Python for MAT1120

Øyvind Ryan, Geir Dahl, Erik Bedos

2. september 2010

Innhold

Figurer

Innledning

Dette notatet inneholder det viktigste du trenger å vite om Python med tanke på behovene i MAT1120. Det inneholder en oversikt over kode du støter på i læreboka, i ukeoppgavene, på forelesningene, og i de obligatoriske oppgavene i kurset. Referanser i heftet er med hensyn på boka til Lay. Notasjonen i funksjonene følger også boka til Lay tett. På cd'en som følger med boka til Lay er det også en study guide. Denne inneholder en del nyttig informasjon, men har ikke innføringer i alle kommandoer vi bruker her. Der finner vi heller ikke litt lenger kodeeksempler som illustrere setninger i boka, slik dette heftet fokuserer på.

Heftet inneholder også en del laboratorieøvelser, som er tatt fra obligatoriske oppgaver i kurset de siste årene. Hver øvelse har et anvendt tema det fokuserer på. Spesielt er bildekompresjon og lyd brukt som temaer.

For MAT1120 har vi samlet programmeringstips i dette heftet, i motsetning til MAT1110, som har en del slikt i læreboka i tillegg. Dette heftet har dessuten mer fokus på programmering av litt lengre biter. Det er også fokus på å vise ting ved eksempler, ikke ved lange, omstendelige forklaringer. Heftet inneholder få oppgaver, siden boka til Lay allerede har en del programmeringsrelaterte oppgaver.

Det er ønskelig at du som student bruker dette heftet som et oppslagsverk når du lurer på hvor du skal starte når du skal løse en oppgave som krever programmering. Det er en fin måte å lære på å gå gjennom all kode i dette kompendiet - dels fordi man lærer programmering av det og og forskjellige kommandoer, men også fordi man får omsatt mange setninger fra læreboka til praksis på den måten.

Under den detaljerte undervisningsplanen for MAT1120 finner du også linker til foiler brukt på forelesningene og plensumsregningene. Mye av denne koden finner du i dette heftet, men ikke alt.

Heftet inneholder også en stjernemerket seksjon. Denne er ikke pensum i kurset.

Kapittel 4

Seksjon 4.1: Vektorrom og underrom

Nedenfor ser du hvordan man plotter punkter og vektorer i et koordinatsystem med Python.

```
# coding=utf-8
from numpy import *
from scitools.easyviz import *

L = 2
U = 10

figure(1)
axis([ - L, U, - L, U])

A = array([[ - L, U], [0, 0]])
B = array([[0, 0], [ - L, U]])

plot(A[0, :], A[1, :], 'b', grid='on') # tegner opp en linje langs x-aksen
                                       # (med grid)
hold('on')
plot(B[0, :], B[1, :], 'b')            # tegner opp en linje langs y-aksen

for k in xrange(7):                     # Plotter tilfeldige (x,y)
    x = (U - 1) * random.rand(2)       # x og y er mellom 0 og 9.
    plot([x[0]], [x[1]], 'or')         # r står for rødt. o er
                                       # mønster punktene tegnes i

# Nedenfor plotter vi som vektorer
plot([0, U * random.rand(1)], [0, U * random.rand(1)], '-og')
# g står for grønt. o står for mønsteret
plot([0, 5], [0, 2], '-ok') # k står for svart
                            # o står for mønsteret
hold('off')
raw_input('Press Enter to exit:')
```

Python tegner ikke opp koordinataksene automatisk for oss. I koden ser du at aksene tegnes eksplisitt, som fire linjer fra origo, i det første kallet på funksjonen `plot`.

Å gange en matrise med en vektor svarer til å lage en lineær kombinasjon av søylene i matrisen, ved å bruke koeffisienter tatt fra vektoren:

```
from numpy import *

A = matrix([[1.,2.,3.],[4.,1.,2.],[1.,4.,2.]])
print 'A = ', A
x = floor(matrix(3*random.rand(3,1)))
print 'x = ', x
linkomb = A*x
print 'linkomb = A*x = ', linkomb
```

`rref` kan brukes til å løse likningssystemer, alternativt sjekke om en vektor er en lineær kombinasjon av søylene i en matrise:

```
from numpy import *
from MAT1120lib import *

A = array([[1.,2.],[4.,1.],[1.,4.]])
print 'A = ', A
b = array([4.,9.,6.]) # Her er b = 2*a1 + a2,
                      # der a1 og a2 er 1. og 2. kolonne i A
print 'b = ', b
R = rref(hstack((A,matrix(b).T)))
print 'R = ', R

b = floor(4*random.rand(3))
print 'b = ', b
R = rref(hstack((A,matrix(b).T)))
print 'R = ', R
```

Du trenger faktisk ikke bruke `rref` for å løse likningssystemet $A\mathbf{x} = \mathbf{b}$. Det er bedre å skrive `x=linalg.solve(A,b)` .

Seksjon 4.2: Nullrom, søylerom, og lineære transformasjoner

Nullrommet til en matrise `A` finner du ved å skrive

```
null(A)
```

Søylene i den returnerte matrisen vil danne en basis for nullrommet. Antall søyler i den returnerte matrisen er dimensjonen til nullrommet.

Koden nedenfor genererer en tilfeldig matrise der tallene er 0 og 1. Hvis nullrommet bare består av $\mathbf{0}$ skriver den ut en melding. Hvis ikke tester den at første vektor i den returnerte matrisen faktisk ligger i nullrommet.

```
# coding=utf-8
from numpy import *
from MAT1120lib import *

n = 4
A = floor(matrix(1.5*random.rand(n,n))) # tilfeldig nxn-matrise
                                # der tallene er 0 eller 1
print 'A = ', A
V = null(A) # ortonormal basis for nullrommet
print 'V = ', V
r,s = V.shape

if s == 0:
    print 'Nullrommet består bare av 0, dvs A har full kolonnerang.\n'
else:
    v = V[:,0]
    print 'A*v = ', A*v # sjekk at dette er 0
```

En basis for søylerommet Col$A$ til matrisen A finner du enklest ved hjelp av kommandoen `orth(A)`. Søylene i den returnerte matrisen er en basis for søylerommet, slik at rangen til A er lik antall søyler i den returnerte matrisen. Hvis du i stedet vil ha en basis for radrommet kan du skrive

```
(orth(A')'
```

Dette er det samme trickset vi brukte for funksjonen `sum` tidligere, når vi ville summere over radene i stedet for søylene.

Koden nedenfor genererer en tilfeldig matrise, og skriver ut en basis for nullrommet og søylerommet. Til slutt verifiseres rangteoremet i Seksjon 4.6.

```
# coding=utf-8
from numpy import *
from MAT1120lib import *

n = 5
A = floor(2*random.rand(n,n))
print 'A = ',A
V = null(A) # ortonormal basis for nullrommet
print 'V = ',V
W = orth(A) # ortonormal basis for kolonnerommet
print 'W = ',W
m1,n1 = V.shape # n1 er dimensjonen til nullrommet
m2,n2 = W.shape # n2 er dimensjonen til kolonnerommet
print 'n1 + n2 = ',n1+n2
print 'n = ',n # Verifiserer teorem 14 i seksjon 4.6, n1+n2=n
```

Seksjon 4.3: Lineært uavhengige mengder; basiser

Koden under summerer opp et par nyttige kommandoer.

```
# coding=utf-8
from numpy import *
from MAT1120lib import *

A = array([[16.,2.,3.,13.],
           [5.,11.,10.,8.],
           [9.,7.,6.,12.],
           [4.,14.,15.,1.]])
# A er et magisk kvadrat
print 'A =',A
# Skriv ut radsummer
for i in xrange(4):
    radsum = 0.0
    for j in xrange(4):
        radsum += A[i,j]
    print radsum
# Skriv ut kolonnesummer
for j in xrange(4):
    kolonnesum = 0.0
    for i in xrange(4):
        kolonnesum += A[i,j]
    print kolonnesum

print 'orth(A) =',orth(A)
print 'rref(A) =',rref(A)
```

Ofte er vi interessert i å finne ut om en mengde av funksjoner er lineært uavhengig. En mye brukt teknikk til dette er å velge ut et sett med punkter, og så sette opp en likning for hvert punkt. Finner vi ingen andre løsninger enn null-løsningen vil funksjonene være lineært uavhengige. Som et eksempel, la oss finne ut om funksjonene

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

er lineært uavhengige. Vi velger ut punkter $t = t_0, t_1, t_2, t_3, t_4$ (like mange som det

er funksjoner), og setter opp relasjonene vi bruker for lineær uavhengighet:

$$\begin{aligned} c_0 + c_1 \sin t_0 + c_2 \sin^2 t_0 + c_3 \sin^3 t_0 + c_4 \sin^4 t_0 &= 0 \\ \vdots &= \vdots \\ c_0 + c_1 \sin t_4 + c_2 \sin^2 t_4 + c_3 \sin^3 t_4 + c_4 \sin^4 t_4 &= 0 \end{aligned}$$

Koeffisientmatrisen til dette likningsystemet er

$$\begin{bmatrix} 1 & \sin t_0 & \sin^2 t_0 & \sin^3 t_0 & \sin^4 t_0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \sin t_4 & \sin^2 t_4 & \sin^3 t_4 & \sin^4 t_4 \end{bmatrix}.$$

Det er klart at hvis den reduserte trappeformen til denne er identitetsmatrisen så har likningssystemet vårt bare triviell løsning, og dermed er funksjonene lineært uavhengige. For å velge punkter $t = t_0, t_1, t_2, t_3, t_4$ og gjøre alt dette kan du i skrive

```
from numpy import *
from MAT1120lib import *

t = array([linspace(0,0.4,5)]).T
A = hstack(( (sin(t))**0, (sin(t))**1, (sin(t))**2,
             (sin(t))**3, (sin(t))**4 ))
print rref(A)
```

Sørg spesielt her for at du forstår koden som produserer koeffisientmatrisen A. Her anvendes først funksjonen `cos` på en vektor, deretter opphøyes den resulterende vektoren komponentvis i et sett potenser.

