FY1008_M03

September 23, 2025

1 Module03: Introduksjon til numpy

I denne modulen skal vi introdusere Python modulen som kalles *numpy*. Denne modulen er spesielt laget for at python skal bli kunne brukes for å gjøre numeriske beregninger, noe som står sentralt i fysikken.

Den sentrale typen som numpy er basert rundt er den såkalte *numpy.ndarrays*. Denne typen er en array type, dvs. vektorer, matriser og flerdimensjonale arrayer, som blir brukt til å lagre numeriske data som blir brukt i numeriske beregninger. Denne typen kan *bare* lagre numeriske data av de innebygde typene integers, floats eller complex. I motsetning til hva som er tilfelle for de innebygde Python typende har numpy støtte for *både* single og double presisjons flyttall (kalt float og float64); dette er tilsvarende som for de vanlige kompilerte programmeringsspråkene some C, C++, eller Fortran (samt julia).

Som en første introduksjon til numpy, vil vi bruk deler av de mange gode online numpy resursene som finnes.

Spesielt vil vi anbefale:

- NumPy Quickstart
- NumPy for beginners
- NumPy fundamentals
- NumPy Reference Manual

Det er *viktig* at du mestrer de fundamentale aspektene rundt numpy da det er en viktig, muligens, den viktigste, byggesteinen for å gjøre numeriske beregninger i python (på en effektiv og forholdvis rask måte). Det finnes mange ulike funksjoner og metoder som er del av numpy. Det er derfor ikke trolig at du på kort sikt vil ha full oversikt over dem alle bare ved å følge forelesningene. Jeg oppdager fortsatt nye aspekter ved denne modulen. Dog, det viktigste er å komme igang, og så lærer man mens man jobber med numpy. I starten er det viktigst å mestre

- opprettelsen av numpy arrays
- indeksering av elementene i arrayen
- oppdatering av enkelt elementer i arrayen
- reshaping
- kjenne til og å kunne bruke sentrale array funksjoner (f.eks. sum, eig,…)
- …

Vi starter nå forsiktig med å ta for oss deler av innholdet i quickstart guiden. Denne, og de tre andre linkene ovenfor, anbefaler jeg *sterkt* at du går gjennom på egen hånd og prøver å kjøre eksemplene som blir gitt der. Det som blir presentert nedenfor gir bare en liten smakebit på numpy, og vil ikke

være tilstrekkelig for å mestre de sentrale elementene av denne ytterst viktige modulen i python for numeriske beregninger.

[ ]:
```
import numpy as np            # import statement

print(np.__version__)         # Hvilken versjon av numpy kjører vi

print( np.__doc__ )           # Hva inneholder numpy modulen
```

[ ]:
```
# Alternativt for innholdet i en modul:  dir( <modulnavn> )
dir( np )

# Eller, hold muspekeren over np.
```

1.0.1 Deklarere en numpy array

[ ]:
```
a = np.array( [1,  2,  3,  4,  5] )                      # Integer
#a = np.array( [1.1,2.2,3.3,4.4,5.5] )                    # Default: double␣
 ↪presisjon
#a = np.array( [1.1,2.2,3.3,4.4,5.5], dtype=np.float32 )  # Single presisjon;␣
 ↪do not use float (uten np.) herS

print(a)

print( type( a )  )
print( a.dtype )                                         # typen til elementene
```

Summen og produktet av elementene i en np.array er:

[ ]:
```
a_sum = a.sum()          # summen av elementene
a_prod = a.prod()        # produktet av elementene

print(" Summen av elementene i numpy array a er    : ", a_sum )
print(" Produktet av elementene i numpy array a er : ", a_prod)
```

np.array generert fra en liste…….(der var forsåvidt hva vi gjorde ovenfor)

[ ]:
```
list =  [1,  2,  3,  4,  5]
aa =np.array( list )

print(aa)

if np.linalg.norm( a-aa) < 5e-14:      # Normen | aa -a |
    print("Arrayene er LIKE!")
else:
    print("Arrayene er ULIKE!")
# end if
```

eller via en tuple

[ ]:
```
# np.array via en liste
a = np.array( [1,  2,  3,  4,  5] )

# np.array via en tuple
b = np.array( (1,  2,  3,  4,  5 ) )

print( type(b) )      # b er en numpy.ndarray

a is b                # a og b er ikke det samme objektet

if np.linalg.norm(b-a) < 5e-14:       # Normen | aa -a |
    print("Arrayene er LIKE!")
else:
    print("Arrayene er ULIKE!")
## end if
```

