Python i LinAlys og MekRel

Niels Bohr Institutet

Aug 03, 2021

CONTENTS

Jonathan Melcher, Johann Bock Severin, Linea Stausbøll Hedemark, Børge Svane Nielsen & Sune Olander

I fysik benytter vi Python som programmeringssprog. Denne side kommer til at indeholde alt materiale om Python, som bliver brugt i kurserne MekRel og LinAlys i blok 1 og 2. Siden er bygget op, så der er en fælles installationsguide og en lille guide til, hvordan man kommer i gang med at kode Python. Derefter vil brugen af Python dog være meget forskellig fra LinAlys og MekRel og der er derfor to sektioner, hvor hvert kursus har noter liggende.

I MekRel-sektionen vil man finde noter, der introducerer, hvordan man foretager numeriske udregninger, fitter funktioner og plotter, så man kan lave databehandling i laboratoriet. Dette gøres primært ved brug af pakkerne NumPy, SciPy og Matplotlib.

I LinAlys-sektionen er fokus på at benytte Python som CAS-værktøj til at foretage og tjekke symbolske udregninger. Dette gøres med pakken SymPy, hvorfra vi vil introducere udvalgte værktøjer.

Hvad er Python?

Når vi programmerer en computer, vælger vi et programmeringssprog. Dette sprog afgør, hvordan vores instruktioner bliver oversat til maskinkode, som så kan køre på vores computer.

Der findes mange forskellige programmeringssprog med hver deres fordele og ulemper. Python er bygget op til at ligge så tæt som muligt på det engelske sprog. Dette betyder ofte, at Python ikke er ligeså hurtigt at køre, men forhåbentlig lidt mere *intuitivt* end mange andre sprog. Desuden har Python den fordel, at det er i meget hurtig vækst, så man kan næsten finde implementeringer af alle tænkelige algoritmer i sproget.

Hvad skal vi bruge det til?

På fysik benyttes Python til mange forskellige opgaver, og det er derfor en rigtig god investering af tid at lære. Allerede i løbet af det første halve år vil I se det blive brugt til databehandling i laboratoriet og til symbolske udregninger i LinAlys. Senere i studiet kan man se det brugt til simuleringer, numeriske udregninger og måske bare til at automatisere udregninger, som man foretager ofte.

I blok 1 og 2 forventer vi dog ikke, at du for alvor lærer at programmere. Der kommer senere på studiet et dedikeret programmeringskursus. Det er her i starten snarere meningen, at du skal stifte bekendtskab med Python som et værktøj. Man kunne have valgt andre værktøjer, der til de enkelte opgaver ville være mindst lige så gode, men vi har på førstedelen af fysikstudiet valgt at bruge Python som værktøj overalt hvor det er muligt, så den tid, du bruger på at lære Python, vil komme dig til gode i andre kurser. Vi håber at du med de forskellige noter på denne side kan stykke små koder sammen, som kan løse de problemer, du støder på. Ellers må du endelig søge assistance: både i Mekrel og LinALys er der hjælp at hente hver eneste uge.

CHAPTER

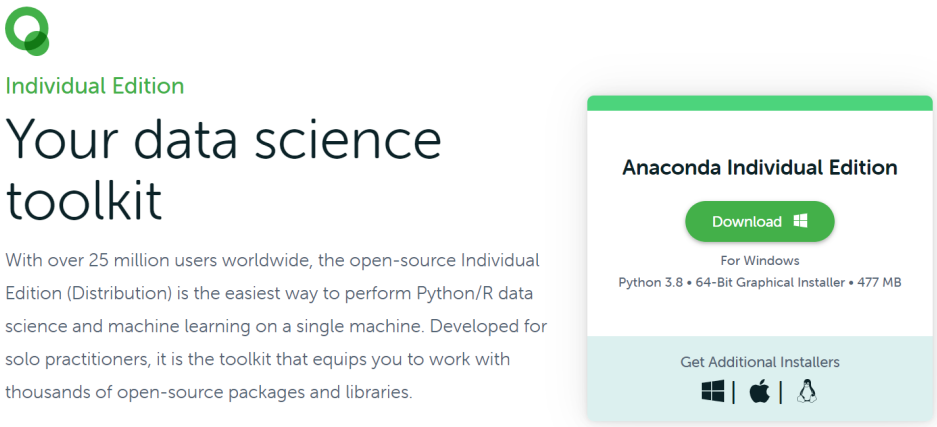
ONE

INSTALLATION AF PYTHON

Før vi kan gå i gang med at bruge Python, må vi først få installeret. På fysik har vi besluttet at installere Anaconda implementationen, da denne kommer med stort set alle de pakker, som man skal bruge, når man starter op. Vi kan derfor kun nøjes med at foretage en installation.

1.1 Download Anaconda

Gå først ind på https://www.anaconda.com/products/individual. Det bør gerne se således ud;



Scroll ned til bunden og tryk på den installation, der passer til dit system.

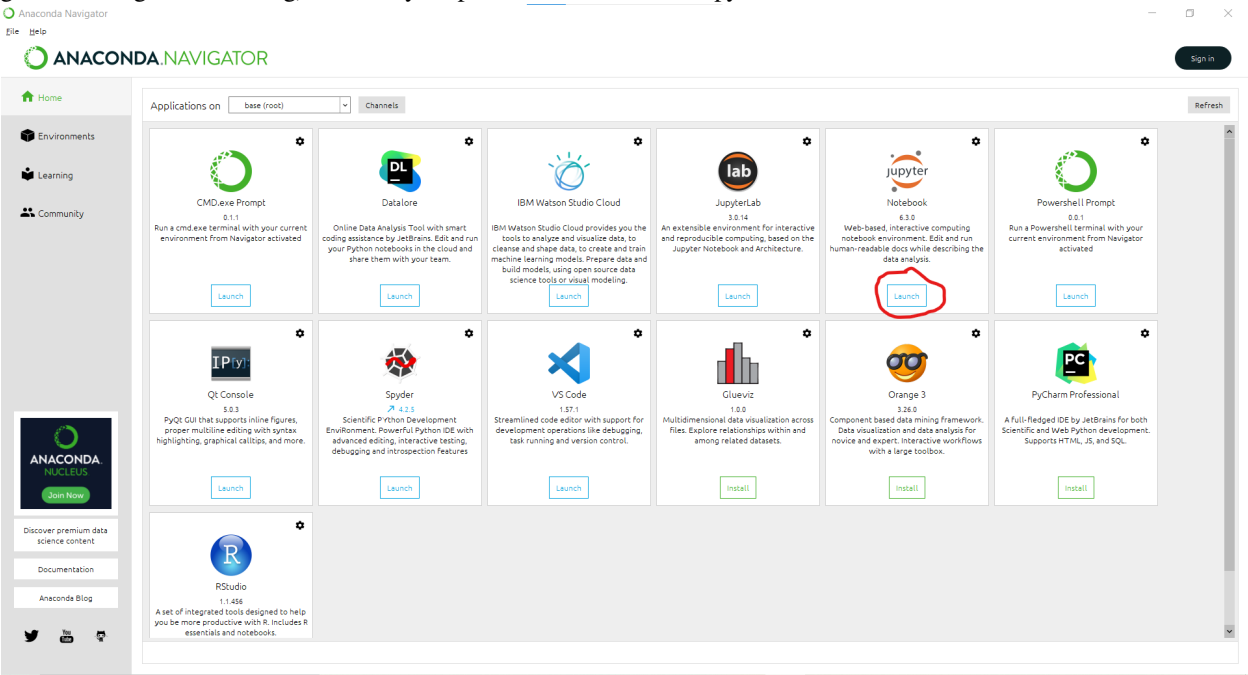
Gem installeren og kør den.

1.2 Installation

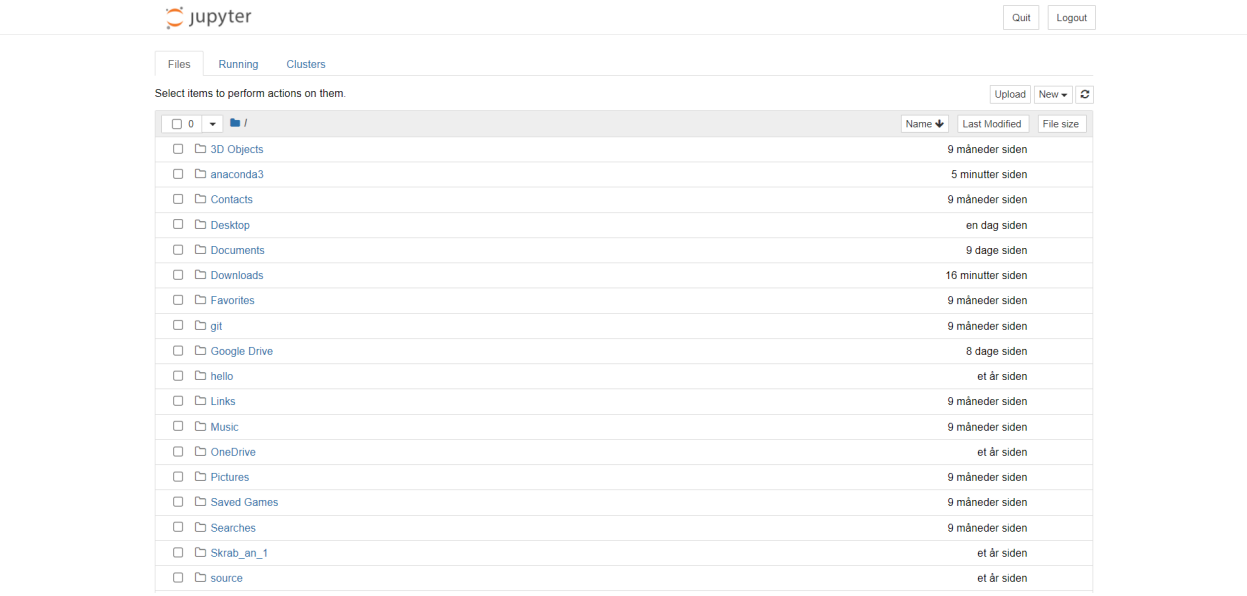
Tryk *Next* > *I agree* > *Next*, vælg det sted du vil installere Anaconda (evt. bare den foreslået lokation), > Tryk ja i de to ticks, hvis du aldrig har arbejdet med Python før. *Install* > *Next* > *Next* > Fjern de to ticks og tryk finish.

1.3 Åbne Jupyter Notebook

Åben Anaconda Navigator, og kør den (vær ikke bange hvis der kommer en terminal op, det gør der også hos mig). Tryk på *launch* under Jupyter Notebook, markeret med rød under

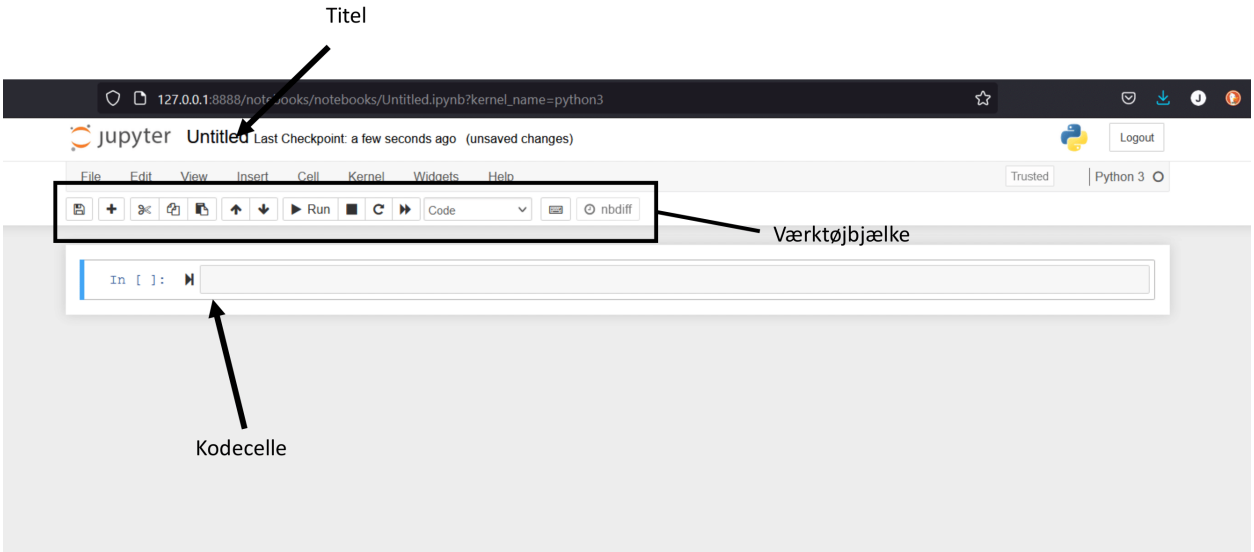


OBS: Her kan du se den mappe, som min åbner. Jeg vil ikke gemme min kode her, og det vil du nok heller ikke. For at holde styr på sit arbejde, kan man med fordel gemme sin kode i forskellige mapper. Derfor skal man navigere derhen, hvor man gerne vil gemme sin kode. Det kunne være en mappe, der hedder `MekRel_Python_aflevering_1`.

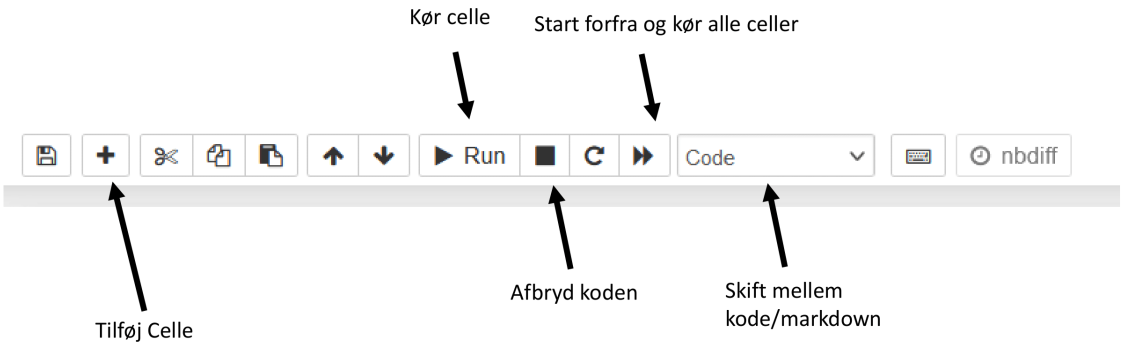


Tryk så på *New* (oppe til højre) > Python 3.

Du har nu åbnet en Python notebook og kan nu skrive Python. Selv Notebooken har følgene layout:



De vigtigste knapper at kende på værktøjsbjælken kan findes her:



Nu er du klar til at programmere i Python.

CHAPTER

TWO

KOM I GANG MED PYTHON

På fysik stifter I bekendtskab med Python i de to førsteårskurser MekRel og LinAlys. Der er mange forskellige måder at skrive Python på, men vi vil foreslå, at I benytter Jupyter Notebooks, som kan findes i en installation af Anaconda.

I Notebooks skriver man sin kode i celler, som derefter kan "køres". Der bliver primært brugt to slags celler: "Code" eller "Markdown". Denne celle er en Markdown-celle og indholdet formateres som tekst, når cellen køres (der laves altså ingen beregninger). Markdown-celler kan bruges til at skrive noter i: det kunne være ens overvejelser i forbindelse med et forsøg i laboratoriet eller en forklaring af tankegangen bag en beregning. Kodeceller er de celler, der indeholder de kommandoer, der laver vores beregninger når cellen køres.

Tip: Man kan med fordel bruge tastaturgenvejen `shift` + `enter` for at køre en celle, og bevæge sig videre til den næste celle i rækken.

2.1 Hvordan virker Python?

Når vi skriver kode i en celle og kører cellen, bliver koden læst linje for linje. Et computerprogram bliver altså en liste af computer-operationer, som skal foretages i en bestemt rækkefølge. Hvis man vil markere noget, som comptueren ikke skal køre, kan det markeres med #. Dette bruges ofte til kommentar til den, der læser koden.

Når man arbejder i Python, gemmes svaret på en operation ikke nødvendigvis. Vi benytter derfor = til at definere en variabel. Dette kunne eksempelvis være a = 5 . Da vil vi kunne indsætte a i et udtryk, og a vil nu have værdien 5.

En god måde at se en værdi på, er ved at bede Python om at printe den. Dette gøres ved at skrive print(X), hvor X kan være et hvilket som helst variabelnavn.

Eksempelvis kan vi definere variablen hilsen og printe den.

```
hilsen = "Hello World" # Anførselstegnene (eller "gåseøjnene") indikerer at variablen␣
↪tildeles et tekstudtryk, en såkaldt "string"
print(hilsen)
```

```
Hello World
```

I en notebook er det faktisk ikke nødvendigt at printe den *sidste* linje, man kan blot skrive variabelnavnet, og så vil Jupyter selv printe det:

```
hilsen
```

```
'Hello World'
```

2.1.1 Variabeltyper

Udover tekststrenge, som vi så i ovenstående eksempel, findes flere forskellige typer variable, som man kan benytte sig af. I Python er det ikke nødvendigt at angive typen af en variabel, da Python selv finder ud af det fra konteksten.

De mest almindelige variabeltyper er:

- Integers (heltal): her skriver vi blot et helt tal, som eksempelvis a = 6
- Float (decimaltal): indgår der et decimalpunktum, '.', får vi automatisk en float-variabel, som eksempelvis kroner_per_euro = 7.46.
- String (tekststreng): Når vi vil have en variabel til at indeholde tekst, markeres det med anførselstegn/gåseøjne, f.eks. som ovenfor hilsen = "Hello World"
- Boolean (sandt/falsk): Hvis vi skal angive en værdi som enten sandt eller falsk, bruger vi en "boolean variable" og kan skrive switch = True. Falsk skal gives ved False, og bemærk igen at der er forskel på Svar = True og Svar = "True", idet Svar i sidstnævnte tilfælde istedet bliver en string, som indeholder ordet True, og ikke et logisk udtryk.

2.1.2 Lister af variable

Vi kan sammensætte lister af flere værdier, eksempelvis en samling af tal. Dette gøres mest almindeligt ved at lave en liste.: tal = [4, 7, 10]. Nu vil tal altså indeholde værdierne 4, 7 og 10. Hvis vi ønsker et enkelt element fra vores liste, kan vi referere til det ved at skrive tal[index], hvor værdien af index angiver placeringen i listen af det ønskede tal.

Tip: Python starter ved nul (nul-indeksering), så det første element er altså tal[0].

```
tal = [4, 7, 10]
tal[1]
```

```
7
```

Vi ser altså, at når vi kalder tal[1] får vi det andet element ud, som i definitionen var givet ved 7. Dette er et resultat af nul-indeksering. Dette er en hyppig kilde til fejl, når man ikke er vant til Python, så tjek det, hvis koden ikke virker eller giver uventede resultater.

Elementer kan tilføjes til listen ved at benytte koden .append(element) efter vores liste. Dette tilføjer elementet til den pågældende variabel i slutningen af listen. Eksempelvis, hvis vi ønsker at udvide vores tal-liste, kan vi skrive:

```
tal.append(6)
tal
```

```
[4, 7, 10, 6]
```

og vi har nu tilføjet 6 som et ekstra element i vores liste.

2.2 Basale regneoperationer

Python indeholder som udgangspunkt de mest basale regneoperationer, som man kender fra sin lommeregner. Eksempelvis kan vi lægge til, trække fra, gange og dividere ved blot at skrive udtrykket op med '+', '-', '*' og '/'. På sædvanlig vis kan vi også benytte parenteser til at give rækkefølgen for operationerne. Nogle enkelte eksempler er givet nedenunder:

```
# Vi adderer blot to tal
2 + 2
```

```
4
```

```
# Angiver vi ikke parenteser, følger Python selv regne-hierakiet
9 - 3 * 2
```

```
3
```

```
# Men vi kan selvfølgelig ændre dette med parenteser
(9 - 3) * 2
```

```
12
```

```
# Og vi skal selvfølgelig ikke dividere med 0, hvis ikke vi ønsker en fejl
100 / 0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-9-5e10904e16be> in <module>
      1 # Og vi skal selvfølgelig ikke dividere med 0, hvis ikke vi ønsker en fejl
----> 2 100 / 0

ZeroDivisionError: division by zero
```

Udover disse operationer, kan vi også opløfte i en potens ved at benytte **.

```
print(17 ** 2) # Kvadratet på 17

print(81 ** (1/2)) # Eller benytte at kvadratroden af x er x**(1/2) til at finde␣
↪kvadratroden af 81
```

```
289
9.0
```

```
print(81 ** 1/2) # Bemærk at dette IKKE beregner kvadratroden af 81 ligesom ovenfor:␣
↪forskellen er parentesen.
```

```
40.5
```

Ønsker man at benytte funktioner som logaritmer, trigonometriske funktioner eller lignende, bliver man ofte nødt til at importere et modul, som kan indeholder disse funktioner. Men det vil vi komme ind på senere i noterne om SymPy og NumPy.

2.2.1 Ændring af variable

Der kan være mange årsager til, at man skal ændre værdien på en varibel. Det kunne f. eks. være i en simulation, hvor positionen af et objekt skal opdateres efter et lille tidsskridt. Benytter vi samme variabelnavn i en ny definition ændrer vi blot værdien:

```
vari = 15
print(vari) # Her er variablen lig med 15
vari = 20
print(vari) # Nu er den sat til 20
```

```
15
20
```

Når vi benytter et lighedstegn, så udregnes først værdien på højre side, inden værdien bliver gemt med variabelnavnet til venstre. Vi kan altså godt benytte en variabel til at redefinere sig selv. Her benytter vi `vari` til at gøre `vari` 10 gange større.

```
vari = 7
print(vari)       # Sat til 7
vari = 10 * vari # `vari` bruges til at udregne en ny værdi for samme variabel
print(vari)
```

```
7
70
```

Der er dog også nogle indbyggede smutveje til at ændre værdien af variable i faste trin. Hvis man gerne vil lade en variabel vokse i skridt af increment = 2.5, kan man benytte smutvejs-notationen += til at ændre variablen i skridt af increment

```
increment = 2.5
value = 5
print(value)

value += increment # Vi øger value med værdien af den variable increment
print(value)
```

```
5
7.5
```

Dette virker også for de andre basale operationer ved at skrive -=, *=, og man kan eksempelvis benytte /= 2, til at redefinere et tal til at være halvt så stort.

2.3 Kontrolstrukturer og Løkker

Denne side bygger ovenpå den tidligere, og omhandler hvordan man kan bygge kode op med brugen af kontrolstrukturer og løkker til eksempelvis at gentage kode. I MekRel og LinAlys benytter vi dog primært moduler, som tager sig af den logik og de løkker, som man skal bruge. Derfor er det ikke strengt nødvendigt at læse den her side, selvom det er nogle meget brugbare emner, der helt sikkert vil være med til at forbedre forståelsen for Python.

2.3.1 Logik i Python

Når vi skriver programkode, er vi ret ofte interesseret i, om et udtryk er sandt eller falsk for vide, hvordan vi skal eksekvere resten af koden. Til dette benytter vi boolean variable, som enten kan være `True` eller `False`. Enten kan vi selv definere en variabel som værende sand ved at skrive `switch = True`, men oftere skal vi dog tjekke om noget er sandt.

Når man benytter to lighedstegn == tjekker Python om udtrykkene på hver side er lig hinanden. Eksempelvis kan vi se om vores defineret værdi er lig med 10:

Bemærk denne forskel: Et lighedstegn definerer en variabel og to lighedstegn efter hinanden sammenligner to udtryk:

```
i = 10 # Definer variabel
i == 10 # Sammenlign variabel med værdi
```

```
True
```

```
i += 2 # Vi øger værdien med 2
i == 10 # Og sammenligner igen
```

```
False
```

I det andet tilfælde har vi redefineret `i` til `i = 12` og udtrykket "`i` er lig med 10" er altså ikke længere sandt.

Udover at benytte == kan vi benytte forskellige andre tegn til at danne udsagn, der kan være sande eller falske:

- < eller > fortæller om en værdi er større eller mindre end en anden
- <= eller >= er ligesom overstående men inkluderer også lig med
- != er det modsatte af ==, altså *er forskelligt fra*, og svarer til det matematiske tegn $\neq$.
- not ændrer sandhedsværdien af et udtryk til det omvendte, eksempelvis er værdien af not True netop False og dermed er not 1 == 2 sandt idet 1 og 2 jo ikke er ens.
- in kan bruges til at spørge om et element er i en liste. Eksempelvis vil 7 in tal give `True`, hvis `tal` er en liste, som indeholder 7, som det gjorde på tidligere side.

2.3.2 If-else-statements

Boolean-logik er særligt nyttigt hvis vi skal udføre forskellige beregninger når et bestemt udtryk er sandt og noget andet, hvis det ikke er. Her kan vi benytte if/else statements. Her ser syntaksen sådan ud:

```
if Statement:
    ~ gør det her ~
else:
    ~ gør noget andet ~
```

Hvad der sker, afhænger af sandhedsværdien af udtrykket `Statement`. Hvis det logiske udsagn `Statement` er sandt, udføres første del af koden, ellers udføres den anden del. Man kan udelade else, og i dette tilfælde vil der kun ske noget såfremt `Statement` er sandt. Bemærk at indrykningen spiller en vigtig rolle i at afgrænse, hvad der skal gøres i de enkelte tilfælde.

Her er et eksempel, hvor vi vil tage absolutværdien af et tal:

```
a = -100
if a < 0: # Hvis a er negativt, så gør følgende:
    # Vi printer, hvis vi går ind i denne del af koden.
    print("Bingo: Vi startede med et negativt tal, så if-betingelsen var opfyldt")
    a = - a   # Sæt a lig med sig selv med omvendt fortegn
print(a)
```

```
Bingo: Vi startede med et negativt tal, så if-betingelsen var opfyldt
100
```

Hvis vi gentager testen nu hvor a er blevet positiv, vil betingelsen i if-sætningen ikke være opfyldt, og der udføres ingen beregninger i if-sætningen:

```
if a < 0: # Hvis a er negativt, så gør følgende:
    # Vi printer, hvis vi går ind i denne del af koden.
    print("Bingo: Vi startede med et negativt tal, så if-betingelsen var opfyldt")
```

(continues on next page)

(continued from previous page)

```
    a = - a
print(a)
```

```
100
```

Bemærk at print(a) ikke er indrykket og dermed ikke er en del af if-sætningen. Værdien af a udskrives derfor uanset fortegn.

Et andet eksempel kunne være, hvis vi ønsker at dividere et tal med et andet, men have den tomme variabel None, hvis vi bliver bedt om at dividere med 0 (og dermed undgår en fejlmeddelelse). Til dette kan vi benytte vores if-else:

```
tæller = 10
nævner = 0

if nævner != 0: # Hvis nævneren er forskellige fra nul, kan brøken beregnes
    resultat = tæller / nævner
else: # Ellers kan vi ikke beregne resultatet
    resultat = None   # None er et tomt element. Denne har ingen værdi

print(resultat)
```

```
None
```

Hvis der er brug for flere rangordnede betingelser for at afgøre hvilken handling, koden skal udføre, kan man benytte elif second_statement (elif er kort for else-if). Så opbygger man syntaksen som

```
if Statement:
    ~ gør det her ~
elif Second_statement:
    ~ gør en andet ting ~
else:
    ~ ellers gør det her ~
```

Hvis man vil tjekke flere ting, kan man blot tilføje flere elif statements under hinanden.I dette tilfælde vil koden altid køre præcis én kodebid: Koden checker først `Statement` og hvis denne er sand vil den **kun** køre den tilhørende del af koden uanset om `Second_statement` også er sand. Hvis `Statement` er falskt, vil den tjekke elif-statementerne i rækkefølge og eksekvere koden hørende til den første sande betingelse. Hvis ingen af de givne betingelser er opfyldt, vil den køre koden der følger efter else.

