Programmering for kjemikere

Kompendium i IN-KJM1900
Høst 2019

Andreas Haraldsrud

Copyright © 2019 Andreas Haraldsrud

A.D.HARALDSRUD@KJEMI.UIO.NO

Kompendiet er under utarbeiding. Ta gjerne kontakt for ris og ros, feil og mangler, ideer og innspill.

Innhold

1 Programmering i kjemi **4**
1.1 Programmering som verktøy 4
1.2 Algoritmer 5
1.3 Modellering 5

2 Behandling av eksperimentelle data **7**
2.1 Plotting 7
2.1.1 Plotting av lister 7
2.1.2 Plotting av funksjoner 11
2.2 Lesing og plotting av datafiler med *numpy* 13
2.3 Utviding av datasett 16
2.3.1 Regresjon 18
2.4 Statistikk 21
2.4.1 Gjennomsnitt 21
2.4.2 Spredning av målepunkter 22
2.4.3 Datatilpasning 24
2.4.4 Grafisk framstilling av statistisk spredning 24

3 Tolking av eksperimentelle data **27**
3.1 Numerisk derivasjon 27
3.1.1 Newtons kvotient 28
3.1.2 Feilanalyse 29
3.1.3 Numerisk derivasjon av diskrete data 30
3.2 Løsing av likninger 33
3.2.1 Newtons metode 33

4 Simuleringer **36**
4.1 Differensiallikninger 36
4.1.1 Metode 1: Forward Euler 37
4.1.2 Metode 2: Backward Euler 41
4.1.3 Metode 3: Runge-Kutta 4 45
4.1.4 Oppsummering 46

4.2 Stokastiske simuleringer 48
4.2.1 Tilfeldige bevegelser 50
4.3 Oppsummering 54

Kapittel 1

Programmering i kjemi

Kjemi er et eksperimentelt fag. Vi samler inn og analyserer data, utvikler modeller for hvordan disse dataene henger sammen, og strukturerer modellene så de skal henge sammen med eller erstatte allerede eksisterende teori. Strukturering og generalisering av informasjon kan ses på som selve definisjonen av *kunnskap*, og kunnskapen om stoffers reaksjoner og egenskaper er kjemiens domene.

Kjemi blir også kalt "den sentrale vitenskapen" fordi den tar opp i seg elementer fra fysikk, biologi, geologi og andre realfaglige disipliner. Hvis du er interessert i litt av hvert, er kjemi noe for deg. Kjemi er realfagenes godtepose der alle finner en favoritt. Men det gjør også kjemi krevende, for en må ha nokså god oversikt for å kunne forstå det store bildet. Her kan programmering hjelpe oss litt på vei.

Programmering som verktøy er i utgangspunktet domeneuavhengig. Mange av de metodene vi bruker, er felles for alle naturvitenskapelige disipliner. Dette kan bidra til god tverrfaglig forståelse dersom programmering blir brukt på en god måte. Programmering er en måte å utforske, eksperimentere og tenke kritisk på.

1.1 Programmering som verktøy

Vi skal se på hvordan programmering kan hjelpe oss med å beskrive sentrale problemstillinger i kjemi. Hovedpoenget er altså ikke programmeringen i seg selv, men programmering som et verktøy for å forstå og jobbe med kjemifaglige temaer på en god måte. Noen sentrale temaer fra kjemi som programmering og simuleringer kan hjelpe oss med, er:

1. Støkiometri og statistikk – Hvordan kan vi tolke eksperimentelle data på en fornuftig måte, og hva slags grad av troverdighet har disse dataene?

2. Bindinger og reaksjoner – Hvilke krefter styrer ulike typer bindinger, og hvordan brytes og bindes de slik at nye forbindelser dannes?

3. Stoffers egenskaper – Hvilke egenskaper har stoffer som nye materialer og legemidler?

4. Bevegelse av partikler – Hvordan foregår flyten av partikler og energi gjennom et system?

5. Biokjemiske prosesser – Hvordan regulerer proteiner transport gjennom cellemembranen? Hvordan fungerer viktige enzymer og hormoner i kroppen?

Programmering kan gi oss ny innsikt i både fundamentale fenomener og nye anvendelser. Vi kan både gjøre mer og forstå mer dersom vi bruker programmering på en god måte. Det betyr ikke at programmering alltid er den beste løsningen til alle problemer, men at det er et uvurderlig verktøy i kjemikerens verktøykasse. Det er viktig å lære seg når programmering er nyttig å bruke. Selv om du har fått en ny fin drill som gjør mange oppgaver enklere, bør du kanskje ikke bruke denne drillen neste gang du skal slå i stykker en gipsvegg.

1.2 Algoritmer

Definisjon 1.2.1: Algoritmer

En algoritme er en presis beskrivelse av en serie operasjoner som skal utføres for å oppnå et visst resultat.

Velkjente eksempler på algoritmer er strikkeoppskrifter, kakeoppskrifter og algoritmene som gir oss anbefalte filmer på Netflix og annonser på Facebook. Algoritmer i kjemi er på sin side en serie operasjoner som løser et kjemisk problem. Noen algoritmer er svært kompliserte, mens andre er enkle å forstå. Python har mange biblioteker som kan importeres for å få tilgang til tusenvis av nyttige algoritmer.

Men selv om det finnes ferdigproduserte algoritmer, må vi også lære oss å lage dem selv. Dette gjør vi av flere årsaker. For det første forstår vi algoritmene bedre hvis vi lager dem selv. For det andre har vi muligheten til å legge til modifikasjoner dersom vi skulle trenge det. Desto mer du forstår, desto mer kontroll har du på hva som skjer. Sist, men ikke minst, kan det å lage spesifikke algoritmer gi enda bedre innsikt i det kjemifaglige innholdet. Alle disse er gode grunner til at vi skal programmere algoritmene fra bunnen av, men når du først har lært algoritmene, skal vi også se på importering av algoritmer fra biblioteker.

1.3 Modellering

Definisjon 1.3.1: Modellering

Modellering er en prosess for å finne en forenklet representasjon av et fenomen i virkeligheten, altså en modell.

Noen ganger kan kjemi virke som en restaurant der du blir servert alt for mange retter, i form av formler. Den og den formelen beskriver det og det, og din oppgave er å kombinere

formlene på en måte som gjør at du får riktig svar. Men dette er ikke essensen av kjemi. Det er selvfølgelig en nødvendig del av kjemien å kunne bruke, tolke og forstå matematiske sammenhenger mellom kjemiske størrelser. Men det er også viktig å forstå at dette bare er modeller som stort sett er et produkt av systematiske undersøkelser, og som derfor er basert på et sett med forenklinger og antakelser.

Hver modell har sine styrker og svakheter. Kun virkeligheten er virkeligheten i seg selv. Modeller er "bare" forenklinger, men dette er egentlig en nødvendighet for å kunne systematisere virkeligheten. Det finnes ulik grad av forenklinger i modeller, og selv om de ikke er eksakte avbildninger av virkeligheten, er de ikke nødvendigvis *feil*.

Ta for eksempel atommodeller. En av de enkleste atommodellene vi har, er Bohrs atommodell. Selv om det kan se ut som om den prøver å fortelle oss hvordan et atom ser ut, sier den oss noe helt annet. Den sier noe om energinivåene til elektronene, ikke om plassering og bevegelse. Elektroner går ikke i bane rundt atomkjernen. Modellen har begrensninger, men det har alle andre modeller også. Schrödingers likning og orbitalmodellen sier mer om hvor et elektron kan befinne seg og hvilken energi den kan ha, men den er mer kompleks og ikke alltid så enkel å ha med å gjøre. I visse tilfeller, for eksempel hvis vi skal beregne spektrallinjene til hydrogen, holder det med Bohrs atommodell. For å forstå bindingsforholdene i enkle stoffer som vann og metan, kan vi bruke VSEPR-modellen (Valence shell electron pair repulsion theory), men den er utilstrekkelig for å forstå bindingsforhold i aromatiske forbindelser som Benzen. Da må vi bruke *molekylorbitaler*. Ingen av modellene er altså direkte feil, de har bare sine begrensninger, noe som gjør at de kan forutsi enkelte fenomener, men ikke andre.

For hver enkelt modell er det altså viktig å være oppmerksom på begrensningene og forutsetningene som gjelder. Dette er enklere å bli bevisst på når en lager og/eller utforsker modellene selv, og dette er lettere å få til med programmering. Programmering er et viktig verktøy for å forstå og utforske kjemi, og jeg håper at du etter hvert kommer til å sette pris på det å bruke programmering til både rutineprosedyrer og kreativ utforsking i kjemi.

Kapittel 2

Behandling av eksperimentelle data

En viktig del av kjemi som et eksperimentelt fag er å kunne representere og behandle data på en hensiktsmessig måte. I dette kapitlet skal vi se på følgende innen behandling av eksperimentelle data:

1. Plotting av data.
2. Avlesning av data fra filer.
3. Interpolasjon: Tilpasse nye punkter til datasettet.
4. Regresjon: Tilpasse en funksjon til punktene.
5. Statistikk.

2.1 Plotting

2.1.1 Plotting av lister

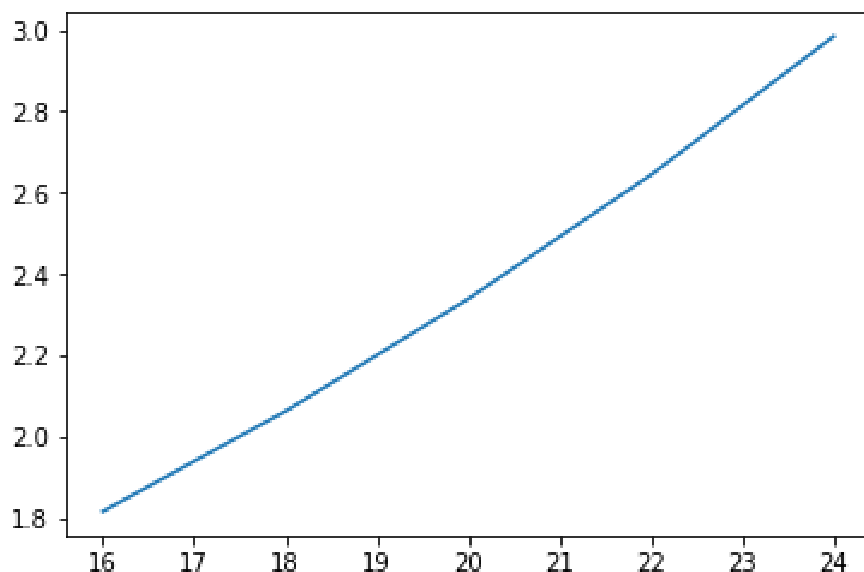
Vi kan plotte både kontinuerlige funksjoner og diskrete data med biblioteket *matplotlib*. Det mest interessante for kjemikere og andre som driver med eksperimentelle fag, er plotting av *diskrete data*. Først ser vi derfor på et enkelt eksempel der vi definererer dataene i lister. Denne framgangsmåten egner seg godt hvis du skriver inn datapunkter fra eksperimenter manuelt når det ikke er for mange punkter. La oss ta et eksempel der vi har målt damptrykket til vann i et glassrør ved ulike temperaturer. Vi har fått følgende data:

Temperatur (°C)	**Damptrykk (kPa)**
16	1.817
18	2.063
20	2.339
22	2.644
24	2.984

Vi har her få målepunkter, så dette kan vi enkelt legge direkte inn i lister og plotte.

```
import matplotlib.pyplot as plt
import numpy as np

T = [16, 18, 20, 22, 24] # Temperatur i grader celsius
trykk = [1.817, 2.063, 2.339, 2.644, 2.984] # Damptrykk i kPa

plt.plot(T,trykk) # Plotter y som funksjon av x
plt.show() # Viser plottet
```

Dette gir følgende plott:

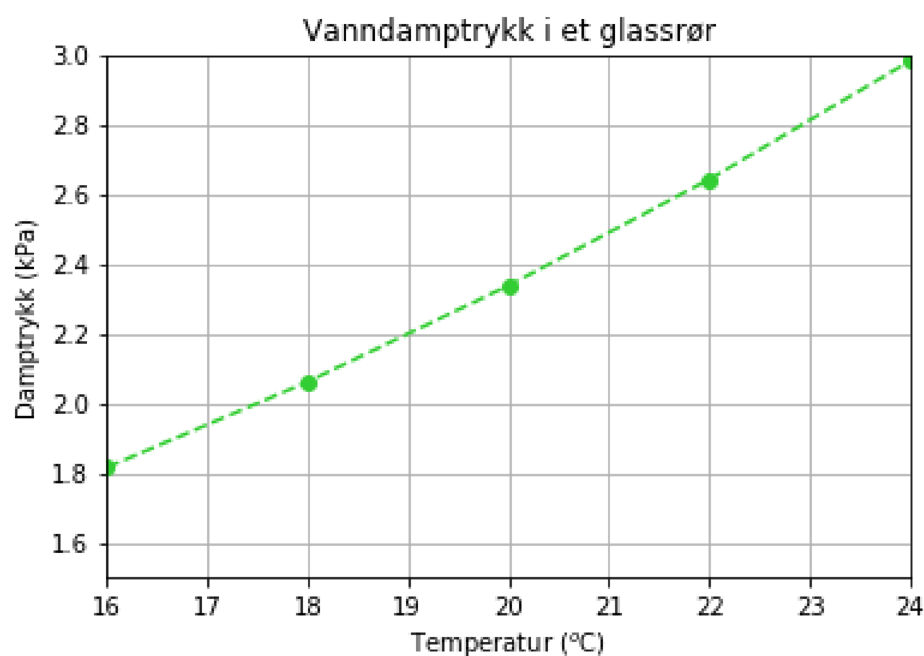


Figur 2.1.1: Enkel figur for damptrykk av vann ved ulike temperaturer.

Hvis vi har lyst til å modifisere og pynte på plottet vårt, har vi mange muligheter til det. Her er noen forslag til en del nyttige modifikasjoner i programmet ovenfor.

```
plt.plot(T,trykk,color='limegreen',marker='o',linestyle='--')
plt.title('Vanndamptrykk i et glassrør')  # Tittel på plottet
plt.xlabel('Temperatur ($^o$C)')          # Aksetittel på x-aksen
plt.ylabel('Damptrykk (kPa)') # Aksetittel på y-aksen
plt.xlim(16,24) # Definisjonsmengde
plt.ylim(1.5,3) # Verdimengde
plt.grid()  # Slår på rutenett
plt.show()
```

Dette gir følgende modifiserte plott:



Figur 2.1.2: Modifisert figur med damptrykk av vann ved ulike temperaturer.