I koden ovenfor valgte vi punktene $0, 0.1, 0.2, 0.3, 0.4$, og det lyktes oss med dette valget å se at funksjonene er lineært uavhengige. Det er imidlertid ikke opplagt hvordan disse punktene skal velges for at denne strategien skal lykkes, og det er ofte slik at flere valg av punkter ikke vil hjelpe oss. For eksempel vil ikke valgene $0, 2\pi, 4\pi, 6\pi, ...$ i koden ovenfor hjelpe oss til å konkludere med lineær uavhengighet. Den beste strategien her er å prøve seg litt frem med forskjellige sett med punkter.

Seksjon 4.4: Koordinatsystemer

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

```
from numpy import *

b1 = matrix([[1.],[2.]]) # basisvektorer
b2 = matrix([[1.],[-1.]])
P = hstack((b1,b2)) # basisskifte-matrisen (b1,b2-basis til std.basis)
print 'P = ', P
print 'det(P) = %f' %(linalg.det(P))
x = matrix([[3.],[3.]])
print 'x = ', x
c = linalg.inv(P)*x # koordinatvektoren til x i b1,b2-basis
print 'c = ', c
print 'P*c = ', P*c # Gir tilbake opprinnelige koordinater
```

Seksjon 4.6: Rang

Funksjonen `rank` gir deg rangen til en matrise. Vi vet at dette er dimensjonen til radrommet til matrisen, som også er lik dimensjonen til søylerommet til matrisen. Rangteoremet kan verifiseres slik:

```
from MAT1120lib import *

# rang til matriser - rang teoremet
m=3
n=5
A=floor(1.5*random.rand(m,n)) # "tilfeldig" matrise
N=null(A)              # kolonnene i N er basis for nullrommet til A
(p,q)=N.shape          # dimensjonen til N: q er dim. til Nul A
r=rang(A)
print('rank A = ',r,'dim Nul A = ',q,'Ant. kolonner i A n= ',n,'\n')
R=rref(A)              # redusert trappeform
```

Radrommet til matrisen kan du finne ved hjelp av funksjonen `rref`. Radene i resultatmatrisen vil være en basis for radrommet siden `rref` bare anvender elementære radoperasjoner, og disse forandrer ikke radrommet. Antallet rader som er $\neq 0$, eller antall pivot-elementer, vil være rangen til radrommet. Rangen kan også finnes på andre, og bedre måter. Vi kommer tilbake til dette i kapittel 7. Et problem er at, selv om to matriser ligger veldig nær hverandre, så kan det tenkes at de har forskjellig rang. Dette kan gi oss numeriske problemer når vi beregner rang.

Seksjon 4.8: Anvendelser på differenslikninger

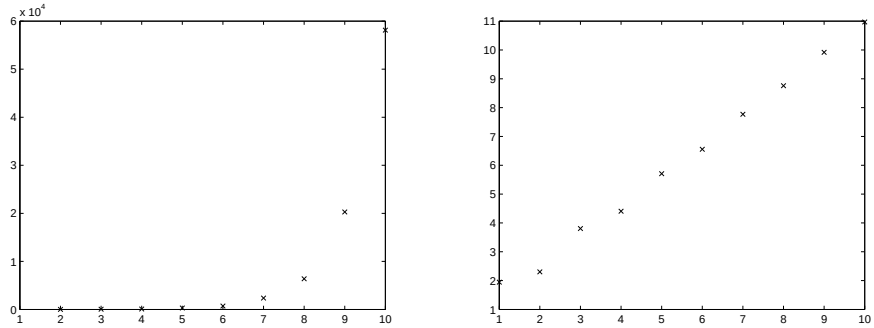
Koden nedenfor viser hvordan vi kan plotte løsningen av den (ikkehomogene) differenslikningen

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

```
# coding=utf-8
from numpy import *
from scitools.easyviz import *

n = 10 # prøv n=40 etterpå!
y = zeros(n)
y[0] = random.rand(1)
y[1] = 1
y[2] = 1

for k in xrange(n-3):
    y[k+3] = 2*y[k+2] + 5*y[k+1] - 6*y[k]

print y
figure(1)
plot(y)
figure(2)
plot(log(y))
raw_input('Press Enter to exit:')
```

Koden produserer to plott, begge vist i Figur 1. Grunnen til at vi også plottet logaritmen til løsningen av differenslikningen, er at løsningene av differenslikningen kan uttrykkes ved eksponentialfunksjoner: i dette tilfellet er løsningen

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



(a) Plot av vektoren `y`

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

Figur 1: Plott av løsningene til en differenslikning

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

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

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

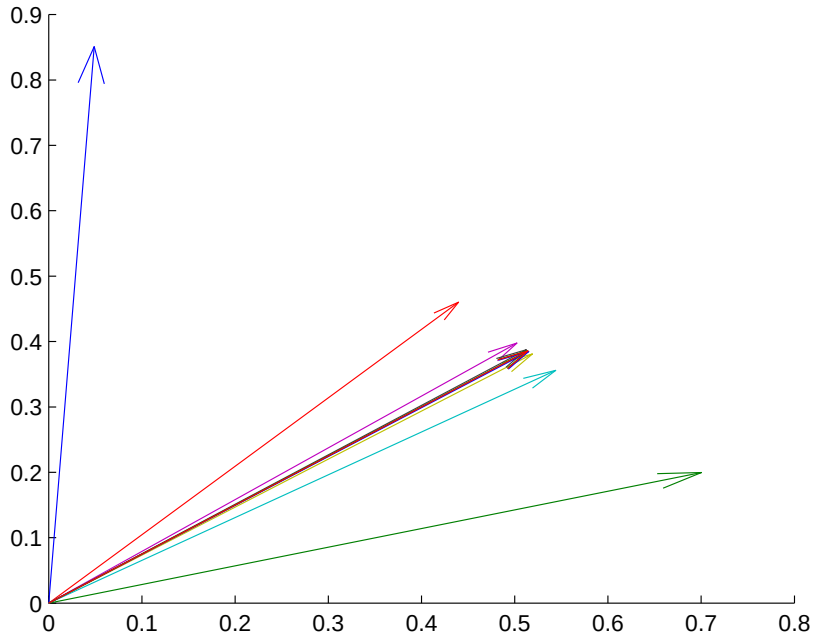
```
roots(array([1,-2,-5,6]))
```

i Python, som gir $r = 1$, $r = -2$, $r = 3$ som løsninger, som er tall vi finner igjen i (2).

Seksjon 4.9: Anvendelser på Markovkjeder

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

```
# coding=utf-8
from numpy import *
from time import sleep
from scitools.easyviz import *

n = 10
P = matrix([[.4,.8],[.6,.2]]) # Definerer en stokastisk matrise
r = random.rand(1)
x = matrix([r,1-r])           # Definerer en sannsynlighetsvektor
y = matrix(zeros((2,n),double))
hold('on')

for k in xrange(n):
    y[:,k] = x
    quiver([0],[0],[y[0,k]],[y[1,k]]) # Plotter vektorene i
                                      # Markov-kjeden.
                                      # Ser ut som de konvergerer.
    sleep(1)
    x = P*x

print y                   # y inneholder de første 10 leddene i en
                          # Markov-kjede definert fra P og x.

raw_input('Press Enter to exit:')
```



Figur 2: Plott av vektorene i en Markov-kjede

Legg merke til den litt spesielle bruken av funksjonen `quiver`. `quiver` brukes vanligvis til å plotte et vektorfelt, men her bruker vi den til å plotte en og en vektor om gangen (inne i løkka). Videre brukes funksjonen `sleep` , slik at vi kan se hvordan Markov-kjeden utvikler seg gradvis. Vi har plottet vektorene i Markov-kjeden i Figur 2. Det ser ut som de konvergerer mot en bestemt vektor. Denne vektoren kan du finne ved å regne ut egenvektoren for den største egenverdien for matrisen `P`.

Markov-kjeder dukker opp i mange andre sammenhenger. Et mye brukt eksempel er surfing på nettet, der å endre fra en tilstand til en annen svarer til å surfe fra en side til en annen:

```
# coding=utf-8
from numpy import *
from scitools.easyviz import *

# Eksempel på Markov kjede:
# en gruppe vid.skole elever "vandrer på nettet" der bare tre
# nettsider er tilgjengelige (pga datafeil!): uio.no, ntnu.no, og vg.no
# I hvert tidskritt bytter noen nettsider med følgende andeler
#                         Fra:
#                 uio      ntnu     vg
#        uio      0.7      0.3      0.25
# Til:   ntnu     0.1      0.5      0.15
#        vg       0.2      0.2      0.6
# Vi stater med jevn fordeling på sidene x=(1/3,1/3,1/3).
# Hvordan blir fordelingen etterhvert?

N=14     # antall tidsskritt
# P=overgangsmatrisen
P=matrix([[0.7,0.3,0.25],
          [0.1,0.5,0.15],
```

```
          [0.2,0.2,0.6]])
# x=matrix([[1.0/3],[1.0/3],[1.0/3]]) # uniform startfordeling
# alternativt: vilkårlig startfordeling
x =matrix(random.rand(3,1))
x=x/sum(x)

F=x
for k in xrange(1,N):
   x=P*x
   F = hstack((F,x))
print F
plot(range(0,N),F[0,:],range(0,N),F[1,:],range(0,N),F[2,:])
# viser fordelingen som funk. av tiden for hver nettside
raw_input('Press Enter to exit:')
```

Plottet vi får gir inntrykk av konvergens mot en likevektsvektor, og vi leser av verdiene $(0.47, 0.19, 0.33)$ for komponentene i denne fra plottet. Disse verdiene kan vi også regne ut eksakt ved å regne ut egenvektorene til matrisen som er oppgitt i programmet.

For å få tak i en tilfeldig generert regulær stokastisk matrise kan du skrive

```
# coding=utf-8
from numpy import *

def randstocmatrix(n):
  # Returnerer en regulær stokastisk matrise
  A = random.rand(n,n)+1
  for k in xrange(0,n):
    A[:,k] = A[:,k]/sum(A[:,k])
  return A
```

Du ser her at vi inne i koden legger til 1 for å kun få positive tall inne i matrisen, slik at den resulterende matrisen blir regulær.