2.3.3 Løkker/loops

Ofte bruger vi computeren til at gentage opgaver mange gange. Dette kan vi gøre ved brug af løkker (ofte kaldet *loops* ligesom på engelsk). Der er to typer: en for-løkke og en while-løkke. I denne Notebook vil vi dog kun vise for-løkker, som bruges ved at sætte følgende struktur op:

```
for variabel in liste:
    ~ gør det her ~
```

Løkken tildeler `variabel` værdier fra listen. Kodebiden, som er indrykket køres nu en gang for hvert element i listen, hvor `variabel` hver gang refererer til det næste element i listen. Det er normal praksis at kalde den variable i løkke-definitionen for i hvis der ikke er nogen særlig grund til at bruge et andet navn. Hvis vi eksempelvis vil beregne og udskrive kvadratet på 2, 3, 5 og 7, kan vi skrive det som:

```
tal = [2, 3, 5, 7]

for i in tal:      # Variablen hedder 'i' og vil efter tur antage værdierne i listen
 ↪'tal'.
    print(i ** 2) # Print kvadratet på tallet `i`
```

```
4
9
25
49
```

Hvis vi er interreseret i at gøre noget et bestemt antal gange, kan vi benytte range(antal) til at generere en liste med tallene fra 0 og op til antal - 1 (på grund af nulindekseringen så er der `antal` tal i listen). Lad os prøve at gange et tal med 10, printe det og så gentage dette 5 gange:

```
a = 3
for i in range(5): # Vi har denne gang ikke tænkt os at benytte 'i' til noget, men␣
 ↪det skal stadig være der
    a *= 10 # redefiner a ved at gange det med 10
    print(a)
```

```
30
300
3000
30000
300000
```

Vi kan også give range() flere argumenter. Kaldesekvensen er range(start, stop, skridtstørrelse) hvor alle tallene er heltal. Det er vigtigt at notere sig, at værdien 'stop' ikke kommer med i listen, så hvis vi ønsker at finde summen af alle tal i 5-tabellen fra 0 til 40, kan vi skrive:

```
summen = 0
for i in range(0, 45, 5): # For at nå op til 40 skal 'range' have et helt tal mellem␣
 ↪41 og 45 som værdien for 'stop'
    print(f"Vi lægger {i} til {summen}")
    summen += i

print(f"Og summen er {summen}")
```

```
Vi lægger 0 til 0
Vi lægger 5 til 0
Vi lægger 10 til 5
Vi lægger 15 til 15
Vi lægger 20 til 30
Vi lægger 25 til 50
Vi lægger 30 til 75
Vi lægger 35 til 105
Vi lægger 40 til 140
Og summen er 180
```

Vi havde fået samme resultat, hvis vi i stedet for 45 havde skrevet 41, da range forsætter indtil i'et er større eller lig med vores stop.

CHAPTER

THREE

INTRO TIL PAKKER

Python er et af de mest brugte programmeringsprog i verden, specielt i den akademiske verden. Derfor er der også en kæmpe mængde pakker, som kan løse alverdens problemer. Her i starten skal I stifte bekendskab med 4:

- Sympy (dokumentation)
- Numpy (dokumentation)
- Matplotlib (dokumentation)
- Scipy (dokumentation)

Det ligger næsten i navnet hvad de kan, og hvad de skal bruges til:

- Sympy (Symbolic Python) er til symbolsk matematik
- Numpy (Numerical Python) er til numeriske beregninger
- Matplotlib (Mathematical ploting library) er til at lave plots
- Scipy (Scientific Python) her skal vi kun bruge en funktion

Ligesom det er vigtigt at bruge den rigtige metode til at løse en ligning, er det ligeså vigtigt at bruge den rigtige pakke til et programmeringsproblem. Som tommelfingerregel skal I bruge Sympy i LinAlys til at foretage symbolske udreginger for at kontrollere svar og lave nogle simple plots af funktioner. I laboratoriet skal I bruge Numpy og Matplotlib til at behandle og illustrere jeres data og Scipy til at lave fits.

Nogle af pakkerne har noget overlap, og de er lavet til at kunne bruges sammen. Til at starte med vil vi dog prøve at undgå dette, da det hurtig kan blive meget kompliceret og nok en kilde til fejl.

3.1 Import

For at hente en af de 4 ovenstående pakker ind i et python script benytter vi os af `import` nøgleordet, som fortæller python, at funktionerne skal hentes fra den pågældende pakke. Med `as` kan vi desuden give dem et andet (ofte kortere) navn, så man slipper for at skrive hele navnet hver gang man bruger pakken. Vi kan hente de fire pakker ved at skrive følgende i en celle: (i `scipy.optimize` specificerer vi her, at vi kun vil hente én funktion derfra, `curve_fit`)

```
import sympy as sp
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
```

Selvom det er fristende altid at hente alle pakkerne, så er det en fordel kun at hente de pakker ind, som man skal bruge. Dette vil både gøre koden hurtigere, fordi Python ikke skal hente et stort bibliotek af funktioner, for at man kan bruge en enkel funktion. Det er også en fordel, hvis andre personer skal læse koden, at man tydeligt kan se, hvilke pakker og funktioner, som skal bruges for at køre ens kode.

Nogle pakker bruger vi så ofte, at vi gerne forkorter deres navne, så vi slipper for at skrive det fulde navn mange gange i løbet af en enkel notebook. Det er her vi bruger `as` nøgleordet. Når I gør dette, så brug dog gerne den konventionelle forkortelse (altså den som vi har brugt i cellen overfor), så vil det nemlig være nemmere for jer selv, jeres medstuderende og jeres instruktorer at læse og forstå den kode, som I skriver.

3.2 Hvordan bruger man en pakke?

Når man har hentet sin pakke, kan man kalde de funktioner, som er i den. Hvis man har gjort som `curve_fit` ovenfor og kun importeret en enkelt funktion, kan man blot skrive `curve_fit(_input_)` i sin kode. Hvis man til gengæld har hentet en hel pakke, skal man også specificere pakken for at benytte en funktion. Man vil generelt skrive det som: `[forkortelse for pakke].[funktion]([variable]`. F.eks. giver `np.exp(2)` værdien af pakken np's eksponentialfunktion i punktet 2, altså e^2.

Tænk på det som et bibliotek af funktioner. Numpy er biblioteket og exp er en bog i biblioteket. For at Python kan finde en funktion, skal du altså først fortælle, hvilket bibliotek Python skal gå til og derefter hvilken bog/funktion skal findes. Nogle biblioteker har også sektioner. Fx bruger vi udelukkende afdelingen "pyplot" af matplotlib-biblioteket.

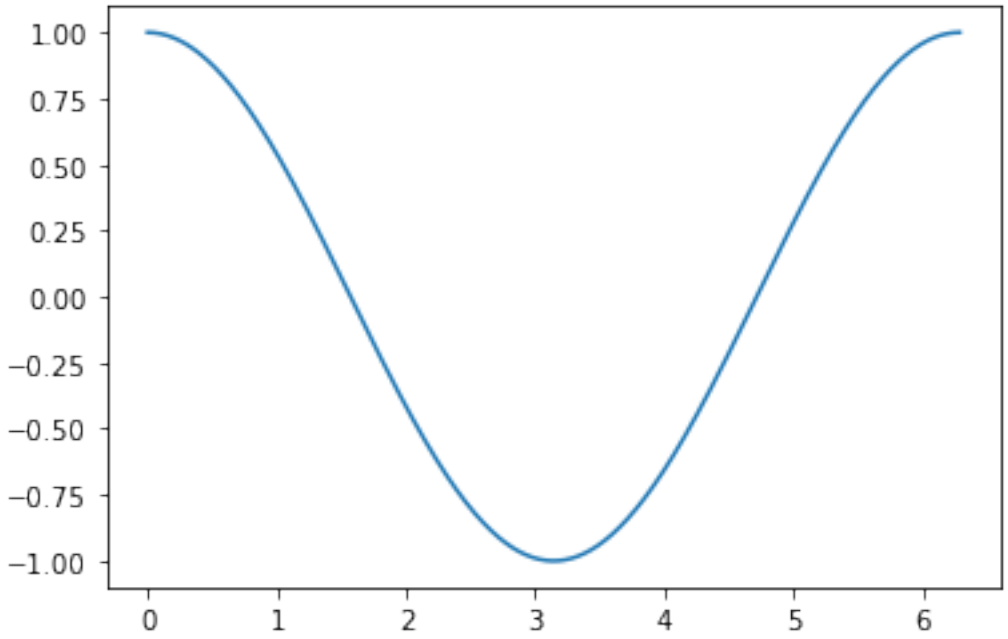
Herunder kan man se nogle få eksempler på funktionskald:

```
# Både sympy og numpy har de kendte trigonometriske funktioner samt værdien for pi
print(sp.cos(sp.pi))
print(np.cos(np.pi))
```

```
-1
-1.0
```

```
x = np.linspace(0, 2 * np.pi, 1001) # denne funktion laver en liste af tal startende
 ↪i 0, sluttende i 2*pi og med 1001 elementer
print(x)
y = np.cos(x)
plt.plot(x, y); # Plotter cosinus mellem 0 og 2 pi.
```

```
[0.         0.00628319 0.01256637 ... 6.27061894 6.27690212 6.28318531]
```



Flere eksempler kan ses herunder, hvor de mest brugte pakker: numpy, matplotlib og sympy er kort gennemgået. Dette er dog kun en oversigt, for at se en mere grundig gennemgang skal man finde noterne under enten "Python i Mekrel" eller "Python i LinAlys".

3.3 Numpy

Numpy er den pakke, som vi bruger til numeriske udregninger. Helt centralt er numpy arrays, som er lister, der tillader os at udføre matematiske operationer på alle listens elementer samtidig. Vi starter med at importere numpy:

```
import numpy as np
```

Og vi kan nu tage en liste af nogle tal og konvertere til et numpy array:

```
array_af_tal = np.array([1, 4, 9, 16, 25])
array_af_tal
```

```
array([ 1,  4,  9, 16, 25])
```

Hvis vi nu benytter en regneoperation, foretages operationen på alle elementerne:

```
array_af_tal + 5
```

```
array([ 6,  9, 14, 21, 30])
```

Derudover har numpy også en del indbyggede matematiske funktioner, som eksempelvis `cos`, `sin`, `sqrt` mm., som også kan bruges på enten et enkelt tal, eller på et helt array:

```
np.sqrt(36)
```

```
6.0
```

```
np.sqrt(array_af_tal)
```

```
array([1., 2., 3., 4., 5.])
```

Yderligere benyttes numpy ofte til at lave statistik på et helt array, eksempelvis gennemsnittet af overstående array:

```
np.mean(array_af_tal)
```

```
11.0
```

For en mere grundig gennemgang af numpy i MekRel-sammenhænge, se *siden om numpy i MekRel*.

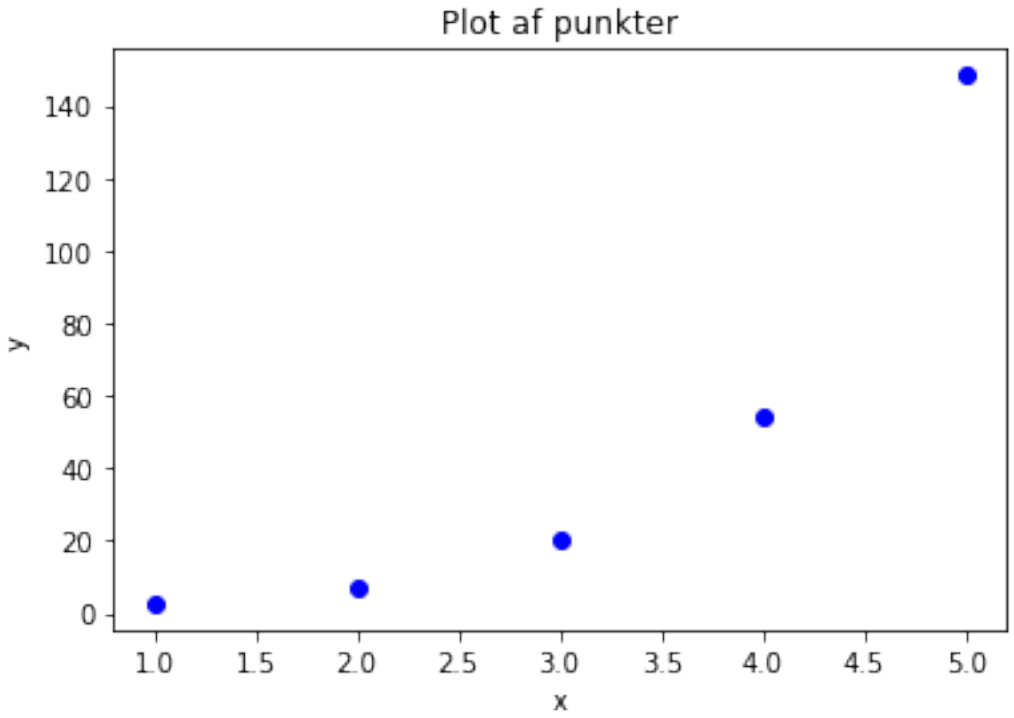
3.4 Matplotlib Pyplot

Matplotlib bruges til at plotte numeriske værdier, der f.eks. kunne være måledata fra laboratoriet. Vi importerer det ved:

```
import matplotlib.pyplot as plt
```

Herefter kan man nu benytte `plt.plot` til at plotte datapunkter:

```
x = np.array([1, 2, 3, 4, 5])
y = np.exp(x)

plt.plot(x, y, 'bo')
plt.title("Plot af punkter")
plt.xlabel("x")
plt.ylabel("y");
```



Matplotlib kan desuden bruges til rigtig mange forskellige plottyper, og det er et super vigtigt værktøj i laboratoriet. Se hvordan det ellers bruges på *siden om matplotlib i MekRel*.

3.5 SymPy

Når vi skal foretage symbolske udregninger såsom at udregne grænseværdier, foretage differentialregning eller løse ligninger benytter vi SymPy.

Når vi benytter SymPy starter vi oftest med at gøre Python opmærksom på, at vi ikke har med numeriske værdier at gøre, men i stedet noget mere abstrakt, som altså ikke skal afrundes eller udregnes, men lige præcis behandles som et symbol.

Først importerer man SymPy-biblioteket samt de symboler, som man gerne vil bruge fra `sympy.abc`

```
import sympy as sp                    # Importér biblioteket
from sympy.abc import x, a, b, phi    # Og de symboler som vi vil bruge. Vi kan altid
↪hente flere
```

Vi kan nu kombinere symbolerne til nye udtryk. Eksempelvis kan vi danne et udtryk ved at gange og dividerer vores symboler med hinanden:

```
a * b / x
```

$$\frac{ab}{x}$$

Eller vi kan sammensætte de variable til nye variable:

```
f = x * a * b
g = phi ** x

display(f)
display(g)
```

$$abx$$

$$\phi^{x}$$

Her benytter vi `display()` i stedet for `print()`, da dette viser symbolerne med matematisk formattering istedet for en tekst-version svarende til det, vi selv skrev ovenfor:

```
print(g)
```

```
phi**x
```

Vi kan nu kombinere de to udtryk med hinanden:

```
f * g ** (-f) - 1
```

$$abx\left(\phi^{x}\right)^{-abx} - 1$$

SymPy kommer nu også med en samling af funktioner, som vi kan bruge. Eksempelvis:

```
display(sp.cos(x))                      # Udtryk med cosinus
display(sp.exp(f) ** 2 + sp.sqrt(g)** 2) # Udtryk med ekspoentialfunktionen og
↪kvadratrødder
```

$$\cos(x)$$

$$\phi^x + e^{2abx}$$

Der, hvor vi virkelig får glæde af SymPy er, når vi benytte det til differentiering, integration og til eksempelvis at finde grænseværdier. Betragt f.eks. funktionen:$f(x) = \frac{x^2-4}{x+2} \cdot e^{-x}$

kan vi benytte sympy til at finde grænseværdien for $f(x)$ for $x \to 2$:

```
from sympy.abc import x
f = (x ** 2 - 4)/(x + 2)* sp.exp(-x)
sp.limit(f, x, 2)
```

$$0$$

Vi kan også bestemme den afledte (eller differenterede) funktion:

```
sp.diff(f, x)
```

$$\frac{2xe^{-x}}{x+2} - \frac{\left(x^2-4\right)e^{-x}}{x+2} - \frac{\left(x^2-4\right)e^{-x}}{\left(x+2\right)^2}$$

Eller det bestemte integral for $f(x)$ mellem grænserne a og b:

```
from sympy.abc import a, b
sp.integrate(f, (x, a, b))
```

$$-(1-a)e^{-a} + (1-b)e^{-b}$$

For en nøjere gennemgang af SymPy og de mange andre muligheder, som det tilbyder, referer vi til *noterne til python i LinAlys*.

CHAPTER

FOUR

PYTHON I MEKREL

Når I arbejder i laboratoriet i MekRel, bruges Python til at analysere jeres data og lave grafer til jeres rapport. Meget af programmeringen vil ske i laboratoriet, hvor I arbejder i grupper. Husk dog at bytte rundt på rollerne i gruppen, så alle får prøvet at kode i Python. Det er vigtigt, da I også skal lave fire individuelle Python afleveringer i løbet af kurset. Python skal også bruges i jeres næste fag, Data og Projekt, hvor I skal lære endnu mere om dataanalyse. Altså er det vigtigt at alle får prøvet kræfter og bliver fortrolig med Python tidligt.

Det er vigtigt at understrege at I **ikke** skal bruge Jupyter Notebooks som logbøger til labafleveringer. Logbøger eller rapporter skal ikke indeholde kode, kun resultatet af jeres kode (ex. plots og resultater), og kan med fordel skrives i Word, LaTeX el.lign. og **skal** afleveres i .pdf-format. Derimod **må** nogle af de fire såkaldte Pythonafleveringer, som afleveres individuelt, *godt* afleveres som Notebooks, men det hører I mere om senere.

Et typisk arbejdsmønster når Python bliver brugt i MekRel, er at importere data og behandle det med `numpy` operationer og lave et plot med `matplotlib` og derefter måske lave et fit med `scipy`. Det anbefales derfor, at du først læser `numpy`-noten og derefter `matplotlib`-noten. `Scipy`-noten er lidt mere teknisk og giver bedst mening senere.

4.1 Numpy

Numpy bruges til at lave numeriske beregninger og vil være jeres primære værktøj i laboratoriet. Det skyldes at man kan bruge `numpy arrays`. Det er en data type som gør operationer på mange tal hurtigt og forhåbenligt intuitivt når I har arbejdet lidt med det.

Tip: Hvis du gerne vil downloade og køre denne notebook skal du også bruge filen `Testdata.csv` som kan hentes her link til host

4.1.1 Numpy arrays

For at lave et numpy array kan man skrive forskellige ting. Her er nogle eksempler,

```
import numpy as np # Husk at importere numpy ellers virker det ikke.
```

```
array1 = np.array([3,4,5,1]) #manuelt indtastede værdier
array2 = np.arange(0,3,0.5) #et array med tallene fra 0 til 3 (ekslusiv 3) i 0.5␣
↪skridtstørrelse
array3 = np.linspace(0,8,5) #et array med tallene fra 0 til 8 (inklusiv 8) i 5 skridt

print(array1) #ønsker du at se noget af det du har defineret så "print" det således
print(array2)
print(array3)
```

```
[3 4 5 1]
[0.  0.5 1.  1.5 2.  2.5]
[0. 2. 4. 6. 8.]
```

Læg mærke til at NumPy funktionen `arange` tæller efter skridtstørrelse, altså hvor stort mellemrum der skal være mellem hver værdi i intervallet man angiver, og `linspace` tæller efter, hvor mange skridt der skal være i alt, og giver så en jævn fordeling af værdier i det interval, man angiver.

Vil man gerne trække tal ud af sine arrays, gøres det ved at skrive indekset på det man gerne vil have ud,

```
array4 = np.array([3,4,5,1,6,8]) #Definerer et array
print(array4)

print( array4[0], array4[2]) #vælger den første og tredje værdi i arrayet

print(array4[1:5]) #vælger værdien fra indeks 1 til og uden 5 (husk at Python␣
↪indekserer fra 0)

print(array4[-1]) #man kan også vælge den sidste værdi ved at tælle baglæns
```

```
[3 4 5 1 6 8]
3 5
[4 5 1 6]
8
```

Man kan også lave arrays i 2D, og trække værdier ud på samme måde,

```
array5 = np.array([[1,2,3],[4,5,6],[7,8,9]])
print(array5) #viser hele arrayet

print(array5[1,1]) #udskriver indeks (1,1)

print(array5[:,0]) #udskriver første kolonne i matricen

print(np.sum(array5)) #bestemmer summen af alle tallene
print(np.sum(array5,axis=0)) #bestemmer summen af kolonnerne
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
5
[1 4 7]
45
[12 15 18]
```

En af grundene til at vi bruger NumPy arrays er, at vi nemt kan bruge NumPy funktioner til at lave regneoperationer på arrays. Her følger et par eksempler,

```
tal = np.arange(0,10) #heltal fra 0 til 9

kvadrat = tal**2 + 1 #tager kvadratet plus 1 for alle tal i arrayet

Exp = np.exp(tal) #tager exp til alle tal i arrayet

#printer array, og de to resultater
print(tal)
```

(continues on next page)

(continued from previous page)

```
print(kvadrat)
print(Exp)
```

```
[0 1 2 3 4 5 6 7 8 9]
[ 1  2  5 10 17 26 37 50 65 82]
[1.00000000e+00 2.71828183e+00 7.38905610e+00 2.00855369e+01
 5.45981500e+01 1.48413159e+02 4.03428793e+02 1.09663316e+03
 2.98095799e+03 8.10308393e+03]
```

4.1.2 Funktioner til statistik

Når vi laver databehandling i lab, skal vi ofte bruge gennemsnit og spredning på vores data, og det kan vi gøre nemt med et par `numpy`-funktioner. Arbejder man med 2D-datasæt, f.eks. ved gentagne målinger for skiftende variable, hvor vi vil have gennemsnittet for hver opstilling, skal man være opmærksom på at indikere akse i datasættet. Aksen sættes enten til 0 eller 1, afhængig af om man vil tage gennemsnittet lodret gennem kolonner eller vandret gennem rækker.

```
data1 = np.array([3.2,4.1,3.7,4.3,3.3]) #indtaster et datasæt
gennemsnit1 = np.mean(data1) #tager gennemsnit
spredning1 = np.std(data1) #finder spredningen
print("Gennemsnit: ", gennemsnit1,"\nSpredning: ", spredning1) #\n sætter blot det␣
 ↪efterfølgende printet på ny linje

data2 = np.array([[1,2,3], [1.2,1.9,3.2], [0.8,2.1,3.4]]) #indtaster et 2D datasæt
gennemsnit2 = np.mean(data2, axis = 0) #vi ønsker at tage gennemsnittet gennem␣
 ↪kolonner.
spredning2 = np.std(data2, axis = 0)

print("Det her er et 2D-array, \n", data2)
print("Gennemsnit: ",gennemsnit2, "\nSpredning: ", spredning2)
```

```
Gennemsnit:  3.72
Spredning:  0.43081318457076023
Det her er et 2D-array,
 [[1.  2.  3. ]
 [1.2 1.9 3.2]
 [0.8 2.1 3.4]]
Gennemsnit:  [1.  2.  3.2]
Spredning:  [0.16329932 0.08164966 0.16329932]
```

4.1.3 Importering af datafiler

Når vi arbejder i lab har vi ofte et måleinstrument der tager målingerne, så det både er mere præcist og man kan tage flere målinger hurtigt. Men når man gør det får man store mængder data og det gider vi ikke sidde og taste ind manuelt. Derfor vil vi bruge Python til det i stedet. For at gøre dette bruger vi igen NumPy pakken. Vi skal bruge kommandoen np.genfromtxt, syntaxen af denne er;

```
np.genfromtxt(fname, delimiter = None, skip_header = 0, skip_footer = 0)
```

hvor fname er filnavnet, delimiter er hvordan dataen er opdelt i filen (mere om det om lidt), skip_header er hvor mange linjer af toppen af filen der skal springes over og skip_footer er det samme bare fra bunden af. Dette er en forsimplet syntax. Den hele findes her hvis du gerne vil læse lidt mere np.genfromtxt syntax. Den måde '=0' skal forstås er at hvis du ikke skriver skip_header='noget' så antager Python at du mener 0.

Tip: Det er vigtigt at datafilen som hedder fname ligger i den samme mappe som koden. Så sørg for at gemme jeres Notebook og datafil i samme mappe.

Her følger et eksempel, med datafilen "Testdata.csv" som kan hentes her.

```
data = np.genfromtxt('Testdata.csv', delimiter = ';', skip_header=2) #vores data er␣
↪adskilt med semi-kolon, og vi springer 2 linjer over


print(data)
```

```
[[ 1.2  2.5]
 [ 5.  10.3]
 [ 7.5 14.8]]
```

Inden man importerer datafilen er det vigtigt at kigge lidt på filen først. Det gør I ved at åbne jeres datafil i jeres foretrukne filredigeringsprogram. Hent den fil der hedder Testdata.csv som ligger her, for at kunne kigge lidt på den og forstå importeringen. Hvis du åbner Testdata.csv, så kan du se at der er noget tekst allerøverst, derfor sætter vi skip_header=2, da der er to linjer af tekst vi ikke vil have importeret i vores kode. Videre ser vi at tallene er adskilt med semikolon. Det betyder at vi skal sætte delimiter = ';'. Udover det ser vi at vi får et todimensionelt array af data. Vi vil i dette tilfælde helst have kolonnerne hver for sig. En kolonne kan importeres således,

```
kolonne1 = data[:,0]
print(kolonne1)
```

```
[1.2 5.  7.5]
```

Nu bør du kunne hente data fra en fil og begynde at lave små data behandlinger som at tage gennemsnittet af alle kolonnerne, under ses et eksempel på dette.

```
data = np.genfromtxt('Testdata.csv', delimiter = ';', skip_header=2)
print(data)

Gns_Row = np.mean(data, axis = 0)
print(Gns_Row)
```

```
[[ 1.2  2.5]
 [ 5.  10.3]
 [ 7.5 14.8]]
[4.56666667 9.2       ]
```

4.2 Matplotlib

Vi vil ofte i lab gerne formidle vores data i form af en figur. For at lave figurer bruger vi pakken Matplotlib.

Matplotlib er et kæmpe bibliotek og kan rigtig rigtig meget. Det er faktisk så stort at du kun skal bruge en lille del af det `pyplot`. Derfor skriver man `matplotlib.pyplot`, det svarer lidt til kun at læse bøgerne i fysikafdelingen af et bibliotek.

Man skal også passe på, når man laver plots med matplotlib. Det mixer dårligt med `Sympy`, da matplotlib laver plots ud fra tal og ikke analytiske udtryk, som er det Sympy giver. Hvis man skal lave et plot med Sympy så kig i noten om Sympy.

