Python for MAT1110 (revidert versjon våren 2010)

Klara Hveberg, Tom Lindstrøm, og Øyvind Ryan

12. januar 2010

Innhold

Innledning

Dette lille notatet gir en kort innføring i Python med tanke på behovene i MAT 1110. Hensikten er å gi deg litt starthjelp slik at du kommer i gang med enkel programmering Avansert bruk må du lære deg andre steder (se lenkene på kursets hjemmeside). Du kan også få hjelp fra innebygde hjelpemenyer og demo'er (kommandoen `help()` i et Python Shell gir deg en oversikt over hva som er tilgjengelig). Notatet forutsetter at du sitter ved maskinen og taster inn kommandoene vi gjennomgår, men det kan kanskje være lurt å ha bladd fort gjennom på forhånd slik at du vet hva som vil bli tatt opp.

I de første seksjonene i notatet (1-7) konsentrerer vi oss hovedsakelig om hvordan du kan utføre enkle beregninger. I de siste seksjonene (8-11) ser vi på litt mer avanserte ting som hvordan du tar vare på (utvalgte deler av) arbeidet ditt, hvordan du programmerer, og hvordan du lager dine egne rutiner som du kan kalle på igjen og igjen. Det kan hende at du har bruk for noe av dette stoffet før du er ferdig med de første sju seksjonene. Heftet inneholder også en seksjon om bildebehandling. Denne er ikke pensum i kurset, men kan bli brukt i forbindelse med obligatoriske oppgaver.

Du vil i dette heftet se mange kodeeksempler. Vi har prøvd å gi mange kommentarer i koden, slik at det skal være mulig å lese seg til hva som skjer. I koden vil du se at en kommentar begynner med `#`. Det tolkes da slik at kommentaren løper til slutten av linjen, slik at man ikke trenger å angi hvor kommentaren slutter. Kommentarene er ikke en del av selve programmet som blir kjørt. Det er en god vane å legge inn forklarende kommentarer i koden. Du vil se at funksjoner i dette heftet inneholder noen kommentarer helt i starten på hvordan input og output for funksjonen skal tolkes. Dette er helt vanlig programmeringsskikk og i mange andre språk.

Legg merke til at forskjellig bakgrunnsfarge blir brukt for forskjellige kodebiter. Dette gjøres for å skille mellom operativsystemkommandoer, shellkommandoer, ufullstendige kodebiter, og fullstendige programmer. Fargeleggingen er lik den som er brukt i kurset INF1100. Koden for alle funksjonene i heftet er også å finne på kurssidene i filer med samme navn som funksjonen.

Til slutt i heftet finner du noen større lab-oppgaver. Disse er brukt som oppgaver i kurset de siste årene. Du finner også noen enkle oppvarmingsoppgaver til labene. 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 fasiten. Øvelsene er 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.

Det er også laget mye annet stoff ved Universitetet i Oslo for MAT1110 enn det som finnes i dette heftet. Læreboka inneholder for eksempel også en del Matlab-tips.

Vi retter en takk til Karl Leikanger og Magnar Bugge for mye av Python-programmene i dette heftet.

1. Det aller enkleste

Du åpner Python som et hvilket som helst annet program. I Unix kan du skrive

```
ipython &
```

Du får da opp et "prompt". Vi skal indikere et slikt prompt med `>>>`, selv om det ikke er akkurat slik `ipython` viser det. I kodeeksemplene i dette heftet vil vi bare vise promptet hvis vi trenger skille mellom våre egne kommandoer, og returverdiene fra dem. Etter promptet kan du skrive en kommando du vil ha utført. Skriver du f.eks.

```
>>> 3+4
```

og skifter linje, blir svaret

```
7
```

, og gir deg et nytt "prompt" til neste regnestykke (prøv!). Du kan prøve de andre standard regneoperasjonene ved å skrive inn etter tur

```
3-4
3*4
3.0/4
3**4
```

(husk å avslutte med et linjeskift for å utføre kommandoene). Den vanlige prioriteringsordningen mellom regneoperasjonene slik du er vant til fra avanserte lommeregnere brukes. Derfor gir kommandoen

```
8.0**2.0/3.0
```

noe annet enn

```
8.0**(2.0/3.0)
```

Legg merke til at `2/3` (heltallsdivisjon) og `2.0/3.0` er forskjellige ting i Python.

For å få tilgang til mye brukte matematiske funksjoner i Python bør vi i koden først inkludere linjen

```
from math import *
```

Hvis vi bare er interessert i et par matematiske funksjoner kan vi skrive

```
from math import sin, exp, sqrt, abs
```

Python kjenner alle vanlige (og mange uvanlige!) funksjoner, så skriver du

```
sin(0.94)
```

blir svaret sinus til 0.94 radianer. Vær nøye med multiplikasjonstegn: Skriver du `15sin(0.94)`, blir svaret en feilmelding. du må skrive `15*sin(0.94)` for å få regnet ut $15 \sin 0.94$.

En funksjon du må passe litt på, er eksponentialfunksjonen e^x som du må skrive som `exp(x)`. Vil du regne ut $e^{2.5}$, skriver du altså `exp(2.5)` (og ikke potenstegnet over). I tillegg kan det være greit å vite at kvadratrotfunksjonen heter `sqrt` og at absoluttverdifunksjonen heter `abs`. Skriver du

```
exp(sqrt(abs(-0.7)))
```

regnes ut $e^{\sqrt{|-0.7|}}$. De andre standardfunksjonene heter det du er vant til: `sin`, `cos`, `tan`, `log`. I tillegg heter arcusfunksjonene `arcsin`, `arccos`, `arctan`.

Vår gamle venn π kalles for `pi`. Kommandoen

```
pi
```

gir den numeriske tilnærmingen 3.14159265... til π.

NOTE

Regner du ut `sin(pi)` får du deg kanskje en overraskelse. Istedenfor 0 blir svaret `1.2246e-16`. Dette skyldes at beregninger primært skjer med avrundede tall.

Det finnes også funksjoner som runder av tall på flere måter:

```
floor(a)  # Runder ned til nærmeste hele tall,
ceil(a)   # Runder opp til nærmeste hele tall,
round(a)  # Runder av til nærmeste hele tall.
```

Skal du bruke et tall flere ganger i en beregning, kan det være greit å gi det et navn. Tilordner du først verdier til variablene `a` og `b`, kan du gange dem sammen slik:

```
a=0.3124
b=2.41
a*b
```

La oss avslutte denne innledende seksjonen med noen knep det kan være greit å vite om (blir det for mye på en gang, får du heller komme tilbake til denne delen senere). Du kan ikke endre på en kommando som allerede er blitt eksekvert. Har du laget en liten trykkfeil, må du derfor føre inn kommandoen på nytt. Dette kan du gjøre ved klipping-og-liming, men du kan også bruke piltastene opp og ned.

Får du problemer med en kommando og ønsker å avbryte, trykker du Ctrl-c (altså kontrolltasten og c samtidig).

Oppgaver til Seksjon 1

Oppgave 1

Regn ut: $2.2 + 4.7$, $5/7$, 3^2, $\frac{2 \cdot 3 - 4^2}{13 - 2.2^2}$

Oppgave 2

Regn ut verdiene, og sjekk at resultatene er rimelige: e^1, $\sqrt{16}$, $\cos \pi$, $\sin \frac{\pi}{6}$, $\tan \frac{\pi}{4}$, $\arcsin \frac{1}{2}$, $\arctan 1$.

Oppgave 3

Definer $x = 0.762$ og $y = \sqrt{9.56} + e^{-1}$ og regn ut: $x + y$, xy, $\frac{x}{y}$ og $\sin x^2 y$.

2. Matriser og vektorer

Støtten for matriser i Python finner vi i pakken `numpy`, slik at vi må skrive

```
from numpy import *
```

for å få tilgang til funksjonalitet for matriser (alternativt kan vi importere bare de klassene fra `numpy` vi trenger). I en del korte kodeeksempler er denne linjen droppet i det følgende. Husk at forskjellige moduler i Python kan implementere funksjoner med like navn. Hvis du skal bruke en funksjon i modulen `numpy`, og denne funksjonen også finnes i `math` (dette er tilfelle for flere funksjoner, og det er også tilfelle for andre moduler, slik som `numpy` og `scitool`), så er det viktig at du importerer `numpy` etter at du importerer `math`: Python vil velge funksjonen fra den siste modulen som du importerte. De viktigste klassene vi trenger fra `numpy` er `matrix` og `array`. Vi skal først se på `matrix`, og komme tilbake til `array` senere. Dersom vi ønsker å definere matrisen

$$A = \begin{pmatrix} 2 & -3 & 1 \\ 0 & 1 & -3 \\ -4 & 2 & 1 \end{pmatrix}$$

skriver vi

```
A=matrix([[2,-3,1],
          [0,1,3],
          [-4,2,1]])
```

(vi bruker hakeparenteser for å definere matriser). Synes du dette tar for stor plass, kan du isteden skrive

```
A=matrix([[2,-3,1],[0,1,3],[-4,2,1]])
```

Regneoperasjoner for matriser utføres med enkle kommandoer. Har du lagt inn matrisene A og B, kan du skrive vil kommandoene

```
A+B   # regner ut summen A+B
A-B   # regner ut differensen A-B
A*B   # regner ut produktet AB
```

Dette forutsetter at matrisene har riktige dimensjoner slik at regneoperasjonene er definert (hvis ikke blir svaret en feilmelding av typen: `Objects are not aligned`). Eksempler på andre viktige kommandoer er (noen av disse befinner seg i pakken `numpy.linalg`)

```
3*A  # Ganger matrisen A med tallet 3
A**7 # Regner ut sjuende potensen til A (A ganget med seg selv 7 ganger)
A.T  # Finner den transponerte av A
linalg.inv(A)       # Finner den inverse matrisen
linalg.det(A)       # Regner ut determinanten til A
D,V = linalg.eig(A) # Finner egenverdier og egenvektorer til A
                    # Søylene i matrisen V er egenvektorene til A
                    # D er et array med egenverdiene til A
                    # Egenvektorer/egenverdier kommer i samme rekkefølge:
                    # første egenverdi tilhører første egenvektor osv.
```

Her er et eksempel på en kjøring:

```
>>> from numpy import *
>>> A = matrix([[3,1,2],     # taster inn matrisen A
                [3,0,4],
                [2,1,4]])
>>> u,v = linalg.eig(A)      # ber Python finne egenvektorer og -verdier
>>> print u                  # ber Python skrive ut u
[ 6.6208359   1.43309508 -1.05393098]
>>> print v                  # ber Python skrive ut v
[[-0.50695285 -0.79426125  0.18076968]
 [-0.60226696  0.03095748 -0.97598223]
 [-0.61666305  0.60678719  0.12157722]]
```

Dette forteller oss at A har egenvektorer

$$\begin{pmatrix} -0.5070 \\ -0.6023 \\ -0.6167 \end{pmatrix}, \quad \begin{pmatrix} -0.7943 \\ 0.0310 \\ 0.6068 \end{pmatrix} \quad \text{og} \quad \begin{pmatrix} 0.1808 \\ -0.9760 \\ 0.1216 \end{pmatrix}$$

med egenverdier henholdsvis 6.6208, 1.4331 og -1.0539. Vær oppmerksom på at egenvektorene alltid normaliseres slik at de har lengde én, og at dette kan få dem til å se mer kompliserte ut enn nødvendig. Husk også at de tre siste kommandoene vi har sett på (`inv`, `det`, og `eig`) forutsetter at A er en kvadratisk matrise.

Rangen til en matrise får du ved hjelp av funksjonen `rang`, som du kan finne i `MAT1120lib`. Legg merke til at `rank` betyr noe annet enn `rang` i Python, siden den førstnevnte returnerer 1 for vektorer, 2 for matriser, e.t.c.. Med andre ord er `rank` dessverre to helt forskjellige ting i Matlab og Python. Kommandoen returnerer (den estimerte) rangen til `A`. Vi er vant til å radredusere matrisen A og telle antall pivotsøyler for å finne rangen. Dersom verdiene i matrisen er sensitive til avrundingsfeil, og vi setter en datamaskin til å gjøre utregningene ved radreduksjonen, vil ofte avrundingsfeil kunne føre til at det ser ut som om matrisen har full rang, selv om det egentlig ikke er tilfelle. I stedet kan datamaskinen beregne rangen ved å telle antall *singulærverdier* som er ekte større enn null! (I MAT1120 lærer vi ranger er lik antall positive singulærverdier σ_j til A). Python bruker denne fremgangsmåten, og antar at $\sigma_j > 0$ dersom $\sigma_j > \epsilon$ for en angitt toleranse $\epsilon > 0$. Dersom man ønsker å forandre den forhåndsdefinerte toleransen ϵ kan man angi dette ved å sende med en ekstra parameter til kommandoen ved å skrive

```
r = rang(A, $\epsilon$)
```

hvor ϵ er den ønskede toleransen.

Vektorer blir oppfattet som spesialtilfeller av matriser. En radvektor $\mathbf{b} = (b_1, b_2, \ldots, b_n)$ er altså en $1 \times n$-matrise. Vektoren $\mathbf{b} = (-2, 5, 6, -4, 0)$ lastes derfor inn med kommandoen

```
b=matrix([[-2,5,6,-4,0]])
```

En søylevektor

$$\mathbf{c} = \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{pmatrix}$$

oppfattes som en $m \times 1$-matrise. Du kan laste inn vektoren $\mathbf{c} = \begin{pmatrix} 7 \\ -3 \\ 2 \end{pmatrix}$ ved for eksempel å skrive

```
c=matrix([[7],[-3],[2]])
```

Når du skriver vektorer er det viktig å ha tenkt igjennom om du ønsker radvektorer eller søylevektorer; du kan ikke regne med at det blir forstått at du mener en søylevektor når du skriver en radvektor! Legg merke til at hvis

$$a = (a_1, a_2, \ldots, a_n)$$

er en radvektor, vil den transponerte være søylevektoren

$$a^T = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix}$$

Har du lastet inn en matrise A og en søylevektor $\mathbf{c}$ kan du regne ut produktet $A\mathbf{c}$ ved å skrive

```
A*c
```

Vi har spesialkommandoer for å regne ut skalar- og vektorprodukt av vektorer. Disse funksjonene forutsetter at objektene er av typen `array` i `numpy`

```
dot(a,b)
cross(a,b)
```

Den siste kommandoen forutsetter naturlig nok at vektorene er tredimensjonale.

Oppgaver til Seksjon 2

Oppgave 1

La

$$A = \begin{pmatrix} 1 & 3 & 4 & 6 \\ -1 & 1 & 3 & -5 \\ 2 & -3 & 1 & 6 \\ 2 & 3 & -2 & 1 \end{pmatrix} \text{ og } B = \begin{pmatrix} 2 & 2 & -1 & 4 \\ 2 & -1 & 4 & 6 \\ 2 & 3 & 2 & -1 \\ -1 & 4 & -2 & 5 \end{pmatrix}.$$

Regne ut; A^T, B^T, $(AB)^T$, A^TB^T, B^TA^T, A^{-1}, B^{-1}, $(AB)^{-1}$, $A^{-1}B^{-1}$, $B^{-1}A^{-1}$. Blir noen av resultatene like?

Oppgave 2

Vi har to vektorer $a = (1, -2, 3)$ og $b = (2, 2, -4)$. Sjekk at lengdene til vektorene kan finnes ved kommandoene `linalg.norm(a)` og `linalg.norm(b)`. Forklar at du kan finne vinkelen mellom vektorene ved kommandoen

```
arccos(dot(a,b)/(linalg.norm(a)*linalg.norm(b)))
```

Finn vinkelen.

Oppgave 3

La

$$A = \begin{pmatrix} 1 & 2 & -1 \\ 3 & -1 & 0 \\ -4 & 0 & 2 \end{pmatrix}$$

Finn A^{-1}, A^T og $\det(A)$. Finn også egenverdiene og egenvektorene til A.

3. Komponentvise operasjoner

Ovenfor har vi sett på de algebraiske operasjonene som brukes mest i vanlig matriseregning. Det finnes imidlertid andre operasjoner slik som det komponentvise matriseproduktet (eller *Hadamard-produktet*) der produktet av matrisene

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \cdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \quad \text{og} \quad \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \cdots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{pmatrix}$$

er

$$\begin{pmatrix} a_{11}b_{11} & a_{12}b_{12} & \ldots & a_{1n}b_{1n} \\ a_{21}b_{21} & a_{22}b_{22} & \ldots & a_{2n}b_{2n} \\ \vdots & \vdots & \ldots & \vdots \\ a_{m1}b_{m1} & a_{m2}b_{m2} & \ldots & a_{mn}b_{mn} \end{pmatrix}$$

Selv om slike komponentvise operasjoner ikke brukes mye i lineær algebra, er de viktige i Python siden Python bruker matriser til mer enn tradisjonell matriseregning. Python støtter slike operasjoner ved hjelp av klassen `array`. der alle operasjoner som var tillatt for klassen `matrix` også eksisterer, men er i stedet komponentvise. Et objekt av typen `matrix` kan konverteres til et `array` ved hjelp av metoden `asarray` i `numpy`. Skriver du f.eks.

```
A = asarray(A)
B = asarray(B)
A*B     # Komponentvis multiplikasjon
A/B     # Komponentvis divisjon
A**B    # Komponentvis potens
```

vil vi få regnet ut matrisene der den ij-te komponenten er $a_{ij}b_{ij}$, $\frac{a_{ij}}{b_{ij}}$, og $a_{ij}^{b_{ij}}$, respektive. Vi kan også bruke disse operasjonene når den ene matrisen erstattes med et tall; f.eks. vil

```
3.0/A   # Komponentvis divisjon av 3
A**2    # Komponentvis kvadrat
```

produsere matrisene med komponenter $\frac{3}{a_{ij}}$ og ${a_{ij}}^2$, respektive. En del kjente funksjoner er også tilgjengelige på matriser, er allerede komponentvise, og kan kombineres: Skriver vi

```
sin(A)       # Komponentvis sinus
exp(A)       # Komponentvis eksponentialfunksjon
exp(A*B**2)  # Kombinasjon av komponentvise operasjoner
```

får vi matrisene med komponenter $\sin a_{ij}$, $\exp a_{ij}$, og $e^{a_{ij}b_{ij}^2}$, respektive. En annen viktig komponentvis funksjon er random-generering av tall:

```
from numpy import random

random.rand(m,n)   # Tilfeldig generert mxn-matrise
random.rand(n)     # Tilfeldig generert vektor av lengde n
random.rand()      # Tilfeldig generert tall
```

Alle tall blir her tilfeldig generert mellom 0 og 1. Når vi skal illustrere at en setning holder kommer vi ofte til å lage en tilfeldig generert matrise, og vise at setningen holder for denne matrisen. Som et eksempel, studer følgende kode:

```
A = matrix(random.rand(4,4))
B = matrix(random.rand(4,4))

print (A+B).T - A.T - B.T
print (A*B).T - B.T * A.T
```

Ser du hvilke to kjente setninger som blir illustrert her hvis du kjører koden?