Kapittel 5

Seksjon 5.1: Egenvektorer og egenverdier

Koden nedenfor plotter vektorene `x` og `A*x` mot hverandre.

```
from numpy import *
from scitools.easyviz import *

A = matrix([[2.,4.],[4.,1.]])
x = 10*matrix(random.rand(2,1))
x = x/linalg.norm(x)
print 'x = ', x
y = A*x
print 'y = ', y
figure(1)
plot([0,x[0]],[0,x[1]],'b')
hold('on')
plot([0,y[0]],[0,y[1]],'r')
axis([0,5,0,5])
hold('off')
raw_input('Press Enter to exit:')
```

Plottet viser om `x` er en egenvektor for `A` eller ikke. I så fall er de to vektorene parallelle.

En måte regne ut egenverdier og egenverdier på er å bruke funksjonen `linalg.eig()` . Skriver du

```
D,V =linalg.eig(A)
```

vil `D` være en vektor som inneholder egenverdiene, mens `V` inneholder egenvektorer for tilsvarende egenverdi i samme søyle. Eksempelvis er den andre egenverdien D[2] , med tilhørende egenvektor V[:,2] . Koden nedenfor verifiserer at disse er tilhørende egenverdi/egenvektorpar.

```
from numpy import *

A = matrix([[1.,2.],[4.,3.]])
print 'A = ', A
(D,V) = linalg.eig(A)
print 'D = ',D
print 'V = ',V
Lambda = D[1]
print 'Lambda = %f' %(Lambda)
x = V[:,1]
print 'x = ',x
print 'A*x = ', A*x
print 'Lambda * x = ',Lambda*x
```

Hvis du ikke lister opp returparametrene i kallet på `eig(A)` vil funksjonen returnere en søylevektor hvor verdiene er egenverdiene med eventuelle gjentakelser. Gitt at vi har en egenverdi `lambda`, så kan vi alternativt finne tilhørende egenvektor ved å løse systemet `(A-lambda*eye(2))*x=0`, det vil si ved å finne nullrommet til `A-lambda*eye(2)`. Koden nedenfor viser hvordan vi kan gjøre dette, og verifisere til slutt at resultatet faktisk er en egenvektor.

```
# coding=utf-8
from numpy import *
from MAT1120lib import *

A = matrix([[1.,2.],[4.,3.]])
Lambda = 5
V = null(A - Lambda*identity(2)) #ortonormal basis for nullrommet
x = V[:,0]
print 'x = ', x
print 'A*x - Lambda*x = ', A*x - Lambda*x
```

Egenverdier og egenvektorer hjelper oss til å forstå likevektstilstander for dynamiske systemer. En likevektstilstand er jo ikke noe annet enn en egenvektor med 1 som tilhørende egenverdi. Se på koden

```
from numpy import *
from time import sleep

A = matrix([[.6,.3],[.4,.7]]) # stokastisk matrise
x = matrix([[100.],[40.]])    # startpopulasjon
print x
sleep(1)

for k in xrange(20):
    x = A*x
    print x
    sleep(1)
```

Kjører du koden vil du se at systemet ser ut til å konvergere mot en bestemt vektor over tid. Forklaringen er at matrisen A er regulær stokastisk, og dermed vil systemet konvergere mot en unik likevektstilstand.

Seksjon 5.2: Den karakteristiske likningen

Koden nedenfor viser hvordan man kan beregne verdien av et polynom, gitt at koeffisientene i polynomet er samlet i en vektor. Koden viser også hvordan man kan finne det karakteristiske polynomet til en matrise.

```
# coding=utf-8
from numpy import *

p = array([1.,-6.,5.])        # p(x) = x^2 - 6x + 5
print 'p(2) = ', polyval(p,2) # evaluer p(2)
print 'røtter: ', roots(p)    # røtter til p

A = array([[1.,2.,3.],[4.,5.,6.],[7.,8.,9.]])
p = poly(A)
print 'Kar. pol. til A: ', p
print 'Egenverdier: ', roots(p)
```

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

Seksjon 5.3: Diagonalisering

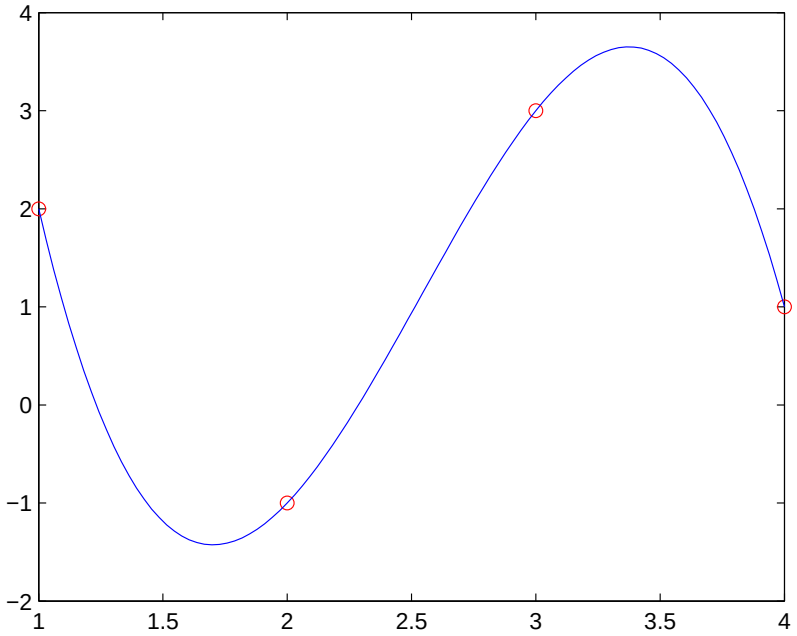
Følgende kode verifiserer at matrisene med egenvektorer og egenverdier som Python returnerer faktisk kan brukes til å diagonalisere matrisen.

```
# coding=utf-8
from numpy import *
from MAT1120lib import *

A = matrix([[1.,2.,3.],[4.,5.,6.],[7.,8.,9.]])
print 'A = ',A
(D,P) = linalg.eig(A)
D = diag(D) # eig() gir en array av egenverdier, ikke diagonal matrise
print 'P = ',P
print 'D = ',D
print 'A*P = ', A*P
print 'P*diag(D) = ', P*matrix(diag(D))
# Se på rangen til matrisene, f.eks. rang(P)

A = matrix([[1.,2.],[0.,1.]]) # ikke diagonaliserbar
print 'A = ', A
(D,P) = linalg.eig(A)
print 'P = ', P
print 'rank P = ', rang(P)
```

Seksjon 5.4: Egenvektorer og lineære transformasjoner

I et tilleggsnotat til dette avsnittet lærte vi litt mer om matriserepresentasjoner og lineære avbildninger. Spesielt lærte vi hvordan polynomer unikt kan identifiseres ved et sett av punkter de går gjennom. Starter vi med et gitt antall punkter kan vi finne et *interpolerende* polynom som går gjennom disse punktene, og plotte det sammen med punktene slik:

```
from numpy import *
from scitools.easyviz import *

a=array([[1],[2],[3],[4]])
b=array([[2],[-1],[3],[1]])
M=array([[1,1,1,1],[1,2,4,8],[1,3,9,27],[1,4,16,64]])

q=linalg.solve(M,b)  # q er koeffisientene til det interpolerende polynomet

t=linspace(1,4)
p=q[0] + q[1]*t + q[2]*t**2+ q[3]*t**3

for j in xrange (0,4):
  plot(a[j], b[j],'or')
  hold('on')

plot(t,p,'b')
raw_input('Press Enter to exit:')
```



Figur 3: Et interpolerende polynom som går gjennom 4 punkter

Plottet denne koden produserer er vist i Figur 3. Interpolasjon med trigonometriske funksjoner er også vanlig, og det er rett fram å modifisere koden over til å bruke trigonometriske funksjoner i stedet for polynomer.

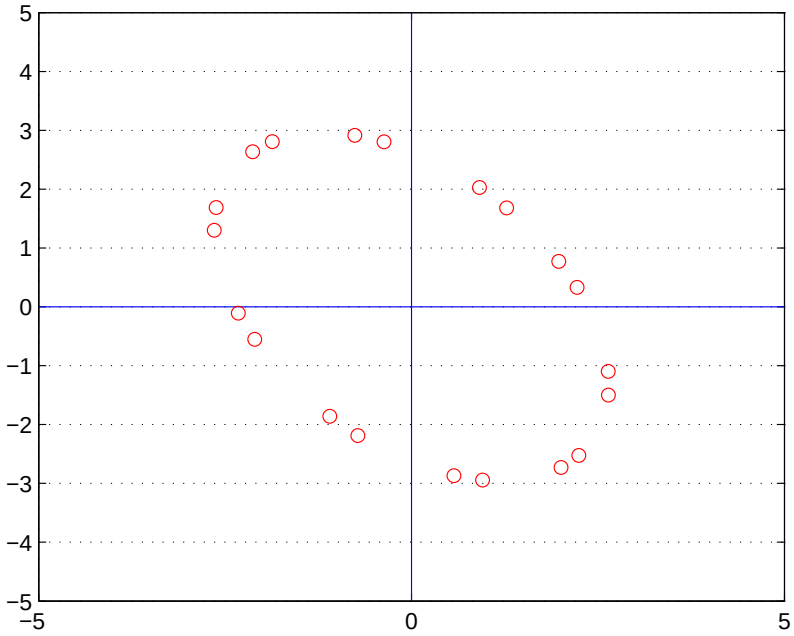
Oppgave 1

Programmer en funksjon som tar søylevektorer `a` og `b` av samme lengde (n) som parametre, og som plotter et polynom av grad $n-1$ som går gjennom disse punktene (som koden over). Koden skal returnere en feilmelding hvis det ikke lar seg gjøre å finne et slikt polynom. Når er det ikke mulig å finne et slikt polynom?