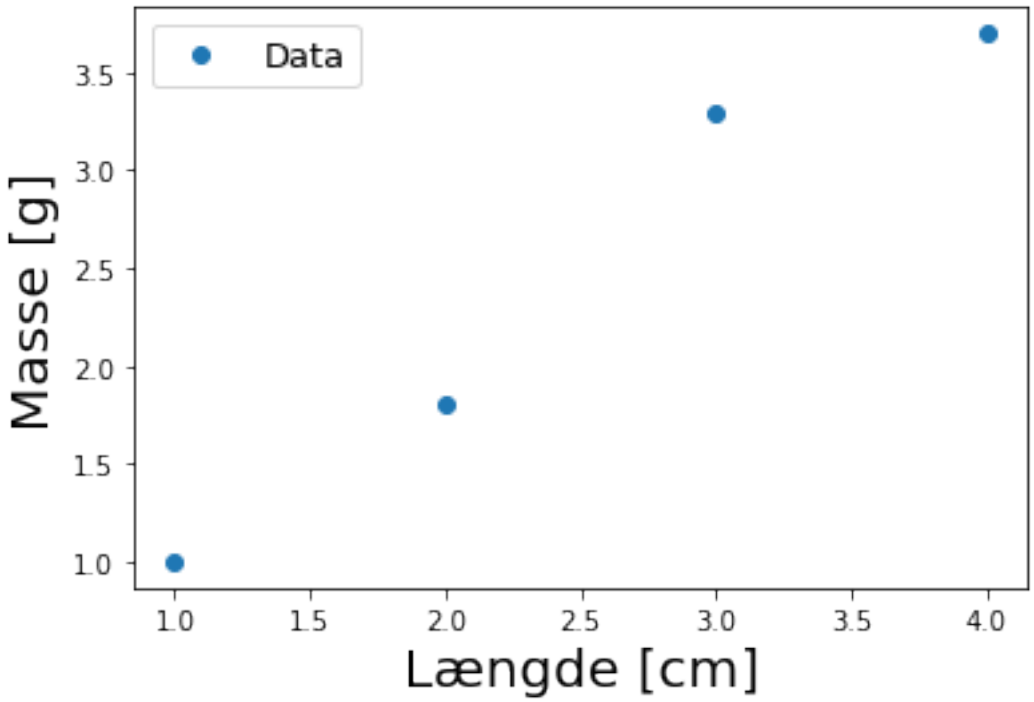
4.2.1 Plots

For at lave et plot skal vi først have data at arbejde med.

```
import numpy as np
xData = np.array([1,2,3,4])
yData = np.array([1,1.8,3.3,3.7])
```

Når man laver et plot med Matplotlib bygger man det op trinvis. Først laves plottet ud af data med `plt.plot`. Derefter bygges aksenavne og en legend på. Et eksempel ses her,

```
import matplotlib.pyplot as plt

plt.plot(xData, yData, 'o', label = "Data") #Vi plotter xData mod yData som store␣
↪prikker, og giver datasættet en titel

plt.xlabel('Længde [cm]', fontsize = 20) #Tilføjer etiketter på akserne og angiver␣
↪skriftstørrelse
plt.ylabel('Masse [g]', fontsize = 20)

plt.legend(fontsize = 13) #plotter en "legend", navnet hentes fra label i plt.plot

plt.savefig("figure.png") #gemmer figuren som en .png fil

plt.show() #viser figuren under den kørte celle
```

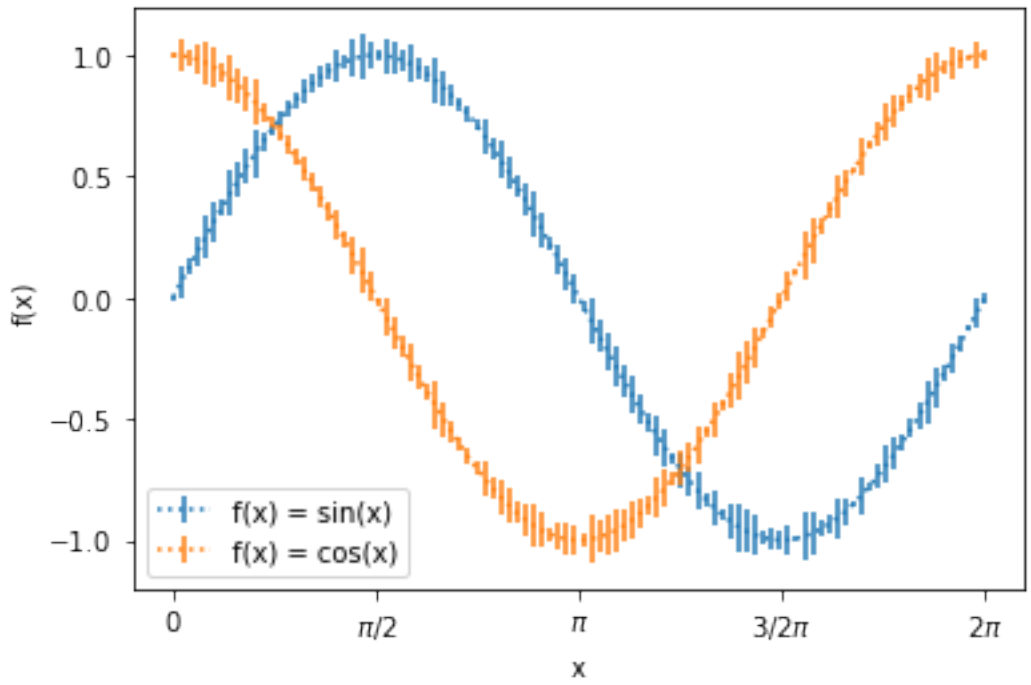


I behøver ikke nødvendigvis manuelt indstille skriftstørrelsen på akser og legend hver gang, men det kan nogle gange være nødvendigt at justere, afhængig af hvor meget figurerne skal fylde i jeres logbog. Regelen er at man altid skal kunne læse akse-titlerne uden at zoome ind eller hive forstørrelsesglas frem. Vær også opmærksom på, at når I bruger funktionen plt.savefig(), gemmes jeres figur i samme mappe som jeres Notebook ligger, medmindre I indstiller det anderledes.

Det kan også være smart at ændre hvad der er på akserne, et eksempel med radianer ses under. Der er også blevet sat errorbars på data. Det gøres ved at plotte med `plt.errorbar` i stedet for `plt.plot`.

```
#flere eksempler på at plotte

xVærdi = np.linspace(0,2*np.pi,100) #laver et array af værdierne fra 0 til pi,␣
↪fordelt over 100 datapunkter

#bruger NumPy funktioner til at plotte sinus og cosinus til vores x værdier, og␣
↪indikerer at punkter skal være en
#prikket linje

yerr = np.random.uniform(0.01,0.1,(100,1))
plt.errorbar(xVærdi, np.sin(xVærdi), yerr, fmt = ':', label = "f(x) = sin(x)")
plt.errorbar(xVærdi, np.cos(xVærdi), yerr, fmt = ':', label = "f(x) = cos(x)")

plt.xlabel('x') #sætter titler på akserne
plt.ylabel('f(x)')

# Følgende skridt er lidt unødvendigt, men plotter man ex. trigonometriske funktioner␣
↪kan det være fedt at indstille
# aksen så værdierne "giver mening" ift. funktionen.
# Her indikerer første array hvor på x-aksen man vil have "ticks", og dernæst et␣
↪array med hvad der skal stå på hvert
# "tick"
plt.xticks([0,np.pi/2,np.pi,3*np.pi/2,2*np.pi], ['0','$\pi/2$','$\pi$','$3/2\pi$','$2\
↪pi$'])

plt.legend()

plt.show()
```

```
<ipython-input-9-d5d355b23b32>:9: MatplotlibDeprecationWarning: Support for passing a␣
↪(n, 1)-shaped error array to errorbar() is deprecated since Matplotlib 3.1 and will␣
↪be removed in 3.3; pass a 1D array instead.
  plt.errorbar(xVærdi, np.sin(xVærdi), yerr, fmt = ':', label = "f(x) = sin(x)")
<ipython-input-9-d5d355b23b32>:10: MatplotlibDeprecationWarning: Support for passing␣
↪a (n, 1)-shaped error array to errorbar() is deprecated since Matplotlib 3.1 and␣
↪will be removed in 3.3; pass a 1D array instead.
  plt.errorbar(xVærdi, np.cos(xVærdi), yerr, fmt = ':', label = "f(x) = cos(x)")
```



Dette er sådan vi forventer at jeres plots ser ud nå i aflevere dem i laboratoriet.

Der er en masse forskelige andre ting man kan med at plotte men dette bør være nok til det I skal her.

4.3 Funktioner, fits og usikkerheder i Python

I denne note vil vi introducere jer til funktioner, hvordan man fitter en model til data og behandler usikkerheder i Python.

4.3.1 Funktioner

Før vi kan fitte en funktion med Python skal vi kunne definere en funktion. En funktion er et meget bredt begreb i Python, og man kan rigtig meget med dem. Fordelen ved funktioner er, at man kan samle kode, som man bruger mange gange, så det kan gøre større projekter meget overskuelige. På nuværende tidspunkt er den primære brug af funktioner dog, at vi skal bruge dem til at fitte.

Man definerer en funktion ved at skrive

```
def FUNKTION_NAVN(INPUTS):
    __Udregninger__
    return OUTPUT
```

Så en funktion bliver nu primært defineret ved input og outputtet af denn, hvor der så ligger en del udregninger imellem. Et eksempel kan ses nedenfor:

```
def E_kin(m,v):
    return 1/2 * m * v**2
```

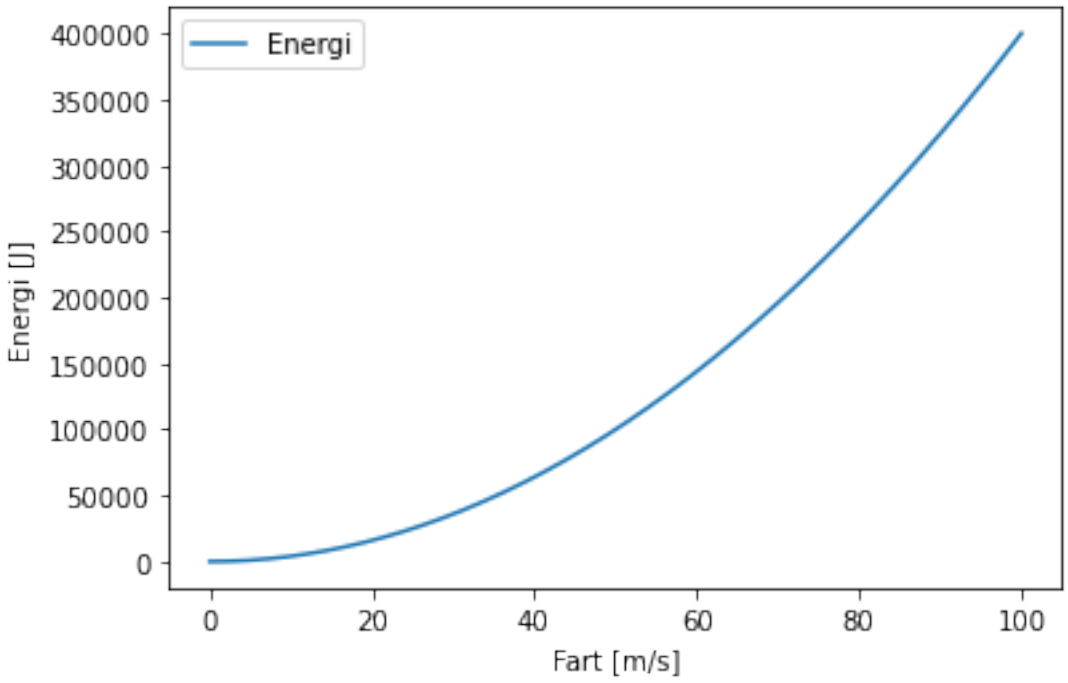
Her har jeg defineret en funktion som jeg kalder E_kin, den tager 2 argumenter, m og v, og returnerer $\frac{1}{2}mv^2$. Den kan vi så prøve af for m = 80 og v = 5

```
print(E_kin(80,5))
```

```
1000.0
```

Funktioner kan også bruges til at gøre kode mere overskuelig og nemmere at debugge. Her er der fx defineret en funktion som laver et plot. Den tager ingen værdier og retunerer heller ikke noget, men er nem at debugge og man kan lade være med at kalde den, når man har fået sit plot.

```
import numpy as np

v_data = np.linspace(0,100,1000)
E_data = E_kin(80,v_data)

def plot_E_kin():
    import matplotlib.pyplot as plt
    plt.plot(v_data,E_data, label = 'Energi')
    plt.xlabel('Fart [m/s]')
    plt.ylabel('Energi [J]')
    plt.legend()
    plt.show()
    plt.close('all')

plot_E_kin()
```



4.3.2 Fitting

Vi skal nu bruge funktioner til at fitte en model til data. Her bruger vi optimize.curve_fit. Denne kommer fra pakken SciPy og importeres således:

```
from scipy.optimize import curve_fit
```

For at fitte en model skal man have noget data at fitte til. Her bruger jeg den data der lå på jeres første Python aflevering.

```
data = np.array([
    [0.19,1.98,2.05,2.16,2.16,2.07],
    [0.3,2.61,2.6,2.58,2.81,2.68],
    [0.36,3,3,3,2.87,2.97],
    [0.53,3.47,3.83,3.54,3.6,3.5],])
print(data)
```

```
[[0.19 1.98 2.05 2.16 2.16 2.07]
 [0.3  2.61 2.6  2.58 2.81 2.68]
 [0.36 3.   3.   3.   2.87 2.97]
 [0.53 3.47 3.83 3.54 3.6  3.5 ]]
```

For at fitte en model til data, skal man have ét datapunkt for hver varieret måling. Her har vi 5 målinger per pendullængde, derfor skal vi altså tage gennemsnit af vores svingningstider.

```
L = data[: , 0] #udvælger kolonnen med pendullængde
svingningstider = data[:,1:5] #her udvælger vi de fire kolonner  med svingningstider

gns_svingning = np.mean(svingningstider, axis = 1) #tager gennemsnit af␣
 ↪svingningstiderne, "axis" angiver om vi går vandret eller lodret i datasættet

print(L)
print(gns_svingning)
```

```
[0.19 0.3  0.36 0.53]
[2.0875 2.65   2.9675 3.61  ]
```

Nu har vi to lister med datapunkter, der hænger sammen, og vi kan begynde at undersøge hvilket fit vi skal bruge. Vi regner med at skulle bruge en lineær funktion, så vi skal have defineret det i vores kode. Vi kalder funktionen "linfunc", og angiver så variabel og parametre i parentes. I return skriver vi hvad funktionen skal give som resultat når vi bruger den.

```
def linfunc(x, a, b):
    y = a*x + b
    return y
```

Når man fitter er det meget **vigtigt** at det første argument i ens funktion, her x, er det som man har data på og vil have Python til at fitte efter, ellers fungerer fittet ikke. Dette er en fejl som ofte opstår, så hvis I får en fejl så tjek lige om i har det rigtige stående som det første I jeres parentes.

Når man så har en funktion defineret rigtigt, kan man fitte vores funktion til dataen med kommandoen `curve_fit()`. Funktionen `curve_fit` skal bruge funktionen, som vi ønsker at fitte, samt de tilhørende x- og y-værdier, som i dette tilfælde er pendullængde og svingningstid. Funktionen giver to outputs, først giver den fitte-parametrene, det vil sige fittets bedste bud på, hvad parametrene i fit-funktionen bør være. Dernæst giver den covariansen. Er der flere parametre angives covariansen som en matrix med flere værdier. Covariance-matrixen kan bruges til at finde ussikkerheder på fit parametre, men mere om det til sidst.

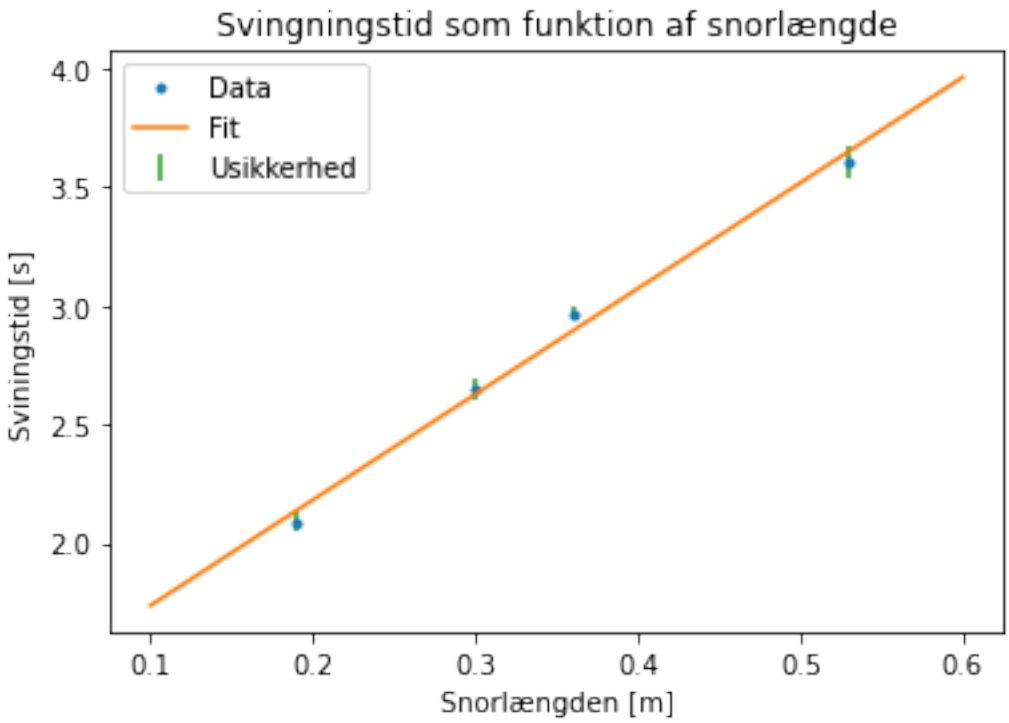
```
par, cov = curve_fit(linfunc, L , gns_svingning)
print(par) #printer parametrene
print(cov) #printer covariansen
```

```
[4.4553719  1.29164669]
[[ 0.08314673 -0.02868562]
 [-0.02868562  0.01115413]]
```

Nu kan vi plotte vores data sammen med vores fit og vurderer, hvor god vores model er. Vi skal selvfølgelig også have plottet vores usikkerheder, og da vi arbejder med gennemsnit vil vi plotte usikkerheden ved hjælp af standardafvigelsen på vores data. Man kan bruge NumPy funktionen np.std() til at bestemme standardafvigelse.

For at plotte vores fit laver vi et array af værdier i det interval, som vi gerne vil plotte med np.linspace funktionen. Vi bruger så disse værdier i vores lineære funktion, hvor vi bruger de parametre vores fit bestemte med `curve_fit`.

```
import matplotlib.pyplot as plt

afvigelse = np.std(svingningstider, axis = 1)/np.sqrt(len(svingningstider))

X = np.linspace(0.1,0.6,1000) #laver et array af 1000 jævnt fordelte tal mellem 0.1␣
 ↪og 0.6
Y = linfunc(X,par[0],par[1])  #laver et array af vores lineære fit på det ovenstående␣
 ↪array

plt.plot(L , gns_svingning, '.', label = 'Data')
plt.plot(X,Y , label = 'Fit')
plt.errorbar(L , gns_svingning , yerr=afvigelse , fmt='none' , label = 'Usikkerhed')
plt.xlabel('Snorlængden [m]')
plt.ylabel('Sviningstid [s]')
plt.title("Svingningstid som funktion af snorlængde")
plt.legend()
plt.show()
```



Man kan også lave et fit, hvor der tages højde for usikkerheden på datapunkterne. Det gør man ved at angive et sigma når man bruger curve_fit.

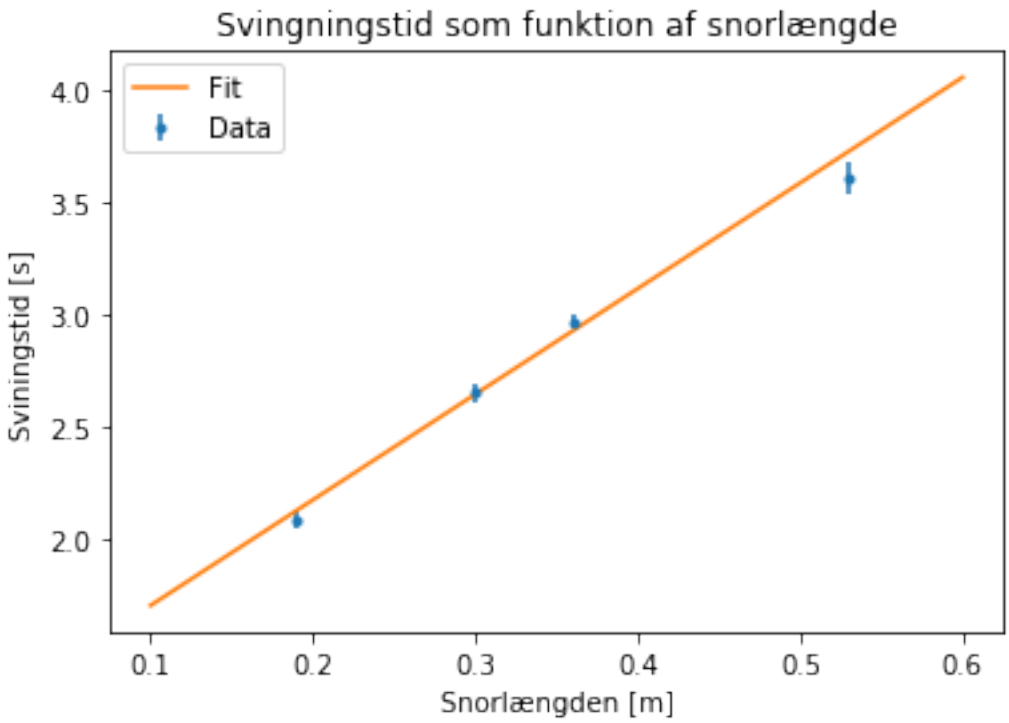
Tip: Husk at bruge `absolute_sigma = True` ellers for i problemer med jeres ussikerheder.

```
par_with_error, cov_new = curve_fit(linfunc, L , gns_svingning, sigma = afvigelse,␣
↪absolute_sigma = True)
print(par_with_error)
```

```
[4.71061446 1.23176413]
```

Paremeterne er ikke ændret så meget da fejlen på punkterne er meget ens, men hvis der havde været stor forskel på usikkerhederne, ville det have en større effekt. Endeligt kan man så plotte punkter med usikkerheder og fit som tager højde for usikkerheden.

```
Y_2 = linfunc(X,par_with_error[0],par_with_error[1])

plt.errorbar(L , gns_svingning,yerr= afvigelse, fmt='.', label = 'Data')
plt.plot(X,Y_2 , label = 'Fit')
plt.xlabel('Snorlængden [m]')
plt.ylabel('Sviningstid [s]')
plt.legend()
plt.title("Svingningstid som funktion af snorlængde")
plt.show()
```



For at finde usikkerheden på de fittede parametre, kan man bruge covariance-matrixen. På dens diagonal er nemlig variansen for de fittede parametre. Når man har variansen kan man finde usikkerheden, da kvadratroden af variansen er ussikerheden. I Python kan det gøres med et for-loop.

```
for i in range(len(cov_new)):
    sigma = np.sqrt(cov_new[i,i])
    print(par[i], ' +/-' , sigma)
```

```
4.455371899695233  +/- 0.3624954078593303
1.2916466946051461  +/- 0.120762382852016
```

Her er fit-værdierne for a og b printet med usikkerheder fundet fra covariance-matrixen.

CHAPTER

FIVE

PYTHON I LINALYS

LinALys-kurset indeholder to spor (analyse og lineær algebra) og har fokus på "blyantsregning" og forståelse af den bagvedliggende teori. Ind imellem får vi brug for et værktøj til at tegne funktioner og foretage både numeriske og symbolske beregninger for at illustrere en problemstilling og/eller støtte eller kontrollere beregningerne i hånden. I dette kursus har vi valgt at bruge Python-modulet SymPy som CAS-værktøj (Computer Algebra System). Python og SymPy er ikke nødvendigvis det optimale værktøj til alle de opgaver, vi løser med Python i kurset, men vi har valgt Python fordi det er standardprogrammeringssproget på fysikstudiets førstedel, så de kræfter, du kommer til at bruge på Python, får du forhåbentlig gavn af andre steder (f.eks. i MekRel-kurset). På denne side har vi samlet en række af forskellige noter, som forhåbentlig kan introducerer alt det Sympy, som I kan få brug for i LinAlys kurset.

Det er vigtigt at være opmærksom på, at modsat CAS-værktøjer som Maple, TI_Nspire og WordMat, så er Python ikke lavet specifikt til symbolske udregninger, men er et generelt anvendeligt programmeringssprog. Man skal derfor vænne sig til både Pythons syntaks og programmering som helhed for at få det maksimale ud af SymPy. Når det til gengæld er klaret, er det muligt at benytte SymPy i en lang række sammenhænge, og f.eks. nemt skifte mellem numeriske og symbolske anvendelser. Dette vil vi vende tilbage til, når vi har forudsætningerne på plads.

5.1 Hvad er SymPy?

I denne sektion vil vi kigge på, hvordan vi benytter SymPy. Selve SymPy er en pakke i Python, der lader os arbejde med symbolske udtryk. Vi starter med at importere SymPy på samme måde. Derfor vil vi starte alle notebooks med at importere SymPy og benytte forkortelsen "sp" for pakken: import sympy as sp.SymPy har flere indbyggeret printeværktøjer, der gør det muligt at vise symbolske udtryk i LaTeX, som er et *typesetting*-system der er særligt velegnet til matematisk notation (du kender måske LaTeX fra Words Wordmat-plugin). Den normale print() i Python skriver output i "string"-format, mens funktionen display() (som er indbygget i Jupyter) vil være vores foretrukne valg til at vise SymPy-udtryk som LaTeX. Når vi skriver et udtryk til sidst i en celle, vil Jupyter oftest skrive resultatet og vil automatisk bruge `display`.

```
import sympy as sp                    # Importer SymPy
```

5.1.1 Symboler og tal

På samme måde som vi kan have variable i Python som *strings*, logiske booleans eller talværdier, så tilføjer SymPy muligheden for symbolske variable, svarende til hvad vi f.eks. kalder en ubekendt i ligningsløsning eller den (uafhængige) variabel i en funktion. Vi ønsker således at have muligheden for angive en variabel som x i $f(x) = x^2 + 3x - 2$ og opfatte den som et abstrakt objekt i modsætning til at tildele x en værdi. Den nemmeste måde at definere symboler på er at importere dem fra underbiblioteket sympy.abc, som indeholder de fleste symboler, som vi til dagligt bruger. Vi kan definere f.eks. x, a, b og ϕ som symboler ved at skrive:

```
from sympy.abc import x, a, b, phi
```

Nu kan vi benytte disse variable i beregninger:

```
a + b + phi # Vi kan lægge dem sammen
```

$$a + b + \phi$$

Vi kan også danne nye udtryk med symbolske værdier. Udtrykkene kan sættes sammen ved at benytte normale Python-operationer såsom: +, -, *,/ eller **. Derudover kan vi benytte en del andre regneoperationer ved at skrive sp. foran operationen. Her er samlet de typiske regneoperationer, som man kan finde i SymPy

- Kvadratrødder: sp.sqrt(x) (benyt `sp.root(x, n)` til at beregne $\sqrt[n]{x}$, altså den n'te rod af x)
- Trigonometriske funktioner: sp.cos(x), sp.sin(x), sp.tan(x). De inverse trigonometriske funktioner findes ved eksempelvis acos(x)
- Exponentialfunktion sp.exp(x)
- Logaritmer: sp.log(x). For at få 10-tals-logaritmefunktionen skrives log(x, 10)

En mere omfattende liste over regneoperationer kan findes her.

Så eksempelvis kan vi sammensætte et udtryk ved at benytte exponentialfunktionen sp.exp sammen med vores symboler:

```
a * sp.exp(2*x + phi)
```

$$ae^{\phi+2x}$$

Vi kan også danne nye symbolske variable ud fra eksisterende variable. Vi kan f.eks. definere en funktion f baseret på tal og eksisterende symbolske variable:

```
f = a * sp.exp(2*x + phi)
```

Bemærk at vi ikke behøver at definere f som symbolsk variabel. Python/SymPy kan godt regne ud at f bliver en symbolsk variabel idet den er opbygget af andre symbolske variable. Resultatet kan vises med display()-funktionen:

```
display(f)
```

$$ae^{\phi+2x}$$

5.1.2 Eksakt repræsentation af tal

Python opfatter / som en numerisk operation, og når vi vil have eksakte tal-brøker, må vi eksplicit bede SymPy om at opfatte dem som sådan ved hjælp af Rational(a, b)

```
brøk = sp.Rational(1, 3)
display(brøk)
```

$$\frac{1}{3}$$

En brøk bestående bestående af symboler lider ikke under samme problem, så der kan vi bare bruge almindelig division.

```
c = a/b
display(c)
```

$$\frac{a}{b}$$

Vi kan regne med brøkerne ved hjælp af de almindelige regnearter:

```
p = sp.Rational(1, 3)
q = sp.Rational(2, 5)
display(p - q)

p = sp.Rational(1, 3)
q = sp.Rational(2, 5)
display(p*a/b - q)
```

$$-\frac{1}{15}$$

$$\frac{a}{3b} - \frac{2}{5}$$

Vi får ofte brug for eksakte værdier af π og evt. andre særlige tal. En eksakt værdi af π får vi ved at skrive from sympy import pi. Når vi sammensætter symbolske variable og π med tal i brøker, kan vi som nævnt ovenfor godt bruge almindelige division istedet for sp.Rational(tæller, nævner), da Python på grund af symbolerne ikke kan behandle udtrykkene som numerisk repræsenterede decimaltal (*floats*):

```
from sympy import pi
value = sp.sqrt(3) * pi / 2
display(value)
```

$$\frac{\sqrt{3}\pi}{2}$$

Desuden kan vi hente nogle andre brugbare symboler fra SymPy, som eksempelvis uendelig, oo (der skrives som to små o'er og ligner et uendelighedstegn, hvis man har lidt fantasi).

```
from sympy import oo # Importer værdien uendelig
1 / oo               # 1 divideret med uendelig giver 0, i hvert tilfælde for␣
↪fysikere :o)
```

$$0$$

På samme måde kan vi også importere den imaginære enhed $i = \sqrt{-1}$, som i SymPy er angivet ved et stort I. Hvis man hellere vil lave numeriske beregninger med komplekse tal, benytter man i stedet j og kan f.eks. skrive 2 + 3j. *Hvis du ikke kender til de komplekse tal, så gå ikke i panik. De bliver introduceret i slutningen af blok 1*

```
from sympy import I # Importer I
z = 3 + 3 * I       # definer to tal
q = 1 - 2 * I
display(z, q)       # Vi viser dem med display()
```

$$3 + 3i$$

$$1 - 2i$$

Vi vil vende tilbage til imaginære tal i SymPy i en senere notebook.