[ ]:
```
# Du kan gjøre det samme på denne måten!
c = np.array( range(1,6))
print(c)
```

[ ]:
```
# Merk deg
if np.linalg.norm( aa-c) < 5e-14 :       # Normen | aa -c |
    print("Arrayene er LIKE!")
else:
    print("Arrayene er ULIKE!")
# end if


# Merk deg
# ----------------------------
# Hva blir skrevet ut her?
#print( aa is c )
#
#c = aa.copy()
#print( aa is c )
#
#c = aa
#print( aa is c )
```

Mao kan man lett initiere en numpy array via (a) en liste; (b) en tuple; eller (c) en range!

I numpy er arrays default av type til inputen til np.array funksjonen. Dog kan man overstyre dette med den valgfrie parameteren *dtype*

[ ]:
```
v=np.array( range(5) )
print( v.dtype )

v=np.array( range(5), dtype=np.float64 )
```

```
print( v.dtype )

v=np.array( [0,1,2,3,4+0j] )
print( v.dtype )

v=np.array( range(5), dtype=np.complex64 )
print( v.dtype )
```

Hva med matriser eller multi-dimensjonalle (ofte kalt n-dimensjonalle) arrays? Hvordan kan vi definere dem?

[ ]:
```
# Metode 1: Som flere vektorer.....
A1 = np.array( [[1,2,3],[4,5,6]] )    # Ev. A1 = np.array( [(1,2,3),(4,5,6)] )

print("A1 = \n",A1 )
```

[ ]:
```
# Metode 2: bruk av reshape....
tmp = np.array( [1,2,3,4,5,6] )
A2 = tmp.reshape( (2,3) )

print("A2 =\n",A2 )

# Ev som en kombinert funksjon
A2_0 = np.array(range(1,7)).reshape( (2,3))
print("A2_0 =\n",A2_0 )
```

Hva skjedde her....?

[ ]:
```
help( np.reshape )      # for hjelp

#A2_1 = tmp.reshape( (2,3) )
#print("A2_1 =\n",A2_1 )

#A2_2 = tmp.reshape( (2,3), order='F')
#print("A2_2 =\n",A2_2 )
```

[ ]:
```
# Numpy's range funksjon; Denne genererer en np.array
v = np.arange(5)
print(v, type(v) )

#
v = np.arange(1,6)
print(v)

print( np.arange(1,10,2) )
```

[ ]:
```
# np.arange fungerer også for flyttall
print( np.arange(5.) )
```

```
print( np.arange(0., 1., 0.1) )
```

Det finnes også en nær beslektet numpy funksjon til arange som heter linspace som er nyttig.

[ ]:
```
help(np.linspace)
```

[ ]:
```
v1=np.arange(0.,1.,0.1)
v2=np.linspace(0.,1.)
v3=np.linspace(0.,1, 11)
v4=np.linspace(0.,1, 10, endpoint=False)

print("v1 =", v1)
print("v2 =", v2)
print("v3 =", v3)
print("v4 =", v4)
```

Merk deg forskjellen i default oppførsel mellom *np.arange* og *np.linspace* når det gjelder hvordan endepunktet blir behandlet.

np.linspace funksjonen brukes mye f.eks. skal plotte data. La oss vise hvordan dette kan gjøres for $\cos(2\pi x)$ og $\sin(2\pi x)$ for $x \in [0, 1]$.

[ ]:
```
N=100      # antall punkter
x = np.linspace(0.,1.,N)
Cos = np.cos( 2* np.pi * x )
Sin = np.sin( 2* np.pi * x )

import matplotlib.pyplot as plt
plt.plot(x,Cos)
plt.plot(x,Sin)

plt.xlabel("x")
plt.ylabel(r"$\cos(2\pi x)$ and $\sin(2\pi x)$")
```

1.0.2 Ulike nyttige kommandoer i numpy

[ ]:
```
A = np.array(range(1,7)).reshape((2,3))      # Vår matrise

print( A.ndim )                  # antall dimensjoner (her en matrise)
print( A.shape )                 # Dimensjonen
print( A.size )
print( A.shape[0] )              # Antall rekker i matrisen (index 0)
print( A.shape[1] )              # Antall kolonner i matrisen (index 1)
```

