Programmering i realfagene

Faglig opplegg: programmering i fysikk

kodeskolen simula

kodeskolen@simula.no

Programmering er et utrolig nyttig verktøy som brukes av fysikere i stort omfang, fordi det gjør mange problemer som er svært kompliserte, eller umulige å løse med klassisk matematikk, ganske overkommelige med noen linjer kode.

I dette prosjektet skal vi bruke det dere har lært om programmering i python i løpet av disse ukene og anvende det til å løse et sammensatt problem i fysikk med en sprettball som spretter og er påvirket av luftmotstand.

Innhold

1 Eulers metode

Før vi kan sette i gang med prosjektet vårt, må vi første introdusere en ny måte å tenke på de klassiske bevegelsesligningene.

1.1 Vanlige begrensninger

Alle som har undervist eller blitt undervist fysikk på videregående kjenner garantert til ting som beveger seg langs en *friksjonsfri* flate, med *konstant fart*, eller at vi *ser bort i fra luftmotstand*. Dette er antakelser vi veldig ofte må gjøre fordi dersom vi lar være, blir problemene ofte for kompliserte til å løse. Bevegelsesligningene som blir tatt i bruk i videregående skole tar utgangspunkt i en *konstant akselerasjon*, noe som sjeldent er tilfellet dersom vi ser på bevegelser som inkluderer f. eks luftmotstand. For dette prosjektet trenger vi to bevegelsesligninger:

$$s = s_0 + v_0 t + \frac{1}{2} a t^2 \tag{1}$$

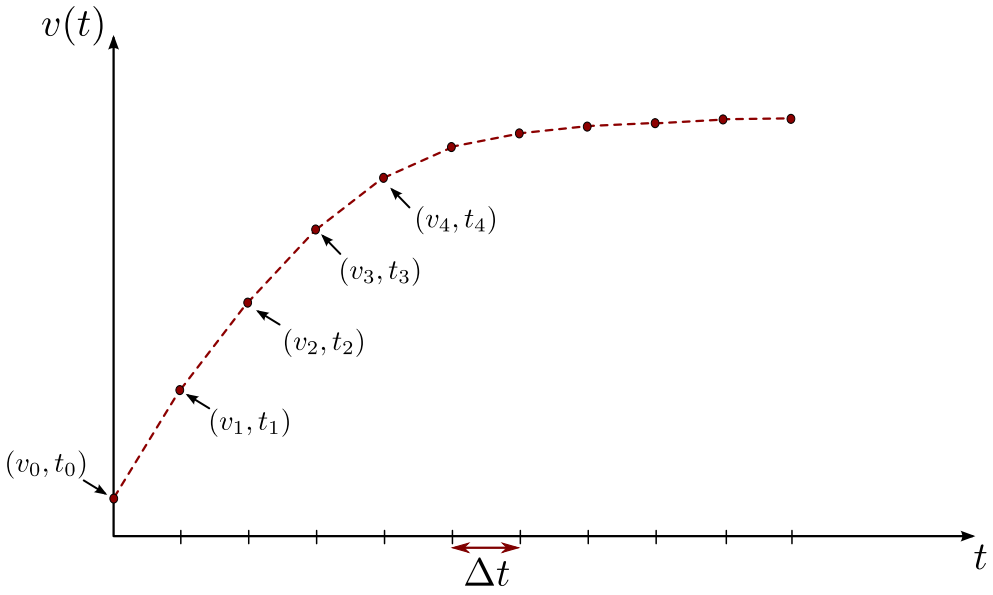
$$v = v_0 + at \tag{2}$$

Det s er strekning/posisjon og v er hastigheten etter en gitt tid t med utgangshastighet og posisjon v_0 og s_0, og en konstant akselerasjon a.

1.2 Diskretisering av bevegelsesligningene

Et veldig enkelt, men veldig kraftfullt verktøy, er å dele en bevegelse opp i bittesmå steg. For en bevegelse som har varierende akselerasjon over tid, kan vi likevel, dersom vi deler bevegelsen opp i små nok tidsintervaller Δt, anta at akselerasjonen er konstant innefor hvert lille tidsintervall! Tiden t_i etter i intervaller er $t_i = i \cdot \Delta t$.

Når vi antar at akselerasjonen er konstant innenfor tidssteget Δt kan vi benytte oss av formlene over, og bruke at s_0 og v_0 er posisjonen og hastigheten ett tidsintervall tidligere. Resultatet blir et rekke med mange posisjoner og farter som, akkurat som tiden, er oppdelt i små biter, slik at $s_i = s(t_i)$, og $v_i = v(t_i)$.



For hvert tidssteg må vi oppdatere akselerasjonen. Bevegelsesligningene blir da:

$$s_{i+1} = s_i + v_i \Delta t + \frac{1}{2} a_i (\Delta t)^2 \tag{3}$$

$$v_{i+1} = v_i + a_i \Delta t \tag{4}$$

$$t_{i+1} = t_i + \Delta t \tag{5}$$

Ved mange slike beregninger, kan vi regne på en bevegelse fra start til slutt, selv om akselerasjonen ikke er konstant. Å gjøre tusenvis av slike beregninger for hånd tar altfor lang tid, men en datamaskin kan gjøre det lynraskt!

Denne måten å regne på bevegelse kalles Eulers metode, etter matematikern Leonhard Euler, og ble beskrevet rundt 1770. I dag finnes den i noen ulike varianter som har blitt utviklet for å øke presisjonen yttligere, men prinsippet er det samme.

1.3 Litt om arrays og indeksering

Nå skal vi implementere Eulers metode i python, men først skal dere lære et par ting om arrays som vi ikke har gått i detalj på tidligere i kurset. Vi skal se at dette gjør det lettere og mer oversiktlig å lagre og hente ut tall, som er nyttig når vi skal lagre mange posisjoner i løpet av bevegelsen, og plotte dem til slutt.

