Kompendium MAT102
Praktiske anvendelser og numeriske metoder i Python

Anita Gjesteland , Erik A. Hanson
Amir Hashemi

Februar 2022

Forord

Matematiske problemer blir ofte motivert ved hjelp av eksempler fra anvendelse i dagliglivet eller vitenskapen. Det kan hjelpe på forståelsen av matematikken og motivere oss når alt virker meningsløst. Men eksemplene vi har jobbet med til nå er gjerne over-forenklede og lite relevante i praksis. De inneholder små, enkle tall som gjør regningen enklere og tar ikke høyde for at virkeligheten aldri er akkurat slik vi tror.

I dette kompendiet skal vi ta for oss hvordan matematikk faktisk brukes i praksis i ekte og realistiske situasjoner. Vi skal bruke mye av matematikken vi har lært i tidligere kurs men i tillegg trenger vi verktøy for å håndtere ekte data og observasjoner og vi trenger begreper som lar oss ta høyde for usikkerhet og variasjon i observasjonene. Vi kommer til å ha store tabeller og lister med tall og vi kommer til å lage ulike plott og grafer. For å gjøre dette på en effektiv måte må vi også lære om hvordan vi kan få datamaskinen til å regne av seg selv. Altså hvordan bruke programmering til å laste inn data, løse likninger eller andre matematiske problemer, før et resultat lagres eller presenteres i form av et plott eller en oversikt.

Den matematiske teorien i dette kurset finner vi i hovedsak i læreboken Gulliksen m.fl ([2]). Dette kompendiet inneholder derfor i grove trekk praktiske aspekter rundt programmering og statistikk i tillegg til litt teori om hvordan løse ligninger med programmering. Det å bruke programmering til å løse matematiske problemer er egentlig et fagfelt i seg selv og kalles numerisk matematikk.

Mye av innholdet i dette kompendiet er hentet direkte fra "Numeriske metoder i MATLAB Tillegg til pensum i MAT102" av Jon Eivind Vatne [4]. Enkelte tilpasninger er gjort i forbindelse med at programmeringsspråk er endret fra MATLAB til Python. Vi vil takke Jon Eivind for å ha gjort alt materiell knyttet til dette arbeidet tilgjengelig for oss.

Innhold

Kapittel 1

Introduksjon til Python

I dette kapittelet skal vi se på ulike aspekter ved programmering i Python. Siden vi bare har gjort et utvalg av funksjonaliteter og elementer, kan det hende du møter på spørsmål som ikke kan besvares av denne introduksjonen. Men heldigvis finnes det mye og god dokumentasjon på hvordan ting skal gjøres i Python. Dokumentasjonen kan vi finne på nett, og det kan være lurt å øve seg på å bruke denne. Teorien vi presenterer i dette kapittelet er for det meste hentet fra denne dokumentasjonen, og dere kan finne den her: `https://www.python.org/doc/`.

1.1 Oppbygging

Når vi nå skal starte denne introduksjonen, starter vi med å forklare hvordan vi skal behandle kodene våre slik at PCen kan utføre oppgavene vi ber den om. I denne delen vil vi ha med noen eksempelkoder, og det er ikke meningen at du trenger å forstå dette enda - forklaringene på selve innholdet i koden vil komme etter hvert.

1.1.1 Script

Det er mulig å kjøre Pythonkode direkte i konsollvinduet i Spyder. Det som skjer da, er at koden blir utført med en gang den blir skrevet. For veldig små og enkle ting, for eksempel hvis vi bare skal regne ut et regnestykke som $4 + 3$, så fungerer dette fint. Etter hvert som vi skal kode lengre ting vil det derimot være enklere å skrive koden i et *script*. Et script er en fil som inneholder kode, men operasjonene i koden blir ikke utført før vi lagrer og kjører filen. Det vil si at dersom jeg skriver inn 1+1 i et script, så skjer det ingenting før jeg kjører scriptet. Her er et eksempel på et script som er skrevet for å regne ut løsningene til andregradslikningen $x^2 - 2x + 1 = 0$:

```
"""Script for aa regne ut losningen til den gitte andregradslikningen.
    """

# Importerer nodvendig pakke
import numpy as np

# Definerer koeffisientene
a = 1
b = -2
c = 1

# Regner ut de to rottene
x1 = (-b + np.sqrt(b**2 - 4*a*c))/(2*a)
x2 = (-b - np.sqrt(b**2 - 4*a*c))/(2*a)

print('Losningene er', 'x1 =', x1, 'og x2 =', x2)
```

Dersom vi kjører dette scriptet vil vi få `Losningene er x1 = 1.0 og x2 = 1.0`. Som vi ser fra kodesnutten, er det forskjellige elementer med i scriptet vårt. Disse elementene er ofte med når vi koder, og vi forklarer derfor kort hva de er.

i) *Kommentarer:* Det første vi møter i koden over, er en kommentar. En kommentar starter enten med en hash, og varer ut linjen, eller skrives inne i triple anførselstegn, som første linje i eksempelkoden over. Ved triple anførselstegn kan kommentaren vare over flere linjer. Dette er tekst eller kode som ignoreres når koden kjøres. Det er lurt å ha med kommentarer underveis i koden, fordi det gjør den enklere å lese, og det kan være til hjelp når du senere ser tilbake på den.

ii) *Pakker:* Det neste som skjer i koden, er at vi importerer en nødvendig pakke. Her importerer vi pakken `numpy` som `np`, slik at vi kan bruke funksjonen `sqrt` som tilhører pakken.

iii) *Variabler:* Videre definerer vi de tre koeffisientene a, b og c. I Python kalles disse *variabler*, og likhetstegnet angir verdien deres. Også x_1 og x_2 som vi så regner ut, er variabler, og vi ser at i utregningen brukes a, b og c som har verdiene vi allerede har gitt dem.

iv) *Streng:* En streng er en sekvens av tegn som er ment å tolkes av Python som tekst.

Mer informasjon om de ulike elementene kan du finne her: `https://docs.python.org/3/tutorial/introduction.html#id4`

1.1.2 Funksjoner

(`https://https://docs.python.org/3/tutorial/controlflow.html#defining-functions`)

Når vi har lært oss å lage script, er steget videre å lage såkalte *funksjoner*. Dette kan være nyttig dersom vi flere ganger i løpet av et script trenger å regne ut den samme tingen. Det vi gjør da, er at vi lager en funksjon av den delen av koden som skal brukes flere ganger. For eksempel kan vi ta utgangspunkt i at vi skal regne ut volumet av kjegler med ulik radius og høyde.

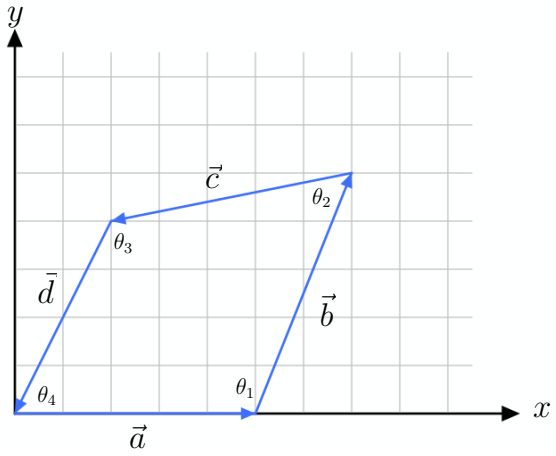
```
def volum_av_kjegle(radius, hoyde):
    """ Funksjon som regner ut volumet av en kjegle med radius og hoyde
        som gitt av inputparametrene."""

    import numpy as np

    volum = (np.pi*(radius**2)*hoyde)/3

    return volum
```

Skriver vi så `volum_av_kjegle(2,3)` i scriptet får vi ut volumet av kjeglen med radius = 2 og høyde = 3. En funksjon starter med `def` etterfulgt av funksjonsnavnet der input-parameterne er gitt i parantes. I dette tilfellet har vi spesifisert at parametrene er radius og høyde, og det betyr at vi må gi funksjonen verdier for disse parametrene når vi vil bruke den.

Legg merke til at etter at vi har definert funksjonen, så følger resten av koden med innrykk. Dette er viktig i Python, og all kode som tilhører funksjonen skal ha et innrykk. Dersom vi skriver en linje etter funksjonen uten innrykk vil ikke denne være en del av funksjonen vår.

Så utføres utregningen av volumet, før vi tilslutt har med `return volum`. Dette gjør vi for å kunne bruke svaret funksjonen gir, videre i koden.

Akkurat i denne oppgaven kunne vi like gjerne regnet volumet direkte i koden ved å skrive inn formelen for volum av en kjegle. Altså har vi ikke tjent så mye på å definere funksjonen. Men, vi trenger ikke å lete etter et veldig avansert eksempel før vi oppdager nytten av funksjoner. Ta nå utgangspunkt i at vi har fått i oppgave å finne vinklene i firkanten i figuren under:



Vi får oppgitt at $\vec{a} = [5, 0]$, $\vec{b} = [2, 5]$, $\vec{c} = [-5, -1]$, $\vec{d} = [-2, -4]$. Vinkelen, θ mellom to vektorer, $\vec{u}$ og $\vec{v}$ kan vi finne fra skalarproduktet, som er gitt ved

$$\vec{u} \cdot \vec{v} = |\vec{u}| \cdot |\vec{v}| \cdot \cos\theta.$$

Vi ser at for å kunne finne én vinkel, er det flere mellomregninger som må gjøres. Først må vi finne prikkproduktet $\vec{u} \cdot \vec{v}$, og deretter må vi finne lengdene av vektorene. Siden vi vil gjøre dette 4 ganger, vil vi kunne spare tid på å skrive en funksjon som regner ut dette for et hvert par av vektorer som vi gir funksjonen.

```
import numpy as np


def vinkel_mellom_vektorer(vek1,vek2):
  """ Funksjon som regner ut vinkelen mellom de to vektorene vek1 og
     vek2 """


  # Finner lengden av vektorene
  lengde_vek1 = np.sqrt(vek1[0]**2 + vek1[1]**2)
  lengde_vek2 = np.sqrt(vek2[0]**2 + vek2[1]**2)

  # Finner prikkproduktet mellom vektorene
  prikkprod = vek1[0]*vek2[0] + vek1[1]*vek2[1]

  # Regner ut vinkelen mellom vektorene
  theta = np.degrees (np.arccos(prikkprod/(lengde_vek1*lengde_vek2)))

  return theta

# Definerer vektorene vi har oppgitt
a = np.array([5,0])
b = np.array([2,5])
c = np.array([-5,-1])
d = np.array([-2,-4])

# Finner de ulike vinklene (husk aa snu en av vektorene slik at de har
    samme utgangspunkt)
theta1 = vinkel_mellom_vektorer(-a,b)
theta2 = vinkel_mellom_vektorer(-b,c)
theta3 = vinkel_mellom_vektorer(-c,d)
theta4 = vinkel_mellom_vektorer(-d,a)
```

```
print(theta1, theta2, theta3, theta4)
```

I dette eksempelet ser vi at koden vår hadde blitt mye lengre, og det ville vært mer tungvint, dersom vi måtte ha gjort alle utregningene som gjøres inne i funksjonen vår, 4 ganger. I tillegg øker det sjansen for feil, siden vi må skrive riktig alle de 4 gangene. Når vi på et relativt enkelt eksempel som dette kan se nytten av funksjoner, kan du tenke deg fordelen dette gir ved mer avanserte og omfattende utregninger!