[ ]:
```
# Null matrisen
np.zeros( (2,3) )      # default float64

#np.zeros( (2,3),  dtype=np.int32 )
```

[ ]:
```
# ener matriser
np.ones( (2,3) )

# fra en maske
B = np.ones_like(A)
print(B)
```

[ ]:
```
# Identitets matrisen (matrisen må være kvadratisk)
np.eye( 3 )
```

Det er mange, mange flere nyttige kommandoer i numpy, men for en beskrivelse av dem henviser vi til ekstern dokumentasjon. Når det er sagt, påpeker vi at det er *vell* verdt å kjenne til samt å kunne bruke disse. Derfor bør du jobbe deg gjennom eksemplene som er gitt i de eksterne linke gitt overfor (eller andre steder).

Verktoriell oppførsel av numpy.ndarray objekter La oss starte med å genere noen random tall:

[ ]:
```
help( np.random.random )
```

[ ]:
```
# Fyller matrisen med tilfeldige (random) tall mellom 0 og 1.
a = np.random.random( (2,3) )
print("a =\n",a)
```

[ ]:
```
# En ny random matrise
b = np.random.random( (2,3) )
print("b =\n",b)
```

Vi kan også sette et såkalt *seed* for den tilfeldige tall generatoren. Dette seeded gjør at vi får generert *samme* sekvens av tall. La oss se på følgende eksempel:

[ ]:
```
np.random.seed(42)
A = np.random.random( (2,3) )
print("A =\n",A)

np.random.seed(42)
B = np.random.random( (2,3) )
print("B =\n",B)
```

La oss ta sinus av alle elementene i a, dvs, S_{ij}=sin(a_{ij}). Man sier at sin-funksjonen er *verktoriell*.

Mange funksjoner i numpy har denne egenskapen.

[ ]:
```
sina = np.sin(a)   # La oss ta sinus av alle elementene i a
print("sin(a) = \n", sina)
```

[ ]:
```
# Et annet eksempel
A = np.arange(6).reshape((2,3))
print("A = \n", A)
```

```
Acumsum = A.cumsum()
print("Acumsum = \n", Acumsum)

# Hva skjedde her?
#help(np.cumsum)
```

[ ]:
```
# La oss ta cumulative sum over kolonner
Acumsum = A.cumsum(axis=1)
print("Acumsum = \n", Acumsum)
```

La oss se hvordan vi man bruke numpy til å løse et lineært ligningsystem $Ax = b$. Dette vil vi gjøre på følgende måte

- Generer en kvadratisk $(N \times N)$ matrise A med random elementer
- Tilsvarende la oss generere en løsnings vektor y som også er tilfeldig
- Høresiden generer vi via matrisemultilikasjon som $b = Ay$
- Ved hjelp av numpy skal vi så løse systemet Ax=b slik at og undersøke om vi får rett løsning, dvs $x = y$.

[282]:
```
# Set opp likningsystemet
N=25
A=np.random.random( (N,N) )
y=np.random.random( N )
b=np.matmul(A,y)

# print("A = \n", A)
# print("b = \n", b)
```

Merk: funksjonen np.random.random() gir uniformt fordelte tall over $[0, 1)$. For normal fordelte tall (Gaussisk) må man bruke *np.random.normal*.

[ ]:
```
# Hva er rutinen for å løse likningsystemet
help( np.linalg )

#help( np.linalg.solve )
#help( np.linalg.matmul )
```

[ ]:
```
# Løs likningsystemet
x = np.linalg.solve(A,b)

error = np.linalg.norm(y-x)
print( "error norm = ", error)

# samelikning av svarene
M=6
for i in range(M):
    print(i, "   x_i= ", x[i] )
# end for
```

Oppgave: Genere A og y med normal fordelte tall!

Sentrale rutiner i np.linalg

- norm : normen av en vektor eller matrise
- matmul : matrise multiplikasjon (ev. matrise vektor multiplikasjon)
- inv : inverse av en matrise
- solve : Løser et linært likningsystem $A\vec{x} = \vec{b}$
- eig : Finner egenverdien til en matrise
- det : Determinanten etc

for mer detaljer bruk *help(np.linalg)* eller *help(np.linalg.funksjons_navn)*.