Vi har sett at arrays ligner på en liste, og at vi kan lage dem med kommandoer som f.eks funksjonen `arange(0,10,0.1)`, som vil lage en array som går fra 0 til 10

med steg på 0,1. Mer generelt er en array en samling med data, kalt elementer i arrayen, som i motsetning til en liste har en gitt størrelse som blir fastsatt når vi lager den. Vi kan med andre ord ikke bruke `append()` for å øke størrelsen, slik vi er vant til lister.

Dette er ikke et problem i dette tilfellet, fordi vi på forhånd bestemmer oss for hvor små biter vi skal dele bevegelsen vår opp i, og dermed også hvor mange elementer arrayen trenger å ha. For å lage en array kan vi bruke kommandoen `zeros(n)`, som lager en array med `n` elementer, der alle elementene er 0.

```
from pylab import *

x = zeros(3)
print(x)
```

Dette vil gi outputen:

```
[0. 0. 0.]
```

For å 'få tak i' et element i en array, gir vi navnet på arrayen etterfulgt av en klammeparentes med et tall i, der hvert tall svarer til ett element i arrayen. Dette kalles *indeksering*. Første element har indeks 0, andre element har indeks 1 osv, til det siste elementet som har indeks `n-1` der `n` er antall elementer i arrayen. I eksempelet over vil `x[0]`, `x[1]` og `x[2]` være lik 0.

Indeksering kan brukes både til å hente ut og endre på elementer i en array. Ved f.eks å skrive

```
x[1] = 5
print(x[1])
print(x)
```

vil vi endre på det andre elementet i arrayen og outputen blir:

```
5.0
[0. 5. 0.]
```

Her er et til eksempel på hvordan vi kan bruke en array og indeksering.

```
tall = zeros(10)

#printerarrayene med bare 0
```

```
print(tall)

#fyller arrayen med kvadrattall:
for i in range(0,10):
    tall[i] = i**2

#printer den "nye" arrayen i sin helhet
print(tall)
#og hvert tall for seg
for i in range(0,10):
    print(tall[i])
```

Og outputen er:

```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[ 0.  1.  4.  9. 16. 25. 36. 49. 64. 81.]
0.0
1.0
4.0
9.0
16.0
25.0
36.0
49.0
64.0
81.0
```

2 Prosjekt: Sprettball!

Da har vi alt vi trenger for å sette i gang og simulere bevegelsen til en sprettball!

2.1 Oversikt over bevegelsen

La oss gå igjennom hva som påvirker ballens bevegelse.

1. Bevegelsen vil begynne som et skrått kast, ikke ulik slik vi har programmert

kast tidligere i kurset. Denne bevegelsen er bestemt av utgangsposisjon, utgangsvinkel og hastighet, i tillegg til tyngdekraften g.

2. Når ballen treffer bakken vil den komprimeres, og denne komprimeringen vil gi en kraft som 'spretter' ballen opp igjen. En god modell for dette er å tenke på sprettballen som en springfjær som presses sammen, og vi kan bruke Hookes lov til å beskrive denne kraften

$$F_S = -kx \tag{6}$$

Der x er lengden på komprimeringen i meter, og k er en fjærkonstant som sier noe om hvor stiv fjæra/sprettballen er.

3. Under hele bevegelsen vil ballen være påvirket av en luftmotstand. Luftmotstand er en komplisert størrelse, men vi bruker her en modell der luftmotstanden er et resulatet av kvadrated av hastigheten og en koeffisient D som blant annet avhenger av tverrsnittet til ballen og lufttettheten:

$$F_D = -D|v|v \tag{7}$$

Her er $|v| = \sqrt{v_x^2 + v_y^2}$, mens v vil være farten i den retningen vi regner på, altså v_x eller v_y.