Når man bruker programmering, for eksempel i matematikk, er det kanskje noen kodesnutter som går igjen i flere av scriptene vi lager. Vi kan for eksempel tenke oss at vi i løpet av et semester skal bruke abc-formelen mange ganger, og i stedet for å måtte skrive den samme koden hver gang, lar Python oss lett kunne definere denne funksjonen én gang, for så å bruke den igjen senere. I neste delseksjon kommer forklaringen på hvordan vi kan gjøre dette.

1.1.3 Importering av ulike pakker

(`https://docs.python.org/3/tutorial/modules.html`)

En pakke i Python er en samlig av moduler, hvor en modul er en fil som inneholder Python-definisjoner. Når vi bruker en modul fra en pakke, hentes denne ved hjelp av punktum: `pakke.modul`, henter modulen `modul` fra pakken `pakke`. Det finnes mange ulike pakker man kan benytte i Python, og vi skal kun se på pakkene `numpy` og `matplotlib` her. Som du kanskje har lagt merke til i eksempelkodene over, starter scriptene med `import numpy as np`. Det vi gjør da, er at vi importerer pakken *numpy* som *np*, slik at funksjonene i pakken blir tilgjengelige for oss. Hvis vi nå for eksempel skal bruke pi, må vi skrive: `np.pi`.

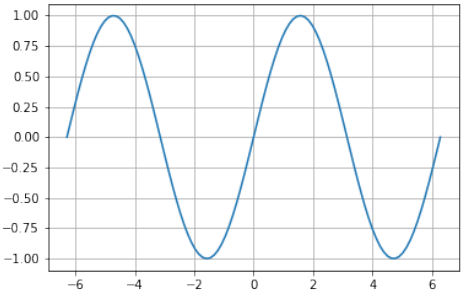
Det er også denne funksjonaliteten vi benytter oss av dersom vi har en kodesnutt som skal brukes gjennom hele semesteret. Hvis vi har definert en funksjon i et annet script, kan vi importere den til andre script. Som et eksempel kan vi tenke oss at det forrige scriptet het `FirkantVinkler`. Hvis vi så vil importere funksjonen som regnte ut vinkelen mellom to vektorer, til et nytt script, gjøres dette ved å skrive `from FirkantVinkler import vinkel_mellom_vektorer`, og så vil funksjonen være klar til bruk i det nye scriptet.

Merk.
Pakken `numpy` er en pakke for vitenskapelige beregninger, og dokumentasjonen kan du finne her: `https://numpy.org/doc/stable/`. Pakken `matplotlib` lar oss visualisere i Python. Dokumentasjonen kan du finne her: `https://matplotlib.org/contents.html`.

Under følger et kort eksempel der vi importerer noen pakker, og bruker dem til en liten beregning

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

# Her skal vi definere en funksjon, og deretter plotte den

# Deler inn x-aksen
x = np.linspace(-2*np.pi,2*np.pi, num=100)
# Regner ut funksjonsverdiene for de gitte x-verdiene
f = np.sin(x)

# Plotter grafen
plt.plot(x,f)
```

Funksjonen `np.linspace(start, end, num = N)`, deler intervallet $[start, end]$ inn i N jevnt fordelte punkter. Resultatet av dette scriptet er denne grafen:



1.2 Nyttige verktøy for kodingen

Her skal vi gå gjennom noen verktøy som ofte brukes i kode for å få utført de oppgavene vi vil PCen skal løse.

1.2.1 For-løkke

Hvis vi vil at koden skal repetere operasjoner et gitt antall ganger, er for-løkker en måte å gjøre dette på. Vi kan for eksempel tenke oss at vi vil regne ut funksjonsverdien til en funksjon for noen gitte punkter i definisjonsmengden. Under kommer et eksempel som demonstrerer hvordan dette kan kodes:

```
""" Kode som viser bruk av for lokker ved aa regne ut
funksjonsverdiene x^2 for gitte x """

# Definerer x-verdiene vi vil regne ut funksjonsverdien for
x = np.array([0,2,4,6,8,10])
f = np.zeros(len(x))

# Starter loopen
for i in range(len(x)):
  # Beregner funksjonsverdien for hver av verdiene til x og legger de
     til i f
  f[i] = x[i]**2

print(f)
```

I denne kodesnutten, importerer vi først pakken `numpy` (som vi har sett tidligere), før vi så definerer en vektor, $x = [0, 2, 4, 6, 8, 10]$. Deretter definerer vi en vektor med bare 0-ere som vi kaller f som vi vil lagre funksjonsverdiene våre i senere. Funksjonen `len`, gir oss lengden av argumentet den tar, og ved å definere f på denne måten, garanterer vi at den er en vektor med samme lengde som x. Deretter starter for-løkken. Her har vi brukt funksjonen `range` [1] for å lage en liste med heltall som er like lang som vektoren x.

Merk.
Python starter indeksing på 0, det vil si at det første elementet i en liste, vektor eller liknende, vil ha posisjon 0.

Når løkken over starter vil $i = 0$, og siden 0 vil være i vår range-liste, vil kommandoene inne i for-løkken utføres. Etter alle operasjonene er utført, vil $i = 1$, og kommandoene i for-løkken utføres på ny, men nå

[1] Denne funksjonen kan ta inn argumentene *start, end* og *step*, men det første og siste argumentet blir satt til 0 og 1, respektivt, dersom vi ikke spesifiserer noe her. *Range*-funksjonen lager en liste med tallene fra og med *start* opptil *end*, med steglengde *step*. For eksempel vil `range(5)`, gi oss en liste som inneholder de 5 elementene $0, 1, 2, 3, 4$. For mer informasjon om denne funksjonen, se `https://docs.python.org/3/tutorial/controlflow.html#the-range-function`.

med ny i-verdi. Slik fortsetter vi så lenge verdien til i er i range-listen. Legg igjen merke til at etter at for-løkken er startet, er det innrykk på de neste linjene som tilhører løkken.

Effektiv og kompakt kode

Det er mange måter å bruke Python på. De eksemplene vi kommer med i dette kompendiet er ikke alltid satt opp på de mest ideelle eller effektive måten. I noen tilfeller ønsker vi heller å ta et begrep om gangen. Selv om vi ønsker å holde oss til `NumPy`, så utnytter vi heller ikke de innebygde egenskapene fullt ut. Som et eksempel vil koden med for-løkken kunne skrives:

```
x = np.array([0,2,4,6,8,10])
#Utnytter egenskapene til numpy array og regner ut alle verdiene
    for x samtidig.
f = x**2
print(f)
```

Hvis vi i stedet for å bruke en `NumPy`-array hadde brukt en vanlig liste kunne vi også fått til det samme med denne koden:

```
x = [0,2,4,6,8,10] #valig liste
f = [] #tom liste
# Starter loopen
for a in x:
# Beregner funksjonsverdien for hver av verdiene til x og legger
    de til i f
 f.append(a**2)
print(f)
```

Den mest kompakte måten vil imidlertid være:

```
f = [a**2 for a in x])
print(f)
```

Hvilken løsning vi velger vil kunne ha betydning for hvor raskt datamaskinen regner, hvor mye minne den bruker og, ikke minst, hvor lett det er for andre å lese koden vår. Prøv deg frem.

1.2.2 If-løkke

En annen type løkke som er nyttig å kjenne til, er *if-løkken*. Noen ganger ønsker vi kanskje å utføre operasjoner bare på elementer som oppfyller noen krav. Da kommer denne typen løkke til nytte. La oss ta for oss en kode som undersøker om en inputvariabel er et partall eller et oddetall. Funksjonen under viser et eksempel på dette.

```
def partall_oddetall(tall):
  """ Funksjon som sjekker om inputvariabelen 'tall' er et partall
      eller et oddetall. Input maa vaere et heltall. """

  if tall%2 == 0:
    print(tall, 'er et partall')
  else:
    print(tall, 'er et oddetall')
```

Løkken starter med `if`, og hvis kravet som følger er oppfylt, utføres operasjonene som følger med innrykk. Videre bruker vi `else`, som sier at dersom det første kravet ikke er oppfylt, så skal operasjonene som følger `else` utføres. Her står % for *modulo*-operatoren. Den regner ut resten av divisjonen tall/2. Siden

et partall er delelig med 2, vil resten alltid bli 0 for partall. Derfor vet vi at dersom det første kriteriet ikke er oppfylt, så må tallet være et oddetall.

1.2.3 While-løkke

Andre ganger vil vi gjøre bestemte utregninger så lenge en betingelse vi har satt på forhånd, er oppfylt. Vi skal vise bruken av denne type løkke ved hjelp av et eksempel.

Eksempel 1.2.1:
Bektrakt summen

$$\sum_{n=1}^{\infty} \frac{1}{n}.$$

Finn ut hvor mange ledd som trengs for å få summen over 15.

Løsning:

```
# Definerer n og Sum ved start
n = 1
Sum = 0

# Vil utfore summeringen saa lenge summen er mindre eller lik 15
while Sum <= 15:
  Sum += 1/n
  n += 1

print('Summen passerer 15 etter', n, 'ledd. Da er summen', Sum)
```

Inne i while-løkken har vi brukt funksjonaliteten `Sum += 1/n`. Denne operasjonen legger til $1/n$ til `Sum`, og endrer `Sum` til det nye resultatet. For at en while-løkke skal fungere, må noe i betingelsen vi setter, endres. Her ser vi at så lenge summen er mindre eller lik 15, så vil vi legge til et ledd i summen, og den vil bli større. Neste gang vi går inn i løkken vil derfor `Sum` ha endret seg, og koden vurderer på nytt om betingelsen er oppfylt. Dersom den fremdeles er det, vil den utføre operasjonene på ny.

1.3 Python som kalkulator

Vi har i dette kapittelet hele veien knyttet kodene våre opp mot matematikken, men i denne seksjonen har vi med et par merknader det er greit å være klar over når vi bruker Pythons kalkulatorfunksjoner. Vi har i eksemplene våre allerede utnyttet at Python kan utføre operasjoner som en kalkulator ville. For eksempel har vi addert, subtrahert, multipliser og dividert tall, og vi har brukt matematiske funksjoner som cos og kvadratrot. De matematiske funksjonene som ln, cos, sin og kvadratrot må derimot importeres fra de nødvendige pakkene før de kan inngå i beregninger slik vi ville gjort på en kalkulator.

- I Python vil `log` tilsvare den *naturlige* logaritmen, altså det vi gjerne kaller ln. Skal vi bruke den briggske logaritmen i Python, må vi spesifisere det ved å skrive `log10`.
- Når vi skal opphøye noe, i Python, bruker vi doble gangetegn for å indikere dette. For eksempel vil x^2 i Python skrives slik:

```
x**2
```

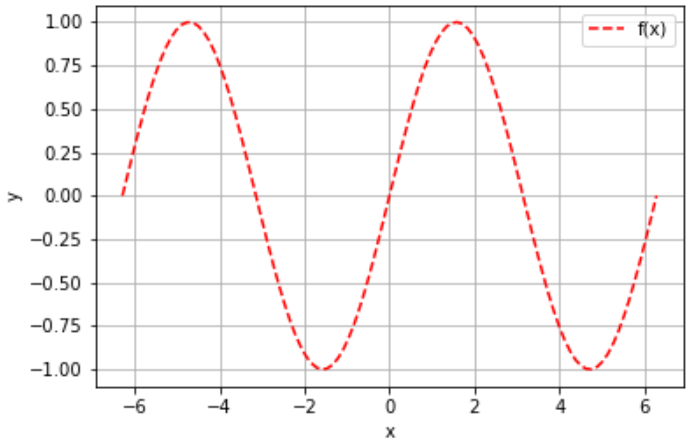
- `abs(input)` gir oss absoluttverdien av inputparameteren, `input`.

1.4 Grunnleggende plotting

Vi har allerede brukt en pakke for plotting i noen av eksemplene vi har presentert så langt, men i denne seksjonen vil vi gå gjennom mer nøye hva vi gjør. Informasjonen i denne seksjonen er hentet fra dokumentasjonen til `matplotlib`, som du kan finne her: `https://matplotlib.org/contents.html`.

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

x = np.linspace(-2*np.pi,2*np.pi, num = 100)
f = np.sin(x)

plt.plot(x,f, '--')
plt.grid()
plt.xlabel('x')
plt.ylabel('y')
plt.legend(['f(x)'])
plt.savefig('MinPythonMappe/eksempel_plott.png')
```

Kjører vi dette scriptet, vil følgende figur plottes *og lagres*.



Elementene vi har tatt i bruk her er følgende:

i) `plt.plot(x,f,'--')`: Dette er kodesnutten som plotter funksjonen. Vi ser at argumentene vi gir er punktene på x-aksen, de tilhørende funksjonsverdiene og til slutt har vi også tatt med `'--'`. Det siste argumentet skifter utseendet på grafen til å være en stiplet i stedet for en sammenhengende graf. Flere argumenter kan også spesifiseres, se dokumentasjonen til matplotlib.

ii) `plt.grid()`: Denne legger på et gitter bak grafen, slik vi er vandt til å ha det i rutebøkene våre.

iii) `plt.xlabel()` og `plt.ylabel()`: Setter navn på aksene til det vi spesifiserer som argument.

iv) `plt.legend()`: Denne funksjonen legger til en tekstboks som vi kan bruke til å beskrive hva grafen heter.

v) `plt.savefig()`: Denne funksjonen gjør at vi kan lagre figuren direkte når vi kjører koden. Her må vi legge til riktig path inne i en streng, slik at bildet blir plassert i den mappen vi ønsker.

Test deg selv:
Kan du finne ut hvordan man endrer farger på grafen?

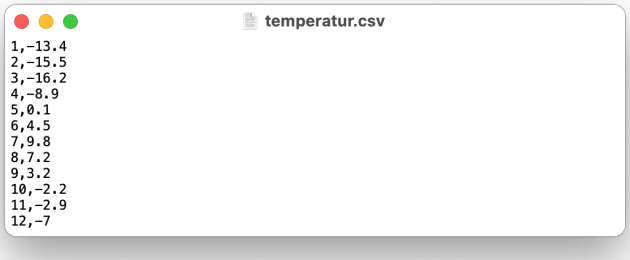
1.5 Laste inn og plotte data

I avsnittene over har vi sett på hvordan vi regner med matematiske funksjoner og plotter dem. I de fleste anvendelser vil beregningene våre ta utgangspunkt i tall hentet fra observasjoner eller eksperimenter. Det kan ofte bli lange lister med tall og data. Vi skal nå se på hvordan vi kan hente inn reelle data til beregningene våre. Det finnes ulike pakker i Python som lar oss laste inn og håndtere data. Vi skal holde oss til funksjonaliteten som alt ligger i `numpy`. I tillegg skal vi benytte noe elementær statistikk. Det matematiske innholdet vil, for de fleste, være kjent fra før, så vårt fokus vil være på praktiske anvendelser.

1.5.1 Fra komma-separert fil til array

Observasjoner og eksperimentelle data kan lagres og redigeres i forskjellige filformater. Det mest vanlige er gjerne et regneark. Regnearkene er praktiske for oss som brukere, men inneholder imidlertid ofte mye ekstra informasjon som vi ikke trenger. Når vi skal få laste informasjonen inn i python kan det være vanskelig å få koden vår til å ignorere den informasjonen som ikke trengs. Datafiler som er ment for å leses av et dataprogram er derfor ofte organisert som 'komma-separerte verdier' (På engelsk: csv, comma-separated values). Disse filene tilsvarer et svært enkelt regneark der hver tekstlinje er en rad og verdiene for de ulike søylene er separert med komma.

Som et eksempel skal vi nå se på filen 'temperatur.csv'. Hver linje i filen er en temperatur-måling for Svalbard lufthavn i 2020[2]. Hver linje inneholder to tall separert med komma. Det første tallet er måneden $(1-12)$ og det andre tallet er temperaturen.



Vi laster inn tallverdiene fra filen som en variabel i Python (en array av float-verdier) vha `numpy.loadtxt`. I eksempelet under laster vi inn en kolonne av gangen spesifisert med `usecols`. Det finnes andre mer generelle måter å laste data fra en CSV-fil, men vi vil her holde oss til `numpy`.

```
import numpy as np

temperaturer = np.loadtxt('temperatur.csv',delimiter=',',usecols=(1))
tidspunkt = np.loadtxt('temperatur.csv',delimiter=',',usecols=(0))
```

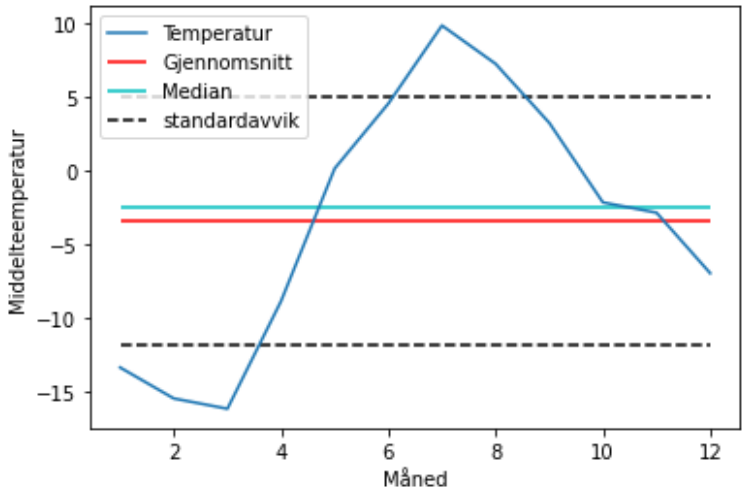
Vi kan nå plotte temperaturene på samme måte som tidligere

```
import matplotlib.pyplot as plt

plt.plot(tidspunkt,temperaturer)
plt.xlabel('Maned')
plt.ylabel('Middelteemperatur')
```

[2]Data hentet fra https://seklima.met.no/observations/ og er formatert for å få en enkel csv-fil

Vi kan nå enkelt beregne årlig middeltemperatur enten vha et vanlig gjennomsnitt eller som en median. Vi kan også beregne f.eks. standardavviket til temperaturdataene. I koden under gjør vi disse beregningene og visualiserer verdiene i et enkelt plot

```
gjennomsnitt = np.mean(temperaturer)
median = np.median(temperaturer)
standardavvik = np.std(temperaturer)

plt.hlines(gjennomsnitt,1,12,colors='r')
plt.hlines(median,1,12,colors='c')
plt.hlines(gjennomsnitt+standardavvik,1,12,colors='k',linestyles='--')
plt.hlines(gjennomsnitt-standardavvik,1,12,colors='k',linestyles='--')
plt.legend(['Temperatur','Gjennomsnitt','Median','standardavvik'])
```



1.6 Laste inn og analysere et større dataset

Eksempelet med filen 'temperatur.csv' i avsnittet over er fortsatt ganske enkelt og idealisert. Vi skal nå se på et mer realistisk eksempel og praktiske problemer som typisk oppstår når vi skal analysere reelle data.

Temperaturene vi har sett på til nå er hentet fra en lengre data-serie med temperaturmålinger helt tilbake til oktober 1898. Dette er målinger som bl.a. blir brukt til forskning [5]. Om man laster ned disse målingene fra Norsk klimaservicesenter får man en stor, uoversiktlig csv-fil. ('table.csv' er et eksempel på en slik fil). Her er det flere ting vi må håndtere:

table.csv

```
Navn;Stasjon;Tid(norsk normaltid);Homogenisert middeltemperatur (mnd)
Svalbard Lufthavn;SN99840;09.1898;3,8
Svalbard Lufthavn;SN99840;10.1898;-5,2
Svalbard Lufthavn;SN99840;11.1898;-11
Svalbard Lufthavn;SN99840;12.1898;-16,4
Svalbard Lufthavn;SN99840;01.1899;-14,8
Svalbard Lufthavn;SN99840;02.1899;-20,7
Svalbard Lufthavn;SN99840;03.1899;-23,1
Svalbard Lufthavn;SN99840;04.1899;-15,6
Svalbard Lufthavn;SN99840;05.1899;-6,7
Svalbard Lufthavn;SN99840;06.1899;4
Svalbard Lufthavn;SN99840;07.1899;8,2
Svalbard Lufthavn;SN99840;08.1899;5,6
```

i) Første rad er beskrivelse av hver enkelt søyle. Dette trenger vi ikke laste inn.

ii) Siste rad (ikke synlig i bildet over) er tilleggsinformasjon om copyright mm som vi ikke trenger å laste inn.

iii) Det er flere søyler med informasjon vi ikke trenger i det hele tatt.

iv) Komma er brukt som desimaltegn (som er vanlig på norsk) i stedet for skilletegn.

v) Semikolon er brukt som skilletegn.