La oss nå se på hvordan noen av disse rutine kan brukes....

Vi starter med å generere en random matrise, med uniformt eller normal fordelte tilfeldige tall.

[ ]:
```
# Oppsett av likningssystemet
import numpy as np
N   = 3       # System dimensjon
RHS = 1       # Antall høyresider

# Uniformt fordelte tall
A = np.random.random( (N,N) )
# A_copy = A.copy()               # en kopi til bruk senere
print("A =\n",A)
xx = np.random.random( (N,RHS) )
print("xx =\n",xx)

bb =np.zeros((N,RHS))
for i in range(RHS):
    bb[:,i] = np.matmul(A,xx[:,i])
# end for
print("bb =\n",bb)
```

Men $\vec{bb}$ er det samme som:

[ ]:
```
b = np.matmul(A,xx)
# Alternativ for matrise multiplikasjon

# b = A @ xx
print("b =\n",b)
```

Men er løsning riktig?

[3]:
```
# Vi definerer en funksjon
def feil_funksjon(x,y):
    feil = np.linalg.norm(x-y)
    return feil
# end feil_funksjon
```

La oss regne ut løsningen $A\vec{x} = \vec{b}$ samt normen til feilen:

[ ]:
```
x = np.linalg.solve(A,b)
print("Feilen er :", feil_funksjon(x,xx) )
```

Dersom antall høyresider $RHS > 1$, hva skjer da? Hva gjør *np.linalg.solve* da?

For å finne det ut, la oss gå tilbake å sette en verdi for RHS som er større enn 1.

[20]:
```
import sys

# Vi definerer en funksjon
def feil_funksjon_2(x,y):
    ndim =x.ndim
    if ndim == 1:
        feil2 = np.linalg.norm(x-y)
    elif ndim == 2:
        M = x.shape[1]
        feil2 = np.zeros(M)
        for i in range(M):
            feil2[i] = np.linalg.norm(x[:,i] - y[:,i])
        # end for
    else:
        print("Arrayene har ikke konsistent dimensjon!")
        sys.exit()
    # end if
    return feil2
# end feil_funksjon_2
```

[ ]:
```
x = np.linalg.solve(A,b)
print("Feilen er :", feil_funksjon_2(x,xx) )
```

En enklere løsning er å lese grundig manualsiden til np.linalg.norm

[ ]:
```
help(np.linalg.norm)
```

[ ]:
```
print("norm(xx[:,0])   :", np.linalg.norm(xx[:,0]))
print("norm(xx)        :", np.linalg.norm(xx))
print("norm(xx,axis=0) :", np.linalg.norm(xx, axis=0))
```

La oss definere en funksjon som har en valgbart (opsjonelt) argument axis:

[7]:
```
def feil_funksjon_3(x,y, axis=None):
    if axis is None:
        feil3 = np.linalg.norm(x-y)
    else:
        feil3 = np.linalg.norm(x-y, axis=axis)
    # end if
    return feil3
# end feil_funksjon_3
```

Mer hvordan kan du lett kan lage en *docstring* for denne funksjonen ved å gi " " " + return:

[9]:
```
def feil_funksjon_3(x,y, axis=None):

    # Gi """ + return
    if axis is None:
        feil3 = np.linalg.norm(x-y)
    else:
        feil3 = np.linalg.norm(x-y, axis=axis)
    # end if
    return feil3
# end feil_funksjon_3
```

[ ]:
```
x = np.linalg.solve(A,b)
print("shape(x)  :", x.shape)
print("shape(xx) :", xx.shape)
print("Feilen er :", feil_funksjon_3(x,xx,axis=0) )
print("Feilen er (men hva betyr dette):", feil_funksjon_3(x,xx) )

# Enklere syntax
print("Feilen er (enklere syntax):", np.linalg.norm(x-xx,axis=0) )
```

Moralen her er, les DOKUMENTASJONENE!!!

OPPGAVE: gjennomfør den samme beregningen, men ved å bruke normal fordelte tall for matrisene!

