Programmering for kjemikere

Kompendium i IN-KJM1900
Høst 2020

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

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 vi må ha nokså god oversikt for å kunne forstå det store bildet. Her kan programmering og beregninger 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 og beregninger 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. Vi kommer til å bruke programmering og *beregninger* litt om hverandre her, selv om beregninger i denne sammenhengen er et videre begrep som mer eller mindre innebefatter all bruk av databeregninger. Noen sentrale temaer fra kjemi som programmering og beregninger kan hjelpe oss med, er:

1. Datahåndtering – 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 nye materialer og legemidler?
4. Bevegelse av partikler – Hvordan beveger og vekselvirker partikler gjennom katalysatorer eller bergarter?
5. Biokjemiske prosesser – Hvordan regulerer proteiner transport gjennom cellemembranen? Hvordan fungerer viktige enzymer og hormoner i kroppen?

Dette er bare noen spørsmål som kan utforskes med programmering. 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

> **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, bør vi også lære oss å lage flere av dem selv. Dette gjør vi av flere årsaker:

- Vi forstår algoritmene bedre hvis vi lager dem selv.
- Vi har mulighetene til å legge til modifikasjoner dersom vi skulle trenge det. Desto mer du forstår, desto mer kontroll har du på hva som skjer.
- Det å lage spesifikke algoritmer kan gi enda bedre innsikt i det kjemifaglige innholdet.

Alle disse er gode grunner til at vi skal programmere algoritmer fra bunnen av, men når du først har lært algoritmene, skal vi også se på importering av algoritmer fra biblioteker.

1.3 Modellering

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

Noen ganger kan kjemi, og andre naturvitenskapelige fag, 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 ulike modeller, og selv om ingen 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 forteller oss om

energinivåene til elektronene, ikke om plassering og bevegelse. Elektroner går ikke i sirkulær 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 *orbitalmodellen* og VSEPR-modellen (Valence shell electron pair repulsion), 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 hver 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 vi 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 det kan gi deg ny og bedre innsikt i viktige kjemiske prinsipper.

1.3.1 Bruk av programvare

Selv om du kan få bedre innsikt i hvordan ting henger sammen ved å programmere alt fra grunnen av, er det i mange tilfeller også svært nyttig å kunne beherske ferdig programvare. Som tidligere nevnt finnes det mange ferdige algoritmer og programmer som utmerket godt kan, og bør, brukes. I tillegg finnes det en del programvare som kan brukes til å utforske fenomener som krever mer kompliserte algoritmer enn det er pedagogisk hensynsmessig å gjennomgå her.

Hovedpoenget er å forstå kjemi bedre. Dersom algoritmene er for vanskelig eller for tidkrevende, kan det hende forståelsen kommer bedre fram ved bruk av programvare enn ved å programmere alt selv. For å bruke en del programvare, og for å kunne bruke biblioteker i Python og andre programmeringsspråk, må du likevel kunne en del programmering. Det er fordi mange typer programvarer, spesielt ikke kommersielle, baserer seg delvis på et CLI (Command Line Interface) kombinert med, eller i stedenfor, et GUI (Graphical Unit Interface). Et grafisk grensesnitt (GUI) er ofte enkelt å beherske for de med litt trening, men kommandolinjegrensesnitt (CLI) krever en del mer. De generelle ferdighetene om datastrukturer og grunnprinsipper i programmering kommer her godt med.

1.4 Numerisk matematikk

I mange tilfeller kan vi ikke finne eksakte svar og analytiske formler for det vi ønsker å finne ut. Mange modeller som beskriver fenomener i kjemi, er

ganske sammensatte. Derfor er de også veldig sjeldent analytisk løsbare, det vil si løsbare med algebra og klassisk matematikk. Disse problemene må vi løse *numerisk*. I numerisk matematikk bruker vi algoritmer for å tilnærme en løsning til et problem. Disse tilnærmingene kan være gode nok til vårt formål, eller de kan være helt uegna. En viktig del av den numeriske matematikken er derfor å kunne analysere mulige feilkilder ved en numerisk metode og avviket fra analytiske resultater, det vil si den numeriske *feilen*.

Numerisk matematikk

Numerisk matematikk er en del av matematikken som handler om å lage algoritmer for tilnærme matematiske løsninger med minst mulig feil.

Vi skal se på en del numeriske metoder som kan brukes til å løse både kjemifaglige og andre problemstillinger. Feilanalysen er litt tonet ned til fordel for kodeimplementering og forståelse, men vi skal adressere det der det er spesielt relevant. Vi vil se på følgende områder innenfor numerisk matematikk:

1. Numerisk løsing av likninger (nullpunktsalgoritmer).
2. Numerisk derivasjon.
3. Numerisk integrasjon.
4. Numerisk løsing av differensiallikninger.

Det finnes utallige algoritmer innenfor disse områdene. Vi starter med noen som bygger på grunnprinsipper i matematikk, og som derfor er ganske intuitive. Det er ofte ikke disse algoritmene som brukes mest fordi de er for enkle til å gi gode nok tilnærminger i mange tilfeller. Men det er en pedagogisk verdi i å se på disse for å bygge opp den faglige forståelsen av metodene som brukes.

1.5 Python

Python er oppkalt etter humorgruppa *Monty Python* fordi det skal være morsomt og lett å lære. Vi har valgt Python som programmeringsspråk av flere årsaker:

1. Det har en enkel syntaks som gjør det lettere å lære og lese enn mange andre programmeringsspråk.
2. Syntaksen ligger ofte tett opp mot notasjoner vi bruker i matematikk.
3. Det har svært mange gode biblioteker med ferdig kode som vi kan bruke.
4. Python har blitt en standard i veldig mange områder innefor akademia og næringslivet.

5. Objektorientering er mulig, men valgfritt.

En åpenbar ulempe er at Python er et svært langsomt språk. Derfor skrives en del av bibliotekene i Python helt eller delvis i raske språk som C eller C++. Det kan med andre ord være en fordel å være flerspråklig. Dersom du velger å investere litt ekstra tid i å lære et nytt programmeringsspråk etter hvert, vil det antakelig føles mye lettere enn å lære det første!

Å kjøre Python-programmerer krever kun at du installere Python. Det kan du gjøre på flere måter. Den enkleste måten er å installere en distribusjon av Python som heter *Anaconda*. Da får du alt du trenger, i hvert fall foreløpig. Det følger også med en editor som heter *Spyder*. En editor er et GUI der du kan skrive kode. Spyder har en del funksjonalitet som du bør bli kjent med. Editorer som Spyder gir også output i det samme GUI-et. Et alternativ til dette er å bruke kommandolinja til å kjøre programmer. Vi skal fokusere på generell kode, ikke brukergrensesnittet, i denne boka.

1.6 Jupyter Notebook

Et verktøy som egner seg svært godt til rapportskriving som inneholder kode, er *Jupyter Notebook*. Dette inneholder både matematisk tekstformatering og en editor som kan kjøre kode. Jupyter Notebook finnes i Anaconda-pakka, så den kan startes herfra. Da åpnes en liste med filer som ligger lokalt på datamaskinen (selv om det åpnes i en nettelser)

Trykk på *new* og *Python 3*, så kommer du til et tomt dokument. Notebooken bygges opp av blokker med innhold. Det finnes i hovedsak to innholdsblokker, *code* og *markdown*. I en kodeblokk kan du skrive Python-kode akkurat som i en annen editor. I tillegg kjører også koden her, og du får en output under kodeblokken. Trykk CTRL + SHIFT for å kjøre koden i hver blokk. Men pass på – all koden i en notebook henger sammen.

Dersom vi skrive tekst, kan vi bruke markdown-blokkene. Her skriver vi i et markeringsspråk som kalles *markdown*. Dette gjør oss i stand til å blande tekst med matematikk, bilder og annet. Markdown har sin egen kode, men det er ikke så mye å holde styr på her. Følgende tabell oppsummerer det viktigste:

Tegn	Forklaring
`#`	Overskrift, nivå 1
`##`	Overskrift, nivå 2
`###`	Overskrift, nivå 3
`_tekst_`	Kursiv
`**tekst**`	Fet skrift
`![bildetekst](bilde.jpg)`	Sett inn bilde
`[tittel](https://www.elg.no)`	Lenke
`|tabelltekst|tabelltekst|`	Tabeller

Dersom vi ønsker å skrive matematikk i teksten, trenger vi å lære oss litt *LaTeX* (uttales «latekj» eller «latek» og er ikke å forveksle med polymeremulsjonen lateks). Latex er et dokumentbehandlingsspråk som brukes for å skrive pent formaterte dokumenter, spesielt med matematiske notasjoner. All matemikk i Latex-kode skrives med dollartegn på hver side av uttrykket. Hvis du ønsker matematikken midtstilt, kan du bruke doble dollartegn på hver side. En del nyttig Latex-kode finner du i tabellen nedenfor:

Tegn	Forklaring
`\alpha`	Gresk bokstav, f.eks. α
`\frac{teller}{nevner}`	Brøk
`^`	Hevet tekst
`_`	Senket tekst
`\sum_{nedre}^{øvre}`	Summetegn
`\int_{nedre}^{øvre}`	Integral
`\lim_{x \to \infty}`	Grenseverdi
`\left(`	Stor venstreparentes
`\right)`	Stor høyreparentes

For å gruppere sammensatte indekser eller eksponenter, eller annen tekst, benyttes krøllparenteser.

Underveisoppgave 1.1

Prøv å skrive tekst og Latex-kode i en markdown-blokk. Bruk doble dollartegn hvis du ikke vil ha formlene i løpende tekst. Prøv å skrive noen summer, integraler og grenseverdier. Legg til en annen blokk med kode som du kjører.

1.7 GitHub*

Du bruker kanskje Dropbox, Google Disk eller OneDrive til fillagring? Disse systemene inneholder *versjonskontroll*. Det vil si at de ikke bare lagrer filene dine, men de sparer også på tidligere versjoner av filene. Dette kan være nyttig også i programmering. Etter hvert kan nemlig programmene dine bli lange, og du vil kunne slite med å holde styr på når og hvordan ulike endringer har blitt gjort. Eller så ønskerd du kanskje å samarbeide med noen om et program, og er interessert i å se hva slags endringer samarbeidspartneren din har gjort, og om de kommer i konflikt med endringer du har gjort samtidig. Dropbox og liknende programmer er ikke optimalt for å holde styre på dette. Derfor introduserer vi her Git.

Git er et versjonskontrollsystem som sparer på absolutt alle tidligere versjoner av filene dine og markerer endringer som er blitt gjort av ulike bidragsytere. Dessuten er det et fint sted å lagre og lese Jupyter Notebooks. Et system

som bruker Git er GitHub. For å komme i gang med GitHub, kan du følge denne oppskriften:

1. Lag en bruker på github.com.
2. Lag et repository (repo). Dette er en slags mappe der du kan legge programmer som hører sammen.
3. Velg mellom *public* (alle kan se og dele filene dine) eller *private* (bare du kan se).
4. Kryss av på «Initialize this repository with a README».

Gratulerer! Du har lagd ditt første repo. Du kan legge inn filer her ved å laste dem opp manuelt på nettsida, men for å utnytte Git-systemet til det fulle, bør du enten bruke GitHub med kommandolinja eller programvaren GitHub Desktop. For å bli kjent med GitHub, kan det være lurt å velge Desktop-versjonen først. Last ned GitHub Desktop og/eller installer GitHub ved hjelp av *terminalen* i MacOS/Linux eller *Powershell* i Windows. Siden du allerede har laget et repo, kan vi nå hente det til datamaskinen ved å *klone* det. I kommandolinja skriver du dette fra det stedet der du ønsker å ha filene lagret:

```
>> git clone https://github.com/reponavn
```

I GitHub Desktop finner du «clone repository» under «fil». Alternativt kan du lage et nytt repo herfra. Nå kan du åpne mappa lokalt og legge inn de filene du måtte ønske å ha i repoet ditt. Nå er de lagt inn lokalt, men for å legge dem til repoet i Git, må vi gjøre dette eksplisitt. Med kommandolinja navigerer du deg til mappa du nettopp har laget som repo. Deretter gjør du følgende:

```
>> git add .
>> git commit -m 'kommentar om endringer i fila'
>> git push
```

Den første kommandoen legger til alle filene dine i en liste over filendringer. Dersom du vil legge til mange uten en spesiell kommentar til hver av dem, kan du bruke punktum som vist ovenfor. Dersom du vil legge til en og en fil, kan du erstatte punktumet ovenfor med filnavnet. Commit-kommandoen brukes for å gjøre endringene klare for Git med en kommentar (*-m*) der du skriver hva du har gjort med fila/filene siden sist. Til slutt sender du endringene til Git med *push*.

Med GitHub Desktop kommer filene automatisk opp, og du ser alle endringer som er lagt til med grønt, og alt som er fjernet med rødt. Deretter skriver du inn en beskrivelse og trykker *commit to master*. Så er det bare å trykke på *push origin*. Første gang du bruker repoet, må du antakelig trykke

publish repository før du kan pushe innholdet. Endringer i filene kan du nå se i GitHub Desktop eller på github.com under *history*.

Dersom noen andre har gjort arbeider på et repo som du eier, eller du har gjort arbeid på en annen lokal datamaskin, må du hente endringene ned til den datamaskinen du jobber på. I Github Desktop trykker du på en knapp der det står «Pull origin». I kommandolinja skriver du isteden følgende:

```
>> git pull
```

Git kan være et uvurderlig verktøy for lagring, utvikling og samarbeid. Og dersom du legger til en Jupyter Notebook, blir formatet stort sett ivaretatt. Bruk av Git er derfor et nyttig verktøy for programmering.

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

2.1 Plotting

Det finnes flere typer biblioteker som kan gi oss fine og profesjonelle plott. Vi begynner med et mye brukt bibliotek som heter *matplotlib*.

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 *data*. Først ser vi derfor på et enkelt eksempel

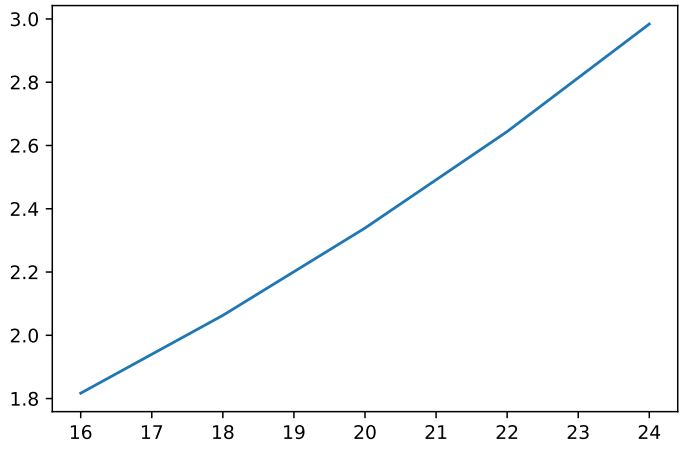
der vi definererer dataene i lister. Denne framgangsmåten egner seg godt hvis du skriver inn datapunkter fra eksperimenter manuelt når det ikke er for mange punkter. La oss ta et eksempel der vi har målt damptrykket til vann i et glassrør ved ulike temperaturer. Vi har fått følgende data:

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:



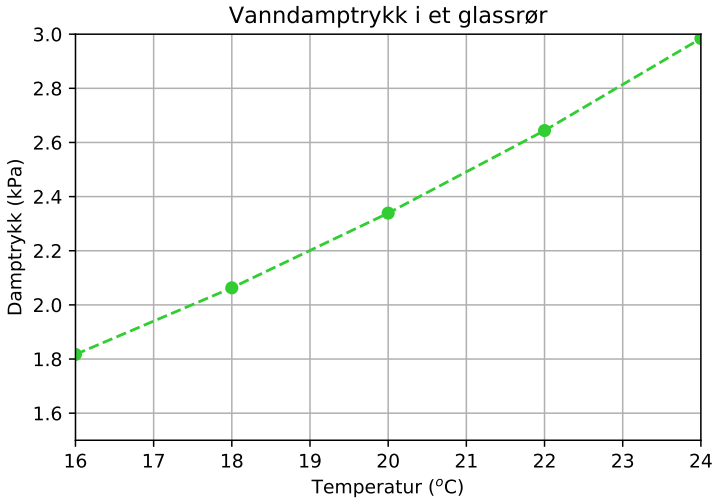
Figur 2.1: Enkel figur for damptrykk av vann ved ulike temperaturer.

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 2.2: 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 2.3: 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

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. Det er fordi det egentlig ikke er mulig å gjøre noe helt kontinuerlig i matematisk forstand på en datamaskin. Alt er diskrete størrelser på datamaskinen, så det beste vi kan gjøre er å tilnærme kontinuerlige funksjoner med mange nok diskrete verdier. Disse verdiene 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()
```

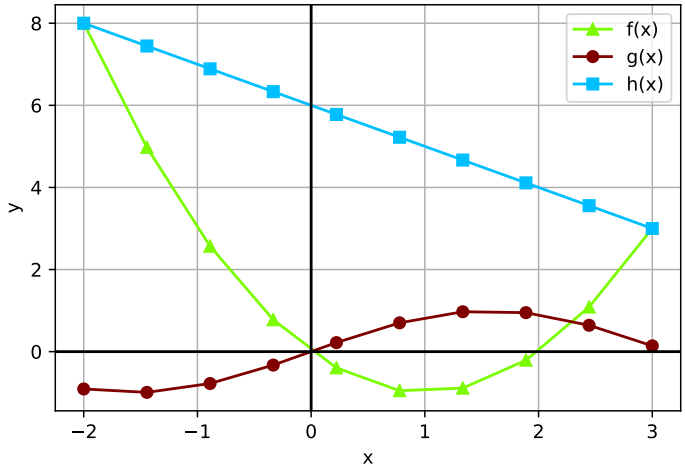
Funksjonen `linspace(a, b, n)` genererer n punkter med lik avstand mellom a og b. Dette gir en glatt og fin graf, men det er verdt å merke seg at den faktisk ikke er helt glatt. Siden vi har å gjøre med diskrete punkter, får vi aldri helt glatte grafer. Dette kan du se hvis du f.eks. prøver å generere 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 2.4: Plotting av flere funksjoner.

Underveisoppgave 2.2

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.

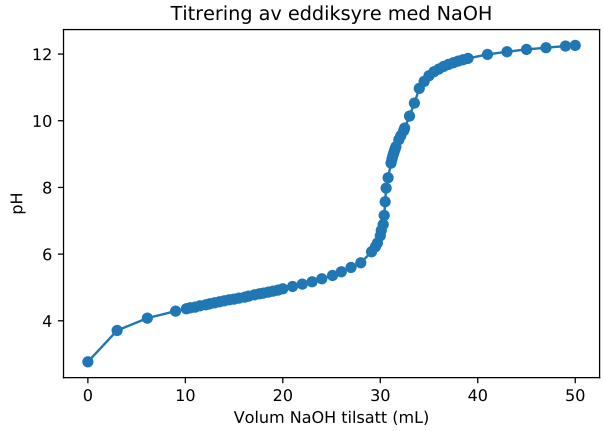
La oss se på et eksempel der vi har gjennomført en titrering av 25 mL 0.12 M eddiksyre med 0.10 M lut (NaOH). Dataene blir lagret i fila `titrering_eddiksyre_NaOH.t` med komma mellom verdiene. Her er et utsnitt av datafila:

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 2.5: 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*.

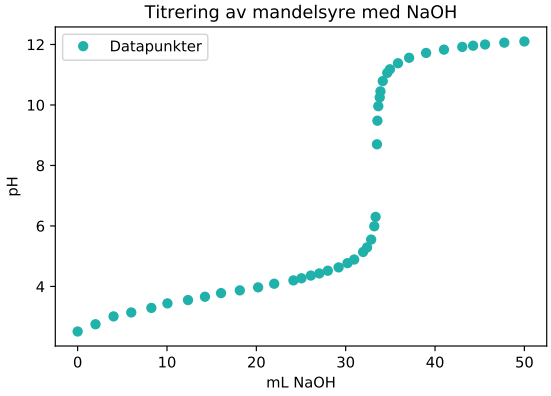
2.3.1 Interpolasjon

Interpolasjon

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

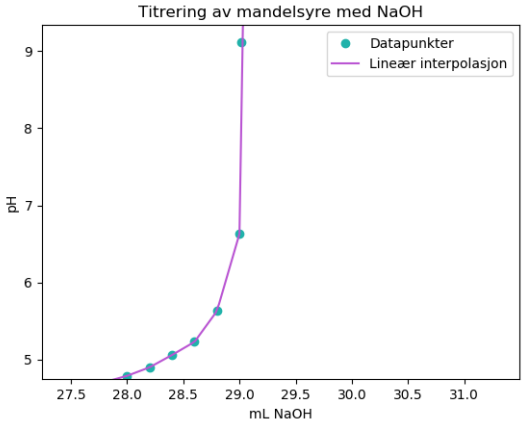
Både interpolasjon og *regresjon* tilnærmer datapunkter med funksjoner. Det er likevel en viktig forskjell mellom disse teknikkene. Regresjon avviker ofte fra en del av de kjente verdiene for å være en best mulig tilpasset *alle* dataene. Interpolasjon passer *eksakt* i datapunktene, men gir derfor ofte 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 ofte er gode tilnærminger til mange ulike datasett. Til dette bruker vi et datasett som vi har laget ved en titrering av mandelsyre med lut (NaOH).



Figur 2.6: Titreringskurve – pH-en endrer seg kraftig ved ekvivalenspunktet.

Ved ekvivalenspunktet endrer pH-en seg svært mye. Det er dermed vanskelig å få datapunkter som gir enkel avlesing av pH-verdien ved ekvivalenspunktet. Figuren på forrige side viser dette.



Figur 2.7: 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!