Som vi skal se i neste seksjon, er de komponentvise operasjonene spesielt nyttige når vi skal tegne grafer, men det finnes også noen enklere anvendelser som vi kan se på allerede nå. Det finnes egne kommandoer for å finne summer og produkter. Har du lagt inn en vektor c, kan du finne summen og produktet til komponentene i c ved å skrive

```
sum(c)
prod(c)
```

Skal vi finne summen av de 10 første naturlige tallene, kan vi dermed skrive

```
a=array([1,2,3,4,5,6,7,8,9,10])
print sum(a)
```

Hvis vi også vil ha summen av de 10 første kvadrattallene, kan vi skrive

```
sum(a**2)
```

Dette er vel og bra så lenge vi har korte summer, men hva hvis vi ønsker summen av de hundre første kvadrattallene? Det blir ganske kjedelig å taste inn vektoren $(1, 2, 3, \ldots, 100)$ for hånd. Koden nedenfor forenkler dette:

```
a=range(1,101)   # Definerer a til å være vektoren (1,2,3,..., 100)
a=range(1,101,2) # Definerer a til vektoren med alle oddetall < 100
a=range(b,c,h)   # Definerer a til å være vektoren med b først,
                 # deretter b+h, b+2h osv. inntil vi kommer til c.
                 # Siste komponent vil være største tallet på formen
                 # a+nh som er mindre enn c.
```

Skal vi regne ut summen av kvadratene av alle partall opptil 100, kan vi altså skrive

```
a=range(2,101,2)
sum(array(a)**2)
```

Til slutt nevner vi kommandoen `linspace` som ofte er nyttig hvis intervallet vi skal dele opp har "uregelmessige" endepunkter. Skriver vi f.eks.

```
a=linspace(0,pi,50)
```

får vi definert `a` som en vektor med 50 komponenter der første komponent er 0, siste komponent er π og avstanden mellom én komponent og den neste allltid er den samme.

Oppgaver til Seksjon 3

Oppgave 1

Legg inn disse n-tuplene: $(1, -9, 7, 5, -7)$, $(\pi, -14, e, 7/3)$, $(1, 3, 5, 7, 9, \ldots, 99)$, $(124, 120, 116, 112 \ldots, 4, 0)$.

Oppgave 2

Legg inn tuppelet $(1, 2, 4, 8, 16, \ldots 4096)$ og finn summen.

Oppgave 3

Legg inn $(0, \frac{1}{100}, \frac{2}{100}, \ldots, 1)$. Bruk denne partisjonen til å lage en nedre og øvre trappesum for funksjonen $f(x) = x^2$ over intervallet $[0, 1]$ (husk MAT1100!), og regn ut disse summene. Sammenlign med integralet $\int_0^1 x^2 \, dx$.

Oppgave 4

Regn ut produktet $\frac{1}{2} \cdot \frac{3}{4} \cdots \frac{99}{100}$.

4. Grafer

For å plotte trenger vi å importene pakken `scitools.easyviz`:

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

Anta at $x = (x_1, x_2, \ldots, x_n)$ og $y = (y_1, y_2, \ldots, y_n)$ er to n-tupler. Kommandoen plot kan brukes på flere måter til å tegne kurver:

```
plot(x,y)  # lager plott der (x_1,y_1) forbindes med (x_2,y_2),
           # (x_2,y_2) forbindes med (x_3,y_3) osv.
           # Forbindelselinjene er heltrukne linjer.
plot(x)    # som plot(x,y), men i stedet med punktene
           # (1,x_1),(2,x_2), (3,x_3) osv.
```

Vær oppmerksom på at figurene kommer i et eget vindu , og at dette vinduet kan gjemme seg bak andre vinduer!

Vi kan utnytte metoden ovenfor til å plotte grafer av funksjoner. Trikset er å velge punktene som skal forbindes til å være (tettliggende) punkter på funksjonsgrafen. Hvis vi skal tegne grafen til $\sin \frac{1}{x}$ over intervallet $[-\pi, \pi]$ velger vi først et tuppel som gir oppdeling av intervallet i mange små biter, deretter velger vi tuppelet av tilhørende funksjonsverdier, før vi til slutt plotter punktene

```
x=linspace(-pi,pi,100)
y=sin(1.0/x)
plot(x,y)               # plotter punktene
```

Hva hvis vi har lyst til å tegne en graf til i samme vindu? Da kaller vi først kommandoen `hold` før vi taster inn den nye funksjonen (nedenfor er denne $z = x^2 \sin \frac{1}{x}$), og plotter den:

```
hold('on')
z=x**2*sin(1.0/x)
plot(x,z)
```

Nå som vi har flere grafer i samme vindu kan det være lett å gå i surr på hva som er hva. Skriver du

```
legend('Dette er den første grafen','Dette er den andre grafen')
```

vil hver graf bli angitt med den assosierte teksten, slik at de kan skilles fra hverandre. Grafene du lager vil fortsette å komme i samme vindu inntil du gir kommandoen

```
hold('off')
```

Det hender at du vil lage en ny figur i et nytt vindu og samtidig beholde den gamle i det gamle vinduet. Du kan aktivere et eksisterende (eller nytt) vindu med kommandoen

```
figure(2)
```

Etter denne kommandoen vil alle plott havne i vindu 2.

Vi skal se på noen tilleggskommandoer som kan være nyttige å kunne. Dersom du selv vil velge størrelsen på vinduet grafene skal vises i, kan du bruke en kommando av typen

```
axis([-pi, pi, -2.5, 2.5]) # Setter grensene på aksene
axis('square')             # Gir et kvadratisk plottevindu
axis('equal')              # Gir deg samme målestokk på begge aksene.
```

Du kan gi figuren en tittel, og sette navn på aksene ved å skrive

```
title('Grafen til en vakker funksjon')
xlabel('x-akse')
ylabel('y-akse')
```

Ønsker du tekst på figuren, kan du skrive

```
text(a, b, 'tekst')
```

der (a, b) er koordinatene til det punktet på figuren der teksten skal begynne. Det er lett å angi farge på grafene, samt måte grafen skal tegnes:

```
plot(x,y,'r')          # en rød graf
plot(x,y,'g')          # gir en grønn graf
plot(x,y,'r--')        # gir en rød, stiplet graf
plot(x,y,'g:')         # gir en grønn, prikket graf
```

Av og til ønsker man å ha flere grafer side om side i den samme figuren. Kommandoen

```
subplot(m,n,p)
```

deler vinduet i $m \times n$-delvinduer, og sørger for at den neste plot-kommandoen blir utført i det p-te delvinduet.

Hvis du ønsker å ta vare på en figur for å kunne bruke den i et annet dokument senere, kan det være lurt å lagre den som en eps-fil (encapsulated postscript format), eller i et annet format. Dette kan du gjøre slik:

```
hardcopy('figur.eps')    # Skriv innhold figurvindu til fil figur.eps
hardcopy('figur.eps',color=True) # Samme som over, men i farger
hardcopy('figur.png')    # Figuren blir nå i stedet lagret med PNG
```

Oppgaver til Seksjon 4

Oppgave 1

Legg inn 6-tuplene $a = (3, 1, -2, 5, 4, 3)$ og $b = (4, 1, -1, 5, 3, 1)$ og utfør kommandoen `plot(a,b)`. Utfør også kommandoene `plot(a)` og `plot(b)`, og bruk `hold on` til å sørge for at de to siste figurene kommer i samme vindu.

Oppgave 2

Bruk kommandoen `plot` til å lage en enkel strektegning av et hus.

Oppgave 3

Tegn grafen til $f(x) = x^3 - 1$ over intervallet $[-1, 1]$. Legg så inn grafen til $g(x) = 3x^2$ i samme koordinatsystem, og velg forskjellig farge på de to grafene.

Oppgave 4

Tegn grafen til funksjonen $f(x) = \sin \frac{1}{x}$ over intervallet $[-1, 1]$. Bruk først skrittlengde $\frac{1}{100}$ langs x-aksen. Tegn grafen på nytt med skrittlengde $\frac{1}{10000}$.

5. Tredimensjonale grafer

Det finnes flere kommandoer som kan brukes til å lage tredimensjonale figurer. Vi skal se på et par av de enkleste. Grafen til en funksjon $z = f(x, y)$ tegnes ved å plotte funksjonsverdiene over et nett av gitterpunkter i xy-planet.

Kommandoen `mesh` lager konturer av grafen ved å forbinde plottepunktene på grafen med rette linjestykker. Resultatet ser ut som et fiskegarn med knuter i plottepunktene. Varianten `meshc` tegner i tillegg nivåkurver til funksjonen i xy-planet. La oss se hvordan vi kan tegne grafen til funksjonen $z = f(x, y) = xy\sin(xy)$ over området $-4 \leq x \leq 4, -2 \leq y \leq 2$.

Vi lager først en oppdeling av de to intervallene vi er interessert i. I koden vår har vi valgt å dele opp begge intervallene i skritt med lengde 0.05, men du kan godt velge en finere eller grovere oppdeling. Neste skritt er å lage et rutenett av oppdelingene våre, regne ut funksjonsverdiene, og til slutt plotte:

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

r=arange(-4,4,0.05,float)       # Lag en oppdeling av
s=arange(-2,2,0.05,float)       # intervallene vi er interessert i
x,y = meshgrid(r,s,sparse=False,indexing='ij') # Lag et rutenett
                                               # av oppdelingene våre
z=x*y*sin(x*y)                  # Regn ut funksjonsverdiene
mesh(x,y,z)                     # selve plottingen
```

Grafen kommer opp i et eget figurvindu akkurat som for todimensjonale figurer. Velger du `surf(x,y,z)` istedenfor `mesh(x,y,z)`, får du en graf der flateelementene er fargelagt. Ved å holde musknappen inne kan dreie flaten i rommet. Dette er ofte nødvendig for å få et godt inntrykk av hvordan grafen ser ut! Ønsker du bare å få ut nivåkurvene til en flate, kan du bruke kommandoen:

```
contour(x,y,z)
contour(x,y,z,n) # n er antall nivåkurver du ønsker
```

Kommandoen `plot3` er nyttig når du skal plotte parametriserte kurver i tre dimensjoner. Skriver du

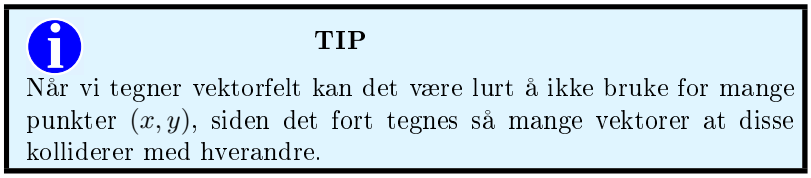
```
t=linspace(0,10*pi,100)
x=sin(t)
y=cos(t)
z=t
plot3(x,y,z)
```

tegnes kurven $\mathbf{r}(t) = (\sin t, \cos t, t)$ for $t \in [0, 10\pi]$.

Det finnes også støtte for å tegne vektorfelt. Skriver du

```
quiver(x,y,u,v)
```

vil et vektorfelt bli tegnet opp der `x`- og `y`-vektorene spesifiserer punktene der vektorfeltet skal tegnes opp, og der `u`- og `v`-vektorene er vektorfeltets verdi i disse punktene. Grafisk betyr dette at det i ethvert punkt (x, y) blir tegnet opp en vektor (u, v).



TIP

Når vi tegner vektorfelt kan det være lurt å ikke bruke for mange punkter (x, y), siden det fort tegnes så mange vektorer at disse kolliderer med hverandre.

Oppgaver til Seksjon 5

Oppgave 1

Tegn grafene til disse funksjonene: $f(x, y) = x^2y^2$, $g(x, y) = \frac{\sin x}{y^2} + x^2$, $h(x, y) = \sin(e^{x+y})$. Vri på flatene for å få et best mulig inntrykk.

Oppgave 2

Tegn kurvene $\mathbf{r}_1(t) = (t, t^2, \sin t)$ og $\mathbf{r}_2(t) = (\sin^2 t, \cos^2 t, e^{-t})$. Vri på koordinatsystemet for å se kurvene best mulig.

Oppgave 3

Bruk kommandoen `plot3` til å lage en tredimensjonal strektegning av en terning.

6. Mer om matriser

Siden matriser står så sentralt i Python, kan det være greit å vite hvordan man kan manipulere matriser på en effektiv måte. Spesielt nyttig er det å kunne legge sammen elementer i en matrise på forskjellige måter. Dette kan vi gjøre med funksjonen `sum` på følgende måter (`A` er en matrise):

```
A.sum(0)      # Returnerer en radvektor der hvert element er
              # summen av tilsvarende søyle
A.sum(1)      # Returnerer en søylevektor der hvert element er
              # summen av tilsvarende rad
A.sum()       # Returnerer summen av alle elementene i matrisen
```

Dette kan brukes for eksempel til å regne ut middelverdien i en matrise, og det vi kaller for *Frobenius normen* til en matrise (ikke pensum):

```
m,n = shape(A)        # m og n er antall rader og søyler i matrisen
A.sum()/(m*n)         # Regner ut middelverdien i matrisen
sqrt(sum(A**2))       # regner ut Frobenius normen til en matrise
                      # (ikke pensum)
```

På samme måte kan vi regne ut maksimum av elementene i en matrise på forskjellige måter:

```
A.amax(0)     # Returnerer en radvektor der hvert element er
              # maksimum av tilsvarende søyle
A.amax(1)     # Returnerer en søylevektor der hvert element er
              # maksimum av tilsvarende rad
A.amax()      # Returnerer maksimum av alle elementene i matrisen
```

Minimum går på samme måte. Vi har også funksjonene `max` og `min`, som virker litt annerledes.

Mens `sum` lager en vektor/skalar fra en matrise, så finnes det også kommandoer som går den motsatte veien. Et eksempel er kommandoen `diag`, som lager en diagonalmatrise med elementene i input-vektoren på diagonalen.

Du har blant annet mange muligheter til å sette sammen og ta fra hverandre matriser. Dersom A er 3×3-matrisen

$$A = \begin{pmatrix} 2 & -3 & 1 \\ 0 & 1 & -3 \\ -4 & 2 & 1 \end{pmatrix}$$

og B er 3×2-matrisen

$$B = \begin{pmatrix} 7 & 4 \\ 2 & 5 \\ -1 & 3 \end{pmatrix}$$

kan vi skjøte sammen A og B til 3×5-matrisen

$$C = \begin{pmatrix} 2 & -3 & 1 & 7 & 4 \\ 0 & 1 & -3 & 2 & 5 \\ -4 & 2 & 1 & -1 & 3 \end{pmatrix}$$

ved å gi kommandoen

```
C=hstack((A,B))
```

Du kan skrive ut komponentene til en matrise med enkle kommandoer:

```
A[0,1]    # skriver ut elementet i første rad, andre søyle i A
A[0,:]    # skriver ut hele den første raden i A
A[:,1]    # skriver ut hele den andre søylen i A
```

Du kan også bruke kolon-notasjon til å plukke ut andre deler av en matrise. Starter vi med 3×5-matrisen C ovenfor, vil

```
C[1:3,0:4]
```

gi deg undermatrisen

$$\begin{pmatrix} 0 & 1 & -3 & 2 \\ -4 & 2 & 1 & -1 \end{pmatrix}$$

som består av komponentene som ligger fra annen til tredje rad (det er dette `1:3` står for), og fra første til fjerde søyle (det er dette `0:4` står for). Kommandoen

```
C[ ix_([0,2],[1,4])]
```

gir deg matrisen

$$\begin{pmatrix} -3 & 4 \\ 2 & 3 \end{pmatrix}$$

bestående av elementene som ligger i første og tredje rad og annen og femte søyle. Legg merke til den noe spesielle formen her, der kommandoen `ix_` blir kalt. Du kan også bruke denne notasjonen til å bytte om på radene eller søylene til en matrise. Skriver du

```
C[[2,0,1],:]
```

får du ut matrisen

$$\begin{pmatrix} -4 & 2 & 1 & -1 & 3 \\ 2 & -3 & 1 & 7 & 4 \\ 0 & 1 & -3 & 2 & 5 \end{pmatrix}$$

der radene i den opprinnelige matrisen C nå kommer i rekkefølgen $3, 1, 2$. Kommandoen

```
C[:,[2,0,1,4,3]]
```

vil på tilsvarende måte bytte om på søylene i C.

Det finnes mange andre spesialkommandoer for å manipulere matriser. Her følger noen kommandoer for å generere matriser man ofte kan ha nytte av. Noen av de første har vi mer eller mindre vært borti tidligere, men vi gjentar dem her for å ha alt samlet på et sted:

```
x=linspace(2,4,12) # 12-tuppelet med 12 tall jevnt fordelt fra 2 til 4
x=range(1,11)      # 10-tuppelet med alle heltallene fra 1 til 10
x=range(1,11,2)    # 5-tuppelet med annethvert tall fra 1 til 10
                   # 2 angir steglengden
A=zeros((3,4))     # 3x4-matrisen med bare nuller
A=ones((3,4))      # 3x4-matrisen med bare enere
A=eye(3,4)         # 3x4-matrisen med enere på hoveddiagonalen,
                   # nuller ellers
A=random.rand(3,4) # 3x4-matrise med tilfeldige tall mellom 0 og 1
triu(A)            # Lager en øvre triangulær matrise fra A, d.v.s. alle
                   # elementene under diagonalen er blir satt til 0
tril(A)            # Lager en nedre triangulær matrise fra A, d.v.s. alle
                   # elementene over diagonalen er blir satt til 0
random.permutation(5) # Lager tilfeldig permutasjon av tall fra 1 til 5
```

Som et enkelt eksempel tar vi med at

```
A=eye(3,4)
```

gir matrisen

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Oppgaver til Seksjon 6

Oppgave 1

Skriv inn matrisene

$$A = \begin{pmatrix} 2 & -3 & 1 \\ 4 & 1 & -5 \\ 1 & 2 & -1 \end{pmatrix}$$

og

$$B = \begin{pmatrix} -1 & 3 & 2 \\ 4 & 5 & 1 \\ 0 & 2 & -1 \end{pmatrix}$$

og gjennomfør operasjonene

```
C=hstack((A,B))
C[1,3]
C[:,[1,2]]
C[[0,2],2:5]
```

Oppgave 2

Bruk matrisen C fra forrige oppgave. Undersøk hva som skjer med matrisen når du skriver

```
C[:,2]=2*C[:,2]
C[[0,2],:]=4*C[[0,2],:]
```

Oppgave 3

Undersøk hva kommandoen

```
vstack((A,B))
```

gjør når A og B er to matriser.

Oppgave 4

Bruk kommandoene

```
random.rand(2,2)
random.rand(3,3)
random.rand(4,4)
random.rand(5,5)
```

til å generere "tilfeldige" matriser. Finn egenverdiene og egenvektorene i hvert enkelt tilfelle. Hvor mange forskjellige egenverdier ser det ut til at en typisk $n \times n$-matrise har?

7. Radoperasjoner

Denne seksjonen forutsetter at du kjenner til radoperasjoner for matriser, og at du kan bringe en matrise til (redusert) trappeform. Har du ikke lært dette ennå, kan du trygt hoppe over denne seksjonen så lenge.

Det finnes en egen kommando som bringer en matrise på redusert trappeform. Den heter `rref` (for *reduced row echelon form*) , og finnes i modulen `MAT1120lib`, som altså må importeres eksplisitt. Denne modulen laget vi ved Universitetet i Oslo, siden klassen `linalg` i `numpy` ikke inneholder alle funksjoner som trengs innen lineær algebra, slik som `rref`. Andre slike funksjoner som `MAT1120lib` inneholder er `null` (som finner en basis for nullrommet til en matrise), og `orth` (som finner en basis for søylerommet til en matrise). Nedenfor viser vi hvordan `rref` er implementert i `MAT1120lib`:

```
def rref(M,*args):
    """
    rref(M,toleranse)
    M: 2-D matrise. Største neglisjerbare element.
    Bestemt av algoritme i programmet dersom ingen ting spesifiseres.