Seksjon 5.5: Komplekse egenverdier

Koden nedenfor plotter utviklingen over tid i et diskret dynamisk system med komplekse egenverdier.

```
from numpy import *
from scitools.easyviz import *
from time import sleep

L = 5
U = 5
figure(1)
axis([-L,U-L,U])
A = matrix([[-L,U],[0,0]])
B = matrix([[0,0],[-L,U]])
plot(A[0,:],A[1,:],'b',grid='on')
hold('on')
```



Figur 4: Plott av utviklingen i et diskret dynamisk system

```
plot(B[0,:],B[1,:],'b')
A = matrix([[.5,-.6],[.75,1.1]])
print 'A = ', A
x = matrix(ones(2,1,double) + random.rand(2,1))

for k in xrange(20):
    plot([x[0]],[x[1]],'or')
    sleep(1)
    x = A*x #regner ut neste tilstand i systemet

hold('off')
raw_input('Press Enter to exit:')
```

Det resulterende plottet er vist i Figur 4. Når du kjører koden vil du se at plotterutinen tar en pause før den plotter det neste punktet. Dette skyldes kallet på funksjonen `sleep` . Du vil også se at systemet hverken beveger seg mot origo, eller mot uendelig. Dette skyldes at egenverdiene til systemet har absoluttverdi 1. Et system som går mot uendelig får du ved å velge en matrise med egenverdier med absoluttverdi større enn 1, slik som koden nedenfor.

```
from numpy import *
from scitools.easyviz import *
from time import *

L=500
U=500
axis([-L,U,-L,U])
axis('manual')
A=array([[-L,U],[0,0]])
```

```
B=array([[0,0],[-L,U]])
plot(A[0,:],A[1,:])
hold('on')
plot(B[0,:],B[1,:])
grid

A=matrix([[1,-1.5],[1.5,1]])    # r = sqrt(13)/2 > 1  !!!
x=matrix(ones((2,1),float)+random.rand(2,1))

for k in xrange(10):
  plot(x[0],x[1],'or')
  hold('on')
  sleep(1)
  x=A*x

raw_input('Press Enter to exit:')
```

Et system som nærmer seg origo får du ved å velge en matrise med egenverdier med absoluttverdi mindre enn 1, slik som koden nedenfor.

```
from numpy import *
from scitools.easyviz import *
from time import *

L=20
U=20
axis([-L,U,-L,U])
axis('manual')
A=array([[-L,U],[0,0]])
B=array([[0,0],[-L,U]])
plot(A[0,:],A[1,:])
hold('on')
plot(B[0,:],B[1,:])
grid

A=matrix([[1,-1.5],[1.5,1]])
B=linalg.inv(A)                         # r= 2/sqrt(13)  < 1  !!!
x=matrix(10*ones((2,1),float)+10*random.rand(2,1))

for k in xrange(10):
  plot(x[0],x[1],'or')
  hold('on')
  sleep(1)
  x=B*x

raw_input('Press Enter to exit:')
```

Seksjon 5.7: Anvendelser på differensiallikninger

I denne seksjonen studeres likninger på formen

$$\begin{bmatrix} x_1'(t) \\ x_2'(t) \end{bmatrix} = A \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix}. \tag{4}$$

Vi løste slike systemer ved å se på egenverdiene og egenvektorene til matrisen A. I læreboka er det angitt formler for basiser for løsningsrommet når egenverdiene er komplekse. Nedenfor har vi tatt med kode for å plotte disse basisfunksjonene for matrisen

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

som er fort gjort å sjekke at har komplekse egenverdier med negativ realdel.

```
# coding=utf-8
from numpy import *
from scitools.easyviz import *

A=array([[-2,-2.5],[10,-2]])
(D,U) = linalg.eig(A)
a = real(D[1])   # a < 0
b = imag(D[1])

y1 = (real(U[:,1])*cos(b*(-2)) - imag(U[:,1])*sin(b*(-2)))*exp(a*(-2))
y2 = (real(U[:,1])*sin(b*(-2)) + imag(U[:,1])*cos(b*(-2)))*exp(a*(-2))
s = arange(-2,3,0.05,float)
for t in s:
  y1 = vstack( (y1,(real(U[:,1])*cos(b*t)
                    - imag(U[:,1])*sin(b*t))*exp(a*t)) )
  y2 = vstack( (y2,(real(U[:,1])*sin(b*t)
                    + imag(U[:,1])*cos(b*t))*exp(a*t)) )

figure(1)
plot(y1[:,0],y1[:,1],'g')
figure(2)
plot(y2[:,0],y2[:,1],'r')
figure(3)
hold('on')

axis([-5,5,-10,10])
plot(y1[:,0],y1[:,1],'g')
plot(y2[:,0],y2[:,1],'r')
r=arange(-5,5,0.5,float)
s=arange(-10,10,0.5,float)
(x,y)=meshgrid(r,s)
u = -2 * x -2.5 * y
v = 10*x-2*y
quiver(x,y,u,v)
raw_input('Press Enter to exit:')
```

Et tilsvarende eksempel for matrisen

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

der egenverdiene istedet er komplekse med positiv realdel, er vist nedenfor.

```
# coding=utf-8
from numpy import *
from scitools.easyviz import *

A=array([[0.5,1],[-1,0]])
(D,U) = linalg.eig(A)
a = real(D[1])   # a > 0
b = imag(D[1])

y1 = (real(U[:,1])*cos(b*(0)) - imag(U[:,1])*sin(b*(0)))*exp(a*(0))
y2 = (real(U[:,1])*sin(b*(0)) + imag(U[:,1])*cos(b*(0)))*exp(a*(0))
s = arange(0,10,0.05,float)
for t in s:
  y1 = vstack( (y1,(real(U[:,1])*cos(b*t)
                    - imag(U[:,1])*sin(b*t))*exp(a*t)) )
  y2 = vstack( (y2,(real(U[:,1])*sin(b*t)
                    + imag(U[:,1])*cos(b*t))*exp(a*t)) )

plot(y1[:,0],y1[:,1],'g')
plot(y2[:,0],y2[:,1],'r')
hold('on')
```

```
figure(1)
axis([-3,6,-10,4])
plot(y1[:,0],y1[:,1],'g')
figure(2)
axis([-3,6,-10,4])
plot(y2[:,0],y2[:,1],'r')
figure(3)
hold('on')

axis([-3,6,-10,4])
plot(y1[:,0],y1[:,1],'g')
plot(y2[:,0],y2[:,1],'r')
r=arange(-3,6,0.4,float)
s=arange(-10,4,0.4,float)
(x,y)=meshgrid(r,s)
u = 0.5 * x + y
v = -x +0*y
quiver(x,y,u,v)
raw_input('Press Enter to exit:')
```

Her har vi brukt to nye funksjoner:

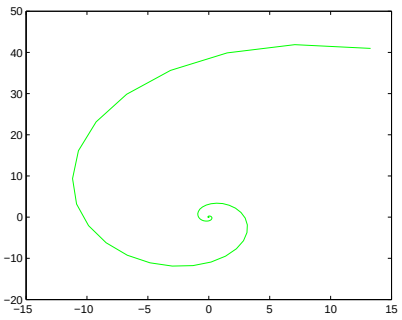
- `real`, som returnerer realdelen til et komplekst tall,
- `imag`, som returnerer imaginærdelen til et komplekst tall.

I Figur 5 og Figur 6 har vi vist plottet av løsningene sammen med vektorfeltet. I MAT1110 lærte vi å tegne vektorfelt ved hjelp av funksjonen `quiver`, som vi også har brukt her. (4) sier at tangenten til løsningene har retning $A \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix}$. I plottene har vi derfor også tegnet vektorfeltet $F(x_1, x_2) = A \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$. Plottet av vektorfeltet er nyttig, siden det er først veh hjelp av dette plottet vi kan se om løsningene beveger seg mot origo, eller om de beveger seg bort fra origo.

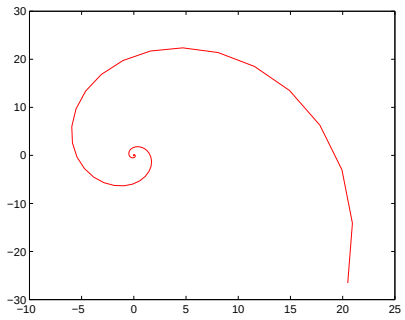
Seksjon 5.8: Iterative estimater for egenverdier

Et viktig ledd i potensmetoden var å skalere vektoren ved hver iterasjon. Hvis vi ikke gjør det vil vektoren vokse mot uendelig, slik vist nedenfor.

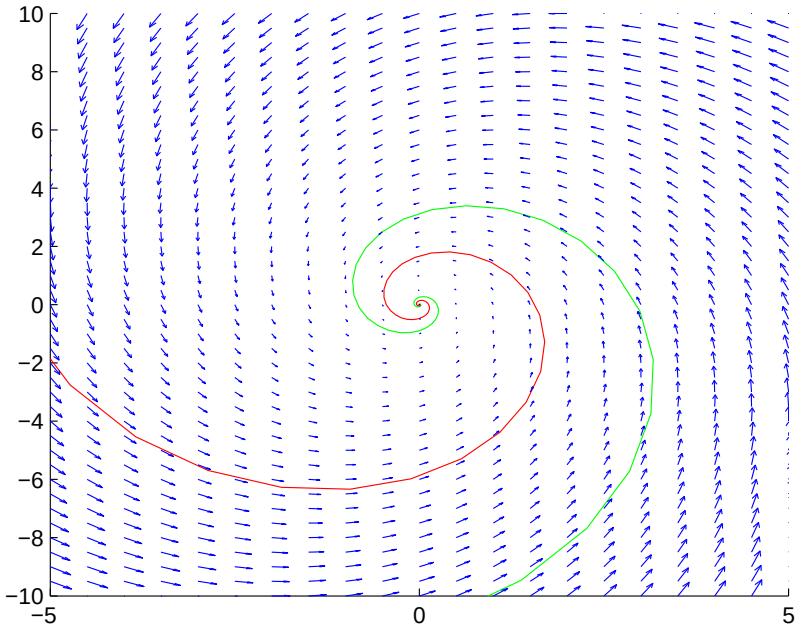
```
# coding=utf-8
from numpy import *
from scitools.easyviz import *
from time import *

L=150
U=150
axis([[-L,U],[-L,U]])
axis('manual')
A=array([[-L,U],[0,0]])
B=array([[0,0],[-L,U]])
plot(A[0,:],A[1,:])
hold('on')
plot(B[0,:],B[1,:])
grid

A=matrix([[0.5,1.5],[1,1]])
x=matrix([[-10],[10]])  # x = startvektor
k=7                        # k= antall iterasjoner
(D,V)=linalg.eig(A)     # Egenverdiene til A er -0.5 og 2
```