```
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()
```

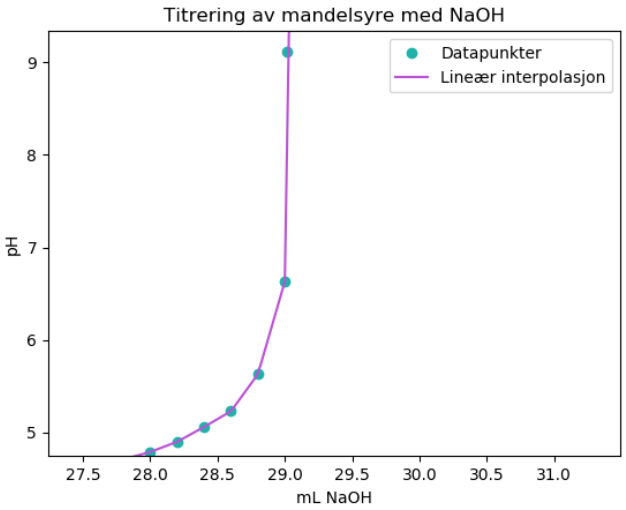
Vi definerer altså en ny funksjon f som inneholder alle de datapunktene vi hadde, men også med en rett linje mellom hver av disse datapunktene. Vi bruker funksjonen *interp1d* til dette. Den tar en x-array og en y-array som første parametre, og deretter skriver vi inn hva slags interpolasjon vi ønsker. 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

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 2.8: Interpolerte data på titrerkurven.

2.3.2 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*.

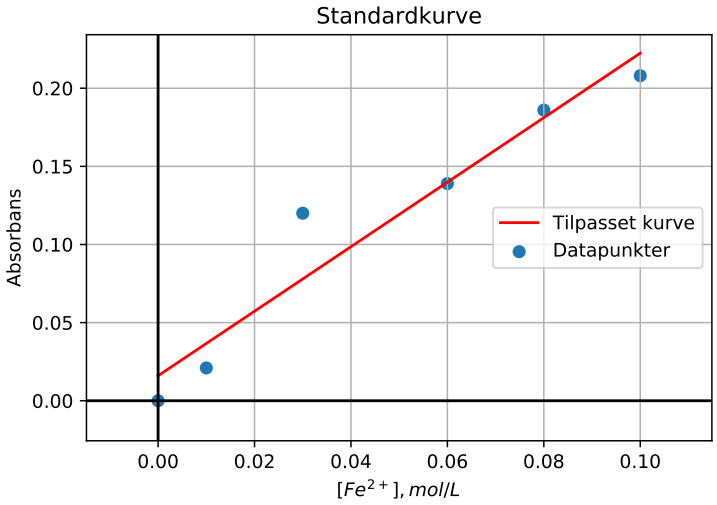
> **Regresjon**
>
> Regresjon er en prosess der vi tilnærmer diskrete data med en kontinuerlig funksjon

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. Denne funksjonen bruker *minste kvadraters metode* til å tilpasse funksjonsverdier til datapunktene på en måte som gir minst mulig *varians*. Vi bruker 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 2.9: 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):

```
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* på trender og utviklingen i et system.

Underveisoppgave 2.4

I oppgave 2.1 lagde du et plott av trykk i kPa som funksjon av temperatur i K ved å bruke følgende data:

$$\text{trykk} = [36, 46.4, 56.7, 67.1, 77.5, 88.0]$$
$$\text{temp} = [173, 223, 273, 323, 373, 423]$$

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:

Gjennomsnitt

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

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i \tag{2.1}$$

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:

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.

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2 \tag{2.2}$$

Underveisoppgave 2.5

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.

Standardavvik

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

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2} \tag{2.3}$$

Underveisoppgave 2.6

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 2.7

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:

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 2.3.2, kan vi gjøre slik:

```
from sklearn.metrics import r2_score

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

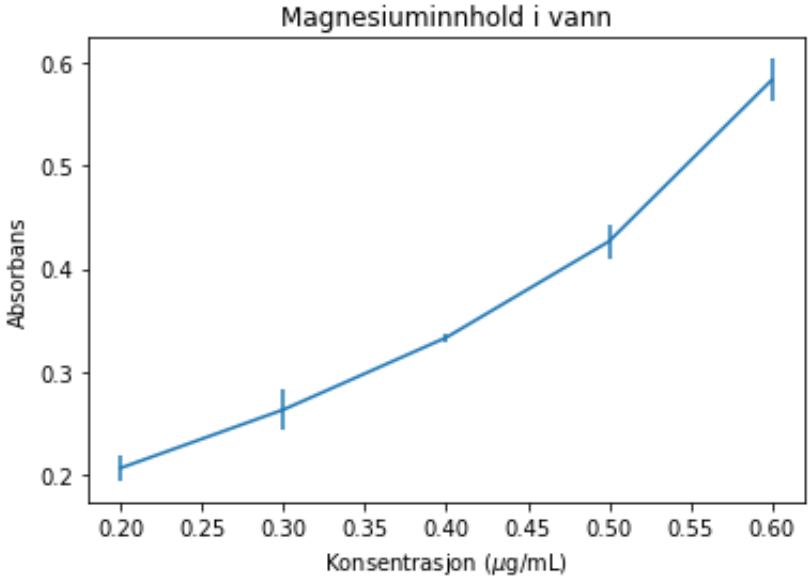
2.4.4 Grafisk framstilling av statistisk spredning

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:

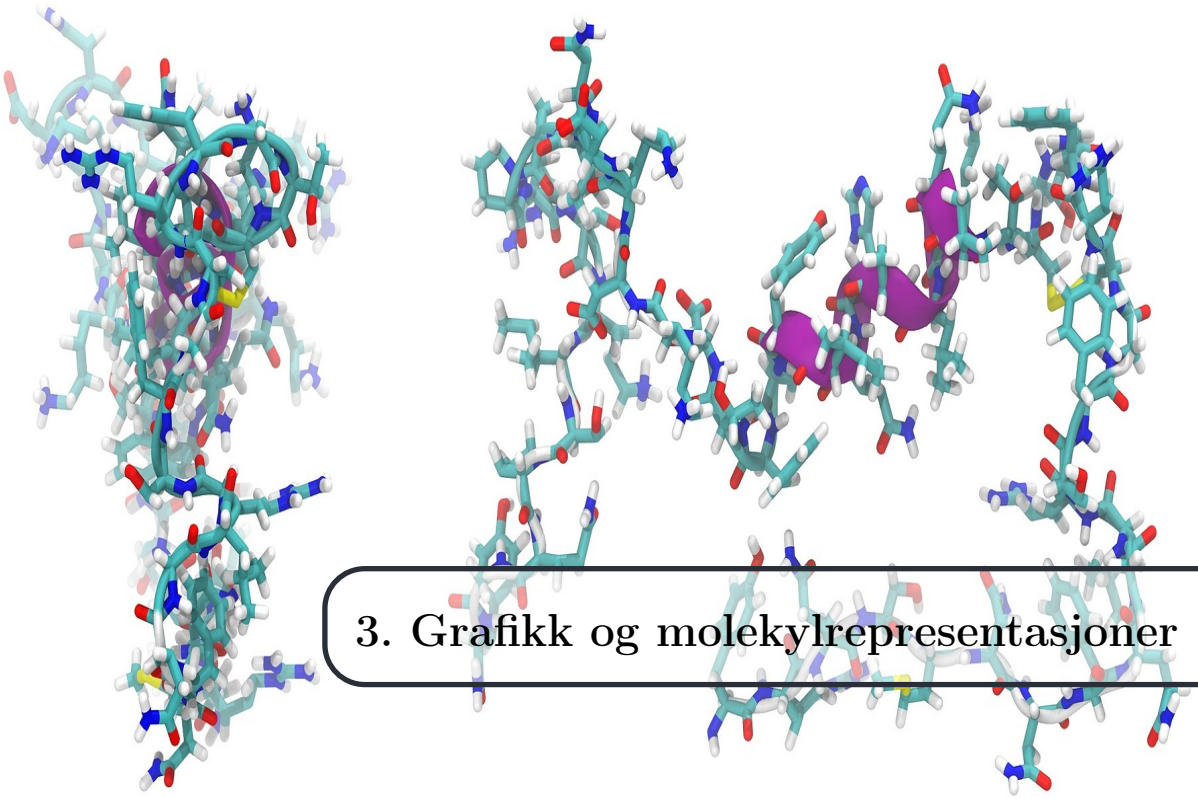
```
import numpy as np
import matplotlib.pyplot as plt

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 2.10: 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'`.



3. Grafikk og molekylrepresentasjoner

Når vi utfører en simulering, får vi en del tall som gir oss informasjon om systemet vi har simulert. I mange tilfeller kan det også være nyttig å kunne representere disse tallene grafisk. Noen ganger holder det med et plott, mens andre ganger kan det være nyttig med en visuell modell.

I dette kapitlet skal vi først se på hvordan vi kan visualisere molekylstrukturer, fra enkle uorganiske forbindelser til store enzymer. Deretter skal vi se på hvordan vi kan visualisere dynamiske simuleringer ved hjelp av kjemivisualiseringsbiblioteker som *py3Dmol* og *nglview*.

Mange av bibliotekene som benyttes til simulering og grafisk visualisering blir ofte mindre vedlikeholdt enn mye brukte biblioteker som numpy, scipy og matplotlib. Det betyr at de ikke alltid vil kunne fungere slik vi viser her. Vi har også prøvd å velge ut biblioteker som kan kjøres på Windows-systemer i tillegg til Linux. Det begrenser også utvalget litt, men vi nevner alternativer som du kan prøve ut dersom du ønsker. Hovedideen med kapitlet er at du blir kjent med hvordan du kan søke opp og benytte relevante biblioteker, og hvordan du kan bruke programmeringen du har lært til nå, til å bruke og forstå hvordan disse bibliotekene virker. Dette er en viktig og generell ferdighet.

3.1 Molekylvisualisering

Et nyttig bibliotek for å visualisere molekyler i Jupyter Notebook, er *Py3Dmol*. Her kan du hente opp molekyler fra to store databaser: PDB (Protein Data Bank) og PubChem. For å importere en struktur bruker du kommandoen `view` og setter parameteren `query` til enten `'cid'` for PubChem eller `'pdb'` for PDB. På PubChem kan du søke etter et molekyl og finne ID-en til dette molekylet (PubChem CID). I PDB starter alle søkeresultater med PDB-ID-en. I eksempelet nedenfor henter vi opp paracetamol fra PubChem og enzymet RNA-polymerase i initiell transkripsjon. Se om du kan finne de samme molekylene i de to databasene.

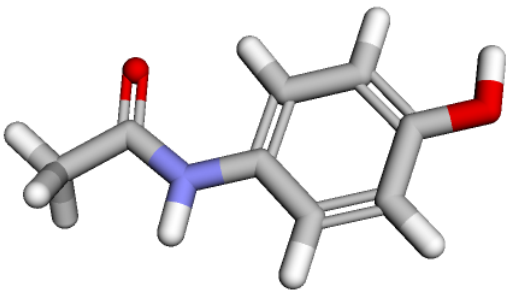
```
import py3Dmol

paracetamol = py3Dmol.view(query='cid:1983')
rna_polymerase = py3Dmol.view(query='pdb:5IYC')
```

Når vi har hentet strukturene, kan vi spesifisere hva slags stil vi ønsker på molekylene med `setStyle`. Som argument kan du benytte en dictionary med nøkkel `'line'`, `'cross'`, `'cartoon'`, `'stick'` eller `'sphere'`. Verdien til nøkkelen kan være nok en dictionary (nøstede dictionaries), med for eksempel `'color'` som en ny nøkkel. Fargen kan ha ulike verdier, men `'spectrum'` gir ofte fine resultater dersom du ønsker å bruke en 3D-modell og ikke bare strekformler.

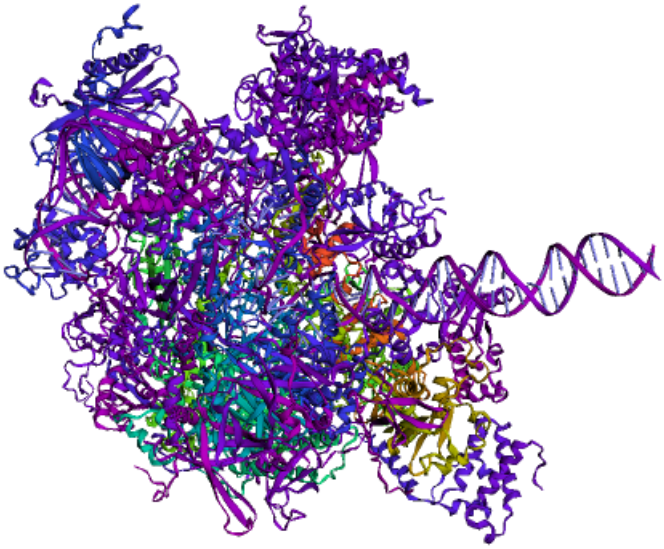
Når du har bestemt deg for alt dette, får du en figur du kan rotere og zoome inn og ut på.

```
paracetamol.setStyle({'stick': {'color':'spectrum'}})
rna_polymerase.setStyle({'cartoon': {'color':'spectrum'}})
```

Koden ovenfor gir følgende figurer:



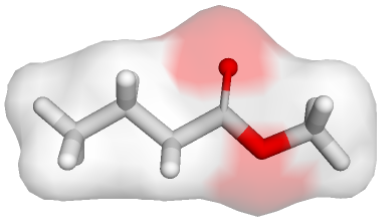
Figur 3.1: Paracetamol.



Figur 3.2: RNA-polymerase med DNA i starten av transkripsjonen.

Vi kan også beregne og tegne elektrontettheten (*potensialkart*) til ulike molekyler. Dette gjøres med `addSurface` som vist nedenfor. `'VDW'` står for van der Waals-krefter, som overflatemodellene tar utgangspunkt i. Du kan modifisere fargene og utseendet til overflaten etter behov. Et veldig vanlig fargespekter er en gradvis overgang (`'gradient'`) mellom rødt, hvitt og blått (`'rwb'`), der rødt er elektronrike områder, og blått er elektronfattige. Følgende kode viser elektrontettheten til esteren metylbutanat (søk: methyl butanoate i PubChem).

```
elektronkart = py3Dmol.view(query='cid:12180')
elektronkart.setStyle({'stick': {'color':'spectrum'}})
elektronkart.addSurface('VDW',{'opacity':0.6,
    'colorscheme':{'gradient':'rwb'}})
```



Figur 3.3: Potensialkart over metylbutanat.

3.2 Simuleringsgrafikk med *nglview*

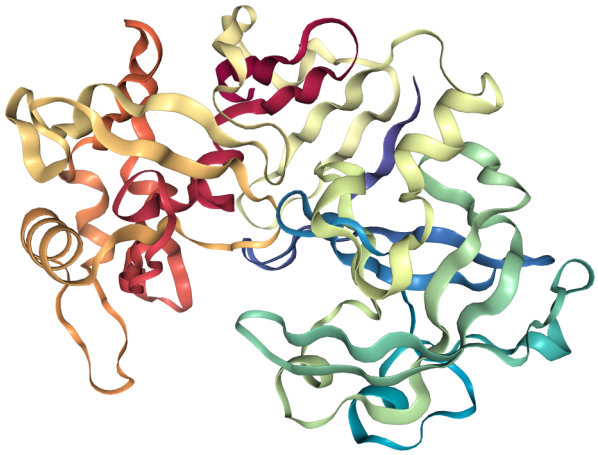
Biblioteket *nglview* er et mye brukt visualiseringsbibliotek som inngår i mange beregningsbiblioteker for kjemi. Det benyttes mest til visualisering og simulering av biomolekyler. For å installere dette biblioteket, er det enkleste å skrive følgende i kommandolinja i Linux-baserte operativsystemer eller i Anaconda Prompt i Windows:

```
pip install nglview
jupyter-nbextension enable nglview --py --sys-prefix
```

Når nglview er installert, kan du teste det ut i en Jupyter Notebook. Akkurat som py3Dmol fungerer kun nglview i Jupyter, og ikke i ordinære editorer. Vi kan hente opp molekyler fra hovedsakelig PDB og ParmEd (en farmasøytisk database), men kan også hente molekyler og beregninger fra andre Python-biblioteker, som vi skal se på seinere. Nedenfor har vi valgt å vise enzymet pepsin (kode 1PSN i PDB), som er et enzym i magen som bryter ned proteiner til mindre polypeptider.

```
import nglview as nv
enzym = nv.show_pdbid("1PSN")
enzym
```

Programmet ovenfor gir følgende bilde:

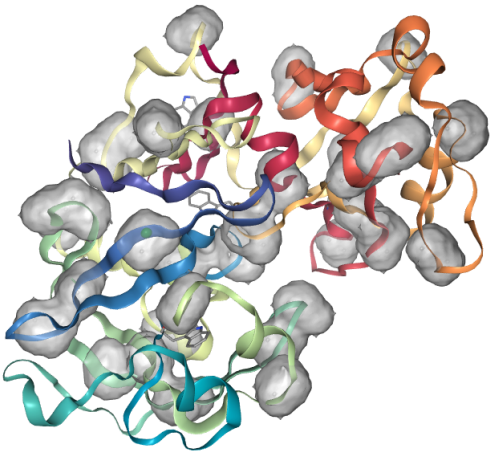


Figur 3.4: Enzymet pepsin.

