; % utvidet matrise med dimensjon m*(n+1)
  U=rref(Aug)
  konsistent=true;
  for i=1:m
    if (U(i,1:n)==zeros(1,n)) & (U(i,n+1) =0)
      disp('Ligningssystemet er inkonsistent')
      konsistent=false;
      break;
    end
  end
  if konsistent==true
    disp('Ligningssystemet er konsistent')
  end
end
```

Løsning til oppgave 3

Et konsistent ligningssystem $A\mathbf{x} = \mathbf{b}$ har uendelig mange løsninger dersom det finnes frie variabler, dvs dersom det finnes søyler som ikke har pivot-elementer. Dette vil være tilfelle hvis matrisen har flere søyler enn rader, eller hvis det finnes null-elementer på diagonalen.

På samme måte som i de foregående oppgavene kan du godt bruke en `for`-løkke og `break`, men vi gir her et forslag som benytter `while`-løkker:

```
function antlosn(A,b)
% Fil: antlosn.m
% Funksjon som tar en matrise A og en vektor b som inn-parameter og
% sjekker om ligningssystemet A*x=b er inkonsistent eller konsistent.
% Dersom ligningssystemet er konsistent, sjekker funksjonen om det
% har uendelig mange løsninger eller en entydig løsning.
% Husk at systemet har uendelig mange løsninger dersom det finnes
% frie variabler, dvs dersom det finnes søyler som ikke har
% pivot-elementer. Dette vil være tilfelle hvis m<n eller hvis det
% finnes null-elementer på diagonalen A(j,j).
[m,n]=size(A);
if length(b) =m
  disp('Vektoren du oppga har feil dimensjon i forhold til matrisen')
else
  Aug=[A,b]; % utvidet matrise med dimensjon m*(n+1)
  U=rref(Aug)
% Sjekker om ligningssystemet er inkonsistent ved å lete etter
```

```
% en rad på formen [0 0 ... 0 c] hvor c er forskjellig fra null.
  i=1;
  konsistent=true;
  while konsistent == true & i<=m
    if (U(i,1:n)==zeros(1,n)) & (U(i,n+1) =0)
      disp('Ligningssystemet er inkonsistent')
      konsistent=false;
    end
    i=i+1;
  end
  if konsistent==true
    disp('Ligningssystemet er konsistent')
    entydig=true;
    if m<n                    % det må finnes pivot-frie søyler
      disp('Systemet har uendelig mange løsninger')
      entydig=false;
    else
      j=1;
      while entydig==true & (j<=n)  % trenger ingen radindeks siden n<=m
        if U(j,j)==0          % det må finnes pivot-frie søyler
          disp('Systemet har uendelig mange løsninger')
          entydig=false;
        end
        j=j+1;
      end
      if entydig==true
        disp('Systemet har nøyaktig en løsning')
      end
    end %if m<n
  end % if konsistent
end % if length
```

Løsning til oppgave 4

a) Resultatet av programkjøringen bør være:

```
>> linavh(A)
U =
     1     0     0
     0     0     1
     0     0     0
Søylevektorene er lineært avhengige
>> linavh(B)
U =
     1     0     0     0
     0     1     0     2
     0     0     1     0
     0     0     0     0
Søylevektorene er lineært avhengige
>> linavh(C)
Søylevektorene er lineært avhengige pga flere søyler enn rader
```

```
>> linavh(D)
U =
     1     0     0     0
     0     1     0     0
     0     0     1     0
     0     0     0     1
     0     0     0     0
Søylevektorene er lineært uavhengige
>> linavh(E)
Søylevektorene er lineært avhengige pga flere søyler enn rader
```

b) Resultatet av programkjøringene bør være:

```
>> inkons(A,a)
U =
     1     0     0     0
     0     0     1     0
     0     0     0     1
Ligningssystemet er inkonsistent
>> inkons(B,b)
U =
     1     0     0     0     0
     0     1     0     2     0
     0     0     1     0     0
     0     0     0     0     1
Ligningssystemet er inkonsistent
>> inkons(C,b)
U =
    1.0000         0         0         0         0    0.9600
         0    1.0000         0    2.0000         0   -1.4800
         0         0    1.0000         0         0    1.6800
         0         0         0         0    1.0000   -2.0400
Ligningssystemet er konsistent
>> antlosn(C,b)
U =
    1.0000         0         0         0         0    0.9600
         0    1.0000         0    2.0000         0   -1.4800
         0         0    1.0000         0         0    1.6800
         0         0         0         0    1.0000   -2.0400
Ligningssystemet er konsistent
Systemet har uendelig mange løsninger
>> inkons(D,d)
U =
     1     0     0     0     0
     0     1     0     0     2
     0     0     1     0     0
     0     0     0     1     0
     0     0     0     0     0
Ligningssystemet er konsistent
>> antlosn(D,d)
```

```
U =
     1     0     0     0     0
     0     1     0     0     2
     0     0     1     0     0
     0     0     0     1     0
     0     0     0     0     0
Ligningssystemet er konsistent
Systemet har nøyaktig en løsning
>> inkons(E,d)
U =
    1.0000         0         0         0         0   -1.0308   -1.2568
         0    1.0000         0         0         0   -0.5992   -0.8876
         0         0    1.0000         0         0    1.6646    1.2700
         0         0         0    1.0000         0    0.3321    1.9416
         0         0         0         0    1.0000    0.2866   -0.0166
Ligningssystemet er konsistent
>> antlosny(E,d)
U =
    1.0000         0         0         0         0   -1.0308   -1.2568
         0    1.0000         0         0         0   -0.5992   -0.8876
         0         0    1.0000         0         0    1.6646    1.2700
         0         0         0    1.0000         0    0.3321    1.9416
         0         0         0         0    1.0000    0.2866   -0.0166
Ligningssystemet er konsistent
Systemet har uendelig mange løsninger
```

Løsningsforslag til lab 4

Løsning til oppgave 1

a) Vi bruker følgende `for`-løkke:

```
>> x=1;
>> for t=1:5
    xny=x-(x^3+2*x^2-2)/(3*x^2+4*x);
    x=xny
   end
```

MATLAB svarer da med å skrive ut alle tilnærmingsverdiene den regner ut:

```
x =
    0.8571
x =
    0.8395
x =
    0.8393
x =
    0.8393
x =
    0.8393
```

Hvis vi bruker kommandoen `>> format long` før vi kjører løkken, får vi se flere desimaler i tilnærmingsverdiene:

```
x =
   0.85714285714286
x =
   0.83954451345756
x =
   0.83928681006802
x =
   0.83928675521416
x =
   0.83928675521416
```

Det ser ut som om tilnærmingene konvergerer mot verdien $x = 0.83928675521416$. For å teste om funksjonsverdien i dette punktet er nær null, gir vi kommandoen

```
>> x^3+2*x^2-2
ans =
    -2.220446049250313e-16
```

Funksjonsverdien i punktet er altså svært nær null.

b) Etter tre iterasjoner forandrer ikke de fire første desimalene i tilnærmingen seg lenger. Etter fire iterasjoner forandrer ikke de tolv første desimalene seg lenger.

Løsning til oppgave 2

a) På m-filen `newton.m` legger vi følgende MATLAB-kode:

```
function x = newton(start,fogdf)
% MATLAB-funksjon som utfører Newtons metode på en funksjon
% f med derivert df spesifisert på filen 'fogdf.m'.
% Innparametre:
% start: startpunkt for iterasjonen
% fogdf: navn på MATLAB-funksjon som returnerer vektor [f,df]
% hvor f er funksjonen vi skal benytte Newtons metode på
% mens df er den deriverte av denne funksjonen
maxit=30;
tol=0.0000001;
diff=tol+1;
x=start;
[f,df]=feval(fogdf,x);
for t=1:maxit
  xny=x-f/df;
  diff=abs(xny-x);
  x=xny;
  [f,df]=feval(fogdf,x);
  fprintf('itnr=%2d  x=%15.12f  f(x)=%15.12f\n',t,x,f)
  if diff<tol
    break;
  end %if
end %for
% Kalles f.eks med kommandoen:
% >> x=newton(1,'fogd')
% NB! Merk apostrofene rundt navnet på MATLAB-funksjonen
% som sendes med som innparameter
```

b) For å kunne bruke MATLAB-funksjonen `newton` på funksjonen $f(x) = x^3 + 2x^2 - 2$, lager vi en m-fil `fogder.m` hvor vi spesifiserer $f(x)$ og dens deriverte $f'(x) = 3x^2+4x$ på følgende måte

```
function [y,d]=fogder(x)
% returnerer funksjonsverdien til f(x)=x^3+2*x^2-2 i punktet x
% sammen med den deriverte f'(x)=3*x^2+4*x i det samme punktet
y=x^3+2*x^2-2;
d=3*x^2+4*x;
```

Vi kjører MATLAB-funksjonen `newton` på funksjonen f med startpunkt $x_0 = 1$ ved å gi kommandoen

```
>> x=newton(1,'fogder')
```

Hvis du har brukt den anbefalte kommandoen `fprintf` for å få pen utskrift på skjermen, vil MATLAB nå svare med følgende utskrift

```
itnr= 1  x= 0.857142857143  f(x)= 0.099125364431
itnr= 2  x= 0.839544513458  f(x)= 0.001410328965
itnr= 3  x= 0.839286810068  f(x)= 0.000000300070
itnr= 4  x= 0.839286755214  f(x)= 0.000000000000
x =
   0.83928675521416
```

c) Først plotter vi funksjonen $f(x) = x^3 + 2x^2 - 30x - 5$ over intervallet $[-8, 6]$ ved å gi følgende kommandoer:

```
>> x=[-8:0.1:6];
>> y=x.^3+2.*x.^2-30.*x-5; %Husk punktum for elementvise operasjoner
>> plot(x,y)
```

For å kunne bruke MATLAB-funksjonen `newton` på funksjonen $f(x) = x^3 + 2x^2 - 30x - 5$, lager vi en m-fil `fogd.m` hvor vi spesifiserer $f(x)$ og dens deriverte $f'(x) = 3x^2 + 2x - 30$ på følgende måte

```
function [y,d]=fogd(x)
% returnerer funksjonsverdien til f(x)=x^3+2*x^2-30*x-5 i punktet x
% sammen med den deriverte f'(x)=3*x^2+2*x-30 i det samme punktet
y=x^3+2*x^2-30*x-5;
d=3*x^2+2*x-30;
```

Vi kjører MATLAB-funksjonen `newton` på funksjonen f med startpunkt $x_0 = 1$ ved å gi kommandoen

```
>> x=newton(1,'fogd')
itnr= 1  x=-0.280000000000  f(x)= 3.534848000000
itnr= 2  x=-0.163433757189  f(x)=-0.047931510926
itnr= 3  x=-0.165018440949  f(x)= 0.000521768852
itnr= 4  x=-0.165001191448  f(x)=-0.000005692524
itnr= 5  x=-0.165001379641  f(x)= 0.000000062104
itnr= 6  x=-0.165001377588  f(x)=-0.000000000678
x =
  -0.16500137758754
```

Vi ser at funksjonsverdien i tilnærmingspunktet er svært nær null, så dette er en god tilnærming til et nullpunkt for funksjonen.

Ved å skifte ut startpunktet med hhv 4 og -4, får vi på tilsvarende måte tilnærmingsverdier $x = 4.66323383636907$ og $x = -6.49823244654976$ for de to andre nullpunktene til funksjonen.

d) For å studere funksjonen $g(x) = x^3 - 5x$ lager vi først et plott av funksjonen over intervallet $[-3, 3]$ ved å gi følgende kommandoer:

```
>> x=[-3:0.1:3];
>> y=x.^3-5.*x;
>> plot(x,y)
```

For å kunne bruke MATLAB-funksjonen `newton` på funksjonen $g(x) = x^3 - 5x$, lager vi en m-fil `gogd.m` hvor vi spesifiserer $g(x)$ og dens deriverte $g'(x) = 3x^2 - 5$ på følgende måte

```
function [y,d]=gogd(x)
% returnerer funksjonsverdien til g(x)=x^3-5*x i punktet x
% og den deriverte g'(x)=3*x^2-5 i samme punkt
y=x^3-5*x;
d=3*x^2-5;
```

Vi kjører MATLAB-funksjonen `newton` på funksjonen g med startpunkt $x_0 = 1$ ved å gi kommandoen

```
>> x=newton(1,'gogd')
```

Vi ser da at tilnærmingsverdiene hopper mellom 1 og -1 (går i sykel), og nærmer seg altså ikke noe nullpunkt for funksjonen (funksjonsverdiene i de to punktene er hhv -4 og 4). Årsaken til denne oppførselen er at tangenten til grafen over startpunktet $x = 1$ skjærer x-aksen i punktet $x = -1$ (som altså blir neste tilnærmingspunkt), mens tangenten til grafen over punktet $x = -1$ skjærer x-aksen i punktet $x = 1$ (slik at det neste tilnærmingspunktet blir lik starpunktet). Dermed fører Newtons metode oss bare i en sykel mellom disse to punktene.

Når vi etterpå prøver med startpunktet $x_0 = 2$, konvergerer tilnærmingsverdiene pent mot et tilnærmet nullpunkt $x = 2.23606797749979$ for funksjonen. Oppgaven illustrerer altså hvor stor betydning valg av startpunkt kan ha for om Newtons metode konvergerer eller ikke: velger vi startpunktet $x_0 = 1$ vil tilnærmingsverdiene gå inn i en sykel og ikke konvergere mot noe nullpunkt, men velger vi startpunktet $x_0 = 2$, får vi rask konvergens mot et nullpunkt.

Vi lager et plott av funksjonen $h(x) = x^{\frac{1}{3}}$ over intervallet $[-4, 4]$ ved å gi følgende kommandoer:

```
>> x=[-4:0.05:4];
>> y=sign(x).*abs(x).^(1./3);
>> plot(x,y)
```

For å bruke MATLAB-funksjonen `newton` på funksjonen $h(x) = x^{\frac{1}{3}}$ lager vi en m-fil `hogd.m` hvor vi spesifiserer $h(x)$ og dens deriverte $h'(x) = \frac{1}{3}x^{-\frac{2}{3}}$ på følgende måte

```
function [y,d]=hogd(x)
% returnerer funksjonsverdien til h(x)=x^(1/3) i punktet x
% og den deriverte h'(x)=(1/3)*x^(-2/3) i samme punkt
y=sign(x)*abs(x)^(1/3);
d=(1/3)*abs(x)^(-2/3);
```

Vi kjører MATLAB-funksjonen `newton` på funksjonen h med startpunkt $x_0 = 1$ og ved å gi kommandoen

```
>> x=newton(1,'hogd')
```

Her vil tilnærmingsverdiene divergere lenger og lenger bort fra nullpunktet til funksjonen. Årsaken til denne oppførselen er at tangenten til grafen over startpunktet $x_0 = 1$, er så slak at det neste tilnærmingspunktet x_1 (som ligger der hvor tangenten krysser x-aksen) blir liggende lenger borte fra nullpunktet origo enn x_0 gjorde. Tangenten i det nye tilnærmingspunktet x_1 er enda slakere, og fører oss derfor enda lenger bort fra origo for å finne neste tilnærmingsverdi x_2 osv.

Løsning til oppgave 3

a) På filen kompleks.m legger vi følgende MATLAB-skript:

```
% Skript som genererer et bilde av hvilke punkter i planet som
% gir konvergens mot samme nullpunkt når vi bruker Newtons metode
% på den komplekse funksjonen f(z)=z^3+a*z+b
% Punktene i planet tilordnes en farge som angir hvilket av
% nullpunktene de gir konvergens mot.
% Vi lager først et m*n-gitter av punkter i planet, og deretter
```

