Universitetet i Oslo

FYS2140
Introduksjonskompendium i programmering

Numerisk løsning av Schrödingers likning

Benedicte Emilie Brækken og Mari Røysheim

31. januar 2016

Innhold

Forord

Innenfor moderne vitenskap kan vi ved hjelp av datamaskinen og litt innsikt i programmering løse enormt avanserte problemer veldig enkelt — problemer som ofte ikke lar seg løse på papiret.

Universitetet i Oslo er et av få læringsinstitusjoner som har satset mye på å legge inn programmering som en naturlig del av den realfaglige utdanningen. Satsingen er kjent som CSE-prosjektet; Computing in Science Education. [1]

I hovedsak handler det om å utsette studenter for programmering allerede i første semester. Antatt at alle har en liten innsikt i programmering kan ellers "kjedelige" kurs gjøres enormt interessante ved å bruke reelle problemer i oppgavesettene i stedet for oppgaver som har masse forenklinger som skal gjøre det mulig å løse på papiret.

Med dette kompendiet ønsker vi å videreføre det CSE-prosjektet står for til kvantefysikk. Vi håper at du finner kompendiet nyttig og at det gir deg god motivasjon for å arbeide med dette utrolig spennende og veldig viktige feltet, samtidig som du utvikler din evne innenfor programmering.

Benedicte Emilie Brækken,

Desember 2013

[1] Du kan lese mer om prosjektet på hjemmesiden, `http://www.mn.uio.no/om/samarbeid/undervisningssamarbeid/cse/`

Kompendiet ble videreutviklet høsten 2015, og et kapittel om numerisk løsning av den tidsuavhengige Schrödingerlikniga ble tillagt. I den forbindelse ble det også gjort små endringer i det påfølgende kapittelet om den tidsavhengige Schrödingerlikninga.

Mari Røysheim,

Desember 2015

Hvordan bruke kompendiet

I dette kompendiet skal vi bruke programmeringsspråket Python til å løse både den tidsuavhengige og tidsavhengige Schrödingerligningen for et utvalg potensialer.

Kompendiet er i hovedsak rettet mot dem som ikke har mye erfaring med programmering fra før. Det vil derfor legges vekt på enkle og intuitive løsningsstrategier der dette er mulig. Vi unngår å skrive kompakt kode som gjør mye på én gang, men har heller fokus på at koden er i stand til å løse problemet (og vier effektiviteten mindre oppmerksomhet).

Vi løser først den tidsuavhengige Schrødingerlikninga, med den samme løsningsmetoden for alle eksempler som gis. Deretter avanserer vi til den tidsavhengige, og i den forbindelse introduserer vi noen litt mer kompliserte løsningsmetoder. Det er viktig å få med seg at det finnes et utall måter å løse problemer på så fort man gjør det numerisk, og at metodene presentert i dette kompendiet kun er en brøkdel av alle muligheter.

Antakeligvis vil kompendiet være til hjelp under hjemmeeksamen og numeriske obliger. Husk også at du lærer best, raskest og smartest ved å prøve deg fram på egenhånd. Ikke vær redd for å prøve å skrive kode og løse oppgaver. Det værste som skjer er at programmet ikke kjører, og du må tolke feilmeldingen du får. Der er det mye god læring å hente.

Lykke til.

Kapittel 1

Introduksjon

Dette kapittelet er todelt. Vi begynner først med å se på terminalen og fortsetter deretter til Python.

1.1 Terminalen

Terminalen er en måte å "kommunisere" med datamaskinen din på. Du skriver kommandoer og kan på den måten utforske datamaskinenen ved å gå gjennom mapper, kjøre filer og skrive tekst. Den gjør det veldig effektivt og behagelig å jobbe på datamaskinen når man har med programmering å gjøre.

Hvis du ikke helt forstår hva terminalen egentlig er, kan du trekke paralleller til et helt vanlig filutforsker-vindu bare at den er fullstendig tekstbasert.

For å starte den kan du på Macer finne den i Verktøy-mappen under Programmer. På linuxmaskiner finnes den ofte i lignende menyer.

Når du først starter terminalen ser den ofte noe slik ut:

```
bruker @ unix $
```

Linjen vil variere fra maskin til maskin, men vil som regel alltid inneholde navnet på brukeren, navnet på datamaskinen og et dollartegn. Dollartegnet markerer slutten og du får lov til å skrive inn ting bak det.

Et godt sted å begynne kan være å finne ut hvor du befinner deg på datamaskinen. Dette kan du gjøre ved å skrive:

```
bruker @ unix $ pwd
```

Og trykke enter. Da kommer det opp en linje som beskriver hvor du befinner deg på datamaskinen. Denne "adressen" om du vil kalles for flere ting. Blant annet *stien*, siden det beskriver en sti du kan følge for å komme deg til der du er. Eller *filbanen*. Men du kan bare tenke på det som mappen du er i, siden det er akkurat det det er.

For å liste opp alle filene som er i mappen du er i for øyeblikket bruker du `ls`.

```
bruker @ unix $ ls
fil1.txt fil2.txt bilde1.jpg minmappe
```

For å bytte mappe til "minmappe" kan jeg skrive `cd` for change directory.

```
bruker @ unix $ cd minmappe
```

Og for å komme meg ut igjen kan jeg skrive `../`.

```
bruker @ unix $ pwd
/minmappe/
bruker @ unix $ cd ../
bruker @ unix $ ls
fil1.txt fil2.txt bilde1.jpg minmappe
```

Denne kommandoen kan du bruke flere ganger etter hverandre siden den for hver gang går et hakk opp i mappehierarkiet. Det er for eksempel også gyldig å skrive:

```
bruker @ unix $ cd ../../../
```

Hvis du vil gå tre mapper oppover.

For å kopiere en fil bruker jeg `cp` (copy file) samt hvilken fil som skal kopieres og hvor den skal kopieres. Slik ser det ut hvis jeg vil kopiere inn `fil1.txt` til minmappe.

```
bruker @ unix $ ls
fil1.txt fil2.txt bilde1.jpg minmappe
bruker @ unix $ cp fil1.txt minmappe/
bruker @ unix $ ls
fil1.txt fil2.txt bilde1.jpg minmappe
```

```
bruker @ unix $ ls minmappe
fil1.txt
```

Når du skal skrive inn navnet på filer og mapper i terminalen er Tab tasten veldig nyttig. Hvis du for eksempel over begynner å skrive "min" og trykker deretter på Tab vil resten av minmappe bli fylt ut og du kan bare trykke Enter. Når man blir vant med å bruke denne går ting mye raskere!

Kommandoen mv brukes på samme måte som cp og brukes til å flytte en fil. Den kan også brukes til å endre navnet på en fil. Da "flytter" du bare filen til der du er, men med et nytt navn.

```
bruker @ unix $ ls
fil1.txt detteerenfil.txt bilde1.jpg
bruker @ unix $ mv detteerenfil.txt fil2.txt
bruker @ unix $ ls
fil1.txt fil2.txt bilde1.jpg
```

For å starte Python skriver du simpelthen

```
bruker @ unix $ python
```

Dette starter opp Python og forandrer noe på hvordan terminalen ser ut. For øyeblikket kan du overse det, da vi kommer mye innpå dette senere. For å få tilbake den vanlige terminalen trykker du Ctrl + D.

For å lage en mappe kan du bruke mkdir og for å slette en mappe kan du bruke rmdir. Legg merke til at for å bruke rmdir må mappen først være tom. Hvis det ikke er tilfelle og du ikke vil ta deg bryet med å fjerne filene én og én kan du bruke rm −r (r står for recursive).

NB! rm er det ikke mulig å reversere — filene legges ikke i en midlertidig søppelmappe slik vi er vant til.

Her er eksempel på bruk av mkdir

```
bruker @ unix $ ls
fil1.txt bilde1.jpg
bruker @ unix $ mkdir nymappe
bruker @ unix $ ls
fil1.txt bilde1.jpg nymappe
bruker @ unix $ mkdir endaenmappe
bruker @ unix $ ls
fil1.txt bilde1.jpg nymappe endaenmappe
bruker @ unix $ rmdir endaenmappe
bruker @ unix $ ls
```

Tabell 1.1: Liste med de mest brukte og nyttigste terminal-kommandoene.

pwd	Print working directory. Forteller hvilken mappe du er i.
ls	List files. Viser alle filene i nåværende mappe.
cd til	Change directory. Bytt mappe.
cp fra til	Copy. Kopier fil.
mv fra til	Move. Flytter en fil. Brukes også til å bytte navn på en fil.
rm fil	Remove. Sletter filen.
mkdir nymappe	Make directory. Lager en ny mappe som heter "nymappe".
rmdir mappe	Remove directory. Sletter den mappen du oppgir antatt at den er tom. Hvis ikke, bruk rm −r mappe.
man program	Manual. Slår opp i manualen til det programmet eller kommandoen du legger til. Veldig praktisk når du vet om en kommando men ikke hvordan du bruker den. Du kan bruke pilene for å gå oppover og nedover i teksten. For å få den bort trykker du q-tasten.

```
fil1.txt bilde1.jpg nymappe
```

I begynnelsen kan det virke litt mer kløvete å bruke terminalkommandoer enn å navigere for eksempel i Finder. Men jo mer du bruker dem, jo raskere går det, og til slutt virker terminalen som det enkleste alternativet.

1.2 Python

I forrige seksjon sa vi at du kan starte Python ved å skrive python i terminalen. Det som skjer da er at du starter et Python-shell. Et shell kan du tenke på som en arbeidsuniform til terminalen - den forandrer på oppførselen, utseende og hva du kan bruke den til. Det finnes mange typer shell, men de

to vi skal bruke er Python og bash. Bash er kun den vanlige terminalen som vi så på i forrige seksjon. Mens Python-shellet har kjennetegnet at linjen begynner med

```
>>>
```

Og at du etter den kan skrive inn Python kommandoer. Nå skriver vi altså i terminalen

```
bruker @ unix $ python
```

for å starte et Python shell. Legg merke til hvordan terminalen nå ser annerledes ut. Husk at for å avslutte Python-shellet og gå tilbake til Bash (vanlig terminal) kan du trykke Ctrl + D.

Nå går vi videre og ser på generell bruk av Python-shellet. I utgangspunktet fungerer Python som en kalkulator, vi kan for eksempel skrive:

```
>>> 2 * 4
8
```

De andre enkle matematiske operatorene fungerer nesten slik du ser for deg:

```
>>> 2 + 2
4
>>> 2 - 2
0
>>> 4 / 2
2
>>> 4**2
16
```

Det er viktig å huske at når man bruker / (dele på) operatoren og vil ha med desimaler, må iallefall ett av tallene være et flyttal. Dette kan gjøres ved å i stedet skrive **float**(2) i nevneren eller legge på et punktum slik at Python selv skjønner det. [1]

```
>>> 1 / 2   # Heltallsdivisjon
0
>>> 1 / 2.0
0.5
>>> float(1) / 2
0.5
```

[1] Det eneste dette utsagnet gjør er å konverte integeren 2 til flyttallet 2.0. En skulle tro at dette var trivielt, men i programmering er hvilken "type" tallet du bruker er, en veldig viktig sak.

Hovedpoenget er at hvis ikke minst ett av tallene allerede er et desimaltall, får vi heltallsdivisjon og på den måten mister vi informasjon.

For å bruke eksponentialer, altså skrive tall på formen $1 \cdot 10^3$, bruker vi bokstaven e på denne måten:

```
>>> print 1e3
1000.0
```

Husk å unngå fellen der du skriver:

```
>>> print 10e3
10000.0
```

Når det du egentlig ville ha var $1 \cdot 10^3$, fordi dette blir det samme som $10 \cdot 10^3$ så da får du en eksponent for mye.

1.2.1 Print kommandoen

For å skrive ut tekst eller data til terminalen bruker vi **print** kommandoen på følgende måte.

```
>>> print 'Dette er tekst.'
Dette er tekst.
```

Denne kommandoen kan brukes sammen med de fleste typer data. Og er veldig nyttig, siden vi ofte vil ha tilbakemelding fra programmet vårt om hva som foregår.

1.2.2 Representere og lagre data

Vi kan beholde tall eller data i variabler for å bruke de senere. For å lagre et tall i en variabel skriver vi:

```
>>> a = 1
>>> print a
1
>>> b = a + 3
>>> print b
4
```

Det er også mulig å lagre andre typer data, eksempler er da tekst (String), boolean (sann / falsk).

```
>>> sann = True
>>> b = 'Dette er tekst.'
>>> print sann
True
>>> print b
Dette er tekst.
```

Ofte vil vi oppbevare mange tall samtidig i en liste. I Python skriver vi lister på denne måten:

```
>>> min_liste = [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
>>> print min_liste
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Innenfor matematikken tenker en på funksjoner som kontinuerlige. Det betyr at man for hvilken som helst valgt x kan finne en funksjonsverdi $f(x)$. Når man da tegner grafen til $f(x)$ tar man hensyn til alle mulige x-verdier. Dette kan vi ikke realisere i programmering.

Grunnen er at vi kun kan ta vare på et forhåndsbestemt antall punkter, og evaluere funksjonen i hver av disse. Dette gjøres enkelt ved bruk av lister. Det er derfor lister er så viktige innenfor programmering. Vi trenger ikke (og kan ikke) ta med uendelig mange punkter, det klarer seg med mange *nok*. Jo flere punkt vi må ta vare på, jo mer datakraft brukes, og det er ikke nødvendig å lagre 1000 punkt dersom vi får et godt nok resultat med 500. Man må justere kravet til nøyaktighet etter hvor nøyaktig svar man trenger.

Som med helt vanlige variable kan vi også holde på andre typer data i lister:

```
>>> min_liste = [ 'Dette er tekst.', 2, 3, True, 'Mer tekst.' ]
>>> print min_liste
['Dette er tekst.', 2, 3, True, 'Mer tekst.']
```

Når vi skal hente ut spesifikke elementer fra listen setter vi firkantede parenteser etter navnet på listen. Det er viktig å merke seg at det første elementet i listen er 0:

```
>>> min_liste = [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
>>> print min_liste[0]
1
>>> print min_liste[2]
3
```

Siste elementet kan også hentes med -1, nest siste element med -2 osv:

```
>>> min_liste = [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
>>> print min_liste[-1]
9
>>> print min_liste[-2]
8
```

I tillegg kan vi bruke "slicing" for å hente ut et utvalg av listen. Da bruker vi syntaksen:

```
>>> print min_liste[ <fra og med> : <til (ikke med)> ]
```

En god huskeregel her er "fra og med men ikke til og med". Eksempel:

```
>>> min_liste = [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
>>> print min_liste[ 0 : 2 ]
[1, 2]
```

Her får jeg med meg elementene med indeks 0 og indeks 1.

Du kan finne ut hvor mange elementer en liste inneholder ved å bruke **len**(), *length*:

```
>>> min_liste = [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
>>> print len( min_liste )
9
```

Multidimensjonale lister

Som sagt er det mulig å hva hvilken som helst type data i disse listene. Det er også mulig å ha en annen liste inne i listen — da blir listen multidimensjonal. Innenfor fysikken bruker vi multidimensjonale lister for eksempel når vi skal representere et problem med x, y og z-verdier.

```
>>> min_liste = [ [1, 2, 3], [4, 5, 6], [7, 8, 9] ]
>>> print min_liste
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Legg også merke til at dette blir det samme som matriser i lineær algebra.

Når vi skal indeksere i disse listene setter vi bare flere firkantede paranteser etter hverandre:

```
>>> min_liste = [ [1, 2, 3], [4, 5, 6], [7, 8, 9] ]
>>> print min_liste[ 0 ][ 0 ]
1
>>> print min_liste[ 1 ][ 0 ]
4
>>> print min_liste[ -1 ][ -1 ]
9
```

Du kan tenke på dette som at med den første firkantede parantesen henter vi ut for eksempel det første elementet i listen.

```
>>> min_liste = [ [1, 2, 3], [4, 5, 6], [7, 8, 9] ]
>>> print min_liste[0]
[1, 2, 3]
```

Men siden det første elementet i listen også er en liste kan vi igjen hente ut det første elementet fra denne listen.

```
>>> a = min_liste[0]
>>> print a
[1, 2, 3]
>>> print a[0]
1
>>> print min_liste[0][0]
1
```

Legg merke til at de to nederste print-linjene henter ut det samme tallet. Det er dette som er ideen, og det fungerer for lister av flere enn 3 dimensjoner også.

1.2.3 Matematiske funksjoner — numpy

Python har mange verktøy innebygd, men ikke alle vi trenger for å gjøre våre dagligdagse matematiske beregninger. Til det formålet er det lagd flere biblioteker med funksjoner som vi kan bruke. Det mest populære av disse heter numpy.

For å ta det i bruk kan vi skrive:

```
>>> from numpy import *
```

Med denne linjen sier vi at vi fra et bibliotek som heter numpy skal vi hente alt (* asterisk betyr alt). Nå har vi tilgang til det meste vi skulle trenge, eksempler er π, e^x, $\sin(x)$ og mye annet. De brukes ved for eksempel:

```
>>> from numpy import *
>>> print sin( pi )
1.22464679915e-16
```

Her ser vi at svaret vi får faktisk ikke er 0, selv om det matematisk riktige er 0. Kan du tenke deg hvorfor det er slik? Husk at i programmering handler det som regel om approksimasjoner. Ettersom vi regner oppstår det avrundingsfeil og andre feil hele tiden. Jo flere operasjoner som blir gjort på et tall, jo mer nøyaktighet mister vi. Selv om feilen som regel er liten kan den ofte manifestere slik som her og forvirre oss hvis vi ikke er forberedt på det.

Vi kaller på en funksjon ved å skrive navnet på funksjonen (sin i dette tilfellet) etterfulgt av paranteser. Inne i parantesene gir vi argumenter. Her er funksjonen logisk nok $\sin(x)$, så det vi gir som argument er x, her π.

Eksempel på bruk av andre matematiske funksjoner:

```
>>> from numpy import *
>>> print exp( 1 ) # Eksponentialfunksjonen
2.71828182846
>>> print cos( pi )
-1.0
>>> print sqrt( 4 ) # Kvadratrot
2.0
```

Arrayer

De innebygde listene i Python er egentlig ikke ment for å bruke til regning. Det kan ofte være vanskelig å lese av hvordan de ser ut og mye avansert funksjonalitet vi skal se på senere fungerer ikke med de. Vi kommer derfor til å i stedet bruke en forbedring av de fra numpy — disse er kjent som arrayer.

Som regel trenger vi ikke tenke på dette, vi kommer til å se at når vi skal lage lister for å ta vare på datapunkter kommer vil til å lage dem gjennom numpy-funksjoner som automatisk gir oss arrayer. Likevel kan det være lurt å kjenne til hvordan du konverterer en liste til en array. Det gjøres på denne måten:

```
>>> from numpy import *
>>> min_liste = [ [1, 2, 3], [4, 5, 6], [7, 8, 9] ]
>>> min_array = array( min_liste )
>>> print min_array
```

```
[[1  2  3]
 [4  5  6]
 [7  8  9]]
```

Legg merke til hvordan matrisen (den multidimensjonale listen) formateres mye bedre når den er konvertert til en array og hvor mye lettere det da er å lese av hvordan den ser ut.

1.2.4 Nyttige funksjoner

Når vi skal gjøre numeriske beregninger i Python har vi et utvalg funksjoner som brukes veldig ofte. Disse gir oss tilbake arrayer eller lister med spesifikke egenskaper som vi vil bruke senere. De mest vanlige er listet i tabell 1.2.

Dette konkluderer det vi ser på av introduksjon til selve Python språket. I neste kapittel fortsetter vi med hvordan man kan samle flere slike Python-kommandoer til et større program ved å bruke en editor. Deretter går vi videre til litt mer avanserte strukturer i kapittel 3.

range(a, b)	Returnerer en liste med tall fra og med a opp til og med b − 1 der tallet øker med 1 for hvert steg. Hvis tallrekken din skal begynne på 0 trenger du bare å oppgi den øverste grensen.
	`>>> range( 5 )`
	`[0, 1, 2, 3, 4]`
	`>>> range( 5, 10 )`
	`[5, 6, 7, 8, 9]`

zeros(n)	Lager en ny array som kun inneholder nuller og er n lang. Fungerer også multidimensjonalt: zeros((nx, ny)), men da må argumentene ha et ekstra par paranteser rundt. Eksempel på bruk:
	`>>> from numpy import *`
	`>>> min_array = zeros( ( 1e4, 3 ) )`
	`>>> print min_array`
	`[[ 0. 0. 0.]`
	`[ 0. 0. 0.]`
	`[ 0. 0. 0.]`
	`...,`
	`[ 0. 0. 0.]`
	`[ 0. 0. 0.]`
	`[ 0. 0. 0.]]`
	Det finnes også en lignende funksjon ones() som gjør akkurat det samme. Men den fyller arrayen med et-tall i stedet for null. Det kan hende du får bruk for det også.

linspace(a, b, n)	Lager en ny array fylt med tall fra og med a til og med b med n jevnt fordelte tall. Eksempel på bruk:
	`>>> from numpy import *`
	`>>> min_array = linspace( 0, 5, 5 )`
	`>>> print min_array`
	`[ 0. 1.25 2.5 3.75 5. ]`

Tabell 1.2: Vanlige numpy-kommandoer.

Kapittel 2

Editorer og tekst

Fra før er du sikkert vant til at filer lagres når du bruker et visst program og for å åpne filen igjen må du bruke nøyaktige det samme programmet.