I koden under laster vi inn temperaturer fra 'table.csv' til en numpy array. Se kommentarer i koden for forklaring på de ulike stegene. Det finnes også andre pakker som er egnet for å laste større, kompliserte dataset, men det blir ikke diskutert her.

```
import numpy as np

""" Bruker np.loadtxt til aa laste inn array med data. Her er det
mulig aa sette delimiter=';' for aa benytte semikolon til skilletegn.
skiprows=1 lar oss hoppe over forste rad. Vi vet at informasjon
om tid ligger i tredje kolonne og setter derfor usecols=(2)
(begynner aa telle paa 0). Siden tidspunktene er gitt i et format
mm.aaaa onsker vi aa laste det inn som en streng og ikke som et
desimaltall (float). Velger derfor dtype = 'str'). """
tidspunkt = np.loadtxt('table.csv', delimiter=';', \
                    skiprows=1, usecols=(2),dtype = 'str')


""" Laster temperatur-data paa akkurat samme maate. Disse tallene
ligger i andre kolonne, saa usecols=(3). Selv om dette er
desimaltall, saa klarer ikke numpy aa lese tallene fordi vi har
komma som desimal-tegn. Leser tallene derfor inn som 'str' """
tempr = np.loadtxt('table.csv', delimiter=';', \
                   skiprows=1, usecols=(3),dtype = 'str')


"""For aa kunne tolke temperaturene som desimaltall, bytter vi ut
alle komma med punktum vha np.char.replac. Dette er som
'finn-og-erstatt' i en teksteditor"""
tempr = np.char.replace(tempr,',','.')

""" Den siste linjen i csv-filen horer ikke til i datasettet
(her er info om opphavsrett mm). Denne linjen gir likevel et
element i arrayen, som vi maa fjerne. Argumentet [0:-1] tilsier
at vi henter ut det forste til det nest siste elementet. """
tempr = tempr[0:-1]
tidspunkt = tidspunkt[0:-1]

"""Ber numpy tolke temperaturene som float i stede for str. Det
gaar greit naar vi har punktum som desimaltegn."""
tempr = tempr.astype(np.float)
```

Som et eksempel på en enkel analyse, skal vi nå plotte tidsserien og beregne middelverdien i forskjellige 20-års intervaller. Siden tidsserien er så lang vil selv plottingen by på noen praktiske problemer. Det er standard å la hvert enkelt datapunkt ha et merke på x-aksen. Det vil det ikke være plass til i dette plottet. Vi kan håndtere markeringen på aksene med `matplotlib.ticker`. Det vil i hvert fall være plass til å legge inn en markering hvert tiende år, altså hver 120. måned. Datasereien begynner i september og det kan være penere å begynne markeringen i januar. Vi legger derfor også inn en forskyvning på 4 måneder.

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

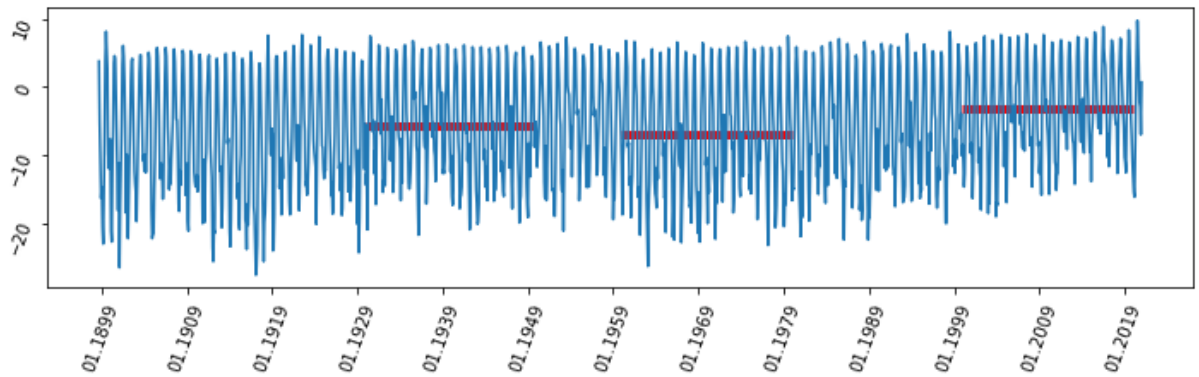
```
import matplotlib.ticker as ticker

# oppretter en figur (med avlang form)
plt.figure(figsize=(12, 3))

#oppretter et subplott i figuren. Vi skal bare ha et plott, men vi
#onsker aa utnytte funksjionalitet fra subplot.
temp_plot = plt.subplot()

#legger inn selve tidsserien i den aktive figuren
plt.plot(tidspunkt,tempr)

#endrer makeringen paa x-aksen
temp_plot.xaxis.set_major_locator(ticker.IndexLocator(base=120, offset
    =4))

#roterer markeringen paa x-aksen 70 grader slik at den blir lettere aa
    lese
temp_plot.tick_params(labelrotation=70)
```

Videre kan vi beregne middeltemperaturen for tre ulike 20-års perioder. Vi velger den første perioden til å være 1930-1950, den andre til å være 1960-1980 og den tredje til å være 2000-2020. Med litt prøving og feiling kan vi komme frem til hvilke element i arrayen av temperaturer som tilsvarer disse tidspunktene. F.eks. får vi at `: tidspunkt[376]` returnerer `'01.1930'`. Det betyr at vi kan beregne den første middelverdien som `np.mean(tempr[376:615])` (der 616 er $376 + 12 \cdot 20$).

Kodesnutten under er en fortsettelse på det vi begynte på over, der vi beregner de tre middelverdiene og plotter dem som horisontale linjer med `matplotlib.pyplot.hlines`.

```
#beregner og plotter middelverdi for 1930-1950
m1 = np.mean(tempr[376:616])
plt.hlines(m1,376,616,colors='r',linewidth=5)

#beregner og plotter middelverdi for 1960-1980
m2= np.mean(tempr[736:976])
plt.hlines(m2,736,976,colors='r',linewidth=5)

#beregner og plotter middelverdi for 2000-2020
m3= np.mean(tempr[1216:1456])
plt.hlines(m3,1216,1456,colors='r',linewidth=5)
```

Koden returnerer følgende plott:



Vi kommer tilbake til dette eksemplet senere og vil da utføre en statistisk test for å avgjøre om det er en signifikant forskjell mellom de tre middelverdiene.

Test deg selv:
Kan du endre koden til å å beregne gjennomsnittet for større intervaller? F.eks. 50 år?
Refleksjon: Er det fornuftig og vitenskapelig redelig å velge intervaller akkurat slik man selv vil? Kan valget av intervaller påvirke hvordan vi tolker plottet?

1.1.
Skriv et script for å regne ut disse geometriske størrelsene:

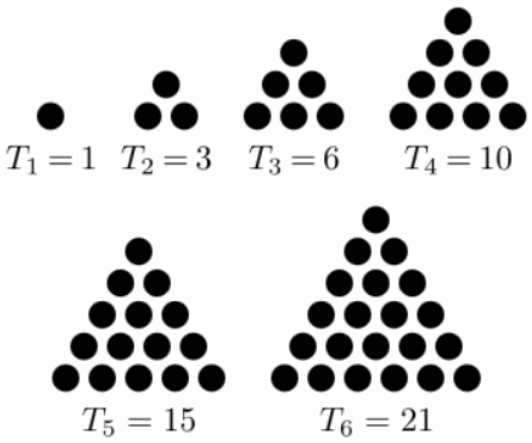
a) Arealet av en trekant med gitt grunnlinje og høyde.

b) Volumet av en boks med gitt lengde, bredde og høyde.

c) Arealet av en sirkelsektor med gitt radius og vinkel.

1.2.
Gjør Test deg selv"oppgaven i seksjon 1.4 i kompendiet.

1.3.
Trekanttallene T1, T2, T3 , ... finnes ved å telle punkter i likesidete trekanter på denne måten:



a) Hvordan kan man beregne T_n fra T_{n-1} fra dette mønsteret?

b) Bruk en for-løkke til å regne ut de første 100 trekanttallene.

1.4.
divergerer mot uendelig (det vil si at summen vokser ubegrenset, den konvergerer ikke mot noe tall).

a) Kan du finne hvor mange ledd som trengs for å få summen over 8 ?

b) Kan du finne hvor mange ledd som trengs for å få summen over 10 ?

c) Regn ut summen av de første 1.000.000 leddene.

1.5.
Løs lærebokens oppgave 2.1.3 i Python. Hint: Du kan bruke if.

1.6.
Løs lærebokens oppgave 4.2.1, ved å forsøke å tegne grafene i Python. Du trenger ikke løse alle bokstavpunktene. Merk at du ved å bruke tankegangen fra læreboken vil kunne få riktigere svar enn ved gjettingen ved hjelp av Python.

1.7.
I hvert spørsmål, bruk Python til å tegne funksjonene. Basert på grafen, velg en passende startverdi og finn et nullpunkt ved Newtons metode. Hvis funksjonen har flere nullpunkt, velg ett av dem.

a) $f(x) = e^x - 4x$

b) $g(x) = e^x + \cos x$

c) $h(x) = x + \ln x$

Oppgave 2.

Bruk Newtons metode på funksjonen

$f(x) = xe^{-x^2}$

Undersøk hva som skjer for varierende startverdier. Prøv spesielt med verdiene 0,5 og 0,51

Oppgave 3.

Bruk Newtons metode på funksjonen

f(x)=x2+1

Hvorfor blir resultatet som det blir?

Oppgave 4.

Ligningen

cos(x)

har akkurat en løsning.

Forklar hvorfor ligningen har minst en løsning. Bruk Newtons metode til å finne løsningen. Vi aksepterer en feilmargin på 10^{-6}. Forklar hvordan vi kan være sikre på at ligningen bare har en løsning.

1.8. **a)** Bruk numerisk derivasjon til å finne tilnærminger til de deriverte av funksjonene fra Oppgave 1 i x=2 og x=5.

b) Bruk numerisk derivasjon til å tegne grafene til de deriverte av funksjonene fra Oppgave 1.

Kapittel 2

Numerisk løsning av likninger

2.1 Newtons metode

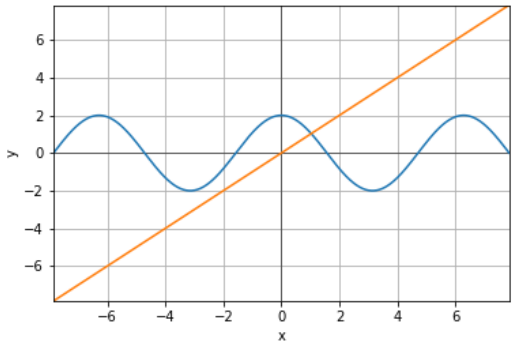
Newtons metode er en metode for å finne løsninger av likninger i én variabel. Som eksempel skal vi se på likningen

$$2\cos(x) = x. \tag{2.1.1}$$

De to grafene $y_1 = 2\cos(x)$ og $y_2 = x$ kan vi plotte i Python med denne koden:

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

# Definerer definisjonsmengden
x = np.linspace(-(5*np.pi)/2, (5*np.pi)/2, num = 1000)

# Definerer de to funksjonene
y1 = 2*np.cos(x)
y2 = x

# Plotter grafene til funksjonene
plt.plot(x,y1, x,y2)
plt.grid()
```

Dette gir oss de følgende grafene:



Merk.
Vi har lagt på litt mer i koden for å få ut plottet akkurat slik det ser ut i figuren. For eksempel har vi

gjort aksene tydeligere, og endret på grensene på aksene. Vi dropper det i kodesnutten for å redusere notasjonen.

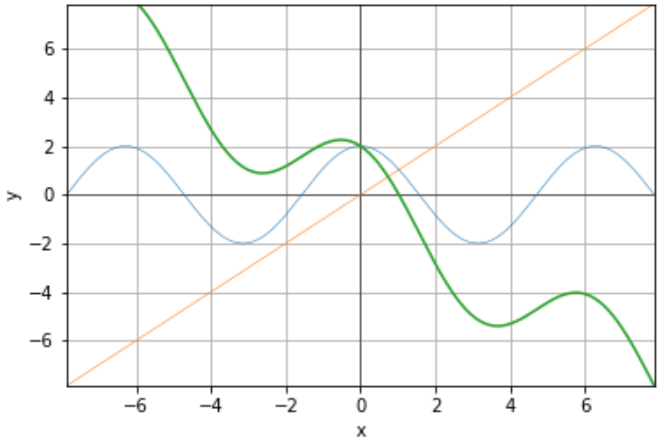
Vi ser at vi har en løsning for en x mellom 0 og 2. For å forenkle fremstillingen endrer vi på oppsettet, og søker nullpunkter til funksjoner heller enn skjæringspunkter mellom to funksjoner. Dette kan vi gjøre fordi

$$2\cos(x) = x$$

er det samme som

$$2\cos(x) - x = 0.$$

La derfor $f(x) = 2\cos(x) - x$. Grafen til denne funksjonen er den grønne grafen som vi har lagt oppå de to grafene fra forrige plott, i figuren under. Fra figuren ser vi at det ser ut som nullpunktet til $f(x)$ har samme x-verdi som skjæringspunktet mellom de to andre grafene (og det *har* det også).



Newtons metode tar utgangspunkt i den lineære tilnærmingen (tangenten) til $f(x)$ rundt et punkt x_n, som er gitt ved (se også læreboken [2] s. 177 eller Gulbrandsen, [4], avsnitt 5.3.)

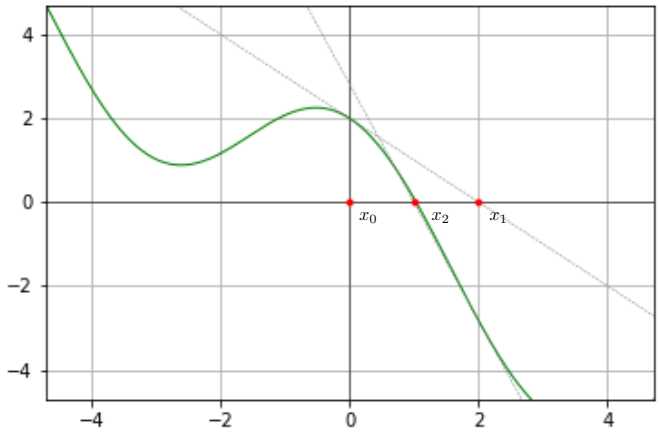
$$f(x) \approx f(x_n) + f'(x_n)(x - x_n).$$

Siden vi ønsker å tilnærme $f(x) = 0$, kan vi sette tilnærmingen over lik null. Det gir

$$\begin{aligned} 0 &= f(x_n) + f'(x_n)(x - x_n), \\ x &= x_n - \frac{f(x_n)}{f'(x_n)}. \end{aligned}$$

I figur 2.1.1 har vi zoomet inn på området vi er interessert i for å illustrerer metodikken. Vi tar utgangspunkt i et punkt x_0, regner ut hva likningen til tangenten til f er i dette punktet, og velger x_1 som nullpunktet til tangenten. For vårt tilfelle, kan det fra grafen se ut til å ikke gi en verdi som er så mye nærmere nullpunktet til f enn den verdien vi startet med, men hvis vi gjør en *iterasjon* til, ser vi at vi kommer mye nærmere. Ved å iterere videre kan vi treffe bedre og bedre.

Definisjon 2.1.1 (Newtons metode)**:**
Newtons metode forsøker å finne nullpunkter til funksjoner. Følgen $\{x_n\}$ vil forhåpentligvis nærme seg et nullpunkt.



Figur 2.1.1: Illustrasjon av Newtons metode

i) Start med et punkt x_0 som du velger i nærheten av der du tror du kan finne et nullpunkt.

ii) Finn det neste punktet fra formelen

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}.$$

iii) Fortsett med å finne nye punkter, x_i, der hvert punkt finnes fra det foregående med formelen

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

iv) Stopp når resultatet er "godt nok".

La oss bruke denne framgangsmåten på funksjonen vår $f(x) = 2\cos(x) - x$. Vi velger $x_0 = 0$ som startpunkt. Vi trenger også den deriverte av funksjonen vår, og den er

$$f'(x) = -2\sin(x) - 1.$$

Ved å kjøre følgende kodesnutt

```
import numpy as np

x0 = 0
x1 = x0 - (2*np.cos(x0)-x0)/(-2*np.sin(x0)-1)
x2 = x1 - (2*np.cos(x1)-x1)/(-2*np.sin(x1)-1)
x3 = x2 - (2*np.cos(x2)-x2)/(-2*np.sin(x2)-1)
x4 = x3 - (2*np.cos(x3)-x3)/(-2*np.sin(x3)-1)

print('x0= ',x0, 'x2= ', x1, 'x2= ', x2, 'x3= ', x3, 'x4= ', x4)
```

får vi disse verdiene

```
x0 =  0
x1 =  2.0
x2 =  0.9951398408944176
x3 =  1.030107306911052
x4 =  1.0298665403159568
```

Om vi sjekker den opprinnelige likningen (2.1.1) med $x = x_4$ får vi:

$$2\cos(1.0298665403159568) = 1.02986651.$$

Altså stemmer likningen med 7 desimalers nøyaktighet.

Men koden over er ikke særlig effektiv. Vi trenger ikke å lagre alle de foregående verdiene av x_n. Siden vi kun er interessert i den siste verdien, kan vi heller overskrive den foregående verdien med den nye. Metoden skal stoppe når vi er kommet "nært nok" 0, og det er best at Python vurderer dette direkte. Dette kan vi gjøre på følgende måte:

```
import numpy as np

x = 0

while abs(2*np.cos(x) - x) > 0.0000001:
  x = x - (2*np.cos(x)-x)/(-2*np.sin(x)-1)

print("x = ", x)
```

2.1.1 Skjæring, kontinuitet og Newtons metode

En typisk oppgave om Newtons metode består i å forklare hvorfor vi kan være sikre på at en funksjon har et nullpunkt, og deretter bruke Newtons metode til å prøve å finne det. Til dette kan vi bruke *skjæringssetningen* (se lærebokens teorem 4.3.1) som sier at en kontinuerlig funksjon som skifter fortegn på et intervall i definisjonsmengden, må ha et nullpunkt der.

Eksempel 2.1.1:
Se på funksjonen

$$f(x) = e^x + x.$$

Forklar hvorfor funksjonen har et nullpunkt i intervallet $(-1, 0)$, og bruk Newtons metode til å finne nullpunktet.

Løsning: Vi sjekker at $f(-1) = e^{-1} - 1 \approx -0.63 < 0$ og at $f(0) = e^0 + 0 = 1 > 0$. Siden f er summen av to kjente kontinuerlige funksjoner, er f selv kontinuerlig. Dermed kan vi ved skjæringssetningen konkludere med at det må være et nullpunkt i definisjonsmengden. Som startpunkt for Newtons metode, bør vi velge et punkt i intervallet, for eksempel $x = -0.5$. Vi regner også ut for hånd at $f'(x) = e^x + 1$.

```
import numpy as np
x = -0.5

while abs(np.e**x+x) > 0.001:
  x = x-(np.e**x+x)/(np.e**x+1)

print('x=', x)
```

Python gir oss svaret $x = -0.5671431650348623$. Om vi sjekker $f(x)$ for denne verdien, får vi

$$f(-0.5671431650348623) = e^{-0.5671431650348623} - 0.5671431650348623 = 1.964804716703128e^{-07},$$

som ikke er så langt i fra 0.

Merk.
Noen funksjoner har nullpunkter uten å skifte fortegn (for eksempel $f(x) = x^2$), så i noen situasjoner kan Newtons metode virke uten at vi kan benytte skjæringssetningen. Men, det er også mulig at Newtons metode (slik den er skrevet her) gir oss et svar, selv om funksjonen ikke har et nullpunkt. Dermed gir skjæringssetningen en ekstra trygghet for at utregningene vi gjør faktisk gir mening.

2.1.2 Litt teori rundt feilestimat i Newtons metode

Feilleddet i den lineære tilnærmingen rundt $x = a$, kan gis som et andregradsuttrykk

$$f(x) = f(a) + f'(a)(x-1) + feil = f(a)f'(a)(x-1) + \frac{1}{2}f''(c)(x-a)^2.$$

Her er c et tall mellom a og x. Hvis vi bruker den lineære tilnermingen rundt $a = x_n$, og antar at x er et nullpunkt (altså at $f(x) = 0$), får vi

$$0 = f(x) = f(x_n) + f'(x_n)(x-x_n) + \frac{1}{2}f''(c)(x-x_n)^2.$$

Justerer vi litt, ender vi opp med

$$x_n - \frac{f(x_n)}{f'(x_n)} - x = \frac{f''(c)}{2f'(x_n)}(x-x_n)^2.$$

vi ser at venstre side er $x_{n+1} - x$. Hvis $f''(c)$ ikke er for stor, $f'(x_n)$ ikke er for liten, og $x - x_n$ er liten, vil $x_{n+1} - x$ bli mindre enn $x_n - x$ fra denne formelen. For et presist utsagn, med bevis, henvises det til en annen lærebok:

Teorem 2.1.1 (Lindstrøm: Kalkulus, Setning 7.3.3)**.**
Anta at $f(a) = 0$, at $f'(a) \neq 0$ og at $f''(x)$ eksisterer og er kontinuerlig i en omegn rundt a. Da finnes det en $\delta > 0$ slik at følgen $\{x_n\}$ i Newtons metode konvergerer mot a når $x_0 \in (a-\delta, a+\delta)$.

Kapittel 3

Numerisk derivasjon og integrasjon

3.1 Numerisk derivasjon

Definisjonen av den deriverte til $f(x)$ i et punkt $x = a$ er

$$f'(a) = \lim_{h \to 0} \frac{f(a+h) - f(a)}{h}.$$

Verdien til $f'(a)$ forteller hva den momentane endringen til $f(a)$ er. Geometrisk svarer det til stigningstallet til tangenten i $f(a)$. Ved definisjonen over, kan vi tolke dette som en grense av sekanter (rette linjer som skjærer grafen til funksjonen i to punkter).

Fra denne definisjonen kan vi få en metode for å finne den deriverte i Python. Da velger vi en passe liten h, og regner ut brøken for denne verdien:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \approx \frac{f(x+h) - f(x)}{h}, \qquad \text{for liten } |h|.$$

La oss først finne verdien av $f'(0.2)$ når $f(x) = \sin(x)$. Vi prøver med verdiene $h = 1$, $h = 0.1$, $h = 0.01$ og så videre, og lar Python regne ut tilnærmingene for en del av verdiene.

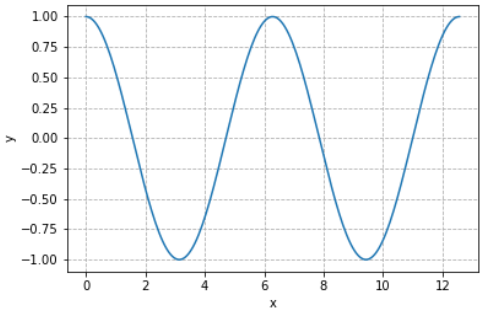
```
import numpy as np