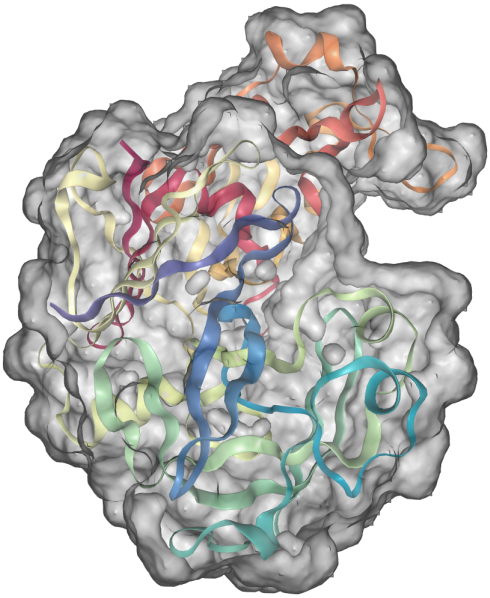
Vi kan også legge til overflater på enzymet med `add_surface`. Vi kan enten gi hele enzymet en overflate, eller vi kan gi overflater til bestemte aminosyrer.

```
enzym.add_surface(selection="GLY", opacity=0.3) # Kun glysin
enzym2.add_surface(selection="protein", opacity=0.3) # Hele proteinet
```

Dette gir følgende bilder:



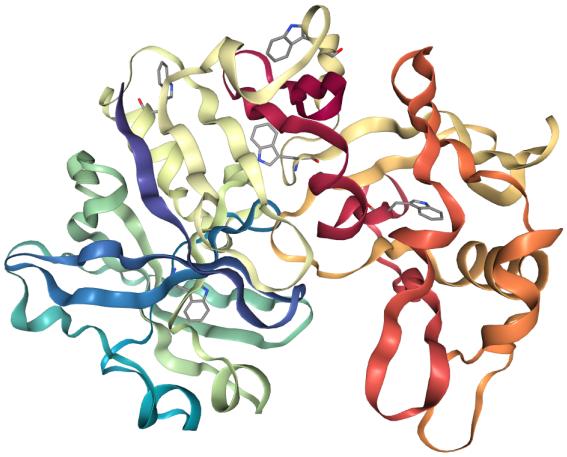
Figur 3.5: Enzymet pepsin med overflate på glysin.



Figur 3.6: Enzymet pepsin med heldekkende overflate.

Slik kan du lettere identifisere ulike aminosyrer i proteinet. Legg også merke til at du du får opp aminosyra du holder over med musepekeren. Vi kan i tillegg vise aminosyrene med pinnemodeller slik:

```
enzym.add_licorice('TRP') # Viser trypsin som pinnemodell
```



Figur 3.7: Enzymet pepsin med pinnemodell av trypsin.

Et statisk, høyoppløselig bilde kan du få ved å kjøre `enzym.render_image()` og `enzym._display_image()` i to separate celler under hverandre.

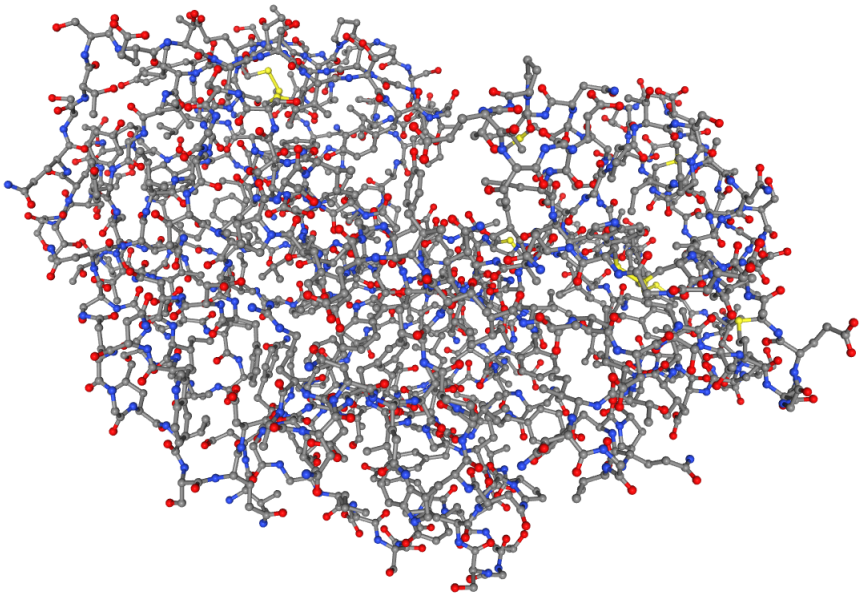
Underveisoppgave 3.1

Visualiser enzymet pepsin i en notebook og roter det slik at du finner det du tror er det aktive setet. Legg på en overflate med gjennomsiktighet. Er det lettere å finne det aktive setet nå?

Pepsin skilles ut som proenzymet *pepsinogen* fra magesekkcellene, men endrer struktur ved den lave pH-en i magesekken. Visualiser pepsinogen (3PSG) med gjennomsiktig overflate. Hva er forskjellen mellom strukturen til pepsinogen og pepsin? Tror du ut fra strukturen at dette proenzymet kan ha samme virkning som pepsin?

For å endre representasjonen fra «cartoon» (fargede bånd) til for eksempel kule-pinnerepresentasjon, kan du skrive følgende:

```
enzym.clear_representations() # Fjerner båndene
enzym.add_representation(repr_type='ball+stick', selection='protein')
```



Figur 3.8: Enzymet pepsin med kulepinnemodell av trypsin.

3.2.1 Simulering og nglview

Som nevnt kan nglview brukes til å visualisere simuleringer. Vi skal ikke gjøre simuleringene her, men heller vise raskt hvordan du kan lese filer fra simuleringer og visualisere disse. Vi tar utgangspunkt i *molekyldynamikksimuleringer* (MD) som er vanlig å bruke for å studere dynamikken mellom molekyler i blant annet biokjemi og materialkjemi. Med MD kan vi for eksempel studere hvordan transportproteiner fungerer over cellemembranen eller hvordan porestrukturen påvirker den kjemiske reaktiviteten til et materiale.

Tre gode biblioteker for å analysere MD-simuleringer er *pytraj* (kun Mac/-Linux), *mdtraj*, og *MDanalysis*. Vi viser eksempler på de to sistnevnte. Eksemplene er ment for å vise mulighetene med visualisering av simuleringer. Eksemplene som lastes inn er MD-filer som nglview har laget for demonstrasjonsformål. Slike filer må genereres i en MD-simulering.

```
import nglview as nv
import mdtraj as md

m_traj = md.load(nv.datafiles.TRR, top=nv.datafiles.PDB)
m_view = nv.show_mdtraj(m_traj)
m_view
```

```
import MDAnalysis as mda
from MDAnalysis.tests.datafiles import PSF, DCD

u = mda.Universe(PSF, DCD)
protein = u.select_atoms('protein')

traj = nv.show_mdanalysis(protein)
traj
```

Underveisoppgave 3.2

Prøv å kjøre programmene ovenfor (du må huske å installere mdtraj og MDAnalysis først). Klikk på play-knappen og se på simuleringsgrafikken. Hva tror du slike simuleringer kan brukes til?

4. Diskret modellering

Mange prosesser i naturen kan modelleres ved å formulere en sammenheng mellom et system ved tida t og tida $t + \Delta t$. Denne sammenhengen, eller *regelen*, er ofte en prosedyre som skal gjentas mange ganger. Et eksempel er en harepopulasjon som øker med en viss prosent i løpet av tida Δt. I matematikken har vi også en del sammenhenger der neste tall er avhengig av foregående tall, for eksempel tallfølger.

Siden vi kan ha å gjøre med svært mange ledd i slike prosesser, kan vi få bruk for løkker for å utforske disse prosessene. Siden prosessen inni ei løkke kalles for en *iterasjon*, sier vi at vi løser et problem *iterativt* når vi repeterer en prosess flere ganger.

Siden vi ikke kjenner tilstanden til systemet ved hvert tidssteg, har vi med en *diskret* modell å gjøre. Dette står i motsetning til *kontinuerlige* modeller, der vi kjenner tilstanden ved hvert eneste tidssteg. Egentlig er alle prosesser diskrete på en datamaskin, siden vi ikke kan ha uendelig små tidssteg. Likevel skiller vi på kontinuerlig og diskret på samme måte som i matematikken, selv om forskjellen på datamaskinen egentlig bare er størrelsen mellom verdiene.

I dette kapitlet skal vi se på hvordan vi kan utforske diskrete modeller av dynamiske systemer, og hvordan disse systemene kan beskrives og løses iterativt. Mange av problemene vi skal se på, kan løses analytisk. Men styrken

til metodene vi bruker her, er at de også kan benyttes på analytisk uløselige systemer. Du introduseres dermed til en løsningsstrategi som gjør deg i stand til å «tenke iterativt», altså tenke i stegvise gjentakende operasjoner, i tillegg til å løse problemer med formler og algebra.

4.1 Differenslikninger

Kontinuerlige modeller for systemer som utvikler seg med tid, beskrives ofte av *differensiallikninger* (se kapittel 7.1), mens diskrete modeller kan beskrives av *differenslikninger*. Et eksempel på sammenhenger som kan beskrives som differenslikninger, er mønsteret i tallfølger.

> **Differenslikninger**
>
> En differenslikning er en likning som beskriver forskjellen mellom etterfølgende verdier til en funksjon av diskrete variabler.

4.1.1 Følger og rekker

Ta som eksempel en aritmetisk følge der hvert ledd er 3 større enn det forrige leddet:

$$1,\ 4,\ 7,\ 10,\ 13,\ 16,\ 19,\ ...$$

Det neste (n+1-te) leddet x i denne følgen kan beskrives ut fra det forrige (n-te) leddet ved følgende differenslikning:

$$x_{n+1} = x_n + 3 \tag{4.1}$$

Indeksen n står her for det n-te elementet av x. Dersom vi følger Python-konvensjonen og starter på element nr. 0, blir element nr. 1 (det andre i følgen) lik:

$$x_1 = x_0 + 3 = 1 + 3 = 4 \tag{4.2}$$

Vi kan se at element k i følgen vil bli $x_k = x_0 + 3(n-1) = 1 + 3(n-1)$, men vi kan også finne element k iterativt:

```
x0 = 1
n = 4
x = x0

for k in range(1,n):
    x = x + 3

print(x)
```

Programmet ovenfor regner ut det fjerde tallet i følgen. Vi ser at programmet benytter er en relativt enkel framgangsmåte, men ofte er hovedutfordringen å formulere regelen som beskriver sammenhengen. Dessuten må vi passe på at vi bruker korrekte indekser.

Underveisoppgave 4.1

Finn det hundrede tallet i denne geometriske tallfølgen:

$$1,\ 2,\ 4,\ 8,\ 16,\ ...$$

Vi kan også summere alle leddene ved å innføre en summevariabel. Med andre ord kan vi finne *rekkesummer* ved hjelp av iterasjon. Ta for eksempel følgende geometriske rekke:

$$2 + 1 + 1/2 + 1/4 + 1/8 + 1/16 + ...$$

Vi kan formulere denne rekka som en sum av n ledd:

$$\sum_{k=0}^{n-1} 2 \cdot \left(\frac{1}{2}\right)^k \tag{4.3}$$

Slike summeformler er ikke alltid like enkle å formulere. Her ser vi at hvert ledd er halvparten av det andre. Derfor kan vi summere dem iterativt slik:

```
x0 = 2
n = 3
x = x0
s = x0

for k in range(1,n):
    x = x*0.5
    s = s + x

print(f'Summen av de {n} første leddene er {s}')
```

Underveisoppgave 4.2

Finn summen av de 100 første tallene i følgende rekke:

$$1 + \tfrac{2}{3} + \tfrac{4}{9} + \tfrac{8}{27} + ...$$

4.2 Biologisk halveringstid

På 1800-tallet gikk mange med hatt. Hatteindustrien blomstret, og en fant stadig opp nye metoder som forbedret hattematerialene. En særskilt god – og dårlig, skulle det vise seg – metode, var å bruke kvikksølvnitrat til å stive av hattebremmen i filthatter. Hattemakerne begynte etter hvert å oppleve symptomer som nummenhet, humørsvingninger, muskelspasmer og andre nevrologiske effekter. Noen har til og med blitt beskrevet som gale.

Figur 4.1: Kvikksølv i hatten var ikke så veldig lurt.

Heldigvis sluttet en å bruke kvikksølv i hattene, men hvor lang tid kan det ta før kvikksølvforgiftningen er ute av kroppen? Tida det tar før halvparten av et stoff mister sin biologiske effekt i kroppen, kalles *biologisk halveringstid*. Det er mange studier på den biologiske halveringstida til uorganisk kvikksølv i hjernen, med estimater fra noen uker til flere år, avhengig av hvordan det blir målt. Mye tyder på at et godt estimat kan være 27.4 år. Det betyr at massen kvikksølv m_{Hg} etter t år kan gis ved:

$$m_{Hg,t+1} = m_{Hg,t} \cdot 0.5^{t/H} \tag{4.4}$$

Dersom vi sløyfer t i eksponenten, men heller gjentar prosessen for hvert år, kan vi løse dette iterativt:

```
m0 = 1200 # Startmasse i mg
m = m0
tid = 100 # Tid i år
H = 27.4 # Halveringstid i år

for t in range(tid):
    m = m*0.5**(1/H)

print(f'Etter {tid} år er det {m:.0f} mg kvikksølv igjen i hjernen')
```

Siden vi har en formel for konsentrasjonen etter tida t, trenger vi strengt tatt ikke egentlig å bruke løkker her. Men ved å løse dette iterativt, kan vi lettere ta inn andre faktorer i modellen vår og studere effekten av dem. La oss si at vi etter 12 år finner en medisin som gradvis endrer halveringstida, eller at vi finner ut at noen næringsstoffer gir positiv effekt over tid. Dersom halveringstida endrer seg underveis, *må* vi faktisk benytte en iterativ modell, spesielt hvis den endrer seg uregelmessig. Her er et eksempel på en enkel modifikasjon:

```
m0 = 1200          # Startmasse i mg
m = m0
tid = 100          # Tid i år
H = 27.4           # Halveringstid i år
tid_medisin = 12 # Medisin inntatt ved år

for t in range(tid):
    if t >= tid_medisin:
        H = H*0.98
    m = m*0.5**(1/H)

print(f'Etter {tid} år er det {m:.0f} mg kvikksølv igjen i hjernen')
```

Underveisoppgave 4.3

Hva er effekten til medisinen i programmet ovenfor?

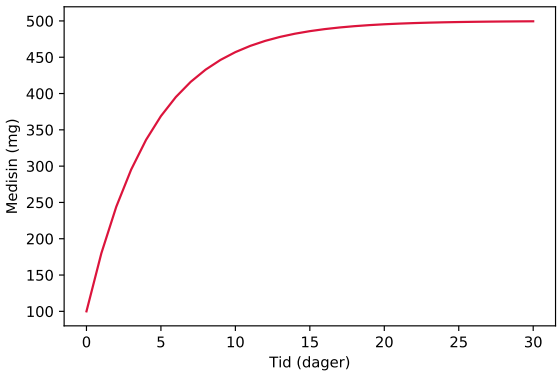
Den samme framgangsmåten kan brukes på for eksempel radioaktiv halveringstid og akkumulering av miljøgifter i en næringskjede. For radioaktive isotoper er halveringstida konstant, så her har vi ikke så mye igjen for å bruke programmering. Men med for eksempel miljøgifter kan vi dra nytte av at vi kan endre ulike spredningsfaktorer og dermed lage en mer fleksibel og dynamisk modell.

4.3 Medisinakkumulering

Vi kan også bruke iterative metoder til å estimere virkningen av stoffer med fysiologisk effekt, som kosttilskudd, medisiner og gift. Dersom for eksempel 20 % av en medisin brytes ned i kroppen hver dag, hvor mye medisin har vi i kroppen til enhver tid dersom vi tar 100 mg medisin hver morgen? Programmet nedenfor beregner total mengde medisin i kroppen i løpet av 30 dager:

```
import matplotlib.pyplot as plt

dose = 100
medisin = dose
nedbrytning = 0.80 # Andel medisin igjen i kroppen
tid = 30

t = []
total = []
t.append(0)
total.append(dose)

for i in range(1,tid+1):
    medisin = medisin*nedbrytning + dose
    total.append(medisin)
    t.append(i)

plt.plot(t,total,color='crimson')
plt.xlabel('Tid (dager)')
plt.ylabel('Medisin (mg)')
plt.show()
```

Programmet gir følgende plott:



Figur 4.2: Medisinakkumulering i kroppen over en måned.

Underveisoppgave 4.4

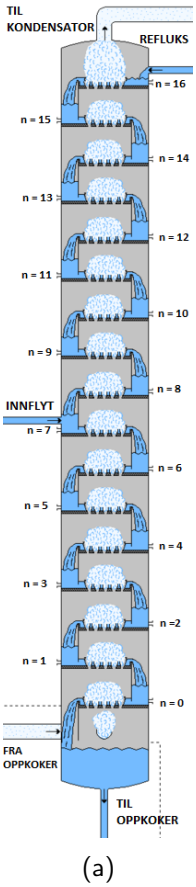
Bivirkninger av medisinen kan oppstå dersom vi får over 350 mg medisin i blodet. Modifiser programmet slik at det finner ut hvor lenge vi kan gå på medisinen dersom vi bruker 100 mg hver dag. Finn også ut hvor mye medisin vi kan ta hver dag uten at vi får bivirkninger.

4.4 Destillasjon

Vi kan beskrive destillasjonsprosesser ved hjelp av differenslikninger. Dette er nyttig for å kunne bestemme størrelse og seksjonering av destillasjonskolonner, spesielt i kjemisk industri.

I en destillasjonskolonne finnes det ulike destillasjonssoner der dampen til en forbindelse er i likevekt med væsken. Destillasjonssonene er merket med nummer på figuren til høyre. Vi kan beskrive antall mol av hvert stoff som blandingen består av, ved å bruke *molfraksjonen* x_i til forbindelse i. Den er definert som stoffmengden av i dividert på den totale stoffmengden til alle forbindelser i blandinga:

$$x_i = \frac{n_i}{n_{totalt}} \tag{4.5}$$



(a)

Vi har også et uttrykk for molfraksjonen til hvert stoff i ved hver destillasjonssone n i en kolonne. Vi betegner molfraksjonen av stoffene i gassfase som $x_{i,n}$ og stoffene i væskefase som $y_{i,n}$. I bunnen av kolonnen er molfraksjonene $y_{i,0}$ og $x_{i,n}$. I tillegg har vi en *refluksrate* R som beskriver andelen destillat som fraktes tilbake igjen til kolonnen. Likevektskonstanten ved hver kolonnesone beskriver forholdet mellom stoffene i væskefase og gassfase:

$$K_{n,i} = \frac{y_{i,n}}{x_{i,n}} \tag{4.6}$$

Vi får følgende uttrykk for molfraksjonen til en væske i sone n:

$$y_{i,n+1} = \frac{R}{R+1}x_{i,n} + \frac{R}{R+1}x_{i,0} \tag{4.7}$$

Når vi omformer 4.6 til et uttrykk for $y_{i,n}$ og kombinerer dette med 4.7, får vi:

$$x_{i,n+1} = \left(\frac{R}{R+1}x_{i,n} + \frac{R}{R+1}x_{i,0}\right) K_{n,i} \tag{4.8}$$

Dersom vi gjennom eksperimenter og/eller beregninger har informasjon om R og K, kan vi estimere molfraksjonen til væske og damp ved hver destillasjonssone. Fordelen med å bruke en simulering her, er at det er enkelt å innføre flere stoffer og eksperimentere med ulike temperaturer og forskjellig antall destillasjonssoner.