(a) Løsningen $y1$

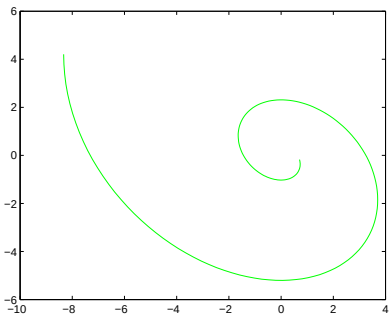


(b) Løsningen $y2$

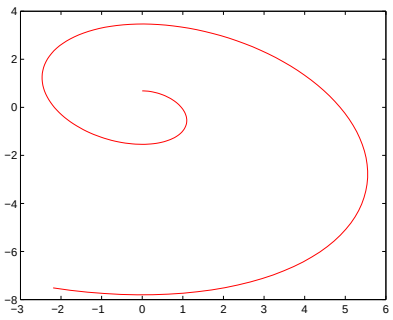


(c) Løsningen sammen med vektorfeltet

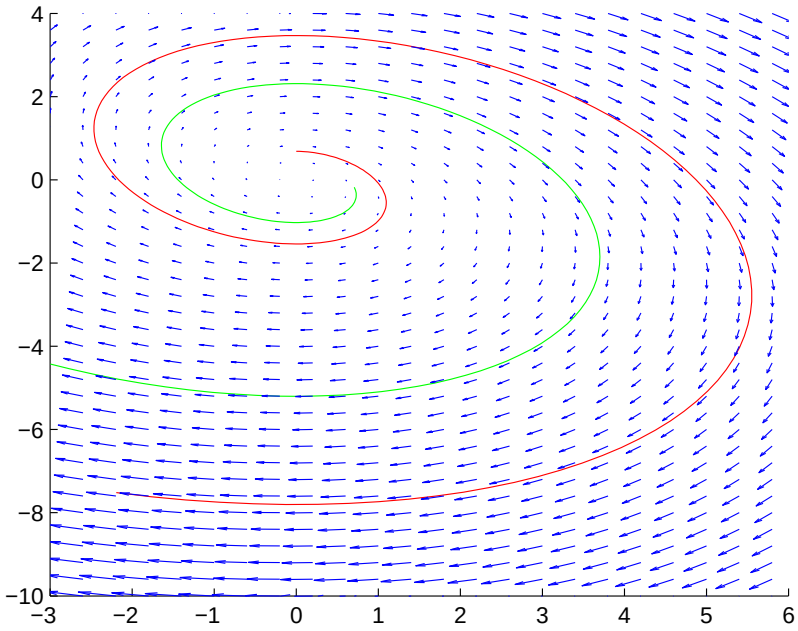
Figur 5: Løsninger av et førsteordens lineært differensiallikningssystem med komplekse egenverdier. Realdelen til egenverdiene er mindre enn 0.



(a) Løsningen $y1$



(b) Løsningen $y2$



(c) Løsningen sammen med vektorfeltet

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

```
plot( [-200*V[0,1],200*V[0,1]],
      [-200*V[1,1],200*V[1,1]],'r')
hold('on')
for j in xrange(k):
  plot([0,x[0]],[0,x[1]],'g')
  hold('on')
  sleep(1)
  x=A*x

raw_input('Press Enter to exit:')
```

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

```
# coding=utf-8
from numpy import *
from scitools.easyviz import *
from time import *

L=12
U=12
axis([[-L,U],[-L,U]])
axis('manual')
A=array([[-L,U],[0,0]])
B=array([[0,0],[-L,U]])
plot(A[0,:],A[1,:])
hold('on')
plot(B[0,:],B[1,:])
grid

A=matrix([[1,2],[-1,-1]]) # A har ingen reelle egenverdier
x=matrix([[-2],[4]])      # x = startvektor
k=9                          # k= antall iterasjoner
(D,V)=linalg.eig(A)       # Egenverdiene til A er i og -i,
                          # så A har ikke en strengt dom. egenverdi

for j in xrange(k):
  plot([0,x[0]],[0,x[1]],'g')
  hold('on')
  sleep(1)
  x=A*x

raw_input('Press Enter to exit:')
```

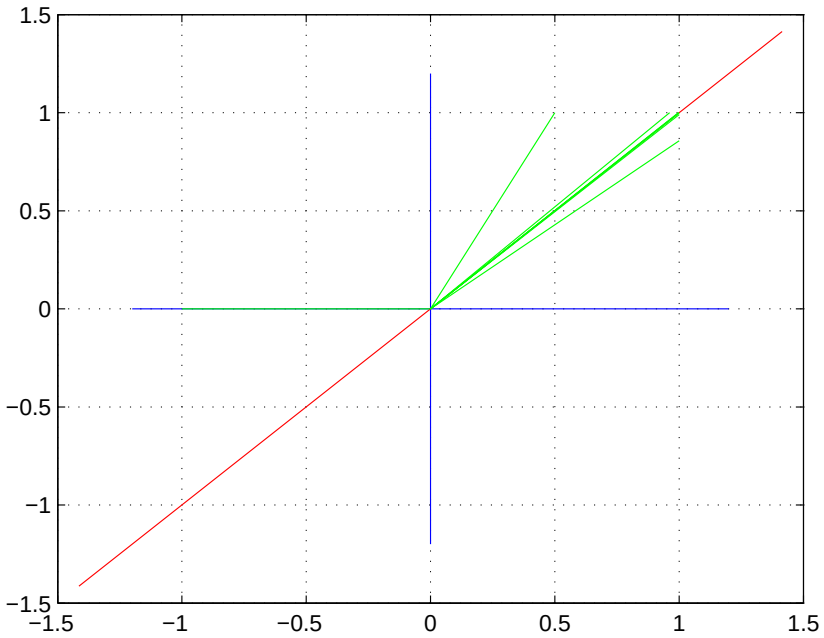
Potensmetoden med skalering kan gjøres slik:

```
# coding=utf-8
from numpy import *

def powermethod(A,x,numtimes):
    """Funksjon som estimerer den dominante egenverdien og
    tilhørende egenvektor for en matrise.
    Input:
    - A: matrisen
    - x: initiell verdi for x
    - numtimes: antall iterasjoner
    Output:
    - mu: estimat for største egenverdi
```

```
    - x: estimat for tilhørende egenvektor
    """
    if A.shape[1] != len(x):
        print 'Feil: dimensjonene til matrisen og vektoren matcher ikke'
        return

    n = len(x)

    for r in xrange(numtimes):
        x = A*x
        maxnr = argmax(abs(y))
        mu = x[maxnr]   # estimat for største egenverdi
        x /= mu         # estimat for tilhørende egenvektor
        error = max(abs(mu*x-A*x))

    return mu,x
```

Metoden returnerer en vektor med estimatene på egenvektorene, og en annen vektor med estimatene på egenverdiene. Koden kjører potensmetoden et maksimalt antall ganger, men avbryter hvis avviket fra å være en egenvektor er mindre enn en angitt toleranse. Her har vi brukt en variant av metoden `max`, som returnerer to verdier: både selve maks-verdien, og den indeksen hvor maksimum inntreffer. En linje er kommentert ut i koden. På denne linjen regnes ut Rayleigh-kvotienten, som også er et estimat på den største egenverdien (faktisk et enda bedre estimat enn μ). Kjører du for eksempel koden

```
# coding=utf-8
from numpy import *
from scitools.easyviz import *
from time import *

L=1.2
U=1.2
axis([[-L,U],[-L,U]])
axis('manual')
plot([-L,U],[0,0])
hold('on')
plot([[0.0,0.0],[-L,U])
grid

A=matrix([[0.5,1.5],[1.0,1.0]])
x=matrix([[-1.0],[0.0]])    # x = startvektor
k=7                            # k= antall iterasjoner
(D,V)=linalg.eig(A)               # Egenverdiene til A er -0.5 og 2

plot([-2.0*V[0,1],2.0*V[0,1]],
     [-2.0*V[1,1],2.0*V[1,1]],'r')
hold('on')

                        # plotter egenrommet til 2 (dominant)
plot([0,x[0]],[0,x[1]],'g')
hold('on')
sleep(1)

for j in xrange(k):
  y=A*x
  r = argmax(abs(y))              # r er komponenten som gir maks
  mu=y[r]
  x=y/mu                          # skalerer med mu
  error=  max(abs(mu*x-A*x))      # angir et mål for feilen
  plot([0,x[0]],[0,x[1]],'g')
  hold('on')
  sleep(1)

raw_input('Press Enter to exit:')
```



Figur 7: 7 iterasjoner i potensmetoden

så vil du få plottet vist i Figur 7. Vi ser her at vektoren gradvis nærmer seg `[1; 1]`, som svarer til egenvektoren med størst tilhørende egenverdi. I dette eksemplet brukte vi matrisen

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

som hare egenverdier 0.5 og 2.