Man kan også løse likningssystemet $A\vec{x} = \vec{b}$ ved å regne den inverse matrisen til A og matrise multiplisere $\vec{b}$ med denne, dvs.

$$\vec{x} = A^{-1}\vec{b}.$$

La oss nå teste dette:

[ ]:
```
x = np.linalg.inv(A) @ b
# x = np.matmul( np.linalg.inv(A), b )        # Alternativ
print("shape(x) :", x.shape )
print("Feilen er (men hva betyr dette):",  np.linalg.norm(x-xx,axis=0))
```

1.0.3 Egenverdier

En egenverdi likning er definert slik at $A\vec{v} = \lambda\vec{v}$ hvor $\vec{v}$ ikke er en null vektor. Spørsmålet er hva disse er $\vec{v}$ og hva er λ. Dette er et standard problem i lineær algebra, så vi vill ikke gå i detalj her.

[ ]:
```
help(np.linalg.eigvals)
help(np.linalg.eig)
```

[ ]:
```
# eigenvalues
w = np.linalg.eigvals(A)
print("Eigenvalues (lambda) :", w)
```

[ ]:
```
# eigenverdier og tilhørende eigenvektorer
w,v = np.linalg.eig(A)
print("Eigenvalues (lambda) :\n", w)
print("Eigenvektorer        :\n", v)
```

La oss teste resultatet $A\vec{v} = \lambda\vec{v}$

[ ]:
```
for i in range(w.size):
    vektor = A @ v[:,i] - w[i] * v[:,i]
    error =  np.linalg.norm( vektor )
    print( "vektro : ", vektor)
    print( "error (",i,") :", error)
# end for
```

Elementer i en numpy.ndarray

[ ]:
```
v = np.array( np.arange(6) )
A = (np.array( np.arange(9) )).reshape( (3,3) )

print( v[0] )      # først element
print( v[1:3] )    # andre til fjerde element
#print( v[:-2] )    # Alle elementer foruten de to siste
#print( v[-2:] )    # Alle elementer foruten de TRE første

#print( A[1,2] )    # etc...
#print( A[:,2] )    # 3. kolonne
```

[ ]:
```
print( "A = \n",A)
```

1.0.4 Determinater

[ ]:
```
help(np.linalg.det)
```

[ ]:
```
AA=np.array( [[ 1, 2],
              [3,4]])
print( " AA = \n", AA)
```

[ ]:
```
# determinanten til denne matrisen er
AAdet = np.linalg.det(AA)
print(" |AA| = ", AAdet)
```

[ ]:
```
print( " A = \n", A, "\n")
Adet = np.linalg.det(A)
print(" |A| = ", Adet)
```

Kopiering av numpy.ndarray

[ ]:
```
v = np.array( np.arange(6) )
```

```
# Intet NYTT Object lages
a = v

# Nytt object lages (deep copy)
b = v.copy()
c = v[:]FY1008_M04.pdf

#print( id(v),id(a), id(b), id(c) )
```

Tilsvarende som vi så tidligere, så må vi være forsiktige når vi kopierer np.arrays.

1.0.5 Operasjoner på numpy.ndarray

[ ]:
```
# definerer to np.arrays
a = np.array( [1,2,3] )
b = np.array( [1,4,6] )

c = a + b              # sum (element vis)
print ("c = ", c )     # samme for suptraksjon

d = a * b              # multiplikasjon (element vis)
print ("d = ", d )     # samme for divisjon (mer resultat er float)

dot = np.dot(a,b)
print("dot = ", dot)     # dot-produkt eller skalar produkt

cross = np.cross(a,b) # cross produktet
print("cross = ", cross )
```

Man kan også enkelt lese data from en file i ASCII format.

[ ]:
```
filename="https://web.phys.ntnu.no/~ingves/Teaching/FY1008/Downloads/Data/
  ↪square.dat"

square=np.loadtxt(filename)

print("Data som var lest er : \n", square )
```

[ ]:
```
import matplotlib.pyplot as plt

plt.plot(square[:,0], square[:,1])
plt.title("Graf av innholdet i data filen!")

plt.show()
```

For å lagre data som ASCII finnes det en tilsvarende routine np.savetxt som kan være nyttig.