Underveisoppgave 4.5

Lag et program som finner molfraksjonen til væske og gass for 30 destillasjonssoner dersom vi kun har ett stoff i kolonnen. Bruk $R = 0.25$, $x_{i,0} = 0.40$ og $y_{i,0} = 0.60$. Anta også at $K = 0.05n$, altså at K øker oppover i kolonnen.

4.5 Rekursjon*

Vi kan beskrive fakultetet av et tall, $n!$, som en differenslikning:

$$x_n = nx_{n-1}, \qquad x_0 = 1 \tag{4.9}$$

Dersom vi for eksempel ønsker å regne ut 4!, kan vi regne ut dette slik: $x_4 = 4(4-1)(4-2)(4-3) = 4 \cdot 3 \cdot 2 \cdot 1 = 24$.

Underveisoppgave 4.6

Lag en Python-funksjon som tar et tall n som parameter, og som regner ut $n!$. Kontroller funksjonen din ved å sammenlikne med numpy-funksjonen *factorial* (den brukes slik: `np.math.factorial(n)`).

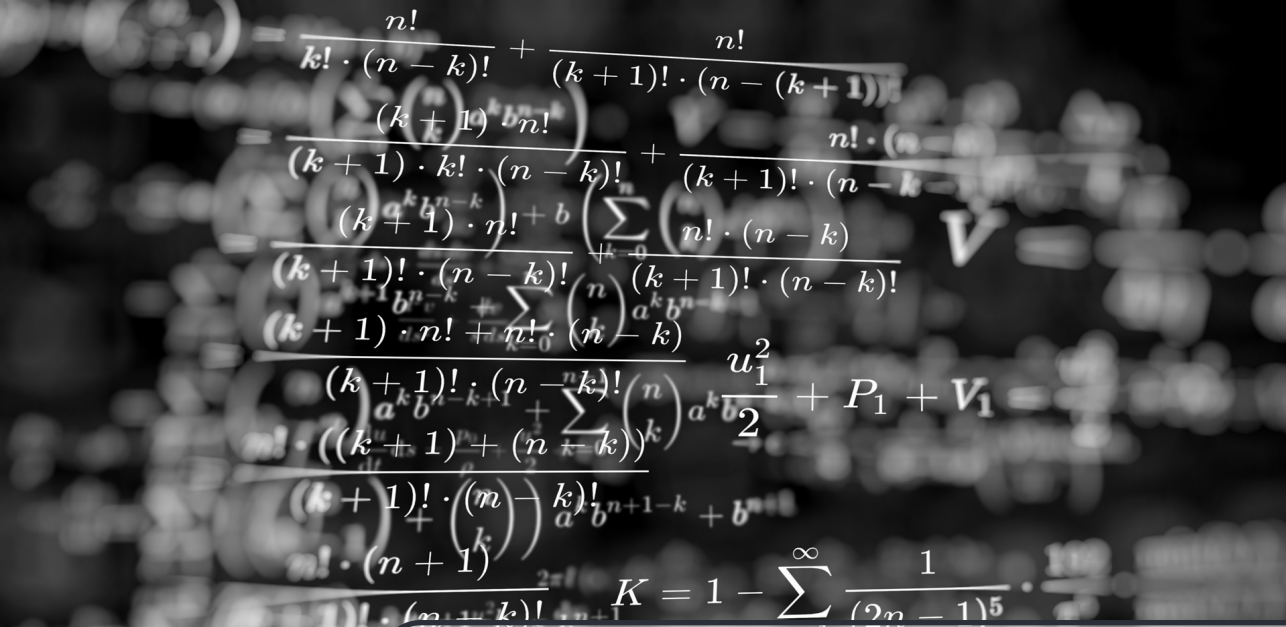
En annen måte å beregne $n!$ på, er å benytte *rekursive funksjoner*. I rekursive funksjoner returneres en annen funksjonsverdi av *samme* funksjon. På en måte kan funksjonen ses på som en funksjon av seg selv (filmen Inception forklarer konseptet nærmere). Her er et eksempel:

```
def fakultet(n):
    if n == 1:
        return n
    else:
        return n*fakultet(n-1)
```

Vi ser at det ikke benyttes løkker i det hele tatt, og det er fordi funksjonen kalles for flere verdier av n etter hverandre. Det er viktig at vi hver gang sjekker om $n = 1$, ellers stopper ikke funksjonen. Da ender vi opp med en rekursjonsfeil: `RecursionError: maximum recursion depth exceeded`.

Underveisoppgave 4.7

Studer programmet ovenfor og skriv ned på et papir hva som skjer trinnvis dersom vi kaller på `fakultet(4)`.



5. Numerisk løsing av likninger

Likninger dukker opp i svært mange sammenhenger. De er derimot ikke alltid analytisk løsbare. Dessuten må vi baserer oss på spesialregler for ulike typer likninger for å løse dem. Andregradsformelen fungerer bare på andregradslikninger, og vi må ha en egen formel for polynomer av tredje grad. Polynomer med grad høyere enn 4 er dessuten ikke analytisk løsbare, ifølge *Abel-Ruffini-teoremet*.

Numeriske metoder bygger derimot på generaliserbare prinsipper, og kan i utgangspunktet løse alle likninger med en gitt presisjon. Vi skal i dette kapitlet se på hvordan vi kan bruke numeriske metoder til å løse likninger ved å finne nullpunktene til funksjoner.

5.1 Løsing av likninger

I mange situasjoner ønsker vi å finne nullpunktene til en funksjon. Dette er det samme som å løse en likning $f(x) = 0$. Dersom vi for eksempel ønsker å løse en likning $x^4 + 3x = 2x^2 - 10$, kan vi løse denne ved å finne nullpunktet til funksjonen $f(x) = x^4 + 3x - 2x^2 + 10$. Vi kan si at vi formulerer likningen som et *nullpunktsproblem*.

Det finnes mange ulike måter å løse slike likninger på, og hver av metodene har sine styrker og svakheter. To metoder som er basert på relativt enkle prin-

sipper, er *halveringsmetoden* og *Newtons metode*. Vi skal se på implementering av disse metodene. I tillegg ser vi litt på styrker og svakheter ved metodene. Til slutt ser vi hvordan vi kan løse likninger ved å bruke ferdige algoritmer som finnes i *scipy*-biblioteket.

5.1.1 Kjemisk problemstilling

Likninger kan løse mange problemer, også de av kjemisk art. Hvis vi for eksempel skal finne ut hvor to funksjoner skjærer hverandre, kan vi løse likningen $f(x) = g(x)$ som et nullpunktsproblem: $f(x) - g(x) = 0$.

La oss si at vi har to uttrykk som for eksempel beskriver konsentrasjon til to produkter over tid. Disse uttrykkene trenger ikke å være enkle, og da får vi problemer med å løse dem analytisk. Vi tar utgangspunkt i følgende sammenhenger:

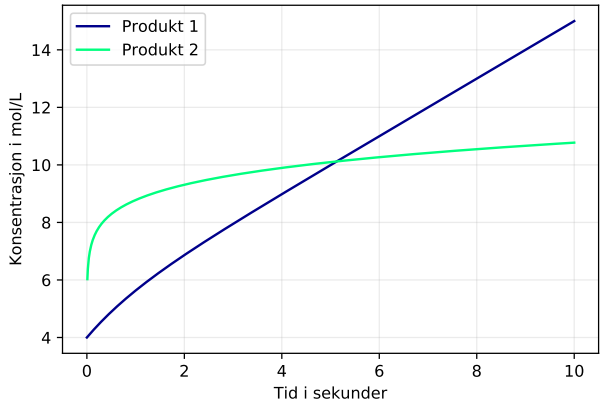
$$c_1(t) = e^{-t} + t + 5 \tag{5.1}$$

$$c_2(t) = \ln(0.006t + 1) + t^{0.3} + 10 \tag{5.2}$$

For å finne ut ved hvilken tid de to produktene har lik konsentrasjon, kan vi løse likningen $c_1(t) = c_2(t)$. Formulert som et nullpunktsproblem får vi:

$$e^{-t} + t + 5 - \ln(0.006t + 1) - t^{0.3} - 10 = 0 \tag{5.3}$$

Grafen nedenfor viser at skjæringen skjer etter omtrent fem sekunder, men vi ser av uttrykkene at dette ikke lar seg løses ved hjelp av konvensjonelle metoder (du kan jo prøve!). La oss derfor se på hvordan vi kan løse dette numerisk.

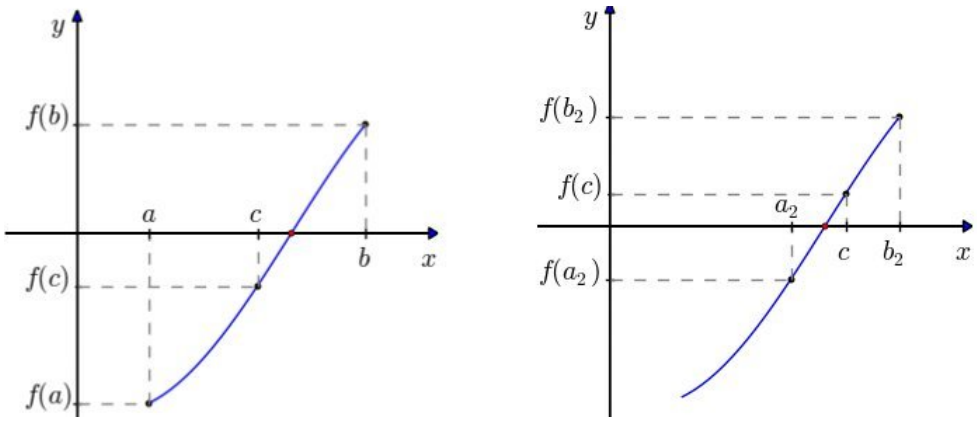


Figur 5.1: Reaksjonsforløpet til to ulike reaksjoner.

5.2 Halveringsmetoden

Halveringsmetoden går ut på å velge et intervall $[a, b]$ der $f(a)$ og $f(b)$ har motsatte fortegn. Intervallet velges i utgangspunktet tilfeldig, men vi kan ofte bruke grafen til å bestemme disse dersom vi plotter den først. Deretter skal vi finne et nytt intervall $[a_1, b_1]$ som er mindre, men slik at $f(a_1)$ og $f(b_1)$ fortsatt har motsatte fortegn. Det kan vi gjøre ved å finne midten mellom a og b. Da får vi et punkt $c_1 = (a + b)/2$, og vi kan finne $f(c_1)$.

Vi undersøker så om $f(c_1) \approx 0$. Hvis ikke evaluerer vi fortegnene til $f(a)$, $f(b)$ og $f(c_1)$. Dersom $f(a)$ og $f(c_1)$ har samme fortegn, setter vi det nye intervallet til $[c_1, b]$ fordi da må $f(c_1)$ og $f(b)$ ha motsatte fortegn. Motsatt setter vi intervallet til $[a, c_1]$ dersom $f(b)$ og $f(c_1)$ har samme fortegn. Prosessen gjentas n ganger til vi har at $f(c_n) \approx 0$. Figuren nedenfor illustrerer metoden med to trinn.



Figur 5.2: Halveringsmetoden i to trinn.

Numerisk metode: Halveringsmetoden

La f være en kontinuerlig funksjon med motsatte fortegn på funksjonsverdiene $f(a)$ og $f(b)$ i intervallet $[a, b]$. Da kan nullpunktene finnes slik:

1. Finn midtpunktet c_k mellom punktene a og b.
2. Undersøker hvilke av $f(a)$ og $f(b)$ som har motsatt fortegn til $f(c_k)$, og sett det nye intervallet til $[a, c_k]$ eller $[c_k, b]$, der start- og sluttverdien i intervallet skal ha motsatt fortegn.
3. Gjenta prosessen n ganger til $f(c_k) \approx 0$.

Underveisoppgave 5.1

Lag et program der du implementerer algoritmen ovenfor før du ser på løsningsforslaget på neste side.

Det er en årsak til at vi sier $f(c_k) \approx 0$ og ikke $f(c_k) = 0$. Numeriske verdier blir sjelden eksakte, og derfor blir de sjelden helt lik 0. Vi må derfor angi en verdi som er svært nærme null, det vil si «0 nok» til at vi er fornøyd med svaret. Vi kan implementere metoden slik i Python:

```
# Definerer funksjonen
def f(x):
    return x**2 - x -2

# Halveringsmetoden
def halveringsmetoden(f,a,b,tol=1E-10,n=100):
    """
    Algoritme som benytter halveringsmetoden til å
    finne nullpunktet til en funksjon f i intervallet [a,b].

    Parametre
    ---------
    f: funksjonen
    a: laveste x-verdi i intervallet
    b: høyeste x-verdi i intervallet
    tol: toleranseparameter/feilmargin
    n: maks antall iterasjoner
    """
    i = 0
    c = (a+b)/2       # Finner første halveringspunkt
    while i < n and abs(f(c)) > tol:
        if f(a)*f(c) < 0:
            b = c
        elif f(b)*f(c) < 0:
            a = c
        c = (a+b)/2
        i += 1
    return c, i

c, i = halveringsmetoden(f,0,4)
print(f'Nullpunktet er x = {c}, og løkka gikk {i} ganger.')
```

Vi har definert halveringsmetoden som en funksjon her. Det er lurt å strukturere numeriske metoder i funksjoner fordi det gir bedre struktur og bedre mulighet for gjenbruk. Men hvis du vil, kan du starte med å lage den uten funksjoner. Det beste er ofte å implementere de numeriske metodene så enkelt

som mulig først, for deretter å gjøre metodene og koden mer robust.

I programmet ovenfor har vi også lagt inn en docstring med forklaringstekst i funksjonen. I tillegg bruker vi både en toleranseparameter *tol* og en parameter n for maks antall iterasjoner. Toleranseparameteren angir hvor nærme null vi trenger å være, og vi tar absoluttverdien av $f(c)$ fordi funksjonsverdien kan nærme seg null fra begge sider av x-aksen.

Både midtpunktet c og antall iterasjoner vi brukte, i, returneres av funksjonen. Det er derimot et par fallgruver her, blant annet hvis $f(a)$ og $f(b)$ har like fortegn, eller hvis toleransen ikke nås med n iterasjoner. I oppgava nedenfor skal du adressere disse problemene.

Underveisoppgave 5.2

Programmet ovenfor kan gjøres enda mer robust. Modifiser det slik at:

1. Dersom vi når maks iterasjoner, skal funksjonen gi en passende feilmelding.
2. Dersom $f(a)$ og $f(b)$ har likt fortegn til å begynne med, skal funksjonen gi en passende feilmelding.

Dersom vi kjører programmet ovenfor, får vi at $x = 2.0$. Nøyaktigheten avhenger av feilen og antall iterasjoner. Svaret er lett å sjekke analytisk med andregradsformelen i dette tilfellet, men da får vi også en løsning til: $x = -1.0$. Vi ser nemlig at funksjonen har to nullpukter dersom vi plotter grafen. Det kan altså lønne seg å plotte grafen før vi bestemmer hva slags intervall vi skal lete etter nullpunktet i. Dersom vi justerer intervallet til for eksempel $[-5, 0]$, vil vi finne det siste nullpunktet (prøv det!).

Underveisoppgave 5.3

Bruk halveringsmetoden og vis at løsningene til likningen $x^5 = 5x^3 + 3$ er $x_1 \approx -2.169, x_2 \approx -0.894$ og $x_3 \approx 2.291$.

La oss se om vi kan benytte denne metoden til å finne ut ved hvilken tid konsentrasjonen er lik i reaksjonene representert ved likning 5.1 og 5.2:

```
import numpy as np

def c1(t):
    return np.exp(-t) + t + 5

def c2(t):
    return np.log(0.006*t + 1) + t**0.3 + 10
```

```
def c_int(t):
    return c1(t) - c2(t)

t, i = halveringsmetoden(c_int,1,100)
c = c1(t)
print(f'Konsentrasjonen var {c:.2f} for begge reaksjoner etter
    {t:.2f} sekunder')
```

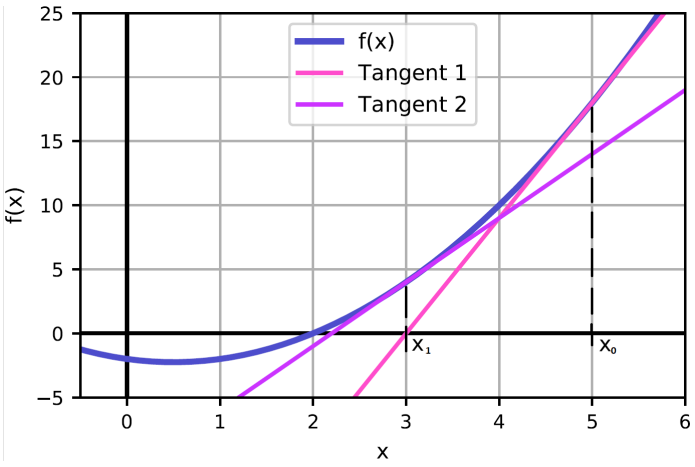
Underveisoppgave 5.4

Test og studer programmet ovenfor og forklar hva som skjer på hver linje.

5.2.1 Newtons metode

Det finnes flere måter å løse likninger på. En metode som ofte fungerer godt, er *Newtons metode* (også kalt *Newton-Raphsons metode*). Den bruker nullpunktet til *tangenten* i et punkt på en funksjon f som en tilnærming til nullpunktet til f.

Vi velger oss først et startpunkt x_0 og regner ut nullpunktet til tangenten i $f(x_0)$ og kaller punktet x_1. Dersom det tilsvarer nullpunktet til funksjonen f, sier vi oss ferdige. Hvis ikke regner vi ut nullpunktet til en ny tangent i $f(x_1)$ og kaller det x_2. Slik fortsetter vi til vi er så nært nullpunktet til f som vi ønsker. Figuren nedenfor viser to iterasjoner av metoden med funksjonen $f(x) = x^2 - x - 2$ og startgjett $x_0 = 5$:



Figur 5.3: Newtons metode i to trinn.

La oss utlede en algoritme for metoden. Stigningen til en linje kan gis ved:

$$a = \frac{\Delta y}{\Delta x} = \frac{f(x) - f(x_0)}{x - x_0} \tag{5.4}$$

Her setter vi x til å være en variabel størrelse og x_0 til et fast punkt. Dersom vi gjør om dette til et uttrykk for linja $f(x)$, får vi følgende:

$$f(x) = f(x_0) + a(x - x_0) \tag{5.5}$$

Dette er likningen for ei linje som går igjennom punktet $(x_0, f(x_0))$. Siden den deriverte i et punkt er lik stigningen i det samme punktet, kan vi skrive:

$$f(x) = f(x_0) + f'(x_0)(x - x_0) \tag{5.6}$$

som er likningen til en tangent gjennom punktet $(x_0, f(x_0))$ på funksjonen f. Vi ønsker å finne nullpunktet til tangenten, og setter derfor $f(x) = 0$:

$$f(x_0) + f'(x_0)(x - x_0) = 0 \tag{5.7}$$

Dersom vi omformer uttrykket slik at den ukjente er nullpunktet x, får vi:

$$x = x_0 - \frac{f(x_0)}{f'(x_0)} \tag{5.8}$$

Dette er Newtons metode. Siden vi ønsker å gjenta prosessen med å finne nullpunktene til tangentene i $(x_0, f(x_0))$, $(x_1, f(x_1))$, $(x_2, f(x_2))$ og så videre, kan vi formulere algoritmen iterativt:

Numerisk metode: Newtons metode

La f være en kontinuerlig, deriverbar funksjon, og la $(x_n, f(x_n))$ være et punkt på funksjonen. Nullpunktet til funksjonen f kan tilnærmes ved nullpunktet til den n-te tangenten til funksjonen. Bruk først et startgjett, x_0. Finn så nullpunktet x_{n+1} til den n-te tangenten ved å bruke nullpunktet, x_n, til den forrige tangenten. Gjenta til $f(x_{n+1}) \approx 0$

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{5.9}$$

Underveisoppgave 5.5

Implementer metoden ovenfor som en Python-funksjon. Test funksjonen på likningen $x^2 = x^3 - 4$.