    Implementering av Gauss-Jordan algoritmen.
    Bringer matrise M (2-D array) på redusert trappeform.

    Bruker lignende algoritme som i matlabs rref funksjon.
    Fjerner små pivotelementer (<tol) som kan skyldes avrundingsfeil.
    For penere utskrifter settes alle elementer < tol lik 0
    i arrayen som returneres.

    OBS: Denne algoritmen kan man ikke stole blindt på.
    Kan for mange ''nesten singulære'' matriser gi helt feil svar.
    """
    # Vi forsikrer oss om at M er et array-objekt med float elementer.
    M=matrix(M,float)

    if rank(M)!=2:
        print 'Feil: Input må være et todimensjonalt objekt.'
        return

    (m,n) = M.shape
    # Leser inn minste neglisjerbare element.
    if len(args)<2:
        # Finner største avrundingsfeil
        maxabs = amax(abs(M).sum(0))
        toleranse = 2.22044*10**(-16) * max([m,n]) * maxabs
    else:
        toleranse=args[1]

    soylenr=-1 # holder orden på hvilken pivot søyle
    for y in xrange(m):
        soylenr += 1
        while soylenr<n:
            # Finn raden blant de gjenværende radene,
            # der 1. element har størst absoluttverdi.
            Mabsolute=abs(M[y:,soylenr])
            storsteelement=Mabsolute.argmax()+y

            # Sjekk om største element er mindre enn toleransen.
            # I så fall: slett raden.
            # Ellers: bruk raden som pivot-rad, og avbryt loopen.
            if Mabsolute[storsteelement-y,0]<toleranse:
                M[storsteelement:,soylenr]=0
                soylenr+=1
            else:
                # bytt om rader
                M[[y,storsteelement],:] = M[[storsteelement,y],:]
                break

        if soylenr<n:
            # Normaliser pivotrad
            M[y,soylenr:]/=M[y,soylenr]

            # Fjern de resterende elementene i søylen
            for z in xrange(m):
                if z!=y:
                    M[z,soylenr:] -= M[y,soylenr:] *  M[z,soylenr]

    # Setter alle tall mindre enn toleranse lik 0
    M_bool=abs(M)>toleranse
    M =matrix(asarray(M)*asarray(M_bool)) # Komponentvis produkt

    return M
```

`MAT1120lib` er brukt i flere av eksemplene senere i heftet . Starter vi med matrisen

$$A = \begin{pmatrix} 1 & 4 & -5 & 7 & 3 & 2 & 8 & -1 \\ 2 & 3 & -2 & 5 & 3 & 3 & 7 & 8 \\ -1 & 1 & 2 & 3 & -1 & 4 & 4 & 4 \end{pmatrix}$$

leder kommandoen

```
B=rref(A)
```

til den reduserte trappeformen

$$B = \begin{pmatrix} 1 & 0 & 0 & -0.48 & 0.88 & -0.2 & -0.04 & 3.36 \\ 0 & 1 & 0 & 2.12 & 0.28 & 1.8 & 2.76 & 2.16 \\ 0 & 0 & 1 & 0.2 & -0.2 & 1 & 0.6 & 2.6 \end{pmatrix}$$

Du kan også foreta nøyaktig de radoperasjonene du vil. Lar du

$$C = \begin{pmatrix} 2 & -3 & 1 & 7 & 4 \\ 0 & 1 & -3 & 2 & 5 \\ -4 & 2 & 1 & -1 & 3 \end{pmatrix}$$

være matrisen fra forrige seksjon, har vi allerede sett hvordan vi kan bytte om på to rader ved å bruke en kommando av typen

```
C[[3,2,1],:]
```

Denne kommandoen gir deg en ny matrise der rad 1 og 3 er ombyttet. Taster du

```
C[0,:]=2*C[0,:]
```

blir svaret

```
matrix([[ 4, -6,  2, 14,  8],
        [ 0,  1, -3,  2,  5],
        [-4,  2,  1, -1,  3]])
```

Den har altså ganget den første raden med 2. Skriver du så

```
C[2,:]=C[2,:]+C[0,;]
```

legges den første raden til den siste, og du får

```
matrix([[4,-6, 2,14,8],
        [0,1,-3,2,5],
        [0,-4,3,13,11]])
```

Du kunne ha slått sammen disse operasjonene til én ved å skrive

```
C[2,:]=C[2,:]+2*C[0,:]
```

(dersom du prøver den siste kommandoen, må du huske å tilbakeføre C til den opprinnelige verdien først!)

Man kan lure på hva som er vitsen med å kunne foreta radoperasjoner "for hånd" på denne måten når kommandoen `rref` er innebygd. Man støter imidlertid ofte på matriser med spesiell struktur, f.eks. svært mange nuller. Da er det ofte mer effektivt selv å programmere hvilke radoperasjoner som skal gjøres enn å kjøre en standardkommando som `rref` som ikke tar hensyn til strukturen til matrisene.

Radoperasjoner kan brukes til å løse lineære likningssystemer, men du har også muligheten til å finne løsningen med én kommando. Dersom A er en ikke-singulær, kvadratisk matrise og b er en vektor, kan du løse vektorlikningen $Ax = b$ ved kommandoen

```
linalg.solve(A,b)
```

Velger vi for eksempel

$$A = \begin{pmatrix} 1 & -1 & 4 & 3 \\ 2 & 1 & -4 & 5 \\ 6 & 3 & 1 & -2 \\ 3 & 3 & -2 & 4 \end{pmatrix}$$

og

$$b = \begin{pmatrix} 1 \\ 3 \\ 0 \\ -2 \end{pmatrix}$$

gir kommandoen over svaret

$$x = \begin{pmatrix} 1.3537 \\ -2.4784 \\ -0.7023 \\ -0.0076 \end{pmatrix}$$

Oppgaver til Seksjon 7

Oppgave 1

Skriv inn matrisene

$$A = \begin{pmatrix} 2 & -3 & 1 & 1 & 4 \\ 4 & 1 & -5 & 2 & -1 \\ 1 & 2 & -1 & 2 & -1 \end{pmatrix}$$

og

$$B = \begin{pmatrix} -1 & 3 & 2 & 3 & 1 \\ 4 & 5 & 1 & 4 & 4 \\ 0 & 2 & -1 & -3 & -1 \end{pmatrix}$$

og gjennomfør operasjonene `rref(A)` og `rref(B)`.

Oppgave 2

Løs likningssystemet

$$\begin{aligned} x + 3y + 4z + 6u &= 3 \\ -x + y + 3z - 5u &= 5 \\ 2x - 3y + z + 6u &= -2 \\ 2x + 3y - 2z + u &= 0 \end{aligned}$$

Oppgave 3

Finn den generelle løsningen av likningssystemet

$$\begin{aligned} 2x - 3y + 4z + 5u &= 6 \\ x + y + 3u &= 4 \\ 4x - 3y + 2z + 3u &= 5 \end{aligned}$$

ved hjelp av kommandoen `rref`.

Oppgave 4

Skriv matrisen

$$A = \begin{pmatrix} 0 & 3 & -2 & 7 & 1 & 2 & 7 & 0 \\ 2 & 0 & 1 & -4 & 3 & 2 & 0 & 4 \\ -1 & 3 & 4 & -1 & 2 & -5 & 6 & 2 \end{pmatrix}$$

på redusert trappeform ved å utføre radoperasjonene én for én (du får altså ikke lov til å bruke `rref` eller en lignende kommando). Beskriv den generelle løsningen til likningssystemet som har A som utvidet matrise.

Oppgave 5

a)

Figuren viser spillebrettet for et enkelt terningspill. Spillerne starter på "Start" og kaster en vanlig terning for å flytte. De trenger eksakt riktig antall øyne for å gå i mål (står de på 11 og kaster en femmer, 'spretter" de altså ut til 10 ved å telle på denne måten 12-mål-12-11-10). La t_i være antall kast du må regne med å gjøre før du går i mål (dvs. det forventede antall kast) dersom du står på felt i.

Mål	12	11	10	9	8	7
Start	1	2	3	4	5	6

Forklar hvorfor

$$\begin{aligned}
t_1 &= \frac{1}{6}t_2 + \frac{1}{6}t_3 + \frac{1}{6}t_4 + \frac{1}{6}t_5 + \frac{1}{6}t_6 + \frac{1}{6}t_7 + 1 \\
t_2 &= \frac{1}{6}t_3 + \frac{1}{6}t_4 + \frac{1}{6}t_5 + \frac{1}{6}t_6 + \frac{1}{6}t_7 + \frac{1}{6}t_8 + 1 \\
&\vdots \quad \vdots \quad \vdots \qquad \vdots \qquad \vdots \qquad \vdots \qquad \vdots \\
t_6 &= \frac{1}{6}t_7 + \frac{1}{6}t_8 + \frac{1}{6}t_9 + \frac{1}{6}t_{10} + \frac{1}{6}t_{11} + \frac{1}{6}t_{12} + 1 \\
t_7 &= \frac{1}{6}t_8 + \frac{1}{6}t_9 + \frac{1}{6}t_{10} + \frac{1}{6}t_{11} + \frac{1}{6}t_{12} + 1 \\
t_8 &= \frac{1}{6}t_9 + \frac{1}{6}t_{10} + \frac{1}{6}t_{11} + \frac{1}{3}t_{12} + 1 \\
t_9 &= \frac{1}{6}t_{10} + \frac{1}{3}t_{11} + \frac{1}{3}t_{12} + 1
\end{aligned}$$

$$\begin{aligned}
t_{10} &= \frac{1}{6}t_{10} + \frac{1}{3}t_{11} + \frac{1}{3}t_{12} + 1 \\
t_{11} &= \frac{1}{6}t_9 + \frac{1}{6}t_{10} + \frac{1}{6}t_{11} + \frac{1}{3}t_{12} + 1 \\
t_{12} &= \frac{1}{6}t_8 + \frac{1}{6}t_5 + \frac{1}{6}t_6 + \frac{1}{6}t_{11} + \frac{1}{6}t_{12} + 1
\end{aligned}$$

Forklar videre hvorfor dette er et lineært likningssystem og bruk kommandoen `rref` til å finne løsningen. Hvor mange kast må du regne med å bruke når du står på start?

b)

Løs problemet i a) når du ikke trenger eksakt antall øyne for å komme i mål.

8. Ta vare på arbeidet ditt

Vi skal nå se på noen mer datatekniske ting om hvordan du kan ta vare på det du har gjort i. Hvis du vil ta vare på kjøringene dine med tanke på senere redigering, kan du skrive

```
%logstart -r -o filnavn
```

i et IPython Shell. Innholdet i kommandovinduet blir nå fortløpende skrevet til filen du oppga, som blir opprettet i katalogen du står i. For å avslutte dagbokføringen, skriver du

```
%logstop
```

Hvis du senere ønsker å skrive til den samme dagbokfilen, gir du kommandoen

```
%logstart -r -o filnavn append
```

Logging kan i tillegg temporært startes og stoppes (uten at du må oppgi den aktive logfila på nytt). Dette gjøres med kommandoene `%logon` og `%logoff`. Du kan senere kjøre innholdet i en lagret logfil ved å skrive

```
%runlog filnavn
```

Log filer kan redigeres på vanlig måte, og egner seg derfor bra til obliger og lignende. Du kan gjøre de kjøringene du ønsker, lagre dem i loggen , og etterpå legge til kommentarer, stryke uaktuelle utregninger osv. Vær oppmerksom på at figurer ikke lagres i loggen !

Noen ganger ønsker vi ikke å ta vare på så mye av det vi har gjort, alt vi trenger er å beholde verdien på visse variable til senere kjøringer. Skriver du

```
%store x
%store y
%store A
```

lagres verdiene til `x, y` og `A`

Når du skal lagre variable, er det ofte lurt å gi dem mer beskrivende navn enn `x, y, A`. Et variabelnavn kan bestå av bokstaver og tall, men må begynne med en bokstav. Det skilles mellom store og små bokstaver. Du bør unngå å bruke innebygde funksjons- og kommandonavn på variablene dine.

Når du har arbeidet en stund, kan du fort miste oversikten over hvilke variabelnavn du har brukt. Kommandoen

```
who
```

gir deg en oversikt. Hvis du vil slette innholdet i variabelen `x`, skriver du

```
%store -d x
```

Vil du slette alle variablene, skriver du

```
%store -z
```

Oppgave til Seksjon 8

Oppgave 1

Lag en log fil og rediger den.

9. Programmering

Skal du gjøre mer avanserte ting må du lære deg å programmere. Har du programmert i et annet språk tidligere bør ikke dette være særlig problematisk, siden syntaksen sannsynligvis ligner mye på det du kan fra før. Har du aldri programmert før, står du overfor en litt større utfordring. Det finnes flere innebygde hjelpefunksjoner som kan hjelpe deg. Blant annet har de fleste Shell en autokompletteringsfunksjon for kode, slik at forslag til navn på funksjoner blir listet opp for deg hvis du ikke helt husker dem selv. Mange Shell hjelper deg også med å sørge for at du setter opp nestede parenteser riktig ved kall på funksjoner.

Et program er egentlig ikke noe annet enn en sekvens av kommandoer som du vil ha utført, men det er to ting som gjør at virkelige programmer er litt mer kompliserte enn de kommandosekvensene vi hittil har sett på. Den ene er at virkelige programmer gjerne inneholder *løkker*, dvs. sekvenser av kommandoer som vi vil at maskinen skal gjennomføre mange ganger. Vi skal se på to typer løkker: *for-løkker* og *while-løkker*. Den andre tingen som skiller virkelige programmer fra enkle kommandosekvenser, er at hvilken utregning som skal utføres på et visst punkt i programmet, kan avhenge av resultatet av tidligere beregninger som programmet har gjort. Dette fanger vi opp med såkalte *if-else-setninger*.

La oss se på et typisk problem som kan løses ved hjelp av løkker. Vi skal ta for oss *Fibonacci-tallene*, dvs. den tallfølgen $\{F_n\}$ som begynner med $F_1 = 1$, $F_2 = 1$ og tilfredsstiller

$$F_n = F_{n-1} + F_{n-2} \quad \text{for } n \geq 3 \tag{1}$$

Ethvert tall i følgen (bortsett fra de to første) er altså summen av de to foregående. Det er lett å bruke formelen til å regne ut så mange Fibonacci-tall vi vil:

$$F_3 = F_2 + F_1 = 1 + 1 = 2$$
$$F_4 = F_3 + F_2 = 2 + 1 = 3$$
$$F_5 = F_4 + F_3 = 3 + 2 = 5$$
$$F_6 = F_5 + F_4 = 5 + 3 = 8$$

og så videre. Vi ser at vi hele tiden utfører regnestykket i formel (1), men at vi for hvert nytt regnestykke oppdaterer n-verdien.

La oss nå lage et lite program som regner ut de 20 første Fibonacci-tallene. Programmet gjør dette ved å lage en 20-dimensjonal vektor der F_k er den k-te komponenten:

```
# coding=utf-8

F=[1,1] # forteller Python at F_0=1, F_1=1
for n in xrange(2,20):     # starter løkken, angir hvor langt den går
  F = F + [F[n-1]+F[n-2]]  # regner ut neste Fibonacci-tall
```

(legg merke til setningen `# coding=utf-8`, som sørger for at tegn blir tolket som UTF-8. Denne setningen er ofte droppet i kodeeksemplene i heftet. Legg også merke til indenteringen i innmaten av `for`-løkka, som er helt sentral i Python). Det burde ikke være så vanskelig å skjønne hvordan programmet fungerer: det utfører de samme beregningene som vi gjorde ovenfor fra $n = 3$ til og med $n = 20$ (dvs. mellom grensene angitt i for-løkken). Vil du nå vite hva det 17. tallet i følgen er, kan du nå skrive

```
print F[16]
```

I for-løkker bestemmer vi på forhånd hvor mange ganger løkken skal gjennomløpes. Ofte ønsker vi å fortsette beregningene til et visst resultat er oppnådd uten at vi på forhånd vet hvor mange gjennomløpninger som trengs. I disse tilfellene er det lurt å bruke en while-løkke. Det neste programmet illustrerer dette ved å regne ut Fibonacci-tallene inntil de når 10 000:

