Introduksjon til Numpy og Matplotlib.pyplot

Ulrik Bernhardt Danielsen

January 20, 2021

0.1 Samarbeidsplattform

Hvor skal koden ligge og hvordan vet man at flere i gruppen ikke skriver det samme? Her har man flere alternativer, men det viktigste er at gruppen kommer frem til noe alle er komfortable med, og at dere tester det ut sammen i starten. Det mest effektive er kanskje å bruke Git og Github, men det forutsetter at alle bruker tid på å sette seg inn i det, da det er en noe uvant måte å jobbe på. Et enklere alternativ er Google Colab, hvor man kan jobbe med en jupyter-notebook på tilsvarende måte som et LaTex-dokument i overleaf.

Tips!

Sett dere ned som gruppe i starten av hvert prosjekt og diskuter hvordan dere skal skrive kode sammen

0.2 Lesbar kode

Når man jobber sammen er det flere enn seg selv som skal lese koden. Dette setter større krav til oversiktlighet og lesbarhet.

0.2.1 Små funksjoner

Del opp koden i så små funksjoner som mulig, gjerne tilstrebe under fem linjer per funksjon. Dette gjør koden mye mer oversiktlig, noe som hjelper både en selv og gruppa.

0.2.2 Variabelnavn

Oversiktlige navn på funksjoner og variabler gjør mer enn man tror. Gode navn på funksjoner gjør at man ikke trenge å finlese koden for å forstå hva den gjør. Er det for eksempel åpenbart hva denne koden gjør?

```python
def f(x):
    for i in range(len(x)):
        print(x[i, i])
```

Samme kode med oversiktlige navn:

```python
def printDiagonalElementsOfMatrix(matrix):
    for i in range(len(matrix)):
        print(matrix[i, i])
```

Dette virker kanskje som mye unødvendig skriving for en så liten funksjon, men har man mange små funksjoner vil man mye raskere kunne skaffe seg en oversikt over koden hvis man er nøye men navngivningen.

0.2.3 Kommentarer

Skriver man kode alene kan man komme langt uten nevneverdig bruk av kommentarer. Dette gjelder ikke når man samarbeider, da er det helt essensielt. Kommenter ofte og presist. Eksempler på hva som er verdt å kommentere er hva funksjoner gjør, og hva variabler representerer. Forrige eksempel kunne man tenke seg å kommentere på følgende måte.

```python
""" Printer ut diagonalelementene i en matrise """
def printDiagonalElementsOfMatrix(matrix):
    # matrix: et nxn NumPy-array
    for i in range(len(matrix)):
        print(matrix[i, i])
```

Oppsummert

Tips for godt samarbeid på prosjektene:

- Finn ut som gruppe hvordan dere vil organisere koden. Sørg for at alle henger helt med før dere går i gang med skrivingen
- Skriv oversiktlig og lesbar kode. Dette oppnår man gjennom korte funksjoner, gode variabelnavn og fokus på å kommentere

1 NumPy

En forståelse av NumPy er essensielt i vitenskapelig programmering og noe man ikke slipper unna i TMA4320. Man bør ha noe erfaring med NumPy fra tidligere fysikkurs, så vi hopper over det helt grunnleggende. Hvis dette er første gang du bruker NumPy finnes det utallige gode introduksjoner et googlesøk unna. Videre i teksten går vi ut ifra at `import numpy as np` er kjørt.

1.1 NumPy-array

Helt sentralt i NumPy er typen `Numpy.ndarray`. I vitenskapelig programmering er denne typen å foretrekke over Pythons egne lister da de er optimalisert for kunne utføre matematiske operasjoner raskere og mer intuitivt.