For eksempel når du skrev stiler på videregående var det som regel Word-filer som skulle leveres inn. Og hvis du var skikkelig uheldig fikk ikke læreren åpnet filen din og du strøk på oppgaven. Hvorvidt det er riktig måte å skrive stilen på eller ikke er ikke det vil skal innpå her. Men vi skal se litt på lukkede og åpne rammer. Burde det du skriver være avhengig av nøyaktig ett program for å kunne åpnes?

Etterpå går vi videre til å se på hvordan man kan kjøre Python-kommandoer etter hverandre ved hjelp av script og litt om forskjellige skriveprogrammer.

2.1 Fra lukkede til åpne rammer

Når du hopper ned i verden av programmerere og datamaskiner vil du plutselig oppleve at de fleste filer inneholder "ren tekst". Med ren tekst menes at filene ikke er fjerne datafiler skapt av f.eks. Word, men er filer som kun inneholder den teksten du har skrevet som rene ASCII-tegn.

Du er sikkert klar over at datamaskinen bruker det binære tallsystemet for å gjøre alt den gjør. ASCII er et av flere kode-systemer som blir brukt for å assoisere et binært tall med et tegn eller symbol. Ren tekst blir altså helt rå data datamaskinen kan lese selv.

Slike filer har du nok vært borti før et eller annet sted. Mange programmer har for eksempel med seg en "readme.txt"-fil. Dette er en ren tekst fil.

Når vi skal programmere må all koden vi skriver skrives inn i slike filer fordi den delen av datamaskinen som tolker instruksene ikke kan bruke tung programvare som Word. Vi må altså kvitte oss med denne uvanen og begynne å skrive i ren tekst.

Nettopp for dette formålet finnes det en hærskare med programvare som alle har sine egne særegenheter, fordeler og ulemper. I tillegg vil du innse at innenfor programmeringskretser er hvilken "editor" (det programmet du bruker til å skrive programmer i) virkelig en følelsessak. Dette er altså noe som er veldig viktig for dem! Og hvis du selv lærer deg å bruke en av de som har hendige funksjoner for å gjøre skrivingen mer effektiv kommer du til å spørre deg selv hvordan du klarte deg uten.

Dette er litt vanskelig å forklare bort, men jeg kan ta et eksempel. Dette kompendiet er et nokså elegant dokument med mange fancy-innlegg som bokser med kode som inneholder farger, referanser som er dynamiske og andre ting. Samtidig er ikke dette et dokument som er skrevet i et program som Word eller lignende. Det er faktisk kodet med et programmeringsspråk kjent som LaTeX (uttales la-tekj, eller la-tech). Det som skiller LaTeX fra andre programmeringsspråk som Python er at i stedet for å ha instruksjoner som datamaskinen skal regne med osv., produserer det et dokument som er formatert etter de innstillingene du skriver i kodefilen.

LaTeX er også veldig mye brukt innenfor vitenskapen på grunn av at det gjør matematikk så enkelt å formatere på datamaskinen.[1]

Det at informasjon lagres i ren tekst på denne måten eliminerer også nødvendigheten av at både den som skriver dokumentet og den som skal lese det må ha samme programmet. Begge benytter en åpen standard som ikke er avhengig av store selskapers dyre programpakker - dette er veldig viktig for mange. I tillegg ville det eliminert problemet videregåendelæreren hadde med Word-filer.

Pdf (som dette dokumentet er lagret i) er også et godt eksempel. Det er en høy sannsynlighet for at vi ikke bruker de samme programmene for å lese pdf-filer. Men likevel ser det helt likt ut.[2]

[1] Hvis du vil lære mer om LaTeX og hvordan du kan skrive obligene dine i det kan du se `www.latex-project.org/guides/`

[2] Det burde iallefall det, siden det er det som er målet med åpne standarder. Men det finnes også rare proprietære løsninger som kan ha uforutsigbare resultater.

2.2 Python-programmer

Som regel er det ikke nok å kun ha et Python-shell å jobbe i. Det er vanskelig når man skal gjøre mange ting etter hverandre. Derfor faller det naturlig å nøste sammen instruksjonene til en lengre oppskrift der alt kjøres av seg selv i den rekkefølgen det står.

Det finnes veldig mange forskjellige editorer som fungerer bra til å skrive Python. Noen har innebygde funksjoner som terminal og Python-shell mens andre kun er et vindu du kan sette tekst i (som for eksempel Notepad på Windows eller TextEdit på mac). De fleste har derimot til felles at de har syntaks highlighting, som gjør underverker for lesbarheten til koden.

Når du skal skrive en Python-fil er det bare å lage en ny fil og skrive kommandoene etter hverandre som linjer. Deretter lagres filen med filendelsen ".py" og kjøres fra terminalen på denne måten:

```
bruker@unix$ ls
mitt_program.py
bruker@unix$ python mitt_program.py
Jeg er et eksempel!
Forresten, 2 + 2 = 4
```

Pythonfilen jeg netopp kjørte så slik ut

```
print 'Jeg er et eksempel!\nForresten, 2 + 2 = %g' % ( 2 + 2 )
```

Under følger en liten introduksjon av et par forskjellige editorer som kan brukes til Python.

2.3 Gedit

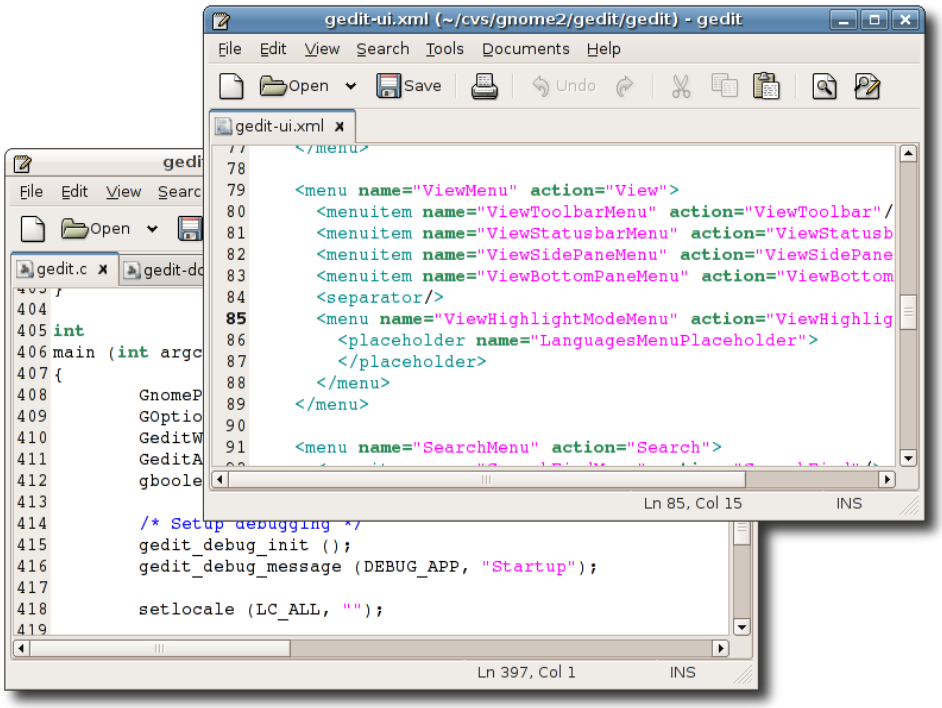
Dette er editoren vi anbefaler hvis du har Ubuntu eller jobber på en av maskinene på Universitetet.

Gedit er en rett-fram editor som fungerer akkurat slik som du ser for deg et tekstprogram skal fungere. Det har syntaks-highlighting og det er i grunnen det. Men til de enkle programmene vi bruker i fysikken trenger du ikke en særlig mer avansert editor.

For å skrive Python i Gedit er det nok å åpne editoren og begynne å skrive. Hvis ikke fargene oppfører seg riktig kan det være lurt å lagre filen og passe

på få med ".py"-endelsen. Da forstår Gedit at dette er en Pythonfil og at den skal oppføre seg deretter.

Hvis du vil installere Gedit på din egen maskin kan du se på hjemmesiden her, det skal finnes ferdige versjoner for både Linux, Mac, Windows.[3]



Figur 2.1: Gedit

2.4 Spyder

Spyder er et IDE. Det står for "integrated developing environment" og betyr rett og slett at programmet har alt du trenger for å skrive og kjøre programmene dine i ett og samme program.

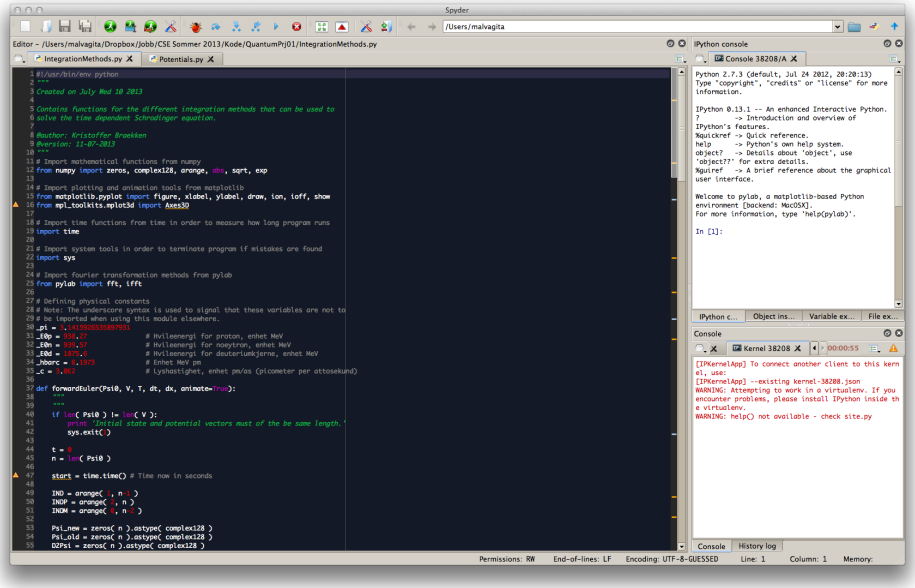
IDE'er er praktiske når du programmerer en del og fint å ha når du skal teste siden du slipper å bytte vindu til terminalen hele tiden.

[3] `https://projects.gnome.org/gedit/`

Spyder er gratis og kan lastes ned fra hjemmesiden deres her. [4] Når du har installert det er det bare å lage en ny fil og begynne å skrive. Snarveistastene kan du sjekke i instillinger og lære deg dem hvis du vil bli rask på tastaturet.

Ulempen med et IDE er at det ofte inneholder mye mer enn det du egentlig trenger og derfor blir det ofte et stort og tungt program å kjøre hvis du bare skal skrive en kjapp liten fysikk-simulering.

I tillegg legger det inn mye kode som kun er der for god konvensjon men som strengt tatt ikke er nødvendig og som kan forvirre hvis du ikke har sett det før.



Figur 2.2: IDEet Spyder kjørende på en Mac.

2.5 Andre

Som nevnt finnes det flust av forskjellige edtorer. Manges favoritt er en hendig liten editor som kalles for VIM. Denne er også gratis og ferdig installert i de fleste Linux-systemer. Hvis du er vant med å bruke denne vil det for-

[4] `http://code.google.com/p/spyderlib/`

enkle veldig mye og gjøre deg veldig kjapp til å skrive. Men den har en bratt læringskurve.

Du kan sjekke om du har den ved å gå i terminalen og skrive:

```
bruker @ unix $ vim
```

Hvis den er installert vil den da åpne seg. Hvis du vil lære deg den finnes det også en fin guide som du kan få opp ved å skrive:

```
bruker @ unix $ vimtutor
```

Macer har vim i terminalen innebygd. Men det finnes også skikkelige utgaver av den. En god versjon er da Macvim som du kan finne på prosjekthjemmesiden her. [5]

En annen veldig populær editor som ligner litt på VIM er Emacs. Den er grunnlagt på GNU og er helt åpen programvare. Den har en noe flatere læringskurve, så det er litt greiere å lære seg den. Og den har det meste av funksjonalitet som VIM har. Det er liksom disse to som er de mest brukte.

Grunnen til at de er så populære er at de har et enormt potensiale for personalisering. Altså, du kan forandre på mer eller mindre hva som helst med dem og få dem til å oppføre seg (og se ut) akkurat slik du selv vil, med litt sjonglering med kode.

De er ikke noe IDE, til å begynne med, men finner du riktig tillegg kan du bruke dem til å navigere i filer, debugge programmer, sortere koden og mye annet.

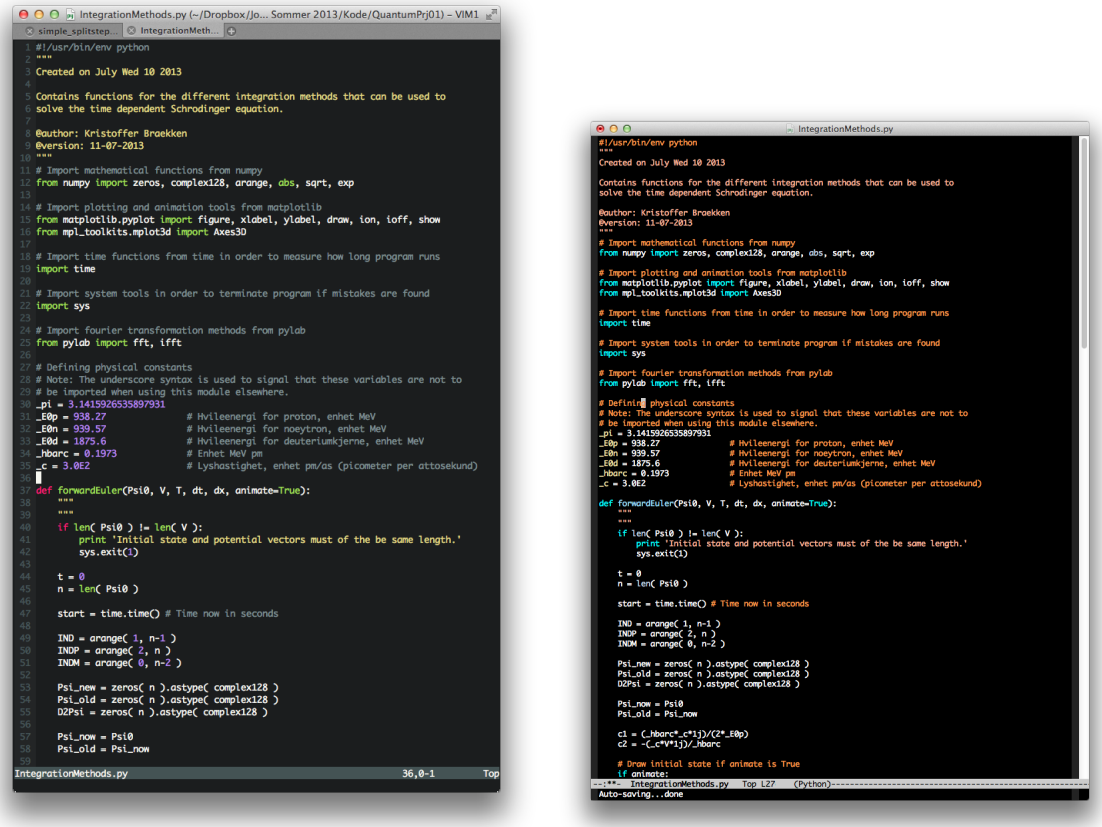
Denne friheten til å sjonglere med kode har skapt et par morsomme fenomener. Blant annet er Emacs kjent for å ha sin egen psykolog, svarene er muligens noe begrenset. Men det er en festlig opplevelse. Har du Emacs installert og åpent kan du prøve den ved å trykke Esc + x, skrive “doctor” og trykke enter.

På Blindern er det en pågående editor-krig mellom informatikerne og fysikerne. Mens fysikerne sverger til Emacs sverger informatikerne til VIM. Tar du et par informatikk-fag ved siden av fysikkfagene kommer du til å oppleve dette. Det sier noe om hvor viktig dette faktisk er for mange.

Du finner prosjekthjemmesiden til Emacs med nedlastinger for Linux og

[5] `http://code.google.com/p/macvim/`

Windows her. [6] Som regel er Emacs allerede installert på Linux-maskiner. På Mac finnes det en enkel terminal-versjon hvis du går i terminalen og skriver emacs. Det finnes også en god Mac-versjon her. [7]



Figur 2.3: Editorene VIM (til venstre) og Emacs (til høyre) med noe Python-kode.

[6] http://www.gnu.org/software/emacs/
[7] http://emacsformacosx.com/

Kapittel 3

Mer kontrollstruktur i Python

I kapittel 1 så vi på grunnleggende bruk av Python som kalkulator med matematiske funksjoner. Her skal vi gå litt videre fra Python-shellet og over i verdenen av programmer. Som nevnt i kapittel 2 er et program kun en samling av Python-kommandoer. Men når man skriver lengre programmer er det enklere å bruke litt mer avansert funksjonalitet. Det er dette vi skal se på her.

3.1 Funksjoner

I seksjon 1.2.3 så vi bruk av matematiske funksjoner vi får utenfra gjennom numpy. Dette er derimot ikke den eneste måten å få funksjoner på. Vi kan lage våre egne for å forenkle beregninger senere.

Si at vi for eksempel vil lage funksjonen (3.1) i Python.

$$f(x) = 4x^2 + 2x - 2 \tag{3.1}$$

For å gjøre dette må vi lære oss hvordan vi lager våre egne funksjoner. Det gjør vi på denne måten:

```
def f( argumenter, adskilt, med, komma ):
    '''
␣␣␣␣Kort␣tekst␣som␣med␣ord␣forklarer␣hva␣funksjonen␣gjoer␣og␣hva
␣␣␣␣som␣kommer␣ut␣av␣den.
␣␣␣␣'''
    return ( funksjonen gir dette tilbake )
```

Denne funksjonen kan nå kalles på med linjen

```
f( like, mange, argumenter, her )
```

Vi kan skrive om funksjonen (3.1) til Python på denne måten:

```
def f( x ):
    '''
␣␣␣␣Regner␣ut␣et␣andregradspolynom.
␣␣␣␣'''
    return 4 * x**2 + 2 * x - 2
```

Hvis vi nå vil evaluere denne funksjonen for et tall kan vi bare kalle på den:

```
>>> print f( 0.5 )
0.0
>>> print f( 1 )
4
```

Nå kan du prøve dette ved å lage alle mulige funksjoner. Det er også ikke begrenset hvor mange argumenter funksjonen din kan ta. Si hvis du for eksempel har en funksjon som er avhengig av både tiden t og en x-koordinat, f.eks. en bølgefunksjon, kan du lage en funksjon på følgende måte:

```
from numpy import *

def wave( x, t ):
    '''
␣␣␣␣Funksjonen␣beskriver␣en␣sinusboelge␣ved␣tiden␣t␣og␣paa␣
    punktet
␣␣␣␣x.
␣␣␣␣'''
    omega = 1
    k = 1

    return sin( k * x - omega * t )
```

Legg merke til at vi her også lagde nye variable innenfor funksjonen, omega og k. Disse vil eksistere hver gang funksjonen kjører og kan bare nås fra innenfra funksjonen.

3.2 If-tester

Noen ganger så vil vi at ting skal skje i programmet kun hvis visse krav er oppfylt. For å utrette dette kan vi bruke if-tester. If-tester bygges opp på

denne måten:

```
if tilfelle1:
    < utfoer noe >
elif tilfelle2:
    < utfoer noe annet >
else:
    < utfoer hvis ingen av de andre slaar til >
```

For å sjekke om ting stemmer eller ikke utnytter vi oss som regel av operatorene ==, som betyr er det samme som, != som betyr er ulik fra. Samt videre > og < som forklarer seg selv. Men også >=, som betyr større enn eller lik og <= som betyr mindre enn eller lik.

```
a = 1

if a == 1:
    print 'Ja, a er lik en.'
elif a == 2:
    print 'a er jo to.'
else:
    print 'a er alt annet.'
```

Når programmet over kjøres vil det produsere resultatet

```
Ja, a er lik en.
```

siden a er lik 1.

Det er viktig å få med seg at det er ikke nødvendig hver gang du skal sjekke noe å ha med både **elif** og **else**, det hadde holdt med kun linjen:

```
a = 1

if a == 1:
    print 'Ja, a er lik en.'
```

Så hvilke du skal ha med avhenger av hvor mye du skal sjekke. Det er kun obligatorisk å få med den første **if** for at det skal fungere.

Obs: når du skal sjekke om noe er likt noe annet med == er det viktig å huske på det vi har vært innom tidligere om at de fleste matematiske operasjoner i datamaskinen er approksimasjoner. Dette kan gjøre at når du vet noe skal være nøyaktig 0 er det bare et veldig lite tall og ødelegger if-testen din. For å unngå det kan du i stedet bruke en liten toleranse og sjekke på den måten:

```
tolerance = 1e-15

if abs( x ) < tolerance:
    print 'x er 0.'
```

Her bruker vi absoluttverdien og sjekker om den er mindre enn en gitt toleranse. Hvis den er det, er tallet 0.

Det er også mulig å sjekke om to krav er oppfylt i samme **if**-testen. Det kan gjøres med ordene **and** og **or**. For eksempel:

```
a = 1
b = 2

if a == 1 and b == 2:
    print 'Hit kommer vi hvis a er 1 og b er 2.'
elif a == 1 or b == 2:
    print 'Hit kommer vi hvis enten a er 1 eller b er 2.'
```

3.3 Indentering

Hver gang du lager kontrollstrukturer som funksjoner, if-tester eller ting vi skal se på senere som løkker er det viktig å være obs på indenteringen i Python.