```
from numpy import *

F=[1,1]
n=2
while F[n-1]<10000:
  F = F + [F[n-1]+F[n-2]]
  n=n+1
```

Legg merke til at i en while-løkke må vi selv oppdatere fra n til $n + 1$, mens denne oppdateringen skjer automatisk i en for-løkke. Dette skyldes at while-løkker er mer fleksible enn for-løkker, og at man også ønsker å tillate andre typer oppdatering enn at n går til $n + 1$.

La oss også se på et enkelt eksempel på bruk av en if-else-setning. Det litt tossete progammet printer ut de like Fibonacci-tallene samt indeksen til alle odde Fibonacci-tall. Legg merke til operatoren `m % k` som gir oss resten når m deles på k (dersom k er lik 2 som i programmet, blir resten 0 når m er et partall, og 1 når m er et oddetall):

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

F=[1,1]
for n in xrange(2,20):     # starter for-løkke
  F = F + [F[n-1]+F[n-2]] # regner ut neste Fibonacci-tall
  if (F[n] % 2) == 0:      # innleder if-else setningen ved å
                           # sjekke om $F(n)$ er delelig med 2
    print F[n]             # skriver ut $F(n)$ hvis det er et partall
  else:                    # innleder else-delen av setningen
    print n                # skriver ut $n$ dersom $F(n)$ er et oddetall
```

Legg merke til at dobbelte likhetstegn (==) er brukt her. Det vanlige likhetstegnet brukes til å tilordne verdier: skriver du

```
C=D
```

får C den (forhåpentligvis allerede definerte) verdien til D. For å sjekke om to allerede definerte verdier er like, må du derfor bruke et annet symbol, nemlig det dobbelte likhetstegnet ==. Nyttige symboler av denne typen, er

```
<       # mindre enn
<=      # mindre enn eller lik
>       # større enn
>=      # større enn eller lik
==      # lik
!=      # ikke lik
```

Når du skriver programmer, får du også bruk for logiske operatorer som `og, eller` og `ikke`. Du skriver dem slik:

```
and     # og (& kan også brukes)
or      # eller (| kan også brukes)
not     # ikke
```

Disse operatorene returnerer de logiske verdien `true` og `false`. Som et eksempel tar vi med et lite program som skriver ut Fibonacci-tall som er partall, men ikke delelig på 4:

```
from numpy import *

F=[1,1]
for n in xrange(2,50):
  F = F + [F[n-1]+F[n-2]]
  if ((F[n] % 2)==0) and ((F[n] % 4)!=0):
    print F[n]
```

I litt større program har vi ofte behov for å avslutte en løkke og fortsette eksekveringen av den delen av programmet som kommer umiddelbart etter løkken. Til dette bruker vi kommandoen `break`. Det neste programmet illustrerer bruken. Det skriver ut alle Fibonacci-tall mindre enn 1000 (siden størrelsen på Fibonacci-tallene når 1000 før indeksen kommer til 50):

```
from numpy import *

F=[1,1]
for n in xrange(2,50):
  F = F + [F[n-1]+F[n-2]]
  if F[n]>1000:
    break
  print F[n]
```

La oss ta en nærmere titt på den generelle syntaksen for for-løkker, while-løkker, if-setninger og break-kommandoer (ikke bry deg om dette hvis du synes det ser rart og abstrakt ut på det nåværende tidspunkt: det kan likevel hende du får nytte av det når du har fått litt mer programmeringstrening). Syntaksen til en for-løkke er:

```
for n in xrange(start,stopp,steg):
  setninger
```

Hvis steglengden er 1, trenger vi bare å oppgi start- og stoppverdiene. Syntaksen til en while-løkke er:

```
while feil < toleranse:
  setninger
```

Syntaksen for if-setninger er:

```
if n < nmaks:
  setninger
```

eller

```
if n < middel:
  setninger
elif n > middel:
  setninger
else:
  setninger
```

Syntaksen for break-kommandoer er

```
for n in xrange(antall):
  setninger
  if feil > toleranse:
    break
  setninger
```

Vi avslutter denne seksjonen med noen ord om effektivitet. I programmene ovenfor har vi begynt med å la F være 2-tuppelet `[1 1]` og så utvidet lengden på tuppelet etter hvert som programmet kjører. Det viser seg at programmet går fortere dersom vi gir n-tuppelet den "riktige" størrelsen fra starten av. Det første programmet vårt blir altså raskere om vi skriver det om på denne måten:

```
from numpy import *

F=zeros(20)
F[0]=1
F[1]=1
for n in xrange(2,20):
  F[n]=F[n-1]+F[n-2]
```

Inntjeningen spiller selvfølgelig ingen rolle når programmet er så kort som her, men for store utregninger kan den ha betydning.

La oss helt til slutt nevne at det egentlig er enda mer effektivt å unngå for-løkker der det er mulig; ofte kan vi erstatte dem ved å bruke vektorer og komponentvise operasjoner isteden. Ønsker vi for eksempel å lage en vektor med de 5 første positive oddetallene som elementer, vil følgende for-løkke gjøre jobben:

```
for i in xrange(5):
  a[i]=2*i+1
```

Men det er mer effektivt å lage vektoren på følgende måte:

```
a=2*range(5)+1
```

Vi skal ikke legge vekt på effektiv programmering i dette kurset, men med tanke på senere kurs er det greit å være oppmerksom på at slik bruk av "vektor-indekseringkan gjøre store programmer mye raskere.

I sammenligningen med andre programmeringsspråk kan vi generelt si at språk som C++ vil gjennomløpe `for`-løkker mye raskere enn Python. Man skulle kanskje tro at operasjoner på matriser utføres gjennom rene implementasjoner med `for`-løkker. For eksempel, matrisemultiplikasjon kan jo implementeres rett-fram slik:

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

def mult(A,B):
  (m,n)=shape(A)
```

```
  (n1,k)=shape(B) # Må egentlig sjekke at n=n1
  C=zeros((m,k))
  for r in xrange(m):
    for s in xrange(k):
      for t in xrange(n):
        C[r,s] += A[r,t]*B[t,s]
  return C
```

Sammenligner du dette med å kjøre `A*B` i stedet vil du merke stor forskjell - ihvertfall når matrisene er store. Python regner derfor ikke alltid ut multiplikasjon av store matriser helt på denne måten.

Determinanten er en annen ting som kan regnes ut på en smartere måte enn ved å følge definisjonen direkte. Et program som bruker definisjonen av determinanten direkte kan se slik ut:

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

def detdef(A):
    """Funksjon som beregner determinanten til en matrise A
    rekursivt ved bruk av definisjonen av determinanten."""

    if A.shape[0] != A.shape[1]:
        print 'Feil: matrisen må være kvadratisk'
        return
    n = A.shape[0]

    # Determinanten av en 2x2-matrise kan regnes ut direkte
    if n == 2:
        return A[0,0]*A[1,1] - A[0,1]*A[1,0]
    # For større matriser bruker vi en kofaktorekspansjon
    else:
        determinant = 0.0
        for k in xrange(n):
            # Lag undermatrisen gitt ved å fjerne kolonne 0, rad k
            submatrix = vstack((A[0:k,1:n],A[k+1:n,1:n]))
            # Legg til ledd i kofaktorekspansjon
            determinant += (-1)**k * A[k,0] * detdef(submatrix)
        return determinant
```

Funksjonen `detdef` er rekursiv: den kaller seg selv helt til vi har en matrise som er så liten at vi kan regne ut determinanten direkte. Sørg her spesielt for at du forstår den delen av koden hvor $(n-1) \times (n-1)$-undermatrisen konstrueres. I koden ser du også at vi legger inn en sjekk på at matrisen er kvadratisk. Hvis den ikke er det skrives en feilmelding. Denne koden er ikke spesielt rask: determinanten kan regnes ut mye raskere med en innebygd metode. Determinanten til en matrise A kan du regne ut ved å skrive `linlag.det(A)` Test ut dette ved å kjøre følgende kode:

```
from detdef import *
from numpy import *
import time

A=random.rand(9,9)
e0=time.time()
linalg.det(A)
print time.time()-e0
e0=time.time()
detdef(A)
print time.time()-e0
```

Her lages en (tilfeldig generert) 9×9-matrise, og determinanten regnes ut på to forskjellige måter. I tillegg tas tiden på operasjonene: `time.time()` tar tiden, og `print` skriver den ut på displayet. Kjør koden og se hvor mye raskere den innebygde determinantfunksjonen er! Du kan godt prøve med en tilfeldig generert 10×10-matrise også, men da bør du være tålmodig mens du venter på at koden skal kjøre ferdig, spesielt hvis du sitter på en litt treg maskin.

Siden mye kode er tidkrevende å kjøre, kan det ofte være lurt å sette inn noen linjer med kode (for eksempel i `for`-løkker som kjøres mange ganger) som skriver ut på skjermen hvor langt programmet har kommet. Til dette kan du bruke funksjonen `print` , som over. Skriver du

```
print 'Nesten ferdig'
```

så vil programmet skrive den gitte teksten ut på skjermen . En litt mer avansert variant er

```
print 'Ferdig med', k, 'iterasjoner av løkka'
```

Hvis `k` er en løkkevariabel, så vil koden over til enhver tid holde styr på hvor langt vi er kommet i løkka.

Oppgaver til Seksjon 9

Oppgave 1

En følge er gitt ved $a_1 = 1$, $a_2 = 3$ og $a_{n+2} = 3a_{n+1} - 2a_n$. Skriv et program som genererer de 30 første leddene i følgen.

Oppgave 2

I denne oppgaven skal vi se på en modell for samspillet mellom rovdyr og byttedyr. Vi lar x_n og y_n betegne hhv. antall rovdyr og antall byttedyr etter n uker, og vi antar at

$$x_{n+1} = x_n(r + cy_n) \qquad\qquad y_{n+1} = y_n(q - dx_n)$$

der r er litt mindre enn 1, q er litt større enn 1, og c og d er to små, positive tall.

a)

Forklar tankegangen bak modellen

b)

Velg $r = 0.98$, $q = 1.04$, $c = 0.0002$, $d = 0.001$, $x_1 = 50$, $y_1 = 200$. Lag et program som regner ut x_n og y_n for $n \leq 1000$. Plott følgene x_n og y_n i samme koordinatsystem. Hvorfor er toppene til x_n forskjøvet i forhold til toppene til y_n?

Oppgave 3

En dyrestamme består av tre årskull. Vi regner med at 40% av dyrene i det yngste årskullet lever videre året etter, mens 70% i det nest yngste årskullet lever videre året etter. Ingen dyr lever mer enn tre år. Et individ i det andre årskullet blir i gjennomsnitt forelder til 1.5 individer som blir født året etter. Et individ i det

eldste årskullet blir i gjennomsnitt forelder til 1.4 individer som blir født året etter. La x_n, y_n, z_n være antall dyr i hvert årskull etter n år, og forklar hvorfor

$$\begin{aligned} x_{n+1} &= 1.5y_n + 1.4z_n \\ y_{n+1} &= 0.4x_n \\ z_{n+1} &= 0.7y_n \end{aligned}$$

a)

Lag et program som regner ut x_n, y_n og z_n for $1 \leq n \leq 100$. Plott alle tre kurvene i samme vindu. Lag et nytt vindu der du plotter alle de relative bestandene $x'_n = \frac{x_n}{x_n+y_n+z_n}$, $y'_n = \frac{y_n}{x_n+y_n+z_n}$, $z'_n = \frac{z_n}{x_n+y_n+z_n}$.

b)

Gjenta punkt a), men bruk andre startverdier, f.eks. $x_1 = 300$, $y_1 = 0$, z_1=0. Sammenlign med resultatene i a). Gjør oppgavene enda en gang med et nytt sett av startverdier. Ser du et mønster?

c)

La $A = \begin{pmatrix} 0 & 1.5 & 1.4 \\ 0.4 & 0 & 0 \\ 0 & 0.7 & 0 \end{pmatrix}$. Forklar at $\begin{pmatrix} x_n \\ y_n \\ z_n \end{pmatrix} = A^{n-1} \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix}$. Bruk dette til å regne ut x_{100}, y_{100} og z_{100} for starverdiene i a). Sammenlign med dine tidligere resultater.

10. py-filer

Hvis du skal skrive inn mange kommandoer som skal brukes flere ganger, kan det være praktisk å samle disse i en egen fil. Slike Python -filer skal ha endelsen .py . De inneholder Python -kommandoer og kan skrives inn i en hvilken som helst editor, f.eks. emacs .

Hvis vi samler kommandoer på en fil som heter `filnavn.py`, vil de utføres når vi skriver

```
run filnavn.py
```

i kommandovinduet. For at Python skal finne filen, må vi stå i samme katalog som filen er lagret i (med mindre filen ligger i søkestien til Python). Vi kan sjekke hvilken katalog vi står i med kommandoen `pwd` (present working directory) og skifte katalog med kommandoen `cd` (change directory).

Hvis vi f.eks ønsker å skrive kode som finner røttene til annengradslikningen $2x^2+3x+1=0$, kan vi legge følgende kommandoer på filen `annengradsligning.py`

```
from numpy import sqrt

a=2.0
b=3.0
c=1.0
x1=(-b+sqrt(b**2-4*a*c))/(2*a)
x2=(-b-sqrt(b**2-4*a*c))/(2*a)
print 'x1', x1
print 'x2', x2
```

Når vi skriver

```
run annengradsligning.py
```

utføres alle kommandoene i filen og svarene returneres:

```
x1  -0.5000
x2  -1
```

En funksjon i en py-fil deklareres med nøkkelordet 'def', og har én eller flere inn- og ut-parametre. Disse parametrene er vilkårlige objekter. Variablene som brukes i funksjoner er lokale (med mindre vi definerer statiske variable).

Hvis vi ønsker å lage en funksjon som finner røttene til en vilkårlig annengradslikning $ax^2+bx+c=0$, kan vi la brukeren spesifisere likningskoeffisientene a, b og c som inn-parametre til funksjonen, og la ut-parameteren som funksjonen returnerer, være en 2-dimensjonal radvektor som inneholder røttene `r1` og `r2`. På filen `annengrad.py` legger vi da følgende funksjonskode:

```
# coding=utf-8
from math import sqrt

def annengrad(a,b,c):
  if a != 0:      # sjekker at a ikke er null
    r1=(-b+sqrt(b**2-4*a*c))/(2*a)
    r2=(-b-sqrt(b**2-4*a*c))/(2*a)
  else:
    print('Koeffisienten til x**2 kan ikke være null')
  return (r1,r2)
```

Vi kan nå finne røttene til annengradslikningen $2x^2+3x+1=0$, ved å gi kommandoen

```
x1,x2 =annengrad(2.0,3.0,1.0)
```

Igjen returneres svarene $x_1 = -0.5000$ og $x_2 = -1$.

Oppgaver til Seksjon 10

Oppgave 1

Determinanten til en 2×2-matrise er definert ved

$$\det \begin{pmatrix} a & b \\ c & d \end{pmatrix} = ad - bc$$

Lag en py-fil (en funksjonsfil) som regner ut slike determinanter. Filen skal ha inn-parametre a, b, c, d og ut-parameter $\det \begin{pmatrix} a & b \\ c & d \end{pmatrix}$.

Oppgave 2

Skriv en py-fil (en funksjonsfil) som gir løsningen tillikningssystemet

$$\begin{aligned} ax + by &= e \\ cx + dy &= f \end{aligned}$$

når det har en entydig løsning. Filen skal virke på denne måten: Inn-parametrene er koeffisientene a, b, c, d, e, f, og ut-parametrene er løsningene x, y. Dersom likningssystemet ikke har en entydig løsning, skal programmet svare: “likningssettet har ikke entydig løsning"(det behøver altså ikke å avgjøre om likningssettet er inkonsistent eller har uendelig mange løsninger).

Oppgave 3

Lag en funksjonsfil som regner ut leddene i følgen $\{x_n\}$ gitt ved

$$x_{n+2} = ax_{n+1} + bx_n \qquad x_1 = c, x_2 = d.$$

Filen skal ha inn-parametre a, b, c, d, m og skal returnere de m første verdiene til følgen.

Oppgave 4

Lag en funksjonsfil som regner ut leddene i følgen $\{x_n\}$ gitt ved

$$x_{n+1} = ax_n(1 - x_n) \qquad x_1 = b.$$

Filen skal ha inn-parametre a, b, m, og skal returnere de m første verdiene til følgen. Sett $m = 100$, se på tilfellene $b = 0.2$ og $b = 0.8$, og kjør programmet for hhv. $a = 1.5$, $a = 2.8$, $a = 3$, $a = 3.1$, $a = 3.5$, $a = 3.9$. Plott resultatene. Eksperimenter videre hvis du har lyst, men behold humøret selv om du ikke finner noe mønster; fenomenet du ser på er et av utgangspunktene for det som kalles "kaos-teori"!

11. Anonyme funksjoner og linjefunksjoner

I forrige seksjon så vi hvordan vi kunne lagre funksjoner ved hjelp av funksjonsfiler. Trenger vi bare en enkel funksjon noen få ganger, er det imidlertid unødvendig komplisert å skrive og lagre en egen fil, og det finnes derfor to innebygde metoder for å definere funksjoner direkte i kommanduvinduet: *anonyme funksjoner* ("anonymous functions") og *linjefunksjoner* (" functions").

La oss begynne med linjefunksjoner. Hvis vi skriver

```
from math import sin
def f (x): return x*sin(1.0/x)
```

lagres funksjonen $f(x) = x \sin \frac{1}{x}$. Ønsker vi å vite hva $f(5)$ er, kan vi derfor skrive

```
>>> f(5)
0.99334665397530608
```

Ønsker vi å plotte grafen til f, går vi frem på vanlig måte:

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

x=arange(0.01,1,0.01,float)
y=f(x)
plot(x,y)
```

Metoden fungerer også på funksjoner av flere variable:

```
>>> def f (x,y): return y*sin(1.0/x)
>>> f(2,3)
1.438276615812609
```

forteller oss at funksjonsverdien til funksjonen $f(x, y) = y \sin \frac{1}{x}$ i punktet $(2, 3)$ er 1.4383. Legge merke til at i sitt svar på kommandoen

```
def f (x,y): return y*sin(1.0/x)
```

er variablene blitt ordnet slik at vi vet at x er første og y er annen variabel. Vil vi tegne grafen til f, skriver vi

```
r=arange(0.01,1,0.01,float)
s=arange(0.01,1,0.01,float)
x,y=meshgrid(x,y,sparse=False,indexing='ij')
z=f(x,y)
mesh(x,y,z)
```

La oss nå gå over til anonyme funskjoner. Disse definerer du på denne måten:

```
g= lambda x,y : x**2*y+y*3
```

Funksjonen $g(x,y) = x^2y + y^3$ er nå lastet inn. Du kan regne ut verdier og tegne grafer på samme måte som ovenfor:

```
>>> g(1,-2)
-8
>>> r=arange(0.01,1,0.01,float)
>>> s=arange(0.01,1,0.01,float)
>>> x,y=meshgrid(r,s,sparse=False,indexing='ij')
>>> z=g(x,y)
>>> mesh(x,y,z)
```

Anonyme funksjoner er nyttige når du skal integrere funksjoner. Vil du f.eks. regne ut integralet $\int_0^2 e^{-\frac{x^2}{2}}\ dx$, så kan du bruke metoden `trapezoidal` som du programmerte i INF1100 (numerisk integrasjon ved hjelp av trapesmetoden), og skrive

```
>>> trapezoidal( lambda (x) : exp(-x**2),0,2,100)
0.88207894884004279
```

Her har vi brukt trapesmetoden med 100 punkter. Utelater du `lambda (x)`, får du en feilmelding. Det går imidlertid greit å bruke denne syntaksen til å integrere en linjefunksjon du allerede har definert:

```
>>> def h (x): return exp(-x**2)
>>> trapezoidal( h,0,2,100)
0.88207894884004279
```

men kommandoen fungerer fortsatt ikke dersom h er definert som en anonym funksjon!

Vi kan bruke anonyme funksjoner i to variable sammen med funksjonen `integrate2D` som vi programmerte i INF1100 til å beregne dobbeltintegraler.