> **Tips!**
>
> Bruk NumPy for det det er verdt! Får man valget mellom å bruke Pythons innebygde typer og funksjoner eller NumPys, da velger man NumPys

1.1.1 Shape

Et NumPy-array har en egenskap `shape` som beskriver dimensjonen. Egenskapen illustreres best med et eksempel:

```
""" Initialiserer et NumPy-array og printer shape egenskapen
    """
A = np.array([[1, 2, 3], [4, 5, 6]])

print(np.shape(A))
print(A.shape)
# Output:
(2, 3)
(2, 3)
```

I eksempelet var `A` et todimensjonalt NumPy-array, med to elementer i det ytterste arrayet og tre elementer i det innerste arrayet, og har dermed `shape` lik `(2, 3)`.

Dette blir svært viktig når man skal bruke NumPy til lineær algebra, spesielt under matrisemultiplikasjon da dimensjonen må stemme for at koden kjører. Å holde styr på shape-egenskapen til matrisene man jobber med sparer en for mye feilsøking. Koden under illustrerer hvordan måten man initialiserer et NumPy-array påvirker arrayet `shape`.

```
A = np.array([1, 2, 3])
B = np.array([[1, 2, 3]])
C = np.array([[1], [2], [3]])

print(A.shape, B.shape, C.shape)
# Output:
(3,) (1, 3) (3, 1)
```

1.1.2 Initialisering

Det er flere måter å initialisere et NumPy-array på. Hva man velger er helt avhengig av hva man trenger det til. Fra Pythons egne lister er man kanskje vant til å initialisere en tom liste og legge til verdier etter hvert. Som hovedregel bør dette unngås når man jobber med NumPy-array, man ønsker heller å initialisere arrayet i den størrelsen den skal ha til slutt. Dette vil ofte kreve litt ekstra kode, men kjører mye raskere. Under følger noen eksempler på initialisering som gjør bruk av NumPys innebygde funksjoner.

```
""" np.array("listelignende objekt")
Initalisering ved konvertering av "listelignende" objekter
   fra python """
A = np.array([1, 2, 3])
B = np.array((1, 2, 3))
C = np.array({1, 2, 3})

""" np.arange(start = 0, stop, step=1)
Initialiserer et NumPy-array med spesifisert steglengde """
D = np.arange(3)
E = np.arange(0, 3)
F = np.arange(0, 3, 0.1)

""" np.linspace(start, stop, num=50)
Initialiserer et NumPy-array med spesifert antall steg """
G = np.linspace(0, 3)
H = np.linspace(0, 3)
I = np.linspace(0, 3, 10)

""" np.zeros(shape, dtype=float)
Initialiserer et NumPy-array med spesifisert shape hvor
alle elementer er null (np.ones() fungerer tilsvarende hvor
elementer har verdi 1) """
J = np.zeros(9)
K = np.zeros(9, float)
L = np.zeros((3, 3), float)

""" np.zeros(shape, verdi, dtype=None)
Initialiserer et NumPy-array med spesifisert shape og verdi
   """
M = np.full((3, 3), 3)
N = np.full((3, 3), 3, float)

""" np.eye(n, indexdiagonal=0, dtype=float)
Initialiserer en nxn identitetsmatrise """
O = np.eye(2)
P = np.eye(3, dtype=int)
```

np.arange() vs np.linspace()

I `np.linspace()` vil stop-argumentet initialiseres som siste element i NumPy-arrayet. Dette er ikke tilfellet ved bruk av `np.arange()`, og er ofte en utslagsgivende faktor for hva man velger.

1.1.3 Indeksering

NumPy-arrays kan aksesseres på samme måte som Pythons lister, men også på en mer kompakt måte. Illustreres best med et kort eksempel:

```
A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(A[1][1]) #Som for Pythons lister
print(A[1, 1]) #Kompakt form
#Output
5
5
```

Slicing og negativ indeksering fungerer på samme måte som for Python-lister.

1.2 Lineær Algebra i NumPy

En av hovedgrunnene til å bruke NumPy er at man enkelt kan bruke NumPy-arrays som matriser. NumPy kommer med utrolig mye innebygd funksjonalitet, og ofte kan man finne enkle, pene løsninger på problemer med litt googling. I denne seksjonen gås det igjennom noen viktige konsepter som er mye brukt i dette faget, men det anbefales å kikke litt i dokumentasjonen for å få et overblikk over hva som er mulig.