Indentering handler om hvor stort innrykk du har etter for eksempel linjen

```
def f( x ):
```

Det er da viktig at indenteringen er den samme gjennom hele funksjonen. Det debatteres stadig hva som er best indentering, noen bruker 2 mellomrom mens andre bruker 4 mellomrom. Det vanligste er nok å bruke Tab-tasten for å lage indentering og den lager som regel 4 mellomrom indentering.

Legg merke til at indenteringen ikke må være den samme gjennom HELE funksjonen, hvis du for eksempel har en if-test innenfor funksjonen din, må du ha et nytt innrykk der, de legger seg oppå hverandre altså. Det er også lurt å passe på at innrykket du har etter if-testen er like stort som det du hadde til funksjonen. Eksempel:

```
def f( x ):
    if x == 2:
        return 3
```

3.4 Løkker

Det mest hendige med datamaskiner er at de ikke blir lei av å gjøre det samme på nytt og på nytt. Når vi har oppgaver som skal gjøres om og om igjen for et visst sett med tall eller parametre kan vi utnytte løkker.

La oss for eksempel skrive ut hvert eneste partall mellom 0 og 20. Dette hørtes ut som kjedelige greier. Men vi kan gjøre det enkelt i Python ved å ha en teller som teller fra 1 til 20 og skrive ut tallet hvis resten du får når du deler det på to er null (da er det et partall). Vi får tak i resten av en divisjon med modulo-operatoren % (prosenttegnet).

Modulo-operatoren brukes på denne måten:

```
>>> print 5 % 3
2
```

Vi får 2 i rest når vi deler 5 på 3 siden 3 går en gang opp i 5 og vi da sitter igjen med 2.

For å lage en løkke med en teller i som teller fra 1 til og med 20 kan vi bruke en **for**-løkke. **for**-løkker brukes til å kjøre gjennom lister og har følgende syntaks:

```
for < navn paa teller > in < listen vi vil kjoere gjennom >:
    < det som skal gjoeres hver gjennomkjoering >
```

For å lage en liste som begynner på 1 og får med seg 20 bruker vi rett og slett **range**(1, 21).

```
for i in range( 1, 21 ):
    if i % 2 == 0:
        print i
```

Disse tre linjene skriver ut alle partall mellom 0 og 21. Legg merke til hvordan vi brukte en if-test inne i en løkke for å kontrollere hva som skulle skje.

for-løkkene er derimot ikke den eneste typen løkker en har tilgjengelig. Du kan også bruke **while**-løkker. Disse kjører så lenge et visst krav er oppfylt. De kan også kjøre i all evighet hvis du f.eks. glemmer å oppdatere det som sjekker kravet ditt hver gang løkka kjører. Hvis du kjører et Python-program og det aldri stopper kan du avbryte det ved å taste Ctrl + C. Det er veldig nyttig når du skal debugge programmer.

Hvis vi skriver om det lille programmet som skrev ut partall med en **while**-løkke ville det sett slik ut:

```
counter = 1

while counter <= 20:
    if counter % 2 == 0:
        print counter

    counter += 1
```

Det er to ting som er forskjellige. For det første trenger vi en teller for å holde styr på hvor vi er, denne må oppdateres manuelt hver gang løkka kjører. I tillegg ser du at da vi skrev løkken brukte vi et krav - akkurat som i if-tester. **and** og **or** vil fungere på samme måten her også.

3.5 Vektorisering

Som vi har sett er det mulig å utføre gjenntatte beregninger for et sett med verdier ved bruk av en løkke. Men løkker blir fort lange og trege når det er mye som skal skje. For å unngå dette har vi et godt verktøy i Python — vektorisering.

Vektorisering handler om at hvis du har en array med verdier og vil kjøre en funksjon på alle tallene samtidig, kan det gjøres ved ett kall på funksjonen. Som vist i her:

```
from numpy import *

x = linspace( 0, 2 * pi, 10 )

# Regner ut sin(x) for alle tallene i x
f = sin( x )
```

Dette er et veldig sterkt verktøy og kan ofte forkorte lange nestede løkker til kun en hovedløkke. I tillegg vil programmet kjøre raskere siden det er færre løkker i det.

Hvis vi skal bruke det på våre egne funksjoner må vi passe på at funksjonen vi lager kan ta et tall som parameter og returnerer et tall.

```
from numpy import *

def f( x ):
```

```
    '''
    Et andregradspolynom.
    '''
    return 2 * x**2 - 3x + 1

x = linspace( 0, 10, 100 )
y = f( x )
```

Du kan for eksempel ikke direkte kjøre if-tester på argumentet i funksjonen fordi testen vil da kjøre på hele arrayen samtidig, og det er ikke funksjonalitet som finnes. Derimot finnes det annen praktisk funksjonalitet du kan bruke for å unngå if-tester, men disse går vi ikke gjennom her.

3.6 Komplekse tall

En stor del av kvantefysikken bruker komplekse tall i utregningene sine. Python har også støtte for komplekse tall, men av og til kan det trenge litt ekstra innsats.

I utgangspunktet bruker man, logisk nok, bokstaven j for å representere den imaginære operatoren. i blir derfor 1j i Python (legg merke til at det ikke er mulig å skrive bare j). Følgende eksempel viser litt bruk av komplekse tall i Python og funksjoner som er nyttige:

```
>>> a = 2 + 1j
>>> b = 2j
>>> print a + b
(2+3j)
>>> print a.imag # Hente kun imaginaer del
1.0
>>> print a.real # Hente kun real del
2.0
>>> print abs(a) # Absoluttverdien
2.23606797749979
>>> print abs(a)**2
5.000000000000001
```

3.6.1 Komplekser tall i arrayer

Som vi skal se senere bruker vi ofte å ha lange arrayer som skal inneholde komplekse tall mens vi regner på de. Av og til kan dette føre med seg feilmel-

dinger i sammenheng med bruk av funksjoner som zeros og ones — numpy lar oss ikke lagre komplekse tall i dem på grunn av ugyldig type.

Man kan unngå dette problemet ved å spesifikk bestemme at arrayen skal inneholdet komplekse tall. Dette gjøres ved kun å legge på et punktum og en funksjon på slutten av for eksempel zeros-kommandoen:

```
from numpy import *
Psi = zeros( n ).astype( complex64 )
```

3.7 Typisk oppbygning

Det er veldig mange konvensjoner ute og går som inneholder retningslinjer om hvordan man burde strukturere og skrive koden sin for at den skal bli så enkel som mulig for andre å lese. Her kommer jeg til å vise en generell struktur og si litt om lesbarhet.

Et typisk Python-program vil se slik ut:

```
'''
Laget < datoen dokumentet ble lagd >.

< Tekstbeskrivelse av hva programmet gjoer. >

@author: < Hvem som har skrevet programmet >
@version: < En maate aa identifisere hvilken versjon, f.eks dato
    >
'''
< alle importer >

< alle funksjoner >

if __name__ == '__main__':
    < det som skal skje i programmet, her bruker du funksjoner,
      utfoerer beregninger osv >

'''
< kjoereeksempel, det skal vise hvordan programmet kjoeres og
  hva som kommer ut i terminalen >
'''
```

Mesteparten her forklarer seg selv, slik som informasjonen om hvem som har skrevet programmet, når det ble lagd osv. Det er derimot to nye ting

som kanskje ser litt ukjente ut. Det ene er en if-test nederst som sjekker om __name__ og '__main__' er like.

__name__ er en variabel som Python lager av seg selv. Denne variabelen vil alltid inneholde strengen '__main__' unntatt hvis filen blir importert.

Importering har vi vært borti før, men vi har aldri sett på hvordan vi kan utnytte det selv. Poenget er at vi kan skrive flere Python-filer, ha de i samme mappe og bruke funksjoner fra dem i forskjellige filer. Hvis jeg for eksempel skriver en fil:

```
def f( x ):
    return x*x
```

og lagrer filen som "funksjon.py". Kan jeg opprette en ny fil der jeg skriver:

```
from funksjon import *

print f( 2 )
```

Som jeg lagrer som "program.py". Lagres disse to i samme mappe vil det printes ut 4 hvis jeg kjører "program.py":

```
bruker @ unix $ ls
funksjon.py program.py
bruker @ unix $ python program.py
4
```

Poenget er at innenfor filen "funksjon.py" vil __name__ nå være strengen 'funksjon' i stedet for '__main__' siden det er "program.py" vi kjører og ikke "funksjon.py".

Men hva i alle dager skal vi med dette? Hvis vi lager større programmer der vi for eksempel vil lagre alle funksjonene våre i et eget program, da kan det hende at vi vil ha noe kode i det programmet som ikke skal kjøres når vi importerer den, men bare kjøres hvis vi kjører den direkte. Når vi importerer en annen fil, er det det samme som å kjøre den.

Det er dette if-testen er til, den kjører kun koden som kommer etter den hvis det er innenfor den filen du faktisk ville kjøre. Ved å ha all koden som gjør noe (ikke funksjoner, importer osv) innenfor denne if-testen unngår vi masse trøbbel hvis vi skal importere den senere. Det er altså god konvensjon å gjøre det på denne måten. Selv om bruksområde kanskje kan være litt vanskelig å forstå til å begynne med.

Den andre nye tingen er tekst-strengen nederst i programmet. Denne gjør ingenting når programmet kjøres, men er kun der for å beskrive hvordan programmet kjøres og hva slags output (hva som printes osv.) det har. Det hjelper ofte hvis det kommer mye tekst ut av programmet å forklare hva som faktisk kommer ut. Det er kjent som et kjøreeksempel.

Her er et eksempel som inneholder litt tekst også:

```
'''
Laget␣Ons␣24.␣juli␣2013.

Programmet␣viser␣hvordan␣et␣typisk␣Pythonprogram␣ser␣ut.

@author:␣Benedicte␣Emilie␣Braekken
@version:␣24-07-2013
'''
from numpy import *

def func1( x ):
    '''
␣␣␣␣Gjoer␣ingenting.
␣␣␣␣'''

    return 0

def func2 ( x, y ):
    '''
␣␣␣␣Gjoer␣ingenting.
␣␣␣␣'''

    return 0

if __name__ == '__main__':
    ingenting = func1( 1 ) + func2( 2 )
    print 'Dette␣er␣tekst.'

'''
bruker␣@␣unix␣$␣python␣filnavnet.py
Dette␣er␣tekst.
'''
```

Følger du denne oppskriften kommer du til å få pene programmer som andre lett kan sette seg inn i.

Kapittel 4

Visualisering og animasjon

Ofte vil vi illustrere tallene vi har regnet på ved hjelp av grafiske verktøy for å lettere kunne fortelle om hva vi har oppdaget samt skaffe oss en intuisjon av hvordan det vi regner på faktisk ser ut. Da kan vi plotte og animere.

I dette kapittelet skal vi gå gjennom diverse måter man kan visualisere data på med Python. Først går vi gjennom vanlig todimensjonal plotting, deretter videre til animasjon og til slutt hvordan man kan lage tredimensjonale figurer.

4.1 Plottebiblioteker

For å kunne plotte med Python må vi importere funksjonaliteten vi trenger utenfra siden dette ikke er noe Python har innebygd. I dette kompendiet kommer vi til å fokusere på en pakke som heter matplotlib. Denne skal være innebygd i de fleste Python-distribusjoner. Det finnes også på datalabene.

Vi kan importere alle funksjoner fra matplotlib ved å skrive

```
from matplotlib.pyplot import *
```

Dette gir oss alle funksjonene vi trenger for å plotte 2-dimensjonal data - x og y.

4.2 Todimensjonal plotting

For å illustrere hvordan man kan plotte en graf med matplotlib vil vi plotte sannsynligheten til grunntilstanden for en harmonisk oscillator som vi møter i kurset. Denne er gitt av bølgefunksjonen,

$$\Psi_0(x,0) = \left(\frac{m\omega}{\pi\hbar}\right)^{1/4} \cdot e^{-\frac{m\omega}{2\hbar}x^2}.$$

Først importerer vi det vi trenger for å plotte samt alle de matematiske funksjonene fra numpy

Listing 4.1: Import av matematiske verktøy samt plotteverktøy.

```
from numpy import *
from matplotlib.pyplot import *
```

Deretter må vi lage et sett med x-verdier å plotte for, samt definere konstantene våre m, ω og $\hbar$. For et proton kan vi slippe å skrive masse og $\hbar$ ved å multiplisere med c^2 oppe og nede i begge brøkene. Formelen blir da

$$\Psi_0(x,0) = \left(\frac{m_p c^2 \omega}{\pi\hbar c^2}\right)^{1/4} \cdot e^{-\frac{m_p c^2 \omega}{2\hbar c^2}x^2}$$

$$\Psi_0(x,0) = \left(\frac{E_{0,p}\,\omega}{\pi(\hbar c)c}\right)^{1/4} \cdot e^{-\frac{E_{0,p}\,\omega}{2(\hbar c)c}x^2} \tag{4.1}$$

Nå inneholder formelen kjente verdier slik at det er lett å velge riktig enhet og unngå veldig små tall. Vi vet at $\hbar c = 0.1973$ MeV pm, $E_{0,p} = 938.27$ MeV, $c = 3 \cdot 10^2$ pm / as. Bestemmer konstantene og lager et sett med x-verdier fra $-\pi$ til π.

```
# Kun for aa teste
omega = 1               # [rad / s]

# Fysiske parametre
hbarc = 0.1973          # [MeV pm]
E0p = 938.27            # [MeV]
c = 3e2                 # [pm / as]

# x-verdier
x = linspace( -pi, pi, 1e4 )
```

Videre må vi implementere funksjonen (4.1). Dette er veldig rett frem nesten bare å skrive om fra matematikk til Python

```
def Psi0( x ):
    '''
    Grunntilstanden for en harmonisk oscillator.
    '''
    A = ( E0p * omega / ( pi * hbarc * c ) )**0.25
    B = exp( - E0p * omega / ( 2 * hbarc * c ) * x**2 )
    return A * B
```

Vi skaffer så en array med funksjonsverdier ved å kalle på funksjonen vi nettopp lagde med arrayen x.

```
Psi = Psi0( x )
```

Nå har vi to sett med tall - x'er og funksjonsverdier. Vi vil plotte disse mot hverandre. Antatt at vi har kjørt importene i **Listing 4.1**, begynner vi med å lage en ny tom figur (et tomt vindu).

```
figure()
```

For å plotte x mot $|\Psi_0|^2$ skriver vi.

```
plot( x, abs( Psi0 )**2 )
```

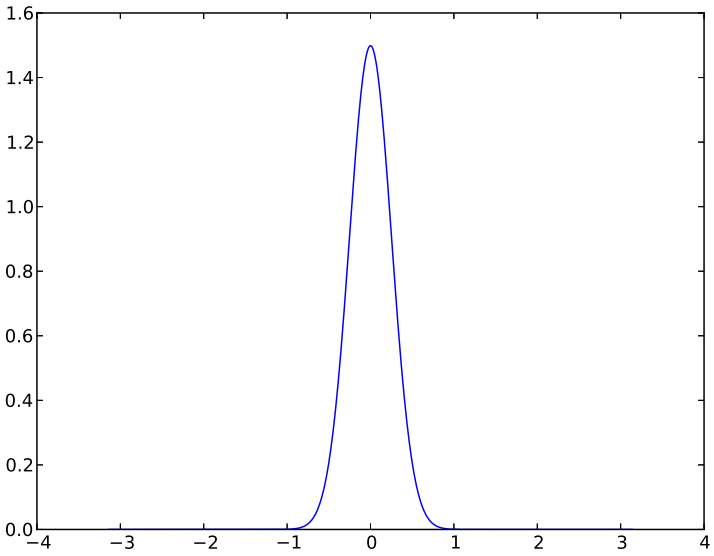
Til slutt må vi kalle på en funksjon til for å vise det vi har plottet.

```
show()
```

Denne funksjonen fryser programmet og viser figur(ene) vi har lagd. Programmet fortsetter ikke før vi har krysset av vinduene. I vårt tilfelle skal programmet produsere figuren **Figur 4.1**.

Det er også enkelt å sette tekst langs aksene og som tittel i plotte. Da kan du bruke følgende kommandoer. [1]

```
# Tekst langs x-aksen
xlabel('$x$ [pm]')

# Tekst langs y-aksen
ylabel('$|\Psi_0(x,0)|^2$ [1/pm]')

# Tittel paa plottet
title('Grunntilstanden for harmonisk oscillator')
```



Figur 4.1: Grunntilstanden $\Psi_0(x, 0)$ for en harmonisk oscillator.

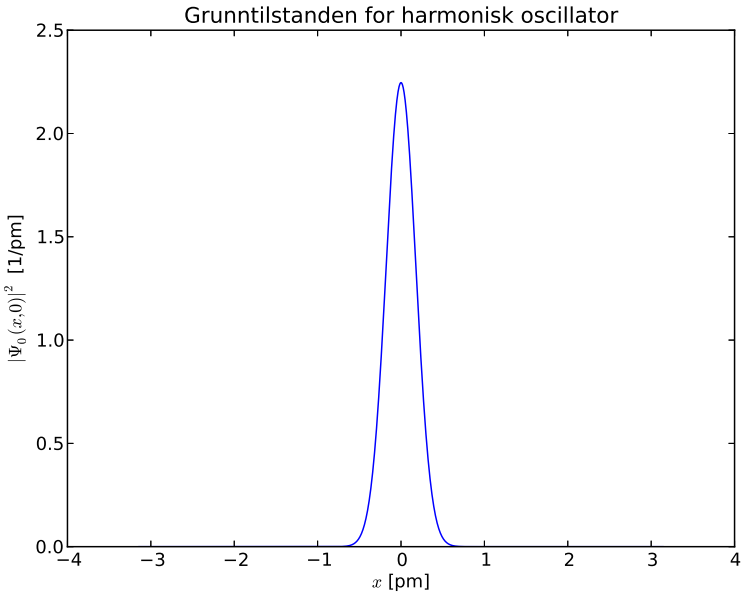
Da ser figuren ut som i **Figur 4.2**.

Settes alt det vi har gjort til nå sammen til et program vil det se ut som i **Listing 4.2**.

Listing 4.2: Et program som plotter initialtilstanden $\Psi_0(x, 0)$ for en harmonisk oscillator.

```
from numpy import *
from matplotlib.pyplot import *

# Kun for aa teste
omega = 1               # [rad / s]

# Fysiske parametre
hbarc = 0.1973          # [MeV pm]
E0p = 938.27            # [MeV]
c = 3e2                 # [pm / as]

# x-verdier
x = linspace( -pi, pi, 1e4 )
```

[1]Det er brukt LaTeX i tekst-strengene for å få matematiske tegnsetting. Derfor er det dollartegn på hver side. Du kan lese mer om LaTeX på `www.latex-project.org/guides/`



Figur 4.2: Grunntilstanden $\Psi_0(x, 0)$ for en harmonisk oscillator med tekst langs aksene og tittel.

```
def Psi0( x ):
    '''
    Grunntilstanden for en harmonisk oscillator.
    '''
    A = ( E0p * omega / ( pi * hbarc * c ) )**0.25
    B = exp( - E0p * omega / ( 2 * hbarc * c) * x**2 )
    return A * B

# Henter funksjonsverdier og lagrer i arrayen Psi
Psi = Psi0(x)

# Lager et nytt figurvindu
figure()

# Plotter x mot Psi0
plot( x, abs( Psi )**2 )

# Tekst langs x-aksen
xlabel('$x$ [pm]')

# Tekst langs y-aksen
ylabel('$|\Psi_0(x,0)|^2$ [1/pm]')

# Tittel paa plottet
title('Grunntilstanden for harmonisk oscillator')

# Viser det vi har plottet
show()
```

4.3 Flere vinduer

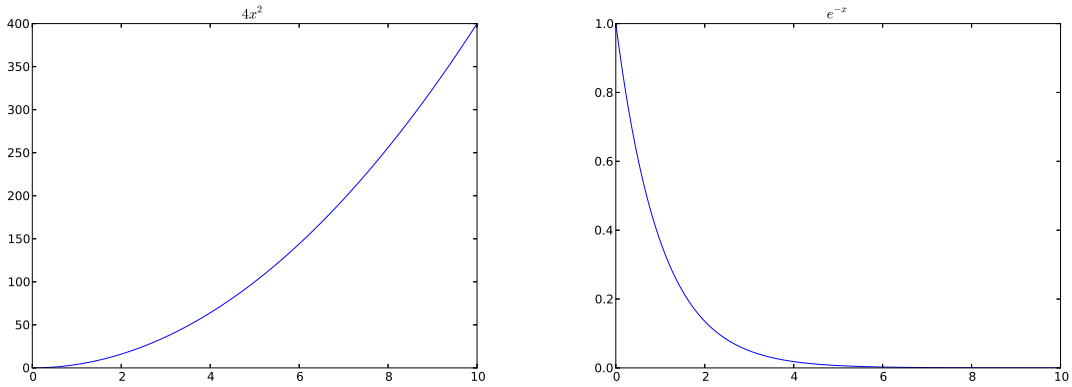
Noen ganger vil vi plotte flere forskjellige vinduer fra samme program. Dette kan gjøres enkelt ved å kalle på figure() flere ganger. Hver gang vi skriver plot(x, y) etter det vil det komme i en ny figur.

Dette er illustrert i **Listing 4.3**. Her lager programmet de to figurene vist i **Figur 4.3**.