[ ]:
```
np.savetxt( "/tmp/kast.dat", square )
#

# Finnes filen?   (%ls er en "magic-command")
%ls -l /tmp/kast.dat

# Hva inneholder denne filen?
#%cat /tmp/kast.dat

# Prøv også dette; Den lagrede filen er komprimert (og binær)
np.savetxt( "/tmp/kast.dat.gz", square )
```

[ ]:
```
# Man kan også lagre data vha np.save, men formatet er et binært internt Python␣
 ↪format (*.npy *.npz).
help(np.load)
```

1.1 Effektivitet av numpy.ndarray objekter

Vi har argumentert for at numpy er en ytterst god module når man skal gjøre numeriske beregninger i Python. Men, hva betyr dette, og hvor mye bedre ytelse vil man typisk kunne få dersom man bruker numpy.ndarrays for sine beregninger. Dette vil vi kort prøve å svare på nedenfor med noen enkle eksempler.

[ ]:
```
# Beregning med lister
import time
import math

# Antall elementer
N = 100000

# Sett starttid
time0 = time.time()

# Gjør beregningen
Sin = []              # Starter med en tom liste
X2  = []
for x in range(N):
    Sin.append( math.sin(x) )
    #X2.append(x*x)
# end for

# Sett sluttid
time1 = time.time()

time_list = time1-time0

print("Beregningen med lister tok (s) : ",time_list)
```

[ ]:
```
# Sett starttid
time0 = time.time()

# Beregning
x=np.arange(N)
Sin= np.sin(x)
#X2 = x*x

# Sett sluttid
time1 = time.time()

time_numpy = time1-time0

print("Beregningen med numpy tok (s) : ",time_numpy)

print("Speedup  (time_list/ time_numpy) : ", time_list/time_numpy)
```

Merk deg at disse eksemplene er bare illustrasjoner. For andre beregninger kan forskjellen være både større og mindre! Videre vil tiden en beregning tar kunne variere fra kjøring til kjøring!

Her er en annen sammelikning (inspirert av https://www.geeksforgeeks.org/python/python-lists-vs-numpy-arrays/):

[ ]:
```
# size of arrays and lists
size = 1000000

# declaring lists
list1 = range(size)
list2 = range(size)

# declaring arrays
array1 = np.arange(size)
array2 = np.arange(size)

#
# Using lists:
# -------------------------
#
# capturing time before the multiplication of Python lists
initialTime = time.time()

# multiplying elements of both the lists and stored in another list
resultantList = [(a * b) for a, b in zip(list1, list2)]

# calculating execution time
dt_list = time.time() - initialTime
print("Time taken by Lists to perform multiplication:",
    (dt_list),"seconds")
```

```
#
# Using numpy
# -------------------------
#
# capturing time before the multiplication of Numpy arrays
initialTime = time.time()

# multiplying elements of both the Numpy arrays and stored in another Numpy␣
 ↪array
resultantArray = array1 * array2

# calculating execution time
dt_numpy = time.time() - initialTime
print("Time taken by NumPy Arrays to perform multiplication:",
    (dt_numpy), "seconds")

# Speedup
print("Speedup : ", dt_list/dt_numpy)
```

1.1.1 Eksempel med bruk av numpy.array objekter

La oss se på problemet fra Øving 3, hvor en ball starter fra bakkenivå $z = z_0 = 0$ m med hastigheten $v_0 = 5$ m/s rett opp i vertikal retning. Den blir bare utsatt for tyngde kreften og vi ble bedt om å finne fremtidig posisjon (og hastighet). Hvordan kan vi løse denne oppgaven ved å bruke numpy.ndarray objekter (istedet for f.eks. lister)?

[294]:
```
import scipy as sp
from scipy import constants     # for å få konstanten g

g = constants.g

def euler(u,v,dt):              # Hva med doc streng her?
    unew = u + dt*v
    vnew = v - dt * g
    return unew, vnew
# end euler
```

[ ]:
```
import time

dt = 0.000005 # tidssteg  (Prøv med e.g. dt = 0.000005 )
t0 = 0        # start tidspunkt
u0 = 0        # start betingelser
v0 = 5

U   = u0 * np.ones((1,))  # for lagring av resultater
V   = v0 * np.ones((1,))
```

```
tid = t0 * np.ones((1,))

u  = u0
v  = v0
t  = t0

# Løser diff likningen
start =  time.time()
while u>=0:
    u,v = euler(u,v,dt)
    t += dt
    # Dette er fristende og det fungerer MEN er ikke effektivt
    tid = np.append(tid, t)
    U   = np.append(U,u)
    V   = np.append(V,v)
# end while
stopp = time.time()
tid_cpu = stopp-start
print("Tidsforbruk (s): ", tid_cpu)


# Plotter resultatene (bør vi definere dette i en funksjon?)
import matplotlib.pyplot as plt

plt.plot(tid,U)
plt.xlabel("t [s]")
plt.ylabel("posisjon [m]")
plt.show()
```

Problemet er at *np.append()* funksjonen generer et nytt *numpy.ndarray* objekt hver gang. Siden denne funksjonen er inne i en loop, er ikke dette effektivt da det blir mye kopieringer og minne allokering.