1.2.1 linalg.norm()

Flere av funksjonene brukes i forbindelse med lineær algebra finnes i NumPys linalgbibliotek. Vi tenker videre i teksten at vi har importert det på følgende vis, `import numpy.linalg as la`. I dette kurset brukes ofte normen til en matrise som en skranke på hvor nøyaktig man ønsker at simuleringen skal være. Da brukes funksjonen `la.norm(array, order=None, axis=None)`. I TMA4320 brukes flere ulike normer. `la.norm()` gir Frobeniusnormen hvis ikke man spesifiserer noe annet. Ønskes andre normer må det spesifiseres i order-parameteren (se dokumentasjon).

1.2.2 linalg.solve()

Å kunne løse lineære ligningssett gjøres veldig enkelt i NumPy med funksjonen `np.la.solve(A, b)`, som returnerer løsningen som et NumPy-array.

```
A = np.array([[3, 1], [1, 2]])
b = np.array([9, 8])
print(la.solve(A, b))
#Output
[2. 3.]
```

1.2.3 Matrisearitmetikk

NumPy er designet slik at det skal være enkelt og effektivt å gjøre matematiske operasjoner med store matriser. I motsetning til listene vi er kjente med fra Python, fungerer operatorene i stor grad slik man vil for matriser. Under følger et eksempel hvor de viktigste operatorene og funksjonene gås igjennom.

```
import numpy as np

A, B = np.array([[3, 2], [0, 1]]), np.array([[3, 1], [1, 2]])

#Elementvis multiplikasjon
print(A * B)
#Output
[[9 2], [0, 2]]
#Matrisemultipliskasjon
print(A @ B, np.matmul(A, B)) #Ekvivalente uttrykk
#Output
[[11 7], [1 2]] [[11 7], [1 2]]

a, b = np.array([2, 3]), np.array([4, 1])
#Skalarprodukt
print(np.dot(a, b))
#Output
11
```

Tips!

Skal man gjøre en spesiell operasjon på en matrise er sannsynligheten stor for at det finnes en innebygd funksjon i NumPy som kan gjøre det for deg. Gjør et kjapt søk og bruk det du finner for det det er verdt, men sørg for at du forstår dokumentasjonen godt før du går i gang, så spares du for mye hodebry.

1.2.4 np.concatenate()

Ofte når vi løser differensialligninger numerisk ønsker vi å lagre verdiene underveis slik at vi kan plotte dem i etterkant. Dette er vi vant til fra ITGK når vi brukte `append()` på våre vanlige Pythonlister. Når man trenger å lagre flere verdier fungerer `np.concatenate()` på lignende vis, dog noe mer å holde styr på. Spesielt viktig blir nå shape-egenskapen og hvilken form man ønsker seg til slutt.

```
""" np.concatenate((a,b), axis=0) """
coords = np.array([[0, 0], [1, 0]])
c1 = np.array([[3, 1]])

print(np.concatenate((coords, c1)))
#Output
[[0 0], [1 0], [3, 1]]
print(np.concatenate((coords, c1.T), axis=1)) #c1.T
    transponerer vektoren
#Output
[[0 0 3], [1 0 1]]
```

2 Pyplot

Hånd i hånd med NumPy går Matplotlib.pyplot. I TMA4320 skal man vise mye grafisk, og god kjennskap til pyplot gjør ting mye enklere. Fra tidligere fysikkurs bør en ha grunnleggende kjennskap til biblioteket, så det grunnleggende dekkes her kun med et enkelt eksempel.

Tips!

Merk også at dette er et enormt stort bibliotek, med flere ulike måter å gjøre ting på. Man kan utforme plottene på detaljnivå akkurat slik man ønsker dem, men det blir ofte komplisert og hardkodet. Husk at grafen skal få frem matematikken, ikke dine pyplot-skills.