5.1.3 Evaluer udtryk

Forestil dig at vi har et symbolsk udtryk, der indeholder den variable a, og vi ønsker at indsætte værdien $a = 2$. Til dette vil vi nu indføre de to metoder .subs() og .evalf(). For et udtryk f, som eksempelvis kunne være f = a ** 2 + b, kan vi indsætte $a = 2$ ved at skrive f.subs(a, 2).

Symbolsk substituering:

Vi kunne for eksempel for et funktion $f(x) = cos(x \cdot \pi/4)$ ønske at finde værdien af f for forskellige værdier af x:

```
# Vi definerer funktionen f:
f = sp.cos(x * pi / 4)
display(f)

# Vi kan nu finde værdien for x = 1 ved at skrive:
display(f.subs(x, 1))
```

$$\cos\left(\frac{\pi x}{4}\right)$$

$$\frac{\sqrt{2}}{2}$$

Vi kan også indsætte et udtryk ind i et andet:

```
from sympy.abc import x, a, b, c     # Vi har allerede x, a, og b til rådighed fra␣
↪ovenfor, men skal have c med.
f = a * b * x + x
display(f)

g = b ** 2 + c ** 2
display(g)

f.subs(x, g)
```

$$abx + x$$

$$b^2 + c^2$$

$$ab\left(b^2 + c^2\right) + b^2 + c^2$$

Her har vi altså defineret to udtryk og erstattet alle forekomster af x i det første udtryk med $g = a^2 + b^2$. Vi kunne have opået det samme (nemmere, men mindre generelt) ved at skrive:

```
g = b ** 2 + c ** 2
f = a * b * g + g

display(f)
```

$$ab\left(b^2 + c^2\right) + b^2 + c^2$$

Numerisk evaluering:

Når vi har et matematisk udtryk, er det en fordel også at kunne finde en numerisk approksimation. Til at gøre dette benytter vi metoden .evalf(cifre), som giver en approksimation med det angivne antal cifre. Det simpleste eksempel er, hvis vi ønsker at finde det første 10 cifre af pi:

```
pi.evalf(10)          # evalf(10), giver os de første 10 cifre
```

$$3.141592654$$

Vi kan benytte de to metoder i kombination, hvis vi eksempelvis vil finde værdien af et udtryk og så approksimere det.

```
f = sp.exp(x / 5)
display(f)

f_3 = f.subs(x, 3) # Substituerer x med 3 og gemmer det i en variabel, der hedder f_3.
# Det er sådan vi gerne vil have svaret skrevet i opgaverne i LinALys: så simpelt som␣
 ↪muligt men stadig eksakt.
display(f_3)

f_3.evalf(3) # Lad os finde værdien som decimaltal med 3 cifre.
# ... hvilket er praktisk f.eks. hvis vi skulle tegne resultatet ind i en illustration
```

$$e^{\frac{x}{5}}$$

$$e^{\frac{3}{5}}$$

$$1.82$$

5.2 Grænser

En anden anvendelse af symbolske udtryk er beregning af grænseværdier. Fokus i kurset er på beregning af grænseværdier med papir og blyant, men det er godt at kunne checke sine resultater eller lave mere avancerede beregninger med SymPy. Dette gøres relativt nemt ved at benytte sp.limit-funktionen. Kaldesekvensen for funktionen er sp.limit(udtryk, variabel, grænse for variabel, retning). Efterlades retningen blank, beregnes grænseværdien oppefra / fra højre, og man får *ingen advarsel* selvom grænseværdierne fra henholdsvis højre og venstre er forskellige.

Hvis vi nu eksempelvis vil beregne $\lim_{x \to 0} e^{-x}$ skriver vi:

```
import sympy as sp              # Hent sympy
from sympy.abc import x         # Vi vil bruge `x`
from sympy import oo            # Så kan vi tage grænserne i uendelig
```

```
expr = sp.exp(-x)                       # Definer udtryk
display(sp.limit(expr, x, 0, '+'))      # Beregn og vis grænseværdien af udtrykket for x␣
 ↪gående mod 0 oppefra ...
display(sp.limit(expr, x, 0, '-'))      # ... og nedenfra
```

$$1$$

$$1$$

hvilket ikke er den store overraskelse, eftersom e^{-x} er defineret i $x = 0$, er kontinuert, og $e^0 = 1$.

Vi kan også beregne grænseværdier for $x \to \infty$:

```
sp.limit(expr, x, oo)     # grænsen af udtrykket for x gående mod uendelig. Vi angiver␣
 ↪ingen retning.
```

$$0$$

Vær særligt opmærksom på situationer, hvor grænseværdien kunne være forskellig oppefra og nedefra, hvilket oftest forkommer for udtryk på brøkform i det/de x-værdier, hvor nævneren antager værdien nul. Betragt f.eks. $\dfrac{x^4 + x^2 + 1}{3x^3 - 19x^2 - x}$ når $x \to 0$:

```
# Definer tæller og nævner hver for sig
poly1 = x ** 4 + x ** 2 + 1
poly2 = 3* x ** 3 - 19 * x **2  - x

# Kombiner dem til det ønskede udtryk og udskriv udtrykket til skærmen
poly_div = poly1 / poly2
display(poly_div)

# Tag grænsen af udtrykket, når x går mod 0 først oppefra / fra højre
sp.limit(poly_div, x, 0, '+')
```

$$\frac{x^4 + x^2 + 1}{3x^3 - 19x^2 - x}$$

$$-\infty$$

```
# ... og dernæst nedefra / fra venstre:
sp.limit(poly_div, x, 0, '-')
```

$$\infty$$

5.3 Ligninger

Første trin for at løse ligninger er at få skrevet ligninger op i et sprog, Python kan forstå. Det afgørende er her at indse at et lighedstegn kan have flere fundamentalt forskellige betydninger. Tidligere har vi tildelt variable bestemte værdier ved f.eks. at skrive k = 4, mens vi her vil bruge lighedstegnet til at beskrive et udsagn om sammenhængen mellem to udtryk. I SymPy-sprog er dette en *equality* og syntaksen er (når vi har importeret SymPy som sp som vi plejer) givet ved sp.Eq(venstre side, højre side). Pythagoras' læresætning kan f.eks. opskrives som:

```
import sympy as sp
from sympy.abc import a, b, c               # Vi definerer a, b og c som symbolske␣
↪variable

Pytha = sp.Eq(a ** 2 + b ** 2, c ** 2)      # Syntaks: sp.Eq(venstre side, højre side)
display(Pytha)
```

$$a^2 + b^2 = c^2$$

Man kan ligeledes danne ligninger ved at sammensætte allerede definerede udtryk, her eksemplificeret ved cosinusrelationen:

```
from sympy.abc import theta

expr = a**2 + b**2 - 2*a*b*sp.cos(theta) # Vi laver nu blot den ene side af ligningen␣
↪som et udtryk (expr for "expression")
display(expr)

cos_relation = sp.Eq(c**2, expr)          # Og vi kan nu sammensætte højre- og␣
↪venstresiderne
display(cos_relation)
```

$$a^2 - 2ab\cos(\theta) + b^2$$

$$c^2 = a^2 - 2ab\cos(\theta) + b^2$$

Vi kunne selvfølgelig også have gjort det hele i ét hug:

```
cos_relation = sp.Eq(c**2, a**2 + b**2 - 2*a*b*sp.cos(theta))
display(cos_relation)
```

$$c^2 = a^2 - 2ab\cos(\theta) + b^2$$

Nu når vi har lighederne på plads, er det blevet tid til at lade SymPy regne for os. SymPy giver os to forskellige værktøjer til at løse ligninger, og det er lidt forskelligt, hvad de hver især er gode til.

5.3.1 Solveset

Den første metode hedder solveset(), som kan oversættes til "løsningsmængde". Dette er den relativ ny metode, og SymPy-teamet arbejder på at denne skal være den primære løsningsmetode i fremtiden. Den løser dog ikke alle opgaver godt endnu, hvorfor vi nedenfor vil præsentere et alternativ.

Lad os prøve at løse en af overstående ligninger. Lad os tage helt standard Pythagoras $a^2 + b^2 = c^2$ (ligningen defineret som Pytha ovenfor). Hvis vi nu kender $c = 5$ og $a = 3$ og ønsker at finde b, indsætter vi først værdierne i ligningen:

```
display(Pytha)
Pytha_indsat = Pytha.subs([(a, 3), (c, 5)])
display(Pytha_indsat)
```

$$a^2 + b^2 = c^2$$

$$b^2 + 9 = 25$$

Vi benytter nu solveset() til at løse denne. For en ligning virker denne funktion ved, at man angiver ligningen og den variabel, som man ønsker at løse for:

```
solution = sp.solveset(Pytha_indsat, b)
display(solution)
```

$$\{-4, 4\}$$

Vi har altså nu fundet løsningerne til ligningen. I tilfældet med Pythagoras' sætning leder vi efter en sidelængde, så vores løsning skal være et positivt tal. Vi kan indskrænke løsningsdomænet ved at give solveset et "domain"-keyword. Dette skal være et såkaldt *SymPy-set*, som er en lidt indviklet størrelse, men i langt de fleste tilfælde kan man slippe afsted med at bede om reelle tal ved at skrive sp.Reals eller ved at give et interval med sp.Interval(*fra*, *til*), hvor man kan bruge oo for uendelig (forudsat at oo er blevet importeret). For mere avancerede løsningsdomæner henvises til dokumentationen her. I dette tilfælde angiver vi et interval:

```
from sympy import oo            # Vi importerer uendelig
pos_solution = sp.solveset(Pytha_indsat, b, sp.Interval(0, oo))
display(pos_solution)
```

$$\{4\}$$

solveset virker også hvis vi istedet for en ligning angiver et udtryk (altså uden lighedstegn). Da sætter SymPy udtrykket lig med 0 og løser den derved fremkomne ligning (i dette eksempel $b^2 - 16 = 0$):

```
pos_solution = sp.solveset(b**2 - 16, b, sp.Interval(0, oo))
display(pos_solution)
```

$$\{4\}$$

Vi vil bruge den samme syntaks her, hvor vi vil finde rødder i et fjerdegrads polynomium.

```
from sympy.abc import x
expr = x ** 4 - 1
solutions = sp.solveset(expr, x) # Da vi har angivet et udtryk, sætter SymPy udtrykket lig med 0 og løser.
display(solutions)
```

$$\{-1, 1, -i, i\}$$

Er vi kun interesserede i reelle løsninger, kan vi indskrænke domænet til de reelle tal:

```
solutions_real = sp.solveset(expr, x, sp.Reals)
display(solutions_real)
```

$$\{-1, 1\}$$

Solveset virker også, hvis vi ikke har en numerisk værdi for alle symboler, men ønsker den generelle løsning. Hvis vi f.eks. vil finde b i cosinusrelationen som defineret tidligere, kan vi skrive følgende:

```
display(cos_relation)
sol_b = sp.solveset(cos_relation, b)
display(sol_b)
```

$$c^2 = a^2 - 2ab\cos(\theta) + b^2$$

$$\left\{a\cos(\theta) - \sqrt{a^2\cos^2(\theta) - a^2 + c^2}, a\cos(\theta) + \sqrt{a^2\cos^2(\theta) - a^2 + c^2}\right\}$$

Solveset() giver os løsninger skrevet op i mængdenotation for at give os de generelle fuldstændige løsninger. Selvom dette kan være meget fint i nogle tilfælde, ender vi dog i andre tilfælde med en generel og ganske ubrugelig løsning. Dette sker eksempelvis, hvis vi prøver at løse for θ direkte:

```
sp.solveset(cos_relation, theta)
```

$$\{\theta \mid \theta \in \mathbb{C} \wedge -a^2 + 2ab\cos(\theta) - b^2 + c^2 = 0\}$$

Hvilket kan læses som "De værdier θ for hvilket det gælder at θ er et komplekst tal og at cosinusrelationen er opfyldt". Det er jo rigtig nok (sammenlign med selve cosinusrelationen!), men fortæller os ikke så meget, vi ikke vidste i forvejen. Det leder os videre til den anden metode til ligningsløsning:

5.3.2 Solve

Solve() er den ældre funktion, som på trods af at være mindre generel oftest giver os en brugbar løsning. Syntaksen for input til Solve() er den samme som for Solveset()

```
sp.solve(cos_relation, theta)
```

```
[-acos((a**2 + b**2 - c**2)/(2*a*b)) + 2*pi,
 acos((a**2 + b**2 - c**2)/(2*a*b))]
```

Når der som her er flere løsninger, giver Solve()løsningerne som elementer i en liste, som display ikke kan konvertere til LaTeX. Vi må derfor nøjes med output i tekstformat eller vi kan bruge en løkke til at vise løsningerne en af gangen med `display`:

```
sols = sp.solve(cos_relation, theta)
display(sols)           # Viser listen med løsninger som tekst ... OK, men ikke så␣
↪kønt
for løsning in sols:    # Løkke, der viser løsningerne en af gangen med LaTeX-
↪formattering
    display(løsning)
```

```
[-acos((a**2 + b**2 - c**2)/(2*a*b)) + 2*pi,
 acos((a**2 + b**2 - c**2)/(2*a*b))]
```

$$-\operatorname{acos}\left(\frac{a^2+b^2-c^2}{2ab}\right)+2\pi$$

$$\operatorname{acos}\left(\frac{a^2+b^2-c^2}{2ab}\right)$$

Når man bruger solve kan man angive en liste af ligheder (eller uligheder), som afgrænser den variable. Hvis vi vil bestemme sidelængen b (som er nul eller positiv) ved hjælp af Pythagoras' sætning, kan opgaven i solve-sprog lyde:

```
display(Pytha_indsat)
sol_b = sp.solve([Pytha_indsat, b >= 0], b)
display(sol_b)
```

$$b^2+9=25$$

$$b=4$$

Overordnet set er solve rigtig god til at give én løsning. Det kan altså ofte bruges i sammenhænge, hvor man vil tjekke et resultat, eller hvis man blot skal bruge en vilkårlig løsning og ikke den fuldstændige løsning. Eksempel: Vi løser $\sin(\theta)=1$:

```
sp.solve(sp.Eq(sp.sin(theta), 1), theta)
```

```
[pi/2]
```

Her får vi altså en løsning, og det vil ofte være den løsning, vi leder efter. Men da $\sin(x)$ er periodisk, ved vi, at der er flere løsninger. Så svaret er ikke udtømmende.

I modsætning hertil har solveset den fordel, at den giver et matematisk stringent svar, og den vil altså returnere hele løsningen:

```
sp.solveset(sp.Eq(sp.sin(theta), 1), theta)
```

$$\left\{2n\pi+\frac{\pi}{2} \mid n\in\mathbb{Z}\right\}$$

5.3.3 Numerisk løsning

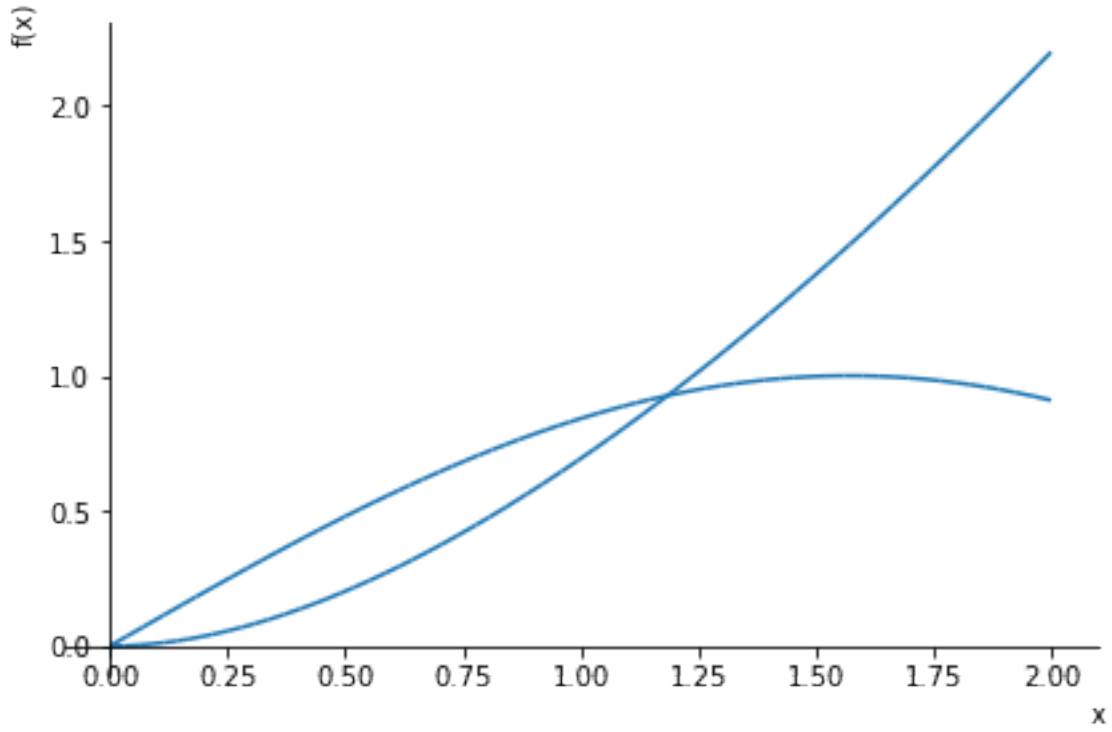
Nogle gange kan vi komme ud for en situation, hvor en opgave ikke har en brugbar eksakt, symbolsk løsning, eller at hverken solve eller solveset giver et svar, vi kan bruge. Vi kan så benytte sp.nsolve til numerisk løsning af ligninger. Vi bruger således SymPy (som er designet til at være et symbolsk værktøj) til et formål, der er på kanten af dets anvendelsesområde, og vi skal derfor bruge værktøjet med forsigtighed. Det er derfor en god idé i disse tilfælde at tegne grafer for at illustrere opgaven. Derved kan vi checke at svaret rent faktisk giver mening i forhold til opgaven, og det tillader os også at give et ret godt gæt på en løsning.

Kaldesekvensen for at bestemme en numeriske løsning er:

```
sp.nsolve(ligning, variabel, startgæt)
```

Hvis nu vi eksempelvis vil finde en løsning til $x\log(x+1) = \sin(x)$, starter vi med at lave et plot (se notesbogen om plotning for en mere grundig forklaring af syntaksen).

```
from sympy.abc import x
from sympy.plotting import plot

figur = plot(x * sp.log(x + 1), sp.sin(x), (x, 0, 2))
```



Heraf ser vi altså, at et godt startgæt vil være omkring $x = 1.2$. Dette vil vi nu bruge.

```
lign = sp.Eq(x * sp.log(x + 1), sp.sin(x))
display(lign)
sp.nsolve(lign, x, 1.2)
```

$$x\log\left(x+1\right) = \sin\left(x\right)$$

$$1.1852999437201$$

Sympy leder med `nsolve` blot efter en løsning. Vi kan altså ende med at få forskellige svar alt efter, hvad vores startgæt er. Gætter vi $x = 0.1$ får vi:

```
sp.nsolve(lign, x, 0.1)
```

$$1.64437672593563 \cdot 10^{-51}$$

som er en numerisk værdi meget tæt på 0, der også er en løsning til ligningen. Det er derfor vigtigt at være opmærksom på, hvilken løsning, der er relevant i en konkret sammenhæng, og lave et godt startgæt inspireret af dette.

5.4 Plotting i SymPy

Vi kan tegne grafer for funktioner og udtryk ved hjælp af SymPy. Det mest grundlæggende værktøj er funktionen plot, som vi importerer fra sympy.plotting her:

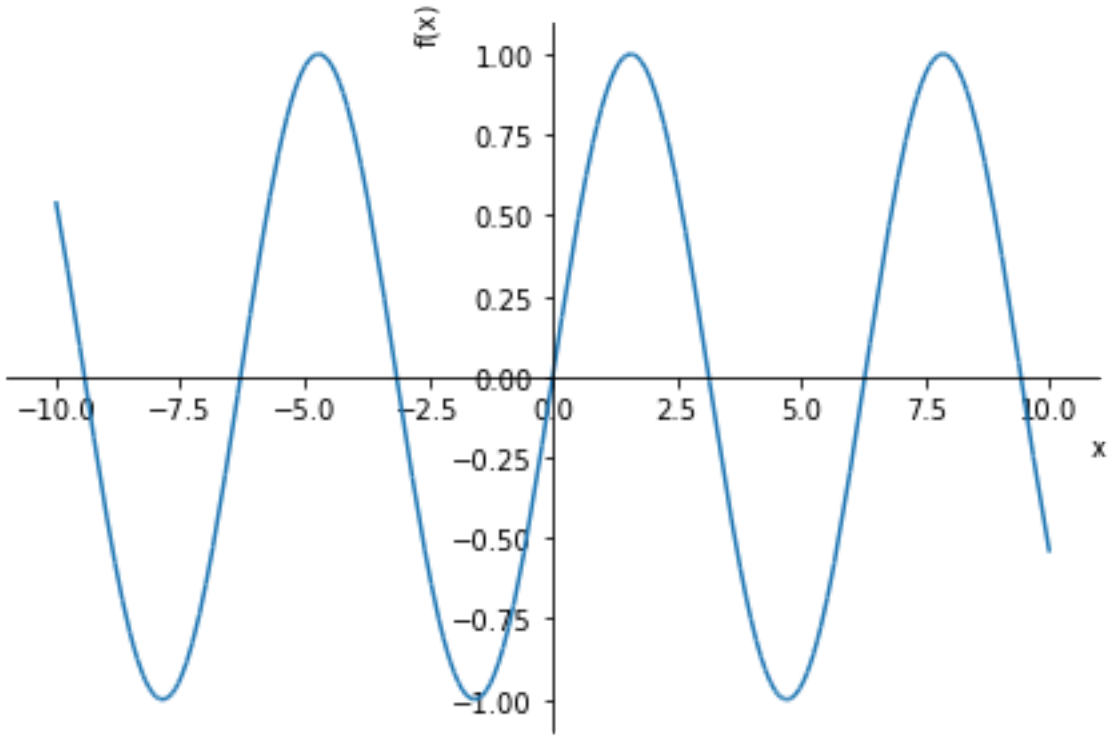
```
# Den anbefalede standardblok for SymPy:
import sympy as sp                    # Importer sympy
from sympy.abc import x               # Vi vælger at importere x som symbolsk␣
↪variabel.

# Specifik import til plotteformål:
from sympy.plotting import plot
```

For at komme i gang vil vi gerne tegne grafen for en sinuskurve og definerer derfor først sinus som et udtryk:

```
expr = sp.sin(x)
```

og tegner derefter ved at bruge funktionen plot.

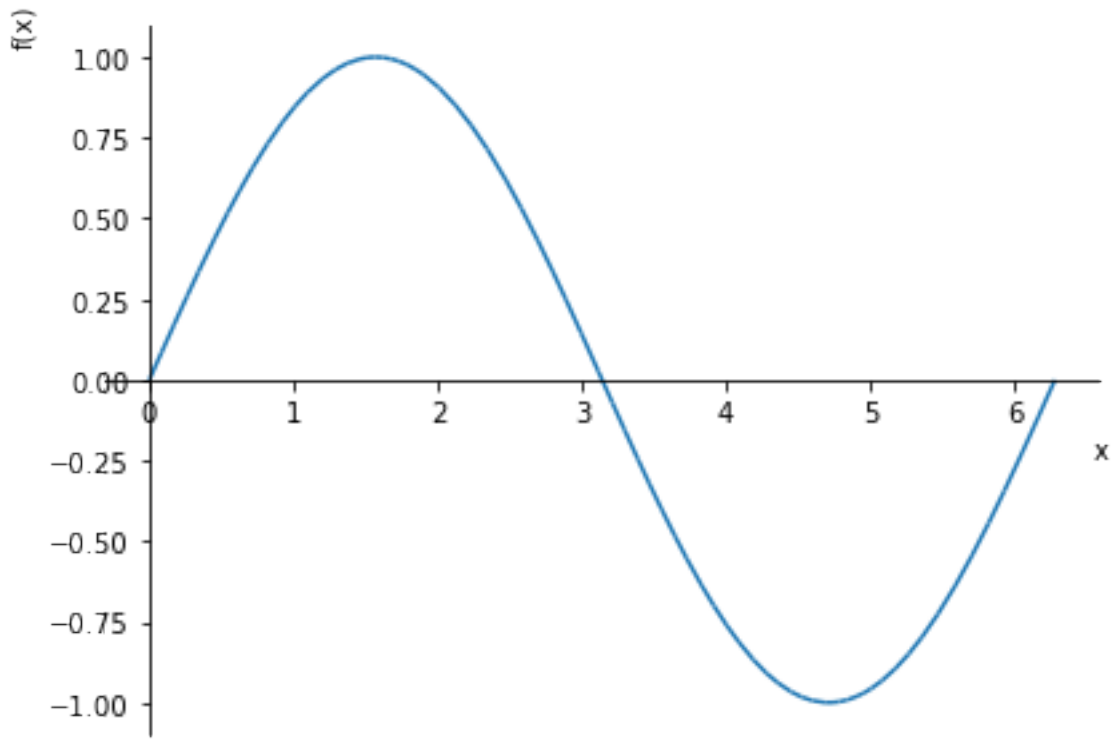
```
plot(expr);
```



Tip: Bemærk semikolonet, der beder Python om ikke at skrive outputtet, som her ville være en kode i stil med sympy.plotting.plot.Plot at 0x136af246608, som ikke er så relevant i denne kontekst.