x = 0.2
h = 1
for i in range(20):
  print((np.sin(x+h) - np.sin(x))/h)
  h = h/10
```

Etter ca 4-5 iterasjoner kommer vi til verdien 0.9801, som vi ser stemmer dersom vi kontrollerer med $\cos(0.2)$. Lar vi derimot koden kjøre videre, ser vi at etter 14-15 iterasjoner skjer det noe galt - verdien beveger seg vekk fra den riktige. Grunnen til dette er at Python setter av begrenset plass til å lagre hver variabel. Det innfører avrundingsfeil i utregninger der det inngår tall som er nærme 0. Lærdommen vi tar med oss fra dette er at vi ønsker å bruke en liten h, men ikke *for* liten. I eksempelet over vil $h = 10^{-6} = 0.000001$ være passe stor.

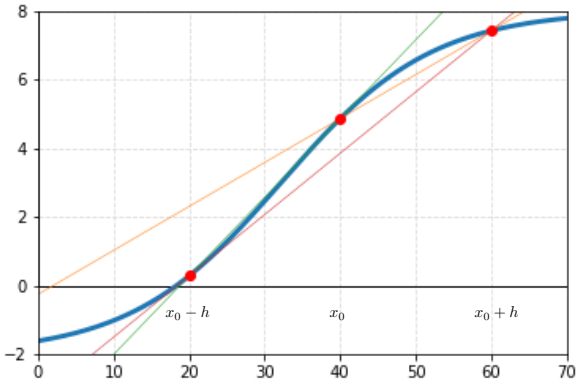
Hvis vi gjør dette for mange x-verdier etter hverandre, og sparer på alle, kan vi plotte den deriverte:

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

N = 200
x = np.linspace(0,4*np.pi, num = N)
y = np.zeros(N)
h = 10**(-6)

for i in range(N):
  y[i] = (np.sin(x[i] + h) - np.sin(x[i]))/h

plt.plot(x,y)
```

Grafen vi får ut ser ut som kurven til $f(x) = \cos(x)$, som er den deriverte av $\sin(x)$.



Tilnærmingen, eller *approksimasjonen*, som vi har brukt over, kalles gjerne for *framoverdifferanse*, fordi vi tar i tar i bruk differansen mellom punktet vi er interessert i (x), og punktet i en avstand h foran dette $(x+h)$. Andre måter å approksimere den deriverte på, er å bruke *bakoverdifferanse* eller *sentraldifferanse*. I figuren 3.1.1 illustrerer den gule linjen en framoverdifferanse, den grønne linjen en bakoverdifferanse og den røde linjen en sentraldifferanse.



Figur 3.1.1: Illustrasjon over framover-, bakover- og sentraldifferanse

Med formler kan disse differansene skrives som

$$f'(x_0) \approx \frac{f(x_0+h)-f(x_0)}{h}, \qquad \text{framoverdifferanse},$$
$$f'(x_0) \approx \frac{f(x_0)-f(x_0-h)}{h}, \qquad \text{bakoverdifferanse},$$
$$f'(x_0) \approx \frac{f(x_0+h)-f(x_0-h)}{2h}, \qquad \text{sentraldifferanse}.$$

Legg merke til at det i sentraldifferansen er delt på $2h$, i stedet for h som i framover- og bakoverdifferansen. Dette er fordi avstanden er h til hvert av punktene vi bruker for å approksimere den deriverte i punktet x_0.

Merk.
Sentraldifferansen er en bedre approksimasjon av den deriverte enn framover- og bakoverdifferansen. Dette er fordi feilleddet i sentraldifferansen er av størrelse: konstant $\cdot\, h^2$, mens det for framover- og bakoverdifferansen er: konstant $\cdot\, h$.

Når vi nå vet hvordan dette virker, kan vi også bruke det på data fra målinger. Dette kan vi gjøre, fordi, som vi ser av de ulike differansene over, trenger vi kun *diskrete* punktverdier for å finne en approksimasjon av den deriverte.

3.2 Numerisk integrasjon

Definisjonen av integrasjon er via en tilnærming med arealer av bokser (høyde ganger bredde). Dersom vi velger mange av disse enkle arealene, slik at bredden til boksene blir liten, kan vi få en god tilnærming til integralet. Vi skal først se på monotone funksjoner, der vi lett kan regne ut tilnærminger som er større eller mindre enn den virkelige verdien. Det gir oss god kontroll over presisjonen i utregningen. Etterpå skal vi se på en metode som virker mer generelt, men der presisjonen er noe vanskeligere å få tak på.

3.2.1 Øvre og nedre Riemann-summer for monotone funksjoner

Vi skal finne en tilnærming til verdien

$$\int_a^b f(x)\,dx.$$

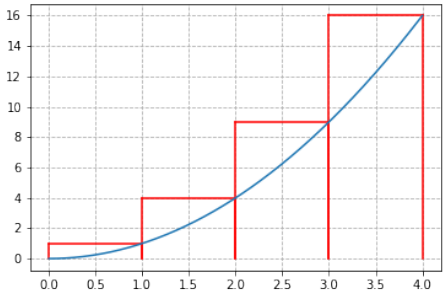
For å gjøre det deler vi inn intervallet $[a,b]$ i like store delintervaller, som har størrelse Δx. Dette kan vi gjøre i Python ved å bruke `linspace`.

Hvis funksjonen er synkende (den deriverte er hele tiden er negativ), vil den største verdien i intervallet $[x, x+\Delta x]$ ligge til venstre (altså i x), og den minste til høyre (altså i $x+\Delta x$). Om funksjonen er voksende (derivert positiv), er det omvendt.

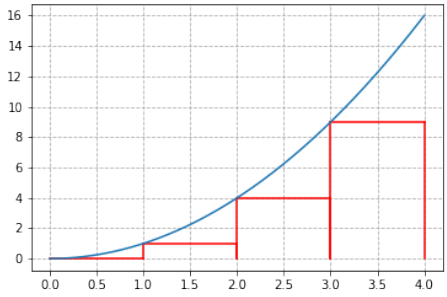
I begge tilfellene vil det største og det minste arealet av en boks med høyde en funksjonsverdi i intervallet, finnes ved å bruke endepunktene i intervallet. Dermed vil den faktiske verdien av integralet ligge mellom summen av arealene til de store og de små boksene. For monotone funksjoner er dette enkelt å regne ut. Se figurene 3.2.1a og 3.2.1b for illustrasjon. Tilnærmingen til integralet ved hjelp av de små boksene, kalles gjerne for den *nedre Riemannsummen*, mens tilnærmingen med de store boksene kalles den *øvre Riemannsummen*.

Eksempel 3.2.1:
I statistikk er verdiene til integralet av funksjonen

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}},$$



3.2.1a: Øvre Riemannsum, arealet av boksene er større enn integralet



3.2.1b: Nedre Riemannsum, arealet av boksene er mindre enn integralet

viktige. Hvis vi begrenser oss til $x > 0$, blir den deriverte negativ, så vi kan bruke tankegangen vår på denne funksjonen. La oss regne ut

$$0.5 + \int_0^2 f(x)\,dx,$$

(0.5 forklares i statistikken). For å finne en tilnærming som er større enn integralet, kan vi bruke de venstre verdiene (øvre Riemannsum), og for å finne en tilnærming som er mindre enn integralet, kan vi bruke de høyre verdiene (nedre Riemannsum):

```
# Ovre Riemannsum
import numpy as np

N = 200
x = np.linspace(0, 2, num = N+1)
delta_x = x[1]-x[0]

Sum = 0

for i in range(0,N):
 Sum = Sum + delta_x*((1/np.sqrt(2*np.pi))*np.exp(-0.5*x[i]**2))

Sum = Sum + 0.5
print(Sum)
```

```
# Nedre Riemannsum
import numpy as np

N = 200
x = np.linspace(0, 2, num = N+1)
delta_x = x[1]-x[0]

Sum = 0

for i in range(1,N+1):
 Sum = Sum + delta_x*((1/np.sqrt(2*np.pi))*np.exp(-0.5*x[i]**2))

Sum = Sum + 0.5
print(Sum)
```

Kodesnuttene over gir oss altså den øvre- og den nedre Riemannsummen, og vi vet dermed at verdien av integralet må ligge mellom de to verdiene vi får, altså i intervallet $[0.9755, 0.9790]$. Ved å bruke en finere oppdeling av x-aksen, kan vi få bedre nøyaktighet på tilnærmingene.

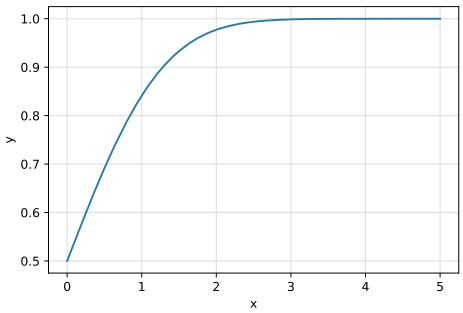
Når vi har funnet en passende oppdeling, kan vi legge på en løkke på alt det vi har gjort, og finne grafen til integralfunksjonen

$$0.5 + \int_0^x f(t)\,dt.$$

Vi tegner den for $x = 5$. For eksempel kan følgende kodesnutt brukes for å plotte integralfunksjonen:

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

delta_t = 0.001
x = np.linspace(0,5, num = 400)
delta_x = x[1]-x[0]

# Lag en tom liste for integralet
integral = []
# Definer en loopindeks for integralet
idx = 0

for x_koord in x:
  # Definer ovre Riemannsum
  ovreSum = 0.5
  for t in np.arange(0, x_koord-delta_t, delta_t):
    ovreSum = ovreSum + delta_t*((1/np.sqrt(2*np.pi))*np.exp(-0.5*(t
        **2)))

  # Definer nedre Riemannsum
  nedreSum = 0.5
  for t in np.arange(delta_t, x_koord, delta_t):
    nedreSum = nedreSum + delta_t*((1/np.sqrt(2*np.pi))*np.exp
        (-0.5*(t**2)))