Videre går vi ut ifra at `import matplotlib.pyplot as plt` er kjørt.

2.1 Grunnleggende eksempel

Kjapt eksempel på et helt enkelt plot hvor man plotter en funksjon av en variabel. Legg spesielt merke til hvordan LaTex kan brukes som tekst i pyplot.

```
x = np.arange(10) #Initialiserer x-verdier
plt.plot(x, x**2, label=r'$f(x) = x^2$') #Plotter f(x)=x^2
plt.title("Grunnleggende eksempel") #Viser tittel
plt.xlabel("x") #Navngir x-aksen
plt.ylabel("y") #Navngir y-aksen
plt.legend() #Viser labels
plt.grid() #Viser rutenettet
plt.show() #Plotter
```

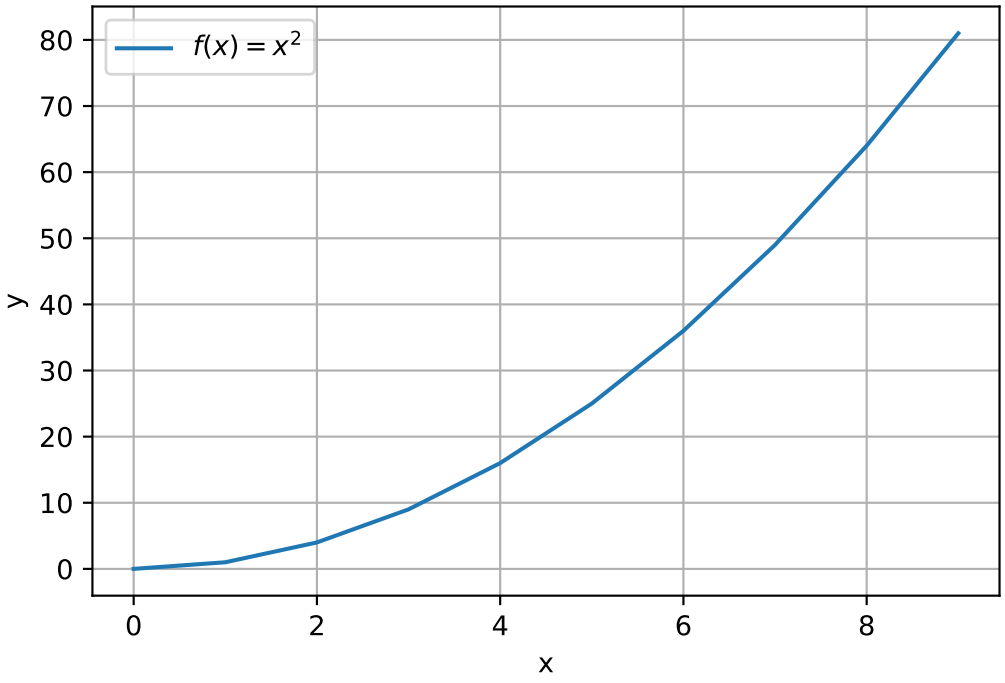

Figure 1: Enkelt plot

2.2 Subplot/Axes

Ofte vil man vise det samme plottet flere ganger, men litt ulike verdier på parameterene. Da er det ofte naturlig og ryddig å benytte seg av subplots hvor man kan ha flere plot i samme figur. Denne funksjonaliteten er implementert ved et objektorientert klassehierarki hvor man har en figur øverst med flere underplot av typen axes. Om man velger å bruke pyplot på den objektorienterte måten er helt opp til en selv, man klarer seg helt fint uten i dette kurset, men merk at eksempler man finner på nett ofte benytter seg av denne metoden, da den på sett og vis er ryddigere.