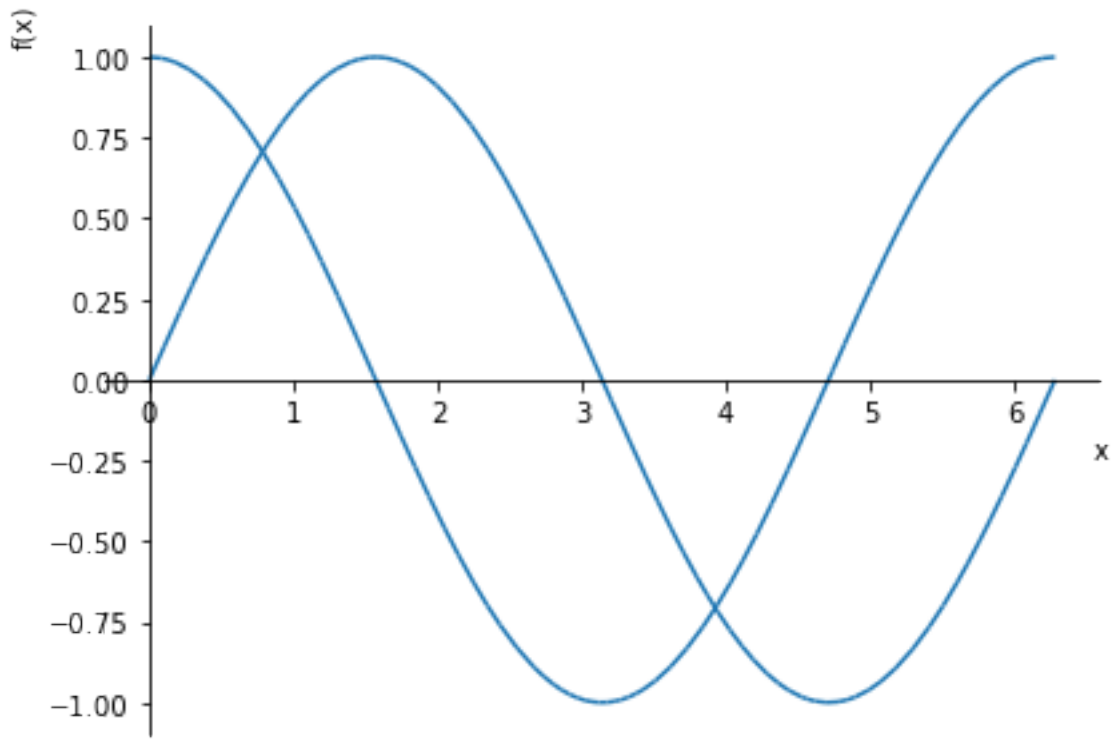
Hvis man ønsker at tegne grafen for et bestemt x-interval, skriver man plot(udtryk, (variabel, start, slut)). Eksempel:

```
from sympy import pi
plot(expr, (x, 0, 2 * pi));
```



Man behøver naturligvis ikke at definere udtrykket først som en variabel, men kan skrive funktionen eller udtrykket direkte ind i plot. Hvis man ønsker at lave flere grafer i samme figur, kan man skrive plot(udtryk_1, udtryk_2, (variabel, start, slut)) som det fremgår her:

```
plot(sp.sin(x), sp.cos(x), (x, 0, 2*pi));
```



5.4.1 Parametre for Plot

Udover at angive funktionen/udtrykket og afgrænsningen af den variable, kan man til plot angive en del andre oplysninger, som angives ved at tilføje keyword = værdi i slutningen af udtrykket. Nogle hyppigt anvendte keywords er:

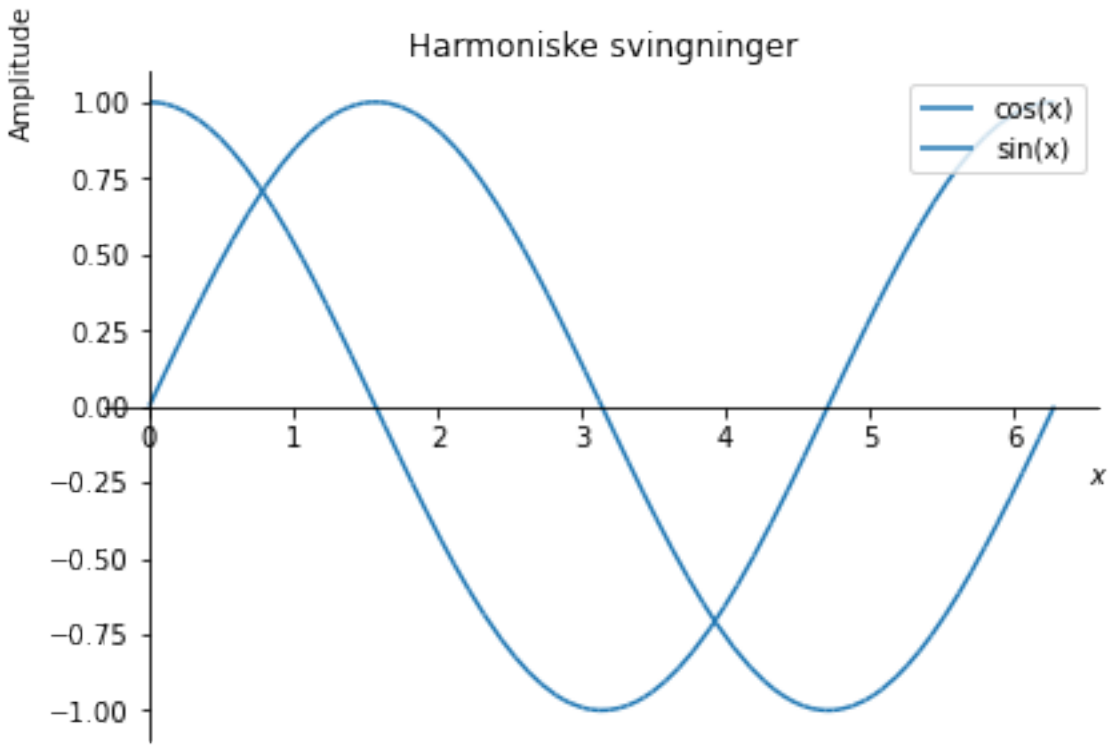
- title angiver titlen på figuren. Angiv værdien som en string, altså som "Ønsket titel"
- legend angiver om der skal stå en beskrivelse af graferne. Angiv i så fald værdien True
- line_color giver grafen en bestemt farve. Angiv værdien som en string, der indeholder en standardfarve, eksempelvis "red", "green", eller som en RGB-farvekode (r, g, b), hvor de tre værdier er mellem 0 og 1
- xlim og ylim fastlægger aksernes afgrænsninger i hhv. vandret og lodret retning. Angiv to værdier (fra, til) for hver akse.
- xlabel og ylabel sætter navne på hhv. $x-$ og $y-$akse. Angiv som string.

Andre keywords kan findes i dokumentationen her

Tip: Når vi angiver tekststrenge på et plot, kan vi skrive `r"$LaTeXkode$"` hvis vi gerne vil have *LaTeXkode* formatteret ved hjælp af Latex. `r` betyder, at tekststrengen skal læses rå og ikke som Python normalt ville fortolke teksten (f.eks. giver `\n` et linjeskift i Python). At kunne bruge LaTeX er særligt praktisk når man vil bruge græske bogstaver som variable.

Med disse muligheder kan vi forbedre vores seneste figur:

```
plot(sp.cos(x), sp.sin(x), (x, 0, 2*pi),
     title = "Harmoniske svingninger",
     legend = True,
     xlabel = r"$x$",
     ylabel = 'Amplitude');
```



Tip: Bemærk her, at vi opdeler et funktionskald på flere linjer ved at indsætte linjeskift efter `,` for at gøre koden mere overskuelig.

Den kompakte kommando til plotning af to grafer tillader desværre ikke umiddelbart at vi giver dem forskellige farver. For at gøre det, kan vi gemme selve figuren som et objekt med et navn ligesom en variabel, hvorefter det er muligt at ændre på de forskellige indstillinger. Vi benytter her `show = False` til ikke at vise vores plot, mens vi stadig redigerer det. Når vi er klar til at vise plottet, skriver vi `figur.show()`.

```
# Vi tegner en graf og navngiver resultatet som et objekt ved navn "figur"
figur = plot(sp.cos(x), 1.5*sp.sin(2 * x), (x, 0, 2*pi), show = False)
```

Når vi vil ændre indstillingerne for en eksisterende graf, gøre dette ved at skrive:

```
figurnavn.keyword = ny_værdi
```

Dermed kan vi nu sætte labels, titler og tilføje en legend ved ligesom overstående at skrive:

```
# Overordnet til hele figure:
figur.legend = True
figur.title  = "Forskellige harmoniske svingninger"
figur.xlabel = r"$x$"
figur.ylabel = r"$A(x)$"
```

Indstillingerne til de enkelte grafer er elementer i en liste, så for at ændre indstillingerne for det i'te plot (husk nulindeksering, så den første kurve er nummer 0), skriver vi:

```
figurnavn[i].keyword = ny_værdi
```

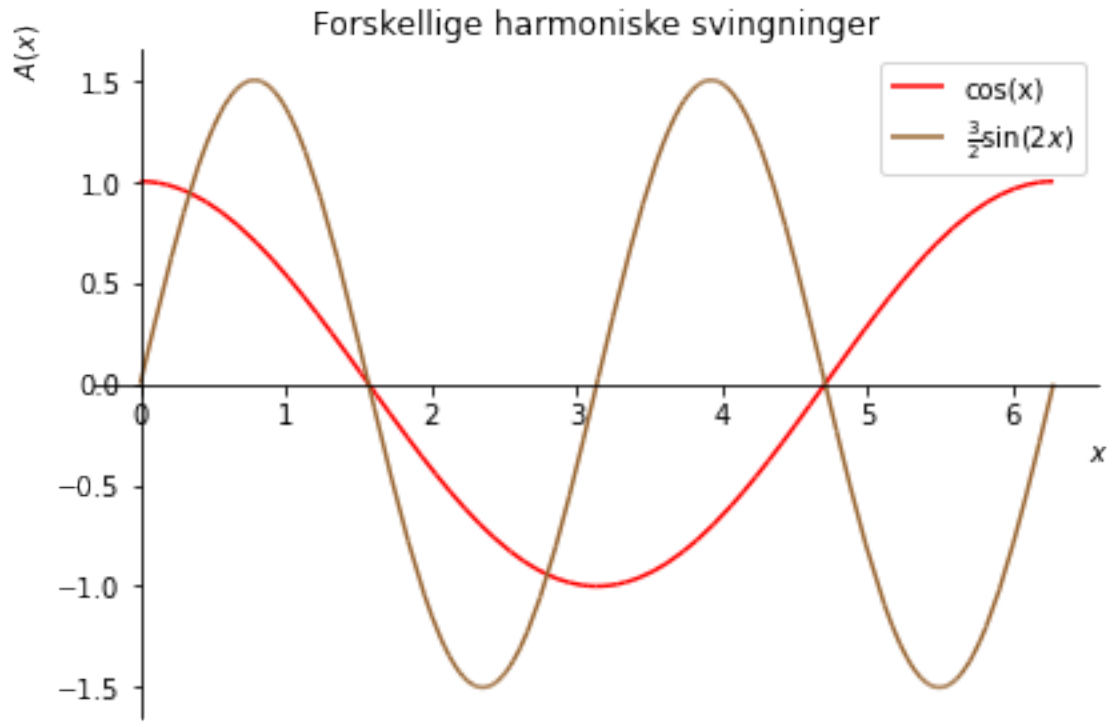
Vi kan altså ændre på de to plots seperat ved at gøre følgende:

```
figur[0].line_color = 'red'   # Der er flere kurver i figuren, og hver af dem har en␣
↪farve.
                              # Kurverne har nummer 0, 1, 2, ... i den rækkefølge, de␣
↪blev tegnet, så cosinus-kurven er 0
```

```
figur[1].line_color = (0.6, 0.4, 0.2)
                              # ... og sinus er nummer 1. Her bruges RGB-farvekode.␣
↪Farven er brun!
figur[1].label      = r"$\frac{3}{2} \sin(2 x)$"
```

Efter at have ændret indstillingerne, beder vi Python om at vise figuren:

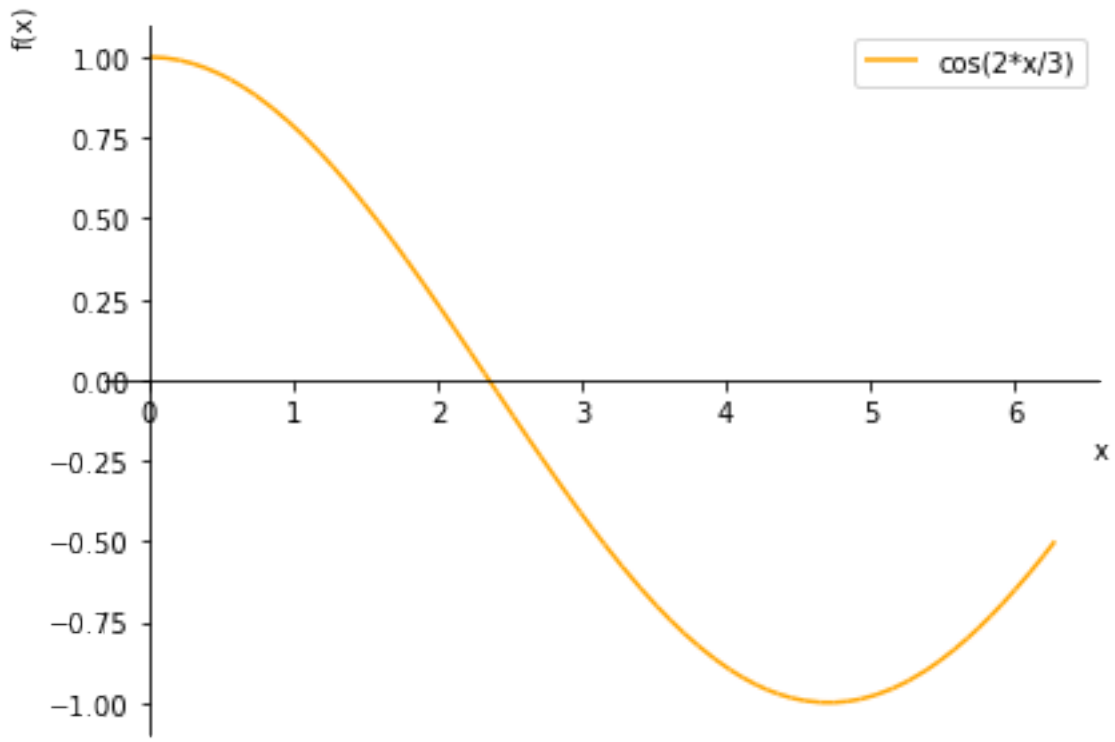
```
figur.show()
```



5.4.2 Figurer som lister

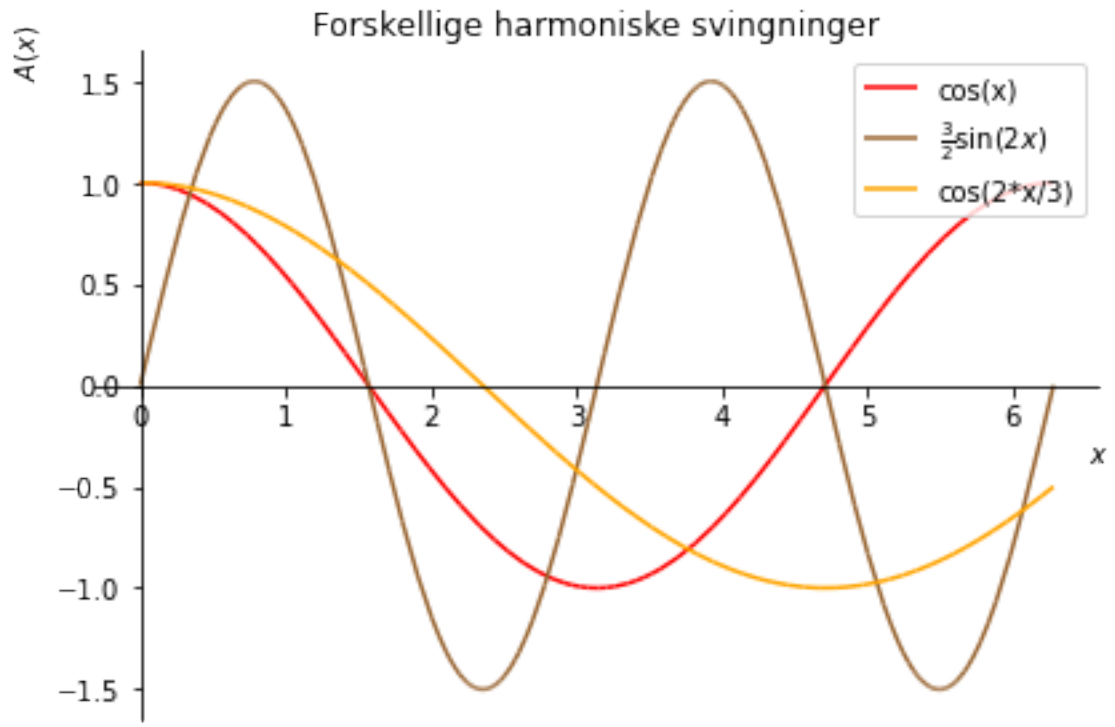
I overstående eksempler har vi benyttet indeksering (som vi kender det fra lister) til at ændre på forskellige parametre. En af fordelene ved dette er at vi kan udvide vores figur ved brug af .append() og .extend() til at bygge videre på en figur, hvis vi ønsker at samle flere plots i en figur. Som eksempel vil vi bruge .append() til at tilføje en ekstra kurve til overstående figur.

```
f = sp.cos(2*x/3)

figur2 = plot(f, (x, 0, 2 * pi), line_color = 'orange', legend = True)
```



Denne kurve har nu fået objektnavnet figur2[0] fordi det er den første kurve i figuren, og vi kan nu tilføje den til figur ovenfor ved hjælp af append:

```
figur.append(figur2[0])
figur.show()
```



Hvis man vil kombinere figurer, som alle har flere kurver, kan man bruge .extend() til at tilføje alle plots fra en figur til den anden. Hvis figur2 således havde haft 2 kurver, ville vi kunne tilføje begge til figur ved at skrive

```
figur.extend(figur2)
```

5.4.3 Gaffelfunktioner

Hvis vi ønsker at lave stykvis definerede funktioner, der har forskellige funktionsudtryk i forskellige intervaller (også kendt som "gaffelfunktioner"), kan vi benytte sp.Piecewise(). Kaldesekvensen er:

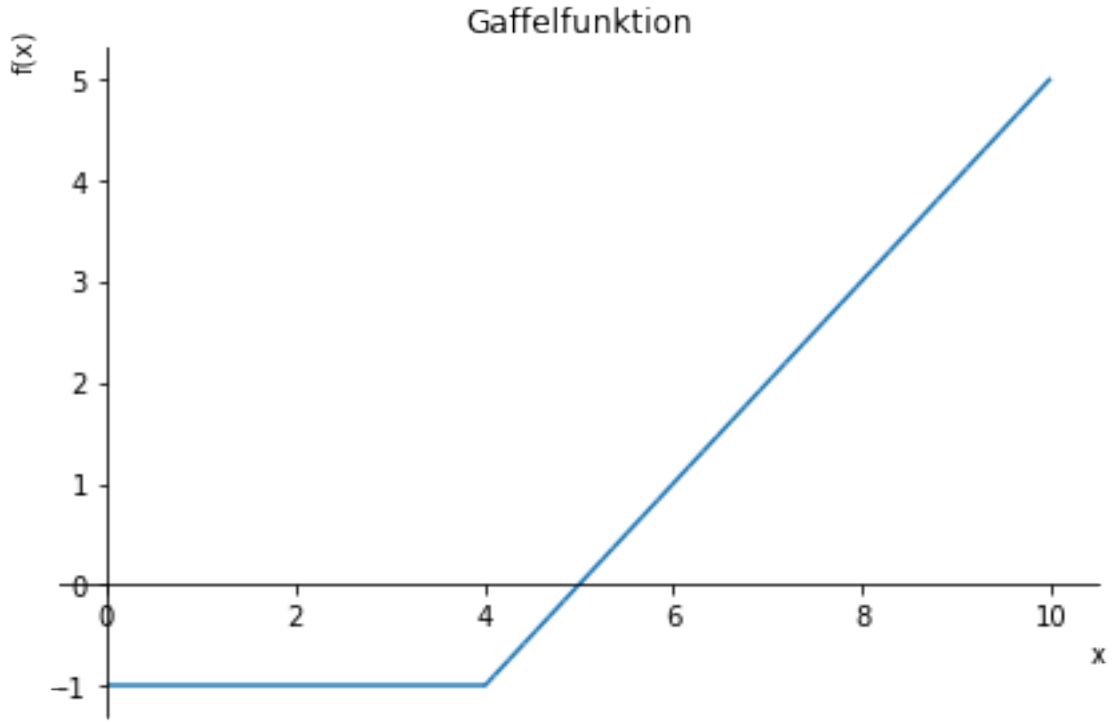
```
sp.Piecewise((udtryk1, intervalbetingelse1), (udtryk2, intervalbetingelse2))
```

Funktionen:

$$f(x) = \begin{cases} -1 & x < 4 \\ x - 5 & x \geq 4 \end{cases}$$

angives derfor som

```
f = sp.Piecewise((-1, x < 4), (x-5, x >= 4))

figur3 = plot(f, (x, 0, 10), title = "Gaffelfunktion")
```



sp.Piecewise kan arbejde med et større antal udtryk og intervalbetingelser end blot to som i eksemplet. Hvis nogle x-værdier opfylder flere af intervalbetingelserne, anvender SymPy det første udtryk hvor et givet x opfylder betingelsen. Man kan bruge dette på en smart måde, hvis man skal tegne en gaffelfunktion, hvor samme funktionsudtryk gælder i flere intervaller for x. Man kan eksempelvis lade det andet udtryk gælde alle steder, hvor den første ikke gælder, ved blot at angive andet udtryks intervalbetingelse som True og angive det som det sidste udtryk. Et eksempel på dette kunne være at betragte gaffelfunktionen:

$$g_1(x) = \begin{cases} -2x & x < -2 \\ x^2 & -2 \leq x \leq 2 \\ 2x & 2 < x \end{cases}$$

som kan skrives kortere som:

$$g_2(x) = \begin{cases} x^2 & |x| \leq 2 \\ 2|x| & \text{ellers} \end{cases}$$

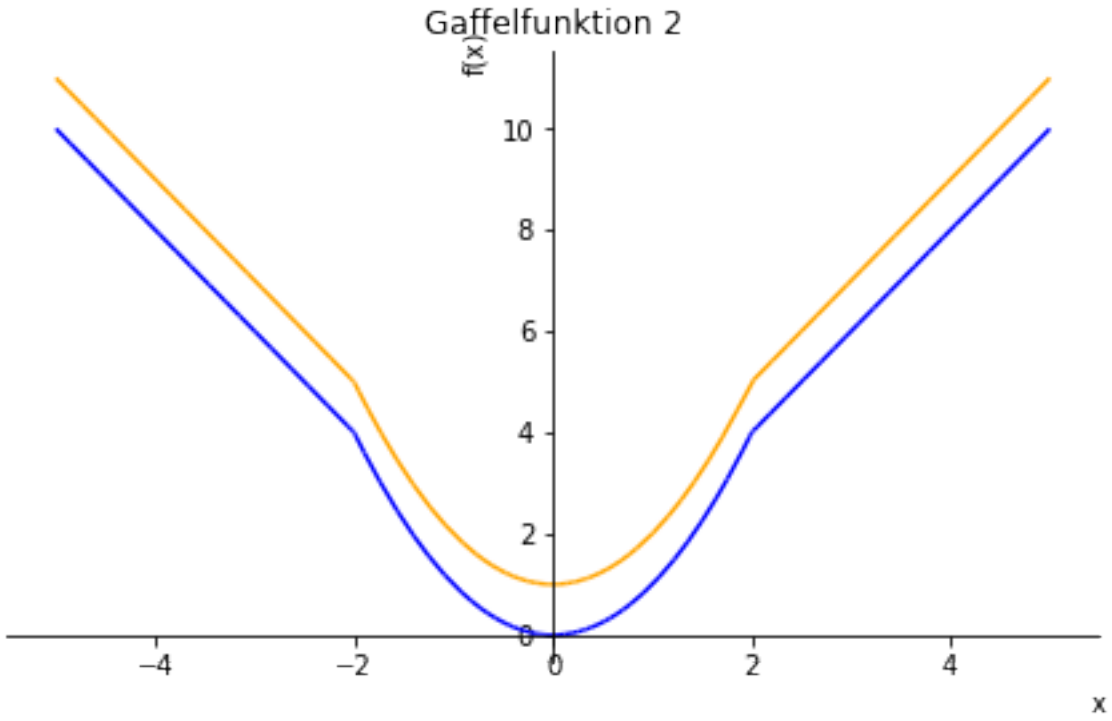
På samme måde kan denne funktion defineres i Python på 2 ækvivalente måder:

```
g1 = sp.Piecewise((-2*x, x<-2), (x**2, abs(x) <= 2),(2*x, x>2))
g2 = sp.Piecewise((x**2, abs(x) < 2),(2*abs(x), True))
```

Bemærk at vi for andet udtryk med vilje ikke skriver nogen betingelse for x, men blot 'True', som altid er opfyldt, idet vi ønsker at bruge det andet udtryk når betingelsen for første udtryk `(abs(x)<=2)` ikke er opfyldt.

Plotter vi nu de to kurver (her separeret visuelt ved at vi har lagt en konstant til den ene), får vi:

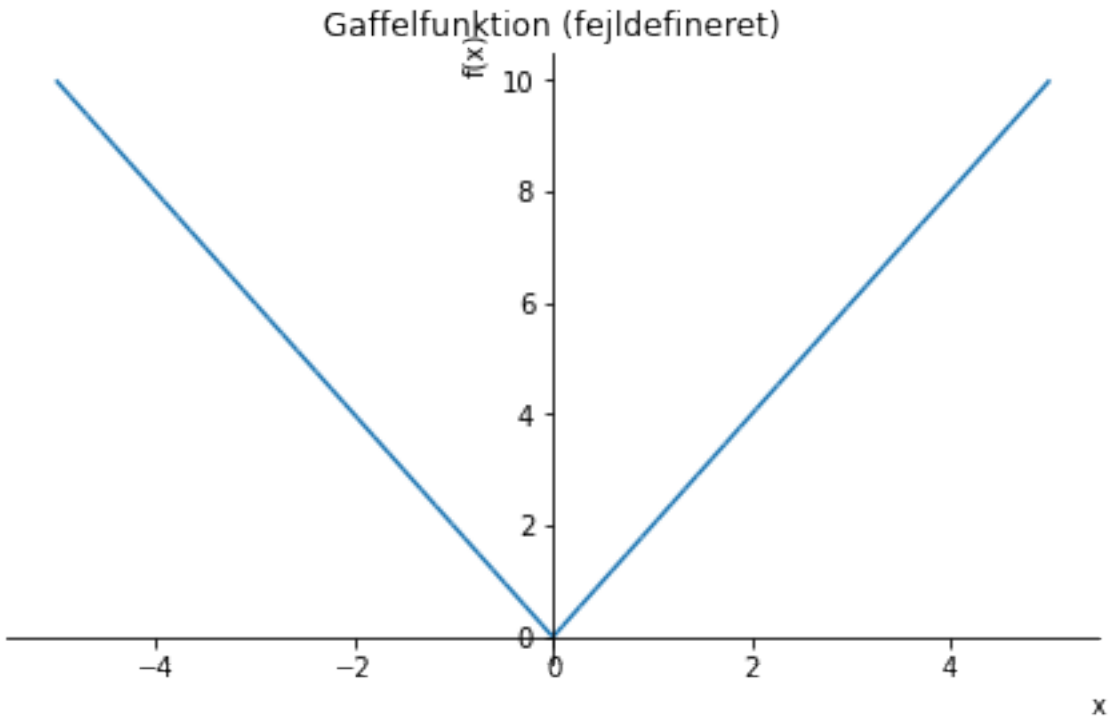
```
figur = plot(g1, g2 + 1, (x, -5, 5), title = "Gaffelfunktion 2", show = False)
figur[0].line_color = "blue"
figur[1].line_color = "orange"
figur.show()
```



Hvilket viser at de to notationsmetoder som forventet giver samme resultat.

Angives udtrykkene derimod i omvendt rækkefølge, kommer udtrykket x^2 aldrig i brug, da betingelsen True altid er opfyldt, og det er blot abs(x), dvs. $|x|$, der tegnes

```
g = sp.Piecewise((2*abs(x), True), (x**2, abs(x) < 2))

figur = plot(g, (x, -5, 5), title = "Gaffelfunktion (fejldefineret)")
```



5.5 Differential- og integralregning (en variabel)

Vi vil i denne notebook gennemgå hvordan vi benytter SymPy til at differentiere og integrere funktioner af en variabel, samt hvordan vi bestemmer Taylorpolynomier.

For simpelhedens skyld vil vi i hele notebooken benytte x som den uafhængige variabel, mens a, b, A og ω er konstanter. Vi starter med at importere SymPy og definere de variable

```
import sympy as sp                          # Importer sympy
from sympy.abc import a, b, A, omega, x     # Vi definerer symboler, som vi kommer til␣
↪at bruge
from sympy import oo, pi                    # Vi importerer uendelig og pi
```

5.5.1 Reduktion

SymPy viser differentierede eller integrerede funktioner på en måde, der nogle gange i højere grad afspejler den bagvedliggende algoritme end almindelig notationspraksis. Inden vi går i gang med at differentiere og integrere vil vi derfor først stifte bekendtskab med SymPys reduktions-værktøj.

En reduktion foretages i SymPy ved at bruge sp.simplify(udtryk) eller `udtryk.simplify()`

Eksempel: Reducér $2\cos^2(x) + \sin^2(x)$

```
# Vi definerer funktionen f:
f = 2 * sp.cos(x) ** 2 + sp.sin(x) ** 2
display(f)

# Vi reducere det nu
```

(continues on next page)

(continued from previous page)

```
f_reduced = sp.simplify(f)
display(f_reduced)
```

$$\sin^2(x) + 2\cos^2(x)$$

$$\cos^2(x) + 1$$

Bemærk først at SymPy viser ledene i det oprindelige udtryk i omvendt rækkefølge i forhold til hvordan vi skrev dem i definitionen. I reduktionstrinnet herefter anvender SymPy den trigonometriske identitet $\cos^2(x) + \sin^2(x) = 1$.

Hvad der udgør et optimalt reduceret udtryk afgøres af SymPys algoritmer, hvilket ikke altid stemmer med vores egne præferencer.