  # Bruker gjennomsnittet av de to summene som integralet
  integral.append((ovreSum + nedreSum)/2)
  # Oppdater loopindeksen
  idx = idx + 1

plt.plot(x,integral)
plt.grid(color='gainsboro')
```

Plottet ser da slik ut:



Merk.
Legg merke til at vi her har bruk `np.arange` i stedet for `range`-funksjonen som vi har brukt tidligere. Det er fordi vi ønsker å loope over tall som ikke er heltall. (Se: `https://numpy.org/doc/stable/reference/generated/numpy.arange.html` for mer informasjon).

Merk.
Det vi har gjort så langt virker bare for monotone funksjoner, siden vi for andre funksjoner ikke kan vite hvor i et intervall $[x, x + \Delta x]$ en funksjon oppnår maksimum og minimum.

3.2.2 Trapesmetoden

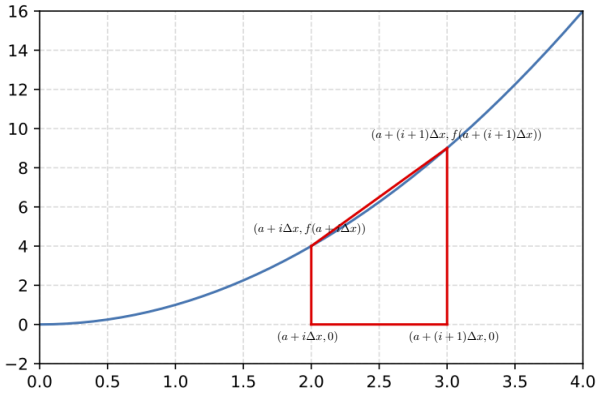
En annen metode for å beregne integraler, som fungerer for mange funksjoner, er *trapesmetoden*. Navnet kommer av at vi approksimerer integralet ved hjelp av *trapeser* (i motsetning til over, hvor vi brukte rektangler). Vi skal nå se på tilnærmingen av integralet

$$\int_a^b f(x)\,dx.$$

Vi starter igjen med å dele inn intervallet $[a, b]$ i en mengde delintervaller (som vi her lar være like store for enkelhets skyld). Hvis vi vil ha n delintervaller, betyr det at vi øker x fra a til b med steg som har størrelse $\Delta x = (b - a)/n$. For hver av disse delintervallene tegner vi trapeset med hjørner

$$\begin{array}{ll} (a + i\Delta x, f(a + i\Delta x)) & (a + (i + 1)\Delta x, f(a + (i + 1)\Delta x)) \\ (a + i\Delta x, 0) & (a + (i + 1)\Delta x, 0) \end{array}$$

se figur 3.2.2.



Figur 3.2.2: Arealet av trapeset er omtrent likt integralet over delintervallet

Arealet til dette trapeset er

$$A_{trapes} = \frac{f(a + i\Delta x) + f(a + (i + 1)\Delta x)}{2} \cdot \Delta x.$$

Vi ser at alle leddene, unntatt for den første x-verdien ($x = a$) og den siste x-verdien ($x = b$), gjentas to ganger. Faktoriserer vi ut Δx får vi dermed:

$$\int_a^b f(x)\,dx \approx \frac{f(a) + 2\sum_{i=1}^{n-1} f(a + i\Delta x) + f(b)}{2} \cdot \Delta x.$$

Hvor god denne tilnærmingen er, varierer med funksjonen. Hvis den dobbeltderiverte ikke er for stor, kan vi være sikre på at vi får en god tilnærming ved å la Δx være liten. Oppsummert består metoden av følgende steg:

Definisjon 3.2.1:
Trapesmetoden for å regne ut tilnærmede verdier for integralet

$$\int_a^b f(x)\,dx,$$

består av følgende steg:

i) Velg antall delintervaller, n.

ii) Sett steglengden til $\Delta x = (b-a)/n$

iii) Legg sammen verdiene $f(a)$, $2f(a+i\Delta x)$ for $i = 1 \ldots n-1$, og $f(b)$

iv) Multipliser med Δx og del på 2

Justér antall delintervallet (n) for å finne et troverdig svar.

Vi kan undersøke kvaliteten på tilnærmingen ved se på en funksjon vi allerede kjenner integralet til, og så anvende teknikken på integraler vi ellers ikke får til senere.

Eksempel 3.2.2:
Vi ser her på et polynom som vi kan beregne integralet av også for hånd:

$$\int_1^4 4x^3 - 2x\,dx.$$

Regner vi ut dette analytisk, får vi:

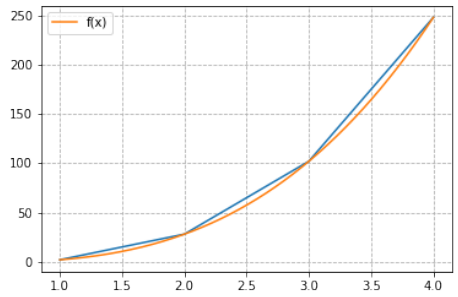
$$\int_1^4 4x^3 - 2x\,dx = \left(x^4 - x^2\right)\Big|_1^4 = (4^4 - 4^2) - (1^4 - 1^2) = 240.$$

Hvis vi vil beregne dette integralet ved hjelp av trapesmetoden kan vi for eksempel bruke denne koden:

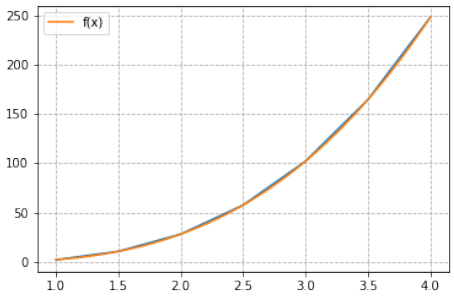
```
import numpy as np
import matplotlib.pyplot as plt

n = 4 # Antall delintervaller

x = np.linspace(1,4,num=n+1)
delta_x = x[1] - x[0] # storrelsen paa delintervallene
f = 4*x**3 - 2*x # funksjonen vi vil integrere

# Definer sum ved start
Sum = 0

# Trapesmetoden
for i in range(n):
 Sum = Sum + ((f[i]+f[i+1])/2)*delta_x

# print resultatet
```

```
print(Sum)

# Definer funksjonen paa ny for glattere graf
y = np.linspace(1,4,10000)
f1 = 4*y**3 - 2*y

# Plot funksjonen og tilnaermingen
plt.plot(x,f)
plt.plot(y,f1, label='f(x)')
plt.grid(ls='--')
plt.legend()
```

Under har vi plottet funksjonen (oransje) og dens tilnærming (blå) med bruk av $n = 3$ og $n = 6$.



3.2.3a: $n = 3$



3.2.3b: $n = 6$

Trapesmetoden gir i disse to tilfellene at de tilnærmede integralene er:

$$n = 3: \quad \int_1^4 4x^3 - 2x\,dx \approx 255.0$$
$$n = 6: \quad \int_1^4 4x^3 - 2x\,dx \approx 243.75$$

Litt om feilen i trapesmetoden

Det er litt arbeid å vise skikkelig feilestimat for trapesmetoden, men det går an å se på resultatet. For et bevis, se: Adams, *Calculus*.

Teorem 3.2.1.
Anta at f er en to ganger deriverbar funksjon, definert på et åpent intervall som inneholder $[a, b]$. Anta også at $|f''(x)| < K$ for alle $x \in [a, b]$. Da er feilen vi gjør ved å bruke trapesmetoden med n delintervaller mindre enn $K(b-a)^3/12n^2$:

$$\left| \int_a^b f(x)\,dx - \frac{f(a) + 2\sum_{i=1}^{n-1} f(a + i\Delta x) + f(b)}{2} \cdot \Delta x \right| < \frac{K(b-a)^3}{12n^2}$$

Eksempel 3.2.3:
I dette eksempelet skal vi se på en formel for lengden av en parametrisert kurve. Denne er gitt ved $x = x(t)$ og $y = y(t)$. Dersom tiden varierer mellom t_0 og t_1, er lengden av denne kurven gitt ved integralet

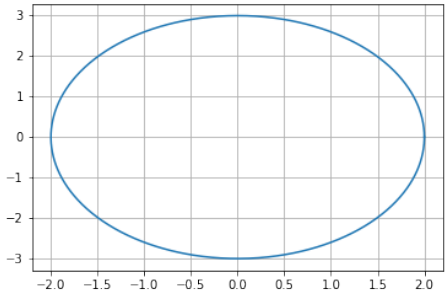
$$\int_{t_0}^{t_1} \sqrt{x'(t)^2 + y'(t)^2}\,dt.$$

Et eksempel på en slik parametrisering, er gitt ved:

$$x(t) = a\cos(t)$$
$$y(t) = b\sin(t).$$

Dette kan vi plotte i Python på følgende måte (velg verdier for a og b):

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

t = np.linspace(0,2*np.pi,num=1000)

a = 2
b = 3

x = a*np.cos(t)
y = b*np.sin(t)

plt.plot(x,y)
```



Figur 3.2.4: Den parametriserte kurven.

Dersom vi ønsker å finne lengden på denne kurven, deriverer vi x og y, og bruker formelen for lengde oppgitt over.

$$\text{lengde} = \int_0^{2\pi} \sqrt{a^2\sin^2(t) + b^2\cos^2(t)}\, dt.$$

Dette integralet har vi ingen fornuftig måte å løse analytisk, så vi kan prøve å bruke trapesmetoden for å finne en tilnærmet verdi.

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

n = 10
t = np.linspace(0,2*np.pi, num = n+1)
delta_t = t[1]-t[0]