En rett-fram implementasjon av den inverse potensmetoden for matrisen

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

kan bli som følger:

```
# coding=utf-8
from numpy import *

A=array([[0.5,1.5],[1.0,1.0]])
x=array([[-1.0],[0.0]]) # x = startvektor
k=7                       # k= antall iterasjoner

(D,V)=linalg.eig(A)     # Egenverdiene til A er -0.5 og 2
                        # skal "finne" egenverdien -0.5
a=-0.2                  # "tipper" denne verdien
B=A-a*eye(2,2)

for j in xrange(k):
```

```
  y=linalg.solve(B,x)
  maxnr = argmax(abs(y))
  mu = y[maxnr]
  nu=a + 1/mu   # nu er tilnærmet lik egenverdien
  x=y/mu        # x er tilnaermet tilh. egenvektoren
  error =  max(abs(nu*x-A*x)) # angir mål for  feilen
  print error
```

Her kjøres potensmetoden 7 ganger, og i hver iterasjon skriver vi ut feilen På funksjonsform vil en implementasjon kunne se ut som følger, der vi lar maksimalt antall iterasjoner, feiltoleranse, samt en verdi α for hvilken verdi vi skal lete etter egenverdi i nærheten av, være parametre til funksjonen:

```
# coding=utf-8
from numpy import *

def inversepowermethod(alpha,A,x,numtimes):
    """Funksjon som estimerer den egenverdien av en matrise
    A som ligger nærmest alpha.
    Input:
    - alpha: gjetning for egenverdien
    - A: matrisen
    - x: initiell verdi for x
    - numtimes: antall iterasjoner
    Output:
    - nu: estimat for egenverdien
    - x: estimat for egenvektoren
    """
    if A.shape[1] != len(x):
        print 'Feil: dimensjonene til matrisen og vektoren matcher ikke'
        return

    n = len(x)
    M = A - alpha*identity(n)

    for r in xrange(numtimes):
        y = linalg.solve(M,x)
        maxnr = argmax(abs(y))
        mu = y[maxnr]
        nu = alpha + 1/mu
        x = y / mu
        error = max(abs(nu*x-A*x))

    return nu,x
```

Kapittel 6

Seksjon 6.1: Indreprodukt, lengde, og ortogonalitet

Nedenfor har du et eksempel på hvordan du kan bruke indreprodukt til å finne vinkelen mellom to vektorer.

```
x=[1; 1; 1; 1];
y=x+2*rand(4,1);
[x' ; y']                % viser de to vektorene
c=x'*y/(norm(x)*norm(y)) % dette er cosinus til
                         % mellomliggende vinkel
rad=acos(c);             % vinkel i radianer
angle=180*rad/pi         % vinkel i grader
```

Seksjon 6.2: Ortogonale mengder

En teknikk vi skal bruke mye i det følgende for å sjekke om søylene i en matrise `A` utgjør en ortogonal eller ortonormal basis er å skrive

```
A'*A
```

Søylene til `A` er ortogonale (ortonormale) hvis og bare hvis svaret her er en diagonalmatrise (identitetsmatrise) (Teorem 6).

Mange funksjoner i Matlab returnerer ortonormale mengder. For eksempel returnerer `orth()` en matrise med ortonormale søyler. Følgende kode bruker ortogonale projeksjoner til å verifisere Teorem 5 i Seksjon 6.2, som kan sees på som en vektorversjon av Pythagoras læresetning.

```
n=4;
A=rand(n);            % tilfeldig matrise
rank(A)               % Sjekker først at matrisen har full rang.
                      % For tilfeldig genererte matriser vil dette
                      % stort sett alltid være tilfellet.
U=orth(A);            % ortonormal basis for Col A
M=U'*U                % verifiser ortonormale kol.
v=floor(10*rand(n,1)) % en vektor med tilfeldige tall mellom 0 og 10
for j=1:n
    c(j)=v'*U(:,j);               % koef. for U(:,j)
    P(:,j)=c(j)*U(:,j);           % proj. langs U(:,j)
end
P
s=sum(P,2);                       % sum av alle projeksjonene
v_og_sum_av_proj=[v s]            % bør være like!
pythagoras=[norm(v)^2  norm(c)^2] % bør være like!
```

Følgende kode finner avstanden fra en vektor til et underrom L utspent av to vektorer i $\mathbb{R}^3$. For å verifisere at denne avstanden faktisk er minste mulige avstand, velges det ut 1000 forskjellige punkter fra L, og vi viser at alle disse har en avstand større enn minsteavstanden til vektoren.

```
clear all;
u1=[2; 1; 4];
u2=[1; -3; 1/4];  % ortogonale vektorer som utspenner et underrom L
v=[1; 1; 1];
proj=((v'*u1)/(u1'*u1))*u1 + ...
     ((v'*u2)/(u2'*u2))*u2       % Teorem 5
dist_v_L=norm(v-proj);
proj'*(v-proj)                   % Skal bli 0
% trekk andre vektorer i L og finn avstand til v
for j=1:1000
    x=[u1 u2]*rand(2,1);
    dist(j)=norm(v-x);
end
m_d=min(dist)
compare=[dist_v_L m_d] % dist_v_L skal være minst, etter hva vi har lært
plot(sort(dist))       % plott avstander sortert
```

Følgende kode sjekker om en matrise har ortogonale eller ortonormale søyler.

```
clear all
n=5; p=4;
A=rand(n,p) % tilfeldig matrise
B=A'*A      % Sjekker om A har ortonormale kolonner. Hoyst
            % usannsynlig at det er tilfelle !
U=orth(A)   % kolonnene i U gir ortonormal basis for W=Col A
S=U'*U      % verifiser ortonormale kolonner i U

for j=1:4         % Lager en matrise med ortogonale kolonner
  V(:,j)=10*rand(1)*U(:,j);
end

V
T=V'*V      % Sjekker om V har ortogonale kolonner

A=[1 2 1; 1 2 -1] % Et eksempel der kolonnene til A er lin.avh.
B=A'*A
U=orth(A) S=U'*U
```

Seksjon 6.3: Ortogonale projeksjoner

Koden nedenfor beregner den ortogonale projeksjonen av en vektor ned på søylerommet til en matrise på to forskjellige måter, og verifiserer at disse er like. Først beregnes projeksjonene ned på underrommene utspent av hver vektor i en ortogonal basis for søylerommet, og disse legges sammen. Deretter beregnes projeksjonen ved hjelp av matrisen UU^T, der U har ortogonale søyler som utspenner søylerommet til matrisen. Koden sjekker også at projeksjonen faktisk ligger i søylerommet til matrisen, at søylene faktisk er ortogonale, og at vektoren oppfyller Teorem 8 i Seksjon 6.3 (om ortogonal dekomposisjon av en vektor).

```
n=4; p=3;
A=rand(n,p)      % tilfeldig matrise
U=orth(A)        % kolonnene i U gir ortonormal basis for W=Col A
```

```
M=U'*U           % verifiserer ortonormale kolonner

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

P               % kolonnene til P gir proj. av y langs kol. til U
Proj_y=sum(P,2) % gir projeksjonen av y på W= Col A

z= y - Proj_y   % gir komponenten til y langs ortogonale kompl. til W
z'*A            % Sjekker at z er ortogonal på W=ColA

Q=U*U'          % Q er matrisen til projeksjonen på W = Col A
Q*y             % skal gi det samme som Proj_y
Proj_y-Q*y      % skal gi nullvektoren (tilnærmet)
```

Seksjon 6.4: Gram-Schmidt prossessen

Gram-Schmidt prossessen kan implementeres rett fram etter Teorem 11 slik:

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

V vil inneholde en ortogonal basis av søylene til matrisen A etter kallet på funksjonen. Legg merke til at matrisen V initialiseres som matrisen A, siden Gram-Schmidt prossessen trekker flere andre søyler fra søylene til A. Hvis vektorene i A er lineært avhengige ville man merke det i Gram-Schmidt prossessen ved at noen av de søylene man får i V blir null-søyler. I praksis vil ikke dette inntreffe siden avrundingsfeil gjør at man sjelden får noe som er eksakt lik null.

Det som trekkes fra i den innerste løkka i koden over kan sees på som et matriseprodukt. Vi kan derfor skrive koden mer kompakt slik, hvor vi ser sammenhengen med Teorem 10 og projeksjoner.

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

Det er lett å modifisere koden slik at alle de ortogonale vektorene får lengde 1, du trenger bare ta bort et kommentartegn i koden. Det er og lett å lage en kodefnutt som utvider en allerede ortogonal basis til en basis for hele $\mathbb{R}^m$:

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

Her ser vi at vi setter på en identitetsmatrise for å finne hvilke søyler vi skal utvide basisen med, og at vi bruker en variant av `rref` som også returnerer hvilke søyler som er pivotsøyler. Disse søylene blir deretter input til Gram-Schmidt prossessen. Vi skal anvende denne metoden når vi senere programmerer utregning av singulærverdidekomposisjonen.

Følgende kode tester at funksjonen faktisk returnerer en ortogonal basis:

```
A=rand(10,4);    % Tilfeldig matrise
V=gramschmidt(A)
V'*V             % Skal bli diagonal hvis søylene i V er ortogonale.
```

Koden vi har brukt her er numerisk ustabil: på grunn av avrundingsfeil vil koden kunne produsere et fullt sett med vektorer, selv om ikke alle vektorene vi startet med var lineært uavhengige. Funksjonen `orth` tar høyde for slike avrundingseffekter, og er derfor å foretrekke. `orth` sørger i tillegg for at vektorene får lengde 1.