Under følger et eksempel på subplot hvor man ikke bruker axes.

```
x = np.linspace(-np.pi, np.pi, 100)
plt.figure() #Initialiserer en ny figur
#Initialiserer et subplot med 2 rader, 1 kolonne og index 1
plt.subplot(2, 1, 1)
plt.plot(x, np.sin(x), label=r'$sin(x)$') #Plotter sinus av x
plt.legend()
plt.grid()
plt.subplot(2, 1, 2) #Initialiserer subplot med index 2
plt.plot(x, np.cos(x), label=r'$cos(x)$') #Plotter cosinus av
    x
plt.legend()
plt.grid()
```

Samme eksempel, men med axes.

```
x = np.linspace(-np.pi, np.pi, 100)
fig = plt.figure() #Initialiserer en ny figur fig
#Initialiserer subplot (2, 1, 1) ax1
ax1 = fig.add_subplot(2, 1, 1)
ax1.plot(x, np.sin(x), label=r'$sin(x)$') #Plotter sinus av x
ax1.legend()
ax1.grid()
#Initialiserer subplot (2, 1, 2) ax2
ax2 = fig.add_subplot(2, 1, 2)
ax2.plot(x, np.cos(x), label=r'$cos(x)$') #Plotter cosinus av
    x
ax2.legend()
ax2.grid()
```

I eksempelet har vi to subplots, men man kan ha så mange man selv ønsker. Det er også mulig å endre størrelsesforholdet mellom dem, eller ha det ene inne i det andre. Det finnes også flere måter å initialisere subplots på enn de to som er demonstrert over, så det er lett å bli litt forvirret. Det anbefales å finne sin måte, og lære den godt.

2.3 Ulike typer

Når man plotter i TMA4320 brukes som oftest funksjonen `plt.plot()`, typisk når man vil vise funksjoner grafisk. Pyplot inneholder også noen andre funksjoner som kan komme godt med på prosjektene. Her nevnes `plt.hist()` og `plt.scatter()`, men det finnes også flere.

2.3.1 plt.hist()

Histogram brukes når man ska vise antall forekomster grafisk. Kan minne om et godt gammeldags stolpediagram, men med for mange ulike "kategorier" til at man gidder å navngi alle manuelt. Typisk har man et array