```
% genererer vi en m*n-matrise T som inneholder fargeverdiene hvert av
% punktene skal ha.
% Fargeverdien i et punkt finner vi ved å bruke Newtons metode på
% punktet, og se om resultatet havner nær ett av nullpunktene for
% funksjonen. Vi gir de punktene som konvergerer mot samme nullpunkt,
% den samme fargen.
% Lager vektorer som inneholder hhv real- of imaginærdelen til
% punktene i gitteret vi skal studere
x=[-2:0.01:2];
y=[-2:0.01:2];
[X,Y]=meshgrid(x,y);
[m,n]=size(X);
maxit=30; % maksimalt antall iterasjoner som skal utføres
tol=0.001; %toleranse
T=27*ones(m,n); %initialiserer fargene til å være 27 (turkis)
% Velger parametere til funksjonen f(z)=z^3+a*z+b
a=0;
b=-1;
% Vi angir koeffisientene foran potensene av z i polynomet z^3+a*x+b:
p=[1 0 a b];
% Vi lar MATLAB finne numeriske verdier for de tre nullpunktene til
% polynomet:
rts=roots(p);
% Så gjennomløper vi ett og ett punkt i gitteret ved en dobbelt
% for-løkke:
for j=1:m
    for k=1:n
        z=X(j,k)+i*Y(j,k);
% Kjører gjentatte iterasjoner av Newtons metode på det aktuelle
% punktet i gitteret inntil det ligger nærmere et av
% nullpunktene enn den angitte toleransen (eller til vi har utført
% maxit antall iterasjoner).
% Fargeverdien til punktet registreres i matrisen T og settes til
% hhv 1 (blå), 44 (gul) eller 60 (rød) avhengig av hvilket nullpunkt
% det gir konvergens mot.
        for t=1:maxit
            zny=z-(z^3+a*z+b)/(3*z^2+a);
            z=zny;
            if abs(z-rts(1))<tol
                T(j,k)=1;
               break;
            end
            if abs(z-rts(2))<tol
               T(j,k)=44;
               break;
            end
            if abs(z-rts(3))<tol
                T(j,k)=60;
                break;
```

```
            end
        end %for t
    end %for k
end %for j
image(x,y,T) %fargelegger punktet (x(j),y(k)) i planet med fargen
             %angitt av verdien av matrise-elementet T(j,k)
axis xy  %setter aksesystemet i vanlig xy-orientering;
colorbar %tegner opp søyle med fargeverdiene ved siden av plottet
% Vi kan eventuelt skrive figuren til fil ved å føye til kommandoen:
% print -depsc2 'fig.eps'
```

b) Den eneste modifikasjonen du trenger å gjøre i skriptet fra punkt a) er å sette parameterverdiene til å være $a = 2$ og $b = 0$, og deretter $a = -1$ og $b = 2$.

Løsning til oppgave 4

a) En modifisert MATLAB-funksjon som gir punktene dypere fargevalør jo raskere konvergens de gir mot et nullpunkt, kan se slik ut:

```
function kompleks(a,b)
% Funksjon som genererer et bilde av hvilke punkter i planet som
% gir konvergens mot samme nullpunkt når vi bruker Newtons metode
% på den komplekse funksjonen f(z)=z^3+a*z+b, hvor koeffisientene
% a og b er innparametre til funksjonen.
% Punktene i planet tilordnes en farge som angir hvilket av
% nullpunktene de gir konvergens mot.
% I denne versjonen gir vi i tillegg punktene dypere fargevalør jo
% raskere konvergensen mot nullpunktet går.
x=[-2:0.01:2];
y=[-2:0.01:2];
[X,Y]=meshgrid(x,y);
[m,n]=size(X);
maxit=30; % maksimalt antall iterasjoner som skal utføres
tol=0.001; %toleranse
T=27*ones(m,n); %initialiserer alle fargeverdiene til 27 (turkis)
% Vi angir koeffisientene foran potensene av z i polynomet z^3+a*z+b:
p=[1 0 a b];
% Vi lar MATLAB finne numeriske verdier for de tre nullpunktene til
% polynomet:
rts=roots(p);
% Så gjennomløper vi ett og ett punkt i gitteret ved en dobbelt
% for-løkke:
for j=1:m
    for k=1:n
        z=X(j,k)+i*Y(j,k);
% Kjører gjentatte iterasjoner av Newtons metode på det aktuelle
% punktet i gitteret inntil resultatet ligger nærmere et av
% nullpunktene enn den angitte toleransen (eller til vi har utført
% maxit antall iterasjoner).
% Fargeverdien til punktet registreres i matrisen T og settes til
% hhv. blå (verdier fra 1 og litt oppover), gul (verdier fra 48
```

```
% og litt nedover) eller rød (verdier fra 64 og litt nedover)
% avhengig av hvilket nullpunkt det gir konvergens mot. Vi
% indikerer konvergenshastigheten ved å justere fargeverdien slik
% at fargevaløren blir lysere jo langsommere konvergensen går.
        for t=1:maxit
            zny=z-(z^3+a*z+b)/(3*z^2+a);
            z=zny;
            if abs(z-rts(1))<tol
                T(j,k)=1+t*(23/maxit);
                break;
            end
            if abs(z-rts(2))<tol
               T(j,k)=48-t*(23/maxit);
                break;
            end
            if abs(z-rts(3))<tol
                T(j,k)=64-t*(23/maxit);
                break;
            end
        end %for t
    end %for k
end %for j
image(x,y,T) %fargelegger punktet (x(j),y(k)) i planet med fargen
             %angitt av verdien av matrise-elementet T(j,k)
axis xy  %setter aksesystemet i vanlig xy-orientering;
colorbar %tegner opp søyle med fargeverdiene ved siden av plottet
% Vi kan eventuelt skrive figuren til fil ved å føye til kommandoen:
% print -depsc2 'fig.eps'
```

b) Nullpunktene er de tre prikkene med mørkest fargevalør i de største fargeområdene av blått, gult og rødt. Vi ser at rundt hvert nullpunkt er det et forholdsvis stort område hvor alle punkter konvergerer mot nullpunktet, men konvergensen går raskere (dypere fargevalør) jo nærmerer nullpunktet startpunktet ligger. Inni hvert hovedfargeområde finner vi imidlertid mindre områder som konvergerer mot de andre nullpunktene, og dette gir det intrikate fraktale bildet.

c) Her setter du bare parameterverdiene til å være $a = 2$ og $b = 0$, og deretter $a = -1$ og $b = 2$.

Løsning til oppgave 5

Den modifiserte MATLAB-funksjonen som gir mulighet til å flytte sentrum og zoome inn på et hvilket som helst punkt, kan se slik ut:

```
function komplekszoom(sentrumx,sentrumy,zoom)
% Funksjon som genererer et bilde av hvilke punkter i planet som
% gir konvergens mot samme nullpunkt når vi bruker Newtons metode
% på den komplekse funksjonen f(z)=z^3+a*z+b
% Punktene i planet tilordnes en farge som angir hvilket av
% nullpunktene de gir konvergens mot, og hvor rask konvergensen er.
% Denne versjonen kan brukes til å zoome inn på et hvilket som helst
% punkt (xsentrum, ysentrum) som angis som innparameter til prosedyren.
```

```
% I tillegg angis ønsket forstørrelse ved innparameteren zoom. Hvis vi
% ikke ønsker å forstørre noe, velges zoom=1, ønsker vi dobbelt
% forstørrelse, velges zoom=2 osv.
% Vi regner ut (halve) lengden på intervallet ved hjelp av den
% oppgitte forstørrelsen gitt ved innparameteren zoom.
% Denne lengden er i utgangspunktet lik 2 før forstørrelse.
lengde=2/zoom;
x=linspace(sentrumx-lengde,sentrumx+lengde,400);
y=linspace(sentrumy-lengde,sentrumy+lengde,400);
[X,Y]=meshgrid(x,y);
[m,n]=size(X);
maxit=30; % maksimalt antall iterasjoner som skal utføres
tol=0.001; %toleranse
T=27*ones(m,n); %initialiserer alle fargeverdiene til å være 27 (turkis)
% Velger parametere til funksjonen f(z)=z^3+a*z+b
a=0;
b=-1;
% Vi angir koeffisientene foran potensene av z i polynomet z^3+a*x+b:
p=[1 0 a b];
% Vi lar MATLAB finne numeriske verdier for de tre nullpunktene til
% polynomet:
rts=roots(p);
% Så gjennomløper vi ett og ett punkt i gitteret ved en dobbelt
% for-løkke:
for j=1:m
    for k=1:n
        z=X(j,k)+i*Y(j,k);
        for t=1:maxit
            zny=z-(z^3+a*z+b)/(3*z^2+a);
            z=zny;
            if abs(z-rts(1))<tol
                T(j,k)=1+t*(23/maxit);
                break;
            end
            if abs(z-rts(2))<tol
               T(j,k)=48-t*(23/maxit);
                break;
            end
            if abs(z-rts(3))<tol
                T(j,k)=64-t*(23/maxit);
                break;
            end
        end %for t
    end %for k
end %for j
image(x,y,T) %fargelegger punktet (x(j),y(k)) i planet med fargen
             %angitt av verdien av matrise-elementet T(j,k)
axis xy  %setter aksesystemet i vanlig xy-orientering, ellers ville
         %verdiene på y-aksen går i motsatt retning av vanlig
```

```
colorbar %tegner opp søyle med fargeverdiene ved siden av plottet
% Vi kan eventuelt skrive figuren til fil ved å føye til kommandoen:
% print -depsc2 'fig.eps'
```

For å lage en liten sekvens av bilder som zoomer inn på origo, kan vi f.eks doble forstørrelsen hver gang ved å gi kommandoene

```
>> komplekszoom(0,0,1)
>> komplekszoom(0,0,2)
>> komplekszoom(0,0,4)
>> komplekszoom(0,0,8)
```

Tilsvarende kan vi lage en sekvens av bilder som zoomer inn på punktet $(-0.8, 0)$ ved å bytte ut første parameter med -0.8.

Løsning på ekstraoppgave: For å lage en animasjon som zoomer mer gradvis inn på punktet $(-0.8, 0)$, kan vi blåse opp forstørrelsene langsommere ved å velge en mindre oppblåsningsfaktor, f.eks `faktor=1.25` (da vi fordoblet bildestørrelsen ovenfor var denne faktoren lik 2). Følgende kommandoer lager en slik animasjon:

```
>> faktor=1.25;
>> makszoom=10;
>> for t=1:makszoom
     komplekszoom(-0.8,0,faktor^t);
     pause(0.1);
     F(t)=getframe; %plott nummer t lagres i F(t)
   end
>> movie(F,1,2); %spiller animasjonen en gang med 2 bilder pr sekund.
```

Løsning til oppgave 6

a) Vi legger følgende MATLAB-kode på filen 'newtonfler.m'

```
function x = newtonfler(start,fogj)
% MATLAB-funksjon som utfører Newtons metode på en vektorfunksjon
% F med Jacobi-matrise J spesifisert på filen 'fogj.m'.
% Innparametre:
% start: søylevektor som inneholder startpunktet for iterasjonen
% fogj: navn på MATLAB-funksjon som returnerer vektor [F,J] hvor F
% er en vektorfunksjon gitt som en søylevektor med komponenter
% gitt av funksjonene i systemet vi skal benytte Newtons metode på,
% mens J er den tilhørende Jacobi-matrisen.
maxit=30;
tol=0.0000001;
diff=tol+1;
x=start;
[F,J]=feval(fogj,x);
for t=1:maxit
  xny=x-J\F;
  diff=norm(xny-x);
  x=xny;
  [F,J]=feval(fogj,x);
  fprintf('itnr=%2d x=[%13.10f,%13.10f] F(x)=[%13.10f,%13.10f]\n',...
```