Et eksempel på dette kunne være følgende udtryk:

```
udtryk1 = (x+a+b)**3 - (x+a-b)**3
display(udtryk1)
display(sp.simplify(udtryk1))
```

$$-(a-b+x)^3 + (a+b+x)^3$$

$$-(a-b+x)^3 + (a+b+x)^3$$

SymPy foretrækker således denne form frem for den udgave, hvor parenteserne er ganget ud. Hvis man ønsker et udtryk af en særlig form, findes der relaterede funktioner i SymPy, der laver omskrivninger til mere specifikke formater. See en liste her. I dette tilfælde kan man f.eks. bruge funktionen expand:

```
display(sp.expand(udtryk1))
```

$$6a^2b + 12abx + 2b^3 + 6bx^2$$

Hvis man blot skal tjekke et resultat uden at være interesseret i logikken bag omskrivningerne, kan det være praktisk at bruge det trick, som vi introducerer i eksemplet her:

I en pointopgave uden hjælpemidler skal vi reducere udtrykket $(\frac{1}{2}x + 3a + 3b)^2 - (\frac{1}{2}x + a + b)^2$.

Vi genkender i et lyst øjeblik den såkaldte tredje kvadratsætning $(c+d)\cdot(c-d) = c^2 - d^2$.

Hvis vi sætter $c = \frac{1}{2}x + 3a + 3b$ og $d = \frac{1}{2}x + a + b$. Kan vi benytte følgende omskrivning:

$(c+d)\cdot(c-d) = (x + 4a + 4b)\cdot(2a + 2b)$

Det er nydeligt, men vi vil gerne tjekke med Python:

```
udtryk2 = (x/2+3*a+3*b)**2 - (x/2+a+b)**2
display(udtryk2)
display(sp.expand(udtryk2))
```

$$-\left(a + b + \frac{x}{2}\right)^2 + \left(3a + 3b + \frac{x}{2}\right)^2$$

$$8a^2 + 16ab + 2ax + 8b^2 + 2bx$$

Dette er også et nydeligt udtryk, og uden at vi kender sammenhængen kan vi ikke afgøre hvilket et, der er det pæneste rent notationsmæssigt. Men hvis vi vil tjekke at de to udtryk er ens (f.eks. for at være sikker på at vi har regnet rigtigt), kunne vi nu gange $(x + 4a + 4b)\cdot(2a + 2b)$ ud og sammenligne led, men vi kan også blot beregne forskellen:

```
udtryk3 = (x/2+3*a+3*b)**2 - (x/2+a+b)**2 - (x+4*a+4*b)*(2*a+2*b)
display(sp.simplify(udtryk3))
```

$$0$$

5.5.2 Differentiation

Syntaksen for differentiation ligner meget det, vi så, da vi beregnede grænseværdier (se [afsnittet om grænseværdier])(Notebook2_limits.ipynb). Vi benytter her funktionen sp.diff() og angiver udtryk og hvilken variabel, vi vil differentere efter: sp.diff(udtryk, variabel).

Hvis vi eksempelvis ønsker at differentiere bx^a med hensyn til x, kan vi altså gøre følgende:

```
expr = b * x ** a                 # Vi definerer udtrykket
display(expr)

diff_expr = sp.diff(expr, x)      # Her differentieres det
display(diff_expr)
```

$$bx^a$$

$$\frac{abx^a}{x}$$

Som kan simplificeres med sp.simplify():

```
sp.simplify(diff_expr)
```

$$abx^{a-1}$$

Ønsker vi at differentiere en funktion flere gange, angiver vi blot antallet af gange, vi vil differentiere funktionen, som ekstra argument: sp.diff(udtryk, variabel, antalgange).

Eksempel: Lad os differentiere funktionen $f(x) = A\cos(\omega x)$ med hensyn til x fem gange.

```
f = A * sp.cos(omega * x)       # Definer udtrykket
diff_f5 = sp.diff(f, x, 5)      # Angiv differentieringen, med det ekstra argument: 5,␣
 ↪for at differentiere fem gange
display(diff_f5)
```

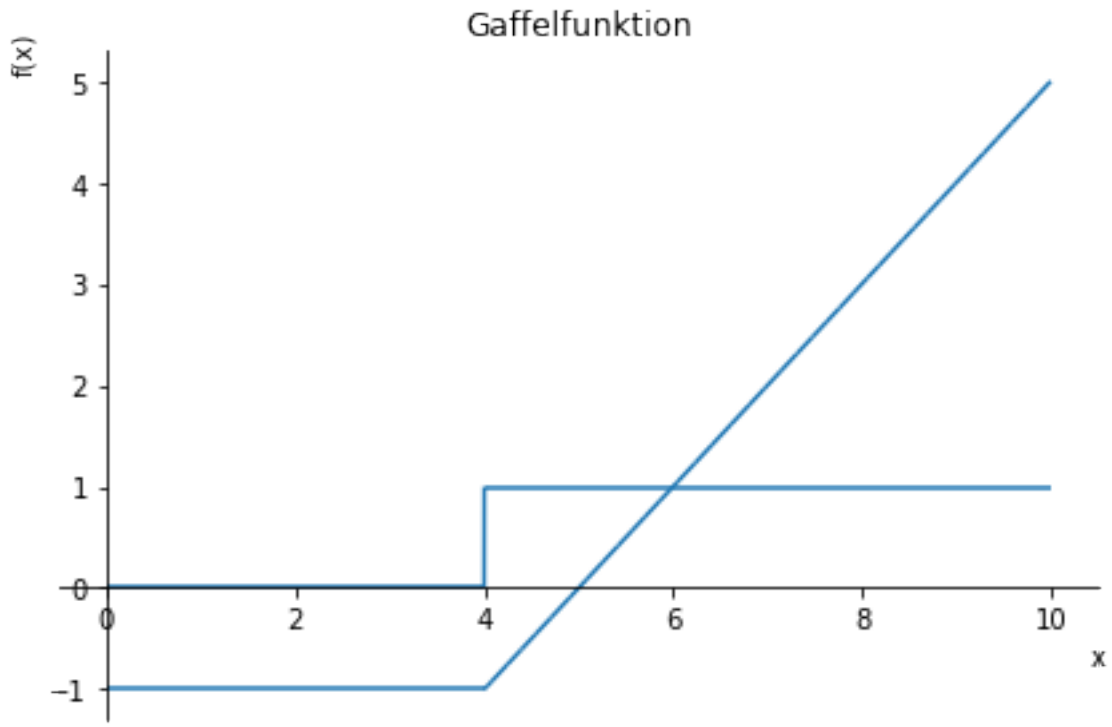
$$-A\omega^5 \sin(\omega x)$$

Bemærk at SymPy godt kan snydes. Vi betragter f.eks. den stykvist lineære funktion fra notebooken i uge 3, og differentierer:

```
g = sp.Piecewise((-1, x < 4), (x-5, x >= 4))
display(g)

diff_g = sp.diff(g, x)
display(diff_g)

from sympy.plotting import plot
figur = plot(g, diff_g, (x, 0, 10), title = "Gaffelfunktion")
```

$$\begin{cases} -1 & \text{for } x < 4 \\ x - 5 & \text{otherwise} \end{cases}$$

$$\begin{cases} 0 & \text{for } x < 4 \\ 1 & \text{otherwise} \end{cases}$$



Vi ser at SymPy håndterer differentiationen rimeligt godt, men at $g'(4)$ angives til 1 selvom g ikke er differentiabel her. Den "lodrette del af grafen" ved $x = 4$ burde også vække vores mistanke.

Et andet relateret eksempel er funktionen $h(x) = \frac{x^2-4}{x+2}$, som ikke er defineret for $x = -2$ fordi nævneren da bliver nul, men som for $x \neq -2$ kan reduceres ved hjælp af tredje kvadratsætning (idet tælleren omskrives $x^2 - 4 = (x+2)(x-2)$) til $h(x) = x - 2$. Vi beder SymPy om at differentiere hhv. med og uden efterfølgende reduktion af resultatet:

```
display(sp.diff((x**2 - 4)/(x + 2)))
display(sp.simplify(sp.diff((x**2 - 4)/(x + 2))))
```

$$\frac{2x}{x+2} - \frac{x^2-4}{(x+2)^2}$$

$$1$$

Vi ser at det første udtryk er korrekt, men at SymPy uden forbehold reducerer udtrykket uden hensyntagen til at $h(x)$ slet ikke er defineret for $x = -2$. Man skal altså selv være særlig opmærksom på punkter, hvor en givet funktion ikke er defineret eller ikke er differentiabel!

5.5.3 Taylorudvikling

Vi kan benytte SymPy til at lave en Taylorudvikling for en given funktion omkring et punkt x_0. Dette gøres med sp.series(), der skal have argumenterne sp.series(funktionsudtryk, variabelnavn, x_0, orden) .Vi kan eksempevis udvikle e^{3x} omkring $x_0 = 0$ til 5. orden ved at skrive:

```
expr = sp.exp(3*x)                      # Definer funktionsudtrykket.
taylor_exp = sp.series(expr, x, 0, 5) # Lav en taylorrække omkring x_0 = 0 af␣
 ↪udtrykket til 5. orden
display(taylor_exp)                     # Vis udtrykket
```

$$1 + 3x + \frac{9x^2}{2} + \frac{9x^3}{2} + \frac{27x^4}{8} + O\left(x^5\right)$$

Et andet eksempel hvor leddene ikke præsenteres i samme logiske orden er udviklingen til 4. orden af funktionen $f(x) = \frac{3}{1-3x}$ omkring $x_0 = 1$.

```
expr = 3 / (1-3*x)                          # Definer
taylor_expr = sp.series(expr, x, 1, 4)      # Beregn taylorudviklingen / Taylorrækken
display(taylor_expr)                        # Vis
```

$$-\frac{15}{4} - \frac{27\left(x-1\right)^2}{8} + \frac{81\left(x-1\right)^3}{16} + \frac{9x}{4} + O\left(\left(x-1\right)^4; x \to 1\right)$$

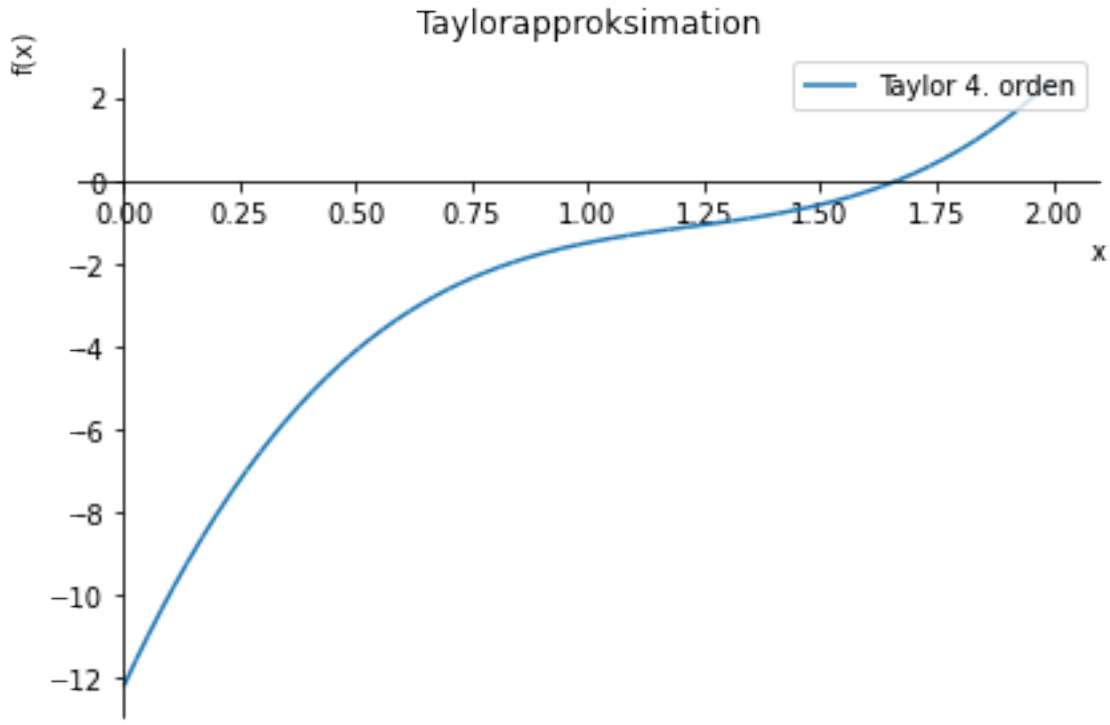
Hvis vi skal regne videre med dette udtryk uden restleddet eller tegne grafer, kan vi fjerne restleddet ved at skrive expr.removeO():

```
taylor_expr = taylor_expr.removeO()
display(taylor_expr)                        # Vis
```

$$\frac{9x}{4} + \frac{81\left(x-1\right)^3}{16} - \frac{27\left(x-1\right)^2}{8} - \frac{15}{4}$$

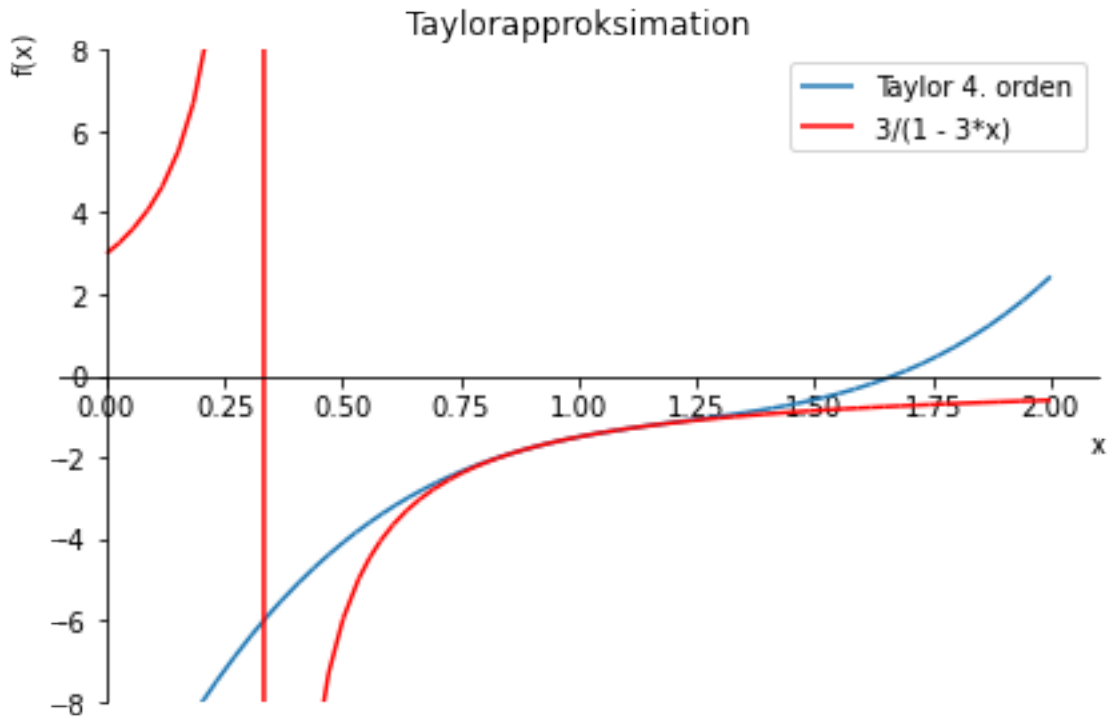
Nu kan vi tegne grafen for Taylorapproximationen til $f(x)$:

```
from sympy.plotting import plot    # Importer plot-funktionen

# Vi genererer nu vores graf som en figur med passende titel og aksegrænser
figur = plot(taylor_expr, (x, 0, 2), legend = True, title = "Taylorapproksimation",␣
↪label = "Taylor 4. orden");
```



Vi tilføjer nu vores originale funktion for at sammenligne:

```
# Vi laver nu en ny graf med den originale funktion.
# Bemærk, at vi skriver show = False for ikke at vise denne inden vi er færdige
original = plot(expr, (x, 0 , 2), line_color = 'red', show = False)

# Vi tilføjer vores original graf til vores figur ved at bruge .append() ... se
 ↪notebooken fra uge 3 for en forklaring.
figur.append(original[0])

# Vi skal nu tilpasse y.aksen, da det originale udtryk får meget store, negative
 ↪værdier for x -> 1/3
figur.ylim = (-8, 8)

# Til sidst viser vi figuren
figur.show()
```



Som ventet er Taylorapproximationen god i nærheden af udviklingspunktet $x_0 = 1$.

5.5.4 Integration

Fremgangsmåden for integration og differentiation minder utrolig meget om hinanden med den oplagte undtagelse, at man både kan beregne bestemte og ubestemte integraler (altså hhv. med og uden grænser). Vi bruger funktionen sp.integrate. Et ubestemt integral findes ved

```
sp.integrate(udtryk, variabel)
```

mens bestemte integraler beregnes med

```
sp.integrate(udtryk, (variabel, fra, til))
```

Bemærk at der skal en parentes omkring den variable og grænserne på samme måde som når man angiver et interval ved graftegning.

Vær opmærksom på, at SymPys integrationsresultater udelader den vilkårlige additive integrationskonstant.

Eksempel: Find stamfunktionen til x^4:

```
expr = x ** 4                                   # Definer udtryk
stamfunktion_expr = sp.integrate(expr, x)       # Integrer udtrykket med hensyn til x
display(stamfunktion_expr)                      # vis
```

$$\frac{x^5}{5}$$

På samme måde findes det bestemte integral $\int_0^{6\pi}(\sin(x))^2 dx$ som også kan skrives som $\int_0^{6\pi}\sin^2(x)dx$:

```
expr = sp.sin(x) ** 2                   # Definer udtryk
sp.integrate(expr, (x, 0, 6 * pi))      # Integrer udtrykket med hensyn til x fra 0 til␣
↪6 pi
```

$$3\pi$$

Oftest kan det også være nyttigt at bruge enten algebraiske symboler eller eventuelt at integrere til uendelig. Her indskriver man bare dette i det bestemte integrals grænser.

```
# Vi kan eventuelt udregne integralet for en gaussisk funktion
expr = sp.exp(- x ** 2)
sp.integrate(expr, (x, -oo, oo))
```

$$\sqrt{\pi}$$

```
# Eller vi kan integrere en polynomium fra a til b
from sympy.abc import a, b
expr = x ** 3 - 1

sp.integrate(expr, (x, a, b))
```

$$-\frac{a^4}{4} + a + \frac{b^4}{4} - b$$

5.5.5 Alternativ Syntaks

I løbet af denne Notebook har vi benyttet os af en forholdsvis stringent metode at skrive funktioner op på, som altid indeholder sp.funktion(). Mange af disse metoder er dog også allerede indbygget i de enkelte udtryks syntaks. Vi kan f.eks. i stedet for at skrive sp.simplify(expr) skrive expr.simplify(). Dette er en lille smutvej, men det giver nogle gange god mening at bruge. Herunder er vist nogle korte eksempler:

```
# Vi viser, at man kan differentiere på følgende to ekvivalente måder:
expr = sp.sin(x)

sp.diff(expr)
```

$$\cos(x)$$

```
expr.diff(x)
```

$$\cos(x)$$

Ligeledes kan vi benytte simplify og integration på denne måde:

```
expr = 2 * (1 - sp.cos(2 * x))

# Vi simplificerer
expr.simplify()
```

$$4\sin^2(x)$$

```
# Beregner et bestemt integral
expr.integrate((x, 0, pi))
```

$$2\pi$$

```
# Finder stamfunktion ved brug af integrate
expr.integrate(x)
```

$$2x - \sin(2x)$$

5.6 Analyse for funktioner af flere variable

Vi har nu udvidet vores funktionsdefinition til at omfatte funktioner af flere variable. Som udgangspunkt kan vi bruge Python/SymPy på samme måde som for funktioner af en variabel. Til at starte med sørger vi for også at importerer variablen `y` fra `sympy.abc`, men vi kunne også tilføje flere variable og gennemføre tilsvarende beregninger for funktioner af 3 eller flere variable.

```
import sympy as sp                    # Importer sympy
from sympy.abc import x, y            # Vi henter vores variabler
```

For funktioner af flere variable bliver diffentialkvotienterne f', f'' afløst af lidt flere begreber. Vi vil i det følgende demonstrere, hvordan man beregner partielt afledede, gradienter og Hesse-matricer (som dog først bliver berørt til sidst i LinAlys-kurset).

5.6.1 Partielt afledede

Når vi går fra en til flere variable, spiller de partielt afledede en central rolle. De beregnes på en helt tilsvarende måde som vi tidligere har beregnet f', idet vi blot skal være opmærksomme på, hvilken variabel, vi differentiere i forhold til. Vi har også ved differentiation af funktioner af en variabel (se notebook til uge 4) angivet navnet på den variable, f.eks. sp.diff(expr, x), men dette var faktisk ikke strengt nødvendigt for simple funktioner, da SymPy i mange tilfælde kan gætte hvad der er den variable. Når vi har to variable, er det derimod afgørende at vi angiver den relevante variabel eksplicit.

Ellers foregår mange beregninger på samme måde:

```
expr = sp.sin(x**2 * y)          # Vi definerer vores funktion af de to variable x og y
display(sp.diff(expr, x))        # Og differentierer, som vi plejer ift x
display(sp.diff(expr, y))        # Og i forhold til y
```

$$2xy\cos(x^2y)$$

$$x^2\cos(x^2y)$$

Derimod giver display(sp.diff(expr)) en fejlmeddelelse. Vi kan også differentiere flere gange efter hinanden i én kommando: Hvis vi f.eks. ønsker at udregne $\frac{\partial^2}{\partial x \partial y}\left(x^2 \cdot e^y\right)$, skriver vi sp.diff(expr, y, x).

Bemærk rækkefølgen af x og y her: i $\frac{\partial^2}{\partial x \partial y}$ står ∂y bagerst, og logikken er, at da dette er nærmest funktionsudtrykket, skal vi differentiere efter y først. Mere formelt er $\frac{\partial^2 f}{\partial x \partial y} \equiv \frac{\partial}{\partial x}\left(\frac{\partial f}{\partial y}\right)$.

Når vi opskriver dette i Python, skal vi derimod liste variable i differentiations/integrationsrækkefølgen i almindelig læseretning. Dette giver os:

```
expr = x**2*sp.exp(y)
display(expr,                    # Display kan bruges til at vise flere ting uden at␣
↪kalde den flere gange.
        expr.diff(x),
        expr.diff(x, y),
        expr.diff(y),
        expr.diff(y, x))
```

$$x^2 e^y$$

$$2xe^y$$

$$2xe^y$$

$$x^2 e^y$$

$$2xe^y$$

Tip: For at gøre koden her mere læselig, nøjes vi med at kalde `display()` en gang, hvor vi til gengæld giver den flere udtryk som den skal vise. Desuden benytter vi, at `expr.diff(x)` er tilsvarende til `sp.diff(expr, x)`.

5.6.2 Gradienter og retningsafledede

Et centralt begreb for funktioner af flere variable er gradienten, som vi (i funktioner af to variable) skriver som:

$\nabla f(x,y) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right)$

Vi tager altså den partielle differentieret i forhold til vores variable og sætter dem sammen som en vektor.

Der er flere måder at beregne gradienten på i Python/SymPy, men her vil vi anvende en måde, der læner sig op af den måde, vi udregner den på i hånden. Vi vil altså lave en vektor bestående af de forskellige partielt afledede. Vi gør dette med sp.derive_by_array(), der som input skal bruge en funktion og en liste af de variable, der skal differentieres efter.

Eksempel: Vi vil finde gradienten af $f(x,y) = e^{-x^2-y^2}$.

```
expr = sp.exp(- x ** 2 - y ** 2)

grad = sp.derive_by_array(expr, [x, y])
grad
```

$$\begin{bmatrix} -2xe^{-x^2-y^2} & -2ye^{-x^2-y^2} \end{bmatrix}$$

Resultatet kan umiddelbart aflæses, men hvis vi vil regne videre med gradienten som en vektor, bliver vi nødt til at benytte sp.Matrix() til at konvertere resultatet til matrixform, som vi kender fra *afsnittet om linær algebra*.

Lad os nu eksempelvis beregne den retningsafledede for ovenstående funktion i retningen $\left(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right)$ i punktet med $(x,y) = (1,2)$. Vi beregner nu jvf. TK sætning 2.57:

```
grad_vec = sp.Matrix(grad)     # Konverterer til vektor/matrix-format
grad_vec                       # Vi viser det lige
```

$$\begin{bmatrix} -2xe^{-x^2-y^2} \\ -2ye^{-x^2-y^2} \end{bmatrix}$$

Vi definerer nu retningsvektoren med `sp.Matrix()`.

```
retning = sp.Matrix([1/sp.sqrt(2), 1/sp.sqrt(2)])
retning
```

$$\begin{bmatrix} \frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} \end{bmatrix}$$

Og nu kan vi beregne prikproduktet med `.dot()`:

```
prikket = grad_vec.dot(retning)
prikket
```

$$-\sqrt{2}xe^{-x^2-y^2} - \sqrt{2}ye^{-x^2-y^2}$$

Til sidst indsætter vi $(x, y) = (1, 2)$:

```
resultat = prikket.subs(x, 1).subs(y, 2)
resultat
```

$$-\frac{3\sqrt{2}}{e^5}$$

5.6.3 Hessematricen

Sidst i kurset vil vi bruge den såkaldte Hessematrix, der er en $n \times n$-matrix, der indeholder alle andenordens afledede for en funktion af n variable. Vi vil specielt bruge determinanten af Hessematricen til at undersøge opførslen af funktioner af 2 variable omkring et stationært punkt (se TK sætning 3.4, der som det fremgår af kommentaren lige under sætningen kan formuleres ved hjælp af Hessematricen).

Man kunne beregne Hessematricen ved at beregne de dobbelt afledede som beskrevet ovenfor, men det er nemmere at importere den fra underbiblioteket af SymPy, som hedder sympy.matrices. Dette gøres med kommandoen from sympy.matrices import hessian. Herefter kan vi nu benytte hessian(udtryk, variabelliste) på samme måde, som vi brugte sp.derive_by_array til at udregne gradienten:

```
from sympy.matrices import hessian          # Importer Hessematricen
expr = sp.exp(- x ** 2 - y ** 2)            # Definer samme funktion som tidligere

H = hessian(expr, [x, y])
display(H)
```

$$\begin{bmatrix} 4x^2e^{-x^2-y^2} - 2e^{-x^2-y^2} & 4xye^{-x^2-y^2} \\ 4xye^{-x^2-y^2} & 4y^2e^{-x^2-y^2} - 2e^{-x^2-y^2} \end{bmatrix}$$

Og hvis vi nu vil beregne værdien for $(x, y) = (0, 1)$, kan vi benytte .subs() to gange:

```
H.subs(x, 0).subs(y, 1)
```

$$\begin{bmatrix} -\frac{2}{e} & 0 \\ 0 & \frac{2}{e} \end{bmatrix}$$

5.7 Graftegning for funktioner af flere variable

Så snart man bevæger sig op i flere dimensioner, begynder det at blive sværere at visualisere de funktioner, man arbejder med. SymPy har en række indbyggede funktioner, der især kan hjælpe os med at analysere funktioner af 2 variable. Den flade, som repræsenterer en funktion af to variable, kan tegnes i 3 dimensioner, og vi vil også give eksempler på hvordan man tegner konturplot og niveaukurver.