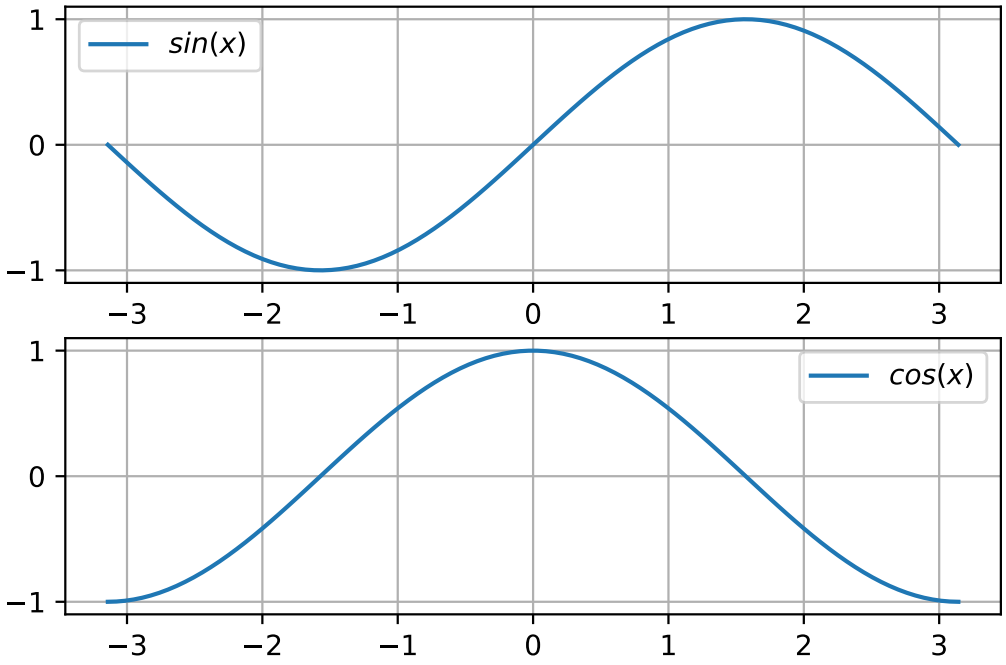

Figure 2: Subplot med to rader og en kolonne

med forekomster man deler plasserer i ulike "bins". Hvor mange bins man vil ha kommer helt an på dataene, og man leker seg ofte litt frem til det ser bra ut. Under følger et eksempel som viser bruk av `plt.hist()` på en standardnormalfordeling.

```
#Initialiserer normalfordelte verdier
x = np.random.normal(size=1000)
""" plt.hist(x, bins=10, density=False, cumulative=False, ec=
    None, alpha=1) """
plt.subplot(2, 2, 1)
plt.hist(x) #Standardhistogrammet
""" ec="black" gir skille mellom bins """
plt.subplot(2, 2, 2)
plt.hist(x, bins=30, ec="black")
""" density=True gir oss sannsynlighetsfordeling, alpha=0.6
    gir en mer gjennomsiktig farge """
plt.subplot(2, 2, 3)
plt.hist(x, bins=30, density=True, ec="black", alpha=0.6)
""" cumulative gir kumulativ fordeling, kwargs fungerer likt
    her som for plt.plot() """
plt.subplot(2, 2, 4)
plt.hist(x, bins=30, density=True, cumulative=True, ec="black
    ", alpha=0.6, label=r'$\copyright$', color='g')
```

```
plt.grid()
plt.legend()
```

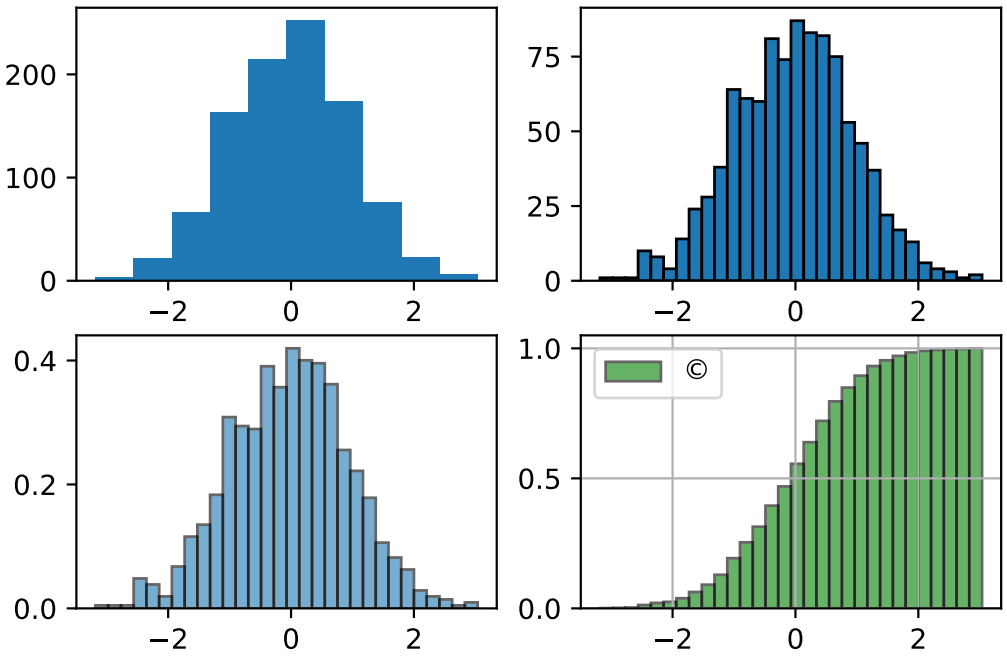

Figure 3: Ulike histogrammer av samme data

2.3.2 plt.scatter()