Listing 4.3: Et program som lager to figurer.

```
from numpy import *
from matplotlib.pyplot import *

# Definerer to tilfeldig valgte funksjoner
def func1( x ):
```

```
    return 4*x**2

def func2( x ):
    return exp( -x )

# Lager x-verdier
x = linspace( 0, 10, 1e3 )

# Lager foerst en figur og plotter i den
figure()
plot( x, func1( x ))
title('$4x^2$')     # Setter tittel

# Lager en figur til og plotter i den
figure()
plot( x, func2( x ))
title('$e^{-x}$')   # Setter tittel

# Viser alle plottende
show()
```



Figur 4.3: De to figurene lagd av programmet i **Listing 4.3**.

Det er som regel lurt å vente med selve kallet på show() til slutt i programmet siden denne kommandoen fryser programmet og viser alle figurene som er lagd. Det er derfor ikke nødvendig å kalle den etter hver gang du lager en ny figur.

4.4 Animasjon

Snart skal vi begynne å se på hvordan vi kan utvikle Schrödingerligningen fremover i tid. Forandringen kan da best illustreres ved å "animere", altså vise et vindu der grafen forandrer seg ettersom tiden går. Dette kan oppnås ved hjelp av det vi har sett på sålangt samt noen ekstra kommandoer.

matplotlib slik vi har brukt det sålangt har vært veldig statisk. Ingenting kom på skjermen før vi hadde kalt på show(). For å få vinduet til å oppdatere seg ved hvert kall trenger vi tre funksjoner til: ion(), ioff() og draw().

ion() og ioff() står for interaktiv modus på (on) og av (off). Interaktiv modus gjør at matplotlib oppdaterer vinduet hver gang vi forteller at vi har oppdatert datane. For å fortelle dette kaller vi på draw().

Gangen i et typisk animeringsprogram blir derfor seende noe slik ut:

```
< importer >

def f( x, t ):
    '''
    En eller annen funksjon som er avhengig av
    en parameter som forandrer seg.
    '''
    return f(x,t)

T = < slutt tid >
dt = < tidssteg >
t = 0

x = < array med tall >
y = f( x, 0 ) # Funksjonen ved t = 0

figure() # Lager vinduet
line, = plot( x, y ) # Plotter paa valig maate
draw() # Fortelle at det er nye data

ion() # Interaktiv modus paa

while t < T:
    # Hente ut nye data
    y = f( x, t )

    # Forandre y-verdiene paa plottet vaart
    line.set_ydata( y )
    draw() # Fortelle at det er nye data
```

```
    # Oppdatere tiden
    t += dt

ioff() # Slaa av interaktiv modus
show() # Passe paa at figurene ikke blir borte
```

Her setter vi en tid T som er hvor lenge simuleringen skal kjøre, lager en Δt (som vi kaller for dt) for å si hvor langt tiden går for hvert tidssteg. Samt en variabel t til å holde på den nåværende tiden.

Vi gjør dessuten noe nytt når vi plotter; vi tar vare på det som kommer ut av plot() på denne måten. [2]

```
line, = plot( x, y )
```

den enkle forklaringen på dette er at her må vi lagre linjen (samlingen av x'er og y'er som gir datane) matplotlib bruker fordi da kan vi lett forandre på den senere gjennom metodene set_ydata(), set_xdata() og set_data(). [3]

4.4.1 Eksempel: Animasjon av sinus-bølge

Nå har vi sett nok til å lage et enkelt eksempel. Her skal vi animere en enkel sinusbølge beskrevet ved:

$$f(x,t) = \sin(kx - \omega t) \tag{4.2}$$

Der k er bølgetallet og ω er vinkelhastigheten. I Python implementerer vi denne funksjonen slik:

```
from numpy import *

def wave( x, t ):
    '''
    Funksjonen beskriver en sinusboelge ved tiden t og paa
    punktet
    x.
    '''
```

[2] Legg merke til kommaet etter ordet line. Funksjonen plot returnerer egentlig en liste med alle "komponentene" eller linjene som finnes i plottet. Siden vi her kun plotter en ting, vil funksjonen returnere en liste med ett element. Det er dette ene elementet vi vil ha ut og kommaet henter det ut for oss. Prøv det ut selv i Python ved å lage en liste som består av f.eks kun ett tall!

[3] Vi kommer mer innpå hva dette betyr og bruken av det litt senere i kapittelet under funksjonsanimasjon og tredimensjonal plotting.

```
    omega = 1    # Vinkelhastighet
    k = 1        # Boelgetall

    return sin( k * x - omega * t )
```

Videre trenger vi, som nevnt over, noen parametre for å holde styr på hvor lenge animasjonen skal kjøre, hvor langt fremover i tid den går for hvert steg i løkken osv. I tillegg må vi sette hvilke x'er som skal brukes og hvor mange punkter.

```
n = 1e3
t = 0
T = 10
dt = 0.01
x = linspace( -pi, pi, n )
```

Satte her x'ene til å gå fra $-\pi$ til π. Videre må vi tegne initialtilstanden vår. Utnytter at vi satte t til 0.

```
figure()
line, = plot( x, wave( x, t ) )
draw()
```

Til slutt må vi skrive beregningsløkken. Denne skal oppdatere datane i line samt tidstelleren.

```
while t < T:
    line.set_ydata( wave( x, t ) )
    draw()

    t += dt
```

Setter vi alt dette sammen og husker å sette matplotlib i interaktiv modus vil programmet se noe slik ut som i **Listing 4.4**.

Listing 4.4: Et program som animerer en sinusbølge ved hjelp av matplotlib.

```
from numpy import *
from matplotlib.pyplot import *

def wave( x, t ):
    '''
␣␣␣␣Funksjonen␣beskriver␣en␣sinusboelge␣ved␣tiden␣t␣og␣punktet␣x
    .
␣␣␣␣'''
    omega = 1    # Vinkelhastighet
    k = 1        # Boelgetall
```

```
    return sin( k * x - omega * t )

n = 1e3
t = 0
T = 10
dt = 0.01
x = linspace( -pi, pi, n )

ion()

figure()
line, = plot( x, wave(x, t) ) # Plotter initialtilstand naar t =
    0
xlim([-pi, pi]) # Setter x-aksen til aa gaa fra -pi til pi.
draw()

while t < T:
    line.set_ydata( wave( x, t ) )
    draw()

    t += dt

ioff()
show()
```

4.4.2 Funksjonsanimasjon

Vi kan også nærmere oss animeringen på en annen måte: Å først lage alle datane for så å bruke funksjons-animasjon til å kjøre gjennom settet vårt. Dette gjør det litt mindre krevende å kjøre animasjon. Med det menes at siden de nye punktene ikke må regnes ut på nytt for hver gang plottet skal oppdateres, kan vi oppdatere plottet flere ganger og få "filmen" til å se ut som den går mye glattere.

Ulempen er at du ikke får se animasjonen før alt er regnet ut. Derfor hvis en har mange punkter som skal regnes ut, tar det lang tid før du får opp resultater og hvis de resultatene da er feil kan det virke veldig demotiverende i lengden.

For å få gjort noe av dette må vi importere animasjonspakken fra matplotlib:

```
from matplotlib import animation
```

Før vi ser på hvordan vi skal animere må vi først forandre på programmet vårt slik at i stedet for å overskrive Psi-arrayen hver gang lagrer vi i stedet alle datane for alle tidsstegene. Dette kan vi gjøre ved å lage en multidimensjonal liste.

Lister har den enorme fordelen at vi ikke trenger å spesifisere hvor lang den skal være på forhånd. Vi kan opprette den tom, også bruke en funksjon append() for å legge til et nytt element på enden av listen. For eksempel kan vi legge til den nåværende Psi på slutten av listen for hver gang. Da får vi en liste som ser slik ut til slutt:

$$Psi = \begin{bmatrix} \Psi(\vec{x}, 0) \\ \Psi(\vec{x}, \Delta t) \\ \vdots \\ \Psi(\vec{x}, T) \end{bmatrix}$$

Der hvert element, for eksempel $\Psi(\vec{x}, 0)$ er en array i seg selv som inneholder $\Psi(\vec{x})$ for det tidspunktet.

Utregningsløkken blir derfor seende litt annerledes ut. Den vil ha en typisk slik form:

```
Psi = [] # Lager ny tom liste

while t < T:
    Psi_new = # Regner ut ny Psi

    # Legger den til i Psi-listen
    Psi.append( Psi_new )

    t += dt
```

Når utregningen har kjørt må vi begynne å lage det vi trenger for å sette sammen den store matrisen (multidimensjonale listen) vi nå har til en animasjon. For å gjøre dette skal vi bruke en funksjon som heter FuncAnimation(). Den skal vi fore med 6 argumenter.

1. Figuren som skal vise animasjonen. Denne lager vi ved å skrive:

   ```
   fig = figure()
   ```

så lenge vi har husket å importere hele pyplot-pakken tidligere.

```
from matplotlib.pyplot import *
```

2. Animasjonsfunksjonen. Dette er en funksjon som tar ett argument frame som sier hvilken steg som skal plottes på dette tidspunktet. Funksjonen skal gi tilbake den linjen som korresponderer til nåværende frame.

   For eksempel i starten blir argumentet 0 gitt, da skal funksjonen gi tilbake plottet for $\Psi(\vec{x}, 0)$.

   Siden vi kommer til å opprette et line-objekt før selve animasjon akkurat slik som i forrige seksjon kan vi bruke line.set_data(x, y) for å gi tilbake den nåværende linjen: [4]

```
def animation( frame ):
    '''
    '''
    line.set_data( x, abs( Psi[ int( frame ) ] )**2 )
    return line,
```

   Her forutsetter vi at vi har en tabell Psi som inneholder $\Psi(\vec{x})$ for alle t'ene våre.

   Når FuncAnimation lager filmen vi ser, kjører den gjennom animasjonsfunksjonen vår med frame satt til alle heltall fra 0 til og med antall tidssteg vi har satt (minus en slik at det passer med indekseringen). Det er derfor vi kan bruke frame-argumentet for å vite hvilken koordinat av matrisen vi skal sende tilbake.

3. Rensefunksjonen. Dette er en funksjon som ligner på selve animasjonsfunksjonen, men den tar ingen argumenter og skal bare returnere en linje som ikke inneholder noe. Vi kan skrive funksjonen på akkurat samme måte som animasjonsfunksjonen og bare sette linjen til ikke å inneholde noe.

```
def init():
    '''
    '''
    line.set_data( [ ], [ ] )
    return line,
```

[4] Legg merke til at vi bruker objektet line inne i funksjonen. Med andre ord må vi sørge for at det blir opprettet et sted utenfor før vi faktisk kaller på animation. Dette blir mye enklere å se når vi har satt sammen hele programmet.

I animeringen blir denne funksjonen brukt bak kulissene for å "rense" vinduet mellom hver gang et nytt bilde skal tegnes.

4. Antall bilder. Dette er en integer som forteller hvor mange bilder animasjonen skal tegne. I vårt tilfelle vil dette være antall tidssteg.

5. Forsinkelsen mellom hvert nye bilde. Dette er et tall som sier hvor lenge animasjonen skal vise hvert bilde. En god måte å definere denne på er å lage en variabel FPS, som forteller hvor mange bilder du vil ha per sekund. Dette er en så naturlig størrelse at den er lett å bestemme ut av det blå. For eksempel er 60 en veldig pen animasjon mens 30 også vil fungere bra.

   Forsinkelsen mellom hvert bilde vil da være

```
FPS = 60
forsinkelse = 1. / FPS
```

   Her har vi sagt at vi vil ha 60 bilder per sekund.

6. Blit. Dette er et boolean argument, hvilket betyr at den skal være enten True eller False. Det den sier er hvorvidt animasjonen alltid skal tegne hele bildet på nytt eller bare tegne det som har forandret seg. Målet er å få en mer effektiv animasjonsprosess.

   I vårt tilfelle vil nok forskjellen den utgjør være minimal. Men for animasjoner der det er mye som foregår på en gang vil den gjøre en større forskjell. Det er nok derfor lurt å gjøre seg vant med å ha den med.

4.4.3 Eksempel: Funksjonsanimasjon av sinus-bølge

Nå kan vi lage det samme eksempelet med animasjon av en sinusbølge med funksjonsanimasjon i stedet og se på forskjellen.

Siden vi nå skal utføre hele regningen på forhånd — før vi animerer må vi vite nøyaktig hvor mange tidssteg vi skal ta. Hvis vi på forhånd har en variabel T som sier hvor lenge simuleringen skal kjøre og en variabel dt som forteller størrelsen på tidssteget. Kan vi enkelt finne antallet tidssteg:

```
nt = int( T / dt )
```

Her konverterer vi den også til en integer slik at vi slipper å gjøre det senere.

Videre lager vi den tomme listen som skal ta vare på alle tilstandene våre:

```
all_waves = []
```

Deretter kan vi skrive løkken, som blir seende slik ut:

```
while t < T:
    all_waves.append( wave( x, t ) )

    t += dt
```

Nå kan vi tegne initialtilstanden og lage line-objektet som vi kommer til å bruke i rense-funksjonen og animasjonsfunksjonen.

```
fig = figure()
line, = plot( x, all_waves[0] )
draw()
```

Her plotter vi altså rett og slett første koordinaten i matrisen. Vi må også ta vare på det figure gir tilbake slik at vi kan gi den til FuncAnimation etterpå.

Nå kan vi sette animasjons-konstantene samt lage rensefunksjonen og animasjonsfunksjonen.

```
# Konstanter til animasjonen
FPS = 60 # Bilder i sekundet
inter = 1. / FPS # Tid mellom hvert bilde

def init():
    '''
␣␣␣␣'''
    line.set_data( [], [] )
    return line,

def get_frame( frame ):
    '''
␣␣␣␣'''
    line.set_data( x, all_waves[ frame ] )
    return line,
```

Legg merke til hvordan get_frame kun bruker frame argumentet til å hente ut riktig element fra matrisen.

Til slutt må vi lage selve animasjonen samt kalle på show slik at vi får se animasjonen.

```
anim = animation.FuncAnimation( fig, get_frame, init_func=init,
                                frames=nt, interval=inter, blit=
                                    True )

show()
```

Settes alt sammen til ett program burde det se noe ut som i **Listing 4.5**.

Listing 4.5: Et program som bruker funksjonsanimasjon til å animere en sinusbølge.

```
from numpy import *
from matplotlib.pyplot import *
from matplotlib import animation

def wave( x, t ):
    '''
    Funksjonen beskriver en sinusboelge ved tiden t og punktet x
    .
    '''
    omega = 1    # Vinkelhastighet
    k = 1        # Boelgetall

    return sin( k * x - omega * t )

T = 10
dt = 0.01
nx = 1e3
nt = int( T / dt ) # Antall tidssteg
t = 0

all_waves = [] # Tom liste for aa ta vare paa boelgetilstandene
x = linspace( -pi, pi, nx )

while t < T:
    # Legger til en ny boelgetilstand for hver kjoering
    all_waves.append( wave( x, t ) )

    t += dt

# Tegner initialtilstanden
fig = figure() # Passer paa aa ta vare paa figuren
line, = plot( x, all_waves[0] )
draw()

# Konstanter til animasjonen
FPS = 60 # Bilder i sekundet
inter = 1. / FPS # Tid mellom hvert bilde
```

```
def init():
    '''
␣␣␣␣'''
    line.set_data( [], [] )
    return line,

def get_frame( frame ):
    '''
␣␣␣␣'''
    line.set_data( x, all_waves[ frame ] )
    return line,

anim = animation.FuncAnimation( fig, get_frame, init_func=init,
                                frames=nt, interval=inter, blit=
                                    True )

show()
```

Hvis du når du kjører det programmet kun får opp den hvite bakgrunnen uten noen graf i den, kan det hende at matplotlib ikke bruker riktig backend. Backendet er det som oversetter informasjonen matplotlib lager til vinduer og bilder som vises på dataskjermen. Du kan løse dette ved å legge inn disse to linjene helt i toppen av programmet:

```
import matplotlib
matplotlib.use('TkAgg')
```

Linjene forteller matplotlib at det skal bruke et backend som støtter funksjonsanimasjon. [5]

4.5 Tredimensjonal plotting

matplotlib gjør det også mulig å plotte tredimensjonal data. Sett for eksempel at vi har en funksjon som er avhengig av både x og y, kan vi lage oss et sett med tall for x og y, evaluere dem for en 2-dimensjonal funksjon $f(x, y)$ og få ut en tredimensjonal figur.

For å kunne ta denne funksjonaliteten i bruk må vi importere 3D-pakken fra matplotlib:

```
from mpl_toolkits.mplot3d import Axes3D
```

[5] Hvis animasjonen fungerer helt fint er det bare å ignorere dette.

I tillegg må vi bruke matplotlib på en litt annen måte, som vi skal se. I seksjon 4.4.2 var vi såvidt borti det objektorienterte API-et til matplotlib — hvordan vi kan bruke matplotlib på en annen måte ved å ha informasjonen i *objekter* i stedet for bare å kalle på funksjoner. Dette betyr bare at vi tar vare på "objektene" eller det funksjoner som figure og plot kaster ut for senere å manipulere det de viser. Nå skal vi bruke det litt mer slik at vi tegner 3D-figurer i stedet for vanlige 2D-grafer.

Først og fremst må vi ha en funksjon som vi vil plotte. Her bruker vi rett og slett en todimensjonal gauss-kurve. Den er definert som:

$$f(x, y) = Ae^{-\left[(x-x_0)^2/2\sigma_x^2 + (y-y_0)^2/2\sigma_y^2\right]} \tag{4.3}$$

Vi kan implementere den i Python som en funksjon som tar to argumenter:

```
from numpy import *

def gauss_2d( x, y ):
    '''
    Todimensjonal gauss-kurve.
    '''
    # Konstant
    A = 1

    '''Sentrum og bredden (sigma) kan settes her eller tas som
       argumenter'''
    x0 = 0
    y0 = 0
    sigma_x = 1
    sigma_y = 1

    X = ( x - x0 )**2 / ( 2 * sigma_x**2 )
    Y = ( y - y0 )**2 / ( 2 * sigma_y**2 )

    return A * exp( - ( X + Y ) )
```

Når det er gjort må vi definere det "rommet" vi vil se på funksjonen i. Siden vi nå går over i tre dimensjoner er det ikke lenger nok med kun en liste med tall. Vi må i stedet ha to, en for x-retning og en for y-retning. I tillegg må vi bruke en funksjon som heter meshgrid for å lage de matrisene som faktisk representerer det rommet vi er i.

Dette er nødvendig fordi hvis vi kun lager to arrayer med tall får vi kun verdiene langs x- og y-aksene i stedet for også alle mulige kombinasjoner av tall som utgjør alle punktene i rommet. Man kan tenke på x- og y-tallene vi

lager som akser til rommet. Men vi trenger matriser for å representere alle mulige punkter i rommet. Det er veldig enkelt å lage dem:

```
from numpy import *

# Aksevektorene i hver retning
x = linspace( -4*sigma_x, 4*sigma_x, n )
y = linspace( -4*sigma_y, 4*sigma_y, n )

# Lager de to tabellene som inneholder hvert punkt i rommet
# de to aksevektorene over utspenner
X, Y = meshgrid( x, y )
```

Arrayene X og Y inneholder nå tabeller som sammen representerer rommet x og y utspenner. Når vi senere skal kalle på funksjonen gauss_2d er det altså X og Y vi må bruke, ikke x og y.

Dette er alt vi må gjøre i forkant. Nå vil vi lage selve plottet. Først og fremst lager vi en figur på vanlig måte, men tar vare på den i en variabel:

```
fig = figure()
```

Videre må vi manuelt legge inn et bilde i figuren, det gjør vi ved å bruke add_subplot funksjonen. Vi skal fore den med to argumenter. Det ene er tre heltall etter hverandre, 111. Dette er et helt spesielt argument, det er ikke matematisk. Argumentet sier at her har vi ett plott i høyden, ett plott i bredden og det plottet vi legger til nå er plott nummer 1. Når vi bare skal plotte en ting er det nok med 111 altså.

Når vi skal plotte flere ting i samme vinduet kan vi bruke dette argumentet som vi vil. F.eks hvis vi vil ha to plott i høyden kan vi bruke argumentene 211 (for det øverste) og 212 (for det nederste). Legg merke til at det er det siste tallet i kombinasjonen som bestemmer hvilket plott du velger akkurat der og da.

Det andre argumentet sier kun at bildet skal være en tredimensjonal figur:

```
ax = fig.add_subplot( 111, projection='3d' )
```

Vi kaller bildet ax av god konvensjon. Det kommer av at det bildet som ligger i figuren innenfor matplotlib-dokumentasjonen er kjent som akser - man legger til akser i figuren og akser kan være så mangt. I vårt tilfelle blir det her tre akser.

Legg også merke til at vi skriver fig punktum add_subplot. Det er derfor vi måtte ta vare på fig i forrige steg, vi bruker *objektet* fig for å legge akser i det. [6]

Videre kan vi bruke ax-objektet for å legge data i det. Antatt at vi har lagd den todimensjonale funksjonen samt definert et rom å evaluere i, da kan vi bare utføre plottingen. Innenfor tredimensjonal plotting er det mange typer plott vi kan velge i. Blant annet et overflateplott, konturplott osv. I vårt eksempel kan det passe seg å velge et overflateplott. Da må vi bruke funksjonen plot_surface med ax-objektet:

```
# Utfoerer plott
ax.plot_surface( X, Y, gauss_2d( X, Y ) )

# Viser plottet
show()
```

Setter vi alt dette sammen til et helt program vil det se ut som i **Listing 4.6**. Og det vil produsere figuren **Figur 4.4**.