Plotting i flere dimensioner kan hurtigt blive en omstændelig proces. Vi regner derfor ikke med, at I skal kunne komme op med disse kodestumper selv, men snarere at I kan tilpasse dem til den opgave, I står overfor.

```
import sympy as sp
from sympy.abc import x, y
```

5.7.1 Graftegning i 3D

For at tegne flader i 3D starter vi på samme måde som vi gjorde for 2D-tilfældet, men i stedet for at importere plot fra sympy.plotting, importerer vi nu i stedet plot3d. Herefter kan vi skrive plot3d(expr, (x, y)) eller f.eks. plot3d(expr, (x, -2, 2), (y, -2, 2)) hvis vi manuelt vil bestemme akseskaleringen. Der er oftest nødendigt at kunne rotere flader i 3D for at få et fyldestgørende indtryk af figurer, så vi benytter her %matplotlib notebook som gør vores figurer interaktive. Hvis man eksempelvis vil plotte xy^2 i området omkring origo, kan man gøre følgende:

```
from sympy.plotting import plot3d                # importer plot3d
%matplotlib notebook

expr = x * y**2                                  # Definer udtryk
plot3d(expr, (x, -2.5, 2.5), (y, -2, 2),         # Plot med fastsatte akse-intervaller
       xlabel = "x",
       ylabel = "y");
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

5.7.2 Konturer

Konturplot findes ved at lave en 2D-afbildning af værdien af den pågældende funktion af flere variable langs en linje. I tilfældet med en funktion af 2 variable ligger linjen i xy-planen, og konturen svarer til grafens skæring med den plan, der kan rejses vinkelret på xy-planen og som indeholder den pågældende linje. Se TK afsnit 1.2.1. De simpleste konturer fås ved at holde værdien af enten x eller y fast og så plotte $f(x, y)$ som funktion af den anden variabel. Vi vil her følge TK eksempel 1.6 og betragter funktionen:

$f(x, y) = 2x^2 + 4x - y^2 + 4y$

Vi kan lave et konturplot ved at kigge på funktionens variation når x varierer mens vi eksempelvis sætter $y = 1$:

```
from sympy.plotting import plot

# Definer funktion
f = 2 * x ** 2 + 4 * x - y ** 2 + 4 * y

# Lav et plot
plot(f.subs(y, 1), (x, -5, 5));
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

For at få en fornemmelse af grafen for f, er et enkelt sådan snit ikke nok, så vi vil gerne lave en serie af konturplots. Vi kan her bruge PlotGrid fra sympy.plotting, som lader os sammensætte flere plots i en gitterstruktur. Vi bruger funktionen ved at skrive:

```
PlotGrid(antal_rækker, antal_kolonner, plot1, plot2 ...)
```

Vi kan altså nu lave flere plots og så sætte dem sammen. Hvis vi nu vil lave seks plots hvor $y = -1, ..., 4$ kan vi lave de enkelte grafer ved hjælp af en løkke og sætte dem sammen til sidst:

```
y_værdier = [-1, 0, 1, 2, 3, 4] # Definer liste
figurer   = []                  # Tom liste til figurer

for y_val in y_værdier:  # Pas på med at skrive y her, da vi allerede bruger den som␣
↪symbol-variabel
    figur = plot(f.subs(y, y_val), show = False)    # Indsæt værdien af y og plot uden␣
↪at vise grafen
    figur.axis_center = (0,0)                       # For at gøre de enkelte grafer␣
↪ens, sætter vi den samme akseskæring
    figur.xlim = (-4, 4)                            # ... og samme akseafgrænsinger␣
↪for x-aksen
    figur.ylim = (-15, 20)                          # ... og y-aksen
    figur.title = "y = {}".format(y_val)            # Tilføj titel til plot
    figurer.append(figur)                           # Vi tilføjer figuren til en liste␣
↪af figurer
```

Vi har nu en liste af figurer, som er gemt i `figurer`. Selve listen siger os ikke så meget, blot:

```
figurer
```

```
[<sympy.plotting.plot.Plot at 0x7f9e54ca2d00>,
 <sympy.plotting.plot.Plot at 0x7f9e54ca2070>,
 <sympy.plotting.plot.Plot at 0x7f9e54ca2fa0>,
 <sympy.plotting.plot.Plot at 0x7f9e54cc6250>,
 <sympy.plotting.plot.Plot at 0x7f9e54cc6460>,
 <sympy.plotting.plot.Plot at 0x7f9e54cc63a0>]
```

Men hvis vi nu importerer `PlotGrid` og indsætter figurerne med `*figurer`, så får vi en en samling af figurerne.

Tip: `*` foran en liste, betyder at man "udpakker" listen. Derfor vil `*[1, 2, 3]` svarer til at skrive `1, 2, 3`. Dette bruges primært til at kalde funktioner, som kan tage et arbitært antal input.

```
from sympy.plotting import PlotGrid             # Importer gitterplottefunktion
PlotGrid(2, 3, *figurer);                       # Nu samler vi de 6 figurer i et␣
↪gitterplot
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Det er værd at lægge mærke til at vi eksplicit har sat akseafgrænsningen til at være ens for alle 6 grafer. Ellers vil Python skalere dem fra figur til figur efter de enkelte grafers placering, hvilket vil gøre det svært at sammenligne de seks grafer.

Vi kan nu gentage overstående, men skifte x-værdier:

```
x_værdier = [-4, -3, -2, -1, 0, 2]                  # nu er det en liste af x-værdier
figurer   = []

for x_val in x_værdier:
    figur = plot(f.subs(x, x_val), show = False)    # Indsætter nu værdier for x i␣
 ↪stedet
    figur.axis_center = (0,0)
    figur.xlim = (-2, 6)                            # sæt nye mere passende grænser  ␣
 ↪
    figur.ylim = (-15, 20)                          # også her
    figur.title = "x = {}".format(x_val)            # Skriv nu "x =" i titlen
    figurer.append(figur)

PlotGrid(2, 3, *figurer);                           # Nu samler vi de 6 figurer i et␣
 ↪gitterplot
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

5.7.3 Niveaukurver

Mens konturer er snit mellem grafen for en funktion og "opretstående planer", er niveaukurver snit med vandrette planer $z = c$ (se TK 1.2.2). Vi vil altså tegne løsninger til $f(x, y) = c$ i xy-planen. I SymPy kan vi få et hurtigt overblik ved at benytte funktionen plot_contour (bemærk at plot_contour her tegner niveaukurver og *ikke* konturplot som navnet ellers kunne antyde) som importeres fra sympy.plotting.plot.

Funktionen følger samme syntaks som 3dplot, og giver os et bud på, hvordan niveaukurverne ligger. Desværre har vi ret begrænsede muligheder for selv at vælge indstillinger for denne funktion.

Her undersøger vi (ligesom i TK 1.10) funktionen: $f(x, y) = x^2 + 4y^2$

```
from sympy.plotting.plot import plot_contour
f = x ** 2 + 4 * y ** 2

niveau = plot_contour(f, (x, -5, 5), (y, -5, 5));
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Linjerne i plottet markere altså de sammenhørende (x, y)-værdier, hvor f antager en bestemt værdi. Vi kan se at grafen for funktionen er en aflang bakke (eller fordybning), der er symmetrisk omkring x-aksen. Funktionen plot_contour er ret ny i SymPy-biblioteket og er ikke helt færdigudviklet, og det er desværre ikke muligt manuelt at bestemme hvilken værdier, linjerne skal tegnes ved.

Vi har også muligheder for selv at gøre arbejdet ved at benytte sympy til at tegne såkaldte implicitte udtryk, altså eksempelvis alle punkter i xy-planen, der opfylder, at $f(x, y) = c$ for et givet c. Dette illustreres herunder for den interesserede læser.

Vi betragter nu (jvf TK 1.11) f givet ved: $$f(x, y) = x^2 - xy + y$$

Vi benytter plot_implicit fra sympy.plotting til at finde niveaukurven for $c = 3$. Vi danner først en ligning ved at skrive sp.Eq(udtryk, c) og så grænser for, hvor den skal plotte det henne, på samme måde som contour_plot og plot3d.

```
f = x ** 2 - x * y + y

from sympy.plotting import plot_implicit
plot_implicit(sp.Eq(f, 3), (x, -5, 5), (y, -5, 5));
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Hvis vi vil tegne flere niveaukurver i samme koordinatsystem og selv vil bestemme hvilke værdier, der skal tegnes kurver for, kan vi på samme måde som ved konturplottene ovenfor lave en løkke, hvor vi går igennem de ønskede værdier:

Tip: Når vi skriver `line_color = f"C{i}"`, så vælger SymPy at plotte med den i'te farve i matplotlibs bibliotek. Vi kan derfor loope over `range(len(c_værdier))` i nedenstående, således at `i` angiver en bestemt niveaukurve, og så kan vi få værdien for c ved `c_værdier[i]`.

```
c_værdier = [-3, -1, 1, 3, 5]
figurer   = []

# Vi sætter et loop op over vores værdier for c
for i in range(len(c_værdier)):

    # Lav et nyt implicit plot for hver værdi af c:
    ny_figur = plot_implicit(sp.Eq(f, c_værdier[i]),
                             line_color = f"C{i}",
                             show       = False)

    # Tilføj denne figur til en liste
    figurer.append(ny_figur)

figurer
```

```
[<sympy.plotting.plot.Plot at 0x7f9e54835d30>,
 <sympy.plotting.plot.Plot at 0x7f9e54aa9790>,
 <sympy.plotting.plot.Plot at 0x7f9e54aa98b0>,
 <sympy.plotting.plot.Plot at 0x7f9e54833190>,
 <sympy.plotting.plot.Plot at 0x7f9e547c8850>]
```

For nu at sætte det sammen, laver vi nu en figur med den første af de plots, som vi har lavet, og så looper vi over resten af listen og indsætter de figurer:

```
vis_figur = figurer[0]              #Lav en ny figur ud fra den første figur i listen

# Loop nu over resten af listen ved at indeksere med [1:]
for tilføj_figur in figurer[1:]:
    vis_figur.append(tilføj_figur[0])

# Vis figuren
vis_figur.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Desværre understøtter `plot_implicit` ikke, at vi kan give linjerne labels. Det bedste bud er at kende rækkefølgen af de første kurve-farver i matplotlib, som er blå, orange, grøn, og så må man prøve at danne sig et overblik ud fra dette.

Vi ser at Python finder at $f = 1$ ikke kun er opfyldt langs linjerne $x = 1$ og $y = 1 + x$ (hvilket vi nemt kan indse ved at indsætte i udtrykket for f), men også i et område nær hvor disse to linjer krydser. Dette skyldes at Python/SymPy løser $f(x, y) = c$ numerisk med en given (og tilsyneladende ikke ret høj) opløsning. Ved at betragte f i 3D kan vi se at funktionen rigtigt nok er meget flad omkring $(x, y) = (1, 2)$:

```
figur = plot3d(f, (x, 0.8, 1.2), (y, 1.8, 2.2),
               xlabel = "x",
               ylabel = "y");
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

5.8 Vektorer og matricer

I denne notebook vil vi have fokus på hvordan man definerer, manipulerer og foretager forskellige udregninger med matricer og vektorer. Som altid starter vi med at importere SymPy.

```
import sympy as sp                    # Importer sympy
```

Det grundlæggende er at definere matricer og vektorer på den nemmeste måde. Dette gøres ved at benytte den funktion, der hedder Matrix() fra SymPy. Da man oftest skal bruge denne mange gange, kan det være praktisk at importere den særskilt ved at skrive from sympy import Matrix, men det er lige så fint at skrive sp.Matrix().

5.8.1 Vektorer - definition og regneoperationer

Vektorer defineres ved at give en liste til Matrix funktionen. Vi kan altså skrive Matrix([1, 1, 0]) for at få vektoren: $\begin{bmatrix}1\\1\\0\end{bmatrix}$

Vi kan angive de enkelte indgange som tal eller bruge symbolske variable som vi i forvejen har importeret fra `sympy.abc` ligesom vi tidligere har gjort med x.

Eksempel:

```
from sympy import Matrix
from sympy.abc import a, b, c

vektor1 = Matrix([a, b, c])
vektor1
```

$$\begin{bmatrix}a\\b\\c\end{bmatrix}$$

Det ligger os selvfølgelig også frit for at kombinere tal og variable:

```
vektor2 = Matrix([a, sp.Rational(3, 4), b ** 2 / 2])    # Husk at bruge sp.Rational()␣
↪til at definere brøker
vektor2
```

$$\begin{bmatrix}a\\\frac{3}{4}\\\frac{b^2}{2}\end{bmatrix}$$

De to vektorer vektor og vektor2 kan nu kombineres på sædvanlig vis ved addition, subtraktion og skalarmultiplikation:

```
vektor1 + vektor2    # Blot læg de to vektorer sammen med et almindeligt +
```

$$\begin{bmatrix} 2a \\ b + \frac{3}{4} \\ \frac{b^2}{2} + c \end{bmatrix}$$

```
a * vektor2        # a er importeret som et symbol, så vi kan bruge det som en skalar
```

$$\begin{bmatrix} a^2 \\ \frac{3a}{4} \\ \frac{ab^2}{2} \end{bmatrix}$$

Hvis vi til gengæld forsøger at multiplicere de to vektorer får vi en fejl, da * mellem to Matrix-elementer er matrix-multiplikation, som ikke er defineret for to 3 x 1 matricer. Vi vender tilbage til matrixmultiplikation nedenfor, mens krydsprodukter kan findes i *sektionen om flere metoder i linæer algebra*.

Et produkt, der derimod *er* defineret for vektorer, er det fra gymnasiet velkendte skalarprodukt eller prikprodukt $\mathbf{v_1} \cdot \mathbf{v_2}$ som er et eksempel på et indre produkt (i Messers notation $\langle \mathbf{v_1}, \mathbf{v_2} \rangle$).

Et indre produkt mellem to vektorer vektor1 og vektor2 beregnes med vektor1.dot(vektor2):

```
vektor1.dot(vektor2)
```

$$a^2 + \frac{b^2 c}{2} + \frac{3b}{4}$$

Som forventet får vi summen af produkterne af vektorens indgange, dog i en anden rækkefølge end vi normalt ville vælge ved udregning i hånden.

5.8.2 Matricer - definition og regneoperationer

Matricer defineres på samme måde, som vi har gjort det med vektorer, altså ved at benytte Matrix() - funktionen. Når vi laver en matrix, så skal vi dog give en liste af rækker, der hver især selv er lister. Formatet er det samme, som man bruger til at lave arrays i numpy. Hvis man eksempelvis blot vil lave en tabel med indgange fra 1 til 9, skal man angive det som:

```
A = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
display(A)
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Det er naturligvis en forudsætning, at alle rækkerne har lige mange elementer.

Som for vektorer kan vi benytte symboler i matricer:

```
B = Matrix([[a, b, c], [b, c, b], [c, b, a]])
display(B)
```

$$\begin{bmatrix} a & b & c \\ b & c & b \\ c & b & a \end{bmatrix}$$

Også addition og skalarmultiplikation fungerer som for vektorer:

```
A + 3 * B
```

$$\begin{bmatrix} 3a+1 & 3b+2 & 3c+3 \\ 3b+4 & 3c+5 & 3b+6 \\ 3c+7 & 3b+8 & 3a+9 \end{bmatrix}$$

```
42*B
```

$$\begin{bmatrix} 42a & 42b & 42c \\ 42b & 42c & 42b \\ 42c & 42b & 42a \end{bmatrix}$$

Matricer kan også multipliceres med hinanden (se Messer afsnit 5.1) så længe de har passende dimensioner. Man kan opfatte indgangen på plads (i,j) i produktet AB som prikproduktet af den i'te række i A med den j'te søjle i B, og for at dette kan lade sig gøre, skal antallet af søjler i A være det samme som antallet af rækker i B. Vi kan også skrive betingelsen som at hvis A er en matrix af dimension $m \times n$, skal B have dimension $n \times p$). I dette tilfælde kan vi gange dem sammen ved at benytte *:

```
A * B
```

$$\begin{bmatrix} a+2b+3c & 4b+2c & 3a+2b+c \\ 4a+5b+6c & 10b+5c & 6a+5b+4c \\ 7a+8b+9c & 16b+8c & 9a+8b+7c \end{bmatrix}$$

Ligesom med arrays i numpy, kan vi også trække rækker/kolonner ud af matricer. Hvis vi vil have det øverste element til venstre i ovenstående matrix, kan vi gøre det ved:

```
C = A * B
C[0, 0]
```

$$a+2b+3c$$

Eller vi kan tage den tredje søjle ved:

```
C[:, 2]
```

$$\begin{bmatrix} 3a+2b+c \\ 6a+5b+4c \\ 9a+8b+7c \end{bmatrix}$$

5.8.3 Indbyggede matricer

Til at definere en række meget almindeligt forekommende matricer er der indbygget smutveje.

Disse matricer skal importeres fra sympy.matrices.

Identitetsmatricer: findes ved funktionen eye(dimension). Navnet skyldes at "eye" udtales omtrent som "I", der ofte anvendes som symbol for identitetsmatricen, men allerede er reserveret til den imaginære enhed for komplekse tal. Da identitetsmatricen pr. definition er kvadratisk, angiver man blot dimensionen n og får en $n \times n$ matrix med 1 i diagonalen og 0 alle andre steder:

```
from sympy.matrices import eye

Id4 = eye(4)
display(Id4)
```

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

0- og 1-matricer: Funktionerne zeros(dimension) og ones(dimension) giver en matrix af den ønskede størrelse fyldt med hhv. 0 eller 1-taller. Her kan dimension være to tal n, m eller et enkelt tal n, hvilket resulterer i en $n \times n$ matrix:

```
from sympy.matrices import zeros, ones

nul_matrix = zeros(4)
display(nul_matrix)
```

$$\begin{bmatrix}0 & 0 & 0 & 0\\0 & 0 & 0 & 0\\0 & 0 & 0 & 0\\0 & 0 & 0 & 0\end{bmatrix}$$

```
fire_matrix = 4 * ones(3, 4)
display(fire_matrix)
```

$$\begin{bmatrix}4 & 4 & 4 & 4\\4 & 4 & 4 & 4\\4 & 4 & 4 & 4\end{bmatrix}$$

Diagonalmatricer: laves med funktionen diag(liste). Denne funktion giver man som input blot en liste med de ønskede diagonalelementer. Alle andre indgange er 0:

```
from sympy.matrices import diag

D = diag(1, 2, 3, 4)
display(D)
```

$$\begin{bmatrix}1 & 0 & 0 & 0\\0 & 2 & 0 & 0\\0 & 0 & 3 & 0\\0 & 0 & 0 & 4\end{bmatrix}$$

Man kan også indsætte matricer langs diagonalen i større maricer, således at man får en blok-diagonal-matrix. Man kan f.eks. indsætte en matrix, der beskriver en rotation i planen, i en større matrix, der beskriver samme rotation i (x, y)-planen indlejret i et tredimensionalt rum:

```
from sympy.abc import theta   # Importer theta

# Definer matrix
plan_rot = Matrix([[sp.cos(theta), sp.sin(-theta)], [sp.sin(theta), sp.cos(theta)]])
plan_rot
```

$$\begin{bmatrix}\cos\left(\theta\right) & -\sin\left(\theta\right)\\\sin\left(\theta\right) & \cos\left(\theta\right)\end{bmatrix}$$

```
# Lav diagonal med et 1-tal og så en matrice. diag fylder nu 0 ud alle andre steder.
rum_rot = diag(plan_rot, 1)
rum_rot
```

$$\begin{bmatrix}\cos\left(\theta\right) & -\sin\left(\theta\right) & 0\\\sin\left(\theta\right) & \cos\left(\theta\right) & 0\\0 & 0 & 1\end{bmatrix}$$

Det er desuden muligt at sætte matricer sammen og definere matricer hvor hver indgang er resulatet af en beregning eller logisk test. Disse metoder kan findes i dokumentationen her men vil ikke være nødvendige i LinALys-kurset.

5.9 Matrix-reduktion, ligningsløsning og inverse matricer

En af de store fordele ved lineær algebra er, at vi nemt kan løse lineære ligningssystemer. Til dette kan vi gøre brug af lidt forskellige værktøjer i SymPy, som vil blive gennemgået i denne sektion.

```
import sympy as sp            # Importer SymPy
from sympy import Matrix      # Vi kommer til at definerer mange matricer i denne␣
↪Notebook
```

5.9.1 Matrix-reduktion

Python/SymPy indeholder en række indbyggede metoder til at reducere matricer. Særligt brugbart i LinALys er funktionen, der bringer en matrix på reduceret række-echelonform. For en matrix A kan vi finde den ækvivalente matrix på reduceret række-echelonform ved .rref():

```
A = Matrix([[1, 1, -2, 1], [3, 2, 1, -5], [2, 1, 3, -6]])
display(A)
A.rref()
```

$$\begin{bmatrix}1 & 1 & -2 & 1\\3 & 2 & 1 & -5\\2 & 1 & 3 & -6\end{bmatrix}$$

```
(Matrix([
 [1, 0,  5, -7],
 [0, 1, -7,  8],
 [0, 0,  0,  0]]),
 (0, 1))
```

Resultatet er den reducerede række-echelonform og en liste over nummeret på de kolonner, der har en ledende indgang. Hvis man vil udelade denne liste, kan man i stedet skrive .rref(pivots = False). *Man kan få printet tingene fint, hvis man i starten af Notebooken kalder* `sp.init_printing()`*, men det giver lidt bøvl med vores noteformat, og vi har derfor udeladt dette. Alternativt kan man blot indeksere løsningen:*

```
display(A.rref()[0],
        A.rref()[1])
```

$$\begin{bmatrix}1 & 0 & 5 & -7\\0 & 1 & -7 & 8\\0 & 0 & 0 & 0\end{bmatrix}$$

```
(0, 1)
```

5.9.2 Ligningssystemer med een løsning

Som det nok er gået op for de fleste, anviser lineær algebra en effektiv måde til at bestemme løsningerne til lineære ligningssystemer ved hjælp af matrixregning. Vi kan f.eks. omskrive:

$$\begin{aligned}x_1 + 3x_3 &= 20\\4x_2 + 6x_3 &= 74\\3x_1 + 6x_2 + 11x_3 &= 136\end{aligned}$$

til:

$$\begin{bmatrix} 1 & 0 & 3 \\ 0 & 4 & 6 \\ 3 & 6 & 11 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 20 \\ 74 \\ 136 \end{bmatrix}$$

Når vi har et ligningssystem på denne form, og der findes en entydig løsning, kan vi uden videre beregne løsningen med SymPy. Vi benytter blot LUsolve-metoden på en matrix, og giver den højre side af vores ligningssystem som inputparameter:

```
#Definerer her matrix A og vektor b
A = Matrix([[1, 0, 3], [0, 4, 6], [3, 6, 11]])
b = Matrix([20, 74, 136])

#Løsningen findes nu blot ved:
sol = A.LUsolve(b)
display(sol)
```

$$\begin{bmatrix} 5 \\ 11 \\ 5 \end{bmatrix}$$

Skulle vi være mistroiske overfor moderne datamater, kan vi nu verificere at

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 5 \\ 11 \\ 5 \end{bmatrix}$$

virkelig er en løsning ved at multiplicere matricen A med løsningsvektoren:

```
A * sol
```

$$\begin{bmatrix} 20 \\ 74 \\ 136 \end{bmatrix}$$

Vi kan lave tilsvarende operationer med symbolske variable. Lad A være matricen angivet ovenfor, men i stedet ønsker vi nu at finde en generel løsning til

$$\begin{bmatrix} 1 & 0 & 3 \\ 0 & 4 & 6 \\ 3 & 6 & 11 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} d \\ e \\ f \end{bmatrix}$$

```
from sympy.abc import d, e, f
b = Matrix([d, e, f])

A.LUsolve(b)
```

$$\begin{bmatrix} -\frac{2d}{7} - \frac{9e}{14} + \frac{3f}{7} \\ -\frac{9d}{14} - \frac{e}{14} + \frac{3f}{14} \\ \frac{3d}{7} + \frac{3e}{14} - \frac{f}{7} \end{bmatrix}$$

5.9.3 Omvendte/inverse matricer

Som beskrevet i Messer afsnit 5.2 kan matricer inverteres. Dette gøres meget simpelt for en matrix A med syntaksen A.inv():

```
A.inv()
```

$$\begin{bmatrix} -\frac{2}{7} & -\frac{9}{14} & \frac{3}{7} \\ -\frac{9}{14} & -\frac{1}{14} & \frac{3}{14} \\ \frac{3}{7} & \frac{3}{14} & -\frac{1}{7} \end{bmatrix}$$

Såfremt den omvendte/inverse matrix for A eksisterer, kan vi derfor også løse det ovenstående ligningssystem som

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 3 \\ 0 & 4 & 6 \\ 3 & 6 & 11 \end{bmatrix}^{-1} \begin{bmatrix} d \\ e \\ f \end{bmatrix}$$

hvilket i Python-syntax gøres ved

```
A.inv() * b
```

$$\begin{bmatrix} -\frac{2d}{7} - \frac{9e}{14} + \frac{3f}{7} \\ -\frac{9d}{14} - \frac{e}{14} + \frac{3f}{14} \\ \frac{3d}{7} + \frac{3e}{14} - \frac{f}{7} \end{bmatrix}$$

5.9.4 Matrixrang

Bemærk at man hverken kan finde løsninger med LUsolve eller ved at beregne den inverse matrix hvis ikke ligningssystemet har netop én løsning (dvs. hvis ligningerne i ligningssystemet har indbygget en modstrid eller løsningen har en eller flere frie variable). I disse tilfælde vil begge metoder give fejlmeddelelser.

Årsagen hertil er at koefficientmatricen ikke har fuld rang. Rangen for en matrix A kan findes ved A.rank(), og angiver antallet af ledende et-taller i den matrix, der fremkommer ved at føre A på reduceret række-echelonform.

Af denne grund, og i overensstemmelse med Messer sætning 5.12, er det derfor ingen overraskelse at se at A fra overfor, der repræsenterer 3 lineært uafgængige ligninger med 3 ubekendte, har rang 3:

```
A.rank()
```

```
3
```

5.9.5 Ligningssystemer med flere løsninger: frie parametre

Vi kommer dog ofte ud for ligningssystemer med flere ubekendte end der er lineært uafhængige ligninger, svarende til at løsningen kan parametriseres med et antal frie variable/parametre. Dette svarer til at rangen af den tilsvarende koefficientmatrix A er mindre end antallet af variable i ligningssystemet. I sådanne tilfælde er A ikke invertibel og LUsolve vil derfor ikke virke.

Eksempel: Vi vil gerne finde den fuldstændige løsning til:

$$\begin{bmatrix} 1 & 0 & 3 & 1 \\ 0 & 4 & 6 & 1 \\ -1 & 4 & 3 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

En mulighed er at føre A på reduceret række-echelonform:

```
A = Matrix([[1, 0, 3, 1], [0, 4, 6, 1], [-1, 4, 3, 0]])
A.rref()[0]
```

$$\begin{bmatrix} 1 & 0 & 3 & 1 \\ 0 & 1 & \frac{3}{2} & \frac{1}{4} \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Vi sætter $x_3 = r$ og $x_4 = s$ da der ikke er ledende indgange i 3. og 4. kolonne. Herefter opskriver vi løsningen som

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} -3 \\ -\frac{3}{2} \\ 1 \\ 0 \end{bmatrix} \cdot r + \begin{bmatrix} -1 \\ -\frac{1}{4} \\ 0 \\ 1 \end{bmatrix} \cdot s, \qquad r, s \in \mathbb{R}$$

Men vi kan også benytte metoden .gauss_jordan_solve() som efterligner denne metode, blot med en lidt anden typografi:

```
b = Matrix([0, 0, 0])
display(A.gauss_jordan_solve(b)[0],
        A.gauss_jordan_solve(b)[1])
```

$$\begin{bmatrix} -3\tau_0 - \tau_1 \\ -\frac{3\tau_0}{2} - \frac{\tau_1}{4} \\ \tau_0 \\ \tau_1 \end{bmatrix}$$

$$\begin{bmatrix} \tau_0 \\ \tau_1 \end{bmatrix}$$

Her får vi altså først en løsning parametriseret ved $\tau_0 ... \tau_n$, og dernæst en vektor over de frie parametre.

5.10 Flere metoder i Linær Algebra

Målet for denne sektion er at give en oversigt over de funktioner, som man kan få brug for, når man benytter SymPy som redskab i lineær algebra.

Langt de fleste funktioner i denne notebook virker på samme måde som matrixinversion, som for en matrix A findes ved A.inv().

Først importerer vi de rette pakker:

```
import sympy as sp                  # Importer sympy
from sympy import Matrix            # Vi kommer til at lave mange matricer
```

Tip: Mange metoder i SymPy returnerer en liste over forskellige løsninger. Da disse ikke automatisk bliver vist som matematik, har vi mange steder sat en `*` foran metoden inde i `display`. I stedet for at skrive `display(*liste)` kan man i sin egen Notebook benytte sig af `sp.init_printing()` i starten af notebooken. Dette giver SymPy muligheden for at vælge, hvordan lister skal printes. Vi har desværre ikke kunne bruge denne mulighed på grund af formatteringen af vores notesbøger.

5.10.1 Transponering, adjungering og konjugering

Hvis vi har givet en matrix, kan vi nemt beregne den transponerede matrix med at ombytte rækker og kolonner:

```
A = Matrix([[1, 2], [3, 4]]);
display(A)
```

$$\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix}$$

Den transponerede beregnes med .T

```
A.T    # Bemærk, at der ikke skal parenteser bag T
```

$$\begin{bmatrix}1 & 3\\2 & 4\end{bmatrix}$$

Messer arbejder ikke med komplekse tal som indgange i matricer, men hvis man gør (og det er der mange gode grunde til, f.eks. i kvantemekanik), vil man også møde den såkaldte konjugerede matrix A.conjugate() (der fremkommer ved kompleks konjugering af indgangene af A) og den såkaldte adjungerede (matricen der er As transponerede konjugerede matrix) A.adjoint() eller nemmere A.H . Notationen med H kommer af at den adjungerede matrix også kaldes den hermitisk konjugerede. Eksempelvis kan vi finde den konjugerede matrix her:

```
from sympy import I, pi # Imaginære tal og pi skal hentes

A = Matrix([[1, 3+I], [2-I, sp.exp(I*pi/3)]])
display(A)
A.conjugate()
```

$$\begin{bmatrix}1 & 3+i\\2-i & e^{\frac{i\pi}{3}}\end{bmatrix}$$

$$\begin{bmatrix}1 & 3-i\\2+i & e^{-\frac{i\pi}{3}}\end{bmatrix}$$

hvor vi ser at alle imaginærdelene har skiftet fortegn. Som eksempel på den adjungerede (altså komplekst konjugerede og transponerede) matrix får vi

```
A.H    # Som med .T indeholder syntaksen ikke ()
```

$$\begin{bmatrix}1 & 2+i\\3-i & e^{-\frac{i\pi}{3}}\end{bmatrix}$$

5.10.2 Nulrum og søjlerum

For en matrix M kan vi finde dennes nulrum og søjlerum ved at bruge hhv. M.nullspace() og M.columnspace().