```
integrate2D(lambda x,y: x**2*y,0,1,-1,2,100,100)
```

beregner dobbeltintegralet av funksjonen x^2y over rektangelet $[0,1] \times [-1,2]$, med 100 delepunkter langs x- og y-aksen. Svaret blir 0.5, med de fire første desimalene.

Oppgaver til Seksjon 11

Oppgave 1

Definer $f(x) = \frac{\sin x}{x}$ som en linjefunksjon. Tegn grafen til f og regn ut integralet f fra $\frac{1}{2}$ til 2.

Oppgave 2

Gjenta oppgave 1, med definer nå f som en anonym funksjon.

Oppgave 3

Definer $f(x,y) = x^2y - y^2$ som en linjefunksjon og tegn grafen.

Oppgave 4

Gjenta oppgave 3, med definer nå f som en anonym funksjon.

12. Bilder og animasjoner*

Et bilde kan tolkes som en matrise, et rektangulært rutenett hvor hver rute tilsvarer et element i matrisen. En rute kalles helst *et piksel* når vi snakker om bilder. I svart-hvitt-bilder har hvert piksel én verdi, og hver verdi angir hvilken gråtone pikselet har. Ofte lagres svart-hvitt bilder med 256 ulike gråtoner, det vil si som matriser hvor elementene er verdier mellom 0 og 255, hvor 0 tilsvarer svart og 255 tilsvarer hvit. Fargebilder lagres ofte med såkalt rgb-format; hvert piksel får tre komponenter, én for rød(r), én for grønn(g), og den siste for blå(b). (Når vi kombinerer disse tre fargene på ulike måter kan vi produsere alle mulige farger). Det vil si at vi egentlig trenger 3 matriser for å lagre bildet.

Python er et utmerket verktøy til å analysere og gjøre operasjoner på bilder. De tre viktigste funksjonene er

```
A = imread('filnavn.fmt') # leser inn et bilde med angitt
    # filnavn og format, og lagrer det som en matrise A. ('fmt') kan
    # være et hvilket som helst format som Python støtter.
imshow(A) # lager et objekt fra A som kan vises frem som et bilde.
show() # viser objektet (bildet) du laget ved hjelp av imshow.
    # Fra menyene her kan du blant annet lagre bildet til fil.
```

Disse kommandoene blir tilgjengelige i Python hvis du skriver

```
from pylab import imread, imshow, show
```

Funksjonen `imread()` returnerer en MxN-array for svart-hvitt bilder, MxNx3-array for RGB-bilder (Red Green Blue) og MxNx4-array for RGBA-bilder (Red Green Blue Alpha). Enkelte ganger kan et bilde tilsynelatende være svart-hvitt, mens `imread` likevel returnerer en MxNx3-array. For å konvertere til et rent svart-hvitt bilde kan følgende kommandoer benyttes.

```
A=mean(A,2)
gray()
```

Det er viktig å være klar over at en bilde-array i utgangspunktet ikke vet hvilken orientering som er riktig". Dette er grunnen til at noen vil oppleve at bildet vises oppned, mens andre vil få bildet riktig fremstilt. Brukeren må med andre ord spesifisere eksplisitt bildets orientering. Dersom bildet vises oppned, legger man til følgende argument når man kaller funksjonen `imshow`;

```
imshow(A,origin="lower")
```

Dette sørger for at A[0,0] plasseres nederst til venstre i aksesystemet.

Når vi leser inn et bilde vil typisk verdiene lagres som heltall. Siden mange av operasjonene vi bruker antar at verdiene er av typen `double`, så må vi konvertere verdiene til denne typen. Vi må altså skrive

```
from pylab import imread
from numpy import *

img = imread('mm.gif')
img = double(img)
```

Verdiene i matrisen `img` ligger mellom 0 og 255. Størrelsen på bildet finner du ved å skrive

```
m,n=shape(img)
```

I pakken scitools.easyviz ligger funkjonen `pcolor()` som gjør oss i stand til å generere 2D-bilder. Funksjonen tar som input et grid og en matrise med lik dimensjon og fargelegger hvert punkt i henhold til tallverdien i matrisen. Kommandoen `colorbar()` gir en fargesøyle med oversikt over korrespondansen mellom tallverdier i matrisen og farger. I det følgende vises et lite eksempel.

```
x_min=...
x_max=...
y_min=...
y_max=...
A=matrix([[..],..,[..]])                        # m*n matrise
x=linspace(x_min,x_max,m)
y=linspace(y_min,y_max,n)
X,Y=meshgrid(x,y,sparse=False,indexing='ij')    # lager et m*n grid
pcolor(X,Y,A,shading='flat')                    # lager et 2D bilde
colorbar()                                      # fargesøyle
```

Dersom man inkluderer argumentet `shading='flat'` (som vi har gjort ovenfor), fjernes den svarte rammenrundt hvert fargeleggingspunkt. Det er hensiktsmessig å observere hvordan `pcolor()`-funksjonen fungerer ved å tegne bildet"av en tilfeldig matrise A i et grovt oppløst grid og i tillegg se virkningen av `shading='flat'`-argumentet.

Vi skal nå liste opp noen flere eksempler på operasjoner på bilder. Disse finnes alle i biblioteket `bildelib.py`.

Kontrastjustering

Som et eksempel på en operasjon vi kan gjøre på et bilde, la oss se på kontrastjustering. Pikselverdiene justeres da ved hjelp av en funksjon som avbilder $[0,1]$ på $[0,1]$, en såkalt kontrastjusterende funksjon. Eksempler på kontrastjusterende funksjoner er

$$\begin{aligned} f(x) &= \frac{\log(x+\varepsilon)-\log(\varepsilon)}{\log(1+\varepsilon)-\log(\varepsilon)}, \\ g(x) &= \frac{\arctan(n(x-1/2))}{2\arctan(n/2)}+1/2. \end{aligned}$$

For å bruke en kontrastjusterende funksjon må vi i tillegg avbilde pikselverdiene fra [0,255] til [0,1], og til slutt avbilde de fra [0,1] til [0,255]. Kode for kontrastjustering med $f(x)$ blir da seende slik ut:

```
def contrastadjust(img):
  epsilon = 0.001               # Prov også 0.1, 0.01, 0.001
  maks = img.max()              # Finner maksverdi i img
  newimg = double(img)/maks     # Avbilder pikselverdiene på [0,1]
  newimg = (log(newimg+epsilon) - log(epsilon))/\
           (log(1+epsilon)-log(epsilon));
  newimg = newimg*maks # Avbilder verdiene tilbake til [0,maks]
  return newimg
```

Et bilde, sammen med de kontrastjusterte versjoner med $f(x)$ og $g(x)$ ser du i Figur 1.

(a) Originalt

(b) Kontrastjustert med $f(x)$

(c) Kontrastjustert med $g(x)$

Figur 1: Et bilde med og uten kontrastjustering.

Utglatting

En annen vanlig operasjon på bilder er utglatting. Ved utglatting erstattes verdiene i bildet med vektede middelverdier av de nærliggende pikslene i bildet. Vektingen er definert ved en matrise, kalt en vektmatrise (eller konvolusjonskjerne), der "midterste"verdi i matrisen bestemmer vekten for det aktuelle pikselet, verdien rett over denne bestemmer vekten for pikselet rett over det aktuelle pikselet, o.s.v. Summen av alle elementene i vektmatrisen er vanligvis 1. Utglatting med en vektmatrise kan programmeres slik:

```
def smooth(img,vektmatrise):
  m,n=shape(img)            # Forutsetter at img er en mxn-array
  newimg = zeros((m,n))
  k,k1 = shape(vektmatrise) # Vi skal ha k==k1, odde
  sc = (k+1)/2
  for m1 in xrange(m):
    for n1 in xrange(n):
      slidingwdw = zeros((k,k))
        # slidingwdw er den delen av bildet som
        # vektmatrisen blir anvendt på for piksel (m1,n1)
      slidingwdw[ max(sc-1-m1,0):(min(sc+m-2-m1,k-1)+1) ,
                  max(sc-1-n1,0):(min(sc+n-2-n1,k-1)+1)] = \
      img[max(0,m1-(sc-1)):(min(m-1,m1+(sc-1))+1) ,
          max(0,n1-(sc-1)):(min(n-1,n1+(sc-1))+1)]
      newimg[m1,n1] = sum(vektmatrise * slidingwdw)
  return newimg
```

Inne i `for`-løkken ser du kode som kan virke litt kryptisk. Koden skyldes at i ytterkantene av bildet er det ikke nok piksler til å matche størrelsen på vektmatrisen, slik at litt ekstra kode må til for å håndtere dette. Utglatting med matrisen

$$A_1 = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

kan vi nå gjøre slik

```
from numpy import *
from smooth import *

vektmatrise=(1.0/16)*array([[1.0,2.0,1.0],[2.0,4.0,2.0],[1.0,2.0,1.0]])
newimg = smooth(img,vektmatrise)
```

I Figur 2 viser vi bildet utglattet med vektmatrisen A_1, samt vektmatrisen

$$A_2 = \frac{1}{1024}\begin{bmatrix} 1 & 6 & 15 & 20 & 15 & 6 & 1 \\ 6 & 36 & 90 & 120 & 90 & 36 & 6 \\ 15 & 90 & 225 & 300 & 225 & 90 & 15 \\ 20 & 120 & 300 & 400 & 300 & 120 & 20 \\ 15 & 90 & 225 & 300 & 225 & 90 & 15 \\ 6 & 36 & 90 & 120 & 90 & 36 & 6 \\ 1 & 6 & 15 & 20 & 15 & 6 & 1 \end{bmatrix}$$

Som du ser gir den større vektmatrisen A_2 en større utglattingseffekt.

Gradienten til et bilde

Ved å erstatte vektmatrisen med andre matriser kan vi regne ut andre interessante bilder, for eksempel som viser hvordan bildet forandrer seg i x-retningen, eller i

(a) Utglatting med A_1 (b) Utglatting med A_2

Figur 2: Utglatting av bildet vårt med to forskjellige vektmatriser.

y-retningen. Mer presist ville vi bruke

$$\texttt{vektmatrisex} = \begin{bmatrix} 0 & 0 & 0 \\ -1/2 & 0 & 1/2 \\ 0 & 0 & 0 \end{bmatrix}$$

$$\texttt{vektmatrisey} = \begin{bmatrix} 0 & 1/2 & 0 \\ 0 & 0 & 0 \\ 0 & -1/2 & 0 \end{bmatrix}.$$

Disse svarer til approksimasjoner av $\frac{\partial}{\partial x}$ og $\frac{\partial}{\partial y}$, respektive, anvendt på bildet. Gradienten til et bilde kan vi regne ut slik:

```
from numpy import *
from smooth import *

newimg = sqrt( smooth(img,vektmatrisex)**2 + \
               smooth(img,vektmatrisey)**2 )
```

Med denne koden kan det hende at de nye pikselverdiene faktisk ligger utenfor $[0, 255]$. Dette kan vi løse ved å avbilde verdiene til $[0, 1]$ ved hjelp av følgende kode:

```
def mapto01(img):
  minval = img.min()
  maxval = img.max()
  newimg = double(img - minval)/(maxval-minval)
  return newimg
```

Deretter kan vi avbilde disse verdiene tilbake på $[0, 1]$. I Figur 3 har vi vist hvordan de partielt deriverte bildene og gradientbildet da blir seende ut.

Eksempler på andre operasjoner på bilder

Det er også mange andre operasjoner enn kontrastjustering og utglatting som kan gjøres enkelt. Her er noen flere eksempler:

(a) x-partielt deriverte av bildet

(b) y-partielt deriverte av bildet

(c) Gradienten til bildet

Figur 3: De partielt deriverte og gradienten av bildet

```
def speilingoppned(img):
  m,n=shape(img)
  imgupdown = zeros((m,n))
  for k in xrange(m):
    imgupdown[k,:] = img[m-1-k,:]
  return imgupdown
```

```
def speilingvenstrehoyre(img):
  m,n=shape(img)
  imgleftright = zeros((m,n))
  for k in xrange(n):
    imgleftright[:,k] = img[:,n-1-k]
  return imgleftright
```

```
def bildenedskalering(img):
  m,n=shape(img)
  r=..
  scimg=zeros((m/2**r,n/2**r))
  for m1 in xrange(m/2**r):
    for n1 in xrange(n/2**r):
      xstart = (m1-1)*(2**r)+1
      ystart = (n1-1)*(2**r)+1
      # Nedskalering skjer ved at blokker i bildet blir
      # erstattet med et piksel med verdi lik middelverdien
      # i blokken:
      scimg[m1,n1] = sum(img[xstart:(xstart+2**r),
                             ystart:(ystart+2**r)])/(2**(2*r))
  return scimg
```

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 fil ved hjelp av funksjonen `hardcopy`. Hvis plottene blir lagret i filene `tmp0.eps`, `tmp1.eps`, o.s.v. kan vi etterpå lime sammen disse til en animert `gif`-fil ved hjelp av kommandoen

```
movie('tmp*.eps',encoder='convert',fps=m,output_file='movie.gif')
```

hvor `m` angir antall figurer som skal vises per sekund. Den animerte gif-fila `movie.gif` kan til slutt åpnes i andre programmer, slik som i Microsoft Internet explorer, eller ved hjelp av programmet `animate`, som finnes på Linux-maskiner.

Oppvarmingsøvelse 1: Matrisemultiplikasjon

Vi skal ta utgangspunkt i den komponentvise definisjonen av matrisemultiplikasjon, og ser på hvordan multiplikasjonen kan beskrives på alternative måter ved hjelp av lineærkombinasjoner av radene eller søylene i matrisene.

Produktet av en matrise og en søylevektor

Lag en matrise og en søylevektor

```
C=matrix([[1,5,2],[9,6,8],[8,7,4]])
x=matrix([[2],[5],[8]])
```

Siden søylevektoren $\mathbf{x}$ er en 3×1-matrise, kan vi bruke den komponentvise definisjonen av matrisemultiplikasjon og regne ut matriseproduktet ved å skrive

```
C*x
```

Bruk deretter søylene i C som vektorer, og beregn lineærkombinasjonen av disse hvor vi bruker elementene i søylevektoren $\mathbf{x}$ som koeffisienter:

```
c1=C[:,0]
c2=C[:,1]
c3=C[:,2]
print x[0,0]*c1+x[1,0]*c2+x[2,0]*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=matrix([[2,4,6],[1,3,5],[7,8,9]])
B=matrix([[1,4],[2,5],[3,6]])
```

og regn ut matriseproduktet AB. Bruk deretter søylene i B som vektorer

```
b1=B[:,0]
b2=B[:,1]
```

og regn 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.

Sjekk at resultatet stemmer for matrisene A og B som vi definerte før oppgave 2.

Produktet av en radvektor og en matrise

Lag en matrise og en radvektor

```
R=matrix([[1,3,8],[4,6,2],[5,7,9]])
y=matrix([[3,-4,6]])
```

Siden radvektoren y er en 1×3-matrise, kan vi bruke den komponentvise definisjonen av matrisemultiplikasjon og regne ut matriseproduktet slik:

```
y*R
```

Bruk deretter radene i R som vektorer, og beregn lineærkombinasjonen av disse hvor vi bruker elementene i vektoren y som koeffisienter:

```
r1=R[0,:]
r2=R[1,:]
r3=R[2,:]
print y[0,0]*r1+y[0,1]*r2+y[0,2]*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=matrix([[1,2,3],[4,5,6],[7,8,9]])
E=matrix([[2,4],[1,3],[7,8]])
```

og regn ut matriseproduktet DE. Bruk deretter radene i D som vektorer

```
d1=D[0,:]
d2=D[1,:]
d3=D[2,:]
```

og regn 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.

Sjekk at resultatet stemmer for matrisene D og E som vi definerte før oppgave 5.

Oppvarmingsøvelse 2: Egenskaper ved determinanter

Vi skal se på 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.

Multiplisere en rad med en skalar

Lag en 4×4-matrise, og beregn determinanten

```
A=matrix([[2,4,6,8],[1,3,5,7],[9,8,7,6],[5,4,3,2]])
linalg.det(A)
```

Vi skal nå se hva som skjer med determinanten hvis vi erstatter første rad i A med 5 ganger seg selv:

```
A[0,:]=5*A[0,:]
linalg.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 generere tilfeldige 4×4-matriser med heltallige komponenter mellom -10 og 10 slik:

```
A=20*random.rand(4,4)-10
A=A.round()
```

funksjonen `rand` genererer her 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` runder av til nærmeste heltall.

Oppgave 1

Generer en tilfeldig 4×4-matrise som over, beregn determinanten, og sammenlign denne med determinanten til matrisen som fremkom ved å erstatte første rad med 5 ganger seg selv. Gjenta eksperimentet et par ganger.

a)

Hva skjedde med determinanten til matrisene når du multipliserte første rad med 5? Gjelder dette også dersom du multipliserer en annen rad med 5? Hva skjer med determinanten hvis du isteden multipliserer raden med 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 5 ved kommandoen

```
A[:,2]=5*A[:,2]
```

og deretter beregne determinanten.