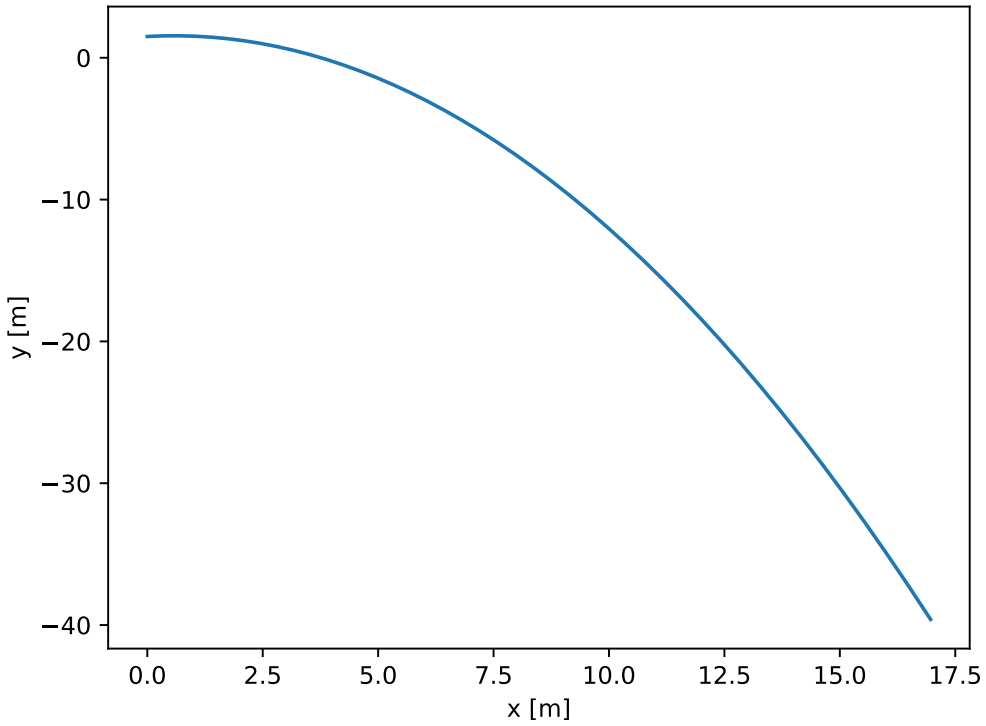
2.2 Skrått kast

Vi begynner med å programmere ett skrått kast slik vi har gjort tidligere, med med Eulers metode denne gangen. Vi vil lagre posisjonen og farten underveis i bevgelsen, så vi lager arrays for dette der hvert element i arrayene er posisjonen og farten i ett gitt tidssteg. Vi separerer bevegelsen i x og y retning, slik at vi har en array for x-posisjonen og en for y-posisjonen, og tilsvarende for farten. For å bestemme størrelsen på arrayene velger vi et tidssteg `dt` og en total tid `T` og bestemmer `n = int(T/dt)`. Vi programmerer:

```
from pylab import *

#konstanter:
g = 9.81 #(m/s^2) gravitasjon

#initialbetingelser:
h0 = 1.5 #(m) utgangshøyde
v0 = 8 #(m/s) utgangsfart
```

```
alpha = 45 #utgangsvinkel
theta = alpha/180 * pi

T = 3 #(s) totaltid
dt = 0.001 #(s) tidssteg
n = int(T/dt) #totalt antal steg/iterasjoner

#konstruere tomme arrays:
sx = zeros(n)
sy = zeros(n) #posisjon i x og y retning
vx = zeros(n)
vy = zeros(n) #fart i x og y retning
t = zeros(n) #tid

#setter h0 og v0 som første element i arrayene.
sy[0] = h0
vx[0] = sin(theta)*v0
vy[0] = cos(theta*v0)

ay = -g #konstant akselerasjon
ax = 0

#gjennomfører alle tidssegene
for i in range(0,n-1):
    sx[i+1] = sx[i] + vx[i]*dt + 0.5*ax*dt**2
    sy[i+1] = sy[i] + vy[i]*dt + 0.5*ay*dt**2

    vx[i+1] = vx[i] + ax*dt
    vy[i+1] = vy[i] + ay*dt

    t[i+1] = t[i] + dt

plot(sx,sy)
show()
```

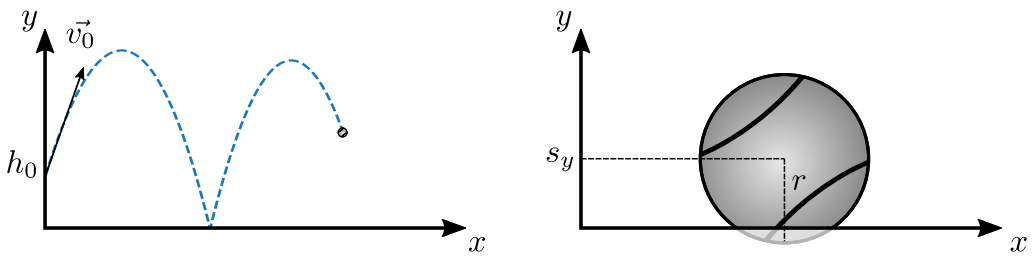
Da har vi simulert et skrått kast med Eulers metode! Som vi ser forsvinner ballen langt under $y = 0$. Det skal vi fikse nå, når vi legger til fjæregenskaper til sprettballmodellen vår.



2.3 Fjærmodell av sprettball

For å få til dette skal vi ta i bruk Hookes lov (6). Fjærkonstanten k kan vi variere som vi vil, men hvordan bestemmer vi komprimeringen?

Vi ser for oss at ballen har en gitt radius r. Når ballen er i ro på bakken burde avstanden fra ballens kordinat til x-aksen altså være r. Dette kan vi bruke! Dersom avstanden fra ballens koordinat s_y til x-aksen er mindre enn r kan vi innføre en fjærkraft $F_S = kx$, der x er komprimeringen, altså $x = r - s_y$.



Om denne kraften skal være der eller ikke i et gitt tidsintervall, kan vi sjekke med en if-test. Dersom $s_y < r$ får vi et tillegg til akselerasjonen slik at

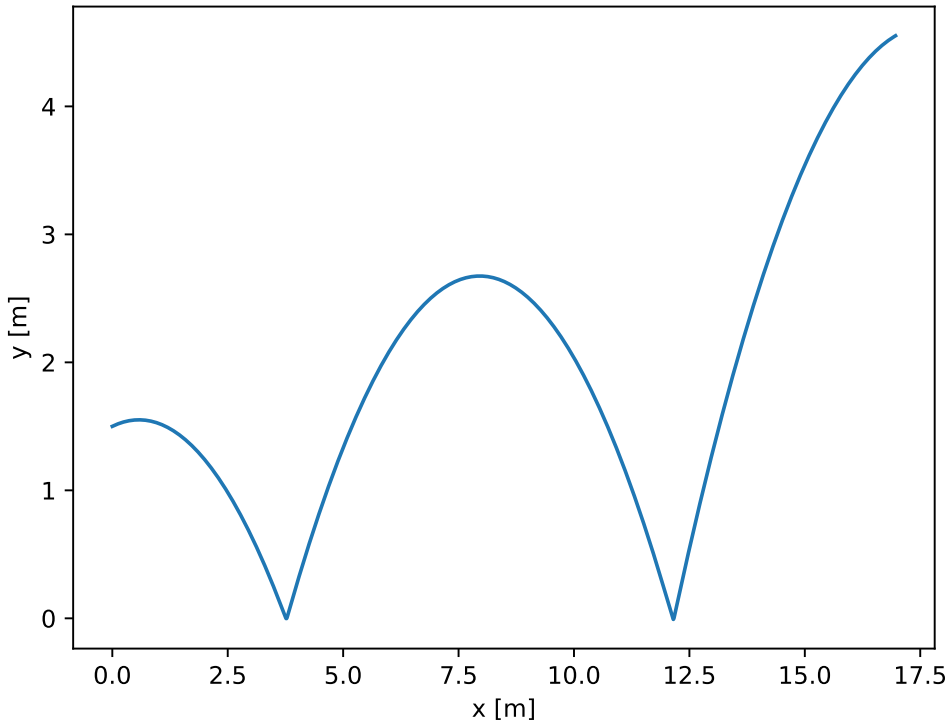
$$a_y = -g + \frac{F_S}{m} = -g + \frac{(r - s_y)k}{m} \tag{8}$$