Listing 4.6: Et program som lager et overflateplott med en todimensjonal gauss-funksjon.

```
from matplotlib.pyplot import *
from numpy import *
from mpl_toolkits.mplot3d import Axes3D

def gauss_2d( x, y ):
    '''
    Todimensjonal gauss-kurve.
    '''
    # Konstant
    A = 1

    X = ( x - x0 )**2 / ( 2. * sigma_x**2 )
    Y = ( y - y0 )**2 / ( 2. * sigma_y**2 )

    return A * exp( - ( X + Y ) )

# Antall punkter hver retning (reelt blir det n^2 for hele
    rommet)
n = 3e2

# Senteret gauss-kurven ligger paa
```

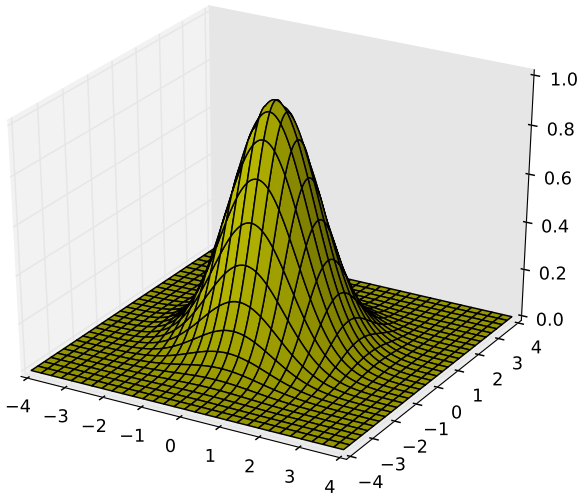
[6] Dette kalles for objektorientert programmering og benytter seg av noe som heter klasser. Innenfor programmering er objektorientering veldig stort. Du kan lære mer om objektorientering med Python i kurs som INF1100 og INF3331.

```
x0 = 0
y0 = 0

# Bredden paa gauss-kurven
sigma_x = 1
sigma_y = 1

# Enhetsvektorne i hver retning
x = linspace( x0 - 4. * sigma_x, x0 + 4. * sigma_x, n )
y = linspace( y0 - 4. * sigma_y, y0 + 4. * sigma_y, n )

# Lager de to tabellene som inneholder hvert punkt i rommet
# de to enhetsvektorne over utspenner
X, Y = meshgrid( x, y )

# Lager figur
fig = figure()

# Lager akser
ax = fig.add_subplot( 111, projection='3d' )

# Lager overflateplott, color='y' gjoer figuren gul
ax.plot_surface( X, Y, gauss_2d( X, Y ), color='y' )

# Viser
show()
```

Legg merke til at slike plott fort kan bli tunge hvis det er for mange punkter, det lønner seg derfor å velge en n som ikke er for høy med mindre funksjonen varierer veldig. Her er det mulig å prøve seg frem til man finner en som er stor nok for problemet man skal visualisere.

Det finnes i tillegg til overflateplottet flere typer plott man kan bruke i tre dimensjoner. For å lese mer om disse kan du se i matplotlib-dokumentasjonen her. [7]

[7] http://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html



Figur 4.4: En todimensjonal gauss-kurve produsert av programmet i **Listing 4.6**.

Kapittel 5

Den tidsuavhengige Schrödingerlikninga

Vi er nå endelig fremme ved kvantefysikken i dette kompendiet, og vi begynner med å se på hvordan vi kan finne numeriske løsninger for den tidsuavhengige Schrödingerlikninga (TUSL). Som vi har lært så beskriver kvantefysikken systemer ved hjelp av bølgefunksjoner med komplekse verdier, $\Psi(x,t)$, som er løsninger av Schrödingerlikninga. Absoluttverdikvadratet, $|\Psi(x,t)|^2$, tolker vi som sannsynlighetstettheter, og er det eneste som kan måles fysisk.[1]

Bølgefunksjonen $\Psi(x,t)$ er generelt tidsavhengig, men i dette kapittelet skal vi begrense oss til å finne løsninger av typen

$$\Psi(x,t) = \psi(x)\phi(t) = \psi(x)e^{iEt}, \tag{5.1}$$

hvor løsningen kan separeres i to funksjoner, ψ og ϕ, som er henholdsvis posisjons og tidsavhengig. Dette kalles **stasjonære tilstander**, og vi har vist på forelesning at tidsavhengigheten kun er en kompleks fase som avhenger av energien E, og som forsvinner når vi regner ut sannsynlighetstettheten. De stasjonære tilstandene er altså tilstander der sannsynlighetstettheten er konstant i tid — ingenting "skjer". Vårt mål er å finne $\psi(x)$, som er løsninger av den **tidsuavhengige Schrödingerlikninga** (TUSL).[2]

[1] Det er verdt å merke seg konsekvensen av at alle fysiske observable avhenger av absoluttverdikvadratet. Det betyr at løsninger som skiller seg på en faseforskjell på e^{ia}, hvor a er en reelt konstant (ikke tidsavhengig), vil beskrive samme fysikk. For eksempel er et minustegn bare en faseforskjell på $e^{i\pi} = -1$, og irrelevant for fysikken.

[2] Det er viktig å påpeke at vi her begrenser oss til potensialer som kun er posisjonsav-

En utledning av hvordan vi kommer fram til TUSL ved hjelp av teknikken med separasjon av variable for differensiallikninger er gitt i Griffiths, og vi minner kun om at den kan skrives som:

$$\left(-\frac{\hbar^2}{2m}\frac{d^2}{dx^2}+V\right)\psi(x) = E\psi(x), \tag{5.2}$$

$$\hat{H}\psi = E\psi. \tag{5.3}$$

Dette er ei egenverdilikning, hvor ψ er egenfunksjonen til Hamiltonoperatoren $\hat{H}$, med den tilhørende egenverdien E. Det er verdt å merke seg at vi praktisk nok alltid kan finne reelle løsninger av TUSL.[3]

For å bevise dette, anta at en kompleks bølgefunksjon $\psi(x)$, som alltid kan skrives som $\psi(x) = \alpha(x) + i\beta(x)$, hvor α og β er relle funksjoner, er en egenfunksjon for $\hat{H}$. Altså at

$$\hat{H}\psi = E\psi.$$

Da er

$$\hat{H}(\alpha + i\beta) = E(\alpha + i\beta)$$
$$\hat{H}\alpha + i\hat{H}\beta = E\alpha + iE\beta.$$

For å oppfylle likninga, må den reelle biten av venstresiden være lik den reelle biten av høyresiden, og tilsvarende for den imaginære biten. For den reelle biten får vi:

$$\hat{H}\alpha = E\alpha,$$

samtidig gir den imaginære biten at:

$$\hat{H}\beta = E\beta.$$

Altså vil en kompleks egenfunksjon alltid kunne konstrueres fra to reelle egenfunksjoner, og vi vil derfor alltid kunne finne reelle løsninger.

I dette kapittelet skal vi se på numeriske metoder for å løse den andreordens differensiallikninga TUSL. Vi ønsker å finne egenfunksjonen ψ til Hamiltonoperatoren med egenverdi E. Gitt et potensial $V(x)$ og de nødvendige initialbetingelsene, skal vi skrive programmer som gjør dette. Som alltid finnes

hengige, og ikke endrer seg med tiden. Vi begrenser oss også til én romdimensjon, men det er mer av praktiske årsaker.

[3]Men *ikke* hele Schrödingerlikninga!

det mange måter å løse problemet på. En forholdsvis intuitiv vei frem er å bruke den **sentrale differansen**, slik at vi får ei andreordens differenslikning. Effektiviteten i rutinen vi skal vise har stort forbedringspotensiale, men implementasjonen er enklere å forstå enn mange andre metoder.

Vi skal benytte en viktig egenskap ved fysisk akseptable bølgefunksjoner for bundne tilstander. Integrerer vi sannsynlighetstettheten over hele rommet, må vi kreve at den er konstant og summeres opp til én, fordi sannsynlighet må være bevart. Om det finnes en ball et eller annet sted på rommet ditt, så er det helt sikkert at du finner den dersom du leter absolutt overalt. Det er dette vi sier når vi krever normering av bølgefunksjonen, nemlig at

$$\int_{-\infty}^{\infty} |\Psi(x,t)|^2 \, dx = 1. \tag{5.4}$$

Dette kravet forteller oss at en bølgefunksjon som skal beskrive et fysisk system, må dø ut langs aksene når vi går mot uendelig — ellers hadde vi ikke klart å normere den. Dette er et viktig krav som vi skal benytte oss av når vi løser TUSL.

Når vi løser TUSL numerisk, vil dette også skje i dimensjonsløse variable. Derfor skal vi først se på hvordan vi kan skrive om TUSL på dimensjonsløs form, før vi ser på hvordan vi numerisk kan finne et uttrykk for den dobbeltderiverte.

5.1 Dimensjonsløse variable

Når vi løser Schrødingerlikninga numerisk, er det en fordel dersom vi kan gjøre dette med dimensjonsløse variable og med en dimensjonsløs bølgefunksjon. Bruk av dimensjonsløse størrelser gjør ofte at vi kan unngå å måtte hanskes med svært små eller store tall i beregningene våre, samtidig som vi naturligvis slipper å tenke på dimensjonene; det er også en måte å generalisere problemet på til et rent matematisk problem som gjør det lettere å finne frem til analytiske eller numeriske teknikker. Vi vil også få en følelse for hvilke naturlige enheter for lengde og energi som gjelder for systemet vi beskriver.

For et elektron eller et proton er det både tungvint og lite intuitivt å snakke om deres masse i kg. Bedre er det om vi kan referere til elektronmassen som $m = 1$, og deretter oppgi alle andre masser i forhold til denne. Da vil elektronmassen være en 'naturlig enhet' for masse. På samme måte kan en

naturlig lengdeskala for et endelig brønn-potensiale være brønnens bredde a, mens en naturlig energienhet kan være brønnens dybde V_0.

Vi lar den dimensjonsløse avstanden betegnes ved z, det dimensjonsløse potensialet ved W og den dimensjonsløse energien ved ϵ. Det er viktig å merke seg at ϵ *ikke* er energi, men vi kan enkelt finne (den fysiske) energien E, da denne vil avhenge av ϵ. Vi gjør et eksempel i detalj for å tydeliggjøre hva som skjer, og ser på uendelig brønn-potensialet.

5.1.1 Eksempel: Dimensjonsløse variable i uendelig brønn

Vi illustrerer en uendelig brønn med bredde a i figur 5.1. Vi setter for enkelhets skyld potensialet innenfor veggene til å være $V = 0$. Vi ønsker å operere med den dimensjonsløse lengden $z = x/a$, som betyr at $x = za$. Ved variabelbyttet får vi dermed

$$\frac{dz}{dx} = \frac{1}{a}.$$

Vi kan skrive om TUSL i likning (5.2) som

$$\frac{d^2\psi(x)}{dx^2} = \frac{2m}{\hbar^2}\Big(V(x) - E\Big)\psi(x). \tag{5.5}$$

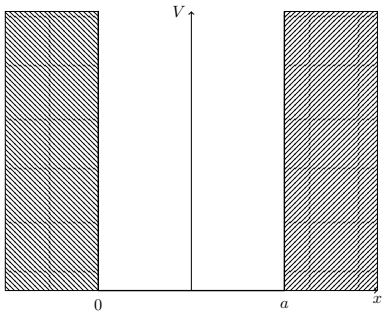
Inne i en uendelig brønn der $V(x) = 0$ får vi da:

$$\begin{aligned}
\frac{d^2\psi(x)}{dx^2} &= -\frac{2mE}{\hbar^2}\psi(x) \\
\frac{d^2\tilde{\psi}(z)}{dz^2}\left(\frac{dz}{dx}\right)^2 &= -\frac{2mE}{\hbar^2}\tilde{\psi}(z) \\
\frac{d^2\tilde{\psi}(z)}{dz^2}\frac{1}{a^2} &= -\frac{2mE}{\hbar^2}\tilde{\psi}(z) \\
\frac{d^2\tilde{\psi}(z)}{dz^2} &= -\frac{2mEa^2}{\hbar^2}\tilde{\psi}(z) \\
\frac{d^2\tilde{\psi}(z)}{dz^2} &= -\epsilon\tilde{\psi}(z),
\end{aligned}$$

hvor vi har satt

$$\epsilon = \frac{2mEa^2}{\hbar^2}, \tag{5.6}$$

som nå er en dimensjonsløs variabel (bare sjekk!).



Figur 5.1: Uendelig brønn med bredde a. Forbudt område med $V = \infty$ er skravert, og bølgefunksjonen må dø ut ved brønnens vegger for å oppfylle kravet (grensebetingelsen) om kontinuitet.

Med disse endringene er problemet nå på dimensjonsløs form. Når vi programmerer er det den korrekte dimensjonsløse ϵ vi leter etter — ikke den fysiske energien E. Men dersom vi vet ϵ, kan vi enkelt finne E av relasjonen i likning 5.6.

Generelt vil altså den dimensjonsløse Schrödingerlikninga være på formen

$$\frac{d^2\psi(z)}{dz^2} = \Big(W(z) - \epsilon\Big)\psi(z),$$

hvor vi også har foretatt en skalering av potensialet som gjør at $W(z)$ er en dimensjonsløs størrelse:

$$W(z) = \frac{2ma^2}{\hbar^2}V(z).$$

Vi vil for enkelhetsskyld herette betegne den dimensjonsløse bølgefunksjonen $\tilde{\psi}$ som ψ.[4]

5.2 Diskret derivasjon

Fordi en datamaskin ikke kan representere alle verdiene til en kontinuerlig funksjon blir vi tvunget til å diskretisere for å komme noen vei med å finne

[4]Vi gjør dog oppmerksom på at $\tilde{\psi}$ har en annen normering enn ψ fordi

$$\int_{-\infty}^{\infty} |\psi(x)|^2\, dx = \int_{-\infty}^{\infty} |\psi(az)|^2\, a dz, \tag{5.7}$$

slik at $\tilde{\psi}(z) = \sqrt{a}\psi(az)$.

$\psi(z)$. Vi deler den dimensjonsløse aksen z opp i n biter av lengde Δz. Dette betyr at punkt nummer j har posisjonen $z_j = j\Delta z$. Dette gir oss følgende notasjon når vi går fra et kontinuerlig til diskret tilfelle:

$$z \to z_j \equiv j\Delta z \tag{5.8}$$
$$\psi(z) \to \psi_j \equiv \psi(z_j) \tag{5.9}$$
$$W(z) \to W_j \equiv W(z_j). \tag{5.10}$$

Husk også at $z + \Delta z \equiv z_{j+1}$.

Vi har sett at TUSL inneholder en annenderivert, og vi må vite hvordan vi kan finne denne numerisk. Vi tar utgangspunkt i at den deriverte til en funksjon $y(x)$ i punktet x, for små Δx, kan tilnærmes som:

$$y'(x) \simeq \frac{y(x + \frac{1}{2}\Delta x) - y(x - \frac{1}{2}\Delta x)}{\Delta x}. \tag{5.11}$$

Dette kalles den **sentrale differansen**.

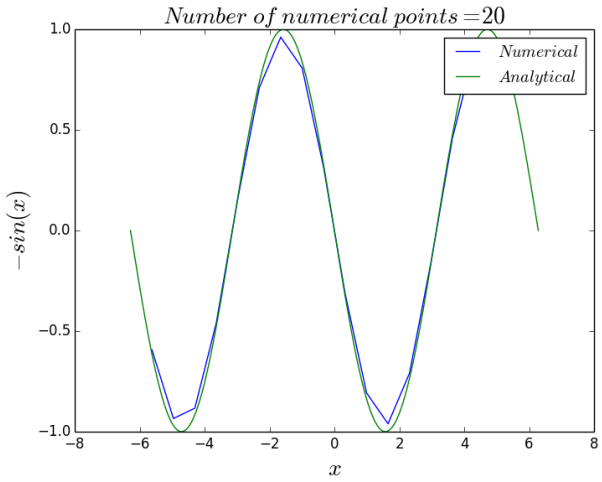
Den andrederiverte kan vi finne ved å derivere dette uttykket

$$\begin{aligned} y''(x) &\simeq \frac{1}{\Delta x}\left(y'(x + \frac{1}{2}\Delta x) - y'(x - \frac{1}{2}\Delta x)\right) \\ &= \frac{1}{(\Delta x)^2}\left(y(x + \Delta x) - y(x) - y(x) + y(x - \Delta x)\right) \\ &= \frac{1}{(\Delta x)^2}\left(y(x + \Delta x) - 2y(x) + y(x - \Delta x)\right). \end{aligned} \tag{5.12}$$

Vi bør med en gang legge merke til at vi vil få problemer i randen med denne metoden, og vi vil aldri klare å beregne den dobbeltderiverte i det første og siste punktet på intervallet vi ser på.

Et eksempel på hvordan en slik beregning av den andrederiverte kan gjennomføres i Python, er:

```
from numpy import *

n = 20
x = linspace(-2*pi,2*pi,n)
f = sin(x)
deriv = -sin(x)
dx = x[1]-x[0]
d2f = zeros(n)

# We can not calculate d2f[0] and d2f[n-1], the first and last
    point
```



Figur 5.2: Den sentrale differansen er brukt for å beregne den andrederiverte til $f(x) = \sin(x)$. Det analytiske uttrykket $f''(x) = -\sin(x)$ er tatt med for sammenlikning, hvor vi har brukt et større entall punkter enn for den numeriske løsningen.

```
for i in range(1,n-2):
        d2f[i] = (f[i+1] - 2*f[i] + f[i-1])/dx**2
```

Her ser vi at vi ikke kan finne den andrederiverte verken for det første punktet $f(-2\pi)$ (som svarer til $i = 0$), eller det siste punktet $f(2\pi)$ (som svarer til $i = n - 1$).

En sammenlikning av den numeriske og analytiske løsningen er gitt i figur 5.2. Her kan man leke seg med hvor mange punkter man bruker. Jo flere punkt, jo mer nøyaktig blir løsningen. Det er ønskelig å finne det optimale antallet punkter som gjør at tilnærmingen er god nok for det man vil oppnå — men ikke bedre. På den måten blir resultatet tilfredsstillende, uten at vi har brukt unødvendig datakraft i beregningene. (Tenk miljø!)

Vi kan bruke denne tilnærmingen av den dobbeltderiverte til å løse TUSL, som vi har sett er på formen:

$$\psi''(z) = (W(z) - \epsilon)\psi(z), \tag{5.13}$$

hvor ϵ er den dimensjonsløse energien og W det dimensjonsløse potensialet. Dersom vi bruker likning (5.12) for å finne $\psi''(z)$ og setter dette inn i likning (5.13), får vi, etter å ha ryddet opp litt,

$$\psi(z + \Delta z) = [2 + (\Delta z)^2(W(z) - \epsilon)]\psi(z) - \psi(z - \Delta z). \tag{5.14}$$

Vi kan nå gå til diskret form

$$\psi_{j+1} = \left[2 - (\Delta z)^2(\epsilon - W_j)\right]\psi_j - \psi_{j-1}. \tag{5.15}$$

Lar vi tilslutt $j \to j+1$, ser vi at

$$\psi_{j+2} = [2 - (\Delta z)^2(\epsilon - W_{j+1})]\psi_{j+1} - \psi_j, \tag{5.16}$$

som gir oss verdien av bølgefunksjonen $\psi(x)$ for punktet $j+2$ dersom vi kjenner den for punktene $j+1$ og j. Vi er nå i prinsippet i stand til å finne bølgefunksjonen for alle punktene dersom vi kjenner den for de to første punktene, som er våre initialbetingelser.[5]

Vi er altså i stand til å beregne bølgefunksjonen fra formen på potensialet, gitt at vi har to initialbetingelser. I tillegg må vi spesifisere hva ϵ skal være. For hvert valg av ϵ vi gjør, vil vi få en forskjellig bølgefunksjon ψ. Vi ønsker å finne de verdier av ϵ som gjør at ψ er en fysisk tillatt bølgefunksjon. Med det mener vi at den må være normerbar; som vi har diskutert over må den dø ut langs aksene slik at normeringsintegralet (5.4) blir endelig. En annen, mer matematisk, måte å si dette på et at den har de riktige **grensebetingelsene**. Dette gjør vi under ved en prøv-og-feil-metode.

5.3 Prøv-og-feil-metoden

Som sagt er TUSL ei egenverdilikning, og vi ønsker å bestemme både egenverdiene ϵ og egenfunksjonene ψ. Vi har tidligere sett hvordan vi rekursivt er i stand til å regne ut ψ, gitt formen på potensialet, to initialbetingelser, og ϵ. Vi trenger imidlertid en metode for å sjekke om dette faktisk er den fysiske bølgefunksjonen vi leter etter. Til dette må vi bruke vår kunnskap om hvordan en bølgefunksjon skal oppføre seg; den må være normerbar. Når vi har beregnet hele bølgefunksjonen, kan vi sjekke om den er normerbar, og godkjenne den som en gyldig egenfunksjon dersom det er tilfelle. Dør den derimot ikke ut, må vi konkludere med at den ikke er fysisk. Det betyr at egenverdien ϵ vi gjettet på, ikke er den rette egenverdien, og vi må gjøre et nytt gjett. Deretter følger nøyaktig samme prosedyre, helt til vi finner en ϵ som gjør bølgefunksjonen normerbar.