Bytte om to rader

La oss først generere en tilfeldig 4×4-matrise A med heltallige komponenter mellom 10 og -10 som over, og beregn determinanten. Vi skal undersøke hva som skjer med determinanten hvis vi bytter om rad 2 og rad 4 i matrisen A

```
A=A[[0,3,2,1],:] # Her vektorindekserer vi radene i den rekkefølgen
                 # vi vil ha dem (rad 2 og 4 har byttet plass)
print linalg.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.

b)

Hva tror du resultatet blir hvis du isteden bytter om to søyler i A? Du kan for eksempel bytte om søyle 2 og 3 med å skrive

```
A=A[:,[0,2,1,3]]
```

og beregne determinanten.

Addere et skalarmultiplum av en rad til en annen rad

Generer en tilfeldig 4×4-matrise med heltallige komponenter mellom 10 og -10, og beregn determinanten. Vi skal nå se hva som skjer med determinanten hvis vi adderer 3 ganger andre rad til første rad:

```
A[0,:]=A[0,:]+3*A[1,:]
```

Hva skjedde med determinanten? Vi 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[:,1]=A[:,1]+3*A[:,3]
```

og beregne determinanten

Øvre triangulære matriser

Vi kan generere en tilfeldig øvre triangulær 5×5-matrise med heltallige komponenter ved å skrive

```
U=10*random.rand(5,5)+1
U=triu(U.round())  # U står for Upper diagonal matrix
```

Vi kan så beregne determinanten. Deretter kan vi ta produktet av alle elementene på hoveddiagonalen til matrisen U ved hjelp av en for-løkke:

```
diag=U[0,0]
for i in range(1,5):
  diag=diag*U[i,i]
```

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 kan generere en tilfeldig nedre triangulær 5×5-matrise med heltallige komponenter ved å skrive

```
L=10*random.rand(5,5)+1
L=triu(L.round())            # L står for Lower diagonal matrix
```

Vi kan så beregne determinanten. Deretter kan vi ta produktet av alle elementene på hoveddiagonalen til matrisen L ved hjelp av en for-løkke:

```
diag=L[0,0]
for i in range(1,5):
  diag=diag*L[i,i]
```

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. Vi kan generere en 5×5-identitetsmatrise ved kommandoen

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

Vi kan deretter beregne determinanten.

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 en 1-er i hver søyle. Den fremkommer ved en ombytting (permutasjon) av radene i identitetsmatrisen. Når vi skal lage en 5×5 permutasjonsmatrise 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[[0,2,1,3],:]
```

Vi kan også generere en tilfeldig 5×5-permutasjonsmatrise P ved å skrive

```
P=I[random.permutation(5),:]
```

og deretter regne ut determinanten.

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 til i begynnelsen av denne laben kan brukes til å definere hva determinanten til en kvadratisk $n \times n$-matrise A 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 en faktor $a_{i\sigma_i}$ fra hver rad og hver søyle. Det betyr at søyleindeksene $\sigma_1, \sigma_2, ..., \sigma_n$ er en permutasjon (omstokking) av linjeindeksene $1, 2, ..., n$. Hvert elementært produkt kan utstyres med et fortegn $sign(\sigma) = (-1)^{inv(\sigma)}$ som er bestemt av antall inversjoner $inv(\sigma)$ 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

$$\det A = \sum_{\sigma} sign(\sigma) \prod_i a_{i\sigma_i}.$$

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)

Lab 1: Lineæravbildninger og matriser

Vi skal se på sammenhengen mellom lineære transformasjoner og matriser. Vi bruker plottefunksjonen 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.

Opptegning av enkle figurer

For å tegne opp et kvadrat kan vi angi koordinatene til de fire hjørnepunktene i den rekkefølgen vi vil forbinde dem, og tegne inn rette streker fra ett hjørnepunkt til det neste. For å tegne opp kvadratet med hjørner i punktene $(0,0)$, $(0,4)$, $(4,4)$, og $(4,0)$, lager vi først en vektor x som består av førstekomponentene til de fire hjørnepunktene og en vektor y som består av andrekomponentene:

```
x=[0,0,4,4,0]
y=[0,4,4,0,0]
plot(x,y)
axis([-10,10,-10,10])
axis('square')
```

Legg merke til at vi har lagt til det første punktet til slutt i x- og y-vektorene også: for å få tegnet opp et fullstendig kvadrat, må vi avslutte med samme hjørnepunkt som vi startet i.

Når vi har mange plottepunkter (la oss si n stykker) er det mer oversiktlig å kunne angi plottepunktene i en $2\times n$-matrise hvor første rad består av x-koordinatene til punktene og andre rad består av y-koordinatene. Hver søyle i matrisen representerer da et plottepunkt. Kvadratet ovenfor blir nå representert med (2×5)-matrisen

```
K= matrix([[0,0,4,4,0],[0,4,4,0,0]])
```

For å plotte kvadratet skriver vi nå

```
x=asarray(K)[0,:]  # x-vektor skal være første rad i matrisen K
y=asarray(K)[1,:]  # 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 kommandoene hver gang lager vi en funksjon `tegn`, som kan kalles med en hvilken som helst $2 \times n$-matrise `X` som inn-parameter, og som tegner den tilsvarende figuren:

```
def 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 = asarray(X)[0,:]
  y = asarray(X)[1,:]
  plot(x, y, 'b-',axis=[-10,10,-10,10]) # trekker blå linje mellom punktene
```

Etter å ha definert matrisen `K` som inneholder plottepunktene, kan vi nå få tegnet opp kvadratet ved å skrive

```
tegn(K)
```

Istedenfor å definere matrisen `K` direkte i kommandovinduet, kan vi også lage en liten funksjon `kvadrat` som returnerer matrisen som inneholder plottepunktene for kvadratet:

```
def kvadrat():
  # Returnerer en matrise X som inneholder data for tegning av et kvadrat.
  X= matrix([[0,0,4,4,0],[0,4,4,0,0]])
  return X
```

Vi kan nå bruke funksjonen `kvadrat` som inn-parameter til funksjonen `tegn`, og tegne kvadratet ved å skrive

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

Det kan kanskje virke unødvendig omstendelig å definere 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 tegne opp et bilde av de transformerte punktene kan vi altså skrive

```
A=matrix([[-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 tegnefunksjon som også markerer to fritt valgte plottepunkter i forskjellige farger. Når vi allerede har laget en funksjon `tegn` er det lett å føye til det lille som trengs for å gjøre dette, og lagre koden som en ny funksjon `tegnmedpunkter`:

```
def 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 = asarray(X)[0,:]
  y = asarray(X)[1,:]
  plot(x,y,'b-')
  hold('on')
  plot([x[m]],[y[m]],'or',axis=[-10,10,-10,10])
  plot([x[n]],[y[n]],'og',axis=[-10,10,-10,10])
  hold('off')
```

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. Ved å sammenligne med de opprinnelige plottepunktene i matrisen `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

Skriv funksjonen `kvadrat` og `tegn` som beskrevet ovenfor. Kjør koden

```
tegn(A*kvadrat)
```

for å studere kva 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. Vi lager følgende funksjon for opptegning av et hus (tegn inn punktene på et papir først for å være sikker på at du forstår hvordan opptegningen foregår):

```
def 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=matrix([[-6,-6,-7,0,7,6,6,-3,-3,0,0,-6],\
            [-7,2,1,8,1,2,-7,-7,-2,-2,-7,-7]])
  return X
```

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

Vi lager følgende funksjon for opptegning av en seilbåt (igjen bør du starte med å tegne inn punktene på et papir så du skjønner hvordan figuren fremkommer).

```
def 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=matrix([[0,7,4,-4,-7,0,0,2,0,5,0],\
            [-4,-4,-7,-7,-4,-4,8,6,6,-2,-2]])
  return X
```

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 opp ned). 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 funksjon `monogramMH()` som returnerer en matrise med plottepunktene for å tegne opp monogrammet MH:

```
def 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=matrix([[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=matrix([[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=hstack((M,H)) # Skjøter sammen de to matrisene med M etterfulgt av H
  return X
  # 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.
```

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 sjekke resultatet for følgende skjær-matrise:

```
A=matrix([[1,0.2],[0,1]])
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=matrix([[0.8,0],[0,1]])
tegn(B*monogramMH())
```

De to foregående typene matriser har vi sett tidligere (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, og deretter lage denne smalere:

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

Vi får samme resultat om vi først danner produktmatrisen BA, og anvender denne på plottepunktene til monogrammet:

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

Sammensetning av avbildninger svarer altså 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 skrive

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

d)

Lag tre korte funksjoner `speilomx()`, `speilomy()`, og `roter180()`, 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 skrive ting som

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

Vi skal illustrere generelle rotasjonsmatriser i form av bevegelser til timeviseren på en klokke, og starter derfor med å lage funkjoen `klokkeskive` for å tegne opp en klokkeskive med timemarkører:

```
def klokkeskive():
  h=0.3
  text(5*cos(pi/2)-h,5*sin(pi/2),'12')
  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')
  t=arange(0,2*pi,0.05,float)
  x=6*cos(t)
  y=6*sin(t)
  plot(x,y,axis=[-10,10,-10,10]) # tegner sirkelskive med radius 6
  title('Klokke')
  axis('equal')
```

Det neste vi trenger er en timeviser. På samme måte som i de foregående seksjonene kan vi 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). Vi lager derfor en funksjon `klokkepil` som returnerer matrisen med plottepunktene for timeviseren:

```
def 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 = matrix([[-0.2,-0.2,-0.4,0,0.4,0.2,0.2,-0.2],\
              [0,3,3,3.8,3,3,0,0]])
  return X
```

Vi kan nå tegne en klokke der timeviseren peker på 12, ved å skrive

```
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 $-\frac{\pi}{2}$ radianer). For å finne rotasjonsmatrisen til en dreining på $-\frac{\pi}{2}$ radianer, ser vi hva basisvektorene $e_1 = (1, 0)$ og $e_2 = (0, 1)$ avbildes på: bildene av disse blir søylene i rotasjonsmatrisen. Slik finner vi raskt ut at rotasjonsmatrisen for vinkelen $-\frac{\pi}{2}$ er

```
A=matrix([[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

1. matrisen A dreier viseren en time frem
2. matrisen B dreier viseren to timer frem

Her skal du selv beregne rotasjonsmatrisene til de aktuelle vinklene ved hjelp av metoden beskrevet ovenfor. 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`. 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 laben med å bruke funksjonene vi allerede har laget til å simulere en tikkende klokke. Dette gir en enkel illustrasjon av hvordan vi kan bruke for-løkker. 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, men eksemplet illustrerer hvordan vi kan bruke for-løkker til å generere bilder i bevegelse:

```
def seilendebaat():
  # Viser en båt som seiler av sted fra høyre billedkant mot venstre.
  # Henter inn plottepunktene til båten fra matrisen generert av
  # funkjonen baat().
  # Tegner opp alt i et aksesystem med skala [-10 10] i begge retninger
  # ved hjelp av funksjonen tegn
  B=baat() # leser inn plottepunktene fra baat.py
  X=1.0/4*vstack((B[0,:]+32, B[1,:]))
  # krymper båten og plasserer den i høyre billedkant
  tegn(X) # tegner det første bildet av båten
  for t in range(160):
    X=vstack((X[0,:]-0.1*t, X[1,:])) # flytter båten skrittvis mot venstre
    tegn(X)                      # tegner båten i den nye posisjonen
    sleep(1)
    # hardcopy('tmpsb%03d.eps' % (t)) #
  # Lager animasjon:
  # movie('tmpsb*.eps',encoder='convert',fps=20,output_file='moviesb.gif')
```

"filmsnutten" kan nå kjøres ved å kalle funksjonen `seilendebaat()`

Oppgave 6

Bruk fremgangsmåten ovenfor til å skrive en funksjon `tikkendeklokke` som tegner opp en en klokke som går en runde med timeviseren. Du kan f.eks la klokka gå med en fart på 5 sekunder per sekund ved å skrive

```
from time import sleep
sleep(1)
```

som venter ett sekund før neste bilde tegnes.

For å lage en animasjon av den seilende båten i det forrige eksemplet, så kan vi bare fjerne kommentartegnet foran de to linjene som kaller kaller funksjonene vi lærte om i Seksjon om animasjoner, i funksjonen `seilendebaat` over.

Oppgave 7 (ekstraoppgave)

Lag en animasjon av den tikkende klokken.

Lab 2: Gauss-eliminasjon

Vi skal se på hvordan vi kan lage funksjoner som utfører Gauss-eliminasjon på matriser, dvs. som bringer dem på trappeform ved hjelp av elementære radoperasjoner. 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 funksjonen `rref`).

Gauss-eliminasjon (enkel versjon uten ombytting av rader)

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 noe forskjellig fra 0 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 0 i posisjonen under pivot-elementet, så vi skriver

```
k=float(A[1,0])/A[0,0]
A[1,:]=A[1,:]-k*A[0,:]
```

Deretter sørger vi for å få 0 i tredje rad ved å trekke `A(3,1)/A(1,1)=3/6=1/2` av første rad fra tredje rad:

```
k=float(A[2,0])/A[0,0]
A[2,:]=A[2,:]-k*A[0,:]
```

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=float(A[2,1])/A[1,1]
A[2,:]=A[2,:]-k*A[1,:]
```

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
```

Det er viktig å være klar over at kommandoen

```
A=matrix([[3,5,-4],[-3,-2,4],[6,1,-8]])
```

definerer en heltalls-matrise og man er derfor utsatt for feilberegninger i form av heltallsdivisjon når man manipulerer rader og søyler i matrisen. Dersom du ønsker å beregne den reduserte trappeformen til en matrise ved hjelp av funksjonen rref() i `MAT1120lib`, trenger du ikke å bekymre deg for dette. I oppgavene nedenfor skal du imidlertid programmere funksjoner som tar inn en matrise og manipulerer denne, og det er da svært viktig at du tenker over hva slags type matrise du jobber med. Den enkleste løsningen er å konvertere matrisen til en double-matrise (uavhengig om den allerede er en slik matrise) som det første du gjør i funksjonen. Dette kan gjøres ved bruk av kommandoen:

```
A=matrix(A,double)
```

Oppgave 1

Lag en funksjon `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 nestet inni hverandre). I hvert trinn skal du altså selv regne ut de riktige multiplumene av pivotraden som skal trekkes fra radene nedenfor.

Oppgave 2

Bruk ideen fra funksjonen i forrige oppgave til å lage en 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 pivotelementene 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=shape(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 i posisjon `(1,1)` i første søyle. Vi leter da nedover i første søyle etter et ikke-trivielt element, og finner dette i rad 2. Vi kan derfor skaffe oss et ikke-trivielt pivot-element ved å bytte om første og andre rad:

```
A[[0,1],:]=A[[1,0],:]
```

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 0 i posisjonen rett under pivot-elementet, og trenger derfor ikke å gjøre noe med rad 2. For å få 0 i tredje rad, trekker vi `float(A[2,0])/A[0,0]=1/5` ganger første rad fra tredje rad:

```
k=float(A[2,0])/A[0,0]
A[2,:]=A[2,:]-k*A[0,:]
```

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 0, leter vi nedover i søylen etter et ikke-trivielt element å flytte opp. Men alle elementene nedenfor er også 0, 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=float(A[2,2])/A[1,2]
A[2,:]=A[2,:]-k*A[1,:]
```

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 0, 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` 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 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 funksjonen ovenfor. 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 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 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 0 eller ikke) kalles funksjonen *Gauss-eliminasjon med delvis pivotering*. Denne funksjonen er enkel å implementere ved hjelp av en ferdig innebygd funksjon `max` som finner det største elementet i en søylevektor:

```
maksverdi = max(y)
p=argmax(y)
```

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

Gauss-eliminasjon 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 at vi i visse situasjoner må avbryte en løkke ved hjelp av kommandoen `break`. Vår programskisse er som følger:

```
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>
  if <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>
  <øk radindeksen med 1 for å lete etter neste pivot i raden nedenfor.>
  <avbryt løkken dersom radindeksen blir større enn antall rader>
  # (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)
```

Oppgave 3

Lag en 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. Du kan følge programmskissen ovenfor, men kan gjerne bruke en while-løkke isteden.

Gauss-Jordan-eliminasjon

Hittil har vi bare sett på funksjoner som bringer en matrise over på trappeform ved hjelp av Gauss-eliminasjon. I dette avsnittet skal vi se på hvordan vi kan utvide disse til en funksjon som bringer matrisen på redusert trappeform. 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 skrive 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: Ligningssystemer og lineær uavhengighet

Vi skal lage en funksjon 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. I tillegg til å gi deg en test på om du har fått med deg sammenhengen mellom disse begrepene, vil lab-øvelsen gi deg trening i å skrive funksjoner som er litt mer omfattende enn de du tidligere har laget.

Oppgave 1

Skriv en 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 ikke lov til å bruke den innebygde funksjonen `rank` selv om du skulle kjenne til begrepet rang). Du kan bruke funksjonen `rref` til å bringe en matrise på redusert trappeform.

Oppgave 2

Skriv en 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 matriselikningen $A\mathbf{x} = \mathbf{b}$ er konsistent eller inkonsistent. (Hvis systemet er konsistent, trenger ikke denne funksjonen å avgjøre hvor mange løsninger systemet har).

Oppgave 3

Utvid 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 matriselikningen $A\mathbf{x} = \mathbf{b}$ 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:

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

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

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

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: $(A, \mathbf{a})$, $(B, \mathbf{b})$, $(C, \mathbf{b})$, $(D, \mathbf{d})$, $(E, \mathbf{d})$. Dersom funksjonen svarer at et likningssystem er konsistent, skal du bruke funksjonen `antlosn` til å avgjøre hvor mange løsninger systemet har.

Lab 4: Newtons metode for flere variable

Vi lager en funksjon som utfører Newtons metode på funksjoner av flere variable, for å finne tilnærmede løsninger til systemer av ikke-lineære ligninger. Vi ser også på hva som skjer hvis vi bruker Newtons metode på funksjoner av en kompleks variabel. 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. Hvis du føler at du har god kontroll på Newtons metode for funksjoner av en variabel, 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

Newtons metoden i en variabel lager stadig bedre tilnærminger x_n til nullpunkter til en funksjon f ved hjelp av formelen

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

La oss først se et enkelt eksempel på hvordan vi kan utnytte denne formelen for å få 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 f 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 regne ut de 5 første tilnærmingsverdiene fra Newtons metode med startverdi $x_0 = 1$, kan vi derfor bruke følgende for-løkke:

```
x=1.0
for t in range(5):
  x=x-(x**2-2)/(2*x) # Beregner neste tilnærmingsverdi
  print x
```

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

```
x**2-2
```

Oppgave 1

a)

Bruk en for-løkke 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 regne ut funksjonsverdien i dette punktet.

b)

Hvor mange iterasjoner må gjennomføres før de fire første desimalene i tilnærmings verdien ikke endrer seg? Hvor mange iterasjoner må gjennomføres før de tolv første desimalene ikke endrer seg?