Har man punkter i to dimensjoner kan det av og til være nyttig å visualisere dem ved bruk av `plt.scatter()`. Denne fungerer ganske likt som `plt.plot()`, men kan tilegne andre egenskaper til hvert eneste punkt. For eksempel kan man ha flere punkter i planet med ulik størrelse, farge, transparenthet, osv. Hva som avgjør disse egenskapene avhenger igjen av dataene, og det er opp til fantasien og kreativiteten hvordan det blir seende ut! Under følger et eksempel på hvordan `plt.scatter()` kan brukes.

```
x = np.random.rand(50) #Tilfeldige x-verdier
y = np.random.rand(50) #Tilfeldige y-verdier
colors = np.random.rand(50) #Tilfeldige farger
sizes = 1000 * np.random.rand(50) #Tilfeldige storrelser

plt.scatter(x, y, c=colors, s=sizes, ec="black", alpha=0.4)
plt.colorbar() #Hviser hvilke farger som representerer hvilke
    tall
```

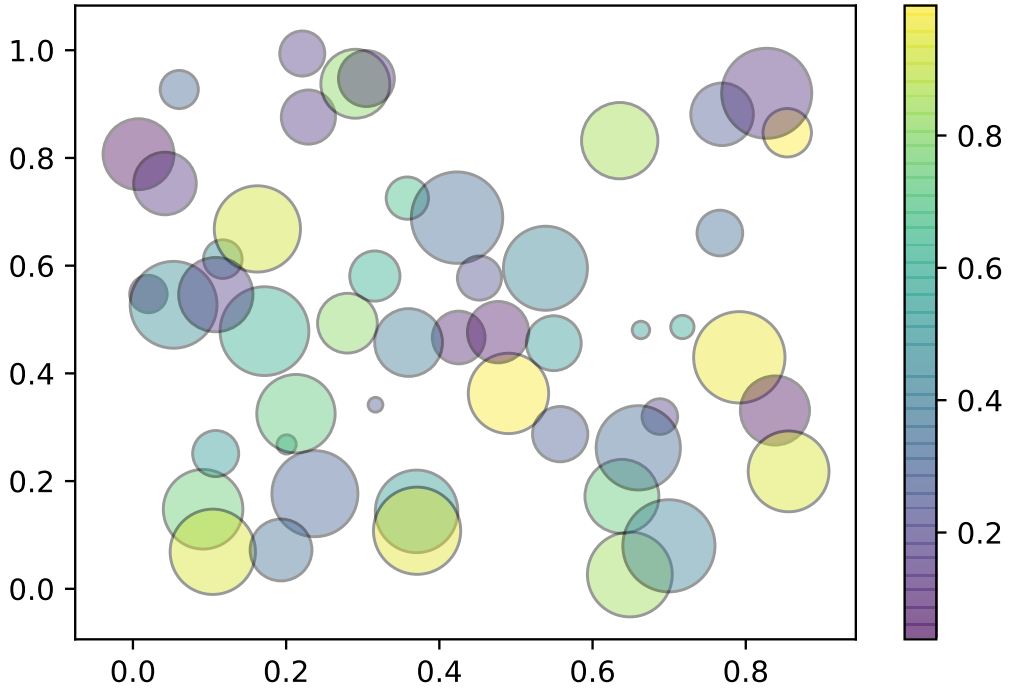

Figure 4: Scatterplot

2.4 Logaritmiske akser

Pyplot gir oss også muligheten til å velge logaritmiske akser over de vanlige lineære. Dette kan ofte være veldig nyttig når man vil sammenligne funksjoner eller verdier av ulik orden. Skal man for eksempel vise hvordan feilen i numeriske metoder endrer seg med steglengden, kan logaritmiske akser komme til nytte. Man kan velge om begge aksene skal være logaritmiske eller kun den ene. Under følger en grafisk sammenligning av to funksjoner. Hvilke viser best hvordan funksjonene oppfører seg?

```
""" Initialiserer x-verdier mellom 10^0 og 10^4 """
x = np.logspace(0, 4)
""" Definerer funkjsonene vi vil sammenligne """
def f(x):
    return x

def g(x):
    return x**3

plt.subplot(2, 1, 1)
""" Plotter med lineare akser """
plt.plot(x, f(x), label="f")
```

```
plt.plot(x, g(x), label="g")
plt.grid()
plt.legend()
plt.subplot(2, 1, 2)
""" Plotter med logaritmiske akser """
plt.plot(x, f(x), label="f")
plt.plot(x, g(x), label="g")
plt.yscale("log")
plt.xscale("log")
plt.grid()
plt.legend()
```