Grovt skissert vil vi altså algoritmisk gå fram på følgende måte for å finne både bølgefunksjon og energi:

[5]Det burde ikke forundre oss at en annengrads differensiallikning behøver to initialbetingelser.

1. Gjett en egenverdi ϵ.
2. Gi initialbetingelser ψ_0 og ψ_1 (mer om dette senere).
3. Beregn hele ψ ved hjelp av initialbetingelsene og likning (5.16).
4. Sjekk om den beregnede bølgefunksjonen er fysisk (dør den ut?)
5. Hvis den er ufysisk, gjenta prosedyren fra 1.
6. Hvis ψ er fysisk, har vi funnet en egenverdi ϵ og vi har en løsning.
7. Til slutt kan vi regne ut energien E fra ϵ.

Denne prosessen kan gjentaes for å finne flere egenverdier ϵ.

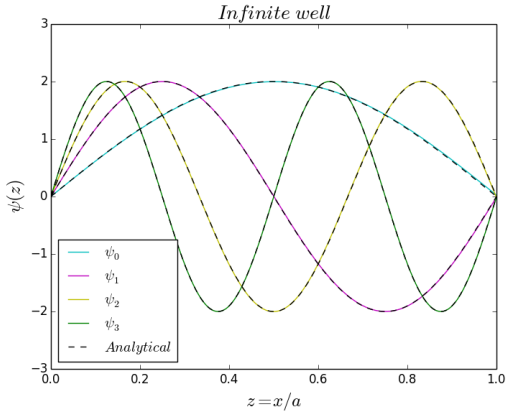
Disse punktene kan oppsummeres med følgende kode:

```
epsilon = []     # numerical eigenvalue list to be filled
epsilon_trial = 0 # trial eigenvalue
tol = 0.002 # tolerance level

# Running through trial values to find correct eigenvalue
while epsilon_trial < 11:

        # Determining wave function
        for j in range(1,N-1):
                Psi[j+1] = (2 - dz**2*(epsilon_trial-V[j]))*Psi[
                    j] - Psi[j-1]

        # Normalizing wave function
        Psi /= sqrt(scipy.integrate.simps(abs(Psi)**2,dx=1e-3))

        # Check if last element is within tolerance (if yes:
            trial eigenvalue is eigenvalue),
        if abs(Psi[-1])<tol:
                epsilon.append(epsilon_trial)

        # Update trial eigenvalue
        epsilon_trial += 0.05
```

Her har vi sjekket at bølgefunksjonen er tilstrekkelig nær null ved enden, som betyr at den kan normaliseres. Jo flere tilstander vi ønsker å finne, jo høyere verdier må vi la testverdiene for ϵ ta

For uendelig brønn kjenner vi den analytiske løsningen, og vi kan plotte denne for sammenlikning med resultatet fra metoden over, i figur 5.3. Den dimensjonsløse energien i den analytiske løsningen er $\epsilon = n^2\pi^2$, hvor vi kun

er interessert i heltallige $n \neq 0$.[6] Legg merke til at energinivåene øker som n^2, som betyr at det ganske raskt bli store gap imellom energinivåene. Dette gjør det tidkrevende å finne mange eksiterte tilstander numerisk.



Figur 5.3: Grunntilstand og tre første eksiterte tilstander i uendelig brønn er funnet numerisk. Analytisk løsning er plottet som stiplet sort linje for sammenlikning.

Vi kommer også i den situasjonen at flere verdier av ϵ passerer toleransekravet. Det hjelper ikke å senke toleransen, fordi dette kan gjøre at vi mister løsninger andre steder. Vi må derfor ha en rutine som plukker ut den ϵ som gir minimal ψ-verdi for store z av alle de som passerer toleransen omkring den samme verdien. Dette kan vi gjøre her på følgende måte:

```
# Running through trials to find correct eigenvalue
while epsilon_trial < 160:

        # Calculating wave function
        for j in range(1,N-1):
                Psi[j+1] = (2 - dz**2*(epsilon_trial-W[j+1]))*
                    Psi[j] - Psi[j-1]

        # Normalizing wave function
        Psi /= sqrt(scipy.integrate.simps(abs(Psi)**2,dx=1e-3))

        # Store value of last element in Psi
        Psi_end = abs(Psi[-1])
```

[6] Det holder at vi begrenser oss til positive heltall n fordi løsningene består av sinusfunksjoner hvor $\sin(-n\theta) = -\sin(n\theta)$, og disse løsningene beskriver den samme fysiske tilstanden da fortegnet ikke har noen betydning. Samtidig vil $n = 0$ gjøre bølgefunksjonen null overalt.

```
        # Check if last element is within tolerance
        if abs(Psi[-1])<tol:
                epsilon.append(epsilon_trial)
                lastpsi.append(Psi_end)
                Psi_list.append(list(Psi)) # Add as list to make
                    things behave as they should

                # Only keep the epsilons and wave functions
                    giving minimal value of Psi[-1]
                if len(lastpsi) > 2 and epsilon[-1] - epsilon
                    [-2] < 1:
                        if lastpsi[-1] < lastpsi[-2]:
                                lastpsi.remove(lastpsi[-2])
                                epsilon.remove(epsilon[-2])
                                Psi_list.remove(Psi_list[-2])
                        if lastpsi[-1] > lastpsi[-2]:
                                lastpsi.remove(lastpsi[-1])
                                epsilon.remove(epsilon[-1])
                                Psi_list.remove(Psi_list[-1])
        # Update trial eigenvalue
        epsilon_trial += 0.05
```

5.4 Valg av ϵ og initialbetingelser ψ_0 og ψ_1

I eksemplet i forrige avsnitt valgte vi å begynne med $\epsilon = 0$ i vårt søk etter egenverdier. Hvorfor det? Jo, vi vet at det laveste punktet i potensialet er $V_{\text{min}} = 0$ (nede i brønnen). Vi vet også at alle tillatte tilstander må ha $E > V_{\text{min}}$. Dette har vi vist på forelesningene. På grunn av sammenhengen mellom ϵ og energien E i likning (5.6) er det derfor ikke noe poeng å begynne lavere enn $\epsilon = 0$. Kan vi samtidig tenke oss en høyeste mulig verdi for ϵ, slik at vi kan lete i et endelig intervall? Dessverre, for den uendelig brønnen så kan energien være vilkårlig stor. Men, i andre situasjoner, for eksempel for en endelig brønn, vet vi at det finnes en øvre grense på energien så lenge vi leter etter bundne tilstander. Energien kan ikke være større enn toppen på potensialbrønnen, da vi vil få en fri tilstand isteden. Dette setter da en grense på hvor stor den enhetsløse energien ϵ kan være.

For å bestemme bølgefunksjonen må vi løse ei andreordens differenslikning, og følgelig trengs to startverdier for å komme i gang. Men hvordan velger man egentlig disse? Det finnes ingen helt sikker fasit på dette spørsmålet, fordi det er avhengig av hvordan potensialet ser ut.

Dersom potensialet for eksempel har én uendelig høy potensialbarriere, er det lurt å starte itereringa ved veggen til barrieren. Da vet vi at $\psi_0 = 0$ fordi bølgefunksjonen må være kontinuerlig i overgangen mellom det tillatte og forbudte området, og siden bølgefunksjonen må være null på det forbudte området, må den også være null i første punkt innenfor det tillatte. Videre kan ψ_1 velges vilkårlig fordi normaliseringen til slutt vil ordne forholdet mellom ψ_0 og ψ_1, og fordi en endring på fortegnet til bølgefunksjonen ikke har noen fysiske konsekvenser.[7] Dersom vi definerer kvantemekanikken på $x \in (-\infty, \infty)$, kan vi alltid si at $\psi_0 = 0$ fordi ψ må dø ut.

For symmetriske potensialer vet vi at de tillatte løsningene enten er symmetriske eller antisymmetriske (odde eller like), og dette kan vi utnytte. Odde bølgefunksjoner har ingen amplitude i symmetrisenteret (tenk på hvordan $\sin(x)$ ser ut om origo), og hvis vi starter beregninga av ψ her, betyr det at $\psi_0 = 0$. Videre vet vi at den deriverte i dette punktet ikke er null. Det uttrykker vi ved å velge $\psi_1 \neq 0$, der det nok en gang vil være likegyldig hvilken verdi vi velger på grunn av normaliseringa.

For like funksjoner er det motsatt: de *har* amplitude i symmetrisenteret (tenk på hvordan $\cos(x)$ ser ut om origo), mens deres deriverte i punktet er null. Dette *tilnærmer* vi ved å sette $\psi_0 \neq 0$ og $\psi_1 = \psi_0$. Det er dog ikke helt riktig, fordi det ikke er det samme som at den deriverte er null i dette punktet, men dersom steglengden vi opererer med er liten nok, vil det som regel være en god nok tilnærming.

Symmetriegenskapene gjør at det er tilstrekkelig å finne bølgefunksjonen fra symmetriaksen og ut på den ene siden. Da kan vi speile løsningen, og dermed dekke hele området. Husk at de odde løsningene vil få et fortegnsbytte i speilingen. Dette er egenskaper vi for eksempel kan utnytte i harmonisk oscillator-potensialer.

5.5 Eksempel: harmonisk oscillator

Vi forlater den uendelige brønnen, og ser på harmonisk oscillator-potensialet. Dette har formen $V(x) = \frac{1}{2}m\omega^2x^2$, og TUSL blir:

$$\frac{d^2\psi(x)}{dx^2} = -\frac{2m}{\hbar^2}\left(E - \frac{1}{2}m\omega^2x^2\right)\psi(x). \tag{5.17}$$

[7] Du kan imidlertid ikke også velge $\psi_1 = 0$. Ser du hvorfor? *Hint:* Det har noe med (5.16) å gjøre.

Vi begynner også her med å gjøre likninga dimensjonsløs, og innfører i første omgang $z = \frac{x}{a}$, der a er en (foreløpig ukjent) lengdeparameter. Med dette variabelbyttet, blir TUSL

$$\frac{d^2\psi(z)}{dz^2} = -\left(\frac{2mEa^2}{\hbar^2} - \frac{m^2\omega^2a^4}{\hbar^2}z^2\right)\psi(z). \tag{5.18}$$

Selv om vi jobber dimensjonsløst når vi skal løse likninga, ønsker vi å gjenskape dimensjonene senere, og derfor må vi vite hvordan vi kan uttrykke a ved hjelp av de kjente parametrene i systemet. Den eneste kombinasjonen av konstantene som inngår i likninga, $\hbar, \omega$ og m, som gir enhet lengde, er $a = \sqrt{\hbar/m\omega}$, og dersom vi setter inn dette i (5.18) blir energidelen av likninga $\frac{2E}{\hbar\omega}$, som også er en dimensjonsløs størrelse. Derfor setter vi $\epsilon = \frac{E}{\hbar\omega}$, og likninga blir til slutt på formen:

$$\frac{d^2\psi(z)}{dz^2} = \left(z^2 - 2\epsilon\right)\psi(z). \tag{5.19}$$

Da har vi altså innført $z = \frac{x}{a}$ hvor $a^2 = \frac{\hbar}{m\omega}$, og $\epsilon = \frac{E}{\hbar\omega}$, og det dimensjonsløse potensialet er $W(z) = z^2$. Dette er benyttet i programmet `ho_example.py` som løser harmonisk oscillator-potensialet. Også her kjenner vi de analytiske løsningene, så vi kan sjekke om numerikken stemmer, noe som er vist i figur 5.4. Programmet skriver også ut egenenergiene ϵ, så vi kan sjekke at disse stemmer med det analytiske uttrykket. Vi vet at den analytiske energien er gitt ved $E = \hbar\omega(n + \frac{1}{2})$ hvor $n = 0, 1, 2, \ldots$, slik at vi burde få $\epsilon = n + \frac{1}{2}$.

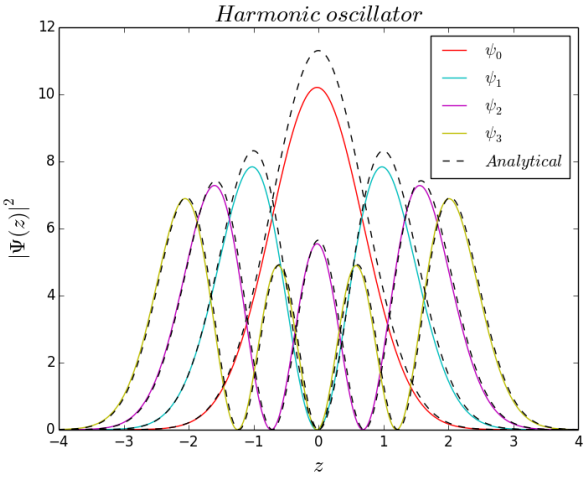
5.6 Eksempel: endelig brønn

Som et siste eksempel skal vi se på endelig brønn-potensialet som vist i figur 5.5. Som tidligere, har vi løst problemet ved å tilnærme den dobbeltderiverte med en sentral differanse, og spart på de ϵ som gir oss en normaliserbar bølgefunksjon. Resultatet av en slik tilnærming er illustrert i figur 5.6.

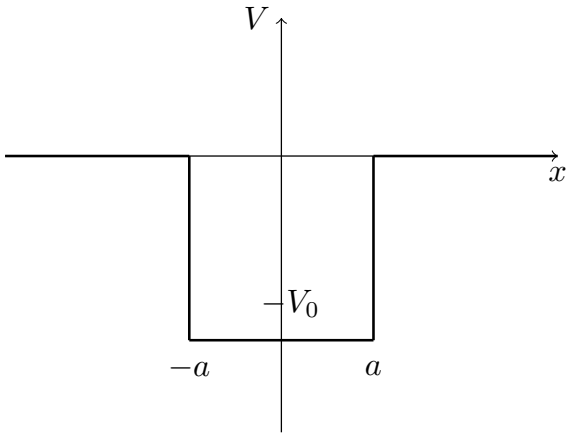
En annen, mer analytisk angrepsmåte, er å løse TUSL både innenfor og utenfor brønnen, før vi krever at den kontinuerlig i overgangene. Dette gjøres i Griffiths, og for de symmetriske løsningene får vi den transcendente[8] likninga

$$\tan(z) = \sqrt{\left(\frac{z_0}{z}\right)^2 - 1}, \tag{5.20}$$

[8]Det vil si: ikke (alltid) analytisk løsbare.



Figur 5.4: Sammenlikning av numerisk og analytisk løsning for harmonisk oscillator. Den numeriske løsningen har relativt dårlig oppløsning (og derfor høy toleranse) for å vise forskjellen mellom numerisk og analytisk løsning.



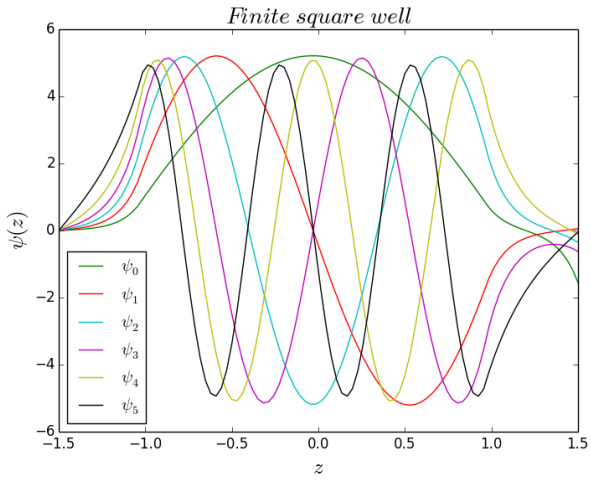
Figur 5.5: Endelig brønn med dybde $-V_0$ og bredde $2a$, sentrert om origo.

hvor både z_0 og z er dimensjonsløse variable definert ved hjelp av parametrene for brønnen

$$z_0 = \frac{\sqrt{2ma^2V_0}}{\hbar},$$

og

$$z = \frac{\sqrt{2ma^2(E+V_0)}}{\hbar}.$$



Figur 5.6: Numerisk løsning av endelig brønn-potensiale ved bruk av sentral differanse. Toleransen er relativt høy da oppløsningen i z er liten.

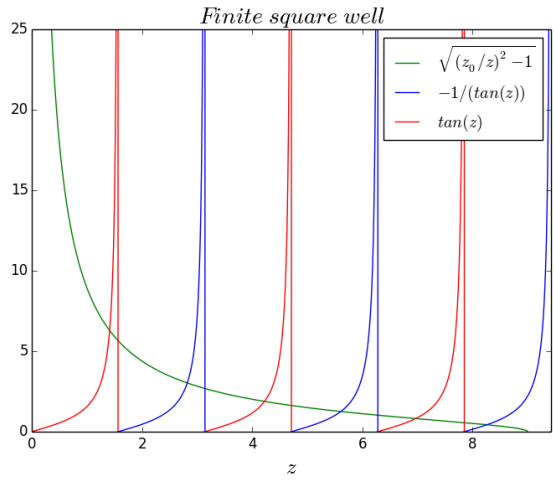
Vær oppmerksom på at z slik den er definert i Griffiths *ikke* er den samme som vi tidligere har benyttet oss av! Legg også merke til at vi alltid vil ha $z_0 > z$ med disse definisjonene (kan du se hvorfor?). Det betyr at vi aldri får problemer med roten i likning (5.20), så vi må sørge for at dette også er tilfelle når vi velger verdiene for z_0 og z hvis vi skal finne løsningene numerisk.

Vi ønsker å finne for hvilke z de to funksjonene skjærer hverandre. Ut ifra det, kan vi regne oss tilbake til hvilken energi E dette svarer til.[9] Et eksempel på hvordan man kan finne skjæringspunkter er vist i figur 5.7, hvor løsningene finnes der den grønne linja krysser de røde (symmetriske løsninger) og blå (antisymmetriske løsninger). Vær klar over at det kun vises fem skjæringspunkter, selv om det ser ut som vi har det dobbelte. De vertikale stripene er der på grunn av den asymptotiske oppførselen til $\tan(z)$.

Skjæringspunktene er bestemt med koden under:

```
# Finite well, transcendental equation

# Constants and parameters
z = np.linspace(zstart,zend,N)  # dimensionless position array
z0 = 9 # dimensionless variable related to well width
a = 0.2 # half the width of well [nm]
```

[9] Legg merke til at z_0 relateres til brønnens dybde gjennom V_0, og vi forventer å finne flere bundne tilstander dersom vi øker z_0 (svarer til å gjøre brønnen dypere).



Figur 5.7: Numerisk løsning av endelig brønn-potensiale ved å finne skjæringspunkter. Symmetrisk løsning i rødt, antisymmetrisk i blått.

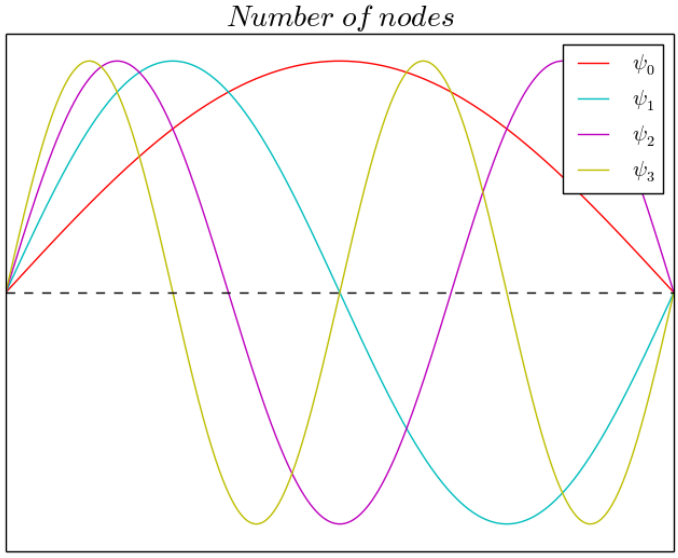
```
V0 = z0**2*hbarc**2/(2*mc2*a**2)

rhs = sqrt((z0/z)**2 - 1) # right hand side of trancendental
    equation
lhs_even = tan(z) # even left hand side of trancendental
    equation
lhs_odd = -1/(tan(z)) # odd left hand side
tol = 0.1 # tolerance
diff_even = abs(lhs_even - rhs) # (absolute) difference of rhs
    and lhs, even
diff_odd = abs(lhs_odd - rhs) # (absolute) difference of rhs and
     lhs, odd

intersection = [] # points (z) of intersection
E = [] # list to be filled with physical energies (numerical)

# Running through differences to check where we find the
    intersections
# Odd solutions
for i in range(0,len(diff_odd)):
        di = diff_odd[i]
        zi = z[i]
        if di < tol:
                # Test to pick only the smallest difference (in
                    case several pass the tolerance)
                if diff_odd[i+1] > di and diff_odd[i-1] > di:
                        intersection.append(zi)
```

```
# Sort intersections
intersection.sort()
```

5.7 Å telle noder

Fordi vi ikke kan la ϵ ta alle verdier, er det klart at nøyaktigheten vi finner egenenergiene med, henger nøye sammen med hvor fint søket vårt i ϵ er. Vi kan tillate små toleranser dersom oppløsningen i ϵ er stor, og tilsvarende må vi la toleransen være stor dersom oppløsningen er dårlig.