Nå løser vi samme problem som med halveringsmetoden. Merk at vi her benytter en veldig enkel utgave av Newtons metode for å illustrere poenget.

```
def newtons_metode(f,fder,x,tol=1E-10):
    while abs(f(x)) > tol:
        x = x - f(x)/fder(x)
    return x

def c1(t):
    return np.exp(-t) + t + 5
def c2(t):
    return np.log(0.006*t+1) + t**0.3 + 10

def c1_der(t):
    return -np.exp(-t) + 1
def c2_der(t):
    return 1/(t*np.log(10)) + 0.3*t**(-0.7)

def C(t):
    return c1(t) - c2(t)
def C_der(t):
    return c1_der(t) - c2_der(t)
```

Underveisoppgave 5.6

Forbedre funksjonen `newtons_metode` ovenfor ved å legge til:

1. Maks antall iterasjoner som er tillatt.
2. Feilmelding når maks antall iterasjoner nås før toleransen.
3. Docstring som forklarer parametre og hva funksjonen gjør.

En fordel med denne metoden er at den konvergerer relativt raskt mot nullpunktet. Det kreves derfor som regel færre iterasjoner å finne nullpunktet enn når vi bruker halveringmetoden. Dette er en stor fordel når vi skal gjøre tunge beregninger og løse mange likninger. I tillegg krever Newtons metode kun ett startgjett, ikke et intervall. Spesielt når vi løser mange likninger etter hverandre, kan det være vanskelig å automatisere valget av intervaller. Da er det nyttig å bare ta utgangspunkt i ett punkt.

Newtons metode støter derimot på en del problemer der halveringsmetoden klarer seg fint. For det første trenger vi den deriverte av funksjonen. Stort sett er ikke dette problematisk, men i noen tilfeller kan det være det. Da kan vi ty til den numerisk deriverte (se kapittel 6), men det kan være en ulempe siden vi får nok et feilledd i utregningen vår.

For det andre kan vi få et problem ved lokale ekstremalpunkter. Her kan metoden gi tangenter på hver sin side av for eksempel bunnpunktet, uten at de konvergerer mot nullpunktet.

Vi kan for så vidt også få et problem dersom $f'(x) = 0$ et eller annet sted i iterasjonen, fordi vi da må dele på null i algoritmen. Dette oppstår ikke så ofte, men det kan være lurt å tenke på, spesielt med hensyn til startgjett.

Både Newtons metode og halveringsmetoden er grunnleggende, men relativt robuste metoder som kan gi gode nok tilnærminger i mange tilfeller. Newtons metode er en såkalt *Householder-metoder* av første orden. Vi kan få enda bedre tilnærminger ved å bruke metoder av høyere orden n. Med $n = 2$ får vi en algoritme som kalles *Halleys metode*. Ulempen med slike metoder er at vi trenger den $n-$te-deriverte av funksjonen. Fordelen er at de trenger svært få iterasjoner for å oppnå veldig gode tilnærminger. Scipy-biblioteket inneholder en del metoder som kan benyttes gjennom funksjonen `root_scalar`.

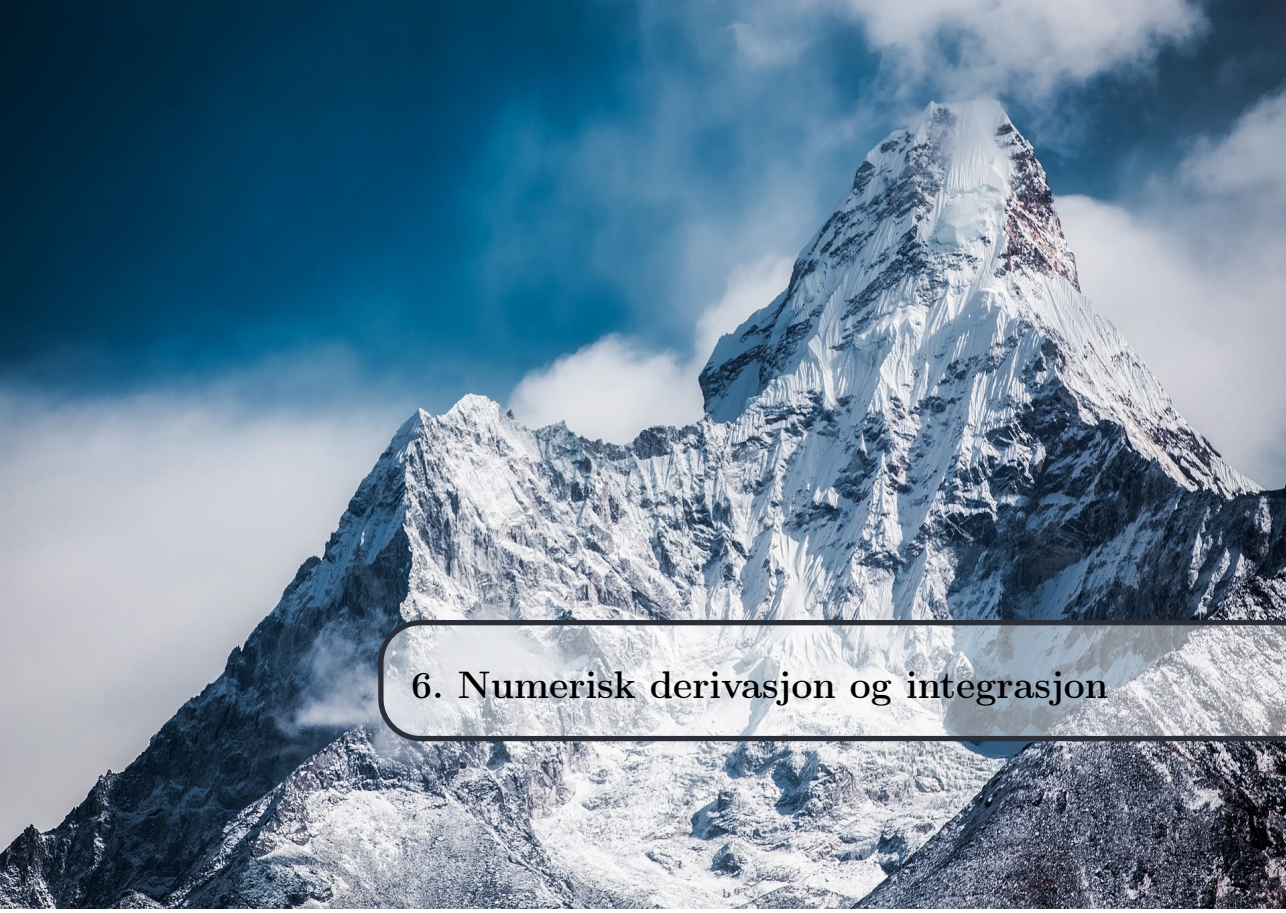
5.3 Numeriske biblioteker

Et mye brukt bibliotek for numeriske metoder er *Scipy* (forkortelse for *Scientific Python*.

```
from scipy.optimize import root_scalar
import numpy as np

def f(x):
    return x**3 - 1
def dfdx(x):
    return 3*x**2
def df2dx2(x):
    return 6*x

# Nullpunkter
nullpunkt_halverings = root_scalar(f,method='bisect',bracket=[0,5])
nullpunkt_newton = root_scalar(f,method='newton',fprime=dfdx,x0=5)
nullpunkt_halley = root_scalar(f,method='halley',fprime=dfdx,
    fprime2=df2dx2,x0=5)
print("Halveringsmetoden:",nullpunkt_halverings)
print("Newtons metode:",nullpunkt_newton)
print("Halleys metode:",nullpunkt_halley)
```

Underveisoppgave 5.7

Kjør programmet ovenfor og forklar hvordan funksjonen `root_scalar` fungerer. Prøv å endre parametrene og test funksjonen på andre problemer med ulike metoder.



6. Numerisk derivasjon og integrasjon

Derivasjon handler om endring. Den deriverte beskriver stigning og forandring. Mer presist beskriver den deriverte *momentan endring*, altså endringen mellom to tilstander som er så nære hverandre (typisk i tid) som overhodet mulig. Vi har derfor nytte av derivasjon i mange tilfeller der vi ønsker å beskrive en utvikling, og det er ikke sjeldent!

Du er antakelig kjent med både derivasjon og integrasjon fra før. Eller, rettere sagt, *analytisk* derivasjon og integrasjon. Vi kan utlede og bruke ulike regler for hvordan vi kan derivere og integrere et uttrykk eller en funksjon. Men slike regler kan ikke brukes dersom vi har diskrete punkter istedenfor kontinuerlige funksjoner. Og når vi gjør eksperimenter, får vi jo nettopp enkeltstående datapunkter. Da må vi ty til numeriske metoder.

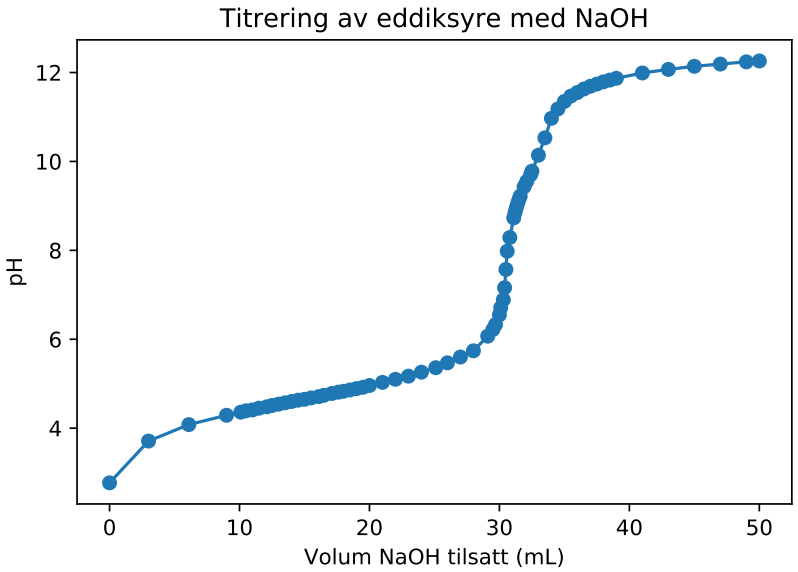
I dette kapitlet skal vi først se på hvordan vi kan bruke numerisk derivasjon til å beskrive endringen i eksperimentelle data. Deretter ser vi på hvordan vi kan utføre den motsatte operasjonen av derivasjon, nemlig integrasjon, ved hjelp av numeriske metoder.

6.1 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 6.1 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 6.1: Titrerkurve fra figur 2.5

Siden vi ser 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.

6.1.1 Newtons kvotient

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

$$f'(x) = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x} \tag{6.1}$$

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

Numerisk metode: Numerisk derivasjon (framoverdifferansen)

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

$$f'(x) \approx \frac{f(x + \Delta x) - f(x)}{\Delta x} \tag{6.2}$$

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 derivasjonsfunksjon
   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 6.1

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

6.1.2 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 fder_analytisk(x):
    return 4*x + 1

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

punkt = 1
delta_x = [10**-i for i in range(1,18)]
fder = fder_analytisk(punkt)

for i in range(len(delta_x)):
    print("For delta_x =", delta_x[i],"er feilen:",
          abs(derivert(f,punkt,delta_x[i])-fder))
```

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 er feilen: 0.20000000000000462
For delta_x = 0.01 er feilen: 0.02000000000024443
For delta_x = 0.001 er feilen: 0.001999999999284796
For delta_x = 0.0001 er feilen: 0.00019999999601338914
For delta_x = 1e-05 er feilen: 2.000003441082754e-05
For delta_x = 1e-06 er feilen: 1.9999885125798755e-06
For delta_x = 1e-07 er feilen: 2.005390342674218e-07
For delta_x = 1e-08 er feilen: 3.038735485461075e-08
For delta_x = 1e-09 er feilen: 4.1370185499545187e-07
For delta_x = 1e-10 er feilen: 4.1370185499545187e-07
For delta_x = 1e-11 er feilen: 4.1370185499545187e-07
For delta_x = 1e-12 er feilen: 0.00044450291170505807
For delta_x = 1e-13 er feilen: 0.003996389186795568
For delta_x = 1e-14 er feilen: 0.0262008496792987
For delta_x = 1e-15 er feilen: 0.3290705182007514
For delta_x = 1e-16 er feilen: 5.0
```

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 den deriverte med 10^{-16} null som svar! Dette gir naturlig nok et avvik på 5 fra den analytiske verdien.

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. Dersom vi gjør en mer generell feilanalyse, viser det seg at 10^{-8} er en god verdi å velge her.

Legg merke til at vi ofte bruker symbolet h istedenfor Δx for enkelhets skyld. Dessuten er det veldig vanlig å bruke notasjonen $\frac{df}{dx}$, som er det samme som $f'(x)$. Vi skal benytte disse notasjonene en del videre i boka.

6.1.3 Ulike tilnærminger

I tilnærmingen ovenfor tar vi utgangspunkt i definisjonen av den deriverte. Den baserer seg på $\Delta y = f(x + \Delta x) - f(x)$. Dette kaller vi *framoverdifferansen* fordi den tar utgangspunkt i «verdien i neste punkt minus verdien i nåværende punkt». Vi kan tilnærme denne verdien på flere måter. Dersom vi istedenfor sier at $\Delta y = f(x) - f(x - \Delta x)$, tar vi utgangspunkt i «verdien i nåværende punkt minus verdien i forrige punkt». Da får vi *bakoverdifferansen*:

Numerisk metode: Numerisk derivasjon (bakoverdifferansen)

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

$$\frac{df}{dx} \approx \frac{f(x) - f(x - h)}{h} \tag{6.3}$$

Dersom vi kombinerer disse to metodene, får vi *sentraldifferansen*. Da tar vi utgangspunkt i «gjennomsnittet av verdien i neste punkt og verdien i forrige punkt»:

Numerisk metode: Numerisk derivasjon (sentraldifferansen)

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

$$\frac{df}{dx} \approx \frac{f(x + h) - f(x - h)}{2h} \tag{6.4}$$

Underveisoppgave 6.2

Gjør en feilanalyse med de tre ulike tilnærmingene for ulike verdier av Δx. Bruk funksjonen $f(x) = \sin(x)$ og sammenlikn med den analytiske verdien av den deriverte, $f'(x) = \cos(x)$

En nyttig notasjon å introdusere her, er bruken av *operatorer*. Operatorer er mye brukt i blant annet kvantekjemi, og det er greit å stifte bekjentskap med notasjonen. Vi har akkurat sett på en operator, nemlig differensialoperatoren $\frac{d}{dx}$. Den betyr at det skal utføres derivasjon på noe. Hvis vi skriver $\frac{d}{dx}f = \frac{df}{dx}$, betyr det at funksjonen f skal deriveres med hensyn på x. Tilsvarende har vi for eksempel integraloperatoren $\int dx$.

Operator

En operator i matematikken er en funksjon eller regel som utfører en operasjon på ett eller flere elementer og gir én eller flere verdier basert på dette.

Ved å innføre operatorer kan vi forkorte skrivemåten litt når vi snakker om ulike tilnærminger. For numerisk derivasjon kan vi innføre operatorene D^+, D^- og D:

Tilnærming	Operasjon	Operator
Framoverdifferanse	$\frac{df}{dx} \approx \frac{f(x+h)-f(x)}{h}$	$\mathbf{D}^+$
Bakoverdifferanse	$\frac{df}{dx} \approx \frac{f(x)-f(x-h)}{h}$	$\mathbf{D}^-$
Sentraldifferanse	$\frac{df}{dx} \approx \frac{f(x+h)-f(x-h)}{2h}$	$\mathbf{D}$

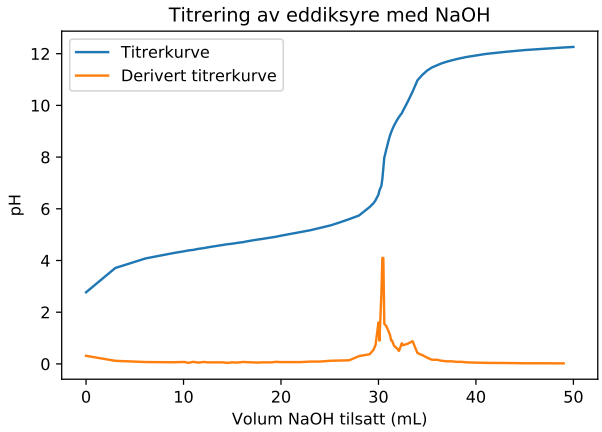
Framover- og bakoverdifferansen gir ofte samme feil, mens sentraldifferansen ofte gir en bedre tilnærming enn de to andre. Likevel kan framoverdifferansen være god nok i mange tilfeller. Årsaken til at vi introdusere flere metoder, er at de også brukes til å utlede løsningsalgoritmer for differensiallikninger, som vi skal se på i kapittel 7.1.