QR-faktorisering

Det står forklart i boka at Gram-Schmidt prossessen er det samme som QR-faktorisering, uten at dette er utdypet helt. For å se dette, skriv Teorem 11 på formen

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

Setter vi

$$\begin{aligned} A &= [\mathbf{x}_1 \quad \mathbf{x}_2 \quad \cdots \quad \mathbf{x}_n] \\ Q_1 &= [\mathbf{v}_1 \quad \mathbf{v}_2 \quad \cdots \quad \mathbf{v}_n], \end{aligned}$$

og R til å være den øvretriangulære matrisen med elementer

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

ser vi at (7) ikke er noe annet enn produktet $A = Q_1 R_1$ tatt søyle for søyle. Denne faktoriseringen oppfyller alle egenskapene til en QR-faktorisering, med unntak av at søylevektorene i Q_1 ikke nødvendigvis har lengde 1. Dette kan vi oppnå ved å skrive $A = Q_1 D^{-1} D R_1$, der D er diagonalmatrisen med $D_{ii} = \|\mathbf{v}_i\|$. Regner vi ut $Q = Q_1 D^{-1}$ får vi at søyle i kan skrives $\mathbf{u}_i = \frac{\mathbf{v}_i}{\|\mathbf{v}_i\|}$, som svarer til vektorene fra Gram-Schmidt prossessen i normalisert form. Regner vi ut $R = DR_1$ får vi at

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

Vi ser nå at $A = QR$ oppfyller alle egenskaper til QR-faktoriseringen. Med andre ord, hvis vi normaliserer vektorene etter hvert steg i Gram-Schmidt prossessen, så vil $\mathbf{x}_j \cdot \mathbf{u}_i$ (som vi jo da regner ut underveis i Gram-Schmidt prossessen) bli elementene i matrisen R, mens de ortonormale vektorene som fremkommer i Gram-Schmidt prossessen blir elementene i matrisen Q. Koden for QR-faktorisering blir dermed slik:

```
function [Q,R]=qrfact(A)
  [m,n]=size(A);
  Q = A;
  R = zeros(n,n);
  for j=1:n
    R(1:(j-1),j) = Q(:,1:(j-1))'*Q(:,j);
    Q(:,j) = Q(:,j) - Q(:,1:(j-1))*R(1:(j-1),j);
    R(j,j) = norm(Q(:,j));
    Q(:,j) = Q(:,j)/R(j,j);
  end
```

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

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

```
A=rand(10,4)    % tilfeldig matrise
[Q,R]=qrfact(A) % R Skal bli øvretriangulær med positive elem. på diag.
Q'*Q            % Blir identitsetsmatrisen hvis V-søyler er ortonormale
Q*R             % Vi skal ha A=Q*R
```

I boka står det at du kan bruke QR-faktoriseringen til å finne egenverdiene til en matrise. For å begrunne dette nærmere, se på følgende kode:

```
A = rand(6,6) % tilfeldig matrise
for k=1:10
  [Q,R]=qrfact(A);
  R
  A = Q'*A*Q; % Svarer til å erstatte Q*R med R*Q (vis det)
end
```

Kjører du denne koden vil du se at matrisen `R` blir mer og mer lik en diagonalmatrise. Man kan faktisk vise at elementene på diagonalen konvergerer mot egenverdiene til `A`. Beviset for dette går vi ikke inn på i MAT1120.

Seksjon 6.5: Minste kvadraters metode

I Seksjon 6.5 ble det forklart at vi kunne finne minste kvadraters løsninger på to måter:

1. først regne ut projeksjonen ned på underrommet, og deretter løse likningssystemet med denne som høyreside,

2. sette opp normallikningene direkte og løse disse.

Koden nedenfor viser eksempler på begge delene.

```
A=[3 5 1; 1 1 1; -1 5 -2; 3 -7 8]
b=[1;0;0;0]
R=rref(A)

% Metode 1
fprintf('** ortonormal basis\n');
W=orth(A) % W'*W blir identitetsmatrisen
fprintf('** projeksjon\n');
proj_b=(b'*W(:,1))*W(:,1)+(b'*W(:,2))*W(:,2)+(b'*W(:,3))*W(:,3)
x=A \ proj_b
fprintf('** sjekker projeksjon\n');
linkomb1=A*x
norm(linkomb1-b)

% Metode 2
fprintf('** løser normalikningene\n');
x_MK=A'*A \ A'*b
linkomb2=A*x_MK
```

Seksjon 6.8: Anvendelser av indreproduktrom

En kompakt form for å løse normallikningene for vektedede minste kvadraters metode ser slik ut:

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

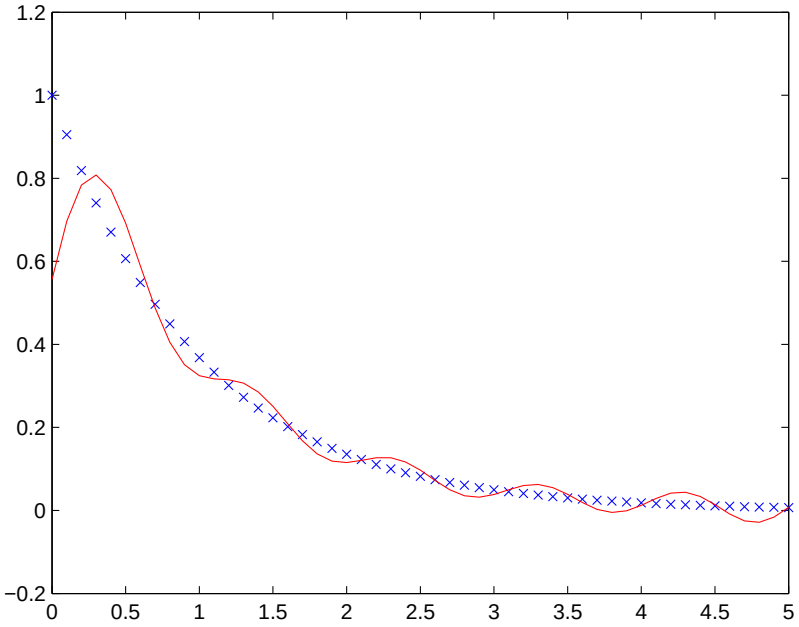
Her er `W` vektmatrisen, `X` designmatrisen, og `y` observasjonsvektoren, på samme måte som i Seksjon 6.6. I en ekstraoppgave skulle vi finne en vektet minste kvadraters linje for punktene $(2,3)(3,2),(5,1),(6,0)$, gitt at det andre og tredje punktet er dobbelt så sikre som det første og siste. Kode for å regne ut minste kvadraters linje og vektet minste kvadraters linje, og plotte dem sammen med punktene, blir som følger.

```
W = diag([ 1 2 2 1 ]);
A = [ 1 2 ; 1 3 ; 1 5 ; 1 6 ];
y = [ 3 2 1 0 ]';
utenvekting = A'*A \ A'*y;
medvekting = (W*A)'*W*A \ (W*A)'*W*y;
plot(A(:,2),y,'kx')     % Plotter observasjonene
hold on
t=linspace(0,7,100);
plot(t,utenvekting(1)+utenvekting(2)*t,'g'); % Uten vekting
plot(t,medvekting(1)+medvekting(2)*t,'b');   % Med vekting
legend('punkter','Uten vekting','Med vekting')
```

Sørg spesielt her for at du gjenkjenner den større sikkerheten i det andre og tredje punktet fra plottet for den vektede minste kvadraters linjen.

Fourier approksimasjoner

I Seksjon 6.8 lærte vi hvordan vi kunne bruke integraler til å regne ut Fourier approksimasjonene til funksjoner. I praksis er funksjonene gitt ved et sett datapunkter, slik at vi bare regner ut approksimasjoner av disse integralene. Koden nedenfor viser hvordan en slik approksimasjon kan regnes ut i praksis.



Figur 8: Plottet generert av koden for å regne ut en Fourier approksimasjon

```
clear all;
t=0:0.1:5;    % Funksjonen er 0 på (5,2*pi)
y=exp(-t);    % Legg inn data i y med y(t)=e^(-t)
N=length(t);  % lengden av datavektoren
kmax=6;       % antall ledd (frekvenser) i approksimasjonen

% Beregn Fourier koeffisienter (tilnærmer integralene)
% ved å bruke formel (7) i seksjon 6.8
for k=1:kmax
  A(k) = sum( y.*cos(k*t)*0.1/pi ); % 0.1 er bredden på partisjonen
  B(k) = sum( y.*sin(k*t)*0.1/pi );
end
A0 = sum( y.*0.1/pi ); A0 = A0/2;

% Rekonstruer funksjonen (signalet)
for n=1:N
  ynew(n) = A0 + ...
    sum( A(1:kmax).*cos((1:kmax)*t(n))+B(1:kmax).*sin((1:kmax)*t(n)) );
end

plot(t,y,'x')    % Plott data (altså y)
hold on
plot(t,ynew,'r') % Plott Fourier approksimasjonen (rekonstruksjonen)
hold off
```

Approksimasjonen til integralet kan regnes ut på andre måter også, for eksempel med Euler metode. Legg merke til at funksjonen er 0 på $(5, 2\pi)$. I denne koden kan det være lurt å eksperimentere med forskjellige verdier for `kmax` (antall ledd i Fourier-approksimasjonen. Test også gjerne med andre funksjoner, for eksempel $y = t$. Approksimasjonen du får ved å kjøre koden over er tegnet opp i Figur 8

Kapittel 7

Seksjon 7.2: Symmetriske matriser og kvadratiske former

I MAT1110 gikk vi gjennom kommandoer for å tegne tredimensjonale grafer. La oss bruke disse til å tegne kvadratiske former som er positiv definit, negativ definit, og indefinit. Den positiv definite formen $z = 3x^2 + 7y^2$ kan vi plotte slik:

```
r = -1:0.05:1;
s = -1:0.05:1;
[x,y] = meshgrid(r,s);
z = 3*x.^2 + 7*y.^2;
mesh(x,y,z)
```

Plottet er vist i Figur 9, der vi også har vist den negativ definite formen $z = -3x^2 - 7y^2$, og den indefinite formen $z = 3x^2 - 7y^2$ (for å få disse plottene trenger du kun å forandre den nest siste linjen med koden over). For å plotte løsningen til $3x^2 + 7y^2 = 3$ kan du erstatte siste linje ovenfor med koden

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

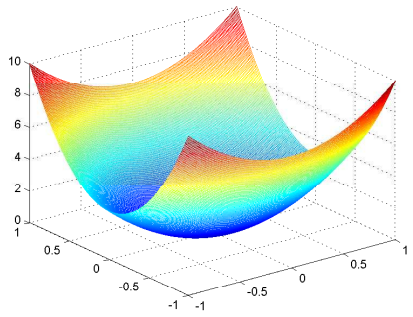
(vektoren `[3 3]` indikerer at det bare skal plottes en nivåkurve, nemlig for $z = 3$). Det tilsvarende plottet finner du i Figur 10. Siste parameteren til `contour` kan også spesifisere hvor mange nivåkurver vi skal plotte. Hvis vi utelater det vil Matlab selv velge antall nivåkurver som plottes. Det viser seg at nivåkurvene kan hjelpe oss med å avgjøre om formen er indefinit eller ikke. I Figur 11 har vi plottet nivåkurvene til den positiv definite formen over, og den indefinite formen over. Som vi ser er nivåkurvene til den positive definite formen ellipser, for den indefinite formen er de hyperbler. Nivåkurvene som er hyperbler vet vi at har asymptoter. Disse er også synlige på figurene.