Dersom vi lar ϵ starte på 0 og øke med 0.05 hver gang vi gjør nye beregninger, får vi bare sjekket bølgefunksjonen for $0, 0.05, 0.1, 0.15, 0.2, ..$ osv. Om den riktige verdien er $\epsilon = 0.12$, så vil vi aldri få sjekket hvordan bølgefunksjonen blir for akkurat dette tilfellet. Det nærmeste vi kommer er $\epsilon = 1.0$. Det er dog ikke sikkert at denne verdien gjør at bølgefunksjonen dør ut tilstrekklig, slik at den passerer toleransekravet som er satt. Da vil vi komme i en situasjon der vi mister en løsning. Det finnes måter å unngå dette på, som å la oppløsningen i ϵ øke, eller velge en større toleranse. Før vi eventuelt gjør dette (det vil gå på bekostning av effektivitet), er det greit å vite om vi har funnet alle energiene eller ikke. Det er en enkel måte å sjekke dette på, og det er å telle noder. For tilstander ordnet etter stigende energier E_n der $n = 0, 1, 2, ...$, er det slik at tilstandene har n noder. Grunntilstanden har dermed ingen noder, første eksiterte tilstand har én, osv. Det er med andre ord enkelt å telle om vi har fått med alle egenenergier eller ikke, dersom vi plotter ψ og teller hvor mange ganger de ulike bølgefunksjonene passerer null.



Figur 5.8: Vi kan telle antall noder for å avgjøre hvilken eksitert tilstand vi har. ψ_n har n noder når $n = 0, 1, 2, ...$

Kapittel 6

Den tidsavhengige Schrödingerlikninga

I forrige kapittel så vi på separable løsninger av Schrödingerlikninga på formen $\Psi(x,t) = \psi(x)\phi(t)$, som ga oss stajonære tilstander; sannsynlighetstettheten endret seg ikke med tiden. Vi går nå bort ifra å anta at $\Psi(x,t)$ tar en slik separabel form, og ser på utviklingen av sannsynlighetstettheten som en funksjon av tiden.

Når vi løser Schrödingerlikninga, løser vi ei differensiallikning. Vår oppgave er å finne den funksjonen $\Psi(x,t)$ som oppfyller likninga. Likninga forteller oss hvordan den deriverte til funksjonen ser ut, og et eksempel på ei førsteordens differensiallikning er

$$\frac{\partial y(x,t)}{\partial t} = f(x,t), \tag{6.1}$$

hvor $f(x,t)$ er en kjent funksjon, mens $y(x,t)$ er den vi ønsker å finne.

Differensiallikninger er et veldig stort felt innen fysikk, da de fleste problemer kan løses på denne måten. Det finnes derfor mange forskjellige numeriske algoritmer som kan brukes for diskret å regne seg fra t til $t + \Delta t$, så lenge likninga er på formen vist i (6.1).

Schrödingers likning kan skrives på formen

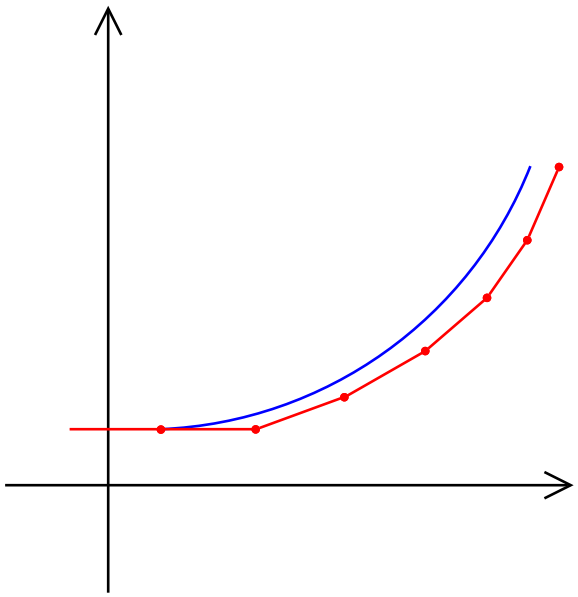
$$\frac{\partial \Psi}{\partial t} = -\frac{i}{\hbar}\hat{H}\Psi.$$

Denne er på samme form som likning (6.1), der $-\frac{i}{\hbar}\hat{H}\Psi$ tilsvarer funksjonen $f(x,t)$.

Vi skal ta for oss noen numeriske metoder for å løse denne likninga. Hvor "gode" de ulike metodene er, måles i hvor liten feil de produserer og hvor stabile de er. Feil er umulig å unngå, fordi metodene alltid benytter tilnærminger til funksjonen vi ønsker å finne.

6.1 Eulers metode

Den enkleste metoden vi skal se på er Eulers metode, også kjent som Forward Euler. Denne metoden baserer seg på en lineær tilnærming fra ett punkt til et annet, ved å gå en gitt lengde langs tangeneten til det punktet vi står i. Prinsippet er illustrert i **Figur 6.1**, men for Schrødingerlikninga blir denne figuren litt for enkel. For hvert tidspunkt t skal vi beskrive hele området x, og vi beregner derfor en hel kurve for hvert nye tidssteg vi ser på.



Figur 6.1: Illustrasjon av Eulers metode. For hvert nye steg går metoden en viss lengde langs tangenten. Den røde linjen illustrerer funksjonen approksimert med Eulers metode mens den blå linjen er den eksakte funksjonen.

Metoden lar seg gjennomføre fordi vi alltid har tilgang til den tidsderiverte i det punktet vi er i, siden den tidsderiverte er gitt helt eksplisitt når uttrykket er på formen i (6.1). Ved tiden t, vil løsningen etter et kort tidssteg Δt med

Eulers metode være gitt som

$$y_{t+\Delta t} = y_t + \Delta t \cdot y'_t$$
$$y_{t+\Delta t} = y_t + \Delta t \cdot f(x).$$

For Schrödingerlikninga blir dette det samme som

$$\Psi_{t+\Delta t} = \Psi_t - \Delta t \frac{i}{\hbar} \hat{H} \Psi_t.$$

Dette ligner på det vi gjorde i kapittel 5, her med en førstederivert i tid istedet for en andrederivert med hensyn på posisjon, men det er, som nevnt ovenfor, viktig å huske at Schrödingerlikninga på hvert tidspunkt fortsatt også avhenger av x. Diskret må vi altså utføre denne beregningen på alle x'ene i settet for å finne Ψ ved neste tidssteg. Bedre notasjon er altså

$$\Psi^{x_i}_{t+\Delta t} = \Psi^{x_i}_t - \Delta t \frac{i}{\hbar} \hat{H} \Psi^{x_i}_t. \tag{6.2}$$

Dette kan implementeres i Python på følgende måte. Hvis vi antar at vi har en array som inneholder Ψ_t og ferdigdefinerte parametre, kan beregningsløkken se slik ut:

```
while t < T:
    Psi_new = Psi_old - * dt * 1j / hbar * H * Psi_old

    t = t + dt
    Psi_old = Psi_new
```

Legg merke til at i denne koden står $\hat{H}$ som en konstant. Det er veldig få tilfeller der man kan bruke $\hat{H}$ på denne måten. Her er denne skrivemåten kun brukt som eksempel. I seksjon 5.2 så vi på hvordan vi kan få implementert selve operatoren med derivasjon.

Eulers metode er en førsteordens metode. Dette betyr at hver approksimasjon gir opphav til en feil som er proporsjonal med tidssteget Δt. Altså er ikke dette en veldig nøyaktig metode. I tillegg blir den fort ustabil når funksjonen vi vil approksimere endrer seg raskt. Stabilitetsproblemet er noe vi skal se på senere i kapittel 7.

6.1.1 Vektorisering

Før vi går videre vil vi forklare hvordan vi numerisk kan regne ut den dobbeltderiverte raskere enn før. Tidligere har vi laget for-løkker og iterert oss

fra ett punkt til et annet. En mer effektiv metode er ved vektorisering, hvor vi regner med hele arrayer. Da unngår vi for-løkken, noe som kan spare oss for mye tid. Å gjøre beregninger på vektorisert form i python oppnår vi ved bruk av "slicing", som forklart i kapittel 1. Fra dette kapittelet ser vi at vi kan utnytte slicing for å få ut kun det spesifikke settet med tall vi vil ha. Dersom vi for eksempel ønsker å skrive ut alle tall unntatt det siste fra en array x, kan vi indeksere den med

```
x[ 0:-1 ]
```

siden 0 er første indeks og -1 er siste indeks. Husk at regelen "fra og med men ikke til og med gjelder" for arrays i python.

Når vi skal beregne den annenderiverte i et punkt, trenger vi punktet før, punktet etter og det aktuelle punktet vi står i. Dette fører til at den dobbeltderiverte i det første og siste punktet ikke kan beregnes. For å kunne vite den deriverte i disse punktene, måtte vi ha visst funksjonsverdien i punktet før første punkt, og punktet etter siste punkt.

Anta nå at vi har en array x, som er n lang. Da kan vi indeksere på følgende måte:

```
# Alle tallene unntatt de to siste
prev = x[ 0:n-2 ]

# Alle tallene unntatt ett paa hver side
middle = x[ 1:n-1 ]

# Alle tallene unntatt de to foerste
next = x[ 2:n ]
```

Ved hjelp av disse tre utvalgene fra x kan vi beregne den annenderiverte av $\sin x$ vektorisert på denne måten:

```
from numpy import *
import matplotlib.pyplot as plt

n = 20
x = linspace(-2*pi, 2*pi, n)
dx = x[1]-x[0]
f = sin(x)
y = -sin(x)

d2f = ( f[2:n] - 2 * f[1:n-1] + f[0:n-2] ) / dx**2
```

For hver komponent i arrayen d2f som regnes ut for den deriverte av funksjonen, vil f[2:n] gi funksjonsverdien for det neste punktet (siden den er forskjøvet fremover et steg), f[0:n−2] gi funksjonsverdien for det forrige punktet (siden den er forskjøvet et steg bakover) og f[1:n−1] gi funksjonsverdien for det punktet som vi regner ut den deriverte for.

6.2 Eksempel: Reisende gaussisk bølgepakke

Nå har vi nok verktøy til å løse den tidsavhengige Schrödingerlikninga for et enkelt tilfelle. Vi skal se på en reisende gaussisk bølgepakke som initialtilstand og utvikle den i tid ved å bruke Python.

Vi skal bruke initialtilstanden for et proton beskrevet ved en gaussisk bølgepakke og se hvordan den utvikler seg i tid når den ikke påvirkes av et potensiale. Tilstanden er beskrevet med

$$\Psi(x,0) = Ae^{-(x-x_0)^2/4a^2}e^{ilx}.$$

Der $x_0 = -100$ fm, $a = 5.0$ fm, $l = 100$ fm^{-1} og

$$A = \left(\frac{1}{2\pi a^2}\right)^{1/4}.$$

Vi har også de fysiske parametrene

$$\begin{aligned} E_{0,p} &= 938.27 \text{ MeV} \\ \hbar c &= 0.1973 \text{ MeV pm} \\ c &= 3 \cdot 10^2 \text{ pm / as.} \end{aligned}$$

Det kan derfor lønne seg å gjøre om konstantene til pm og for å få de til å samsvare. De blir da

$$\begin{aligned} x_0 &= -0.100 \text{ pm} \\ a &= 0.005 \text{ pm} \\ l &= 100 \cdot 10^3 \text{ pm}^{-1}. \end{aligned}$$

Et godt sted å begynne er å lage en funksjon som returnerer initialtilstanden. Dette gjøres på denne måten:

```
def Psi0( x ):
    '''
    Initialtilstand for gaussisk boelgepakke. Deler formelen
    opp i konstant og faktorer for aa faa det mer oversiktlig.
    '''
    x0 = 0.100 # [pm]
    a = 0.005 # [pm]
    l = 100.0e3 # [1 / pm]

    A = ( 1. / ( 2 * pi * a**2 ) )**0.25
    K1 = exp( - ( x - x0 )**2 / ( 4. * a**2 ) )
    K2 = exp( 1j * l * x )

    return A * K1 * K2
```

Videre må vi bestemme hvilket spekter av x vi skal se på: Antall punkter n og avstanden mellom hvert punkt Δx. Som tommelfingerregel kan man tenke på at jo høyere n er og jo mindre Δx er, jo mer nøyaktig blir utrengningen. Men som nevnt over med begrensningen i datamaskinen vil ikke dette alltid være tilfellet. Det er tilstrekkelig å velge en liten "nok" Δx og stor "nok" n. Her tar vi derfor ut av det blå verdiene,

$$dx = 0.001 \text{ pm} \qquad\qquad n = 801$$

og velger at spekteret med x'er skal gå fra $-\frac{1}{2}n\Delta x$ til $\frac{1}{2}n\Delta x$. Det er også mulig å prøve seg frem med disse verdiene til man finner noe som passer (at spekteret dekker hele tilstanden). Bestemmer disse parametrene slik:

```
n = 801 # Antall punkter i x-retning
dx = 0.001 # Avstand mellom x-punkter

a = -0.5 * n * dx
b = 0.5 * n * dx
x = linspace( a, b, n )
```

Nå trenger vi arrayer for å ta vare på den nåværende Ψ og for å holde på den annenderiverte. Må også passe på at den annenderiverte er erklært til å kunne inneholde komplekse tall som beskrevet i seksjon 3.6.1.

```
dPsidx2 = zeros( n ).astype( complex64 )
Psi = Psi0( x )
```

Det neste vi trenger er tidsparametre for å holde styr på hvor lenge simuleringen skal kjøre fremover i tid samt hvor langt tidssteget mellom hver

beregning skal gå. Jo mindre tidssteg jo mer nøyaktig (og mer stabil) løsning får vi (så lenge det ikke blir for lite). Når vi bruker Eulers metode er stabiliteten veldig avhengig av at vi velger et lite tidssteg, metoden blir fort usikker og "eksploderer" hvis vi ikke har et lite nok. Disse parametrene er satt slik:

```
T = 0.012 # Hvor lenge simuleringen skal kjoere [as]
dt = 1e-7 # Avstand mellom tidssteg [as]
t = 0 # Teller tid
```

Nå har vi alt vi trenger for å begynne beregningene. Beregningene må utføres i en løkke som for hvert tidssteg beregner den deriverte og lager en ny Ψ utifra fremgangsmåten i (6.2). Vi kan også forenkle uttrykket siden vi vet at

$$\hat{H} = -\frac{\hbar^2}{2m_p}\frac{\partial^2}{\partial x^2}.$$

Setter inn

$$\begin{aligned} -\frac{i}{\hbar}\hat{H} &= -\frac{i}{\hbar}\left(-\frac{\hbar^2}{2m_p}\frac{\partial^2}{\partial x^2}\right) \\ &= i\frac{\hbar}{2m_p}\frac{\partial^2}{\partial x^2} \end{aligned}$$

som kan videreføres for å forenkle beregningene til

$$\begin{aligned} -\frac{i}{\hbar}\hat{H} &= i\frac{(\hbar c)c}{2m_p c^2}\frac{\partial^2}{\partial x^2} \\ &= i\frac{(\hbar c)c}{2E_{0,p}}\frac{\partial^2}{\partial x^2}. \end{aligned}$$

For hvert tidssteg kan vi altså regne oss til det neste ved hjelp av uttrykket

$$\Psi(x_i, t + \Delta t) = \Psi(x_i, t) + i\Delta t\frac{(\hbar c)c}{2E_{0,p}}\frac{\partial^2}{\partial x^2}\Psi(x_i, t)$$

Dette beskriver Eulers metode for Schrödingerlikninga. Beregningen inneholder altså et konstantledd som vi kan ha utenfor løkken,

$$k_1 = i\frac{(\hbar c)c}{2E_{0,p}}.$$

Vi lager det direkte med.

```
k1 = ( 1j * hbarc * c) / (2. * E0p )
```

Det eneste som nå står igjen er å skrive løkken i seg selv. Vi trenger da en løkke som kjører så lenge t, tellern vår er mindre enn T. Det er da naturlig å bruke en **while**-løkke. Dette er ikke nødvendig, man kan skrive løkken som både **for** og **while**, men hvorfor gjøre det mer avansert enn man må?

```
while t < T:
    # Regner ut den deriverte
    dPsidx2[1:nx-1] = ( Psi[ 2:nx ] - 2 * Psi[ 1:nx-1 ] + Psi[
        0:nx-2 ] ) / dx**2

    # Regner ut ny Psi
    Psi = Psi + dt * c1 * dPsidx2

    # Legger til et tidssteg
    t += dt
```

Setter vi alt dette sammen til et program vil det se ut som i **Listing 6.1**.

Listing 6.1: Et program som simulerer hvordan en gaussisk bølgepakke utvikler seg i tiden.

```
from numpy import *
from matplotlib.pyplot import *

E0p = 938.27          # Hvileenergi for proton [MeV]
hbarc = 0.1973        # [MeV pm]
c = 3.0E2             # Lyshastighet [pm / as]

def Psi0( x ):
    '''
    Initialtilstand for gaussisk boelgepakke. Deler formelen
    opp i konstant og uttrykk for aa faa det mer oversiktlig.
    '''
    x0 = 0.100 # [pm]
    a = 0.0050 # [pm]
    l = 100000.0 # [1 / pm]

    A = ( 1. / ( 2 * pi * a**2 ) )**0.25
    K1 = exp( - ( x - x0 )**2 / ( 4. * a**2 ) )
    K2 = exp( 1j * l * x )

    return A * K1 * K2

nx = 801 # Antall punkter i x-retning
np = 1e2 # Plotte kun hver hver np'te utregning (bedre "
    framerate")
```

```
dx = 0.001 # Avstand mellom x-punkter

# Lar halvparten av x-verdiene vaere paa hver sin side av null
a = -0.5 * nx * dx
b = 0.5 * nx * dx
x = linspace( a, b, nx )

# Erklaerer tomme arrayer for den deriverte og Psi.
# Spesifiserer at den deriverte er kompleks
dPsidx2 = zeros(nx).astype(complex64)
Psi = Psi0( x )

# Tidsparametre
T = 0.012 # Hvor lenge simuleringen skal kjoere [as]
dt = 1e-7 # Avstand mellom tidssteg [as]

# Aktiverer interaktiv modus
ion()

# Tegner initialtilstand
figure()
line, = plot(x, abs(Psi)**2)
draw()

c1 = ( 1j * hbarc * c) / (2. * E0p ) # Konstant

t = 0 # Teller tid
c = 1 # Teller antall utrengninger
while t < T:
    # Regner ut den deriverte
    dPsidx2[1:nx-1] = ( Psi[ 2:nx ] - 2 * Psi[ 1:nx-1 ] + Psi[
        0:nx-2 ] ) / dx**2

    # Regner ut ny Psi
    Psi = Psi + dt * c1 * dPsidx2

    # Sjekker om den skal plotte denne utregningen
    if c == np:
        line.set_ydata(abs(Psi)**2)
        draw()
        c = 0 # Nullstiller teller

    t += dt # Legger til et tidssteg
    c += 1 # Legger til en utregning

# Av med interaktiv modus
ioff()
# Beholder vinduene
```

```
show()
```

6.3 Eksempel: Et potensialsteg

I seksjon 6.2 så vi på hvordan vi kunne løse den tidsuavhengige Schrödingerlikninga for en reisende gaussisk bølgepakke. Dette var for en fri partikkel - den ble ikke påvirket av noe potensial. Hvordan må vi da gå frem for også å la partikkel være påvirket av et potensial?

Det som forandrer seg er operatoren $\hat{H}$, nå får vi også med potensialleddet

$$\hat{H} = \left(-\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2} + V(x) \right) \Psi.$$

For hvert steg i beregningen må vi altså regne ut

$$\Psi(\vec{x}, t + \Delta t) = \Psi(\vec{x}, t) + \Delta t \left[c_1 \frac{\partial^2}{\partial x^2} + c_2 V(\vec{x}) \right] \Psi(\vec{x}, t).$$

Der $\vec{x}$ nå er vektoren, eller listen, med alle x-verdiene i beregningen vår og konstantene er

$$c_1 = i\frac{\hbar}{2m} \qquad\qquad c_2 = -\frac{i}{\hbar}$$

For å implementere et potensiale i Python er det lureste å bruke en funksjon. Denne funksjonen må ta inn et sett med x-verdier og returnere ett tall for hver x. Funksjonen kan jo også avhenge av andre parametre. For eksempel avhenger også potensiale til en harmonisk oscillator av bølgetallet. Poenget er at den må returnere ett tall for hver x-verdi.

I dette eksempelet implementerer vi et potensialsteg. Dette potensiale er definert som:

$$V(x) = \begin{cases} -35 \text{ MeV} & \text{hvis } x < 0 \\ 0 & \text{hvis } x \geq 0 \end{cases}$$

Og vi kan implementere det i Python på denne måten:

```
from numpy import *

def V( x ):
    '''
    Et potensialsteg nedover.
    '''
    # Ny tom array som er like lang som x
    potential = zeros( len( x ) )

    # Kreativ indeksering: Hvert element i potential som
    # har samme posisjon (indeks) som et element i x som
    # er mindre enn eller lik 0 settes til -35
    potential[ x <= 0 ] = -35 # MeV

    return potential
```

Når vi da vil hente ut potensialet for et sett med x'er kan vi gjøre det vektorisert på denne måten:

```
x = linspace( a, b, n )
V_x = V( x )
```

Nå er det bare å legge inn dette i selve beregningsløkken.

```
while t < T:
    # Regner ut den deriverte
    dPsidx2[1:nx-1] = ( Psi[ 2:nx ] - 2 * Psi[ 1:nx-1 ] + Psi[
        0:nx-2 ] ) / dx**2

    # Regner ut ny Psi med potensialet
    Psi = Psi + dt * ( c1 * dPsidx2 + c2 * V_x )

    # Legger til et tidssteg
    t += dt
```

Når dette er gjort er det bare å bygge resten av programmet på akkurat samme måte som vi gjorde uten et potensiale. Når alt settes sammen kan det se ut som i **Listing 6.2**.