Vi legger til det vi trenger av konstanter:

```
#konstanter:
g = 9.81 #(m/s^2) gravitasjon
m = 0.008 #(kg) masse
r = 0.015 #(m) radius
k = 1000 #(N/m) fjærkonstant
```

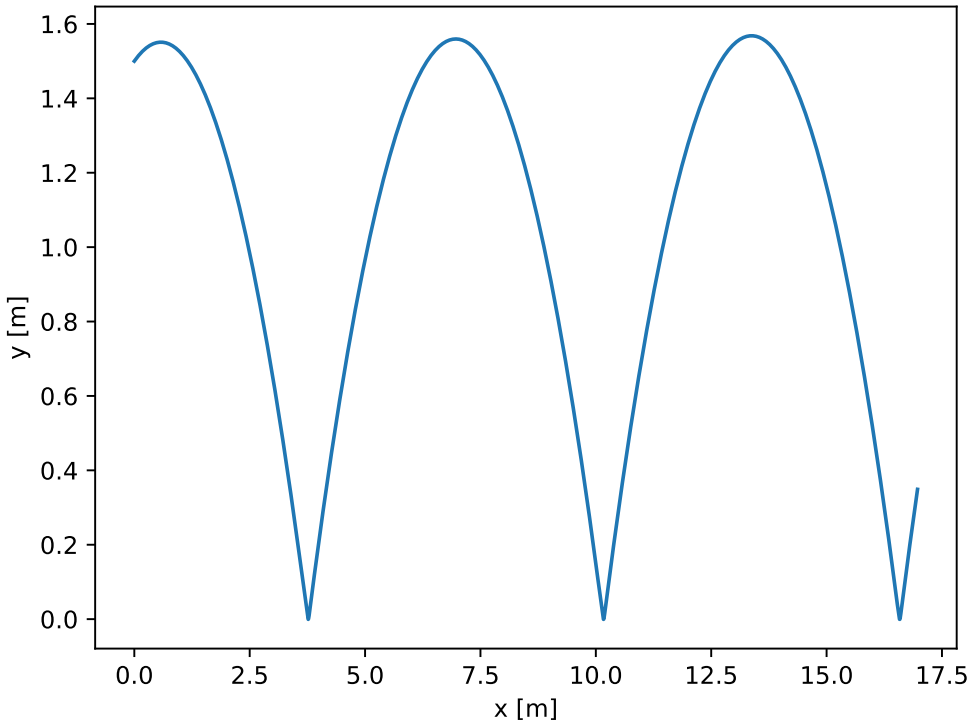
Og oppdaterer for-løkka med den nye akselerasjonen:

```
for i in range(0,n-1):
    if sy[i] < r:
        ay = -g + (r-sy[i])*k/m
    else:
        ay = -g

    sx[i+1] = sx[i] + vx[i]*dt + 0.5*ax*dt**2
    sy[i+1] = sy[i] + vy[i]*dt + 0.5*ay*dt**2

    vx[i+1] = vx[i] + ax*dt
    vy[i+1] = vy[i] + ay*dt

    t[i+1] = t[i] + dt
```



Men her er det noe rart. Intuisjonen vår bør fortelle oss at ballen ikke bør kunne få økt høyde på denne måten, for det betyr at den får mer energi! Vi husker at en forutsetning for å bruke Eulers metode var at tidsstegene var små nok til at vi kan anta at akselreasjonen er konstant i hvert tidssteg. Dersom det ikke er tilfellet, bryter modellen sammen. Vi må altså ha små nok tidssteg til at energien holder seg konstant over lengere tid.

For å sikre dette prøver oss vi frem med ulike verdier for tidsstegene `dt`, til vi ser at den ikke når mer høyde. Dette burde ligge omkring `dt = 1e-5`. Dersom datamaskinen din bruker lang tid på å kjøre programmet med dette tidsintervallet, kan du justere ned k. Dette vil gjøre at du klarer deg med et større tidsintervall.



Vi har nå en ball som ikke får tilført høyde, og er klare for å legge til luftmotstand.

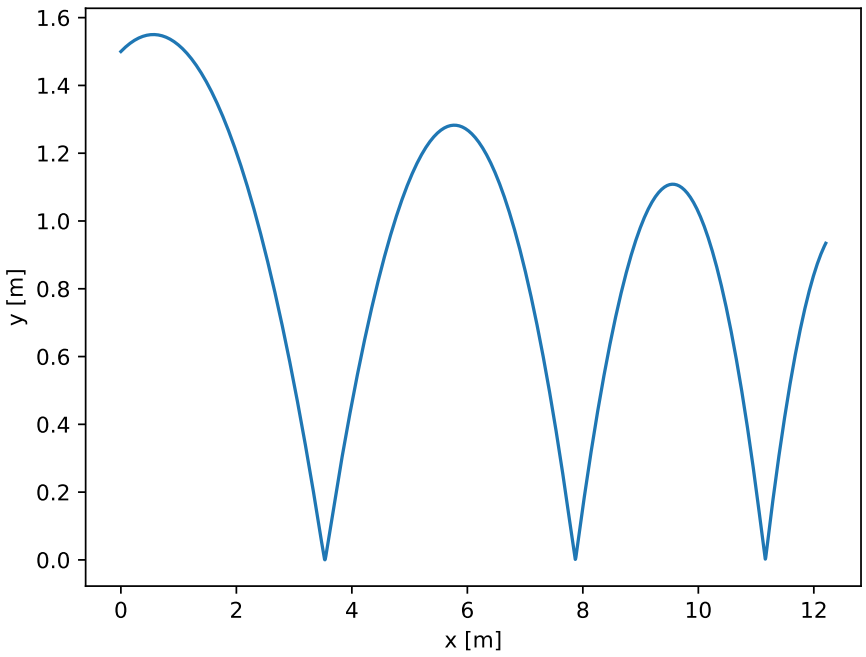
2.4 Luftmotstand

I motsetning til gravitasjonen og fjækraften vi har sett på frem til nå, vil luftmotstanden også påvirke bevegelsen i x-retning. Hvor sterk kraften (drag) F_D er i x- og y-retning, vil avhenge av farten i hver av retningene. Vi legger inn konstanten D i programmet vårt, samt en ny oppdaternig til akselerasjonen i løkken vår, som nå vil være:

$$a = -g - \frac{D|v|v}{m} \tag{9}$$

```
1 D = 3.5e-4 #(kg/m) lutftmotstand
```

```
for i in range(0,n-1):
    #regner ut total hastighet
    v_tot = sqrt(vx[i]**2 + vy[i]**2)

    if sy[i] < r:
        ay = -g -D*v_tot*vy[i]/m + (r-sy[i])*k/m
    else:
        ay = -g -D*v_tot*vy[i]/m

    ax = -D*v_tot*vx[i]/m

    sx[i+1] = sx[i] + vx[i]*dt + 0.5*ax*dt**2
    sy[i+1] = sy[i] + vy[i]*dt + 0.5*ay*dt**2

    vx[i+1] = vx[i] + ax*dt
    vy[i+1] = vy[i] + ay*dt

    t[i+1] = t[i] + dt
```



2.5 Energitap i sprettet

Vi har nå sett på hvordan luftmotstanden vil gjøre at ballen sprettet mindre, men i virkeligheten vil energien som ballen taper når den i kontakt med bakken, være en viktigere faktor til energitap, og dermed tap av høyde. Dette er litt komplisert fysikk, men formelen er ikke spesielt vanskelig. Vi har brukt formelen:

$$F = -kx \tag{10}$$

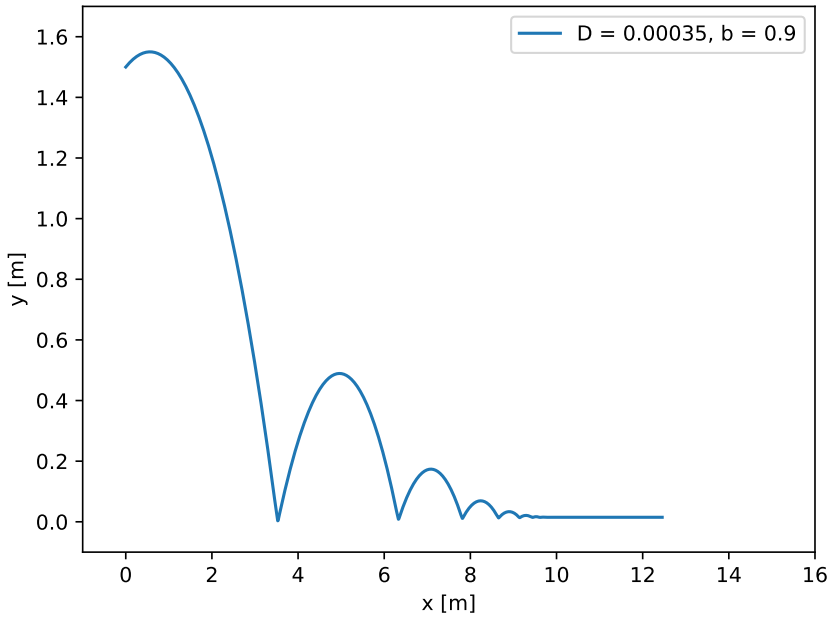
For å beskrive kraften i fjæra. Ved å legge til et dempingsledd $-b \cdot v$, får vi i stedet:

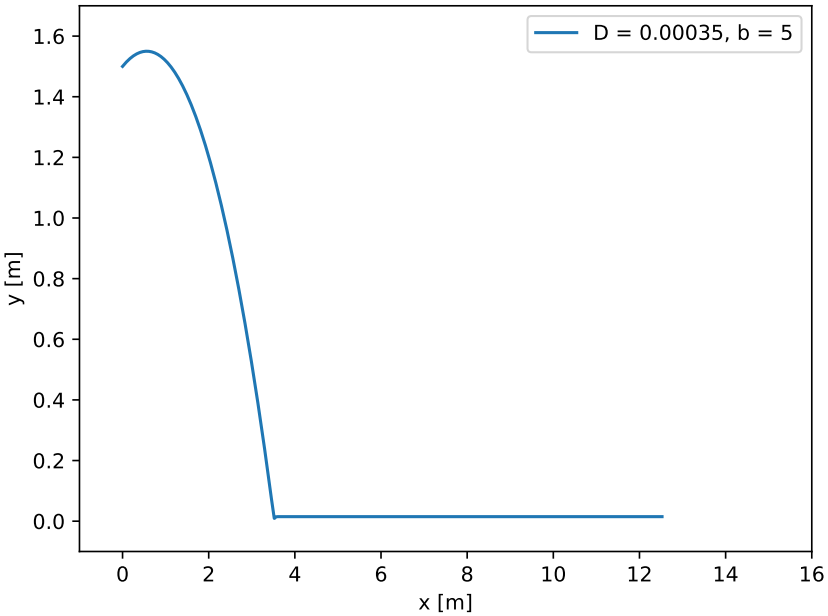
$$F = -kx - bv \tag{11}$$

Der v er farten og b er en dempingskonstant som sier noe om hvor mye energi som tapes i sprettet. Vi oppdaterer akselerasjone igjen:

```
b = 0.9 #demping

...
#i for-løkka:
if sy[i] < r:
            ay = -g -D*v_tot*vy[i]/m + (r-sy[i])*k/m - b*
               vy[i]/m
        else:
            ay = -g -D*v_tot*vy[i]/m
```

Vi kan teste for ulike dempingsfaktorer, f.eks $b = 0.9$ for en tennisball, og $b = 5$ en jernkule:



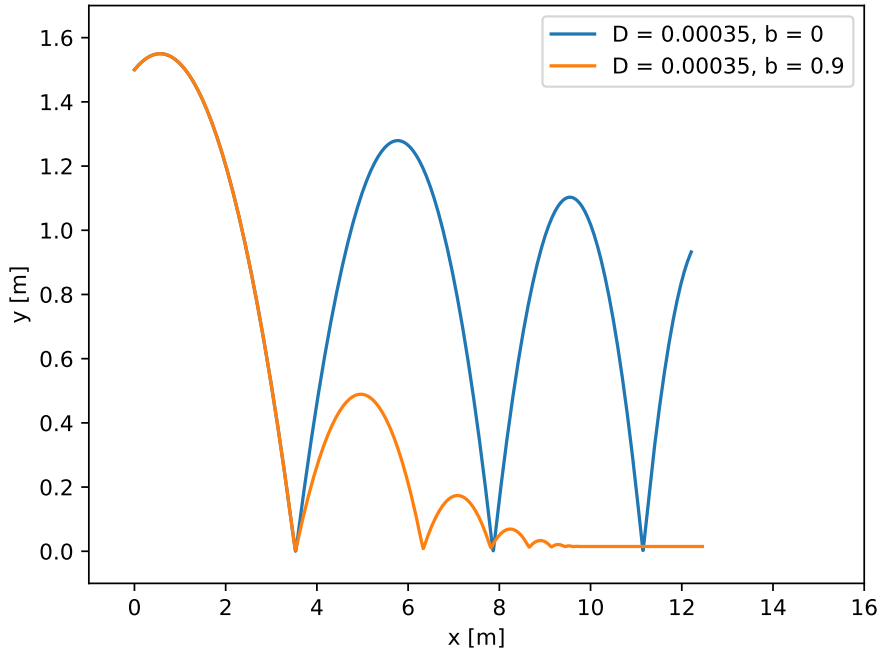


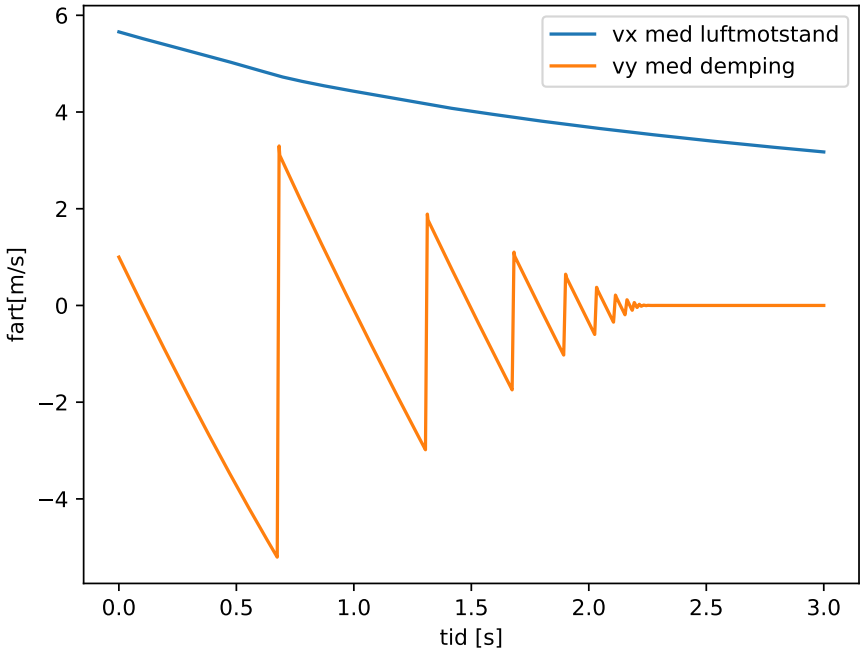
2.6 Litt bonus

Modellen vår er nå komplett, og vi kan leke oss litt med å plotte med luftmotstand og med og uten demping, samt se hvordan farten i y-retning oppfører seg over tid:

```
from pylab import *

#konstanter:
g = 9.81 #(m/s^2) gravitasjon
m = 0.008 #(kg) masse
r = 0.015 #(m) radius
k = 1000 #(N/m) fjærkonstant
D = 3.5e-4 #(kg/m) lutftmotstand
b = 0.9 #demping

#initialbetingelser:
h0 = 1.5 #(m) utgangshøyde
v0 = 8 #(m/s) utgangsfart
alpha = 45 #utgangsvinkel
theta = alpha/180 * pi

T = 3 #(s) totaltid
dt = 5e-6 #(s) tidssteg
n = int(T/dt) #totalt antal steg/iterasjoner

#konstruere tomme arrays:
sx = zeros(n)
sy = zeros(n) #posisjon i x og y retning
vx = zeros(n)
vy = zeros(n) #fart i x og y retning
t = zeros(n) #tid

#setter h0 og v0 som første element i arrayene.
sy[0] = h0
vx[0] = sin(theta)*v0
vy[0] = cos(theta*v0)

ay = -g #konstant akselerasjon
ax = 0

figure(1)
```

```
for b in [0,0.9]:
    #gjennomfører alle tidssegene
    for i in range(0,n-1):
        v_tot = sqrt(vx[i]**2 + vy[i]**2) #regner ut total
            hastighet

        if sy[i] < r:
            ay = -g -D*v_tot*vy[i]/m + (r-sy[i])*k/m - b*
               vy[i]/m
        else:
            ay = -g -D*v_tot*vy[i]/m

        ax = -D*v_tot*vx[i]/m

        sx[i+1] = sx[i] + vx[i]*dt + 0.5*ax*dt**2
        sy[i+1] = sy[i] + vy[i]*dt + 0.5*ay*dt**2

        vx[i+1] = vx[i] + ax*dt
        vy[i+1] = vy[i] + ay*dt

        t[i+1] = t[i] + dt


    plot(sx,sy,label=f"D = {D}, b = {b}")
    xlabel("x [m]")
    ylabel("y [m]")
legend()
show()

figure(2)
plot(t,vx,label="vx med luftmotstand")
plot(t,vy,label="vy med demping")
legend()
xlabel("tid [s]")
ylabel("fart[m/s]")
show()
```


D = 0.00035, b = 0
D = 0.00035, b = 0.9
y [m]
x [m]
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0
2
4
6
8
10
12
14
16


vx med luftmotstand
vy med demping
fart[m/s]
tid [s]
6
4
2
0
−2
−4
0.0
0.5
1.0
1.5
2.0
2.5
3.0