Seksjon 7.4: Singulærverdidekomposisjonen

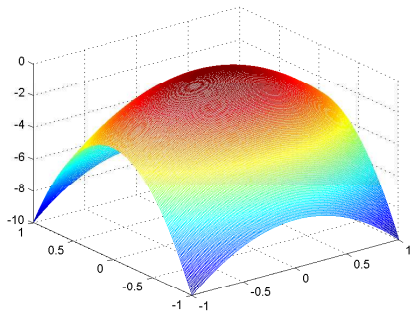
Hvis du skriver

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

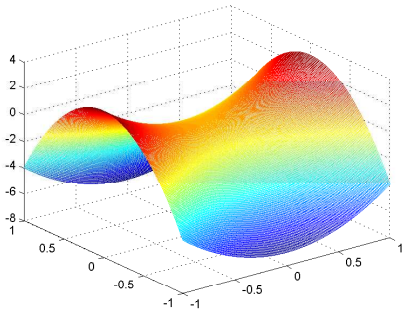
vil Matlab returnere singulærverdidekomposisjonen til `A`. `U` og `V` vil være som i Teorem 10 i Seksjon 7.4, `Sigma` svarer til Σ i Teorem 10. Man kan også regne ut singulærverdidekomposisjonen manuelt ved hjelp av resultatene i Seksjon 7.4. Anta at vi har en 4×4-matrise med singulærverdier $40, 20, 10, 0$ (singulærverdiene til A får du ved å skrive `sqrt(eig(A'*A))`). Vi kan da produsere singulærverdidekomposisjonen slik:



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

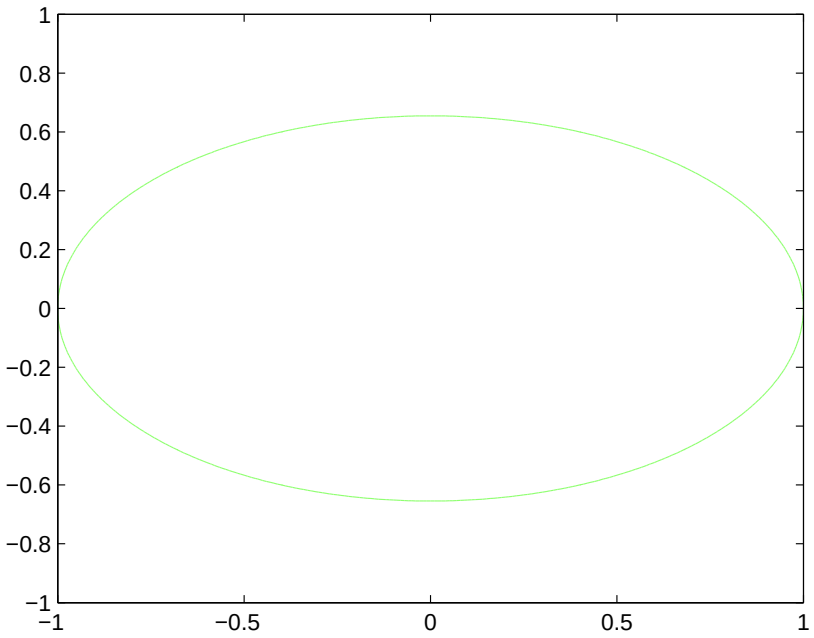


(b) Den negativ definite formen $z = -3x^2 - 7y^2$

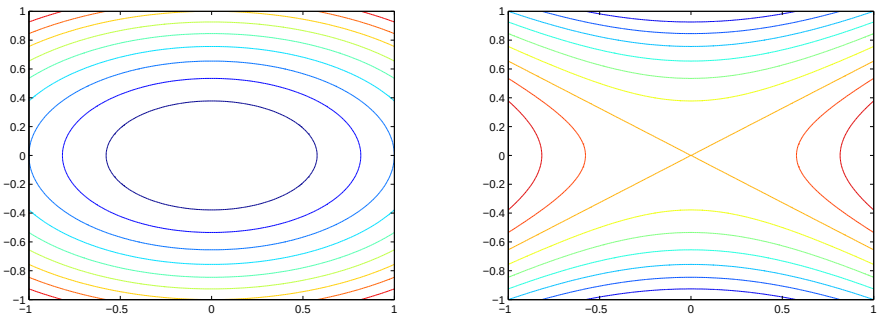


(c) Den indefinite formen $z = 3x^2 - 7y^2$

Figur 9: Plott av positiv definit, negativ definit, og indefinit kvadratiske former



Figur 10: Plott av nivåkurven $3x^2 + 7y^2 = 3$



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

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

Figur 11: Plott av nivåkurver for en positiv definit og en indefinit kvadratisk form

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

V = [ v1 v2 v3 v4 ] % Lager en ortogonal matrise
U = [ (1/40)*A*v1 (1/20)*A*v2 (1/10)*A*v3 ]

u4 = null( U' ); % Gir en vektor ortogonal på søylene i U
U = [ U u4 ]     % Utvider til en ortogonal matrise
Sigma = diag( [40,20,10,0] )
```

En liten perturbasjon kan forandre rangen, selv om forandringen i singulærverdiene tilsynelatende er små. Det kan sees fra følgende kode:

```
A=magic(4)
r=rank(A)
R=rref(A)
s=svd(A)              % singulære verdier

A(4,4)=1.00000000001  % liten perturbasjon
r=rank(A)
R=rref(A)
s=svd(A)
```

Teorem 10 gir en kokebokoppskrift på hvordan U, V, Σ i singulærverdidekomposisjonen kan regnes ut. Alle stegene kan automatiseres med Matlab, men det krever noen nye funksjoner. Koden nedenfor gjør samme jobb som den innebygde `svd`-funksjonen:

```
function [U,Sigma,V]=svdfact(A)
  [m,n]=size(A);
  [V,D]=eig(A'*A);                          % Step 1

  singvals=sqrt(diag(D)); % Vector with singular values
  r=length(find(singvals~=0));
  sortedsingvals = sort(singvals,1);
  sortedsingvalsr = sortedsingvals(1:r);

  Sigma=zeros(m,n);                         % Step 2
  Sigma(1:r,1:r)=diag(sortedsingvalsr);
  V=V(sortedsingvals,:);

  U=A*V(:,1:r)*inv(diag(sortedsingvalsr)); % Step 3
  U=gramschmidtutvid(U);
```

Her brukes funksjonen `find` til å finne singulærverdier som er større enn 0,

Selv om koden for funksjonen `svdfact` er den som er nærmest kokebokoppskriften i Teorem 10, så er ikke `svd` implementert på denne måten, siden koden er numerisk ustabil.

Kapittel 2*

Seksjon 2.5: Faktoriseringer av matriser

LU-faktoriseringen kan implementeres lett i Python hvis du har en matrise hvor du slipper å bytte om på rader når du bringer matrisen på trappeform. Algoritmen nedenfor vil da gjøre jobben for oss, siden U i LU-faktoriseringen ikke er noe annet enn (den ikke-reduserte) trappeformen til matrisen:

```
# coding=utf-8
from numpy import *

def lufact(A):
    """Funksjon som LU-faktoriserer en matrise.
    Input: A: matrisen
    Output:
    - L: nedretriangulær matrise
    - U: øvretriangulær matrise
    """
    if A.shape[0] != A.shape[1]:
        print 'Feil: matrisen må være kvadratisk'
        return

    n = A.shape[0]
    U = A.copy()                # Skal bli en øvre-triangulær matrise
    mults = zeros((n,n),double) # Matrise som lagrer radoperasjonene

    for l in xrange(n):
        for k in xrange(l+1,n):
            mults[k,l] = U[k,l]/U[l,l]
            U[k,:] -= mults[k,l] * U[l,:]
    # mults[k,l] svarer til multiplum av rad l vi har trukket fra rad k

    L = identity(n,double) # Skal bli til en nedre-triangulær matrise
    for l in xrange(n-1,-1,-1):
        for k in xrange(n-1,l,-1):
            # Vi gjør inverse radoperasjoner i motsatt rekkefølge
            L[k,:] += mults[k,l] * L[l,:]

    return L,U
```

Koden nedenfor verifiserer at funksjonen `lufact` faktisk gir en LU-faktorisering av matrisen `A`.

```
>>> from numpy import *
>>> from lufact import *
>>> A=matrix(random.rand(3,3))
>>> print A
```

```
[[ 0.327354    0.20220912  0.86794529]
 [ 0.33031007  0.21050351  0.99944044]
 [ 0.67609602  0.89439694  0.36074888]]
>>> (L,U)=lufact(A)
>>> print L
[[  1.           0.           0.        ]
 [  1.00903017   1.           0.        ]
 [  2.06533603  73.70705851   1.        ]]
>>> print U
[[  3.27354005e-01   2.02209121e-01   8.67945292e-01]
 [  5.55111512e-17   6.46840566e-03   1.23657455e-01]
 [ -4.09156367e-15   0.00000000e+00  -1.05462771e+01]]
>>> L*U
matrix([[ 0.327354  ,  0.20220912,  0.86794529],
        [ 0.33031007,  0.21050351,  0.99944044],
        [ 0.67609602,  0.89439694,  0.36074888]])
```

Når vi genererer en tilfeldig matrise så er det stor sannsynlighet for at vi slipper å bytte om på rader ved radreduksjon (hvorfor?). Internt i Python er LU-faktorisering implementert slik at radpermutasjoner foretas under radredusering hvis dette er nødvendig, slik at LU-faktorisering vil virke under alle omstendigheter.

Register

Bibliografi

[1] K. Bryan and T. Leise. The $25,000,000,000 eigenvector: The linear algebra behind Google. *SIAM Review*, 48:569–581, 2006.

[2] A. N. Langville and C. D. Meyer. *Google's pagerank and beyond: the science of search engine rankings.* Princeton University Press, 2006.