```
  t,x(1),x(2),F(1),F(2))
  if diff<tol
    break;
  end %if
end %for
% Kalles f.eks med kommandoen:
% >> x=newtonfler([-7;1],'fogjac')
% NB! Merk apostofene rundt navnet på funksjonsprosedyren
% som sendes med som innparameter.
```

b) Vi tegner opp nivåkurvene $f(x,y)=0$ og $g(x,y)=0$ i samme figurvindu (og skriver plottet til filen `kontur.eps`) ved følgende lille MATLAB-skript:

```
[x,y]=meshgrid(-10:0.1:10);
f=x.^2+y.^2-48;
g=x+y+6;
contour(x,y,f,[0 0]) % tegner opp en nivåkurve hvor f=0
hold on
contour(x,y,g,[0 0]) % tegner opp en nivåkurve hvor g=0
% axis([-10 10 -10 10])
axis('square')
hold off
print -deps 'kontur.eps'
```

Av figuren ser vi at de to nivåkurvene skjærer hverandre i to punkter, så ligningssystemet har to løsninger.

c) Vi ser av figuren i punkt a) at de to nullpunktene ligger i nærheten av $(-7,1)$ og $(1,-7)$, så det kan være lurt å prøve disse som startverdier.

Før vi kan bruke MATLAB-funksjonen `newtonfler`, må vi definere vektorfunksjonen vi ønsker å studere, sammen med dens Jacobi-matrise på en m-fil vi f.eks kan kalle `fogjac.m`

```
function [F,J]=fogjac(X)
% Returnerer en radvektor [F,J] hvor F er en søylevektor med
% funksjonene f(x,y)=x^2+y^2-48 og g(x,y)=x+y+6
% evaluert i punktet gitt ved søylevektoren X, mens J er den
% tilhørende Jacobi-matrisen evaluert i det samme punktet X.
x=X(1);
y=X(2);
F=[x^2+y^2-48; x+y+6];
f_x=2*x;
f_y=2*y;
g_x=1;
g_y=1;
J=[f_x, f_y; g_x, g_y];
```

For å utføre Newtons metode med startpunkt $(-7,1)$, gir vi nå kommandoen

```
>> x=newtonfler([-7;1],'fogjac')
```

og finner da tilnærmingsverdien $x=(-6.8730, 0.8730)$.

For å finne en tilnærmingsverdi til det andre nullpunktet, bruker vi startpunktet $(1,-7)$ og gir kommandoen

```
>> x=newtonfler([1;-7],'fogjac')
```

Da finner vi tilnærmingsverdien $x = (0.8730, -6.8730)$.

NB! Husk at vi må ha apostrofer rundt funksjonsnavnene vi sender som innparametre! Husk også på at startpunktet skal angis som en søylevektor.

Løsning til oppgave 7

For å bruke MATLAB-funksjonen `newtonfler` fra løsningsforslaget til oppgave 6, trenger vi bare å legge følgende kode på filen 'f3ogjac.m':

```
function [F,J]=f3ogjac(X);
% Returnerer en radvektor [F,J] hvor F er en søylevektor med
% funksjonene f(x,y,z)=y^2+z^2-3, g(x,y,z)=(x^2+z^2-2) og h(x,y,z)=x^2-z
% evaluert i punktet gitt ved søylevektoren X, mens J er den tilhørende
% Jacobi-matrisen evaluert i det samme punktet X.
x=X(1);
y=X(2);
z=X(3);
F=[y^2+z^2-3; x^2+z^2-2; x^2-z];
dfx=0; dfy=2*y; dfz=2*z;
dgx=2*x; dgy=0; dgz=2*z;
dhx=2*x; dhy=0; dhz=-1;
J=[dfx dfy dfz; dgx dgy dgz; dhx dhy dhz];
```

For å kjøre MATLAB-funksjonen `newtonfler` på dette ligningssystemet med startpunkt $x = [1, -7, 5]$, gir vi da kommandoen

```
>> newtonfler([1;-7;5],'f3ogjac')
```

NB! Husk at startpunktet x må angis som en søylevektor.

Vi finner da tilnærmingsverdien $x = (1.0000, -1.4142, 1.0000)$.

Løsningsforslag til øvelse 1

Løsning til oppgave 1

a) Vi kan beregne matriseproduktet $C\mathbf{x}$ ved å ta en lineærkombinasjon av søylene i C med koeffisienter gitt ved komponentene i $\mathbf{x}$.

b) Dette er en generell sammenheng. Den komponentvise definisjonen av matrisemultiplikasjon sier at vi får i-te komponent i søylevektoren $C\mathbf{x}$ ved å ta skalarproduktet av i-te rad i C med vektoren $\mathbf{x}$, det vil si at i-te komponent i $C\mathbf{x}$ er en lineær kombinasjon av elementene i i-te rad fra C med komponentene i $\mathbf{x}$ som koeffisienter. Siden vi altså bruker de samme koeffisientene (fra $\mathbf{x}$) hver gang vi uttrykker en komponent i $C\mathbf{x}$ som en slik lineær kombinasjon, betyr dette at vektoren $C\mathbf{x}$ er en lineær kombinasjon av søylevektorene i C med komponentene i $\mathbf{x}$ som koeffisienter.

Løsning til oppgave 2

a) Vi får første søyle i matriseproduktet AB ved å multiplisere A med første søyle i B. Tilsvarende fremkommer andre søyle i AB ved å multiplisere A med andre søyle i B.

b) Dette er en generell sammenheng: Vi får k-te søyle i matriseproduktet AB ved å multiplisere A med k-te søyle i B. Ifølge den komponentvise definisjonen av matriseprodukt får vi i-te element i k-te søyle i produktmatrisen AB ved å ta skalarproduktet av linje nr. i fra A med søyle nr. k fra B. Det betyr at i-te komponent fra k-te søyle i AB er en lineær kombinasjon av elementene fra i-te rad i A med komponentene fra k-te søyle i B som koeffisienter. Siden vi altså bruker de samme koeffisientene (fra k-te søyle i B) for hver slik lineær kombinasjon, betyr dette at vi får k-te søyle i produktmatrisen AB ved å ta en lineær kombinasjon av søylene i A med koeffisienter fra k-te søyle i B. Men fra oppgave 1 vet vi at dette er det samme som å multiplisere A med k-te søyle i B.

Løsning til oppgave 3

Første søyle i AB er en lineær kombinasjon av søylene i A med koeffisienter fra første søyle i B. Tilsvarende er andre søyle i AB en lineær kombinasjon av søylene i A med koeffisienter fra andre søyle i B.

Vi kan bruke MATLAB til å sjekke at dette resultatet stemmer for matrisene A og B fra oppgave 2 ved å lage tre søylevektorer