6.2 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 $h = \Delta x$ spesifisert av avstanden mellom datapunktene våre. Hvis målefrekvensen er lav, blir h høy, og motsatt. Vi kan bruke titreringsdataene våre fra titreringa av eddiksyre med NaOH (figur 6.1) som eksempel. Vi kan lese inn dataene og derivere dem 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

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

derivert = numder(volum,pH)

plt.plot(volum, pH, label='Titrerkurve')
plt.plot(volum[0:-1],derivert, label='Derivert titrerkurve')
plt.title('Titrering av eddiksyre med NaOH')
plt.xlabel('Volum NaOH tilsatt (mL)')
plt.ylabel('pH')
plt.legend()
plt.savefig('derivert_titrering.pdf')
plt.show()
```

Dette gir følgende plott:



Figur 6.2: Titrerkurve og dens deriverte.

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.

Vi ser at den deriverte er høyest i ekvivalenspunktet. 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 manuelt med utgangspunkt i konsentrasjonen til syra (0.12 M).

Underveisoppgave 6.3

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

6.2.1 Derivasjon: Oppsummering

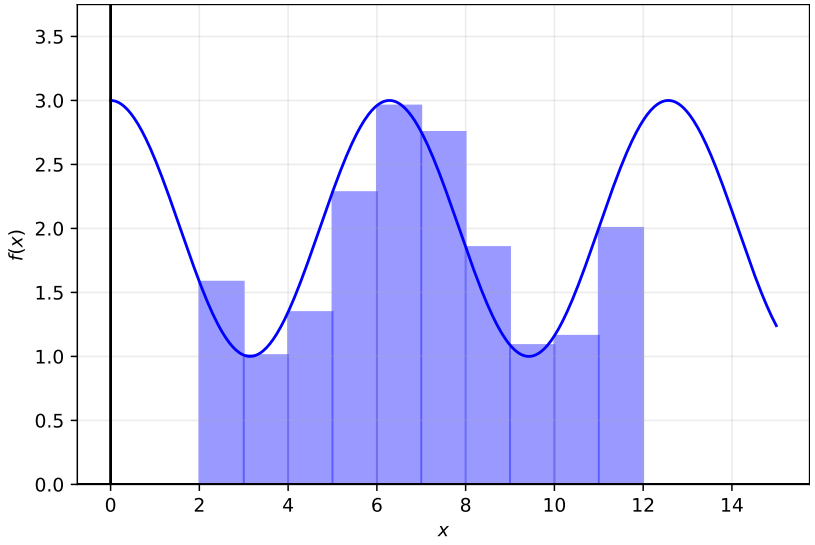
Vi har nå sett på hvordan numerisk derivasjon kan brukes til å derivere funksjoner og analysere kjemiske data. Vi så også på tre ulike tilnærminger til den deriverte: framoverdifferanse, bakoverdifferanse og sentraldifferanse. Alle tre metodene kan brukes, men det er ofte sentraldifferensen som gir minst feil.

6.3 Numerisk integrasjon

Du kjenner kanskje integrasjon som en metode å regne ut arealet under en graf eller volumet til legemer på. I tillegg kjenner du antakelig til integrasjon som den motsatte operasjonen av derivasjon. At derivasjon og integrasjon er motsatte operasjoner, er bevist gjennom *analysens fundamentalteorem*. Termen *analyse* brukes her om den greinen av matematikk som omhandler derivasjon og integrasjon (også kalt *kalkulus*). Siden datamaskinen kun kan operere med bestemte verdier, kan vi med numeriske algoritmer kun tilnærme *det bestemte integralet*. Vi antideriverer derfor ikke i dette kapitlet.

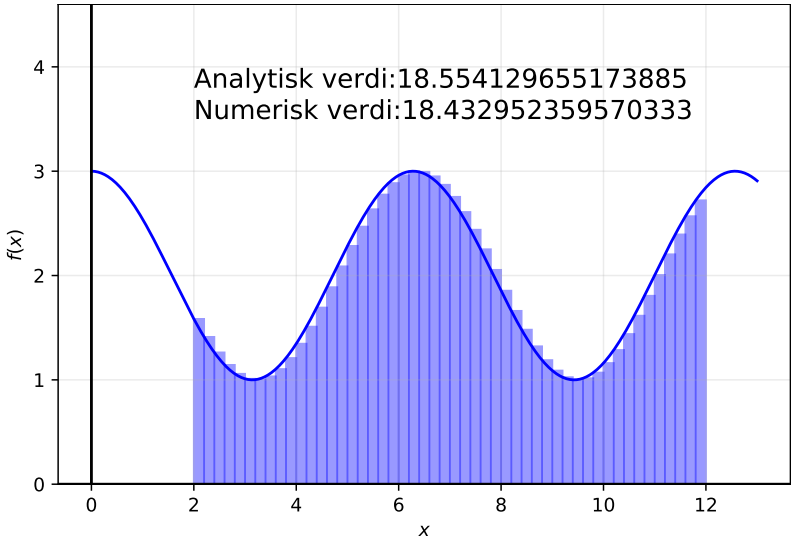
6.3.1 Rektangelmetoden

Vi kan enklest gjøre en tilnærming til det bestemte integralet ved å utnytte at det kan skrives som en grenseverdi av Riemann-summer. En Riemann-sum kan beskrives som en tilnærming til arealet under en graf ved hjelp av arealet til geometriske figurer. En vanlig tilnærming er å bruke rektangler:



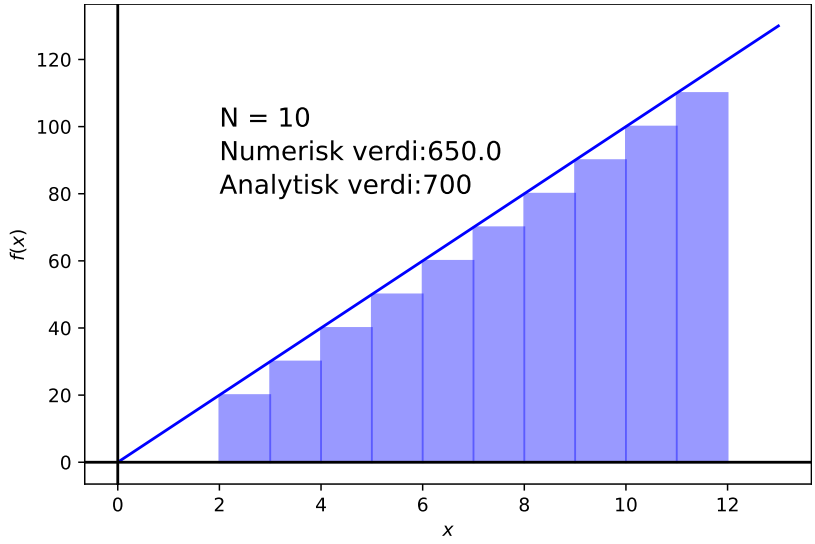
Figur 6.3: Illustrasjon av rektangelmetoden med $n = 10$.

Her benyttes $N = 10$ rektangler for å tilnærme integralet av $f(x) = \cos(x) + 2$ for $x \in [2, 12]$. Bredden av rektanglene må være $(b - a)/N = (12 - 2)/10 = 1$. Vi ser også at høyden av hvert rektangel er $f(x_n)$ der $n \in [2, 11]$, det vil si at vi lar venstresiden av rektangelet gå opp til grafen. Dersom vi regner ut arealet til hver av disse rektanglene, får vi 18.046675645664006, noe som ligger et lite stykke unna den analytiske verdien $((\sin(12) + 2 \cdot 12) - (\sin(2) + 2 \cdot 2) \approx 18.554129655173885$. Dersom vi øker antall rektangler, her til 50, får vi naturlig nok et bedre estimat (her har vi inkludert resultatene i figuren):



Figur 6.4: Illustrasjon av rektangelmetoden med $n = 50$.

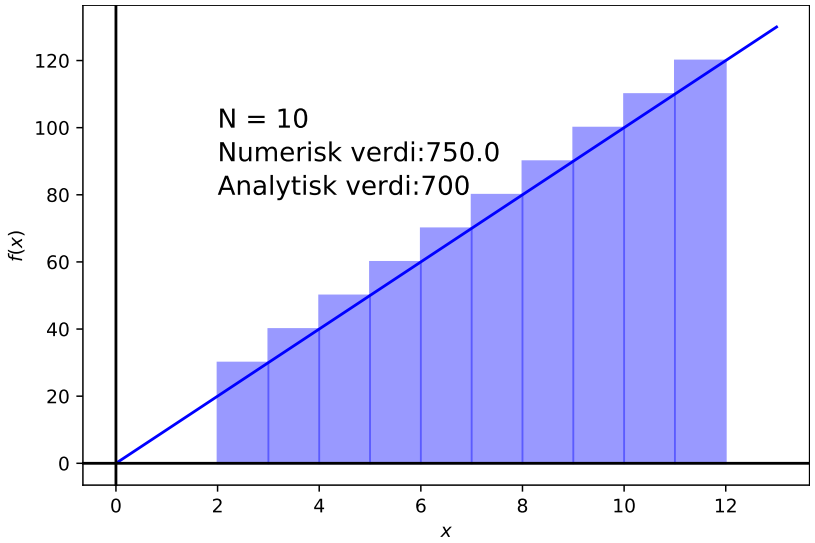
Det er åpenbart i figur 6.3 at et område hos de fleste rektanglene ligger utenfor grafen, og at de også ligger litt for langt under grafen flere steder. Men feilen blir ikke så stor som det kan se ut som fordi vi har områder både over og under grafen. Relativ feil er her ca. 2.7 % med 10 rektangler og 0.65 % med 50 rektangler. Men la oss prøve medoden på en lineær funksjon, $f(x) = 10x$:



Figur 6.5: Rektangelmetoden med venstretilnærming for lineær funksjon.

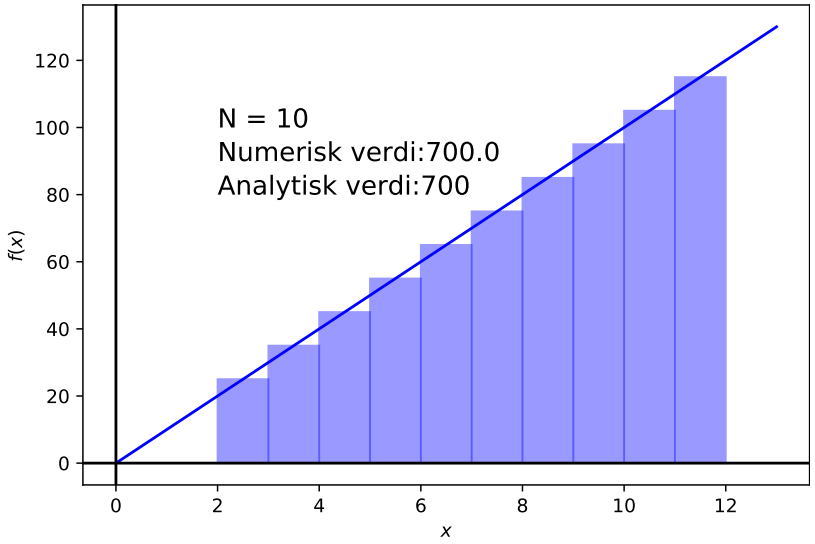
Vi har i disse modellene målt rektangelhøyden på venstre ytterkant av rek-

tangelet. Vi kan også måle høyden av rektanglene på høyre ytterkant:



Figur 6.6: Rektangelmetoden med høyretilnærming for lineær funksjon.

Vi ser her at vi får en stor underestimering med venstretilnærmingen og en stor overestimering med høyretilnærmingen, med en relativ feil på ca. 7.1 %. Det samme skjer for alle funksjoner i intervaller som er enten voksende eller synkende i hele intervallet. En måte å kompensere for dette, er å benytte høyden til rektangelet i midten istedenfor i endepunktet til venstre:



Figur 6.7: Rektangelmetoden med midtpunktstilnærming for lineær funksjon.

Det aller beste resultatet får vi altså med midtpunktstilnærmingen. For lineære funksjoner vil vi få en eksakt verdi (selv med ett rektangel!) fordi rektanglene er like store over og under grafen. Men midtpunktstilnærmingen er generelt bedre også på for eksempel polynomer av høyere grad og trigonometriske funksjoner. La oss nå se på implementering av metodene.

Underveisoppgave 6.4

Lag tre skisser som illustrerer de tre rektangeltilnærmingene til det bestemte integralet.

6.3.2 Algoritmer og implementering

Vi ser først på venstretilnærmingen av rektangelmetoden:

Numerisk metode: Rektangelmetoden (venstretilnærming)

Det bestemte integralet til en funksjon $f(x)$ fra $x = a$ til $x = b$ kan tilnærmes ved arealet til n rektangler med bredden $h = \frac{b-a}{n}$:

$$\int_a^b f(x)\,\mathrm{d}x \approx h\sum_{k=1}^{n} f(x_k) \tag{6.5}$$

En måte å implementere og teste algoritmen på, er slik:

```
def f(x):           #Definerer en funksjon som vi skal integrere.
    return x**3

def f_analytisk(x):   #Definerer analytisk verdi for sammenlikning.
    return (1/4)*x**4

def rektangelmetoden(f, a, b, n):
    A = 0.0
    h = (b-a)/n       #Bredden til rektanglene
    for k in range (n):
        A = A + f(a + k*h)
    return A*h

print("Numerisk verdi:", rektangelmetoden(f, 0, 5, 1000))
print("Analytisk verdi:", f_analytisk(5)-f_analytisk(1))
```

Tilsvarende er en algoritme for høyretilnærmingen slik:

Numerisk metode: Rektangelmetoden (høyretilnærming)

Det bestemte integralet til en funksjon $f(x)$ fra $x = a$ til $x = b$ kan tilnærmes ved arealet til n rektangler med bredden $h = \frac{b-a}{n}$:

$$\int_a^b f(x)\ \mathrm{d}x \approx h \sum_{k=1}^{n} f(x_{k+1}) \tag{6.6}$$

Underveisoppgave 6.5

Implementer algoritmen for høyretilnærmingen som en Python-funksjon. Test og sammenlikn med venstretilnærmingen på integralet $\int_2^8 f(x) = x^2 - 2x + 4\ dx$.

Til slutt ser vi på midtpunktstilnærmingen:

Numerisk metode: Rektangelmetoden (midtpunktstilnærming)

Det bestemte integralet til en funksjon $f(x)$ fra $x = a$ til $x = b$ kan tilnærmes ved arealet til n rektangler med bredden $h = \frac{b-a}{n}$:

$$\int_a^b f(x)\ \mathrm{d}x \approx h \sum_{k=1}^{n} f\left(\frac{1}{2}(x_k + x_{k+1})\right) \tag{6.7}$$

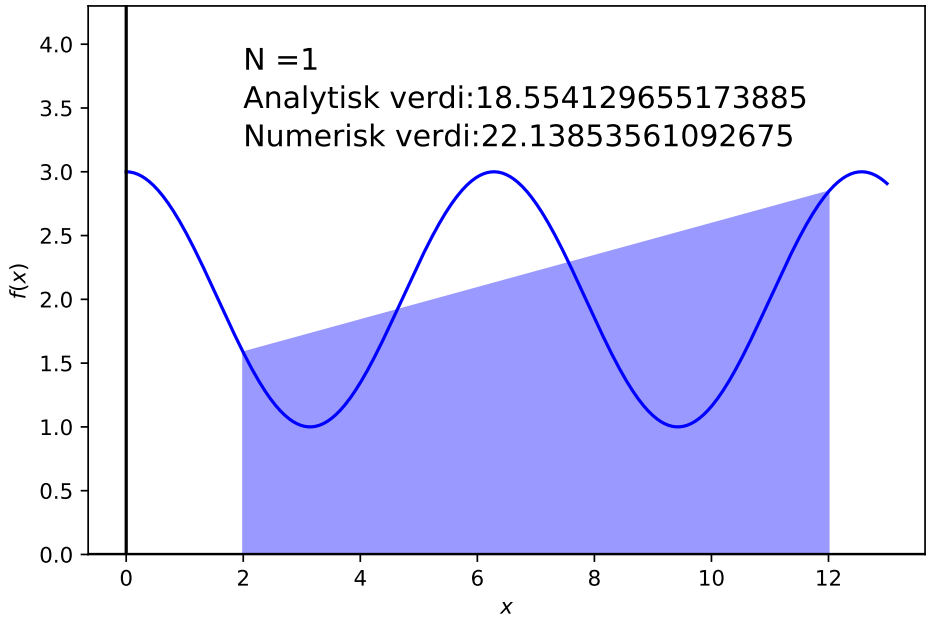
En mulig implementering av denne algoritmen er slik:

```
def rektangelmetoden_midt(f, a, b, n):
    A = 0.0
    h = (b-a)/n       #Bredden til rektanglene
    for k in range (n):
        A = A + f(a + (1+2*k)*(h/2))
    return A*h
```

Dersom vi har funksjoner med stor stor stigning eller minking ($|f'(x) >> 0|$), trenger vi mange rektangler for å få et godt resultat. Dette kan gi langsomme programmer dersom integrasjonen må gjentas flere ganger. Vi skal derfor se på noen forbedringer av metoden.

6.3.3 Trapesmetoden

Vi kan legge merke til at toppstykket i et rektangel er ei rett, horisontal linje. Ei slik linje kan representeres som et polynom av nullte grad, $f(x) = ax^0 = a$, der a er et reellt tall. La oss se på mulighetene for å bytte ut dette toppstykket med et polynom av første grad, $f(x) = ax^1 = ax$. Da får vi *trapeser* istedenfor rektangler. En algoritme for dette er litt mindre intuitiv og litt mer jobb å utlede, men vi spanderer på oss det. La oss ta utgangspunktet i trapesmetoden illustrert med ett trapes i intervallet $[a, b] = [2, 12]$ på $f(x) = cos(x) + 2$:



Figur 6.8: Trapesmetoden med ett trapes.

Arealet av et trapes er gitt ved følgende formel:

$$A_{trapes} = \frac{side1 + side2}{2} \cdot h \tag{6.8}$$

De to sidene x_1 og x_2 er gitt ved henholdsvis $f(a)$ og $f(b)$. Høyden i trapeset blir stykket langs hele x-aksen, altså $b - a$. Arealet blir derfor:

$$A = \frac{f(a) + f(b)}{2} \cdot (b - a) \tag{6.9}$$

La oss nå utvide til n trapeser. Da blir høyden av hvert trapes $h = (x_1 + x_2)/n$, noe som gir dette arealet for hvert k-te trapes:

$$A_i = \frac{f(a) + f(b)}{2} \cdot h \tag{6.10}$$

Summen blir da slik for n trapeser:

$$\frac{f(a)+f(a+h)}{2}\cdot h+\frac{f(a+h)+f(a+2h)}{2}\cdot h+$$

$$\frac{f(a+2h)+f(a+3h)}{2}\cdot h\ldots\frac{f(a+kh)+f(b)}{2}\cdot h$$

Vi multipliserer så alle ledd med h og dividerer dem på 2. Dette setter vi utenfor uttrykket:

$$\frac{h}{2}\cdot(f(a)+f(a+h)+f(a+h)+f(a+2h)+f(a+2h)+\ldots+f(a+kh)+f(b)) \tag{6.11}$$

Trekker vi sammen like ledd, får vi:

$$\frac{h}{2}\cdot(f(a)+2f(a+h)+2f(a+2h)+2f(a+3h)+\ldots+2f(a+kh)+f(b)) \tag{6.12}$$

Siden det bare er $f(a)$ og $f(b)$ som ikke er multiplisert med 2, kan vi forenkle:

$$h\left(\frac{f(a)+f(b)}{2}+(f(a+h)+f(a+2h)+f(a+3h)+\ldots+f(a+kh))\right) \tag{6.13}$$

Den siste samlingen av ledd kan vi skrive som en sum. Da får vi trapesmetoden:

Numerisk metode: Trapesmetoden

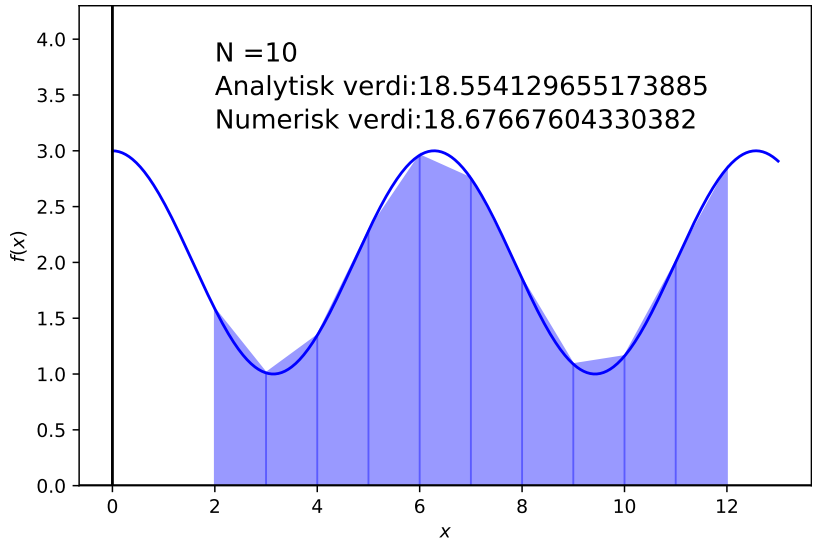
Det bestemte integralet til en funksjon $f(x)$ fra $x=a$ til $x=b$ kan tilnærmes ved arealet til n trapeser med bredden $h=\frac{b-a}{n}$:

$$\int_a^b f(x)dx\approx h\left(\frac{f(a)+f(b)}{2}+\sum_{k=1}^{n-1}f(x_k)\right) \tag{6.14}$$

Underveisoppgave 6.6

Implementer trapesmetoden og test den på funksjonene $f(x)=x^3+2x$ og $f(x)=\sqrt{(}x)$ i intervallet $[2,4]$. Sammenlikn med resultatene du får fra rektangelmetoden med samme antall geometriske figurer.

En illustrasjon av trapesmetoden med ti trapeser er vist nedenfor:



Figur 6.9: Trapesmetoden.

6.3.4 Simpsons metode

Vi har nå sett på to tilnærminger til integralet som bruker henholdsvis nullte- og førstegradspolynomer som toppstykke på de geometriske figurene som vi beregner arealet av. For å få en enda bedre tilnærming, spesielt til oscillerende funksjoner, kan vi bruke et toppstykke av et polynom med høyere grad enn 1. Alle disse metodene kan utledes ved hjelp av interpolasjon, men vi skal ikke gjøre det her. Her nøyer vi oss med å vise og implementere algoritmen for tredjegradstilnærmingen. Denne algoritmen kalles *Simpsons metode*:

Numerisk metode: Simpsons metode

Det bestemte integralet til en funksjon $f(x)$ fra $x = a$ til $x = b$ kan tilnærmes slik:

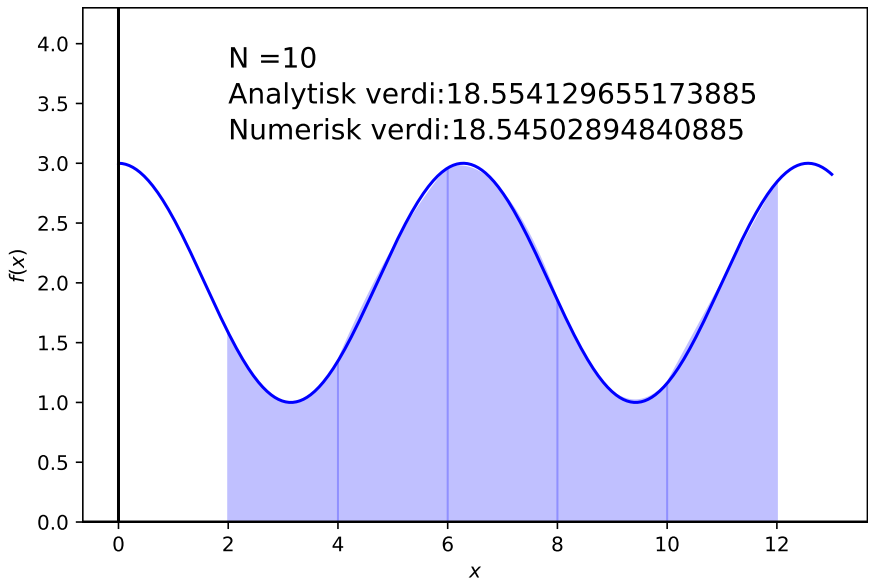
$$\int_a^b f(x)dx \approx \frac{h}{3}\left(f(a) + f(b) + 2\sum_{k=1}^{\frac{n}{2}-1} f(x_{2k}) + 4\sum_{k=1}^{\frac{n}{2}} f(x_{2k-1})\right) \tag{6.15}$$

Underveisoppgave 6.7

Det er lurt å øve seg på å oversette algoritmer til kode (implementering). Prøv derfor å implementere algoritmen ovenfor før du går videre og ser på løsningsforslaget nedenfor.