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 0). 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 iterasjonene av Newtons metode (med startverdi $x_0 = 1$) utføres inntil $f(x_n)$ er mindre enn en toleranse 0.0001, men likevel slik at den ikke utfører mer enn maksimalt 20 iterasjoner:

```
epsilon=0.0001 # angir den ønskede toleransen
n_max=20       # maksimalt antall iterasjoner som skal utføres
n=1            # startverdi
x=1.0          # startverdi
while abs(f(x)) > 0.0001 and n<=n_max:
  x=x-(x**2-2)/(2*x)
  n += 1
```

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 lurt å lage en funksjon `newton` hvor vi kan spesifisere startverdien som innparameter. For å kunne benytte funksjonen på forskjellige funksjoner uten å måtte 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 egne funksjoner `f` og `df`. Disse kan for eksempel være anonyme funksjoner, slik at vi trenger bare sende med nye slike som inn-parametre til metoden `newton` når vi bytter funksjon. Funksjonen vår får følgende signatur:

```
def newton(x,f,df,epsilon=0.0000001,N=30):
  # Funksjon som utfører Newtons metode
  # x: startpunkt for iterasjonen
  # f: funksjon som regner ut funksjonsverdiene
  # df: funksjon som regner ut verdiene til den deriverte
  # Utparameter x: den beregnede tilnærmingsverdien til nullpunktet
  <Programkode: Denne skal du lage i oppgave 2>
```

Ønsker vi f.eks å bruke funksjonen `newton` på funksjonen $f(x) = x^3 + 2x^2 - 2$ (fra oppgave 1), kan vi sende de to anonyme funksjonene

```
lambda x : x**3+2*x**2-2
lambda x : 3*x**2+4*x
```

som inn-parametre til `newton` slik:

```
x=newton(2,lambda x : x**3+2*x**2-2,lambda x : 3*x**2+4*x)
```

Oppgave 2

a)

Lag en funksjon `newton` som leser inn verdiene til en funksjon og dens deriverte fra to anonyme funksjoner, og beregner tilnærmingsverdier for nullpunktet ved hjelp av disse verdiene. Som innparametre skal funksjonen ta et startpunkt start og de to funksjonene `f` og `df` Prosedyren 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 funksjonen returnere den sist beregnede tilnærmingsverdien som utparameter.

Tips: Dersom iterasjonsindeksen din heter `t`, kan du legge til følgende linje i iterasjonsløkka for å få en pen utskrift på skjermen av `t`, den tilhørende tilnærmingsverdien `x` som blir beregnet for nullpunktet i hvert trinn, og tilhørende funksjonsverdi `f(x)` i dette punktet:

```
print 'itnr=', t, ', x=', x, ', f(x)=', f(x)
```

b)

Test funksjonen du lagde i a) 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 funksjonen du lagde i 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 $h(x) = x^{1/3}$. 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.

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

```
y=sign(x)*abs(x)**(1.0/3)
```

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 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:

```
x=arange(-2,2,0.01,float)
y=arange(-2,2,0.01,float)
X,Y = meshgrid(x,y,sparse=False,indexing='ij')
m,n=shape(X)  # tar vare på størrelsen for senere bruk i løkke
```

Deretter vil vi finne numeriske tilnærmingsverdier for nullpunktene til funksjonen vi skal studere (da blir det enkelt å sjekke hvilket nullpunkt tilnærmings verdiene konvergerer mot). Til dette kan vi bruke den innebygde funksjonen `roots`, som returnerer en vektor med nullpunktene til et polynom med gitte koeffisienter. Funksjonen $f(z) = z^3 + az + b$ får koeffisientene representert ved radvektoren $[1, 0, a, b]$ (husk å ta med koeffisienten 0 foran det "usynligeleddet 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
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 i in range(m):
  for k in range(n): # gjennomløper punktene i gitteret
    z=complex(X[i,k],Y[i,k])  # omgjør punkt i gitter til komplekst tall
    # Så kjøres iterasjoner av Newtons metode med punktet som startverdi
    for t in range(maxit):
      z=z-(z**3+a*z+b)/(3*z**2+a)
      if abs(z-rts[0])<tol:
        T[i,k]=1
        break
      if abs(z-rts[1])<tol:
        T[i,k]=44
        break
      if abs(z-rts[2])<tol:
        T[i,k]=60
        break
```

Nå kan vi få fargelagt punktene i det aktuelle området i planet med fargeverdien bestemt i matrisen T ved å skrive

```
pcolor(x,y,T,shading='flat') # 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 en funksjon `kompleks` 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 funksjonen 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 som er utført, f.eks på følgende måte:

```
if abs(z-rts[0])<tol:
  T[j,k]=1+t*(23.0/maxit)
  break
if abs(z-rts[1])<tol:
  T[j,k]=48-t*(23.0/maxit)
  break
if abs(z-rts[2])<tol:
  T[j,k]=64-t*(23.0/maxit)
  break
```

Merknad: Ikke bli 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 koden 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 koden til en 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 funksjonen fra forrige oppgave slik at du fritt kan velge sentrum og forstørrelse. Bruk funksjonen 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 funksjonen 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 lab 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 flere variable

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

$$\begin{aligned} f(x,y) &= 0 \\ g(x,y) &= 0 \end{aligned}$$

Definerer vi den vektorvaluerte funksjonen $\mathbf{F}(x,y) = (f(x,y), g(x,y))$ har vi lært at Newtons metode i to variable tar formen

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

der (x_n, y_n) er tilnærminger til løsningen til likningene våre, og der $\mathbf{F}'(x,y)$ er Jacobimatrisen til $\mathbf{F}$ i punktet (x,y). Vi kan implementere dette på to måter:

```
x=x-linalg.inv(J(x))*f(x)    # Metode 1
x=x-linalg.solve(J(x),f(x))  # Metode 2
```

hvor `J(x)` er Jacobi-matrisen.

Oppgave 6

a)

Lag en funksjon `newtonfler` som leser inn verdiene til en vektorvaluert funksjon $\mathbf{F}$ og dens Jacobimatrise J, og bruker disse verdiene til å til å beregne tilnærmingsverdier til nullpunktet ved hjelp av Newtons metode. Som innparametre skal funksjonen ta et startpunkt `x` (angitt som en søylevektor) og funksjoner `F` og `J` som returnerer funksjonsverdienene og verdiene til Jacobimatrisen. Både `F` og `J` skal være anonyme funksjoner som tar en vektor som input, og returnerer en vektor/matrise som output. Prosedyren skal deretter utføre Newtons metode inntil normen (bruk funsksjonen `norm`) til $F(x_n)$ er mindre enn en toleranse 0.0000001, eller den har utført 30 iterasjoner. Funksjonen skal til slutt returnere den sist beregnede tilnærmingsverdien.

b)

I resten av denne oppgaven skal vi studere ligningssystemet

$$\begin{aligned} x^2 + y^2 - 48 &= 0 \\ x + y + 6 &= 0 \end{aligned}$$

Lag 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.

c)

Bruk figuren du lagde i punkt b) til å finne fornuftige startverdier for Newtons metode, og kjør deretter 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 funksjonen `newtonfler` i forrige oppgave, burde den også fungere for systemer av mer enn to variable. Du kan undersøke dette ved å legge inn vektorvaluerte funksjoner med flere komponenter i de anonyme funksjonene som sendes som innparametre til funksjonen. Vil du for eksempel løse et ligningssystem på formen

$$\begin{aligned} f(x,y,z) &= 0 \\ g(x,y,z) &= 0 \\ h(x,y,z) &= 0 \end{aligned}$$

definerer du bare anonyme funksjoner

```
F = matrix([[f],[g],[h]])
J = matrix([[dfx,dfy,dfz],[dgx,dgy,dgz],[dhx,dhy,dhz]])
```

som representerer funksjonen og dens Jacobimatrise, og sender disse som parametre til `newtonfler`. Prøv ut dette for å finne en tilnærmet løsning på ligningssystemet

$$\begin{aligned} y^2 + z^2 - 3 &= 0 \\ x^2 + z^2 - 2 &= 0 \\ x^2 - z &= 0 \end{aligned}$$

ved hjelp av Newtons metode.

Oppvarmingsøvelse 1: Løsningsforslag

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 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.

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 i i A med søyle k i B. Det betyr at i-te komponent i k-te søyle i AB er en lineær kombinasjon av elementene i i-te rad fra A med komponentene i 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.

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 sjekke at dette resultatet stemmer for matrisene A og B fra oppgave 2 ved å lage tre søylevektorer

```
a1=A[:,0]
a2=A[:,1]
a3=A[:,2]
```

og sjekke at første og andre søyle i AB er lik lineærkombinasjonene

```
b1[0]*a1+b1[1]*a2+b1[2]*a3
b2[0]*a1+b2[1]*a2+b2[2]*a3
```

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

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.

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 DE 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 i i D med søyle 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.

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 sjekke at dette resultatet stemmer for matrisene D og E fra oppgave 5 ved å lage tre radvektorer:

```
e1=A[0,:]
e2=A[1,:]
e3=A[2,:]
```

og sjekke at første, andre, og tredje rad i DE er lik lineærkombinasjonene

```
d1[0]*e1+d1[1]*e2+d1[2]*e3
d2[0]*e1+d2[1]*e2+d2[2]*e3
d3[0]*e1+d3[1]*e2+d3[2]*e3
```

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

Oppvarmingsøvelse 2: Løsningsforslag

Oppgave 1

a)

Determinanten til den nye matrisen blir 5 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.

Oppgave 2

a)

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

b)

Determinanten skifter fortegn.

Oppgave 3

a)

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

b)

Determinanten forblir uforandret.

Oppgave 4

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

Oppgave 5

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

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.

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.

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 $sign(\sigma) = (-1)^{inv(\sigma)}$ skifter til motsatt verdi.

Ved ombytting av linje 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 rader 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 i, det vil si at hvis rad i kan skrives som en sum av to vektorer $\mathbf{v}_1$ og $\mathbf{v}_2$, så blir determinanten lik summen av determinantene til de matrisene vi får ved å putte inn henholdsvis $\mathbf{v}_1$ og $\mathbf{v}_2$ som rad 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 0.

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 0, 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 0, er produktet av elementene på hoveddiagonalen (alle andre elementærprodukter må inneholde et element som ligger under hoveddiagonalen (forklar!) og blir derfor 0).

e)

Vi skal forklare hvorfor determinanten til en nedre triangulær matrise er lik produk tet av elementene på hoveddiagonalen. Dette kommer av at det eneste elementær-produktet som ikke blir 0, er produktet av elementene på hoveddiagonalen (alle andre elementærprodukter må inneholde et element som ligger over hoveddiagona-len og blir derfor 0).

Lab 1: Løsningsforslag

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.

Oppgave 2

Koden som løser a), b), og c) blir som følger:

```
A=matrix([[0.5,0],[0,0.5]]) # a) Skrumper figuren i begge retninger
A=matrix([[0.5,0],[0,1]])   # b) Skrumper bare i x-retningen
A=matrix([[1,0],[0,0.5]])   # c) Skrumper bare i y-retningen
```

Oppgave 3

Koden som løser a) og b) blir som følger:

```
A= matrix([[-1,0],[0,1]]) # a) x-koordinaten må skifte fortegn
A= matrix([[1,0],[0,-1]]) # b) y-koordinaten må skifte fortegn
```

Oppgave 4

Koden som løser a), b), og c) blir som følger:

```
A= matrix([[-1,0],[0,1]])  # a) x-koordinaten må skifte fortegn
A= matrix([[1,0],[0,-1]])  # b) y-koordinaten må skifte fortegn
A= matrix([[-1,0],[0,-1]]) # c) Roter pi radianer (180 grader)
```

d)

Koden for `speilomx()`, `speilomy()`, og `roter180()` blir som følger:

```
def speilomx():
  # Returnerer matrise for speiling om x-aksen.
  A=matrix([[1,0],[0,-1]])
  return A

def speilomy():
  # Returnerer matrise for speiling om y-aksen.
  A=matrix([[-1,0],[0,1]])
  return A

def roter180():
  # Returnerer rotasjonsmatrise for dreining 180 grader.
  A=matrix([[-1,0],[0,-1]])
  return A
```

e)

```
monogramHM = speilomy() * monogramMH()   # Speiling av MH om y-aksen gir HM
tegn(monogramHM)
monogramWH = roter180() * monogramHM     # Rotasjon av HM 180 grader gir WH
tegn(monogramWH)
monogramHW = speilomy() * monogramWH     # Speiling av WH om y-aksen gir HW
tegn(monogramHW)
monogramMH = roter180() * monogramHW   # Rotasjon av HW 180 grader gir MH
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 de to kommandoene

```
speilomx()
speilomy()*roter180()
```

er like, siden 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. Sjekke at matrisene `speilomy()*speilomx()` og `roter180()` er like.

Oppgave 5

a)

Koden som løser i) og ii) blir som følger:

```
# i) dreier pila 30 grader med klokka
A = matrix([[cos(-pi/6),-sin(-pi/6)],[sin(-pi/6),cos(-pi/6)]])
# ii) dreier pila 60 grader med klokka
B = matrix([[cos(-pi/3),-sin(-pi/3)],[sin(-pi/3),cos(-pi/3)]])
```

b)

A 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 for å dreie viseren to timer frem.

c)

Funksjonen `roter()` kan skrives slik:

```
def roter(v):
  # Denne funksjonen tar som input en vinkel v
  # og returnerer matrisen som roterer figuren en vinkel v.
  X = matrix([[cos(v),-sin(v)],[sin(v),cos(v)]])
  return X
```

Vi tester denne ved å skrive

```
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 til å dreie klokkeviseren t timer frem ved å skrive

```
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 vokser eksponentielt med t, mens det å beregne $\sin t$ og $\cos t$ ikke blir noe mer komplisert når t vokser.

Oppgave 6

`tikkendeklokke` kan skrives slik:

```
def tikkendeklokke():
  # Viser en timeklokke som går en runde (1 time per sekund).
  # Henter inn en ferdig opptegnet klokkeskive generert av funksjonen
  # klokkeskive
  # Plottepunktene for en timeviser som peker rett opp (kl 12) er
  # definert i en matrise som genereres av scriptet klokkepil.m
  # Denne timeviseren roteres vha en rotasjonsmatrise og tegnes opp
  # i samme figur som klokkeskiven ved hjelp av funksjonen tegn
  for i in range(1,14):
    klokkeskive() # script som tegner opp klokkeskiven
    hold('on')
    tegn(roter((i-1)*(-pi/6))*klokkepil()) # tegner den roterte viseren
    hold('off')
    sleep(1)
    # hardcopy('tmptk%03d.eps' % (i))
  # Lager animasjon:
  # movie('tmptk*.eps',encoder='convert',fps=1,output_file='movietk.gif')
```

Oppgave 7 (ekstraoppgave)

Vi kan lage en klokkeanimasjon ved å fjerne kommentarene foran to av linjene i funksjonen `tikkendeklokke` over.

Lab 2: Løsningsforslag

Oppgave 1

`gaussA` kan skrives slik:

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

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

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

Oppgave 2

`gauss` kan skrives slik:

```
def 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=shape(A)
  A=matrix(A,double) #Sørger for at matrisen er en double-matrise
  # 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 in range(n): # gjennomgår en og en søyle
    for i in range(j+1,m):       # gjennomgår en og en rad nedenfor pivoten
      k = float(A[i,j])/A[j,j]          # finner riktig multiplum å trekke fra
      A[i,:] = A[i,:] - k*A[j,:] # nuller ut elementene under pivoten"
  return A
```

Oppgave 3

`gauss_delpiv` kan skrives slik:

```
def gauss_delpiv(A):
  m,n=shape(A) # antall rader m og antall søyler n i matrisen A
  A=matrix(A,double) # sørger for at A er en double-matrise
  i=0 # Vi starter i første rad og setter derfor radindeksen 'i' lik 0

  # Vi skal nå gjennomgå en og en søyle:
  for j in range(n):
    # For hver søyle j finner vi elementet med størst absoluttverdi,
    # fra rad i->m, samt raden p til dette elementet.
    maks=max(abs(A[i:m,j]))
    p=argmax(abs(A[i:m,j]))
    p=p+i
    # Så flytter vi rad nr p med det maksimale elementet opp ved å bytte om
    # rad p og rad i (den øverste raden vi holder på med i dette trinnet)
    A[[i,p],:] = A[[p,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 in range(i+1,m):
        k = float(A[r,j])/A[i,j]
        A[r,:] = A[r,:] -k*A[i,:]
      # 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-1):
      break
    # 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.
  return A
```

Bemerkning: Generelt er det 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+1),j] = float(A[(i+1):(m+1),j])/A[i,j]
A[(i+1):(m+1),j:(n+1)] = A[(i+1):(m+1),j:(n+1)] - k[(i+1):(m+1),j]*A[i,j:(n+1)]
```

Oppgave 4

`gauss_jordan` kan skrives slik:

```
def gauss_jordan(A):
  # 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=shape(A)
  A=matrix(A,double) # sørger for at A er en double-matrise
  i=0                # startverdier
  j=0
  while i<m and j<n:
    # Finner den største (absolutte) verdien under pivotposisjonen
    # (inkl pivotelementet) og radposisjonen til dette elementet ved
    # først å sette variablen maks lik pivotelementet og deretter lete
    # gjennom resten av søylen (ved bruk av en for-løkke) etter større
    # absolutte verdier.
    maks=max(abs(A[i:m,j]))
    p=argmax(abs(A[i:m,j]))
    p=p+i
    A[[i,p],:] = A[[p,i],:]          # bytter rad i og p
    if A[i,j] != 0:                  # dersom A[i,j] er ikkenull,
      for r in range(m):             # går gjennom alle radene
        if r!=i:
          k = float(A[r,j])/A[i,j]
          A[r,:] = A[r,:] - k*A[i,:] # nuller ut elementet i søyle j
        else:
            A[r,:]=A[r,:]/float(A[r,j])# ledende 1-er i pivotrad
      i=i+1
      j=j+1
    else:                                        # her er A[i,j] lik 0
      j=j+1
  return A
```

Lab 3: Hint og løsningsforslag

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 kode:

```
# coding=utf-8
from MAT1120lib import rref

def linavh(A):
  # Funksjon som tar en matrise som inn-parameter og sjekker om
  # søylene i matrisen er lineært avhengige eller uavhengige.
  m,n=shape(A)
  if m<n:
    <gi brukeren beskjed om søylene er lineært avhengige eller uavhengige.>
  else:
    U=rref(A)
    uavh=true
    j=0
    while uavh and j<n:
      if <betingelse oppfylt>:
        print 'Søylevektorene er lineært avhengige'
        uavh=false
      j++
    if uavh:
      print 'Søylevektorene er lineært uavhengige'
```