[ ]:
```
# For dokumentasjon
help( np.append )
```

En alternativ metode er å allokere alle np.arrays før man går inn i loopen, men man kjenner ikke hvor stor disse arrayene skal være. En mulig løsning som er mer effektiv blir presentert nedenfor, selv om noen vil hevde at den ikke er spesielt elegant.

[ ]:
```
N   = 500          # ev. N = int(1.3/dt)
U   = np.zeros(N)  # for lagring av resultater
V   = np.zeros(N)
tid = np.zeros(N)

u  = u0;  U[0]   = u0
v  = v0;  V[0]   = v0
t  = t0;  tid[0] = t0
```

```
n  = 0

# Løser diff likningen
start =  time.time()
while u>=0:
    u,v = euler(u,v,dt)
    n += 1
    t  = n * dt
    # Må vi utvide numpy arrayene?
    if n>N-1:
        null = np.zeros((N,) )
        tid  = np.append( tid, null )
        U    = np.append( U, null )
        V    = np.append( V, null )
        N    = N+N
    # end if
    # Her lagrer vi resultatene.....
    tid[n] = t
    U[n]   = u
    V[n]   = v
# end while

# Oppdaterer arrays (ved å fjerne hva man ikke har brukt)
U   = U[0:n-1]
V   = V[0:n-1]
tid = tid[0:n-1]

stopp = time.time()

# Rapporter tidsbruk
print("Tidsforbruk (s): ", stopp-start)
print("Speedup :", tid_cpu/(stopp-start))


# Plotter resultatene
import matplotlib.pyplot as plt

plt.plot(tid,U)
plt.xlabel("t [s]")
plt.ylabel("posisjon [m]")
plt.show()
```

Når vi nå har posisjon og hastighet kan vi lett regne ut kinetisk og potensiell energi for ballen. Dette gjør vi å definere en funksjon. [Merk deg hvordan doc strengen kan lages i code]

[ ]:
```
def energi_beregning(U,V,m=1):
    """
```

```
    Mekanisk energi for en ball som beveger seg vertikalt i tyngdefeltet.
    Som default regner man ut mekanisk energi for en enhets masse (m=1kg),
    eller om man vil, energi per masse enhet.

    Args:
        U (ndarray): posisjon som funksjon av tiden
        V (ndarray): hastighet som funksjon av tiden
        m (int, optional): ballens masse. Defaults to 1.

    Returns:
       Etot (ndarray): Total mekanisk energi som funksjon av tiden
       Ekin (ndarray): Kinetisk energi som funksjon av tiden
       Epot (ndarray): Potensiell energi som funksjon av tiden
    """
    # Kinetisk energi
    Ekin= 0.5 * m * V**2
    # Potensiell energi
    Epot = m * g * U
    # Total energi
    Etot = Epot + Ekin
    #
    return Etot, Ekin, Epot
```

Vi bruker nå denne funksjonen for å beregne energien for ballen og plotter resultatene.

[ ]:
```
# Beregner energien
Etot, Ekin, Epot =  energi_beregning(U,V)

# Plotter energien
plt.plot(tid,Etot,label='total')
plt.plot(tid,Ekin,label='kinetisk')
plt.plot(tid,Epot,label='potensiell')

plt.legend()
plt.xlabel("tid (s)")
plt.ylabel("energi/masse (J/kg)")

plt.show()
```

1.1.2 En siste oversikt over NumPy

En oppsummering av NumPy kan du finne her. Det kan være lurt å gå gjennom dette materialet for å oppfriske noe av de sentrale egenskapene med NumPy!

Bruk NumPy hvor du kan, da dette typiske r raskt for numeriske beregninger.