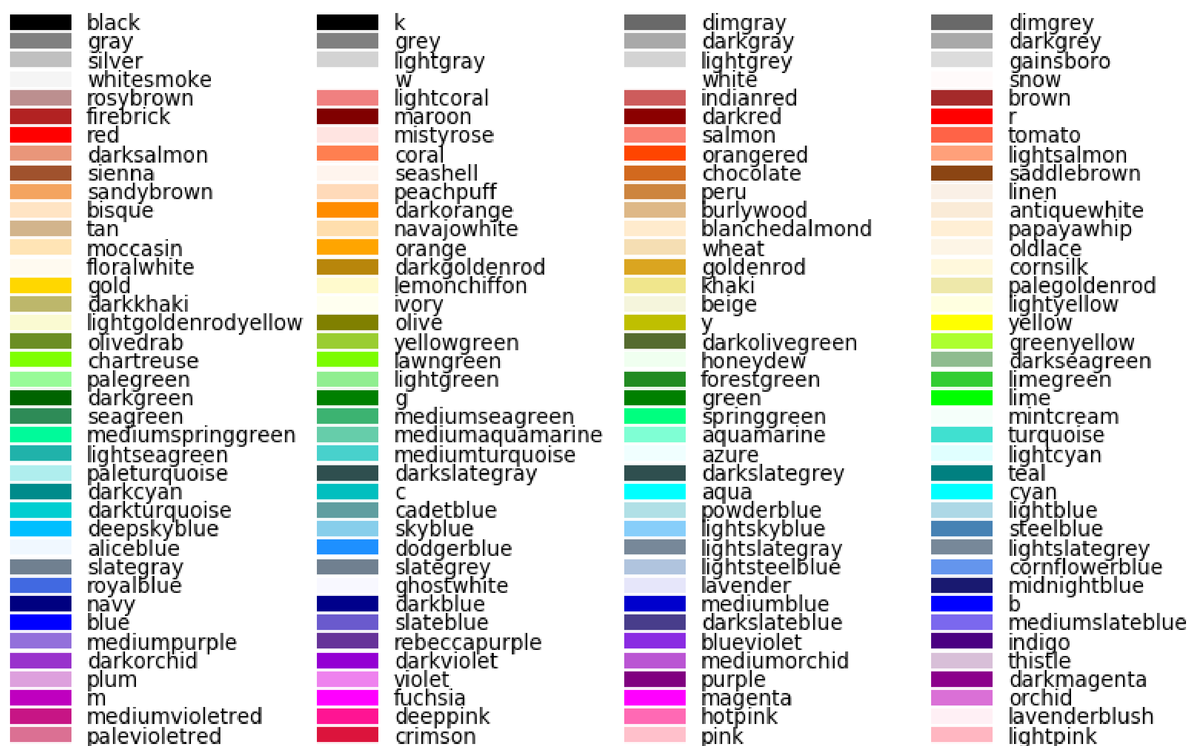
`Plot`-funksjonen kan ta mange ulike parametere, blant annet farge, linjestil og markører, slik som ovenfor. Her er noen eksempler på markører:

Markør	**Forklaring**
'.'	punkt
'o'	sirkel
' '(mellomrom)	ingen markør
'^'	triangel opp
'v'	triangel ned
's'	firkant
'p'	pentagon

Det finnes mange flere markører som en kan søke opp. Alle disse markørene kan også kombineres med ulike linjestiler:

Linjestil	**Forklaring**
'-'	heltrukket linje
'--'	stipla linje (lange)
':'	stipla linje (korte)
'-.'	stipla linje (annenhver lang/kort)
' ' (mellomrom)	ingen linje

I tillegg til alle disse markør- og linjestilene finnes det mange flotte farger en kan pynte grafene med:



Figur 2.1.3: En har også mye kunstnerisk frihet i Python!

Nå har vi plotta data som ikke består av så mange målepunkter ved å legge dem manuelt inn i lister. Vi skal se på hvordan vi kan lese av større mengder data fra filer slik at vi ikke trenger å skrive inn dataene i lister manuelt. Men først ser vi på hvordan vi kan plotte kontinuerlige funksjoner.

Underveisoppgave 2.1.1

Vi har en ideell gass i en beholder. Lag et plott av trykk i kPa som funksjon av temperatur i K ved å bruke følgende data:

trykk = [36, 46.4, 56.7, 67.1, 77.5, 88.0]
temp = [173, 223, 273, 323, 373, 423]

2.1.2 Plotting av funksjoner

Når vi skal plotte kontinuerlige funksjoner, trenger vi et sett med x-verdier. Disse kan vi lage med ei enkel løkke, eller vi kan generere dem med funksjonen `linspace` fra numpy-biblioteket. Et eksempel på dette er:

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(-2, 3, 1000)
y = 2*x**2 - 2*x + 1

plt.plot(x,y)
plt.xlabel('x')
plt.ylabel('y')
plt.grid()
plt.show()
```

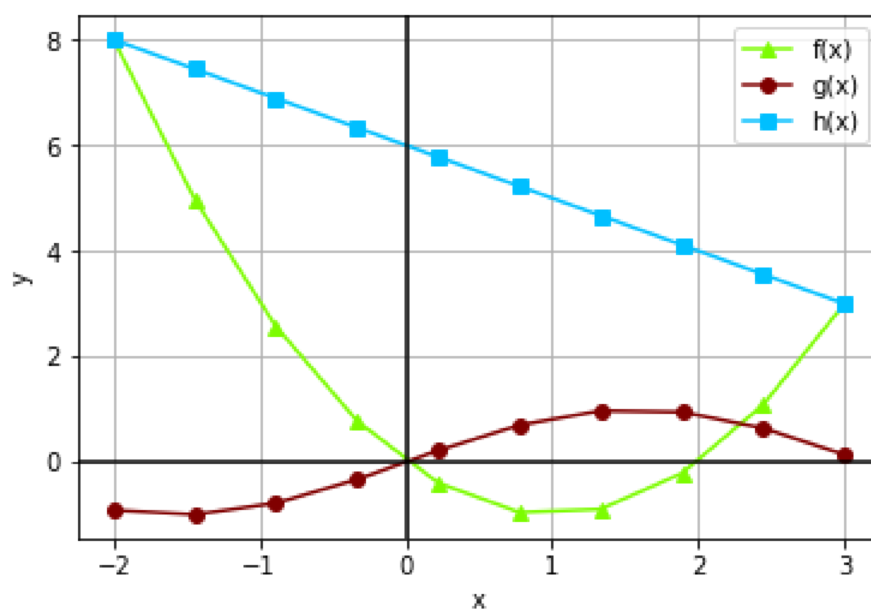
Funksjonen `linspace(a, b, n)` genererer n punkter med lik avstand mellom a og b. Dette gir en glatt og fin graf, men det er verdt å merke seg at den faktisk ikke er helt glatt. Siden vi har å gjøre med diskrete punkter, får vi aldri helt glatte grafer. Dette kan du se hvis du f.eks. prøver å generere grafen ovenfor med 5 punkter (prøv det!). Mellom hvert punkt brukes nemlig *lineær interpolasjon*, som du skal se på seinere i dette kapitlet. Det betyr at det trekkes ei rett linje mellom hvert punkt slik at vi kan tilnærme punkter som ikke finnes i datasettet vårt med punkter på linjene mellom punktene vi allerede har.

Vi kan også ha bruk for å plotte flere funksjoner eller flere datapunkter i samme koordinatsystem. Da kan vi bare kalle på plott-funksjonen flere ganger. For å skille mellom plottene, kan vi bruke merkelapper med funksjonen `legend` som henter opp alle merkelapper (`label`) gitt som parameter til *plot*-funksjonen:

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(-2, 3, 10)

def f(x):
    return x**2 - 2*x

def g(x):
    return np.sin(x)

def h(x):
    return - x + 6

y1 = f(x)
y2 = g(x)
y3 = h(x)
```

```
plt.plot(x,y1,color='lawngreen',label='f(x)', marker='^')
plt.plot(x,y2,color='maroon',label='g(x)', marker='o')
plt.plot(x,y3,color='deepskyblue',label='h(x)', marker='s')
plt.legend() # Viser merkelappene
plt.xlabel('x')
plt.ylabel('y')
plt.axhline(y=0,color='black') # Tegner x-akse
plt.axvline(x=0,color='black') # Tegner y-akse
plt.grid()
plt.show()
```

Dette fungerer også godt med datafiler med ulike data som du ønsker å sammenlikne i samme koordinatsystem. Programmet ovenfor genererer følgende plott:



Figur 2.1.4: Plotting av flere funksjoner.

Underveisoppgave 2.1.2

Plott tre av dine favorittfunksjoner i samme koordinatsystem. Tilpass akser og tittel og pynt på plottet.

2.2 Lesing og plotting av datafiler med *numpy*

Når vi gjør forsøk på laboratoriet, bruker vi ofte automatisk måleutstyr som sensorer og andre instrumenter som gir oss hundre- og kanskje tusenvis av målepunkter. Det er lite effektivt å skrive disse dataene inn i lister manuelt. Heldigvis har vi funksjoner i Python som kan lese av datafiler slik at vi kan behandle store mengder data på en fornuftig måte.

Vi kan lese filer med innebygde funksjoner i Python. Dette gir god programmeringstrening, men i praksis er dette tungvint når vi har å gjøre med lettleste datasett. Derfor skal vi bruke hensiktsmessige biblioteker til å gjøre en del av arbeidet for oss. Vi har brukt *numpy*-biblioteket mye, og her finnes en veldig grei funksjon som heter *loadtxt*.

Det er kun råtekstfiler, altså filer uten formatering, som kan leses inn i Python med `loadtxt`. Dette er fordi formaterte filer, som for eksempel Word- eller Pages-filer, inneholder mye kode som forstyrrer dataene vi er ute etter. Noen vanlige råtekstfiler har filtype `.txt`, `.dat` og `.csv` (*comma separated values*). I mange tilfeller kan også `.xlsx` (Excel-filer) leses direkte.

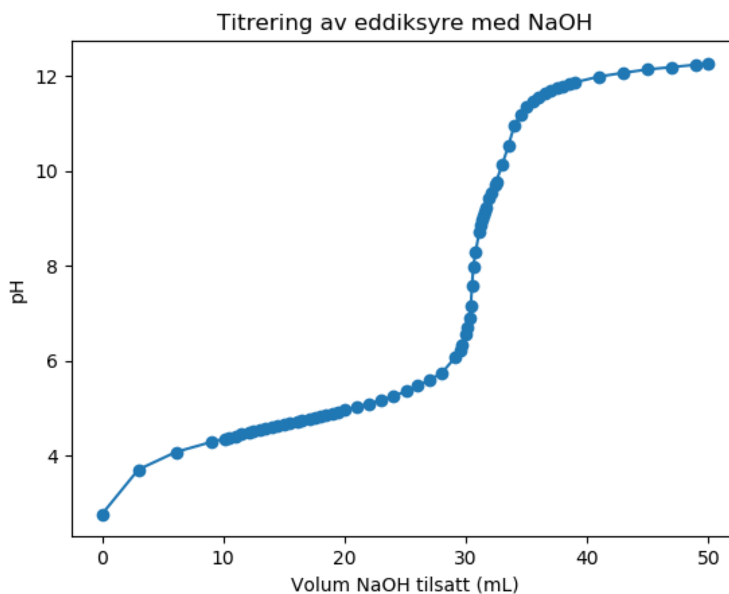
La oss se på et eksempel der vi har gjennomført en titrering av 25 mL 0.12 M eddiksyre med 0.10 M lut (NaOH). Dataene blir lagret i fila `titrering_eddiksyre_NaOH.txt` med komma mellom verdiene. Her er et utsnitt av datafila:

Volum NaOH (mL)	**pH**
0.0,	2.77
3.0,	3.71
6.1,	4.08
9.0,	4.29
10.1	4.36

Vi kan lage et program som leser av og plotter disse dataene slik:

```
import numpy as np
import matplotlib.pyplot as plt

data = np.loadtxt('titrering_eddiksyre_NaOH.txt', delimiter = ',', skiprows=1)
volum = data[:,0] # Volum NaOH tilsatt i mL
pH = data[:,1] # pH i løsningen

plt.plot(volum, pH, marker='o')
plt.title('Titrering av eddiksyre med NaOH')
plt.xlabel('Volum NaOH tilsatt (mL)')
plt.ylabel('pH')
plt.show()
```

Dette gir følgende plott:



Figur 2.2.1: Titrering av eddiksyre med lut.

Funksjonen `loadtxt` tar filnavnet som parameter, i tillegg til en del tilleggsparametre. De nyttigste parametrene er oppsummert i tabellen nedenfor:

Parameter	**Forklaring**
delimiter	skilletegn mellom dataene
skiprows	antall rader som skal hoppes over
usecols	tuppel med kolonner som skal brukes
dtype	datatype (float, int, str osv.)

Hvis datatypene du skal bruke, er av ulik type, kan du f.eks. lese alle som strenger, for så å konvertere alle til de datatypene du trenger.

Det `loadtxt` gjør, er å lese inn dataene og lagre dem i en array (her i variabelen *data*) der hvert arrayelement er en rad i datasettet. Et utsnitt av output hvis vi undersøker dette med `print(data)` er som følger:

```
[[ 0.     2.77]
 [ 3.     3.71]
 [ 6.1    4.08]
 [ 9.     4.29]
 [10.1    4.36]
 ...
 [49.    12.24]
 [50.    12.26]]
```

Når vi da for eksempel skriver `volum = data[:,0]`, henter vi ut alle rader (markert med kolon), men kun kolonne 0 (den første kolonnen i datasettet), som jo tilsvarer volumet. Tilsvarende henter vi da ut alle rader i kolonne 1 (den andre kolonnen i datasettet) ved å skrive `data[:,1]`.

2.3 Utviding av datasett

Når vi samler inn eksperimentelle data, får vi diskrete datapunkter og ikke kontinuerlige funksjoner. Noen ganger mangler vi dermed punkter på kritiske steder i datasettet. Disse punktene kan en i visse tilfeller skaffe ved å gjøre eksperimentet på nytt, men vi kan også tilnærme dem matematisk. Denne prosessen kaller vi *interpolasjon*.