```
>> a1=A(:,1);
>> a2=A(:,2);
>> a3=A(:,3);
```

og sjekke at første søyle i AB er lik lineærkombinasjonen

```
>> b1(1)*a1+b1(2)*a2+b1(3)*a3
```

og at andre søyle i AB er lik lineærkombinasjonen

```
>> b2(1)*a1+b2(2)*a2+b2(3)*a3
```

(Sammenlign enten med `A*B(:,1)` og `A*B(:,2)` eller med `A*b1` og `A*b2`)

Løsning til oppgave 4

a) Vi kan beregne matriseproduktet $\mathbf{y} * R$ ved å ta en lineærkombinasjon av radene i R med koeffisienter gitt ved komponentene i $\mathbf{y}$.

b) Dette er en generell sammenheng. Den komponentvise definisjonen av matrisemultiplikasjon sier at vi får j-te komponent i radvektoren $\mathbf{y}R$ ved å ta skalarproduktet av $\mathbf{y}$ med j-te søyle i R, det vil si at j-te komponent i $\mathbf{y}R$ er en lineær kombinasjon av elementene i j-te søyle fra R med komponentene i $\mathbf{y}$ som koeffisienter. Siden vi altså bruker de samme koeffisientene (fra $\mathbf{y}$) hver gang vi uttrykker en komponent i $\mathbf{y}R$ som en slik lineær kombinasjon, betyr dette at vektoren $\mathbf{y}R$ er en lineær kombinasjon av radvektorene i R med komponentene i $\mathbf{y}$ som koeffisienter.

Løsning til oppgave 5

a) Vi får første rad i DE ved å multiplisere første rad i D med E. Tilsvarende fremkommer andre rad i DE ved å multiplisere andre rad i D med E, og tredje rad i $D * E$ fremkommer ved å multiplisere tredje rad i D med E.

b) Dette er en generell sammenheng: Vi får i-te rad i matriseproduktet DE ved å multiplisere i-te rad i D med E. Ifølge den komponentvise definisjonen av matriseprodukt får vi k-te element i i-te rad i produktmatrisen DE ved å ta skalarproduktet av linje nr. i i D med søyle nr. k i E. Det betyr at k-te komponent i i-te rad i DE er en lineær kombinasjon av elementene i k-te søyle i E med komponentene i i-te rad fra D som koeffisienter. Siden vi altså bruker de samme koeffisientene (fra i-te rad i D) for hver slik lineær kombinasjon, betyr dette at vi får i-te rad i produktmatrisen DE ved å ta en lineær kombinasjon av radene i E med koeffisienter fra i-te rad i D. Men fra oppgave 4 vet vi at dette er det samme som å multiplisere i-te rad i D med E.

Løsning til oppgave 6

Første rad i DE er en lineær kombinasjon av radene i E med koeffisienter fra første rad i D. Tilsvarende er andre (hhv. tredje) rad i DE en lineær kombinasjon av radene i E med koeffisienter fra andre (hhv. tredje) rad i D.

Vi kan bruke MATLAB til å sjekke at dette resultatet stemmer for matrisene D og E fra oppgave 5 ved å lage tre radvektorer

```
>> e1=A(1,:);
>> e2=A(2,:);
>> e3=A(3,:);
```

og sjekke at første rad i DE er lik lineærkombinasjonen

```
>> d1(1)*e1+d1(2)*e2+d1(3)*e3
```

og at andre rad i DE er lik lineærkombinasjonen

```
>> d2(1)*e1+d2(2)*e2+d2(3)*e3
```

og at tredje rad i DE er lik lineærkombinasjonen