Listing 6.2: Programmet bruker Eulers metode (Forward Euler) til å løse den tidsavhengige Schrödingerlikninga med et potensialsteg.

```
from numpy import *
from matplotlib.pyplot import *

E0p = 938.27                # Hvileenergi for proton [MeV]
hbarc = 0.1973              # [MeV pm]
```

```
c = 3.0e2                       # Lyshastighet [pm / as]

def Psi0( x ):
    '''
    Initialtilstand for gaussisk boelgepakke. Deler formelen
    opp i konstant og uttrykk for aa faa det mer oversiktlig.
    '''
    x0 = 0.100 # [pm]
    a = 0.0050 # [pm]
    l = 100000.0 # [1 / pm]

    A = ( 1. / ( 2 * pi * a**2 ) )**0.25
    K1 = exp( - ( x - x0 )**2 / ( 4. * a**2 ) )
    K2 = exp( 1j * l * x )

    return A * K1 * K2

def V( x ):
    '''
    Et potensialsteg nedover.
    '''
    # Ny tom array som er like lang som x
    potential = zeros( len( x ) )

    # Kreativ indeksering: Hvert element i potential som
    # har samme posisjon (indeks) som et element i x som
    # er stoerre enn eller lik 0 settes til -35
    potential[ x <= 0 ] = -35 # [MeV]

    return potential

nx = 801 # Antall punkter i x-retning
np = 1e2 # Plotte kun hver hver np'te utregning (bedre "
    framerate")

dx = 0.001 # Avstand mellom x-punkter

# Lar halvparten av x-verdiene vaere paa hver sin side av null
a = -0.5 * nx * dx
b = 0.5 * nx * dx
x = linspace( a, b, nx )

# Erklaerer tomme arrayer for den deriverte og Psi.
# Spesifiserer at den deriverte er kompleks
dPsidx2 = zeros(nx).astype(complex64)
Psi = Psi0( x )

# Tidsparametre
T = 0.0085 # Hvor lenge simuleringen skal kjoere [as]
```

```
dt = 1e-7 # Avstand mellom tidssteg [as]

# Henter ut potensialet
V_x = V( x )

# Aktiverer interaktiv modus
ion()

# Tegner initialtilstand
figure()
line, = plot(x, abs(Psi)**2)
draw()

# Konstanter
c1 = ( 1j * hbarc * c) / (2. * E0p )
c2 = -( 1j * c ) / hbarc

t = 0 # Teller tid
c = 1 # Teller antall utrengninger
while t < T:
    # Regner ut den deriverte
    dPsidx2[1:nx-1] = ( Psi[ 2:nx ] - 2*Psi[ 1:nx-1 ] + Psi[ 0:
        nx-2 ] ) / dx**2

    # Regne ut ny Psi med potensialet
    Psi = Psi + dt * ( c1 * dPsidx2 + c2 * V_x * Psi )

    # Sjekker om den skal plotte denne utregningen
    if c == np:
        line.set_ydata(abs(Psi)**2)
        draw()
        c = 0 # Nullstiller teller

    t += dt # Legger til et tidssteg
    c += 1 # Legger til en utregning

# Av med interaktiv modus
ioff()
# Beholder vinduene
show()
```

Kapittel 7

Stabilitetsproblemer

Til nå har vi sett på Eulers metode (Forward Euler) samt variasjoner av denne. De har alle til felles at de fort ble veldig ustabile hvis vi ikke hadde et veldig lite tidssteg. Man kan se på hvor gode numeriske løsningsalgoritmer er ved hjelp av to kriterier: Hvor stabile de er og hvor stor feil de lager.

Schrödingerligningen er en vanskelig ligning å løse stabilt ved hjelp av standard numeriske metoder fordi den er "stiv". Det betyr at den inneholder ledd som kan ha veldig raske endringer. For eksempel er e^x-funksjonen som regel problematisk.

Det finnes likevel en måte å unngå dette på uten å måtte senke tidssteget enda mer, men det krever at vi ser på et nytt sett med metoder. Eller snarere en annen måte å definere de samme metodene på.

7.1 Eksplisitte og implisitte metoder

Alle metodene vi har vært borti sålangt har hatt en ting til felles: Uttrykket for å regne ut det neste steget har vært helt eksplisitt - alle verdiene var kjent fra før unntatt den ene vi skulle regne ut. Altså bruker metoden tilstanden i det forrige tidspunktet for å kalkulere det neste.

$$\Psi(x_i, t + \Delta t) = \Psi(x_i, t) - \Delta t \frac{i}{\hbar} \hat{H} \Psi(x_i, t).$$

En implisitt metode bruker i stedet tilstanden i det nåværende tidspunktet og finner løsningen på neste tidspunkt ved å løse en ligning. For eksempel er

den implisitte versjonen av Eulers metode for Schrödingerligningen

$$\Psi(x_i, t + \Delta t) = \Psi(x_i, t) - \Delta t \frac{i}{\hbar} \hat{H} \Psi(x_i, t + \Delta t). \tag{7.1}$$

Den eneste forskjellen er at i stedet for å bruke det forrige tidssteget i siste ledd brukes i stedet det du vil finne ut. Dette blir altså en ligning vi vil løse for $\Psi(x_i, t + \Delta t)$. For å implementere dette må det altså ekstra innsats til.

Som regel har implementasjon av en implisitt metode flere fordeler enn ulemper. De blir kvitt stabilitetsproblemer slik at tidssteget ikke trenger å være like forsvinnende lite lenger. Noen ganger får vi desverre stabilitet på bekostning av nøyaktighet.

I neste seksjon ser vi på hvordan denne kan implementeres og hvilke fordeler og ulemper den har sammenlignet med den eksplisitte Eulermetoden.

7.2 Eulers metode implisitt (Backward Euler)

Selv om den rene Backward Euler-metoden fører med seg noen ulemper når den brukes på Schrödingers ligning (som vi kommer inn på senere) er den ofte nyttig for de fleste problemer.

For å implementere den for å løse den tidsavhengige Schrödingerligningen må vi for hvert tidssteg løse ligningen i (7.1). Dette kan gjøres ved bruk av lineær algebra og sterke funksjoner fra scipy for å løse ligningssett.

For en fri partikkel ser den tidsavhengige Schrödingerligningen ut som

$$\frac{\partial \Psi}{\partial t} = i \frac{\hbar}{2m} \frac{\partial^2}{\partial x^2} \Psi = c_1 \frac{\partial^2}{\partial x^2} \Psi = D\Psi. \tag{7.2}$$

Målet vårt er å få en matrise D som kan inneholde all informasjon som trengs når vi skal til neste steg. Den romlige deriverte kan vi skrive på matriseform slik at når vi lar den virke på vektoren $\vec{\Psi}$ ($\vec{\Psi}$ er kun en vektor som inneholder $\Psi(x_i)$ for hver i) deriverer vi hver $\Psi(x_i)$ ved bruk av differansen vi så på i seksjon 5.2.

Matrisen D kan da skrives som:

$$D = \frac{c_1}{\Delta x^2}\begin{pmatrix} -2 & 1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 1 & -2 \end{pmatrix} \tag{7.3}$$

Differansen vi bruker henter den sentrale differansen. Du kan la matrisen virke på $\vec{\Psi}$ for å sjekke at vi får ut de differansene vi vil ha. Som du sikkert har skjønt er dette akkurat det vi gjorde i implementasjonen av Forward Euler men bare på matriseform.

Nå kan vi reformulere hvert steg til

$$\Psi(\vec{x}, t + \Delta t) = \Psi(\vec{x}, t) + \Delta t D\Psi(\vec{x}, t + \Delta t).$$

Som vi kan skrive om til (husk at vi ikke kan forandre på hva operatoren D virker på, den er ikke kommutativ)

$$\begin{aligned} \Psi(\vec{x}, t + \Delta t) - \Delta t D\Psi(\vec{x}, t + \Delta t) &= \Psi(\vec{x}, t) \\ (I - \Delta t D)\Psi(\vec{x}, t + \Delta t) &= \Psi(\vec{x}, t) \end{aligned}$$

Der I er identitetsmatrisen. Legg merke til at vi nå har en ligning på formen

$$A\vec{x} = \vec{b}.$$

Slike ligninger kan vi løse med metoden spsolve() fra scipy-pakken. Dette er en funksjon som løser ligningssett for oss. Fra lineær algebra vet vi at vi kan skrive ligningssett av n ukjente som en matrise-vektor ligning. Funksjonen kunne da for eksempel brukt Gauss-eliminasjon for å finne de ukjente. [1]

Altså, for videre å implementere dette er det et par ting vi må gjøre:

[1] Selv om dette sannsynligvis ikke er det som foregår bak kulissene her siden Gauss-eliminasjon er en numerisk "tung" algoritme. Det betyr at det er veldig mange beregninger som må gjøres for å finne løsningen. Med tanke på at denne metoden også forutsetter av vi har med "sparse"-matriser, matriser som for det meste inneholder 0, finnes det i stedet andre algoritmer som kan gjøre utregningen veldig effektiv på datamaskinen.

1. Importere det vi trenger fra scipy
2. Lage en identitetsmatrise
3. Lage matrisen D (7.3)
4. Forandre på utregningen i løkken

Det vi trenger fra scipy-biblioteket er to ting: Verktøy for å jobbe med "sparse"-matriser (matriser som i hovedsak inneholder 0) og en pakke med verktøy til lineær algebra. Disse importeres ved å skrive:

```
import scipy.sparse as sparse
import scipy.sparse.linalg
```

Det finnes nå en innebygd funksjon i sparse-pakken for å lage en identitetsmatrise.

```
I = sparse.identity(n)
```

Antatt at du allerede har en parameter i programmet n som sier hvor mange punkter du har langs x-aksen.

Det neste vi kan gjøre er å lage matrisen D. For å lage en matrise og sette spesielle verdier langs diagonalene kan vi bruke funksjonen spdiags(). Den tar fire argumenter:

1. data en matrise som inneholder det som skal ligge langs diagonalen som rader. Nå må vi tenke tilbake på multidimensjonale lister. Vi vil at den skal se slik ut:

   ```
   [ [ 1 1 1 1 ... 1 1 1 1 ]
     [ -2 -2 -2 -2 ... -2 -2 -2 -2 ]
     [ 1 1 1 1 ... 1 1 1 1 ] ]
   ```

   Fordi vi her har tre "diagonaler", alle -2 skal ligge på den samme diagonalen mens det skal ligge 1 på diagonalen under og over der det ligger -2. Se tilbake på (7.3) hvis du tviler.

   For å lage denne matrisen må vi derimot være litt smarte, fordi vi orker ikke å skrive inn tallene selv. Derfor kan vi bruke ones() funksjonen fra numpy og kreativ indeksering. ones() lager en array som kun inneholder ett-tall.

   Hvis vi lager en array ved å skrive

```
data = ones( ( 3, n ) )
```

og deretter setter hele andre rad til -2:

```
data[1,:] = -2
```

vil den være på den formen vi ønsket.

2. diags. Her spesifiserer vi hvilken av de tre radene fra data som skal ligge hvor. I vårt tilfelle vil vi at den med -2 skal ligge i midten og de to andre under og over. Da gir vi 0 for å signalisere i midten og 1 og -1 for under og over. I vårt tilfelle blir det altså:

```
diags = [-1, 0, 1]
```

3. nx og ny. Disse sier hvilken dimensjon vi vil ha på matrisen. I vårt tilfelle skal den være $n \times n$.

For å lage hele denne matrisen kan vi altså skrive:

```
data = ones( ( 3, n ) )
data[ 1,: ] = -2
diags = [ -1, 0, 1 ]
D = c1 * sparse.spdiags( data, diags, n, n ) / dx**2
```

Nå har vi alt vi trenger for å forandre på selve utregningen. Nå skal vi løse dette ligningssettet for hvert tidssteg. Det gir derfor mening å lage nye variable A og b for hver kjøring også løse dem med sparse.linalg.spsolve(A, b). Hovedløkken vil da se noe slik ut:

```
while t < T:
    A = ( I - dt * D )
    b = Psi_old

    Psi_new = sparse.linalg.spsolve( A, b )

    t += dt
```

Hvis du nå setter disse forandringene inn i eksempelet vårt med den gaussiske bølgepakken. Vil du se at den ligner veldig. I tillegg kan du øke tidssteget med et par tierpotenser og fortsatt se at løsningen ikke eksploderer.

En stor ulempe med denne metoden er at den ikke bevarer normeringsbetingelsen:

$$\int_{-\infty}^{\infty} |\Psi|^2 dx = 1$$

Dette kan fikses manuelt ved å normere på nytt for hvert tidssteg på denne måten:

```
Psi = Psi / sp.integrate.simps(abs( Psi )**2, dx=dx)
```

Men det fører til at metoden tar mer kraft. I tillegg er ikke denne metoden like nøyaktig som den vanlige Forward Euler. I neste seksjon ser vi på en videre forbedring som utnytter begge disse metodene for å lage en stabil fremgangsmåte som skaper en liten feil og bevarer normeringsbetingelsen. Metoden er kjent som Crank-Nicolsons metode.

7.3 Crank-Nicolsons metode

Crank-Nicolsons metode er ikke mer avansert enn at den tar gjennomsnittet av bidraget fra et steg med Forward Euler og et steg med Backward Euler. Det kan sies at den på denne måten får det beste fra begge verdener - stabilitet fra Backward Euler og nøyaktighet fra Forward Euler.

Metoden formuleres altså på følgende måte:

$$\Psi(\vec{x}, t + \Delta t) = \Psi(\vec{x}, t) + \frac{\Delta t}{2} D \left[\Psi(\vec{x}, t + \Delta t) + \Psi(\vec{x}, t)\right]$$

Denne ligningen må vi igjen løse for $\Psi(\vec{x}, t + \Delta t)$, siden det er den vi er interessert i.

$$\begin{aligned}
\Psi(\vec{x}, t + \Delta t) &= \Psi(\vec{x}, t) + \frac{\Delta t}{2} D\Psi(\vec{x}, t + \Delta t) + \frac{\Delta t}{2} D\Psi(\vec{x}, t) \\
\Psi(\vec{x}, t + \Delta t) - \frac{\Delta t}{2} D\Psi(\vec{x}, t + \Delta t) &= \Psi(\vec{x}, t) + \frac{\Delta t}{2} D\Psi(\vec{x}, t) \\
\left[I - \frac{\Delta t}{2} D\right] \Psi(\vec{x}, t + \Delta t) &= \left[I + \frac{\Delta t}{2} D\right] \Psi(\vec{x}, t)
\end{aligned}$$

Nå har vi en ligning på den samme formen som for Eulers metode implisitt, men vi har unngått normeringsproblemet og har en mye bedre metode.

Hovedberegningsløkken vil nå se slik ut:

```
while t < T:
    A = ( I - 0.5 * dt * D )
    b = ( I + 0.5 * dt * D ) * Psi_old

    Psi_new = sparse.linalg.spsolve( A, b )

    t += dt
```

7.4 Implementasjon av potensiale

For å utføre beregningen med et potensiale må vi forandre litt på uttrykket, men ikke mye. Vi kommer til å bruke samme fremgangsmåte som i seksjon 6.3 men i stedet bruke metoden Crank-Nicolson.

I stedet for som i (7.2) blir nå den tidsavhengige Schrödingerligningen:

$$
\begin{aligned}
i\hbar\frac{\partial\Psi}{\partial t} &= \left(-\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2} + V(x)\right)\Psi \\
\frac{\partial\Psi}{\partial t} &= \left(\frac{i\hbar}{2m}\frac{\partial^2}{\partial x^2} - \frac{i}{\hbar}V(x)\right)\Psi \\
\frac{\partial\Psi}{\partial t} &= \left(c_1\frac{\partial^2}{\partial x^2} - c_2V(x)\right)\Psi = (D+M)\,\Psi
\end{aligned}
$$

Vi trenger altså en matrise til, $M = c_2V(\vec{x})$, som er en diagonal matrise med vektoren $V(\vec{x})$ langs diagonalen. Hvis du lar M virke på $\Psi(\vec{x})$ skal du få ut $V(x_i)\Psi(x_i)$ for alle x_i i vektoren $\vec{x}$.

Igjen bruker vi samme funksjon som vi brukte for å lage derivasjonsmatrisen i D, spdiags(). Hvis vi antar at vi allerede har skrevet en funksjon V(x) som returnerer potensiale for hver x vi putter inn kan vi, som sett tidligere, hente ut hele potensiale på denne måten:

```
V_x = V( x )
```

Videre vil vi ha en matrise som har denne, V_x langs diagonalen. Den lager vi på denne måten:

```
c2 = -1j * c / hbarc
M = c2 * spdiags( V_x, [0], n, n )
```

Hvis vi minnes hvordan vi lagde den diagonale matrisen D i seksjon 7.2 ser vi at argumentet diags her bare er [0] siden vi bare har en rad og den skal ligge langs diagonalen. Matrisen skal også være like stor som D, så de to siste argumentene blir de samme.

Nå har vi alt vi trenger. Det eneste vi må gjøre nå, er å forandre litt på selve metoden siden ligningen vi skal løse i hvert steg ser noe annerledes ut. Crank-Nicolsons metode er:

$$\Psi(\vec{x}, t + \Delta t) = \Psi(\vec{x}, t) + \frac{1}{2}\Delta t(D + M)(\Psi(\vec{x}, t) + \Psi(\vec{x}, t + \Delta t))$$

Ganger ut og flytter over:

$$\Psi(\vec{x}, t + \Delta t) = \Psi(\vec{x}, t) + \frac{\Delta t}{2}(D + M)\Psi(\vec{x}, t) + \frac{\Delta t}{2}(D + M)\Psi(\vec{x}, t + \Delta t)$$

$$\Psi(\vec{x}, t + \Delta t) - \frac{\Delta t}{2}(D + M)\Psi(\vec{x}, t + \Delta t) = \Psi(\vec{x}, t) + \frac{\Delta t}{2}(D + M)\Psi(\vec{x}, t)$$

$$\left[I - \frac{\Delta t}{2}(D + M)\right]\Psi(\vec{x}, t + \Delta t) = \left[I + \frac{\Delta t}{2}(D + M)\right]\Psi(\vec{x}, t)$$

Nå er uttrykket på riktig form, og vi ser at det eneste som er forskjellig er at vi skal summere sammen de to matrisene D og M for hvert steg. Antatt at matrisen M er definert som i kodeeksemplemene i denne seksjonen vil beregningsløkken se slik ut:

```
while t < T:
    A = ( I - 0.5 * dt * D + M )
    B = ( I + 0.5 * dt * D + M ) * Psi_old

    Psi_new = sparse.linalg.spsolve( A, b )

    t += dt
```

Nå kan vi løse den tidsuavhengige Schrödingerligningen for mange forskjellige potensialer.[2]

[2]Hvis du har planlagt å fortsette til kurset FYS3150 - Computational physics, kommer du til å lære mer nøyaktig hvordan denne metoden fungerer. I tillegg kommer du til å lære mange andre veldig interessante metoder.

Tillegg A

Ressurser

Som man lett får inntrykket av så er programmering et veldig dynamisk emne der funksjoner og biblioteker kommer og går særdeles ofte. I tillegg er det vanskelig å gjengi en liste der man alltid har det man trenger tilgjengelig.

Med det i bakhodet kan du bruke dette kapittelet for å finne steder du kan finne mer informasjon om de forskjellige bibliotekene — både de "innebygde" i Python og andre.

De fleste av disse sidene prøver å gjøre det så oversiktlig som mulig å søke gjennom hele katalogen med funksjoner og finne akkurat den beskrivelsen du trenger.

Nøl ikke med å søke hvis det er noe du lurer på om det er mulig å gjøre. Det finnes ofte mange flere funksjoner, og funksjoner for mange flere formål, enn det man ser for seg!

Skulle alt annet feile er også god bruk av søkemotorer en programmerers beste venn.

A.1 Innebygde biblioteker

- Scipy oppslag

  `http://docs.scipy.org/doc/scipy/reference/`

- Numpy oppslag

`http://docs.scipy.org/doc/numpy/reference/`

A.2 Andre biblioteker

- Matplotlib oppslag

  `http://matplotlib.org/api/`

A.3 LaTeX

- Gode nettsideressurser, inneholder også introduksjon.

  `http://en.wikibooks.org/wiki/LaTeX`

A.4 Numeriske metoder

- Programmeringskompendiet til FYS3150 inneholder noe av det vi har gått gjennom her samt mye mer.

  `http://www.uio.no/studier/emner/matnat/fys/FYS3150/h13/undervisningsmateriale Lecture%20Notes/lectures2013.pdf`

A.5 Editorer

- Emacs for Mac

  `http://emacsformacosx.com`

- MacVim

  `https://code.google.com/p/macvim/`

  God guide for hvordan å innstallere mye ekstra funksjonalitet.

  `http://haridas.in/vim-as-your-ide.html`

- Spyder (Python IDE, alle platformer)

  `https://code.google.com/p/spyderlib/`

- Sublime text (Fin editor for Mac, gratis)

  `http://www.sublimetext.com`

- Notepad++ (God editor for Windows, gratis)

  `http://notepad-plus-plus.org`