```
M = Matrix([[1, 1, 2], [2, 1, 3], [3, 1, 4]])
display(M)
display(*M.nullspace())
```

$$\begin{bmatrix}1 & 1 & 2\\2 & 1 & 3\\3 & 1 & 4\end{bmatrix}$$

$$\begin{bmatrix}-1\\-1\\1\end{bmatrix}$$

Hvilket giver en liste (her med eet element) af vektorer som er basis for matricens nulrum. Altså vektorer som opfylder ligningen $Mx = 0$. Dette gælder derfor også for linearkombinationer af vektorerne i nulrummet.

Søjlerummet findes helt tilsvarende:

```
display(*M.columnspace())
```

$$\begin{bmatrix}1\\2\\3\end{bmatrix}$$

$$\begin{bmatrix}1\\1\\1\end{bmatrix}$$

Herved får vi en liste af vektorer, som udspænder søjlerummet, og som består af de søjler fra M, der indeholder ledende indgange når M er bragt på række-echelonform. Dette verificerer vi ved at betragte Ms ækvivalente matrix på reduceret række-echelonform:

```
display(*M.rref())
```

$$\begin{bmatrix}1 & 0 & 1\\0 & 1 & 1\\0 & 0 & 0\end{bmatrix}$$

```
(0, 1)
```

5.10.3 Gram-Schmidt-ortogonalisering

Er vi givet flere vektorer i rummet som ikke er lineært afhængigt af hinanden, kan vi danne et ortogonalt (og evt. ortonormalt, hvis vi normaliserer vektorerne) sæt af vektorer ved at bruge Gram-Schmidt-ortogonalisering.

Dette kan være en ret omgangsrig procedure, som man i LinALys kun vil blive bedt om at udføre i relativt simple tilfælde. Men selv i disse simple tilfælde er der masser af muligheder for at lave regnefejl, og det kan derfor være rart hurtigt at kunne checke beregningerne, især for når der er tale om mere end 2-3 vektorer. Til dette importerer man GramSchmidt fra sympy.matrices og benytter den på en liste L af vektorer: GramScmidt(L). Som udgangspunkt normaliserer SymPy ikke de ortogonaliserede vektorer, men hvis vi giver funktionen et ekstra argument GramSchmidt(L, True), fortages normaliseringen som en del af beregningen.

Som eksempel genregner vi her eksemplet fra side 162 i Messer med SymPy:

```
from sympy.matrices import GramSchmidt

# Definer en liste med matrix indgange, som nu er vektorer
L = [Matrix([1, 2, 2]), Matrix([0, 1, 0]), Matrix([0, 0, 1])]
display(*L)
```

$$\begin{bmatrix}1\\2\\2\end{bmatrix}$$

$$\begin{bmatrix}0\\1\\0\end{bmatrix}$$

$$\begin{bmatrix}0\\0\\1\end{bmatrix}$$

```
# Vi benytter nu GramScmidt
display(*GramSchmidt(L))
```

$$\begin{bmatrix}1\\2\\2\end{bmatrix}$$

$$\begin{bmatrix}-\frac{2}{9}\\\frac{5}{9}\\-\frac{4}{9}\end{bmatrix}$$

$$\begin{bmatrix}-\frac{2}{5}\\0\\\frac{1}{5}\end{bmatrix}$$

```
# Eller hvis vi vil inkluderer en normalisering gør vi følgende:
display(*GramSchmidt(L, True))
```

$$\begin{bmatrix}\frac{1}{3}\\\frac{2}{3}\\\frac{2}{3}\end{bmatrix}$$

$$\begin{bmatrix}-\frac{2\sqrt{5}}{15}\\\frac{\sqrt{5}}{3}\\-\frac{4\sqrt{5}}{15}\end{bmatrix}$$

$$\begin{bmatrix}-\frac{2\sqrt{5}}{5}\\0\\\frac{\sqrt{5}}{5}\end{bmatrix}$$

hvilket kan ses at passe med Messers

$\left\{\left(\frac{1}{3},\frac{2}{3},\frac{2}{3}\right),\left(\frac{-2}{\sqrt{45}},\frac{5}{\sqrt{45}},\frac{-4}{\sqrt{45}}\right),\left(\frac{-2}{\sqrt{5}},0,\frac{1}{\sqrt{5}}\right)\right\}$

ved anvendelse af kvadratrods- og brøkregneregler.

5.10.4 Determinant

Determinanten beregnes ved at tilføje .det() til matricens navn:

```
A = Matrix([[1, 2], [3, 4]])
display(A)
A.det()
```

$$\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix}$$

$$-2$$

Vi kan også beregne determinanten for en symbolsk matrix:

```
from sympy.abc import a, b, c, d

B = Matrix([[a, b], [c, d]])
display(B)

B.det()
```

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

$$ad - bc$$

i overensstemmelse med, hvordan vi selv ville regne den.

5.10.5 Sporet / Trace

Sporet udregnes ved at tilføje .trace() til den ønskede matrix, hvilket giver summen af diagonalindgangene:

```
A = Matrix([[pi, 1, 1],[0, pi, 1],[0, 0, pi**2]])
display(A)
A.trace()
```

$$\begin{bmatrix} \pi & 1 & 1 \\ 0 & \pi & 1 \\ 0 & 0 & \pi^2 \end{bmatrix}$$

$$2\pi + \pi^2$$

5.10.6 Krydsprodukt, vektorprodukt / Cross product

For to vektorer i tre dimensioner kan vi benytte beregne krydsproduktet mellem v og w med v.cross(w) (i analogi med hvordan vi beregner det indre produkt):

```
v = Matrix([1, 0, 1])
w = Matrix([1, 2, 0])
display(v, w)

display(v.cross(w))
```

$$\begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} -2 \\ 1 \\ 2 \end{bmatrix}$$

5.11 Egenværdier og -vektorer

En central problemstilling i Linær algebra er at finde egenvædier og -vektorer af en matrice. Dette gøres relativt enkelt i SymPy.

```
import sympy as sp
from sympy import Matrix
```

For en matrix findes egenværdier og -vektorer med A.eigenvals() og A.eigenvects().

A.eigenvals() giver os blot alle egenværdierne til en bestemt matrix i sorteret rækkefølge:

```
A = Matrix([[2, 0, 0], [0, 3, 4], [0, 4, 9]])
display(A)
```

$$\begin{bmatrix}2 & 0 & 0\\0 & 3 & 4\\0 & 4 & 9\end{bmatrix}$$

```
A.eigenvals()
```

```
{11: 1, 2: 1, 1: 1}
```

Dette giver os nu en liste over egenværdierne (altså, 1, 2 og 11) og deres algebraiske multiplicitet, altså hvor mange gange den pågældende egenværdi er rod i det karakteristiske polynomium (her har alle egenværdierne algebraiske multiplicitet 1). For at illustrere hvordan resultaterne vises, "beregner" vi her egenværdierne for et trivielt eksempel:

```
from sympy.matrices import eye
B = Matrix([[2, 0, 0, 0], [0, 2, 0, 0], [0, 0, 2, 0], [0, 0, 0, 5]])
display(B)
```

$$\begin{bmatrix}2 & 0 & 0 & 0\\0 & 2 & 0 & 0\\0 & 0 & 2 & 0\\0 & 0 & 0 & 5\end{bmatrix}$$

```
B.eigenvals()
```

```
{2: 3, 5: 1}
```

Funktionen .eigenvects virker på samme måde og giver både egenværdier og -vektorerne. Hvis man bruger `sp.init_printing()`, kan man kalde dette direkte ved blot at skrive `A.eigenvects()`.

```
A.eigenvects()
```

```
[(1,
  1,
  [Matrix([
   [ 0],
   [-2],
   [ 1]])]),
 (2,
  1,
  [Matrix([
   [1],
   [0],
   [0]])]),
 (11,
  1,
  [Matrix([
   [  0],
   [1/2],
   [  1]])])]
```

Hvis vi vil vise resultatet i matematisk notation, er nogle krumspring nødvendige. Vi benytter `sp.latex()` til at konvertere det hele til LaTeX og viser derefter LaTeX-koden ved `display(Math())`.

```
from IPython.display import Math
display(Math(sp.latex(A.eigenvects())))
```

$$\left[\left(1,\ 1,\ \left[\begin{bmatrix}0\\-2\\1\end{bmatrix}\right]\right),\ \left(2,\ 1,\ \left[\begin{bmatrix}1\\0\\0\end{bmatrix}\right]\right),\ \left(11,\ 1,\ \left[\begin{bmatrix}0\\\frac{1}{2}\\1\end{bmatrix}\right]\right)\right]$$

Her er resultatet altså givet som en liste med (egenværdi, multiplicitet, egenvektorer). Vi kan igen bruge det trivielle eksempel med B for at vise hvad der sker, når en egenværdi har flere tilhørende egenvektorer:

```
display(Math(sp.latex(B.eigenvects())))
```

$$\left[\left(2,\ 3,\ \left[\begin{bmatrix}1\\0\\0\\0\end{bmatrix},\ \begin{bmatrix}0\\1\\0\\0\end{bmatrix},\ \begin{bmatrix}0\\0\\1\\0\end{bmatrix}\right]\right),\ \left(5,\ 1,\ \left[\begin{bmatrix}0\\0\\0\\1\end{bmatrix}\right]\right)\right]$$

I matematikkurser har matricerne gerne pæne (ofte heltallige) egenværdier, mens man i praksis sjældent oplever matricer med så velopdragne egenværdier. Dertil kommer, at store matricer kan gøre det meget hårdt for computeren at regne det hele symbolsk. Derfor vil man (når man eksempelvis beregner egenværdier i kvantemekanik) i stedet bruge NumPy, da der her er nogle ret hurtige implementeringer til at give gode numeriske løsninger. Heldigvis ligner metoderne meget hinanden, og hvis man skulle komme ud for at skulle løse et problem, der er for krævende med sumbolske beregninger, kan man finde NumPys LinAlg værktøjer i den relevante dokumentation.

5.11.1 Diagonalisering

En afgørende pointe i kurset er at undersøge hvornår der kan findes en basis hvor en matrix A er på diagonalform, altså hvornår der findes en matrix P, der opfylder at $D = P^{-1}AP$, hvor D er en diagonalmatrix. For at finde P og D kan vi benytte A.diagonalize(). Vi regner videre med matricen A defineret ovenfor:

```
display(A)
```

$$\begin{bmatrix}2 & 0 & 0\\0 & 3 & 4\\0 & 4 & 9\end{bmatrix}$$

```
display(*A.diagonalize())
```

$$\begin{bmatrix}0 & 1 & 0\\-2 & 0 & 1\\1 & 0 & 2\end{bmatrix}$$

$$\begin{bmatrix}1 & 0 & 0\\0 & 2 & 0\\0 & 0 & 11\end{bmatrix}$$

Outputtet for denne funktion er matricerne P og D. Vi genkender at matricen D netop indeholder egenværdierne langs diagonalen og at P består at (multipla af) egenvektorerne. Vi kan få de pågældende matricer ud så vi kan regne videre med dem på følgende måde:

```
(P, D) = A.diagonalize()
display(P, D)
```

$$\begin{bmatrix}0 & 1 & 0\\-2 & 0 & 1\\1 & 0 & 2\end{bmatrix}$$

$$\begin{bmatrix}1 & 0 & 0\\0 & 2 & 0\\0 & 0 & 11\end{bmatrix}$$

Og vi demonstrerer endelig at matricerne sammensættes som forventet. Først $D = P^{-1}AP$:

```
P.inv() * A * P
```

$$\begin{bmatrix}1 & 0 & 0\\0 & 2 & 0\\0 & 0 & 11\end{bmatrix}$$

Eller omvendt $A = PDP^{-1}$

```
P * D * P.inv()
```

$$\begin{bmatrix}2 & 0 & 0\\0 & 3 & 4\\0 & 4 & 9\end{bmatrix}$$

5.12 Komplekse tal i SymPy

I LinALys lærer du at regne med komplekse tal. SymPy kan være en hjælp f.eks. til at visualisere alle løsninger til komplekse ligninger, men er i kurset primært et værktøj som du kan bruge til at checke de beregninger, der indgår i pointopgaverne og den afsluttende eksamen. Især hvis du har erfaring med at du laver mange regnefejl i omfattende beregninger, kan det være godt at checke beregningerne trin for trin i SymPy.

Vi starter med at importere den imaginære enhed ved hjælp af from sympy import I, I er SymPys standardbetegnelse for det imaginære tal i.

```
import sympy as sp
from sympy import I, pi
```

5.12.1 Almindelige regneoperationer med komplekse tal

Når vi benytter I, får vi automatisk et komplekst tal, og SymPy behandler det som sådan ganske automatisk. Vi kan altså skrive $z = 3 + 4i$ som z = 3 + 4 * I:

```
z = 3 + 4 * I
z
```

$$3 + 4i$$

Eller vi kan lave et mere generelt udtryk, hvor vi istedet benytter symboler:

```
from sympy.abc import a, b
w = a + b * I
w
```

$$a + ib$$

Når vi først har defineret vores udtryk, så kan vi regne med dem på samme måde som med alle andre (reelle) tal. Regneoperationerne +, -, * og / fungerer som de skal.

```
z = 1 + 2 * I
w = - 3 * I
z + w
```

$$1 - i$$

```
z - w
```

$$1 + 5i$$

```
z * w
```

$$-3i\,(1 + 2i)$$

Ved multiplikation og division er det nogle gange en god idé at bede SymPy om at reducere udtrykket. Især, hvis man gerne vil have det på en $x + yi$ form:

```
sp.simplify(z * w)
```

$$6 - 3i$$

```
display(z / w,
        sp.simplify(z / w))
```

$$\frac{i\,(1 + 2i)}{3}$$

$$-\frac{2}{3} + \frac{i}{3}$$

Man kan desuden beregne potenser af et kompleks tal på sædvandligvis ved at bruge **:

```
z ** 3
```

$$(1 + 2i)^3$$

I dette tilfælde er der ikke meget hjælp at hente i simplify:

```
sp.simplify(z ** 3)
```

$$(1 + 2i)^3$$

Vi kan til gengæld bede SymPy om at gange parentesen ud med sp.expand:

```
sp.expand(z ** 3)
```

$$-11 - 2i$$

SymPy ved, hvordan den skal håndtere komplekse tal i mange sammenhænge, hvor fortolkningen er entydig. Man kan altså blot bruge komplekse tal i sp.exp(), sp.cos() eller lignende.

5.12.2 Notationsformer og skift mellem disse

Når vi arbejder med komplekse tal, benytter vi enten kartetiske koordinater eller modulus/argument. I SymPy er der en del metoder til at konverterer den ene eller anden vej, som vi vil gennemgå i dette afsnit.

Kartesiske koordinater, real og imaginærdel.

Funktionerne `sp.re()` og `sp.im()` giver real- og imaginærdelen af et komplekst tal. Det er naturligvis trivielt hvis vi starter med et imaginært tal af formen $x + iy$:

```
z = 10 - 7 * I
display(z,
        sp.re(z),
        sp.im(z))
```

$$10 - 7i$$

$$10$$

$$-7$$

Men det er mere oplysende, hvis vores komplekse tal har en anden form, for eksempel $4e^{i\pi/3}$:

```
w = 4 * sp.exp(I * pi / 3)
display(w,
        sp.re(w),
        sp.im(w))
```

$$4e^{\frac{i\pi}{3}}$$

$$2$$

$$2\sqrt{3}$$

Modulus og argument

Vi kan desuden nemt beregne modulus og argument. Modulus for et komplekst tal er det samme som tallets absolutte værdi (TL s. 126) og findes med sp.Abs(). Bemærk at man bruger et stort A for at adskille kommandoen fra Pythons abs-funktion. For symbolske udtryk vil SymPy dog automatisk bruge sp.Abs() selv hvis vi bruger et lille a. Så for de to tal defineret ovenfor får vi:

```
display(sp.Abs(z),
        sp.Abs(w))
```

$$\sqrt{149}$$

$$4$$

Argumentet findes ved sp.arg() :

```
display(sp.arg(z),
        sp.arg(w))
```

$$-\operatorname{atan}\left(\frac{7}{10}\right)$$

$$\frac{\pi}{3}$$

Hvorefter vi som nævnt tidligere kan bruge .evalf(), hvis vi f.eks. vil have et bud på værdien af $-\arctan(7/10)$ med fire decimaler

```
sp.arg(z).evalf(4)
```

$$-0.6107$$

Kompleks konjugering

Vi vil ret ofte benytte os af kompleks konjugering, hvor vi skifter fortegn på imaginærdelen (svarende til at vi spejler i den reelle akse). Dette kan vi gøre ved brug af sp.conjugate(), som virker på imaginære tal uanset notationsform. Bemærk at funktionen hedder det samme som den tilsvarende funktion for matricer, som vi kender fra Lineær Algebra, men som har en anden kaldesekvens:

```
display(z,
        sp.conjugate(z),   # Konjugering af komplekst tal
        z.conjugate())     # Konjugering af matrix ... men da z kan opfattes som en 1␣
↪x 1 matrix, er resultatet det samme
```

$$10 - 7i$$

$$10 + 7i$$

$$10 + 7i$$

```
display(w,
        sp.conjugate(w))
```

$$4e^{\frac{i\pi}{3}}$$

$$4e^{-\frac{i\pi}{3}}$$

5.12.3 Rødder og ligninger

En central egenskab ved de komplekse tal er, at et polynomium af n'te grad altid har netop n rødder (med multiplicitet). Vi skal ofte finde rødder i polynomier og løsninger til ligninger, og her indgår der ofte kvadratrødder og andre rødder af komplekse tal. Se TK 3.4.2 s. 141, hvor det fremgår at der altid er netop n n'te rødder. Når vi vil finde den n'te rod af et komplekst tal, benytter vi sp.root. Syntaksen er således, at $\sqrt[n]{z}$ skrives som

```
sp.root(z, n, hvilken_rod)
```

hvor hvilken_rod er et tal mellem 0 til $n-1$ (husk at Python tæller fra 0) og fortæller SymPy, hvilken af de n rødder, den skal udregne.

```
z = - 3 + 3 * I
display(z)
```

$$-3 + 3i$$

```
sp.root(z, 4, 0)
```

$$\sqrt[4]{-3+3i}$$

Dette er utvivlsomt korrekt, men når man ønsker et mere anvendeligt svar, kan man f.eks. tvinge SymPy til at udregne real-delen og imaginærdelen som vi gjorde ovenfor:

```
r1 = sp.root(z, 4, 0)
display(sp.re(r1))
display(sp.im(r1))
```

$$\sqrt[8]{2}\sqrt[4]{3}\cos\left(\frac{3\pi}{16}\right)$$

$$\sqrt[8]{2}\sqrt[4]{3}\sin\left(\frac{3\pi}{16}\right)$$

Vi vil nu se på rødderne i et simplere tilfælde, nemlig $\sqrt[4]{-4}$.

Hvis vi ønsker at finde alle rødderne på en gang og få dem præsenteret i en liste, kan vi enten skrive det op som en ligning:

```
from sympy.abc import x
display(*sp.solve(sp.Eq(x**4, -4), x))
```

$$-1-i$$

$$-1+i$$

$$1-i$$

$$1+i$$

Eller vi kan beregne rødderne en efter en ved hjælp af en for-løkke:

```
for i in range(4):
    r = sp.root(-4, 4, i)
    display(sp.re(r) + I * sp.im(r))
```

$$1+i$$

$$-1+i$$

$$-1-i$$

$$1-i$$

Vi kan finde rødder til polynomier ved hjælp af sp.solve(). Vi minder om at hvis vi kun giver sp.solve() et udtryk (og altså ikke en ligning, f.eks. dannet ved hjælp af sp.Eq), så finder funktionen løsninger til den ligning, der fremkommer når udtrykket sættes lig 0:

```
from sympy.abc import z
# Definer p som også har imaginære rødder
p =  z ** 3 - z ** 2 + 4 * z + -4

# Løs med sp.solve ved hjælp af udtrykket for p, der automatisk sættes lig nul
display(*sp.solve(p))
```

$$1$$

$$-2i$$

$$2i$$

```
#... eller med en mindre elegant syntaks, hvor ligningen eksplicit er angivet til z3 -
 ↪ z^2 + 4z = 4
display(*sp.solve(sp.Eq(z**3 - z**2 + 4 * z, 4), z))
```

$$1$$

$$-2i$$

$$2i$$

5.12.4 Visualisering i det komplekse plan

Der er ikke et decideret tegneværktøj til komplekse tal i SymPy. Vi tager derfor de to værktøj, som I har set i MekRel frem. *NumPy* og *Matplotlib*, da dette langt hen af vejen vil give de mest elegante løsninger. Når vi gør dette vil vi fra numpy importerer `array`, og så vil vi bruge `plot` fra Matplotlib. Se eventuelt *noterne fra Python i Mekrel*.

Når vi først har en liste af komplekse tal i SymPy, kan vi benytte `array` sammen med nøgleordet `dtype = complex` til at danne et numerisk numpy-array med komplekse tal.

Når vi først har en liste, kan vi lave lister over real-delene og de komplekse dele med hhv. `.real` og `.imag`, og så bruge `plot` fra matplotlib til at vise dette.

Vi viser her et eksempel, hvor vi visualiserer et komplekst tal opløftet i en stigende potens.

```
# Vi tager et eksempel med (1 + i/2)^n for n = 0, ..., 11
z = 1 + I/2

# Tom liste
zs = []
for n in range(12): # Vi kører nu igennem tolv gange (altså fra 0 til 11) og tilføjer␣
 ↪potensen til en liste
    zs.append(sp.expand(z ** n))

display(zs)
```

```
[1,
 1 + I/2,
 3/4 + I,
 1/4 + 11*I/8,
 -7/16 + 3*I/2,
 -19/16 + 41*I/32,
 -117/64 + 11*I/16,
 -139/64 - 29*I/128,
 -527/256 - 21*I/16,
 -359/256 - 1199*I/512,
 -237/1024 - 779*I/256,
 1321/1024 - 6469*I/2048]
```

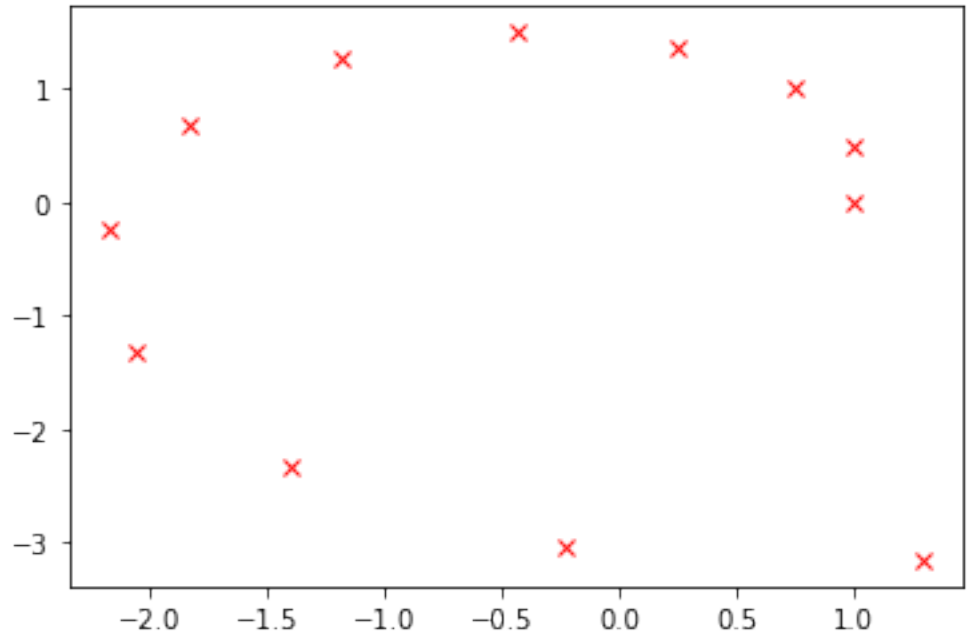
Vi kan nu lave `zs` til et numerisk array ved brug af numpy:

```
from numpy import array
z_numerisk = array(zs, dtype = complex)
```

Nu kan vi plotte `z_numerisk.real` af x-aksen og `z_numerisk.imag` af y-aksen. Vi importerer *Matplotlib* til dette formål og kalder `plt.plot`. Vi fortæller desuden at vi vil have "x" som markeringer og have en rød farve.

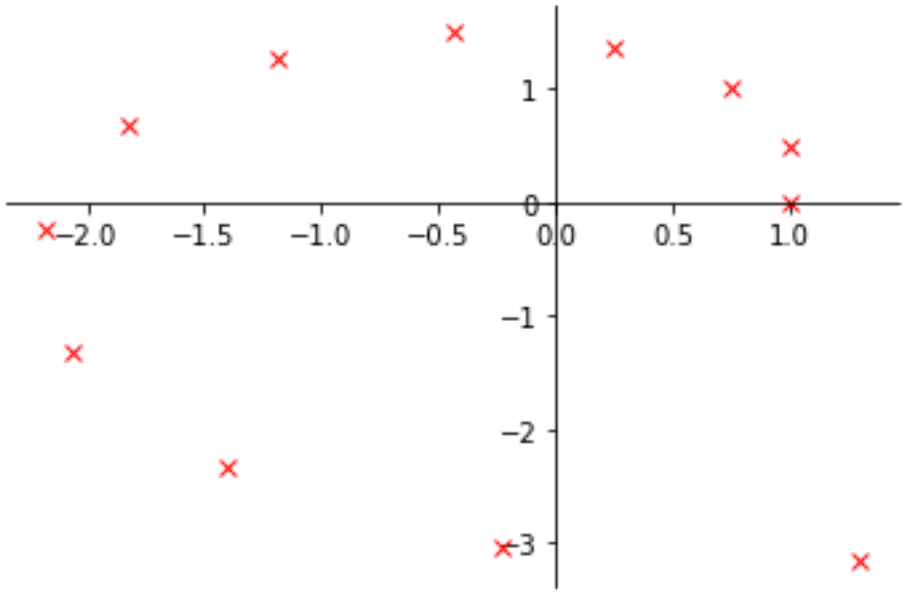
```
import matplotlib.pyplot as plt

plt.plot(z_numerisk.real, z_numerisk.imag, "x", color = "red")
```

```
[<matplotlib.lines.Line2D at 0x7f75ce012970>]
```



Ønsker man at lave plottet med "de klassiske akser", der krydser hinanden i (0,0), kan man gøre følgende:

```
# Samme plot som overstående
plt.plot(z_numerisk.real, z_numerisk.imag, "x", color = "red")

# Ryk venstre akse til midten
plt.gca().spines['left'].set_position("zero")

# Skjul den højre
plt.gca().spines['right'].set_color(None)

# Gentag for nederste og toppen
plt.gca().spines['bottom'].set_position("zero")
plt.gca().spines['top'].set_color(None)
```


1
0
−1
−2
−3
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0