```
>> d3(1)*e1+d3(2)*e2+d3(3)*e3
```

(Sammenlign enten med `D*E(1,:)`, `D*E(2,:)` og `D*E(3,:)` eller med `d1*E`, `d2*e` og `d3*E`)

Løsningsforslag til øvelse 2

Løsning til oppgave 1

a) Determinanten til den nye matrisen blir 3 ganger så stor som determinanten til den opprinnelige. Generelt gjelder det at hvis vi erstatter en vilkårlig rad med en skalar k ganger raden, blir determinanten til den nye matrisen k ganger så stor som den opprinnelige.

b) Hvis vi multipliserer en søyle med skalaren k, blir determinanten til den nye matrisen k ganger så stor som den opprinnelige.

Løsning til oppgave 2

a) Når vi bytter om to rader i en kvadratisk matrise, skifter determinanten fortegn.

b) Determinanten skifter fortegn.

Løsning til oppgave 3

a) Når vi adderer et skalarmultiplum av en rad til en annen rad, forblir determinanten uforandret.

b) Determinanten forblir uforandret.

Løsning til oppgave 4

Determinanten til en øvre triangulær matrise er lik produktet av elementene på hoveddiagonalen.

Løsning til oppgave 5

Determinanten til en nedre triangulær matrise er lik produktet av elementene på hoveddiagonalen.

Løsning til oppgave 6

a) Determinanten til en identitetsmatrise er lik 1.

b) Siden identitetsmatrisen er både øvre triangulær og nedre triangulær, vet vi fra forrige seksjon at determinanten er lik produktet av elementene på hoveddiagonalen. Resultatet følger derfor av at alle elementene på hoveddiagonalen er 1-ere.

Løsning til oppgave 7

Determinanten til en permutasjonsmatrise er 1 eller -1 avhengig av om antall inversjoner i permutasjonen er et partall eller et oddetall. Vi vet fra tidligere at hver gang vi bytter om to rader i en matrise, så skifter determinanten fortegn. Dersom vi må bytte om rader et partall antall ganger, blir determinanten lik determinanten til identitetsmatrisen, altså 1, og hvis vi må bytte om rader et odde antall ganger, blir determinanten lik -1.

Løsning til oppgave 8 (ekstraoppgave)

a) Vi skal forklare hvorfor det er slik at hvis vi erstatter en vilkårlig rad med en skalar k ganger raden, så blir determinanten til den nye matrisen k ganger så stor som den opprinnelige.

Dette kan vi se ut ifra definisjonen av determinanten ved elementære produkter forsynt med fortegn, fordi alle de elementære produktene vil bli k ganger større siden de inneholder nøyaktig ett element fra den nye raden, og dette elementet er k ganger å stort som det opprinnelige elementet.

b) Vi skal forklare hvorfor det er slik at når vi bytter om to rader i en kvadratisk matrise, skifter determinanten fortegn.

Dette kan vi se ut ifra definisjonen av determinanten som en sum av elementære produkter forsynt med fortegn, fordi alle de elementære produktene vil skifte fortegn når vi bytter om to rader i matrisen. Ved en slik ombytting vil vi enten fjerne eller legge til et odde antall inversjoner i søylepermutasjonen σ, slik at fortegnet $\text{sign}(\sigma) = (-1)^{\text{inv}(\sigma)}$ skifter til motsatt verdi.

Ved ombytting av linje nr. i og j (hvor $i < j$) vil to matriseelementer $a_{i\sigma_i}$ og $a_{j\sigma_j}$ som inngår i et elementært produkt $\prod_i a_{i\sigma_i}$, opptre som elementene $a_{j\sigma_i}$ og $a_{i\sigma_j}$ i den nye matrisen, og søyleindeksparet (σ_i, σ_j) vil nå gjenfinnes som det inverterte paret (σ_j, σ_i). Dette bidrar til at antall inversjoner endres med 1.

For $k \neq i, j$ vil søyleindeksparene (σ_k, σ_i) og (σ_k, σ_j) gjenfinnes som (σ_k, σ_j) og (σ_k, σ_i). Hvis $k < i$ eller $k > j$, blir ingen av parene invertert. Hvis $i < k < j$, blir begge parene invertert. Disse tilfellene bidrar altså til at antall inversjoner endres med et partall.

Alt i alt ser vi derfor at antall inversjoner endres med et oddetall ved en ombytting av to linjer i matrisen.

c) Vi skal forklare hvorfor det er slik at når vi adderer et skalarmultiplum av en rad til en annen rad, forblir determinanten uforandret.

Det følger fra definisjonen av determinanten ved elementære produkter at determinanten er additiv som funksjon av rad nr. i, det vil si at hvis rad nr. i kan skrives som en sum av to vektorer v_1 og v_2, så blir determinanten lik summen av determinantene til de matrisene vi får ved å putte inn henholdsvis v_1 og v_2 som rad nr. i.

Fra oppgave 2 vet vi også at dersom vi bytter om to rader i en matrise, så skifter determinanten fortegn. Hvis vi har en matrise der to rader er like, og bytter om disse radene, skal altså determinanten skifte fortegn. Men siden den nye matrisen er lik den opprinnelige (vi byttet jo bare om to like rader), må determinanten samtidig være lik den opprinnelige. Det eneste tallet D som oppfyller $D = -D$, er $D = 0$. Det følger derfor at hvis to rader i en matrise er like, så er determinanten null.

Av disse to observasjonene følger resultatet: Hvis vi erstatter i-te rad i en matrise med i-te rad pluss k ganger j-te rad, blir determinanten til den nye matrisen ved linearitet lik den opprinnelige determinanten pluss k ganger determinanten til en matrise hvor den i-te raden er erstattet med den j-te raden fra den opprinnelige matrisen. Men siden den i-te og den j-te raden i den sistnevnte matrisen er like, slik at determinanten til denne matrisen er null, så følger konklusjonen.

d) Vi skal forklare hvorfor determinanten til en øvre triangulær matrise er lik produktet av elementene på hoveddiagonalen.

Det følger av definisjonen av determinanten ved elementærprodukter at det eneste elementærproduktet som ikke blir null, er produktet av elementene på hoveddiagonalen (alle andre elementærprodukter må inneholde et element som ligger under hoveddiagonalen (forklar!) og blir derfor null).

e) Vi skal forklare hvorfor determinanten til en nedre triangulær matrise er lik produktet av elementene på hoveddiagonalen.

Dette kommer av at det eneste elementærproduktet som ikke blir null, er produktet av elementene på hoveddiagonalen (alle andre elementærprodukter må inneholde et element som ligger over hoveddiagonalen og blir derfor null).