En videreutvikling av oppgaven kan være å plotte kinetisk, potensiell og total energi mot tiden, her er et eksempel på hvordan der kan gjøres. La oss først se på tilfellet uten luftmotstand. Vi trenger ikke endre på koden vår for å fjerne luftmotstanden fra for-løkka, det holder å sette $D = 0$. Den kinetiske og potensielle energien er definert slik:

$$E_k = \frac{1}{2}mv^2 \tag{12}$$

$$E_p = mgh \tag{13}$$

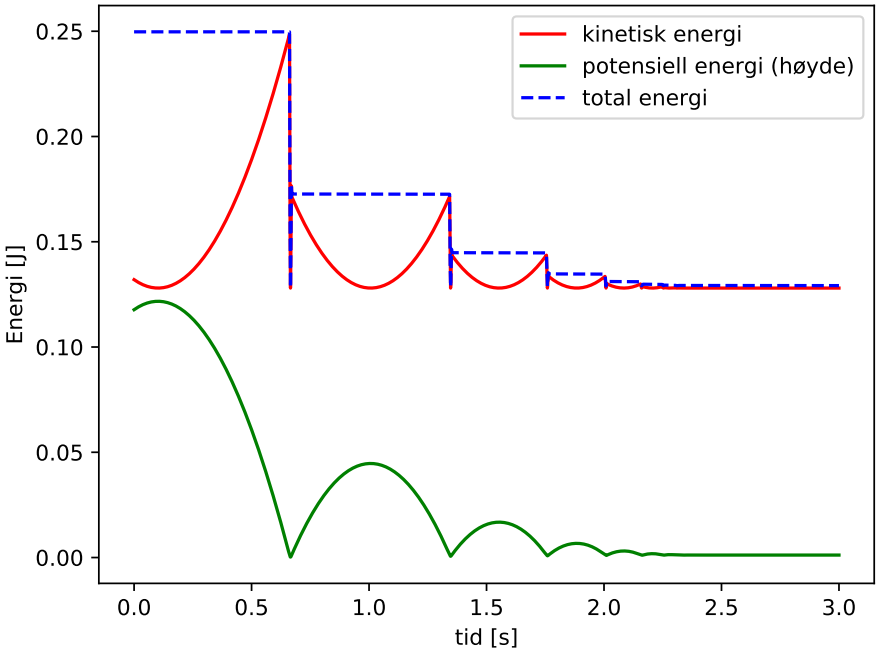
$$E_{\text{tot}} = E_k + E_p \tag{14}$$

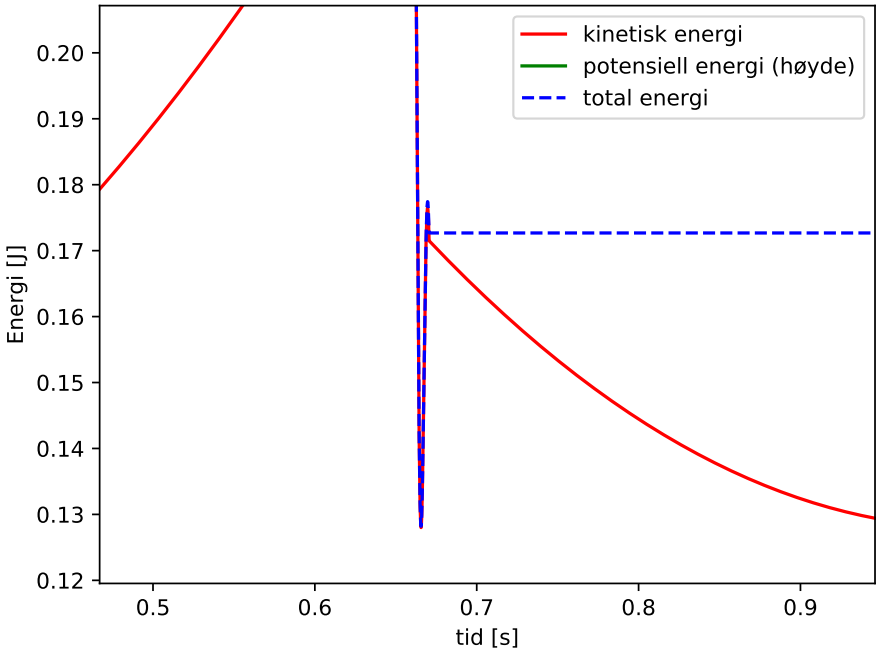
Der v er den totale hastigheten $v = \sqrt{v_x^2 + v_y^2}$, h er høyden over bakken s_y.

```
#finner total fart
v_tot = sqrt(vx**2 + vy**2)

E_k = 0.5*m*v_tot**2
E_p = m*g*sy
E_tot = E_k + E_p

plot(t,E_k,'r-',label='kinetisk energi')
plot(t,E_p,'g-',label='potensiell energi (høyde)')
plot(t,E_tot,'b--',label='total energi')
legend()
xlabel('tid [s]')
ylabel('Energi [J]')
show()
```

På figuren under ser vi at den totale energien går ned i trappetrinn, ballen mister noe energi hver gang den treffer bakken! Ettersom vi ikke har med noe luftmotstand vil den totale energien ende opp som den kinetiske energien vi får fra ballens fart i x- retning.