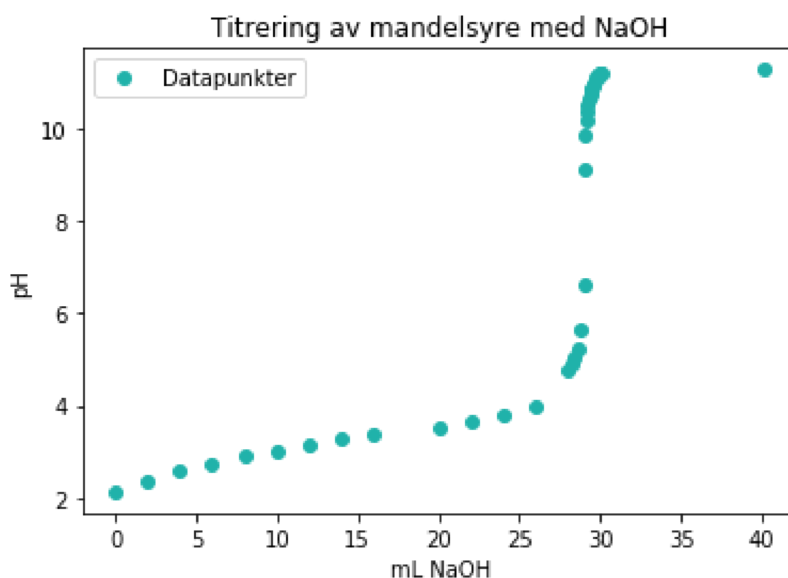
Definisjon 2.3.1: Interpolasjon

Interpolasjon er en prosess der vi finner tilnærminger til datapunkter mellom kjente datapunkter.

Både interpolasjon og *regresjon* tilnærmer datapunkter med funksjoner. Det er likevel en viktig forskjell mellom disse teknikkene. Regresjon avviker ofte fra en del av de kjente verdiene for å være en best mulig tilpasset *alle* dataene. Interpolasjon passer *eksakt* i datapunktene, men gir derfor ofte ikke en glatt funksjon gjennom alle punkter.

Siden vi har å gjøre med diskrete data, må vi tilnærme datapunktene med en funksjon når vi skal interpolere. Det vil si at vi tilnærmer de nye punktene med utgangspunkt i at dataene passer best med for eksempel et andregradspolynom eller en sinusfunksjon gjennom de kjente punktene. Vi skal se på polynominterpolasjon her fordi polynomer av n-te grad er gode tilnærminger til de fleste datasett.

La oss ta et eksempel der vi har utført en titrering av mandelsyre med lut (NaOH). Ved ekvivalenspunktet endrer pH-en seg svært mye. Det er dermed vanskelig å få datapunkter som gir enkel avlesing av pH-verdien ved ekvivalenspunktet.



Figur 2.3.1: Titreringskurve – pH-en endrer seg kraftig ved ekvivalenspunktet.

Vi kan bruke flere ulike metoder for å finne pH ved ekvivalenspunktet. Én av metodene skal vi se på i neste kapittel. Her skal vi nøye oss med å bruke funksjonen `interp1d` for å utføre interpolasjon. Du kan derimot gjerne prøve å lage dine egne algoritmer for dette – det er god trening!

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d

# Lesing av fil
data = np.loadtxt('tirering_mandelsyre_NaOH.txt', delimiter = ',', skiprows = 1)
volum = data[:,0] # Volum tilsatt NaOH i mL
pH = data[:,1]

# Interpolasjon
f = interp1d(volum, pH, kind = 'linear')
xnew = np.linspace(27,30,100)
y = f(xnew)

# Plotting
plt.plot(volum,pH,marker='o',linestyle=' ',label='Datapunkter', color =
    'lightseagreen')
plt.title('Titrering av mandelsyre med NaOH')
plt.xlabel('mL NaOH')
plt.ylabel('pH')
plt.plot(xnew, y, label = 'Lineær interpolasjon',color ='mediumorchid')
plt.legend()
plt.show()
```

Vi definerer altså en ny funksjon f som inneholder alle de datapunktene vi hadde, men også med en rett linje mellom hver av disse datapunktene. Vi bruker funksjonen *interp1d* til dette. Den tar en x-array og en y-array som første parametre, og deretter skriver vi inn hva slags interpolasjon vi ønsker. Her har vi valgt `kind = linear`, altså lineær interpolasjon, men vi kan også velge for eksempel `quadratic` og `cubic`.

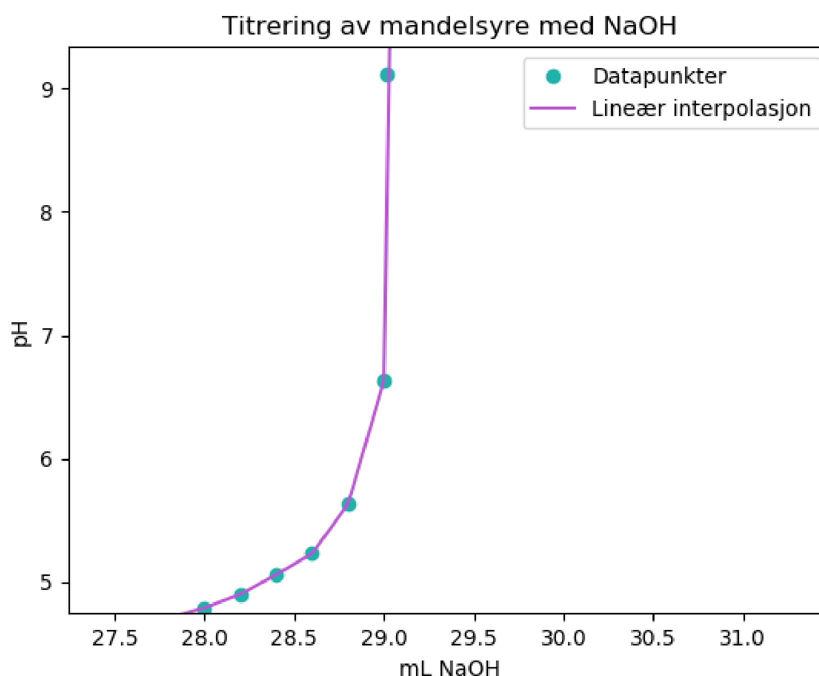
Deretter lager vi en del nye x-punkter som vi bruker som argument i den nye funksjonen vår for å generere et sett med y-verdier. Vi har kun valgt det området på titrerkurven som er interessant, og vi har nøyd oss med 100 punkter i området.

Underveisoppgave 2.3.1

Prøv ut lineær, kvadratisk og kubisk interpolasjon på datasettet ovenfor eller et annet datasett. Tegn grafene i samme plott og marker grafene med merkelapper. Sammenlikn og kommenter resultatene.

Et plott generert av programmet vårt ovenfor, er vist nedenfor. Vi har forstørret området som er interessant. Nå har vi flere datapunkter slik at vi enklere og mer presist kan finne

pH-en ved ekvivalenspunktet.



Figur 2.3.2: Interpolerte data på titrerkurven.

2.3.1 Regresjon

Interpolerte funksjoner går alltid igjennom datapunktene våre. Hvis vi heller vi ha en glatt kurve som gir gjennomsnittlig best mulig tilpasning til alle punktene våre, må vi bruke *regresjon*.

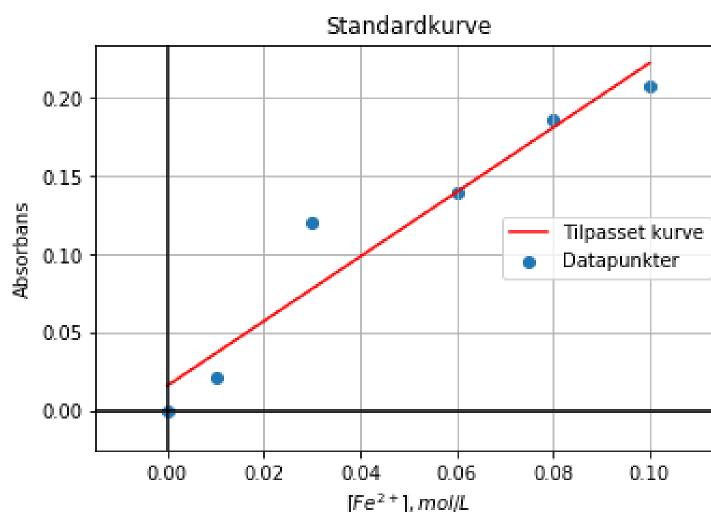
Definisjon 2.3.2: Regresjon

Regresjon er en prosess der vi tilnærmer diskrete data med en kontinuerlig funksjon

La oss ta et eksempel der vi har brukt et spektrometer til å måle absorbansen til ulike standardløsninger av Fe^{2+}. Vi har gjort dette fordi vi har en løsning med ukjent innhold av jernioner. Da trenger vi å gjøre en lineær regresjon for å kunne avgjøre hvilken konsentrasjon absorbansen til den ukjente prøva tilsvarer. Programmet nedenfor lager en standardkurve/kalibreringskurve med funksjonen *polyfit* fra numpy-biblioteket. Vi bruker også plottefunksjonen `scatter` for å slippe linjer mellom datapunktene.

```
import numpy as np
import matplotlib.pyplot as plt

jern = np.array([0,0.010,0.030,0.060,0.080,0.10])
absorbans = np. array([0, 0.021, 0.120, 0.139, 0.186, 0.208])

reg = np.polyfit(jern, absorbans, 1)
y = (reg[0])*jern + reg[1] # uttrykk på formen y = ax + b, der x = jern

plt.scatter(jern,absorbans,label='Datapunkter')
plt.plot(jern, y, color = 'red',label='Tilpasset kurve')
plt.legend()
plt.title('Standardkurve')
plt.xlabel('$[Fe^{2+}], mol/L$')
plt.ylabel('Absorbans')
plt.axhline(y=0,color='black')
plt.axvline(x=0,color='black')
plt.grid()
plt.show()
```

Funksjonen `polyfit` utfører polynomregresjon på dataene våre, og tar som argument x- og y-verdiene i datasettet, og deretter graden av polynomet. Her har vi brukt grad 1 for lineær regresjon. Dette gir en array med koeffisientene a og b for uttrykket $y = ax + b$. Tilsvarende vil en andregradsregresjon gi koeffisientene for $ax^2 + bx + c$ og så videre. Vi kan dermed bruke disse koeffisientene videre til å lage nye y-verdier basert på de opprinnelige x-verdiene. Dette er gjort i linja `y = (reg[0])*jern + reg[1]`. Vi får følgende plott:



Figur 2.3.3: Regresjon av måledata.

Vi ser at én av målingene ligger litt utenfor, så den burde vi kanskje gjort på nytt. Slike betraktninger, der vi bruker modellering til å tilnærme et problem, for så å endre på eksperimentet for å få bedre samsvar med modellen, er en viktig del av programmeringens rolle i kjemifaget. Vi har også mer kvantitative mål på hvor godt regresjonen passer til dataene, som vi skal se på i delkapittel 2.4.

La oss si at den ukjente prøva med jernioner ga absorbans på 0.160. Vi kan bruke regresjonskurven vår til å finne hva slags konsentrasjon dette tilsvarer på følgende måte (vær sikker på at forstår hvorfor):

```python
x = (0.160 - reg[1])/reg[0]
print("Den ukjente konsentrasjonen er:", x, "mol/L")
```

Når vi har utført regresjon, kan vi også forutsi datapunkter som er utenfor datapunktene våre. Dette kaller vi *ekstrapolering*. Ved å ekstrapolere kan vi forutsi hvordan et system *har vært* eller *kommer til å bli*. En skal derimot være svært forsiktig med å trekke slutninger basert på ekstrapolering! Det kan likevel være en god *indikasjon* trender og utviklinger.

Underveisoppgave 2.3.2

I oppgave 2.1.1 lagde du et plott av trykk i kPa som funksjon av temperatur i K ved å bruke følgende data:

trykk = [36, 46.4, 56.7, 67.1, 77.5, 88.0]
temp = [173, 223, 273, 323, 373, 423]
Benytt regresjon til å estimere trykket ved 0 K. Kommenter svaret.

2.4 Statistikk

Statistikk handler om samling, organisering og analyse av eksperimentelle data. Her skal vi se på følgende statistiske operasjoner og konsepter:

1. Gjennomsnitt.
2. Spredning: Varians og standardavvik.
3. Korrelasjon.

For alle operasjonene skal vi ta utgangspunkt i følgende eksempel: En kald høstdag sitter vi med en varm kopp te foran peisen. Vi funderer på hva koffeininnholdet i denne teen kan være, og går derfor til laben for å undersøke dette med væskekromatografi. Vi pipetterer ut noen mL fra tekoppen vår, filtrerer teen og fortynner løsningen. Deretter bruker vi en væskekromatograf (HPLC) for å finne konsentrasjonen. Vi er tålmodige og nøye, og gjør derfor ti gjentakelser av forsøket:

Injeksjon	**Konsentrasjon (mg/mL)**
1	245
2	272
3	252
4	264
5	261
6	272
7	255
8	260
9	268
10	259

Vi bruker disse dataene og innebygde statistikk-funksjoner i numpy-biblioteket i noen av eksemplene i dette delkapitlet. Du oppfordres derimot underveis til å lage *egne* Python-funksjoner som gjør det samme som numpy-funksjonene.

2.4.1 Gjennomsnitt

Gjennomsnittet av målingene er summen av alle målingene delt på antallet målinger. En mer formell definisjon er slik:

Definisjon 2.4.1: Gjennomsnitt

Gjennomsnittet $\bar{x}$ av n verdier $x_1, x_2, \ldots, x_n$ kan uttrykkes slik:

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{2.4.1}$$

Vi kan lage en funksjon som regner ut gjennomsnittet av måledataene ovenfor slik:

```
koffein = [245, 272, 252, 264, 261, 272, 255, 260, 268, 259]

def gjennomsnitt(data):
    summen = 0
    for element in data: # Kan også bruke funksjonen sum() direkte her
        summen += element
    snitt = summen/len(data)
    return snitt

snitt = gjennomsnitt(koffein)
print("Gjennomsnitt:", snitt, "mg/mL")
```

Vi kan dobbelsjekke at vi har fått riktig med numpy-funksjonen `mean`:

```
import numpy as np

snitt = np.mean(koffein)
print("Gjennomsnitt fra numpy:", snitt, "mg/mL")
```

Begge metoder gir et gjennomsnitt på 260.8 mg/mL, så vi kan være fornøyde med implementeringen vår.

2.4.2 Spredning av målepunkter

For å kunne måle hvor stor spredning det er i datasettet vårt, kan vi for eksempel bruke *varians* eller *standardavvik*. Data som er svært like hverandre kan ha samme gjennomsnitt som data som er svært ulike, og et mål på spredning er derfor nødvendig for å si noe om *presisjonen* i måledataene.

Legg merke til at både varians og standardavvik er noe mindre intuitivt definert enn gjennomsnitt. Husk derimot at disse bare er ulike måter å måle spredning på som viser seg å være nyttige. Vi starter med en definisjon av varians:

Definisjon 2.4.2: Varians

Varians er et mål på variasjonen i et datasett og skrives ofte som σ^2. Det er definert som summen av kvadratet av differansen mellom alle målepunktet og gjennomsnittet av målepunktene, delt på antall målepunkter. Det vil si at varians er gjennomsnittet av kvadratet av differansen til måleverdiene.

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2 \tag{2.4.2}$$

Underveisoppgave 2.4.1

Regn ut variansen til datasettet ovenfor for hånd. Kontroller så ved å lage *din egen* Python-funksjon som regner ut variansen til en array eller liste.

Årsaken til at vi tar kvadratet av differansen mellom en måleverdi og gjennomsnittet når vi skal regne ut variansen, er at vi ønsker en positiv verdi. Ulempen er at vi ikke får samme enhet som måledataene – vi får enheten kvadrert. I eksempelet med koffeinanalysen får vi derfor mg^2/mL^2, som er en noe diffus enhet å jobbe med. Dette kan vi løse ved å ta kvadratet av variansen. Da får vi et annet mål på spredning, nemlig standardavviket.

Definisjon 2.4.3: Standardavvik

Standardavvik er et mål på spredningen i et datasett, og defineres som den positive kvadratrota av variansen.

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2} \tag{2.4.3}$$

Underveisoppgave 2.4.2

Regn ut standardavviket til koffeindatasettet for hånd. Kontroller så ved å lage *din egen* Python-funksjon som regner ut variansen til en array eller liste.

Siden standardavviket har samme enhet som måledataene, bruker vi ofte dette som mål på spredning til fordel for varians. Vi kan bruke numpy-biblioteket til å regne ut varians og standardavvik slik:

```
import numpy as np
koffein = [245, 272, 252, 264, 261, 272, 255, 260, 268, 259]

snitt = np.mean(koffein)
varians = np.var(koffein)
standardavvik = np.std(koffein)

print("Gjennomsnitt fra numpy:", snitt, "mg/mL")
print("Variansen fra numpy:", round(varians, 1), "mg/mL")
print("Standardavviket fra numpy:", round(standardavvik,1), "mg/mL")
```

Merk at vi også har brukt funksjonen `round` til å runde av verdiene til ett desimal (`round(verdi, antall desimaler)`).

En viktig betraktning når en gjør statistikk på eksperimentelle data, spesielt med små mengder data, får vi ofte litt underestimering av varians og standardavvik. Dette kan vi kompensere for ved å dele på $N - 1$ istedenfor bare N.

Underveisoppgave 2.4.3

Modifiser varians- og standardavvikfunksjonene dine til å dele på $N-1$ istedenfor N. Sammenlikn de to tilnærmingene.

2.4.3 Datatilpasning

I slutten av delkapittel 2.3 brukte vi regresjon til å tilnærme en funksjon til datapunktene vi hadde for å finne en ukjent konsentrasjonen av jernioner i en løsning. Vi så derimot at et av punktene lå et stykke utenfor linja vi fikk. Det finnes flere måter å sjekke slike "uteliggere" på, men her skal vi nøye oss med å undersøke hvorvidt linja blei en god tilnærming til dataene våre til tross for dette punktet. Dette kan vi gjøre ved å undersøke *determinasjonskoeffisienten* R^2 mellom linja og de eksperimentelle verdiene.

R^2 sier noe om hvor godt modellen vår, i dette eksempelet det lineære linjestykket, passer med de empiriske målingene. Det gir en statistisk måling av tilpasningsgraden til modellen, og har en verdi som vanligvis ligger mellom 0 og 1. Når R^2 er nær 1, passer modellen godt med dataene våre. Det kan være flere årsaker til at R^2 er lav, og det er viktig å huske på at det ikke i alle tilfeller betyr at modellen er dårlig! Motsatt kan R^2 være høy selv om modellen beskriver dårlig framtidig utvikling (ved ekstrapolering).

Vi kan definere R^2-verdien slik:

Definisjon 2.4.4: R^2-verdi

For N målepunkter x_i er R^2 gitt ved:

$$R^2 = 1 - \frac{\sum_{i=1}^{N}(x_i - f_i)^2)}{\sum_{i=1}^{N}(x_i - \bar{x})^2}$$

der f_i er predikert verdi fra modellen.

Vi kan nå importere et nytt bibliotek som er svært nyttig til blant annet maskinlæring og statistisk analyse: *scikit-learn*. For å finne R^2-verdien til regresjonen i 2.3.1, kan vi gjøre slik:

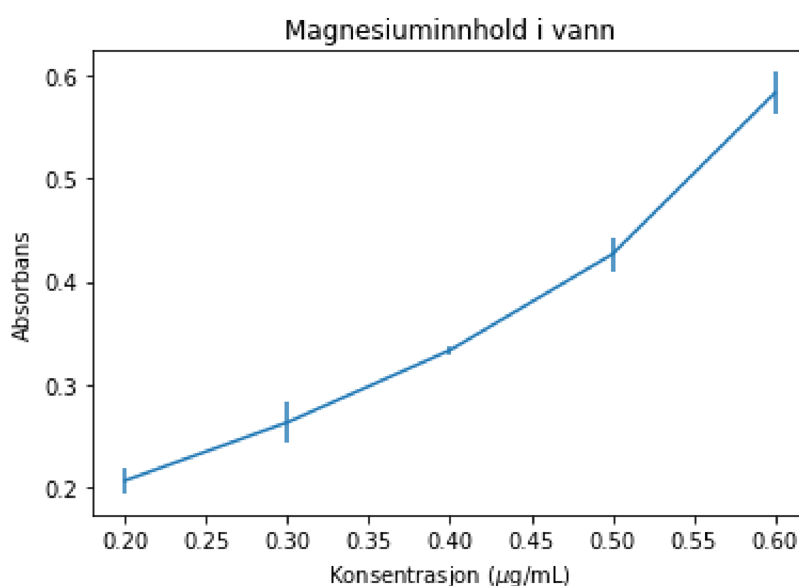
```
from sklearn.metrics import r2_score

R2 = r2_score(absorbans, y)
print(R2)
```

2.4.4 Grafisk framstilling av statistisk spredning

Det finnes mange måter å framstille statistisk spredning i datasett på. En vanlig framstilling er *usikkerhetsstolper*. Denne representerer gjennomsnittet av datapunktene med standardavviket markert på figuren. Dette sier noe om variasjonen som er observert i målingene.

Usikkerhetsstolper kan vi bruke når vi har gjort flere målinger på samme variabel. Vi kan ta som eksempel et eksperiment der vi skal konstruere en standardkurve for magnesiumkonsentrasjonen i en vannprøve. Vi bruker en serie på 0.2, 0.3, 0.4, 0.5 og 0.6 μg/mL Mg^{2+} som vi analyserer tre ganger hver med et flammeatomabsorpsjonsspektrofotometer (et nydelig ord!). Da har vi tre målinger for absorpsjon per konsentrasjon. Vi kan dermed plotte disse målingene og usikkerheten i disse målingene med usikkerhetsstolper:

```
import numpy as np
import matplotlib.pyplot as plt

konsentrasjon = [0.2, 0.3, 0.4, 0.5, 0.6]
absorbans = [[0.21, 0.22, 0.19], [0.26, 0.29, 0.24], [0.33, 0.33, 0.34], [0.41,
    0.42, 0.45], [0.56, 0.61, 0.58]]

standardavvik = []
snitt = []

for element in absorbans:
    snitt.append(np.mean(element))
    standardavvik.append(np.std(element))

plt.errorbar(konsentrasjon, snitt, standardavvik)
plt.title('Magnesiuminnhold i vann')
plt.xlabel('Konsentrasjon ($\mu$g/mL)')
plt.ylabel('Absorbans')
plt.show()
```



Figur 2.4.1: Usikkerhetsstolpeplott.

Vi bruker funksjonen `errorbar` som tar x-verdier, gjennomsnittet og standardavviket som parametre. I tillegg kan vi velge at vi vil vise øvre og nedre grenser med piler (setter `uplims` og `lolims` til `True`). Hvis vi ikke ønsker strekene mellom datapunktene i plottet ovenfor, kan vi legge inn argumentet `fmt='none'`.

Kapittel 3

Tolking av eksperimentelle data

Selv om eksperimentelle data kan framstilles grafisk og behandles statistisk, er det ofte vanskelig å tolke dataene og finne verdier for det vi ønsker å undersøke uten å bare lese av grafisk. Vi trenger derfor en systematisk måte å undersøke diskrete data på.

I klassisk *analytisk* matematikk kan vi for eksempel løse likninger ved hjelp av spesialformler (som andre- og tredjegradsformelen) og derivere uttrykk ved hjelp av derivasjonsregler. Hvis vi derimot skal operere med diskrete data, kan vi ikke bruke disse formlene og reglene. Da må vi bruke *numeriske metoder*. Numeriske metoder tar utgangspunkt i algortimer som gir tilnærmede løsninger, og benytttes der det er vanskelig eller umulig å få eksakte løsninger. Beregninger med diskrete data er kanskje hovedstyrken til numeriske metoder, men de fungerer også på kontinuerlige funksjoner.

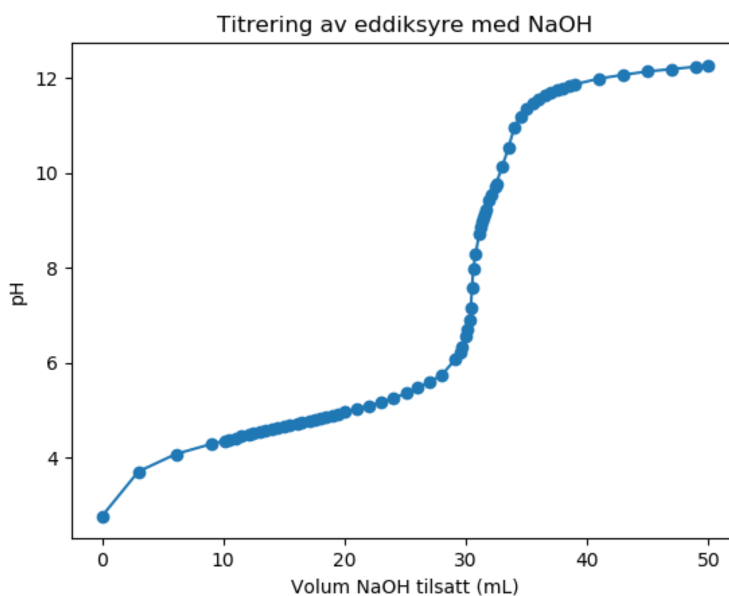
I dette kapitlet skal vi først se på hvordan vi kan bruke numerisk derivasjon til å beskrive endringen i eksperimentelle data. Deretter ser vi på hvordan en generell måte å løse likninger på kan brukes til å løse problemer i kjemi.

3.1 Numerisk derivasjon

Vi begynner med et eksempel som vi har sett på i kapittel 2.2. Der så vi på titrering av eddiksyre med NaOH. I en titrering er vi interessert i hvilket volum som tilsvarer ekvivalenspunktet, og dette kan vi prøve å tilnærme med numerisk matematikk.

Når vi skal løse slike problemer, er det lurt å få en oversikt over hvilke egenskaper grafen har. Disse egenskapene kan vi ofte beskrive med utgangspunkt i *endringen* mellom punkter. Ved hjelp av titrerkurven i 2.2.1 kan vi beskrive sentrale sider ved titreringsdataene slik:

1. Veksten er alltid positiv.
2. Veksten er størst i ekvivalenspunktet.
3. Veksten er svært liten i halvtitrerpunktet.
4. Ekvivalenspunktet er et vendepunkt.



Figur 3.1.1: Titrerkurve fra figur 2.2.1

Siden vi kan se på endring mellom to punkter som er så nær hverandre som mulig, har vi å gjøre med den *deriverte* i disse punktene. Og fordi poenget var å finne pH-en ved ekvivalenspunktet, bør vi utnytte observasjonen om at veksten, altså den deriverte, er størst i ekvivalenspunktet. Da trenger vi verktøy som lar oss derivere diskrete data.

3.1.1 Newtons kvotient

Den enkleste formen for numerisk derivasjon tar direkte utgangspunkt i definisjonen av den deriverte:

$$f'(x) = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x} \tag{3.1.1}$$

Vi kan tilnærme denne grensen der Δx går mot 0 med en svært liten Δx. Denne metoden kalles *Newtons kvotient*.

Numerisk metode 3.1.1: Newtons kvotient

For en liten verdi av Δx kan vi tilnærme den førstederiverte slik:

$$f'(x) \approx \frac{f(x + \Delta x) - f(x)}{\Delta x} \tag{3.1.2}$$

Kontinuerlige funksjoner kan vi enkelt derivere i ulike punkter ved å implementere definisjonen av den deriverte nesten direkte:

```
import numpy as np
import matplotlib.pyplot as plt

def derivert(f, x, delta_x = 1E-8):  # Definerer en funksjon som deriverer
   fder = (f(x+delta_x) - f(x))/delta_x # Newtons kvotient
   return fder

def f(x):                # Definerer en funksjon vi skal derivere
   return 3*x**3 + x**2 - 1

x = np.linspace(-2,2,100)
y = derivert(f,x)
y2 = f(x)

plt.plot(x,y)
plt.show()
```

Her *plotter* vi funksjonen, men vi kan også *regne* ut ulike funksjonsverdier. Husk derimot at vi ikke får et nytt uttrykk når vi deriverer numerisk, men diskrete *verdier*, selv om funksjonen vi deriverer er kontinuerlig.

Underveisoppgave 3.1.1

Deriver ulike matematiske funksjoner i Python og kontroller svaret ved å regne analytisk.

3.1.2 Feilanalyse

La oss nå ta en titt på hvilke verdier av Δx som gir best resultat. Det må vel være den verdien som ligger nærmest 0, altså en så liten verdi som mulig – eller? La oss teste dette ved å skrive ut den deriverte for ulike verdier av Δx:

```
def f(x):
   return 2*x**2 + x - 5

def derivert(f, x, delta_x):
   fder = (f(x+delta_x) - f(x))/delta_x
   return fder

delta_x = [10**-i for i in range(1,17)] # Listeløkke

for i in range(0, len(delta_x)):
    print("For delta_x =", delta_x[i],":",derivert(f,1,delta_x[i]))
```

Legg merke til hvordan Δx defineres ovenfor ved å bruke ei for-løkke som fyller ut verdier i lista – dette er en god måte å automatisere på ved å bruke for-løkker! Vi får disse resultatene:

```
For delta_x = 0.1 : 5.200000000000005
For delta_x = 0.01 : 5.020000000000024
For delta_x = 0.001 : 5.001999999999285
For delta_x = 0.0001 : 5.000199999996013
For delta_x = 1e-05 : 5.000020000034411
For delta_x = 1e-06 : 5.000001999988513
For delta_x = 1e-07 : 5.000000200539034
For delta_x = 1e-08 : 4.999999969612645
For delta_x = 1e-09 : 5.000000413701855
For delta_x = 1e-10 : 5.000000413701855
For delta_x = 1e-11 : 5.000000413701855
For delta_x = 1e-12 : 5.000444502911705
For delta_x = 1e-13 : 4.996003610813204
For delta_x = 1e-14 : 4.973799150320701
For delta_x = 1e-15 : 5.329070518200751
For delta_x = 1e-16 : 0.0
>>>
```

Vi ser at "store" verdier som 0.1 og 0.01 gir en del feil. Men vi ser også faktisk at nøyaktigheten er størst ved 10^{-8}, og at den synker både med økende og med minkende Δx. Og attpåtil gir 10^{-16} null som svar!

Vi forventer kanskje ikke dette resultatet. Dersom vi kun ser på definisjonen av den deriverte, er det ikke spesielt logisk at det skal slå slik ut. Men det hele handler om at tall ikke er representert eksakt i en datamaskin, og når datamaskinen skal operere med svært små tall, kan det bli en liten avrundingsfeil når den regner med tallene. Denne avrundingsfeilen gjør at vi får feil dersom vi velger for små verdier av Δx. Det viser seg dermed at 10^{-8} er en god verdi å velge i de fleste tilfeller.

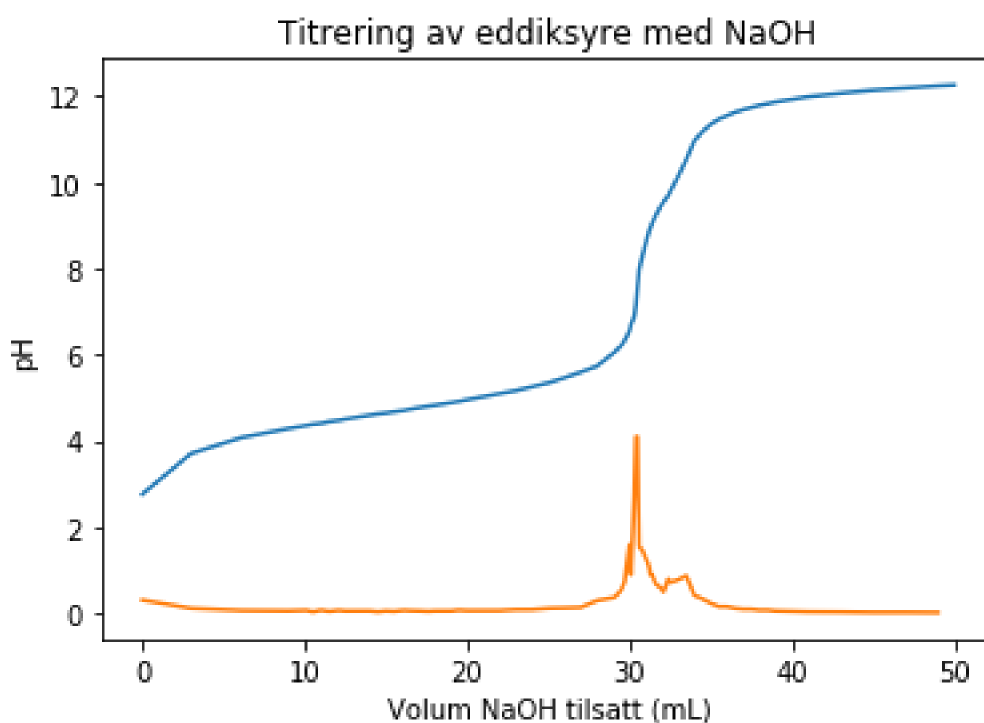
3.1.3 Numerisk derivasjon av diskrete data

Nå kommer vi til den nyttigste delen av numerisk derivasjon, nemlig derivasjon av diskrete data. Vi kan derivere på samme måte som vi gjorde med kontinuerlige funksjoner, men vi har gitt en Δx spesifisert av avstanden mellom datapunktene våre. Hvis målefrekvensen er lav, blir Δx høy, og motsatt. Vi kan bruke titreringsdataene våre fra titreringa av eddiksyre med NaOH (figur 3.1.1) som eksempel. Vi kan lese inn dataene og derivere dem slik:

```python
import numpy as np
import matplotlib.pyplot as plt

data = np.loadtxt('titrering_eddiksyre_NaOH.txt', delimiter = ',', skiprows=1)

volum = data[:,0] # Volum NaOH tilsatt i mL
pH = data[:,1] # pH i løsningen
```

```
def numder(x,y):
    derivert = []
    for i in range(0,len(y)-1):
        der = (y[i+1]-y[i])/(x[i+1]-x[i])
        derivert.append(der)
    return derivert

derivert = numder(volum,pH)

plt.plot(volum, pH)
plt.plot(volum[0:-1],derivert)
plt.title('Titrering av eddiksyre med NaOH')
plt.xlabel('Volum NaOH tilsatt (mL)')
plt.ylabel('pH')
plt.show()
```

Legg merke til at vi kun plotter opp til (men ikke med) siste x-verdi. Det er fordi den deriverte gjenspeiler differansen i verdiene, og da blir lista med deriverte verdier ett element mindre enn lista med verdiene som blir derivert. Plottet blir slik:



Figur 3.1.2: Titrerkurve og dens deriverte.

Vi ser at den egenskapen vi observerte med høyest derivert i ekvivalenspunktet stemmer

veldig godt. I tillegg ser vi at den deriverte ved halvtitrerpunktet er veldig liten, noe som også stemmer. Vi kan nå lage et program som finner den maksimale deriverte i lista og dermed volumet ved ekvivalenspunktet. Dette kan vi gjøre med kommandoen `index`:

```
import numpy as np

data = np.loadtxt('titrering_eddiksyre_NaOH.txt', delimiter = ',', skiprows=1)

volum = data[:,0] # Volum NaOH tilsatt i mL
pH = data[:,1] # pH i løsningen

def numder(x,y):
    derivert = []
    for i in range(0,len(y)-1):
        der = (y[i+1]-y[i])/(x[i+1]-x[i])
        derivert.append(der)
    return derivert

derivert = numder(volum,pH)
maksimum = derivert.index(np.max(derivert)) # Finner indeksen til maksderivert i
    lista
ekv_volum = volum[maksimum]

print("Ekvivalenspunktet ble nådd ved tilsats av", ekv_volum,"mL 0.10 M NaOH.")
```

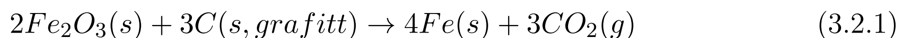
Hvis vi kjører programmet, får vi 30.4 mL, noe som stemmer godt med både grafen og hvis vi regner med utgangspunkt i konsentrasjonen til syra (0.12 M).

Underveisoppgave 3.1.2

Lag en egen funksjon som finner maksimum i lista ovenfor uten bruk av `max` fra numpy.

3.2 Løsing av likninger

I mange situasjoner ønsker vi å finne nullpunktene til en funksjon. Dette er det samme som å løse en likning $f(x) = 0$. La oss ta utgangspunkt i et uttrykk for Gibbs-energien til en reaksjon som funksjon av temperatur. Vi bruker følgende reaksjon som eksempel:



$$2Fe_2O_3(s) + 3C(s, grafitt) \rightarrow 4Fe(s) + 3CO_2(g) \tag{3.2.1}$$

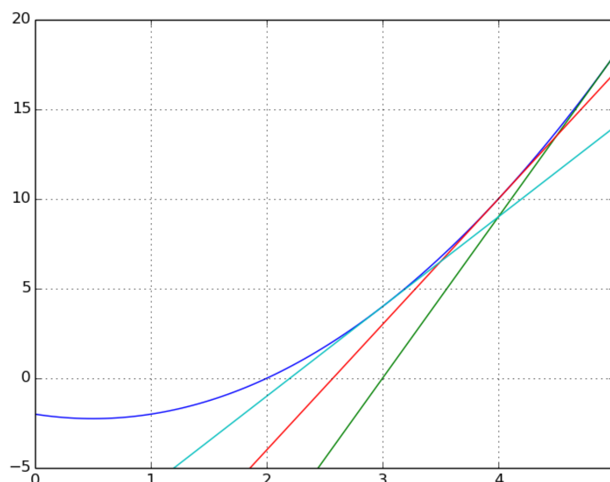
Gitt at vi har at $\Delta H = 467.9\ kJ/mol$ og $\Delta S = 0.5586\ J/(K \cdot mol)$, kan vi skrive uttrykket for Gibbs frie energi som funksjon av temperatur slik:

$$\Delta G = 467.9 - 0.5586 \cdot T \tag{3.2.2}$$

For at en reaksjon skal være spontan, må $\Delta G < 0$. Vi kan derfor sette likning 3.2.2 lik 0 for å finne ut ved hvilken temperatur reaksjonen er spontan. Dette er en enkel likning å løse for hånd, men eksempelet er ment å illustrere metoden så enkelt som mulig. Det er ofte lurt å teste metoder og algoritmer på enkle funksjoner som en lett kan regne på analytisk før en tester dem på mer krevende problemer. Metoden vi skal bruke, er robust og kan brukes på mange likninger, inkludert likninger som er uløselige for hånd. Ikke minst fungerer metoden godt på diskrete data.

3.2.1 Newtons metode

Det finnes mange måter å løse likninger på. En metode som ofte fungerer godt, er *Newtons metode*. Den bruker nullpunktet til *tangenten* i et punkt på en funksjon f som en tilnærming til nullpunktet til f. Her er metoden illustrert med funksjonen $f(x) = x^2 - x - 2$:



Figur 3.2.1: Newtons metode i tre trinn.

Vi plotter først tangenten til funksjonen gjennom $(5, f(5))$ (grønn farge). Vi ser at nullpunktet til denne ligger et stykke fra nullpunktet til funksjonen (mørkeblå farge). Da prøver vi oss med tangenten gjennom $(4, f(4))$ (rød farge), og vi ser at nullpunktet til denne nå er nærmere nullpunktet til funksjonen. Den lyseblå tangenten er tangenten gjennom $(3, f(3))$, og nå begynner vi virkelig å nærme oss. Reint grafisk er det nå antakelig ganske greit å se at tangentenes nullpunkt nærmer seg nullpunktet til funksjonen ettersom x blir mindre.

For å kunne bruke Newtons metode, trenger vi å gjette på det første punktet som vi skal lage den første tangenten gjennom. Dette kan være en kvalifisert gjetning eller du kan estimere det fra f.eks. en graf over dataene. Uansett vil metoden veldig ofte konvergere fort mot nullpunktet.

En ulempe med metoden er derimot at vi trenger et uttrykk for den deriverte for å finne tangenten. Den deriverte kan vi derimot stort sett lett finne ved enten analytisk eller numerisk derivasjon. Da kan vi bruke enkle metoder for derivasjon som Newtons kvotient.

Algoritmen for metoden kan formuleres slik:

Numerisk metode 3.2.1:

La f være en kontinuerlig, deriverbar funksjon, og la a være et punkt på funksjonen. Da kan nullpunktene finnes slik:

1. Finn et uttrykk for tangenten *T(x)* i *a*.

$$T(x) = f(a) + f'(a)(x - a) \tag{3.2.3}$$

2. Finn nullpunktet x_n til den n-te tangenten:

$$x_n = a - \frac{f(a)}{f'(a)} \tag{3.2.4}$$

3. Gjenta prosessen for $a = x_n$ fram til $n = N$.

Underveisoppgave 3.2.1

Vis at likning 3.2.4 stemmer i algoritmen ovenfor.

En begrensning ved metoden, er hvis en støter på et ekstremalpunkt. Da er $f'(a) = 0$, noe som gjør at vi deler på 0 i likning 3.2.4 ovenfor. Da feiler metoden. Det er også viktig å legge merke til at det kun er likning 3.2.4 vi bruker når vi implementerer Newtons metode. Likning 3.2.3 er bare et steg på veien til metoden.

La oss se på hvordan metoden kan brukes til å løse likning 3.2.2. Her kan vi også gjøre et triks som kan være lurt når vi skal bruke ulike funksjoner om igjen. Vi kan nemlig lagre de numeriske metodene våre i et separat program, for å så hente dem opp igjen i andre programmer. Nedenfor har vi lagra funksjonen `derivert` fra forrige delkapittel i en fil som

vi har kalt `numeriske_metoder.py`. Vi importerer den på samme måte som andre biblioteker. Det vi importerer må være i samme mappe som programmet vårt. Hvis ikke må vi spesifisere hvor den ligger.