Hint til oppgave 2

Her trenger du bare å erstatte innmaten i parantesen <betingelser oppfylt> med riktig kode:

```
# coding=utf-8
from MAT1120lib import rref

def inkons(A,b):
  # 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=shape(A)
  if length(b)==m:
    print 'Vektoren du tastet inn har feil dimensjon'
  else:
    Aug=[A,b] # utvidet matrise med dimensjon m*(n+1)
    U=rref(Aug)
    konsistent=true
    i=0
    while konsistent and i<m:
```

```
      if <betingelser oppfylt>:
        print 'Ligningssystemet er inkonsistent'
        konsistent=false
      i++
    if konsistent:
      print 'Ligningssystemet er konsistent'
```

Hint til oppgave 3

Her trenger du bare å erstatte inn maten i de fire parantesene <betingelse oppfylt>, <betingelser oppfylt>, <gi brukeren beskjed om antall løsninger> og <gi variab elen entydig riktig sannhetsverdi>, med riktig kode:

```
# coding=utf-8
from MAT1120lib import rref

def antlosn(A,b):
  # 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=shape(A)
  if length(b)==m:
    print '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=0
    konsistent=true
    while konsistent and i<m:
      if <betingelser oppfylt>:
        print 'Ligningssystemet er inkonsistent'
        konsistent=false
      i++
    if konsistent:
      print 'Ligningssystemet er konsistent'
      entydig=true
      if m<n:
        <gi brukeren beskjed om antall løsninger>
        <gi variabelen entydig riktig sannhetsverdi>
      else:
        j=0
        while entydig and (j<n): # trenger ingen radindeks siden n<=m
          if <betingelse oppfylt>:
            print 'Systemet har uendelig mange løsninger'
            entydig=false
          j++
        if entydig:
          print 'Ligningssystemet har nøyaktig en løsning'
```

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 0 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):

```
def linavh(A):
  # 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=shape(A)
  if m<n:
    print 'Søylevektorene er lineært avhengige pga flere søyler enn rader'
  else:
    U=rref(A)
    print U
    uavh=True
    j=0
    while uavh and j<n:
      if U[j,j]==0: # Sjekker om det fins et null-element på diagonalen
        print 'Søylevektorene er lineært avhengige'
        uavh=False
      j=j+1
    if uavh:
      print 'Søylevektorene er lineært uavhengige'
```

Oppgave 2

Et ligningssystem av typen $A\mathbf{x} = \mathbf{b}$ er konsistent hvis og bare hvis den reduserte trappeformen til den utvidete matrisen $[A \quad \mathbf{b}]$ ikke har en rad på formen $[00\cdots0\mathbf{c}]$ hvor $\mathbf{c}$ er forskjellig fra $\mathbf{0}$:

```
def inkons(A,b):
  # 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=shape(A)
  if shape(b)[0]!=m:
    print 'Vektoren du tastet inn har feil dimensjon'
  else:
    Aug=hstack((A,b)) # utvidet matrise med dimensjon m*(n+1)
    U=rref(Aug)
    print U
    konsistent=True
    i=0
    while konsistent and i<m:
      if (U[i,:]==zeros((1,n))) and (U[i,n]!=0):
        print 'Ligningssystemet er inkonsistent'
        konsistent=False
      i=i+1
    if konsistent:
      print 'Ligningssystemet er konsistent'
```

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:

```
def antlosn(A,b):
  # 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=shape(A)
  if shape(b)[0]!=m:
    print 'Vektoren du oppga har feil dimensjon i forhold til matrisen'
  else:
    Aug=hstack((A,b)) # utvidet matrise med dimensjon m*(n+1)
    U=rref(Aug)
    print U
    # Sjekker om ligningssystemet er inkonsistent ved å lete etter
    # en rad på formen [0 0 ... 0 c] hvor c er forskjellig fra null.
    i=0
    konsistent=True
    while konsistent and i<m:
      if (U[i,:]==zeros((1,n))) and (U[i,n]!=0):
        print 'Ligningssystemet er inkonsistent'
        konsistent=False
      i=i+1
    if konsistent:
      print 'Ligningssystemet er konsistent'
      entydig=True
      if m<n: # det må finnes pivot-frie søyler
        print 'Systemet har uendelig mange løsninger'
        entydig=False
      else:
        j=0
        while entydig and (j<n): # trenger ingen radindeks siden n<=m
          if U[j,j]==0: # det må finnes pivot-frie søyler
            print 'Systemet har uendelig mange løsninger'
            entydig=False
          j=j+1
        if entydig:
          print 'Systemet har nøyaktig en løsning'
```

Oppgave 4

a)

Resultatet av programkjøringen bør være:

```
[[ 1.  0.  0.]
 [ 0.  0.  1.]
 [ 0.  0.  0.]]
Søylevektorene er lineært avhengige
[[ 1.  0.  0.  0.]
 [ 0.  1.  0.  2.]
 [ 0.  0.  1.  0.]
 [ 0.  0.  0.  0.]]
Søylevektorene er lineært avhengige
Søylevektorene er lineært avhengige pga flere søyler enn rader
[[ 1.  0.  0.  0.]
 [ 0.  1.  0.  0.]
 [ 0.  0.  1.  0.]
 [ 0.  0.  0.  1.]
 [ 0.  0.  0.  0.]]
Søylevektorene er lineært uavhengige
Søylevektorene er lineært avhengige pga flere søyler enn rader
```

b)

Resultatet av programkjøringene bør være:

```
[[ 1.  0.  0.  0.]
 [ 0.  0.  1.  0.]
 [ 0.  0.  0.  1.]]
Ligningssystemet er konsistent
[[ 1.  0.  0.  0.  0.]
 [ 0.  1.  0.  2.  0.]
 [ 0.  0.  1.  0.  0.]
 [ 0.  0.  0.  0.  1.]]
Ligningssystemet er konsistent
[[ 1.    0.    0.    0.    0.    0.96]
 [ 0.    1.    0.    2.    0.   -1.48]
 [ 0.    0.    1.    0.    0.    1.68]
 [ 0.    0.    0.    0.    1.   -2.04]]
Ligningssystemet er konsistent
[[ 1.    0.    0.    0.    0.    0.96]
 [ 0.    1.    0.    2.    0.   -1.48]
 [ 0.    0.    1.    0.    0.    1.68]
 [ 0.    0.    0.    0.    1.   -2.04]]
Ligningssystemet er konsistent
Systemet har uendelig mange løsninger
[[ 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
[[ 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
[[ 1.          0.          0.          0.          0.         -1.03077221
  -1.25682214]
 [ 0.          1.          0.          0.          0.         -0.59918715
  -0.88755564]
 [ 0.          0.          1.          0.          0.          1.66460228
   1.26998258]
 [ 0.          0.          0.          1.          0.          0.33210761
   1.94155216]
 [ 0.          0.          0.          0.          1.          0.28662667
  -0.01664409]]
Ligningssystemet er konsistent
[[ 1.          0.          0.          0.          0.         -1.03077221
  -1.25682214]
 [ 0.          1.          0.          0.          0.         -0.59918715
  -0.88755564]
 [ 0.          0.          1.          0.          0.          1.66460228
   1.26998258]
 [ 0.          0.          0.          1.          0.          0.33210761
   1.94155216]
 [ 0.          0.          0.          0.          1.          0.28662667
  -0.01664409]]
Ligningssystemet er konsistent
Systemet har uendelig mange løsninger
```

Lab 4: Løsningsforslag

Oppgave 1

a)

Kjører vi for-løkken

```
x=1.0
for t in range(5):
  xny=x-(x**3+2*x**2-2)/(3*x**2+4*x);
  x=xny
  print x
```

får vi svaret

```
0.857142857143
0.839544513458
0.839286810068
0.839286755214
0.839286755214
```

Det ser ut som om tilnærmingene konvergerer mot et punkt. For å teste om funksjonsverdien i dette punktet er nær 0, gir vi kommandoen

```
>>> print x**3+2*x**2-2
-2.22044604925e-016
```

Funksjonsverdien i punktet er altså svært nær 0.

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.

Oppgave 2

a)

Koden for funksjonen `newton` blir slik:

```
def newton(x,f,df,epsilon=0.0000001,N=30):
  # Funksjon som utfører Newtons metode
  # x: startpunkt for iterasjonen
  # f: funksjon som regner ut funksjonsverdiene
  # df: funksjon som regner ut verdiene til den deriverte
  n=0
  while abs(f(x)) > epsilon and n<=N:
    x=x-float(f(x))/df(x)
    print 'itnr=', n, 'x=', x, 'f(x)=', f(x)
    n += 1
  return x
```

b)

For å kunne bruke funksjonen `newton` på funksjonen $f(x) = x^3 + 2x^2 - 2$, skriver vi to anonyme funksjoner hvor vi spesifiserer funksjonen og dens deriverte $f'(x) = 3x^2 + 4x$:

```
lambda x: x**3+2*x**2-2
lambda x: 3*x**2+4*x
```

Vi kjører funksjonen `newton` på funksjonen f med startpunkt $x_0 = 1$, og får (hvis du har brukt den anbefalte kommandoen `fprintf` for å få pen utskrift på skjermen)

```
>>> x=newton(1.0,lambda x: x**3+2*x**2-2,lambda x: 3*x**2+4*x)
>>> print x
itnr= 1 x= 0.857142857143 f(x)= 0.0991253644315
itnr= 2 x= 0.839544513458 f(x)= 0.00141032896517
itnr= 3 x= 0.839286810068 f(x)= 3.00070024828e-007
itnr= 4 x= 0.839286755214 f(x)= 1.37667655054e-014
0.839286755214
```

c)

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

```
x=arange(-8,6,0.1,float)
y=x**3+2.*x**2-30*x-5
plot(x,y)
```

For å kunne bruke funksjonen `newton` på $f(x) = x^3+2x^2-30x-5$ og dens deriverte $f'(x) = 3x^2 + 4x - 30$ med startpunkt $x_0 = 1$, skriver vi

```
>>> x=newton(1.0,lambda x: x**3+2*x**2-30*x-5,lambda x: 3*x**2+4*x-30)
>>> print x
itnr= 1 x= -0.391304347826 f(x)= 6.98545245336
itnr= 2 x= -0.166734044099 f(x)= 0.0529865592132
itnr= 3 x= -0.165001525003 f(x)= 4.50702727317e-006
itnr= 4 x= -0.16500137761 f(x)= 3.37507799486e-014
itnr= 5 x= -0.16500137761 f(x)= -8.881784197e-016
-0.16500137761
```

Vi ser at funksjonsverdien i tilnærmingspunktet er svært nær 0, 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 skrive:

```
x=arange(-3,3,0.1,float)
y=x**3-5*x
plot(x,y)
```

For å kunne bruke funksjonen `newton` på funksjonen $g(x) = x^3-5x$ og dens deriverte $g'(x) = 3x^2 - 5$ med startpunkt $x_0 = 1$, skriver vi

```
>>> x=newton(1.0,lambda x: x**3-5*x,lambda x: 3*x**2-5)
>>> print x
```

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). Arsaken 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 $x = 2$, 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 $h(x) = x^{1/3}$ over intervallet $[-4, 4]$ ved å skrive:

```
x=arange(-4,4,0.05,float)
y=sign(x)*abs(x)**(1.0/3)
plot(x,y)
```

For å bruke funksjonen `newton` på funksjonen $h(x) = x^{1/3}$ og dens deriverte $h'(x) = \frac{1}{3}x^{-2/3}$ med startpunkt $x_0 = 1$, skriver vi

```
>>> x=newton(1,lambda x : sign(x)*abs(x)**(1.0/3),lambda x : (1/3)*abs(x)**(-2.0/3))
```

Her vil tilnærmingsverdiene divergere lenger og lenger bort fra nullpunktet til funksjonen. Arsaken 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 gjorde. Tangenten i det nye tilnærmingspunktet x er enda slakere, og fører oss derfor enda lenger bort fra origo for å finne neste tilnærmingsverdi x_2 osv.

Oppgave 3

a)

Funksjonen `kompleks` kan skrives slik:

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

# Script 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=arange(-2,2,0.01,float)
y=arange(-2,2,0.01,float)
X,Y=meshgrid(x,y,sparse=False,indexing='ij')
m,n=shape(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*z+b:
p=[1,0,a,b]
# Vi finner 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 i in range(m):
  for k in range(n):
    z=complex(X[i,k],Y[i,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 in range(maxit):
      z=z-(z**3+a*z+b)/(3*z**2+a)
      if abs(z-rts[0])<tol:
        T[i,k]=1
        break
      if abs(z-rts[1])<tol:
        T[i,k]=44
        break
      if abs(z-rts[2])<tol:
        T[i,k]=60
        break
# fargelegger punktet (x(j),y(k)) i planet med fargen
# angitt av verdien av matrise-elementet T[j,k]
pcolor(x,y,T,shading='flat')
axis('xy')
colorbar() # tegner opp søyle med fargeverdiene ved siden av plottet
```

b)

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

Oppgave 4

a)

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

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

def kompleks(a,b):
  # Prosedyre 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 prosedyren.
  # 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=arange(-2,2,0.01,float)
  y=arange(-2,2,0.01,float)
```

```
X,Y=meshgrid(x,y,sparse=False,indexing='ij')
m,n=shape(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:
a=0
b=-1
p=[1,0,a,b]
# Vi finner 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 i in range(m):
  for k in range(n):
    z=complex(X[i,k],Y[i,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 in range(maxit):
      z=z-(z**3+a*z+b)/(3*z**2+a)
      if abs(z-rts[0])<tol:
        T[i,k]=1+t*(23.0/maxit)
        break
      if abs(z-rts[1])<tol:
        T[i,k]=48-t*(23.0/maxit)
        break
      if abs(z-rts[2])<tol:
        T[i,k]=64-t*(23.0/maxit)
        break
# fargelegger punktet (x(j),y(k)) i planet med fargen
# angitt av verdien av matrise-elementet T[j,k]
pcolor(x,y,T,shading='flat')
axis('xy')
colorbar() # tegner opp søyle med fargeverdiene ved siden av plottet
```

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 $b = 2$.

Oppgave 5

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

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

def komplekszoom(sentrumx,sentrumy,zoom):
  # Prosedyre 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=2.0/zoom
  x=linspace(sentrumx-lengde,sentrumx+lengde,400)
  y=linspace(sentrumy-lengde,sentrumy+lengde,400)
  X,Y=meshgrid(x,y,sparse=False,indexing='ij')
  m,n=shape(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 finner 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 i in range(m):
    for k in range(n):
      z=complex(X[i,k],Y[i,k])
      for t in range(maxit):
        z=z-(z**3+a*z+b)/(3*z**2+a)
        if abs(z-rts[0])<tol:
          T[i,k]=1+t*(23.0/maxit)
          break
        if abs(z-rts[1])<tol:
          T[i,k]=48-t*(23.0/maxit)
          break
        if abs(z-rts[2])<tol:
          T[i,k]=64-t*(23.0/maxit)
          break
  # fargelegger punktet (x(j),y(k)) i planet med fargen
  # angitt av verdien av matrise-elementet T(j,k)
  pcolor(x,y,T,shading='flat')
  # setter aksesystemet i vanlig xy-orientering, ellers ville
  # verdiene på y-aksen går i motsatt retning av vanlig
  axis('xy')
  colorbar() #tegner opp søyle med fargeverdiene ved siden av plottet
```

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.

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 in range(1,makszoom+1):
  komplekszoom(-0.8,0,faktor**t)
  hardcopy('tmpzoom%03d.eps' % (t)) # plott nummer t lagres til fil
# spiller animasjonen en gang med 2 bilder pr sekund:
movie('tmpzoom*.eps',encoder='convert',fps=2,output_file='moviezoom.gif')
```

Oppgave 6

a)

Vi kan skrive funksjonen `newtonfler` slik

```
def newtonfler(x,F,J,epsilon=0.0000001,N=30):
  # Funksjon som utfører Newtons metode på en
  # vektorvaluert funksjon
  # x: søylevektor som inneholder startpunktet for iterasjonen
  # F: funksjon som regner ut funksjonsverdiene
  # J: Funksjon som regner ut Jacobimatrisen
  n=0
  while linalg.norm(F(x)) > epsilon and n<=N:
    x=x-linalg.solve(J(x),F(x))
    print 'itnr=', n, 'x=', x, 'F(x)=', F(x)
    n += 1
  return x
```

b)

Vi tegner opp nivåkurvene $f(x, y) = 0$ og $g(x, y) = 0$ i samme figurvindu (og skriver plottet til filen `kontur.eps`) med følgende kode:

```
r=arange(-10,10,0.1,float)
s=arange(-10,10,0.1,float)
x,y = meshgrid(r,s,sparse=False,indexing='ij')
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('square')
hold('off')
hardcopy('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. Vi utfører Newtons metode på vår vektorvaluerte funksjon og på dens Jacobimatrise med startpunkt $(-7, 1)$, ved hjelp av koden

```
>>> x=newtonfler(array([-7,1]),\
                 lambda x : array([x[0]**2+x[1]**2-48,x[0]+x[1]+6]),\
                 lambda x : matrix([[2*x[0],2*x[1]],[1,1]]))
>>> print x
[-6.87298335  0.87298335]
```

Denne verdien er nær å være et nullpunkt, som utskriften fra for-løkken viser oss. For å finne en tilnærmingsverdi til det andre nullpunktet, bruker vi startpunktet $(1, -7)$ og skriver

```
>>> x=newtonfler(array([1,-7]),\
                 lambda x : array([x[0]**2+x[1]**2-48,x[0]+x[1]+6]),\
                 lambda x : matrix([[2*x[0],2*x[1]],[1,1]]))
>>> print x
[ 0.87298335 -6.87298335]
```

Denne verdien er også nær å være et nullpunkt

Oppgave 7

For å kjøre funksjonen `newtonfler` på dette ligningssystemet med startpunkt $x = (1, -7, 5)$, skriver vi

```
>>> x=newtonfler(array([1,-7,5]),\
                 lambda x : array([x[1]**2+x[2]**2-3,x[0]**2+x[2]**2-2,x[0]**2-x[2]]),\
                 lambda x : matrix([[0,2*x[1],2*x[2]],[2*x[0],0,2*x[2]],[2*x[0],0,-1]]))
>>> print x
[ 1.         -1.41421356  1.        ]
```

Også her er verdien som regnes ut veldig nær å være et nullpunkt

Register