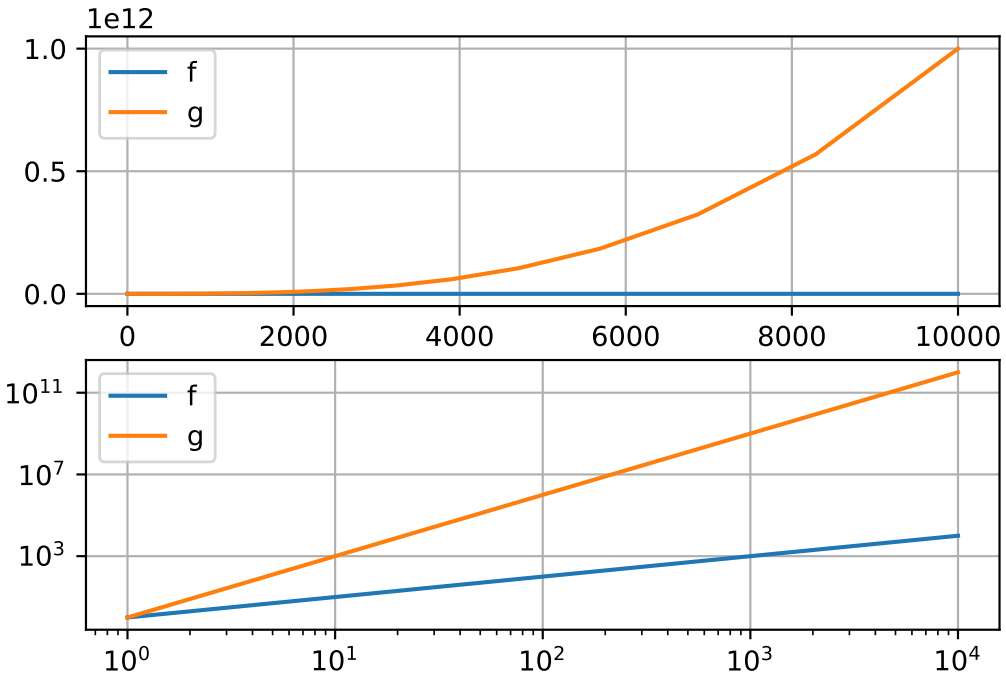

Figure 5: Lineære vs logaritmiske akser

2.5 VitBer-spesifikke tips

Insert innledning

2.5.1 Stoppkriterium

I de iterative ligningsløserene man lærer i dette faget er man nødt til å fastsette når man sier seg fornøyd med svaret. Dette gjør man ved et stoppkriterium hvor man velger seg en toleranse og itererer så lenge feilen er større

enn toleransen. Man setter også ofte en øvre grense på antall iterasjoner slik at selv om stoppkriteriet ikke nås vil funksjonen terminere. Alt ettersom hva man synes passer kan man bruke absolute error, relative error eller hybrid error. Under følger et eksempel på fikspunktiterasjon hvor man tar i bruk stoppkriterium.

```
# Absolute error: np.abs(x - x_prev) > tol
# Relative error: np.abs(x - x_prev)/np.abs(x) >
# Hybrid error: np.abs(x - x_prev)/np.maximum(np.abs(x),
    theta) > tol
def fikspunktiterasjon(g, x0, tol):
    error = 1
    k = 1
    while error > tol and k < 100:
        x = g(x0)
        error = np.abs(x - x0) #Absolute error
        x0 = x
        k += 1
    return x
```