Men - om vi zoomer inn, ser vi at den totale energien får et plutselig fall, før den går opp igjen, hver gang ballen treffer bakken, og den går et 'trappetrinn' ned. Det kan da ikke stemme, for den totale energien burde ikke da gå opp igjen? Årsaken til dette er at vi mangler en energi.


kinetisk energi
potensiell energi (høyde)
total energi
Energi [J]
tid [s]
0.25
0.20
0.15
0.10
0.05
0.00
0.0
0.5
1.0
1.5
2.0
2.5
3.0


kinetisk energi
potensiell energi (høyde)
total energi
Energi [J]
tid [s]
0.20
0.19
0.18
0.17
0.16
0.15
0.14
0.13
0.12
0.5
0.6
0.7
0.8
0.9

I tillegg til den kinetiske energien fra farten, og den potensielle energien fra høyden, vil det også være en potensiell energi i sprettballen når den er komprimert. Dette er som den potensielle energien til en spent fjær. Vi kaller denne E_f, og energiene er definert slik:

$$E_k = \frac{1}{2}mv^2 \tag{15}$$

$$E_p = mgh \tag{16}$$

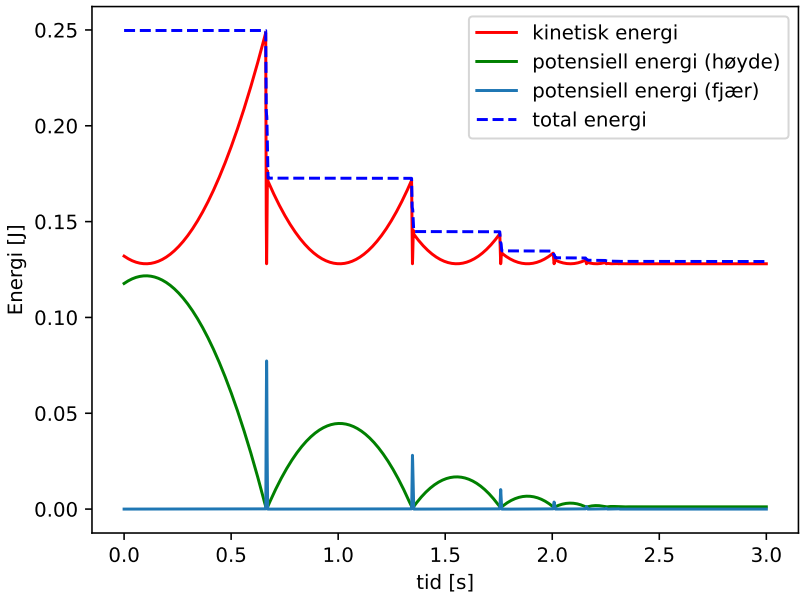
$$E_f = \frac{1}{2}kx^2 \tag{17}$$

$$E_{\text{tot}} = E_k + E_p + E_f \tag{18}$$

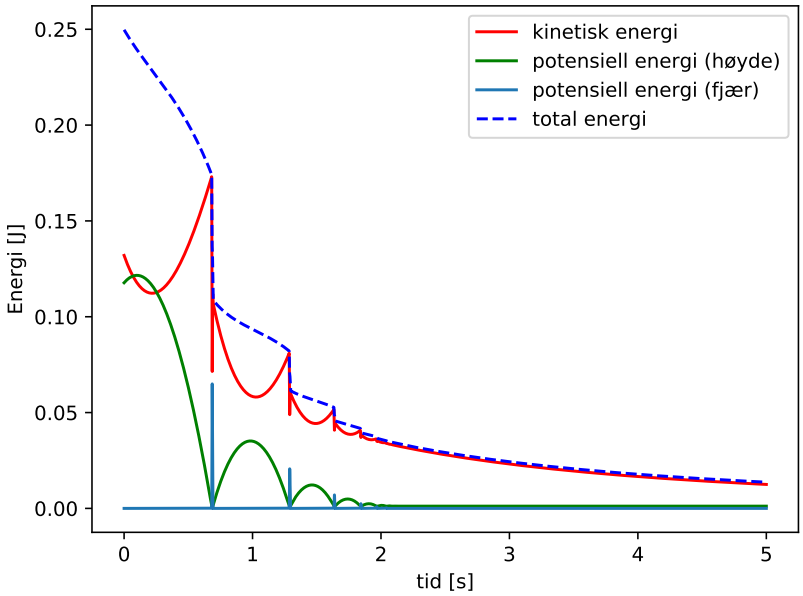
x er kompresjonen $r - s_y$.

Når vi programmerer dette lager vi en egen array `x`, der `x[i]` er hvor mye ballen er komprimert ved tiden t_i

```
v_tot = sqrt(vx**2 + vy**2)

x = zeros(n)

for i in range(n):
    if sy[i] < r:
        x[i] = r-sy[i]


E_k = 0.5*m*v_tot**2
E_p = m*g*sy
E_f = 0.5*k*x**2
E_tot = E_k + E_p + E_f

plot(t,E_k,'r-',label='kinetisk energi')
plot(t,E_p,'g-',label='potensiell energi (høyde)')
plot(t,E_f,label='potensiell energi (fjær)')
plot(t,E_tot,'b--',label='total energi')
legend()
xlabel('tid [s]')
ylabel('Energi [J]')
show()
```



Nå oppfører den totale energien seg slik som den skal, der den mister energi hver gang den spretter i bakken, men energien er ellers bevart. Til slutt kan vi legge til en luftmotstand på f.eks `D=6e-4`, og vi får følgende energier:



3 Oppsummering

I dette eksempelet på et prosjekt, har vi sett hvordan vi kan bruke programmering i fysikken til å regne på et problem med en sammensatt og ikke konstant akselerasjon ved hjelp av Eulers metode. Det er viktig å merke seg at metoden vi har introdusert og tatt i bruk i dag, er veldig generell, og kan brukes på mye mer enn bare de klassiske bevegelseslinjene. Har vi et uttrykk for akselerasjonen, og initialbetingelser, kan vi regne på utallige former for bevegelse. Noen eksempler på prosjekter man kan gjøre i fysikk er f.eks. å regne på et fallskjermhopp med luftmotstand, der fallskjermen løser seg ut etter en gitt tid, eller sirkelbevegelse og sentripetalakselerasjon for f.eks planeter. Det kan være interessant å se hvor lite tidssteget må være for at planetene skal holde seg i bane. I dette prosjektet har vi modelettert

Faktisk er Eulers metode, og varianter av den, et verktøy for å løse generelle differensialligninger, og derfor veldig nyttig, og mye brukt også utenfor fysikk. Implementering av denne metode åpner dører for å løse et hav av tidligere uløste problemer, og med det vi har gått igjennom i dag har dere fått en innføring i noe veldig sentralt innenfor numerisk matematikk.