```
def simpsons(f, a, b, n):
    h = (b-a)/n
    total = f(a) + f(b)
    for k in range(1,n,2):
        total += 4 * f(a + k*h)
    for k in range(2,n-1,2):
        total += 2*f(a + k*h)
    return total*h/3
```

En illustrasjon av Simpsons metode med fem geometriske figurer er vist her:



Figur 6.10: Simpsons metode.

Metodene vi har sett på, bygger på samme prinsipp, nemlig tilnærming av arealet under grafen ved hjelp av geometriske figurer med rektangelbase og et polynom av grad n som toppstykke. Siden prinsippet er det samme, kaller vi dem en *familie* av metoder (hyggelig, ikke sant?). Denne familien heter *Newton-Cotes*. Det vil si at for eksempel trapesregelen kalles en Newton-Cotes-metode av første grad. Det finnes mange andre metoder og familier innenfor numerisk integrasjon, men disse lar vi ligge foreløpig.

6.3.5 Monte Carlo-integrasjon

En helt annen familie av integrasjonsmetoder, kalles *Monte Carlo-integrasjon*. Metodene er oppkalt etter kasinoet i Monte Carlo fordi de benytter tilfeldige tall som grunnlag for det de skal tilnærme. Det er enormt mange anvendelser av MC-metoder, men vi skal nøye oss med å se på integrasjon her. Først ser vi derimot litt på hvordan vi genererer tilfeldige tall i Python.

Tilfeldige tall

Tilfeldige tall på en datamaskin er ikke helt tilfeldig. Det er fordi de genereres ved hjelp av algoritmer, som igjen kan brukes til å forutsi de «tilfeldige» tallene. Men de er tilfeldige nok til at vi kan bruke dem til statistisk simulering. Vi starter med et første eksempel der vi kaster en terning 1000 ganger og legger resultatet i ei liste:

```
import numpy as np
N = 1000 # Antall kast
resultater = []

for i in range(N):
    kast = np.random.randint(1,7)
    resultater.append(kast)
```

Det eneste ukjente fra programmet ovenfor er funksjonen `randint` fra random-biblioteket som finnes i numpy. Det er denne funksjonen som sammen med `uniform` 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 6.8

Modifiser programmet ovenfor slik at det:

1. sparer på antall seksere og printer ut relativ frekvens av seksere til slutt (antall seksere dividert på antall kast). Hvordan sammenfaller dette med sannsynligheten for å få en sekser?
2. plotter relativ frekvens som funksjon av antall kast. Hva sier plottet oss om sannsynlighet?

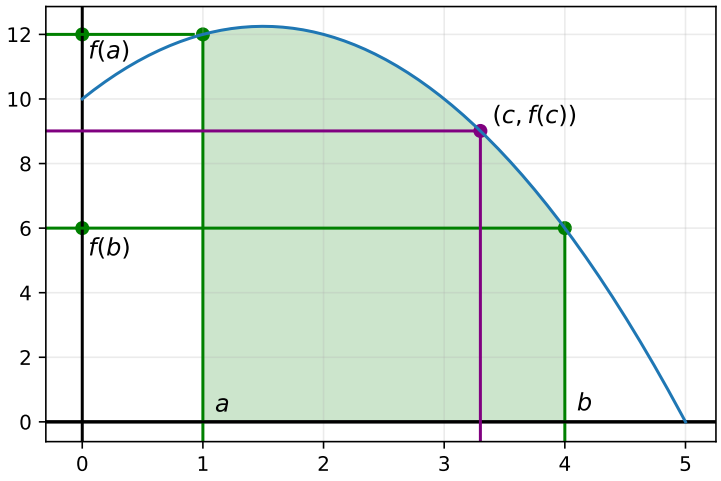
Hvis vi trenger veldig mange tilfeldige tall, spesielt i flere operasjoner, kan vi vektorisere koden slik:

```
import numpy as np

N = 1000
kast = np.random.randint(1,7,N)
```

Dette genererer en array av N tilfeldige tall veldig raskt, noe som kommer godt med i simuleringer som krever en del datakraft.

Monte Carlo-metoder

La oss si at vi har en funksjon f som er kontinuerlig i intervallet $[a, b]$ og deriverbar i (a, b). Dersom vi tilnærmer arealet under grafen med en geometrisk figur, slik som vi gjorde i Newton-Cotes-metodene tidligere i kapitlet, vil det finnes et punkt c som er slik at $f(c)$ er lik høyden til den geometriske figuren. Tilsvarende vil det finnes et punkt c som er slik at $f(c)$ er lik gjennomsnittet av funksjonsverdiene i intervallet. Vi kan illustrere det med en figur:



Figur 6.11: Gjennomsnittlig høyde innenfor det fargede området er gitt ved $f(c)$.

Her er den gjennomsnittlige funksjonsverdien $\bar{f} = f(c)$. Det betyr da at arealet under grafen kan skrives som $\bar{f} \cdot h$, der $h = b - a$ er bredden til figuren. Dermed får vi:

$$\int_a^b f(x)\, dx = \bar{f} \cdot (b - a) \tag{6.16}$$

som igjen gir oss et uttrykk for den gjennomsnittlige funksjonsverdien gitt integralet over et intervall $[a, b]$. Vi kan kalle dette *middelverdisetningen for integrasjon*.

Middelverdisetningen for integrasjon

Gitt en funksjon $f(x)$ som er kontinuerlig i intervallet $[a, b]$ og deriverbar i (a, b), er den gjennomsnittlige funksjonsverdien $\bar{f}$ gitt ved:

$$\bar{f} = \frac{1}{(b-a)} \int_a^b f(x)\, dx \tag{6.17}$$

Dermed trenger vi en tilnærming til den gjennomsnittlige funksjonsverdien for å finne en tilnærming til det bestemte integralet. Vi kan starte med å ta gjennomsnittet av n punkter x_0, x_1, ... x_{n-1}:

$$\bar{f} \approx \frac{1}{n} \sum_{i=0}^{n-1} f(x_i) \tag{6.18}$$

Deretter trenger vi å bestemme hva slags funksjonsverdier vi skal plukke ut. Hvis vi velger dem fra et uniformt intervall, kan vi faktisk ende opp med rektangelmetoden! Men vi skal nå prøve å plukke dem ut tilfeldig, og så ta gjennomsnittet av disse tilfeldige verdiene. Programmet nedenfor viser hvordan dette kan gjøres vektorisert:

```
import numpy as np

def MC_integrasjon(f, a, b, n):
    # Velger n tilfeldige tall i intervallet [a,b]
    x = np.random.uniform(a, b, n)
    # Summerer f(x) for alle tilfeldige tall x
    f_bar = np.sum(f(x))
    # Estimerer integralet
    f_int= (b-a)/n * f_bar
    return f_int
```

Underveisoppgave 6.9

Test metoden ovenfor på funksjoner du allerede har brukt andre numeriske integrasjonsmetoder på. Sammenlikn svarene for ulike verdier av n.

Du kommer til å finne ut at MC-integrasjon gir relativt dårlige resultater sammenliknet med de andre metodene du har brukt. Dessuten trengs det veldig mange funksjonsverdier (høy n) for å få et godt nok resultat. Men selv om MC-integrasjon ikke gir gode verdier her, betyr ikke det at metodene er uten verdi (pun intended). Det er nemlig en glimrende metode for å løse dobbel- og trippelintegraler. Dette gjøres mye i blant annet kvantekjemi, så det er viktig å forstå grunnprinsippene i hvordan slike metoder brukes.

6.3.6 Anvendelser av integrasjon i kjemi

Det finnes mange anvendelser av integrasjon i kjemi. Integrasjon brukes til alt fra å beregne molekylgeometri i kvantekjemi til å beregne arealet til en topp i et NMR-spekter. Dessuten integrerer vi når vi løser differensiallikninger. Da kan vi benytte beslektede metoder til de vi har vist ovenfor, eller vi kan benytte helt andre prinsipper. Dette skal vi se på i kapittel 7.1.

La oss bare for ordens skyld ta et lite eksempel der vi har en funksjon som beskriver hastigheten til et elektron ved tida t: $v(t) = \sqrt{t} + 2$. Vi ønsker å finne totalt tilbakelagt strekning de første 10 sekundene. Da kan vi integrere funksjonen, slik at vi får posisjonen. Vi må også passe på at hastigheten er positiv hele tida, hvis ikke må vi ta hensyn til at partikkelen beveger seg bakover. Her er derimot farten hele tida positiv. En programskisse som løser problemet kan se slik ut:

```
definer v(t) = x**0.5 + 2
avstand = integrer(a=0,b=10,f=v(t),N=1000)
```

Underveisoppgave 6.10

Lag et program som utfører integrasjonen i skissen ovenfor. Benytt en valgfri integrasjonsmetode. Hva blir totalt tilbakelagt strekning for elektronet?

6.4 Numeriske biblioteker

Når du har forstått teorien og prøvd å implementere funksjoner for numerisk likningsløsing, derivasjon og integrasjon, kan du begynne å se på ferdige funksjoner for integrasjon. Disse kan importeres fra det numeriske biblioteket *scipy*. Nedenfor ser du eksempler på bruk av noen metoder du kjenner til, og noen som antakelig er ukjente for deg:

```
from scipy import integrate
from scipy.misc import derivative
import numpy as np
```

```
def f(x):
    return x**3 - 1

def fder(x):
    return 3*x**2

n = 1000
x = np.linspace(0,5,n)
y = f(x)

# Derivasjon
derivert = derivative(f,1,dx=1E-8)
print("f'(x) = ",derivert)

# Integrerasjon
trapes = integrate.trapz(y,x) # Trenger arrayer
simpsons = integrate.simps(y,x) # Trenger arrayer
gauss_kvadratur = integrate.quad(f,0,5) # Trenger funksjon
print("Trapesmetoden:",trapes)
print("Simpsons metode:",simpsons)
print("Gauss kvadratur:",gauss_kvadratur) #Skriver ut svar og
    absolutt feil
```

Underveisoppgave 6.11

Studer og test ut koden ovenfor. Hvordan fungerer de ulike funksjonene?

6.4.1 Dobbel- og trippelintegrasjon

Multippel integrasjon kan også gjennomføres med scipy-biblioteket. La oss prøve å løse følgende integraler med funksjonene `dblquad` og `tplquad`:

$$\int_0^{\frac{\pi}{2}} \int_{-1}^{1} x\sin(y) - ye^x \; dxdy \tag{6.19}$$

$$\int_0^3 \int_0^2 \int_0^1 xyz \; dxdydz \tag{6.20}$$

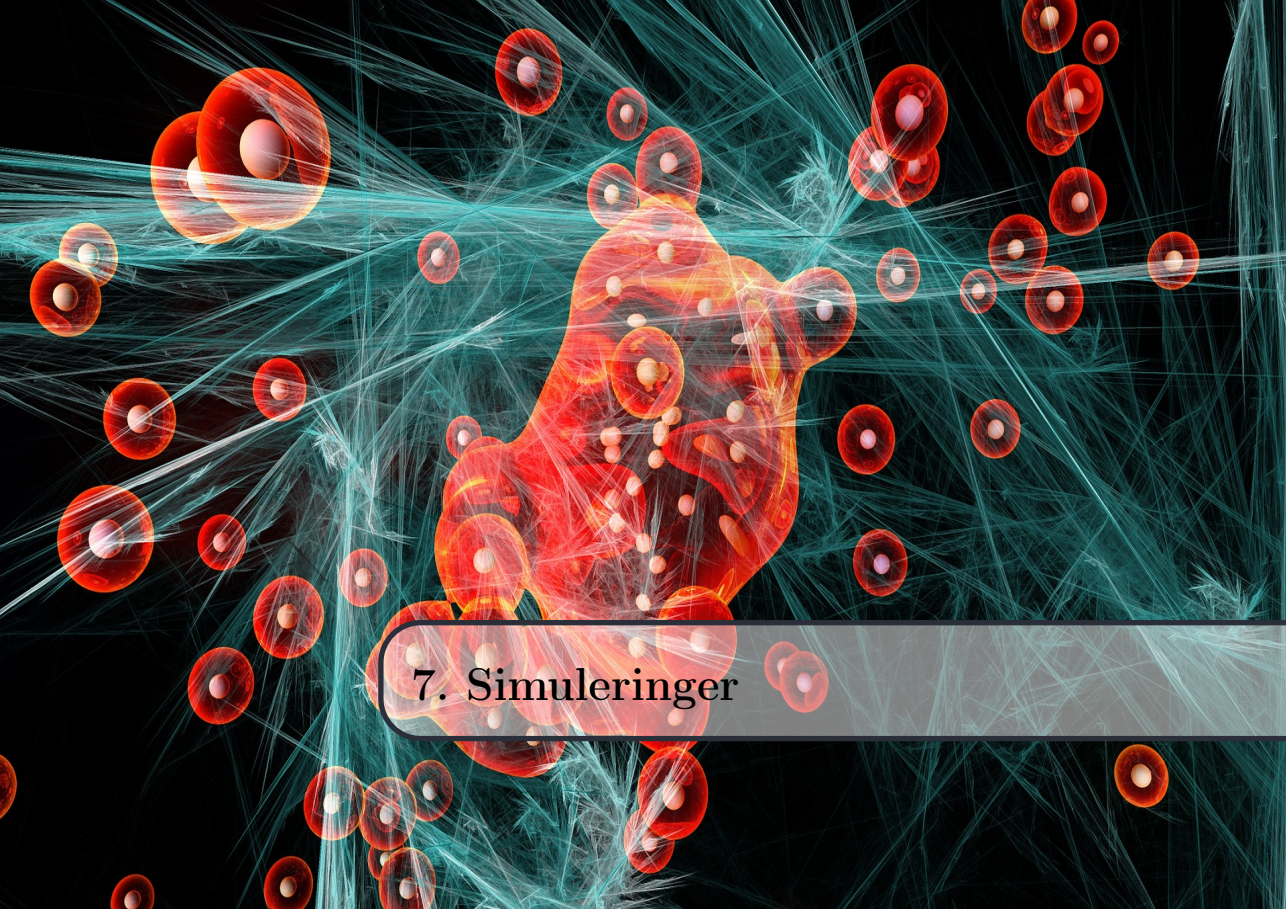
```
from scipy import integrate
import numpy as np

def f(y,x):
    return x*np.sin(y) - y*np.exp(x)

def g(z, y, x):
```

```
    return x*y*z

numerisk_dobbel = integrate.dblquad(f, -1, 1, 0, np.pi/2)
numerisk_trippel = integrate.tplquad(g, 0, 1, 0, 2, 0, 3)

analytisk_dobbel = np.pi**2/8 * (1/np.exp(1)-np.exp(1))
analytisk_trippel = 9/2
print(f'Numerisk verdi av dobbeltintegralet: {numerisk_dobbel[0]}')
print(f'Analytisk verdi av dobbeltintegralet: {analytisk_dobbel}')