```
from numeriske_metoder import derivert

def Gibbs(T):
    Delta_G = 467.9 - 0.5586*T
    return Delta_G

def newtons_metode(f, a, N = 1000, tol = 1E-10):
    i = 0
    while i <= N and abs(f(a)) >= tol:
        a = a - f(a)/derivert(f, a, tol)
        i += 1
    return a, i

nullpunkt, antall = newtons_metode(Gibbs, 500)

print("Reaksjonen er spontan ved", nullpunkt, "K. Metoden itererte", antall,
    "ganger.")
```

Underveisoppgave 3.2.2

Gjennomgå programmet ovenfor linje for linje og gjør to iterasjoner for hånd.

Programmet ovenfor kjører bare to ganger med denne toleransen og dette startpunktet, men det er fordi det er en lineær funksjon. Egentlig burde dette ført til at løkka kun kjørte én gang, men siden datamaskinen ikke representerer alle tall helt eksakt, må den kjøre en runde til for å oppnå den lave toleransen vi har satt.

Underveisoppgave 3.2.3

Lag noen datapunktet og prøv å implementere Newtons metode for diskrete data. Husk at vi da egentlig ikke har å gjøre med tangenter lenger, men heller *sekanter*.

Numerisk metode, en metode hvor en bruker en algoritme og numerisk approksimasjon til å beregne eller løse matematiske problemer

Kapittel 4

Simuleringer

Eksperimenter har stått i sentrum av kjemifaget i flere hundre år. Men med økende datakraft og billigere datamaskiner de siste 50 årene har *simuleringer* blitt et viktig tillegg til dette. Simuleringer kan belyse og supplere eksperimenter, i tillegg til at de kan illustrere viktige fenomener på egen hånd. Ved å studere molekyler og kjemiske reaksjoner med datamaskiner, kan vi få enda større innsikt i de mest fundamentale egenskapene og mekanismene til disse systemene.

Veldig ofte tar kjemiske simuleringer utgangspunkt i modeller fra kvantekjemi, men her skal vi se på simuleringer som har et mer eksperimentelt og klassisk kjemisk grunnlag. De fleste simuleringer tar utgangspunkt i at vi kjenner til endringen til en eller flere tilstander i systemet, og at vi dermed forutsier den nye tilstanden til systemet ut fra dette. Matematisk formuleres endringer i dynamiske systemer som *differensiallikninger*, altså likninger som inneholder den deriverte (endringen) til en funksjon,

4.1 Differensiallikninger

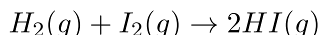
La oss si at vi har et uttrykk for endringen i et system, for eksempler endring i konsentrasjon (ratelikninger) eller endring i posisjon og hastighet (Newtons 2. lov). Disse uttrykkene beskriver den momentane endringen, det vil si den deriverte, som funksjon av tid. Altså kan de klassifiseres som *differensiallikninger*.

Definisjon 4.1.1: Differensiallikninger

En differensiallikning er en likning som inneholder den deriverte av en funksjon, $f'(x)$. I de fleste praktiske situasjoner beskriver slike likninger sammenhengen mellom endringen, $f'(t)$, og tilstanden, $f(t)$, til et system ved tida t.

La oss ta endring i konsentrasjon i reaksjonen mellom hydrogengass og jod i gassfase ved $400\ \circ$ C som et eksempel. Denne reaksjonen har en relativt enkel ratelov. Husk at formen på ratelovene ikke har noen direkte sammenheng med det støkiometriske forholdet mellom

reaktanter og produkter. De er derimot basert på empiriske observasjoner (eksperimenter). Reaksjonslikninga for reaksjonen er slik:



$$H_2(g) + I_2(g) \rightarrow 2HI(g)$$

Rateloven for denne reaksjonen med hensyn på konsentrasjonen av HI (her kalt c) er som følger:

$$c'(t) = k_r[H_2][I_2] \tag{4.1.1}$$

Vi har altså en likning som beskriver endringen i systemet $c'(t)$ (en *differensiallikning*), men ingen informasjon om selve konsentrasjonen $c(t)$. Vi ønsker med andre ord å finne konsentrasjonen av HI ved enhver tid gitt en eller annen startbetingelse (konsentrasjonen av produkter og reaktanter til å begynne med). Det er det samme som å si at vi ønsker å finne funksjonsverdien $c(t + dt)$ for hvert tidssteg dt. La oss se på en enkel metode for å regne ut dette

4.1.1 Metode 1: Forward Euler

Du kjenner faktisk allerede til et uttrykk som inneholder en funksjon og dens deriverte, nemlig definisjonen av den deriverte! Vi bruker den numeriske definisjonen der vi tilnærmer grenseverdiene med en dx (Δx) som er så liten som mulig:

$$f'(x) \approx \frac{f(x + dx) - f(x)}{dx} \tag{4.1.2}$$

Til å begynne med kjenner vi $f(x)$, altså $f(x_0)$. Dette er initialbetingelsen, for eksempel startkonsentrasjonen $c(t_0)$ i eksempelet ovenfor. Vi kjenner også den deriverte gjennom rateloven (f.eks. 4.1.1). I tillegg bestemmer vi selv tidssteget dx, men husk at det verken bør være for lite eller for stort. Den eneste ukjente i likning 4.1.2 er altså $f(x + dx)$, og det er jo nettopp dette vi prøver å finne. Med litt enkel algebra får vi omformet uttrykket slik at det blir et uttrykk for $f(x + dx)$. Vi ganger først med dx på begge sider:

$$f'(x) \cdot dx \approx f(x + dx) - f(x)$$

Deretter får vi $f(x + dx)$ aleine på høyre side og ender opp med følgende likning:

$$f(x + dx) \approx f(x) + f'(x) \cdot dx \tag{4.1.3}$$

Dette er *Eulers metode*, eller nærmere bestemt *Forward Euler*. Den brukes til å løse differensiallikninger, det vil si å integrere den deriverte slik at vi finner funksjonsverdiene. Siden vi ofte har med funksjoner av tid å gjøre, kaller vi gjerne dx for dt.

Numerisk metode 4.1.1: Eulers metode (Forward Euler)

Vi kan finne funksjonsverdiene $f(t_{k+1})$ ved å bruke funksjonsverdien $f(t_k)$ og den deriverte av funksjonen ved tida t_k, $f'(t_k)$ sammen med en steglengde dt som representerer en liten Δt.

$$f(t_{k+1}) = f(t_k) + f'(t_k) \cdot dt \tag{4.1.4}$$

For en generell ratelov som løses med Forward Euler gjelder altså at:

$$c(t + dt) \approx c(t) + c'(t) \cdot dt \tag{4.1.5}$$

Vi skal nå se på hvordan vi kan implementere denne metoden i Python.

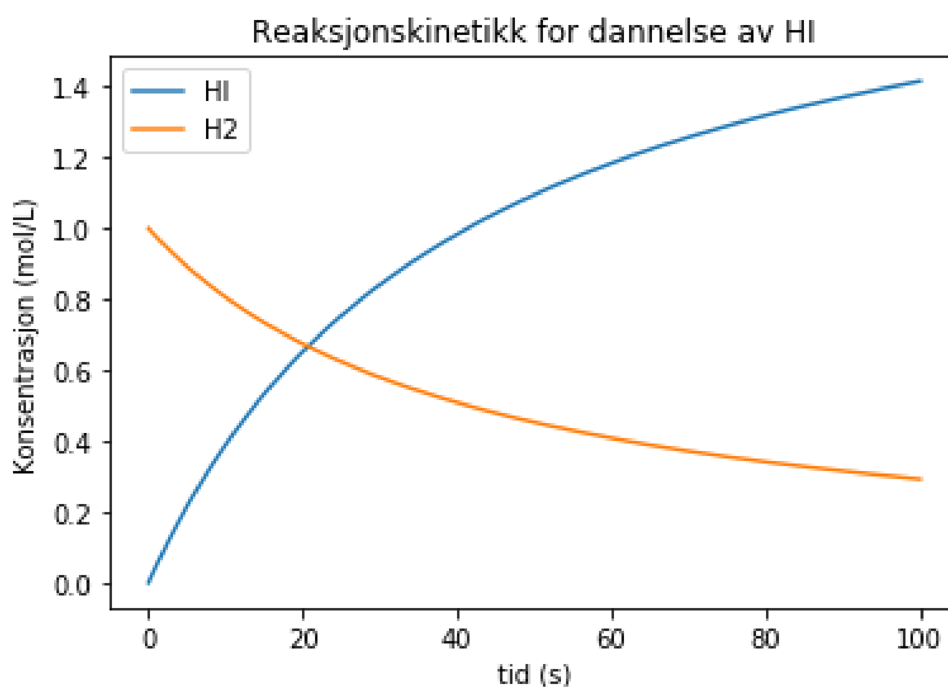
```
import numpy as np
import matplotlib.pyplot as plt

#Initialbetingelser
H0 = 1.0      # Konsentrasjon av hydrogengass i mol/L
I0 = 1.0      # Konsentrasjon av jodgass i mol/L
HI0 = 0       # Konsentrasjon av hydrogenjodid i mol/L
k = 4.84E-2   # Ratekonstanten

#Tidssteg
N = 100000   # antall intervaller
tid = 100    # antall sekunder
dt = tid/(N-1) # tidsintervallet

#Arrayer
t = np.zeros(N)  # Tid i sekunder
H = np.zeros(N)  # Konsentrasjon av H2
I = np.zeros(N)  # Konsentrasjon av I2
HI = np.zeros(N) # Konsentrasjon av HI
Hder = np.zeros(N) # Konsentrasjonsendring av H2
Ider = np.zeros(N) # Konsentrasjonsendring av I2
HIder = np.zeros(N)  # Konsentrasjonsendring av HI

# Initierer arrayene
H[0] = H0
I[0] = I0
HI[0] = HI0

# Eulers metode
for i in range(N-1):
    ... (fyll inn kode her)
```

```
plt.title('Reaksjonskinetikk for dannelse av HI')
plt.xlabel('tid (s)')
plt.ylabel('Konsentrasjon (mol/L)')
plt.plot(t, HI, label = 'HI')
plt.plot(t, H, label = 'H2')
plt.legend(loc=0)              # Merkelapp med bestemt posisjon i vinduet
```

Underveisoppgave 4.1.1

Studer programmet ovenfor trinn for trinn og prøv å fylle inn algoritmen i løkka. Prøv også å plotte for lenger enn 100 sekunder for å se om det støkiometriske forholdet stemmer når reaksjonen når likevekt.

Programmet ovenfor gir oss en graf for de første 100 sekundene som virker nokså logisk. Det kan være lurt å evaluere grafen en får fra et slikt program. Spesielt kan vi observere at [HI] beveger seg mot 2 mol/L dersom vi har 1 mol/L både hydrogen- og jodgass. Dette stemmer med det støkiometriske forholdet i reaksjonslikninga.



Figur 4.1.1: Simulering med Forward Euler.

Ofte kan det være lurt å gruppere kode i funksjoner slik at det er lettere å gjenbruke koden og fordi vi da har større kontroll på når vi gjør hver operasjon. Et eksempel på hvordan dette kan gjøres i koden ovenfor er:

```
import numpy as np
import matplotlib.pyplot as plt

# Konstanter og initalbetingelser
k = 4.84E-2
HI0 = 0
H0 = 1
I0 = 1
t0 = 0

# Tidsparametre
h = 1E-6
tid_slutt = 1000 # Tid i sekunder
N = int((tid_slutt - t0)/h)
t = np.zeros(N)
t[0] = t0

# Difflikninger
def dHI(H,I):
    return k*H*I

def dH2(H,I):
    return -0.5*k*H*I

def dI2(H,I):
    return -0.5*k*H*I

#Forward Euler
def FE():
    # Lager og initierer arrayer
    ... (fyll inn kode her)
    # Initialbetingelser
    ... (fyll inn kode her)
    for i in range(N-1):
        ... (fyll inn kode her)
    return HI, H, I, t

HI, H2, I2, t = FE()
plt.title('Reaksjonskinetikk for dannelse av HI')
plt.xlabel('tid (s)')
plt.ylabel('Konsentrasjon (mol/L)')
plt.plot(t, HI, label = 'HI')
plt.plot(t, H2, label = 'H2')
plt.show()
```

Underveisoppgave 4.1.2

Studer koden ovenfor og sammenlikn med koden i det forrige programmet. Legg inn algoritmen for Forward Euler i funksjonen FE. Modifiser programmet slik at det tar initialbetingelser som parametre i funksjonen FE i stedet for at de blir definert på starten av programmet.

Når vi løser differensiallikninger, er det ikke alltid tilnærmingene gir gode resultater. Spesielt når vi har å gjøre med svingninger gir Forward Euler ustabile og ofte helt feil resultater, spesielt med mindre dt er svært liten. Vi skal derfor se på flere metoder som er mye mer stabile og gir enda bedre resultater. Den første er en modifikasjon av Forward Euler og kalles *Backward Euler*.

4.1.2 Metode 2: Backward Euler

Forskjellen på Forward Euler (FE) og Backward Euler (BE) er liten i teorien, men betydelig for implementering. FE regner ut funksjonsverdien ved tida $t+dt$ med funksjonsverdien ved tida t og den deriverte av funksjonen ved tida t. BE regner ut funksjonsverdien ved tida $t+dt$ med funksjonsverdien ved tida t og den deriverte av funksjonen ved tida $t+dt$. Det er altså bare en forskjell i hvilken deriverte vi tar utgangspunkt i. Men dette gir en ganske stor forskjell i hvordan vi implementerer metoden. Du la kanskje merke til at vi prøver å finne funksjonsverdien $f(t_{k+1})$ med utgangspunkt i den deriverte i akkurat denne funksjonsverdien, $f'(t_{k+1})$. Siden $f'(t_{k+1})$ er avhengig av funksjonsverdien $f(t_{k+1})$, kan vi ikke løse den rett fram slik vi gjorde med FE.

La oss bruke eksempelet med hydrogenjodid som vi løste med FE. Istedenfor uttrykket vi får ved å bruke FE (4.1.5), får vi følgende uttrykk med BE:

$$c(t+dt) \approx c(t) + c'(t+dt) \cdot dt \tag{4.1.6}$$

Med utgangspunkt i c = [HI] får vi derfor:

$$[HI]_{t+dt} = [HI]_t + k \cdot [H]_{t+dt}[I]_{t+dt} \cdot dt \tag{4.1.7}$$

Vi får altså en likning der vi i utgangspunkt har tre ukjente, $[HI]_{t+dt}$, $[H]_{t+dt}$ og $[I]_{t+dt}$. Heldigvis er derimot disse ukjente størrelsene avhengige av hverandre. Det er fordi alle tar utgangspunkt i den samme rateloven for [HI] fordi de er støkiometrisk ekvivalente (forsvinner 1 mol H_2, dannes 2 mol HI, og det brukes 1 mol I_2 osv.). De tre ratelovene kan vi derfor formulere slik:

$$[HI]'(t) = k_r[H_2][I_2] \tag{4.1.8}$$

$$[H_2]'(t) = -0.5[HI]'(t) = -0.5 \cdot k_r[H_2][I_2] \tag{4.1.9}$$

$$[I_2]'(t) = [H_2]'(t) = -0.5 \cdot k_r[H_2][I_2] \tag{4.1.10}$$

Disse likningene kan vi løse ved å omforme 4.1.7 til et nullpunktsproblem:

$$F([HI]_{t+dt}) = [HI]_{t+dt} - [HI]_t - k \cdot [H_2]_{t+dt}[I_2]_{t+dt} \cdot dt = 0 \tag{4.1.11}$$

Tilsvarende får vi for H_2 og I_2:

$$F([H_2]_{t+dt}) = [H_2]_{t+dt} - [H_2]_t - 0.5 \cdot k \cdot [H_2]_{t+dt}[I_2]_{t+dt} \cdot dt = 0 \tag{4.1.12}$$

$$F([I_2]_{t+dt}) = [I_2]_{t+dt} - [I_2]_t - 0.5 \cdot k \cdot [H_2]_{t+dt}[I_2]_{t+dt} \cdot dt = 0 \tag{4.1.13}$$

Disse likningene ("Backward-likningene") kan vi implementere slik:

```
def F_HI(w,y,H,I):
    return w - y - dHI(H,I)*dt

def F_H(w,y,H,I):
    return w - y - dH(H,I)*dt

def F_I(w,y,H,I):
    return w - y - dI(H,I)*dt
```

Her har vi valgt å kalle den ukjente (f.eks. $[HI]_{t+dt}$) for w og den forrige funksjonsverdien (f.eks. $[HI]_t$) for y, men her står du selvfølgelig fritt til å velge det som gir mest mening for deg. Poenget er at både den ukjente verdien ved neste tidssteg og den forrige verdien ved forrige tidssteg tas som parametre i funksjonen, i tillegg til konsentrasjonen av $\mathrm{H_2}$ og $\mathrm{I_2}$. Siden det er w vi skal finne, virker det jo litt merkelig å ha den som parameter i funksjonen, men årsaken til dette er at vi systematisk vil prøve ut ulike verdier for w helt til likninga blir så nær null som mulig. Vi kjenner heldigvis en metode som kan brukes til akkurat dette formålet, nemlig Newtons metode (se kap. 3.2). Til dette trenger vi den deriverte av funksjonene 4.1.11, 4.1.12 og 4.1.13:

$$F'([HI]_{t+dt}) = 1 \tag{4.1.14}$$

$$F'([H_2]_{t+dt}) = 1 - 0.5 \cdot dt \cdot [I_2]_{t+dt} \tag{4.1.15}$$

$$F'([I_2]_{t+dt}) = 1 - 0.5 \cdot dt \cdot [H_2]_{t+dt} \tag{4.1.16}$$

Underveisoppgave 4.1.3

Deriver funksjonene 4.1.11, 4.1.12 og 4.1.13 for hånd og sørg for at du forstår hvorfor vi får resultatene i 4.1.14, 4.1.15 og 4.1.16.

I tillegg til å ha den deriverte av Backward-likningene trenger vi å gjøre en kvalifisert gjetning på $w = c_{t+dt}$. Dette kan vi gjøre med Forward Euler. Vi har nå en idé om hvordan vi kan løse problemet, og kan sette opp en skisse:

1. Formuler differensiallikningene som ordinære likninger som kan løses som et nullpunktsproblem (gjort ovenfor).
2. Deriver disse likningene med hensyn på de respektive konsentrasjonene: $F'([HI]_{t+dt})$, $F'([H_2]_{t+dt})$ og $F'([I_2]_{t+dt})$.
3. Bruk Forward Euler til å regne ut en estimert verdi, $w_{est} = c_{t+dt}$, for hver av produktene og reaktantene, $[HI]_{t+dt}$, $[H_2]_{t+dt}$ og $[I_2]_{t+dt}$.
4. Løs Backward-likningene med Newtons metode. Dette er den nye og mer presise verdien for $w_{ny} = c_{t+1}$.
5. Gjenta prosedyren med $c_t = w_{ny}$.

Her er et forslag til implementering av BE-algoritmen for reaksjonen mellom H_2 og I_2:

```python
# Backward Euler-funksjon
def BE():
    #Initialbetingelser og arrayer
    ... (sett inn kode her)
    # BE-algoritmen
    for i in range(N-1):
        # Initialisering
        HI_prev = c_HI[i]
        H_prev = c_H[i]
        I_prev = c_I[i]
        # Gjetning (Forward Euler)
        HI_est = HI_prev + h*dHI(H_prev, I_prev)
        H_est = H_prev + h*dH2(H_prev, I_prev)
        I_est = I_prev + h*dI2(H_prev, I_prev)
        # Newtons metode med N = 50
        for j in range(50):
            # Regner ut funksjonsverdien i Backward-funksjonene
            f_HI = F_HI(HI_est, HI_prev, H_est, I_est)
            f_H = F_H(H_est, H_prev, H_est, I_est)
            f_I = F_I(I_est, I_prev, H_est, I_est)
            # Regner ut funksjonsverdien i de deriverte Backward-funksjonene
            fder_HI = Fder_HI(HI_est, HI_prev, H_est, I_est)
            fder_H = Fder_H(H_est, H_prev, H_est, I_est)
            fder_I = Fder_I(I_est, I_prev, H_est, I_est)
            # Newtons metode (bruker f_- og f_der-verdiene ovenfor)
            HI_next = HI_est - f_HI/fder_HI
            H_next = H_est - f_H/fder_H
            I_next = I_est - f_I/fder_I
```

```
            # Setter ny estimert verdi
            HI_est = HI_next
            H_est = H_next
            I_est = I_next
        # Oppdaterer arrayene
        c_HI[i+1] = HI_est
        c_H[i+1] = H_est
        c_I[i+1] = I_est
        t[i+1] = t[i] + h
    return c_HI, c_H, c_I, t
```

Underveisoppgave 4.1.4

Sett sammen et fullstendig program som simulerer reaksjonen mellom hydrogen- og jodgass. Bruk kodesnutten ovenfor og simuler systemet i $1 \cdot 10^5$ sekunder.

Metoden kan kreve mye regnekraft siden den må løse tre likninger for hvert tidssteg, men en fordel er at den gir svært gode resultater selv for store tidssteg. Det gjør at den er veldig godt egna til å simulere systemer over lang tid. Hvis vi for eksempel skal studere et system over en million sekunder, blir arrayene og dataene så store for FE at datamaskinens minne blir overbelasta (overflow) fordi vi må ha et tidssteg som er relativt lavt (i hvert fall 10^{-4}) for å få gode resultater. I en slik simulering med BE kan vi derimot benytte tidssteg som er mange størrelsesordner større. Til og med så store tidssteg som $dt = 100$ og $dt = 1000$ vil kunne gi stabile og gode resultater i en slik simulering!

Vi kaller Forward Euler for en *eksplisitt metode* fordi vi finner tilstanden til et system ved et seinere tidspunkt $(t+dt)$ ut fra tilstanden av systemet ved det nåværende tidspunktet (t). Backward Euler er derimot en *implisitt metode* fordi vi finner tilstanden til et system ved et seinere tidspunkt $(t+dt)$ ut fra tilstanden av systemet ved dette seinere tidspunktet $(t+dt)$. Mer formelt kan vi si at hvis y(t) er tilstanden til et system, gjelder for en eksplisitt metode at:

$$y(t+dt) = F(y(t)) \tag{4.1.17}$$

der F er gitt ved et funksjonsuttrykk som er avhengig av tilstanden $y(t)$ (f.eks. er $F(y(t)) = k \cdot [H_2][I_2]$, der $y_1(t) = [H_2]$ og $y_2(t) = [I_2]$). For en implisitt metode finner vi $y(t+dt)$ ved å løse følgende likning:

$$G(y(t), y(t+dt)) = 0 \tag{4.1.18}$$

Vi kan generalisere BE-metoden slik:

Numerisk metode 4.1.2: Backward Euler

Vi kan finne funksjonsverdiene $f(t_{k+1})$ ved å bruke funksjonsverdien $f(t_k)$ og den deriverte av funksjonen ved tida t_{k+1}.

$$f(t_{k+1}) = f(t_k) + f'(t_{k+1}) \cdot dt \tag{4.1.19}$$

der dt = steglengden.

Implisitte metoder er gode på såkalte *stive* differensiallikninger, det vil si likninger som er numerisk ustabile med mindre tidssteget er svært lite. Numerisk ustabile likninger gir løsninger med en betydelig feil som ofte akkumuleres over tid. Som nevnt tidligere gir BE også gode resultater med store tidssteg og egner seg derfor godt til simulering av systemer over lengre tidsintervaller.

4.1.3 Metode 3: Runge-Kutta 4

Forward Euler er en del av en større familie av numeriske metoder kalt *Runge-Kutta-metoder*. Disse metodene blei systematisert og utvikla av de tyske matematikerne Carl Runge og Martin Wilhelm Kutta på starten av 1900-tallet. Den vanligste Runge-Kutta-metoden kalles *Runge-Kutta 4* (RK4), og er en *fjerdeordens* RK-metode. Med det menes det at en funksjon evalueres på fire ulike steder per funksjonsverdiestimat. Dette krever en del regnekraft, men RK4 er en svært mye brukt metode pga. dens stabilitet og presisjon. Det er en eksplisitt metode på lik linje med FE, så for store tidssteg er den ikke så god på stive differensiallikninger. Eulers metode er den enkleste RK-metoden, og er av første orden (RK1).

Vi kan generalisere RK4 slik:

Numerisk metode 4.1.3: Runge-Kutta 4

En differensiallikning $y' = f(t, y)$ kan løses gitt en initialbetingelse $y(t_0)$ og en steglengde dt ved følgende metode:

for n = 0, 1, 2, 3, ..., N:

$k_1 = f(t_n, y_n) \cdot dt$
$k_2 = f(t_n + \frac{dt}{2}, y_n + \frac{K_1}{2}) \cdot dt$
$k_3 = f(t_n + \frac{dt}{2}, y_n + \frac{K_2}{2}) \cdot dt$
$k_4 = f(t_n + dt, y_n + k_3) \cdot dt$

$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$
$t_{n+1} = t_n + dt$

Vi kan implementere metoden slik for eksempelet med H_2 og I_2 ovenfor:

```
def RK4(HI0,H0,I0,h,tid_slutt):
    N = int(tid_slutt/h)
    t = np.zeros(N)
    # Matriser
    HI = np.zeros(N)
    H = np.zeros(N)
    I = np.zeros(N)
    HIder = np.zeros(N)
    Ider = np.zeros(N)
    Hder = np.zeros(N)
    # Initialbetingelser
    HI[0] = HI0
    H[0] = H0
    I[0] = I0
    for i in range(N-1):
        # Første runde
        k1_HI = h * dHI(H[i],I[i])
        k1_H2 = h * dH2(H[i],I[i])
        k1_I2 = h * dI2(H[i],I[i])
        # Andre runde
        k2_HI = h * dHI(H[i] + 0.5*k1_H2, I[i] + 0.5*k1_I2)
        k2_H2 = h * dH2(H[i] + 0.5*k1_H2, I[i] + 0.5*k1_I2)
        k2_I2 = h * dI2(H[i] + 0.5*k1_H2, I[i] + 0.5*k1_I2)
        # Tredje runde
        k3_HI = h * dHI(H[i] + 0.5*k2_H2, I[i] + 0.5*k2_I2)
        k3_H2 = h * dH2(H[i] + 0.5*k2_H2, I[i] + 0.5*k2_I2)
        k3_I2 = h * dI2(H[i] + 0.5*k2_H2, I[i] + 0.5*k2_I2)
        # Fjerde runde
        k4_HI = h * dHI(H[i] + k3_H2, I[i] + k3_I2)
        k4_H2 = h * dH2(H[i] + k3_H2, I[i] + k3_I2)
        k4_I2 = h * dI2(H[i] + k3_H2, I[i] + k3_I2)
        # Utregning av funksjonsverdien
        HI[i+1] = HI[i] + (1/6)*(k1_HI + 2*k2_HI+ 2*k3_HI + k4_HI)
        H[i+1] = H[i] + (1/6)*(k1_H2 + 2*k2_H2 + 2*k3_H2 + k4_H2)
        I[i+1] = I[i] + (1/6)*(k1_I2 + 2*k2_I2 + 2*k3_I2 + k4_I2)
        #Oppdaterer tidssteget
        t[i+1] = t[i] + h
    return HI, H, I, t
```

4.1.4 Oppsummering

Vi har nå sett på tre ulike metoder for løsing av differensiallikninger. Forward Euler er mest ment som en måte å bli kjent med løsing av differensiallikninger på. Den er relativt intuitiv og er nokså enkel å implementere. Algoritmene for Backward Euler og RK4 er derimot verken intuitive eller spesielt enkle å implementere. Dessuten krever de relativt stor regnekapasitet.

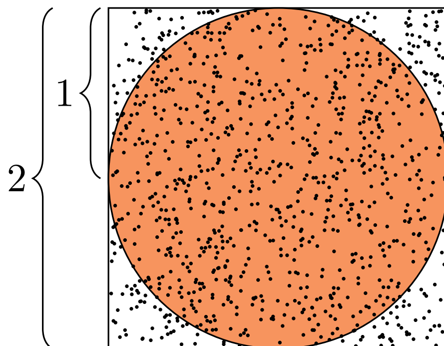
Derimot er det slik metoder som brukes mest i praksis.

BE brukes spesielt når vi trenger å se på større tidsperioder og når difflikningene er numerisk ustabile med andre metoder (stive likninger). RK4 er en typisk "go to"-metode for de fleste andre (ikke-stive) difflikninger, og gir gode resultater uten alt for mye regning. Det er ikke så lett å forstå akkurat *hvorfor* de gir bedre resultater enn FE, og hvorfor algoritmene *ser ut som de gjør*. I utgangspunktet kan en utlede metodene og analysere feilen de gir ved bruk av matriser i lineær algebra. Dette blir derimot for tidkrevende å gå inn på her.

4.2 Stokastiske simuleringer

En stokastiske simulering er en simulering der tilfeldige hendelser inntreffer med en viss sannsynlighet. Det er mange prosesser i naturen som er tilfeldige eller delvis tilfeldige, f.eks. radioaktivt henfall, mutasjoner og diffusjon. I dette kapitlet skal vi primært se på generell bruk av tilfeldige tall, for så å benytte dette på et eksempel fra kjemien.

Vi kan til og med tilnærme noen ikke-stokastiske hendelser med stokastiske forsøk! Et eksempel på dette fra matematikken er når vi tilnærmer selveste *pi* med tilfeldige tall. Pi er jo, som kjent, ikke tilfeldig, men vi kan gå veien om tilfeldige tall for å estimere pi. Siden pi er forholdet mellom omkretsen og diameteren til en sirkel (O/d), eller mellom arealet og kvadratet av radius (A/r^2), vil en sirkel med radius 1 ha $A = \pi$. For å tilnærme arealet til sirkelen lager vi et kvadrat slik at sirkelen passer *akkurat* inn i kvadratet (en *innskreven* sirkel). Kvadratet vil ha en sidelengde på $2r = 2$. Deretter tegner vi N tilfeldige punkter i kvadratet og teller antall punkter M som befinner seg innenfor eller på sirkelen.



Figur 4.2.1: Sirkel med radius lik 1 innskrevet i et kvadrat med sidelengde 2. Her er det 1000 tilfeldig genererte punkter.

Forholdet $\frac{M}{N}$ vil da fortelle oss hvor stort område sirkelen dekker i forhold til kvadratet. Vi kan derfor finne hvor stort arealet til sirkelen er ved:

$$O_{sirkel} \approx \frac{M}{N} \cdot A_{kvadrat}$$

For å sjekke om punktene treffer i sirkelen, kan vi bruke likningen for en sirkel med sentrum i origo:

$$x^2 + y^2 = r^2$$

Da må altså et punkt med koordinat (x, y) tilfredsstille følgende ulikhet for å ligge i en sirkel med radius 1:

$$x^2 + y^2 \leq 1$$

Vi kan implementere algoritmen slik:

```python
from math import pi
import numpy.random

N = 100000
x=np.random.uniform(0,1,N)
y=np.random.uniform(0,1,N)

a = x**2 + y**2

# Teller alle punkter som er innenfor sirkelen
M = 0
for i in range(N):
    if a[i] <= 1:
        M += 1
A_kvadrat = 4
pi_est = M/N*A_kvadrat

print(" Verdien av pi:",pi ," Estimert pi:",pi_est)
```

Det eneste ukjente fra programmet ovenfor er funksjonen `uniform` fra random-biblioteket som finnes i numpy. Det er denne funksjonen som sammen med `randint` og `random` utgjør de viktigste funksjonene for å generere tilfeldige tall i Python. De fungerer slik:

Funksjon	**Forklaring**
uniform(a, b, N)	Genererer N normalfordelte flyttall fra og med a til b.
randint(a, b, N)	Genererer N tilfeldige heltall fra og med a til b.
random()	Genererer et tilfeldig flyttal fra og med 0 til 1.

Underveisoppgave 4.2.1

Prøv ut programmet ovenfor og sjekk hvor godt estimat av pi du klarer å få. Regn gjerne også ut den relative feilen, ϵ, i prosent gitt estimert og teoretisk verdi v:

$$\epsilon = \frac{|v_{estimert} - v_{teoretisk}|}{v_{teoretisk}} \cdot 100\ \% \tag{4.2.1}$$

Estimeringen av pi er et eksempel på en *Monte Carlo-simulering* (MC). Navnet kommer fra det store kasinoet i Monte Carlo, der nettopp tilfeldige tall og spill står sentralt. Prinsippet i MC-metoder er å tilnærme spesielt deterministiske fenomener med store mengder av tilfeldige tall. Dette kan f.eks. benyttes til å beregne integraler, noe som er spesielt nyttig hvis integralene er analytisk uløselige.

4.2.1 Tilfeldige bevegelser

Vi skal her se på en MC-tilnærming til tilfeldig bevegelse av store partikler i løsning. Dette er en enkel modell for diffusjon av ikke-reagererende partikler som kan beskrive såkalte *Brownske bevegelser*. Brownske bevegelser ble først beskrevet av botanisten Robert Brown i 1827. Han oppdaga at små pollenkorn i løsning beveget seg fram og tilbake i et tilfeldig mønster. I dag veit vi at dette skyldes at de små vannmolekylene dytter på pollenkornet i mange tilfeldige retninger. Det samme gjelder større partikler som enkelte luktmolekyler (parfyme) og røyk, som vi jo kan lukte og noen ganger observere direkte i makroskala.

For å simulere det som skjer på mikroskala kan vi lage et program der vi for hvert tidssteg trekker tilfeldige tall som bestemmer retningen til partikkelen. Vi kan først se på hvordan vi kan gjøre dette ved å konstruere et rutenett der en partikkel kan bevege seg i fire retninger (opp, ned, høyre og venstre). Skråbevegelser kan beskrives som en kombinasjon av disse bevegelsene:

	(x,y+1)	
(x-1,y)	(x,y)	(x+1,y)
	(x,y-1)	

Figur 4.2.2: Mulige bevegelser for en enkel partikkel.

Disse bevegelsene kan vi representere med posisjonsarrayer x og y. Posisjonen kan starte i origo, $(0, 0)$, og så kan vi øke eller redusere med 1 i en tilfeldig retning. Dette kan vi gjøre ved å trekke et tilfeldig tall mellom 1 og 4 som representerer bevegelse i rutenettet slik:

	3	
2	0	1
	4	

Figur 4.2.3: Bevegelse som følge av tilfeldig tall.

Hvis vi for eksempel trekker tallet 4, vil partikkelen bevege seg én rute nedover i y-retning. Da trekker vi fra 1 i arrayen som inneholder y-koordinatene. Et eksempel på hvordan dette kan gjøres, er slik:

```
import numpy as np
import matplotlib.pyplot as plt

# Initialisering
N = 100000
x = np.zeros(N)
y = np.zeros(N)

# Genererer tilfeldige koordinater
for i in range(0, N-1):
    verdi = np.random.randint(1, 5)
    if verdi == 1:
        x[i+1] = x[i] + 1
        y[i+1] = y[i]
    elif verdi == 2:
        x[i+1] = x[i] - 1
        y[i+1] = y[i]
    elif verdi == 3:
        x[i+1] = x[i]
        y[i+1] = y[i] + 1
    elif verdi == 4:
        x[i+1] = x[i]
        y[i+1] = y[i] - 1

# PLotting
plt.title("Tilfeldig bevegelse med " + str(N) + " steg")
plt.plot(x, y)
plt.savefig('randwalk_%.d_steg.png' %(N)) # Lagrer figuren som png-fil
plt.show()
```

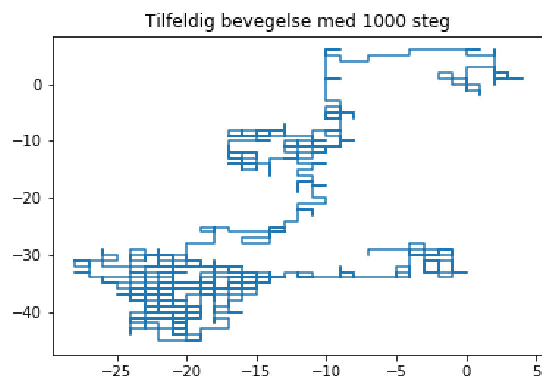
Underveisoppgave 4.2.2

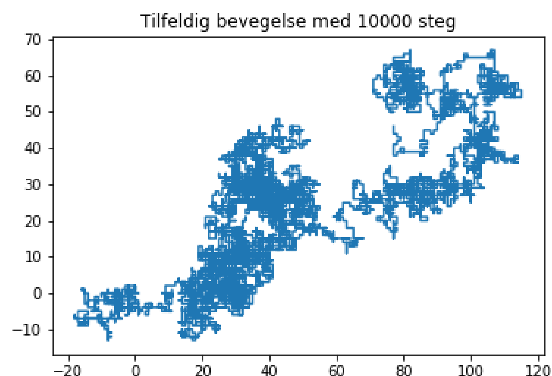
Gi eksempler på situasjoner der programmet ovenfor kan være en grei tilnærming for å beskrive bevegelsen til en partikkel. Hvilke forutsetninger og begrensninger har denne modellen?

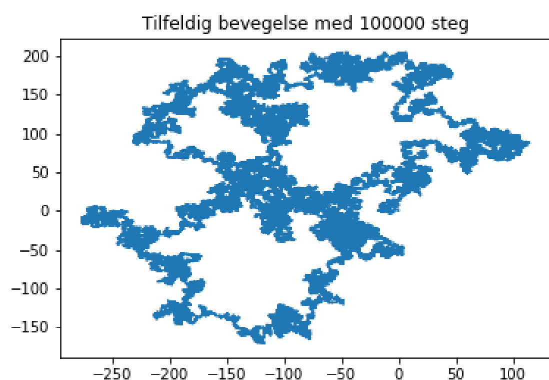
Underveisoppgave 4.2.3

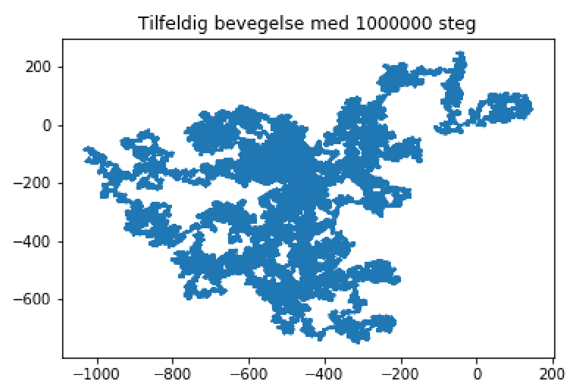
I programmet ovenfor benyttes en forflytning på 1 rute. Modifiser programmet slik at det forflytter seg en tilfeldig avstand, for eksempel mellom 0 og 1, for hver iterasjon i løkka.

Programmet ovenfor *kan* produsere følgende plott (men lykke til med å få *akkurat* de samme plottene!):









Simuleringsgrafikk

Vi kan ha det litt moro med enkel simuleringsgrafikk fra biblioteket *turtle* for å illustrere diffusjons*prosessen*, ikke bare resultatet. Turtle-grafikk har navnet sitt fra at man bruker en peker som gradvis forlytter seg rundt i et koordinatsystem. Denne pekeren kan selvfølgelig ha form som en skilpadde. Vi importerer turtle-biblioteket og konstruerer et såkalt *objekt* som vi kan manipulere.

Et objekt er en konstruksjon som kommer fra en *klasse* som definerer et sett med egenskaper som hvert objekt fra klassen blir tildelt. Fordelen ved å bruke klasser er at vi kan lage mange objekter med samme grunnegenskaper, mens vi samtidig kan manipulere hvert objekt for seg uten å overskrive disse grunnegenskapene. Når vi manipulerer et objekt slik at vi tilegner det andre egenskaper eller handlinger, kaller vi på en *metode* knytta til dette objektet. En metode er som en funksjon, bare at den alltid virker på et objekt. Et eksempel er *turtle*-objektene som vi nå skal se litt nærmere på:

```python
from turtle import * # Importerer alt fra turtle-biblioteket

Raphael = Turtle()    # Lager et turtle-objekt
raphael.color('red')  # Kaller metoden color på objektet
raphael.shape('turtle') # Kaller metoden shape på objektet
```

I programmet ovenfor konstruerer vi et objekt som vi kaller *Raphael* ved å kalle på objektet fra klassen `Turtle`. Klasser har stor forbokstav, så vi kan ofte kjenne dem igjen i kode på grunn av dette. Deretter gir vi objektet to egenskaper ved å kalle på metodene `color` og `shape`. Det finnes mange slike metoder for turtle-objekter, og her er noen av de viktigste:

Metodeeksempel	**Forklaring**
Leonardo = Turtle()	Lager et turtle-objekt
Leonardo.shape('arrow')	Gir form til objektet
Leonardo.color('blue')	Gir farge til objektet
Leonardo.speed(1)	Tegnefart fra 1 til 10 (0 er raskest)
Leonardo.pos()	Gir posisjonen til objektet
Leonardo.forward(50)	Flytter objektet 50 piksler fram
Leonardo.backward(50)	Flytter objektet 50 piksler tilbake
Leonardo.left(45)	Vender 45 grader mot venstre
Leonardo.right(45)	Vender 45 grader mot høyre
Leonardo.penup()	Streken tegnes ikke
Leonardo.goto((20,30))	Flyttes til punktet (20,30)
exitonclick()	Avslutter når en klikker på ruta

Disse metodene kan vi bruke til å lage et partikkelobjekt som beveger seg tilfeldig for hvert tidssteg, og som på denne måten simulerer brownske bevegelser:

```python
from turtle import *
import numpy.random

partikkel = Turtle() # Lager et objekt
partikkel.color('purple') # Fargelegger
partikkel.shape('circle') # Gir sirkelform til objektet
partikkel.turtlesize(0.5) # Endrer størrelsen til objektet
partikkel.speed(0)     # Endrer farten til objektet
N = 1000 # Antall ganger pekeren skal bevege seg
avstand = 10 # Hvor langt pekeren går mellom hver gang

for i in range(N):
    vinkel = numpy.random.randint(0,360) # Tilfeldig vinkel hver gang
    partikkel.right(vinkel) # Snur tilfeldig vinkel
    partikkel.forward(avstand) # Går framover
```

Underveisoppgave 4.2.4

Lag en simulering som ovenfor, bare med to partikler.

4.3 Oppsummering

Nå har du sett på hvordan programmering kan brukes i kjemi til blant annet praktisk datahåndtering, statistisk analyse, numerisk tolking av data og simulering av deterministiske og tilfeldige systemer. Det finnes utallige andre anvendelser av programmering i kjemi, og jeg håper du har fått litt lyst til å utforske disse mulighetene, og at du opplever at programmering er et nyttig verktøy i en kjemikers hverdag. Lykke til på ferden!