print(f'Numerisk verdi av trippelintegralet: {numerisk_trippel[0]}')
print(f'Analytisk verdi av trippelintegralet: {analytisk_trippel}')
```

6.5 Oppsummering

I dette kapitlet har vi sett på ulike måter å derivere og integrere på. Det viktigste du skal kunne fra dette kapitlet, er:

1. Forklare forskjellen mellom analytisk og numerisk derivasjon.
2. Gjøre rede for ulike tilnærminger til den deriverte (framoverdifferanse, bakoverdifferanse, sentraldifferanse).
3. Gjøre enkle feilanalyser av metoder for numerisk derivasjon.
4. Bruke numerisk derivasjon til å derivere funksjoner fra kjemi og diskrete kjemiske data.
5. Gjøre rede for ulike tilnærminger til det bestemte integraler (rektangelmetoden, trapesmetoden, hovedprinsippet bak Simpsons metode).
6. Implementere rektangelmetoden og trapesmetoden og bruke disse til å integrere funksjoner.
7. Bruke scipy-biblioteket til å derivere og integrere funksjoner.



7. 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.

7.1 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. De er derfor *differensiallikninger*.

> **Differensiallikninger**
>
> En differensiallikning er en likning som inneholder den deriverte av en funksjon. 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 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). Reaksjonslikningen for reaksjonen kan skrives 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:

$$c'(t) = k_r[H_2][I_2] \tag{7.1}$$

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.

Likningen ovenfor har i utgangspunkt tre ukjente, $[HI]_{t+dt}$, $[H]_{t+dt}$ og $[I]_{t+dt}$. Heldigvis er 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:

$$[HI]'(t) = k_r[H_2][I_2] \tag{7.2}$$

$$[H_2]'(t) = -0.5[HI]'(t) = -0.5 \cdot k_r[H_2][I_2] \tag{7.3}$$

$$[I_2]'(t) = [H_2]'(t) = -0.5 \cdot k_r[H_2][I_2] \tag{7.4}$$

La oss se på en enkel metode for å tilnærme konsentrasjonen til reaktantene og produktene som funksjon av tid, med andre ord en metode for løsing av differensiallikninger.

7.2 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:

$$f'(x) \approx \frac{f(x+dx) - f(x)}{dx} \tag{7.5}$$

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. 7.1). I tillegg bestemmer vi selv tidssteget dx, men husk at det verken bør være for lite eller for stort. Den eneste ukjente i likning 7.5 er altså $f(x+dx)$. Det er jo nettopp $f(x+dx)$ vi prøver å finne, fordi det beskriver tilstanden til systemet ved neste tidssteg. 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:

$$f(x+dx) \approx f(x) + f'(x) \cdot dx \tag{7.6}$$

Dette er *Eulers metode*, eller nærmere bestemt *Forward Euler*. Metoden kalles dette fordi den tar utgangspunkt i framoverdifferansen til den deriverte ($\mathbf{D}^+$ fra kapittel 6.1. Den brukes til å løse differensiallikninger, det vil si å *integrere* den deriverte slik at vi finner funksjonsverdiene. Siden vi ofte har å gjøre med funksjoner som varierer med tid, kaller vi gjerne dx for dt.

Numerisk metode: 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.

$$f(t_{k+1}) = f(t_k) + f'(t_k) \cdot dt \tag{7.7}$$

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

$$c(t + dt) \approx c(t) + c'(t) \cdot dt \tag{7.8}$$

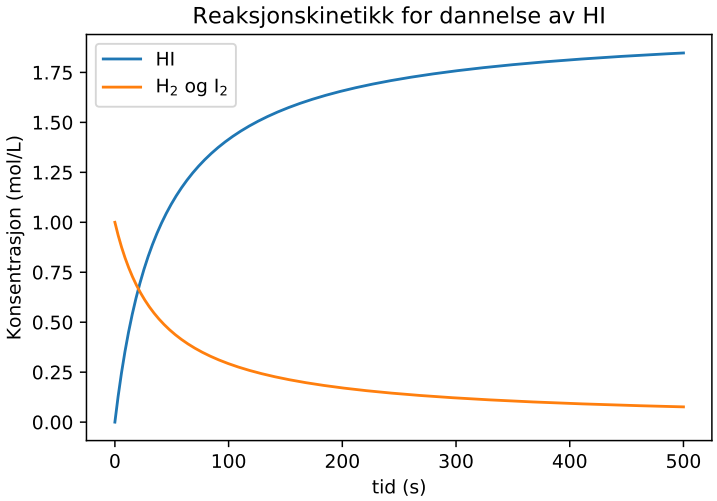
Vi skal nå se på hvordan vi kan implementere denne metoden i Python.

```
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     # Ratekonstant ved ca. 450 grader C

#Tidssteg
dt = 1E-3
tid = 100 # Tid i sekunder
N = int(tid/dt) + 1 # Antall iterasjoner

#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

H[0] = H0
I[0] = I0
HI[0] = HI0

# Eulers metode
for i in range(N-1):
    HIder = k*H[i]*I[i]
    Hder = -0.5*HIder
    Ider = Hder
    HI[i+1] = HI[i] + HIder*dt
    H[i+1] = H[i] + Hder*dt
    I[i+1] = I[i] + Ider*dt
    t[i+1] = t[i] + dt

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 = 'H$_2$ og I$_2$')
plt.legend(loc=0)        # Merkelapp med bestemt posisjon i vinduet
plt.show()
```

Underveisoppgave 7.1

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 er ferdig.

Programmet ovenfor gir oss en graf for de første 100 sekundene som virker nokså logisk. Det kan være lurt å evaluere grafen vi 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 reaksjonslikningen.



Figur 7.1: 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 initialbetingelser
k = 4.84E-2
HI0 = 0
H0 = 1
I0 = 1
```

```
# Tidsparametre
dt = 1E-3
tid = 100 # Tid i sekunder
N = int(tid/dt) + 1
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(.. (fyll inn parametere her)):
    # 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 7.2

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.

Dersom vi lager funksjoner med parametre ovenfor, trenger vi egentlig ikke all informasjonen i begynnelsen av programmet. Vi kan da gi alle startbetingelser som parametre og kalle funksjonen med disse parameterne. Det kan gjøres slik:

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

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

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

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

#Forward Euler
def FE(HI0, H0, I0, k, tid_slutt, t0=0, h=1E-4):
    N = int((tid_slutt - t0)/h)
    HI = np.zeros(N+1)
    H = np.zeros(N+1)
    I = np.zeros(N+1)
    t = np.zeros(N+1)
    HI[0] = HI0
    H[0] = H0
    I[0] = I0
    t[0] = t0
    for i in range(N):
        HIder = dHI(H[i],I[i],k)
        Hder = -0.5*HIder
        Ider = Hder
        HI[i+1] = HI[i] + HIder*h
        H[i+1] = H[i] + Hder*h
        I[i+1] = I[i] + Ider*h
        t[i+1] = t[i] + h
    return HI, H, I, t

HI, H2, I2, t = FE(HI0=0, H0=1, I0=1, k = 4.84E-2, tid_slutt = 500)
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 7.3

Studer programmet ovenfor og forklar hvordan det fungerer.

Ved høye temperaturer vil flere og flere HI-molekyler kollidere og rives løs igjen til I_2 og H_2. Reaksjonen er derfor egentlig reversibel, selv om vi har gjort en forenkling og beskrevet den som irreversibel ovenfor. Det viser seg at den motsatte reaksjonen følger denne rateloven:

$$c'(t) = k_{bakover}[HI]^2 \tag{7.9}$$

Reaksjonen er altså andreordens med hensyn på konsentrasjonen av hydrogenjodid. Den totale rateloven for hydrogenjodid blir derfor:

$$c'(t) = k_{framover}[H_2][I_2] - k_{bakover}[HI]^2 \tag{7.10}$$

Dersom vi kjenner likevektskonstanten K ved den gitte temperaturen, kan vi finne $k_{bakover}$ ved å benytte følgende sammenheng:

$$K = \frac{k_{framover}}{k_{bakover}} \tag{7.11}$$

Underveisoppgave 7.4

Lag et program som simulerer reaksjonen mellom I_2 og H_2 ved 450°C. Sett likevektskontstanten til å være 100 og $k_{framover}$ til å være $4.8 \cdot 10^{-2}$. Lag et plott som viser konsentrasjonen som funksjon av tid, og et plott som viser reaksjonsraten/reaksjonshastigheten som funksjon av tid. Kommenter plottene. Stemmer dette med det du kan om likevekter?

Når vi løser differensiallikninger, er det ikke alltid tilnærmingene er gode. Spesielt når vi har å gjøre med svingninger, gir Forward Euler ustabile og ofte helt feil resultater, spesielt med mindre dt er svært liten. Og dersom dt blir svært liten, får vi også et problem med veldig tunge beregninger. Det kan også bli et problem dersom vi ønsker å studere et system langt fram i tid. Forward Euler er derfor mer ment som en introduksjon til løsing av differensiallikninger, da metoden i seg selv ikke benyttes i stor grad. Vi skal nå se på et utvalg andre metoder.

7.3 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 utgangspunkt 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 $\boldsymbol{t+dt}$. Det er altså bare en forskjell i hvilken deriverte vi tar utgangspunkt i: FE tar utgangspunkt i

framoverdifferansen av den deriverte, mens BE tar utgangspunkt i bakoverdifferansen (jf. kapittel 7.1). Dette ser ubetydelig ut, men det gir en ganske stor forskjell i hvordan vi implementerer metoden.

Vi kaller Forward Euler for en *eksplisitt metode* fordi vi finner tilstanden til et system ved et seinere tidspunkt $(t + dt)$ ut fra endringen av systemet ved det nåværende tidspunktet $(f'(t))$. Backward Euler er derimot en *implisitt metode* fordi vi finner tilstanden til et system ved et seinere tidspunkt $(t + dt)$ ut fra endringen av systemet ved det samme tidspunktet $(f'(t + dt))$. Vi kan generalisere BE-metoden slik:

Numerisk metode: 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}.

$$f(t_{k+1}) = f(t_k) + f'(t_{k+1}) \cdot dt \tag{7.12}$$

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. BE gir også gode resultater med store tidssteg og egner seg derfor godt til simulering av systemer over lengre tidsintervaller, men den gir i utgangspunktet like store feil som FE med samme tidssteg. Den egner seg derfor best som introduksjon til implisitte metoder, og vi skal derfor ikke se på implementering av denne metoden her. Vi skal snart se på hvordan vi kan bruke implisitte metoder fra scipy-biblioteket til å løse stive differensiallikninger. Men før vi gjør det, skal vi kikke litt på en metode som er mye brukt til å løse differensiallikninger, nemlig Runge-Kutta 4.

Underveisoppgave 7.5

Oppsummer hva som er forskjellene og likhetene mellom Forward Euler og Backward Euler, og hva som er forskjellen på en eksplisitt og en implisitt metode.

7.4 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.

En av de vanligste Runge-Kutta-metodene 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: 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:

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

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

Du kan se at vi har brukt en ny notasjon for differensiallikninger ovenfor, nemlig $y' = f(t, y)$. La oss spandere på oss litt forklaring her.

7.4.1 Notasjon

En generell differensiallikning kan skrives på ulike måter. Til nå har vi skrevet dem relatert til konkrete ratelover. Andre eksempel på differensiallikninger er:

$$f'(x) - x = 2$$

$$y' = y$$

$$y' = x - y$$

$$u' = t + 1$$

$$\text{u'(t)} = \text{u(t)}$$

Alle disse er differensiallikner som er skrevet med litt ulike notasjoner. Felles for dem er at de inneholder den deriverte som funksjon av en eller annen variabel. Det kan være nyttig å kalle venstresida av likningen, altså den deriverte,

for $u'(t)$ eller y'. På høyresida kan det være ulike variabler og konstanter, men det vanligste er at vi har en variabel og den integrerte funksjonen, $u(t)$ eller y. Vi kan derfor skrive høyresida som en funksjon av t og y eller t og $u(t)$. To vanlige måter å skrive generelle differensiallikninger på, er derfor:

$$y' = f(t, y) \tag{7.13}$$

$$u'(t) = f(t, u(t)) \tag{7.14}$$

Disse to skrivemåtene betyr det samme, men du kan se at vi har brukt skrivemåten i 7.13 i beskrivelsen av RK4 tidligere. Det er greit å ha oversikt over de ulike måtene å skrive differensiallikninger på fordi du mest sannsynlig kommer til å møte flere ulike notasjoner. Disse differensiallikningene kalles ODE-er (ordinary differential equations) for å skille dem fra PDE-er (partial differential equations). Vi skal ikke se på partielle differensiallikninger her, men mange av prinsippene for å løse dem er like som for ODE-er. Vi kommer til å bruke ODE som forkortelse videre.

7.5 Scipy-biblioteket

Nå skal vi se på hvordan vi bruker noen eksplisitte og implisitte ODE-løsere fra scipy-biblioteket. Vi starter med enkle differensiallikninger for å illustrere de grunnleggende prinsippene. En enkel differensiallikning vi kan begynne med, er:

$$y' = t - y \tag{7.15}$$

Vi kan skrive høyresida her som $f(t, y)$. Deretter kan vi implementere en løser i Python slik:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp

def dy_dt(t, y):
    return t - y

a = 0
b = 4
t = np.linspace(a,b,1000)
y0 = 1
y_int = solve_ivp(dy_dt, [a,b], [1], t_eval=t)

plt.xlabel("t")
plt.ylabel("y")
```

```
plt.plot(y_int2.t, y_int2.y[0])

plt.show()
```

Vi definerer her et sett med t-verdier slik at vi får et intervall å integrere over. Dernest er det viktig å ha en startbetingelse for y (husk: vi trenger den forrige y-verdien for å finne den neste). Funksjonen `solve_ivp` er en generell løser for differensiallikninger og tar som første parameter en funksjon av typen $f(t, y)$ – legg merke til rekkefølgen på parameterne. Deretter legger vi inn tidsintervallet som vi skal integrere over. Dette legges inn som en liste.

Etter tidsintervallet legges startbetingelsene inn. Siden `solve_ivp` er en løser som kan løse systemer av differensiallikninger, må vi lage startbetingelsen som en liste. Deretter har vi en del valgfrie parametre. Vi har brukt parameteren *t_ eval* her fordi den sier hvilke t-verdier vi skal regne ut y-verdier for. Hvis vi ikke gjør dette, får vi integralet kun evaluert i noen få punkter. Det kan være greit hvis vi for eksempel bare ønsker sluttverdien, men ikke hvis vi ønsker å plotte resultatene.

Dersom du prøver å printe ut `solve_ivp`, får du mye ulik informasjon. Derfor henter vi ut spesifikke t- og y-verdier ved å skrive `y_int2.t` som henter ut tidsverdiene, og `y_int2.y[0]` som henter ut y-verdiene. Legg merke til at y kan inneholde flere elementer ettersom vi kan løse systemer av differensiallikninger. Her må vi eksplisitt be om det første elementet (element 0 med Python-språk), selv om arrayen ikke inneholder flere y-verdier.

Underveisoppgave 7.6

Løs differensiallikningen $f'(t) = \cos(t)$ med $f(t_0) = 0$ med `solve_ivp` fra $t = 0$ til $t = \pi$. Plott den analytiske løsningen $f(t) = \sin(t)$ i samme koordinatsystem for å sammenlikne.

Algoritmene som brukes i slike biblioteker, er ofte sammensatte algoritmer som benytter seg av flere prinsipper enn en enkelt metode. Som standard benytter `solve_ivp` seg av en blanding av Runge-Kutta 4 og Runge-Kutta 5, kalt RK45. Vi kan gi en ekstra parameter *method* for å angi andre metoder. For eksempel kan det være nyttig med en «backward-løser» hvis vi skal løse stive differensiallikninger. En slik metode er BDF (Backward Differentiation Formula):

```
y_int = solve_ivp(dy_dt, [a,b], [1], t_eval=t, method = 'BDF')
```

Dette er en samling av bakoverløsere i samme familie. Den benytter automatisk en orden som passer likningene vi skal løse. Dersom ordenen settes lik 1, får vi Backward Euler, som vi har sett på tidligere i kapitlet.

7.5.1 Reaksjonskinetikk med Scipy

Vi har nå en måte å løse de koblede ratelovene på med Scipy-biblioteket. Det som er lurt å gjøre først, er å omformulere ratelovene som én Python-funksjon som returnerer ratelovene som en array eller liste med tre elementer:

```
def ratelover(t,y):
    CHI = y[0]
    CH2 = y[1]
    CI2 = y[2]
    k = 4.84E-2
    dHIdt = k*CH2*CI2
    dH2dt = -0.5*dHIdt
    dI2dt = dH2dt
    return [dHIdt, dH2dt, dI2dt]
```

Her er det viktig å huske på at y må være en array med tre elementer: ett element for hver startbetingelse. Deretter formulerer vi ratelovene ut fra disse startbetingelsene. Ettersom funksjonen integreres, får vi nye verdier for `y[0]`, `y[1]` og `y[2]`.

> **Underveisoppgave 7.7**
>
> Studer funksjonen ovenfor og forklar hver linje med kode, og hva som skjer med disse verdiene når funksjonen integreres.

Videre kan vi løse og plotte ODE-ene på samme måte som tidligere:

```
a = 0
b = 500
t = np.linspace(a,b,100)
y0 = [0,1,1] # Array med startverdier
y_int = solve_ivp(ratelover, [a,b], y0, t_eval=t)

plt.xlabel("t")
plt.ylabel("c")
plt.plot(y_int.t, y_int.y[0]) # Plotter [HI]
plt.plot(y_int.t, y_int.y[1]) # Plotter [H2]
```

7.6 Oppsummering

Vi har sett på flere måter å løse (integrere) differensiallikninger på. Det viktigste du bør kunne fra dette kapitlet, er:

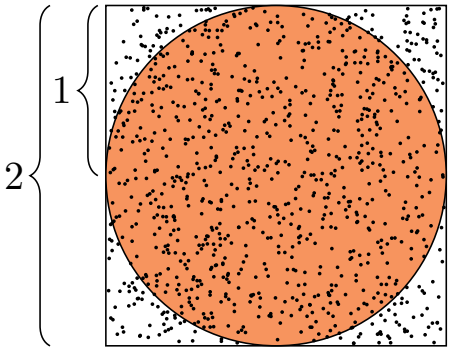
1. Forklare og utlede Forward Euler fra framoverdifferansen til den deriverte.

2. Implementere Forward Euler, både med og uten funksjoner.
3. Forklare hovedforskjellene mellom Forward Euler, Backward Euler og RK4.
4. Løse enkeltstående og sammenkoblede differensiallikninger, spesielt ratelover, med selvimplementert Eulers metode (Forward Euler) og andre metoder fra Scipy-biblioteket.

7.7 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 7.2: 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:

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

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:

```
import numpy as np

N = 100000
x=np.random.uniform(-1,1,N) # tilfeldig x-koordinat
y=np.random.uniform(-1,1,N) # tilfeldig y-koordinat

a = x**2 + y**2 # Sirkellikninga

# 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:", np.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 7.8

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:

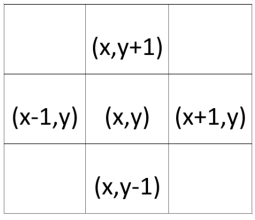
$$\epsilon = \frac{|v_{estimert} - v_{teoretisk}|}{v_{teoretisk}} \cdot 100\ \% \tag{7.16}$$

Estimeringen av pi er et eksempel på en *Monte Carlo-simulering* (MC), som vi så på i kapittel 6.3.5.

7.7.1 Tilfeldige bevegelser

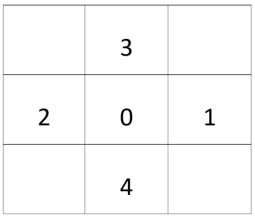
Vi skal her se på en MC-tilnærming til tilfeldig bevegelse av store partikler i løsning. Dette er en enkel modell for diffusjon av ikke-reagererende partikler som kan beskrive såkalte *Brownske bevegelser*. Brownske bevegelser ble først beskrevet av botanisten Robert Brown i 1827. Han oppdaga at små 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:



Figur 7.3: 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:



Figur 7.4: 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)
# Lagrer figuren som png-fil
plt.savefig('randwalk_%.d_steg.png' %(N))
plt.show()
```

Underveisoppgave 7.9

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.

Gi også 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?

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