a = 2
b = 3

f = np.sqrt((a**2)*((np.sin(t))**2) + (b**2)*((np.cos(t))**2))
lengde = 0
```

```
    for i in range(n):
     lengde = lengde + ((f[i] + f[i+1])/2)*delta_t

    print(lengde)
```

Denne koden gir oss `lengde = 15.865710293064106`.

Kapittel 4

Differensiallikninger

Stoffet i dette avsnittet er bl.a. hentet fra [1].

Vi skal først ta for oss førsteordens differensiallikninger på formen

$$y'(x) = f(x, y). \tag{4.0.1}$$

For slike likninger kan vi forstå hva differensiallikningen betyr i et punkt ved å tenke på tangenter. Hvis vi antar at vi har en funksjon $y(x)$ som er en løsning av differensiallikningen, og som går gjennom et punkt (x_0, y_0), så vet vi at stigningstallet til tangenten til grafen i dette punktet er gitt ved den deriverte, altså $y'(x_0)$. Differensiallikningen gir oss informasjon om at dette stigningstallet er lik $f(x_0, y_0)$. Dermed har vi også likningen for tangenten i dette punktet:

$$y = y_0 + f(x_0, y_0)(x - x_0)$$

Nær et punkt på grafen, vil tangenten i dette punktet være en lineær tilnærming til funksjonen, og dermed vil løsningen av differensiallikningen være omtrent lik dette uttrykket for tangenten nær punktet.

4.1 Retningsfelt - Grafisk forståelse av differensiallikninger

Når vi ser på differensiallikninger på formen

$$y'(x) = f(x, y),$$

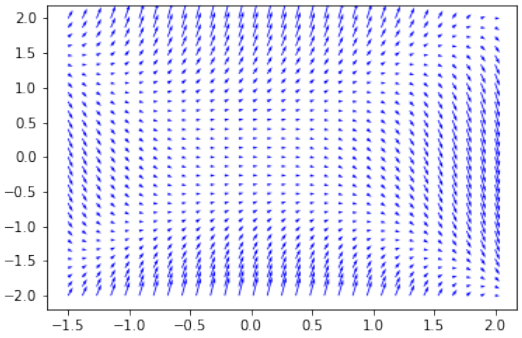
så kan vi utnytte informasjonen vi fant over til å si noe om løsningskurvene til problemet, uten å faktisk løse likningen. Vi utnytter da at i et punk (x, y), så gir differensiallikningen vår oss at stigningstallet til tangenten til løsningen er $y'(x)$ som er gitt av likningen. Vi kan da tegne inn korte linjestykker med dette stigningstallet for mange punkter, og får det som kalles et retningsfelt. Av dette kan vi se hvordan løsningskurvene beveger seg. Vi ser nå på differensiallikningen

$$y' = y^2 - x^2.$$

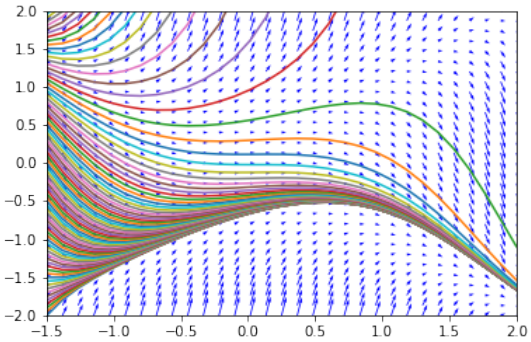
Finn stigningstallet til tangentene i punktene $(-1.5, 0)$, $(-0.5, 0.5)$ og $(0.5, 1)$, og tegn inn korte linjestykker med disse stigningstallene i et x, y-koordinatsystem. Du oppdager kanskje at det ikke er veldig avansert regning, men de tre strekene gir deg lite informasjon om hvordan løsningene oppfører seg. For å få god informasjon ut av retningsfeltet, ønsker vi å tegne inn disse linjestykkene for mange punkter. Å tegne et stort retningsfelt for hånd vil ta ganske lang tid, men vi kan tegne dette i Python ved hjelp av kun få linjer med kode.

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

# Definer antall intervaller langs hver akse
n = 30

# Definer aksene
x = np.linspace(-1.5,2,num = n+1)
y = np.linspace(-2,2,num = n+1)

# Lag et 2D gitter
X,Y = np.meshgrid(x,y)

# Beregn stigningstallet i hvert punkt i gitteret
f = Y**2-X**2

# Definer endringen i x- og y-retning
dx = 1
dy = f

# Plott retningsfeltet
plt.quiver(X,Y,dx,dy, color='blue')
```

Plottet vi da får ut ser slik ut



Vi kan se ut fra hvordan pilene peker, omtrent hvordan løsningskurvene vil se ut. Figuren under viser ulike løsningskurver (beregnet ved hjelp av Eulers metode) for ulike initialbetingelser.



Retningsfeltet er matematisk presist og lett å få frem, mens løsningskurvene vi har plottet er ikke

nøyaktige fordi vi har tilnærmingene i Eulers metode (se neste seksjon). I tillegg krever de en del utregninger. Dermed er retningsfelt en rask og god måte å visualisere løsningskurvene til en gitt differensiallikning.

4.2 Eulers metode

Eulers metode er en metode for å finne numeriske løsninger av differensiallikninger. En *numerisk* løsning av en differensiallikning er en funksjon som er omtrent lik den faktiske løsningen av differensiallikningen.

Vi starter med den generelle likningen $y' = f(x, y)$, sammen med initialbetingelsen (startbetingelsen) $y(x_0) = y_0$. Vi skal finne numeriske verdier for $y(x)$ for ulike verdier av x.

Vi kan for eksempel se for oss at funksjonen $y(x)$ beskriver temperaturen i en stålbjelke i en avstand x fra den ene enden. Dersom vi måler temperaturen $y(0)$ i denne enden, kan vi se hva differensiallikningen forteller oss at temperaturen vil være i en avstand $x = 1\text{mm}$, $x = 2\text{mm}$, $x = 3\text{mm}$, og så videre, fra enden. Vi velger oss da en *steglengde*, Δx, for eksempel $\Delta x = 1\text{mm}$, og lar punktene vi vil evaluere temperaturen i da være gitt ved $x_n = x_0 + n\Delta x$, der n er et positivt heltall. Målet er å finne tilnærmingsverdier til $y(x_1)$, $y(x_2)$, $y(x_3)$, og så videre.

Vi har antatt at vi kjenner startverdien, $y_0 = y(x_0)$. Forutsatt at Δx er valgt tilstrekkelig liten, kan vi nå bruke en *lineær* tilnærming til å finne en *approksimasjon* av $y(x_1)$:

$$y(x_1) \approx y(x_0) + y'(x_0)(x_1 - x_0) = y_0 + y'(x_0)\Delta x.$$

Fra differensiallikningen vet vi at $y'(x_0) = f(x_0, y_0)$, og tilnærmingen over kan dermed skrives som

$$y(x_1) \approx y_1 = y_0 + f(x_0, y_0)\Delta x.$$

(Vi har nå satt at tilnærmingen av $y(x_1)$ heter y_1). Gjentar vi prosedyren, får vi at tilnærmingen av $y(x_2)$ er

$$y_2 = y_1 + f(x_1, y_1)\Delta x.$$

Hvis vi fortsetter slik, kan vi finne at dersom vi har regnet ut tilnærmingen $y_n = y(x_n)$, så vil tilnærmingen til $y(x_{n+1})$ være gitt ved

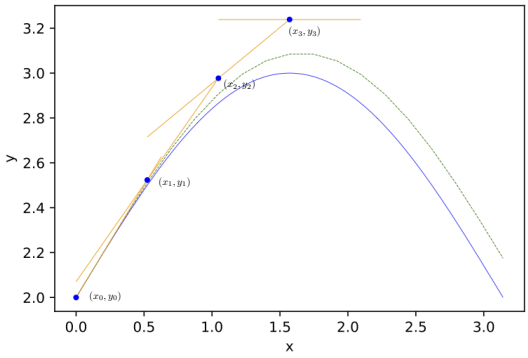
$$y(x_{n+1}) \approx y_{n+1} = y_n + f(x_n, y_n)\Delta x.$$

For at ikke unøyaktighetene i tilnærmingene skal bli for store, må vi velge Δx tilstrekkelig liten, men likevel ikke så liten at avrundingsfeil som skjer i datamaskinen blir betydelige.

Fremgangsmåten som er beskrevet over er den som utgjør *Eulers metode*, og den kan i prinsippet brukes til å finne tilnærmingsløsninger til enhver differensiallikning på formen $y' = f(x, y)$.

I figur 4.2.1 har vi illustrert framgangsmåten. Vi vet verdien for y i x_0, og har dermed det første punktet på grafen vår. Vi bruker differensiallikningen til å finne tangenten i dette punktet, og følger tangenten et stykke, Δx, før vi finner y_1 ved hjelp av denne. Når vi har funnet y_1, finner vi tangenten i dette punktet, igjen ved hjelp av differensiallikningen, følger tangenten et stykke, før vi finner y_2. Slik fortsetter vi over hele domenet vårt.

Legg merke til at når vi finner tangenten i x_1, så er det flere elementer vi bruker, som er tilnærminger. For eksempel er det ikke sikkert at $y_1 = y(x_1)$, så punktet vi bruker for å finne tangenten er ikke nødvendigvis helt riktig. I tillegg vil $f(x_1, y_1)$ ikke nødvendigvis være lik $y'(x_1)$, så stigningstallet er også bare tilnærmet riktig. Til slutt er også tangenten en (lineær) tilnærming til en kurve. Dermed vil det neste punktet vårt, (x_2, y_2), være det beste vi får til med alle disse tilnærmingene. Vi kan minske betydningen av disse problemene dersom vi velger en kortere steglengde, Δx (altså at vi beveger oss kortere langs hver tangent).



Figur 4.2.1: Illustrasjon av Eulers metode

I figur 4.2.1 er den blå grafen den eksakte løsningen (som er ukjent), den grønne stiplete grafen er en numerisk løsning med finere inndeling enn det vi har tegnet inn for å få fram de oransje tangentene.

Eulers metode - oppsummert

Gitt en førsteordens differensiallikning

$$y' = f(x, y),$$

med initialbetingelse

$$y(x_0) = y_0,$$

og steglengde Δx, kan vi finne tilnærmelsesverdier y_n til $y(x_n)$ ved å bruke

$$y_{n+1} = f(x_n, y_n)\Delta x + y_n,$$

hvor vi har satt $x_n = x_0 + n\Delta x$.

Eksempel 4.2.1:
La oss teste metoden på differensiallikningen

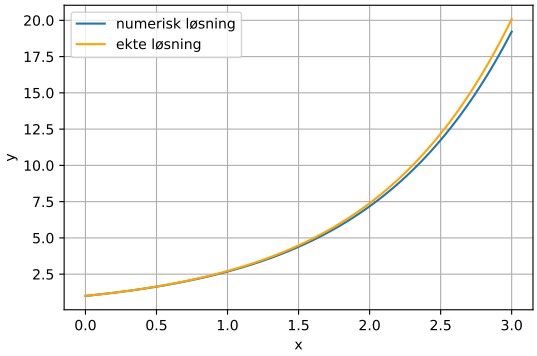
$$y'(x) = y, \tag{4.2.1}$$

med initialbetingelse $y(0) = 1$. Denne differensiallikningen kan vi løse for hånd, og løsningen er $y(x) = e^x$. Det gjør at vi kan undersøke hvordan feilen blir dersom vi bruker Eulers metode for å løse den numerisk. Vi kan bruke følgende kode for å løse problemet (vi lar $x \in [0, 3]$).

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

# Definer antall delintervaller
n = 100

# Del inn x-aksen og definer steglengden
x = np.linspace(0,3, num = n+1)
delta_x = x[1] - x[0]

# Definer den ekte losningen
```

```
y_ekte = np.exp(x)

# Initialiser losningsvektor, og sett initialbetingelsen
y = np.zeros(n+1)
y[0] = 1

# Loop over alle punktene paa x-aksen
for i in range(n):
  y[i+1] = y[i]*delta_x + y[i]

# Plott numerisk og ekte losning for sammenlikning
plt.plot(x,y)
plt.plot(x,y_ekte,color='orange')
```

Med denne koden, får vi at den numeriske løsningen i $x = 3$ er $y = 19.218631980856237$. Den ekte løsningen i dette punktet er $y = 20.085536923187668$, altså ser vi at vi er et stykke i fra den korrekte løsningen. Her kan det være lurt å minske steglengden (øke antall delintervaller, n) for å få bedre nøyaktighet. Plottet som blir generert av koden over vises i figur 4.2.2



Figur 4.2.2

Eksempel 4.2.2 (Torricellis lov, (Eksempel 12.2.1 og 12.2.5 i [1]))**:**
Toricellis lov sier at når væske strømmer ut av et kar gjennom en liten åpning, så er hastigheten, v, i utløpet gitt ved $v = \sqrt{2gh}$, der h er høydeforskjellen mellom væskeoverflaten og åpningen, og g er tyngdeakselerasjonen. I dette eksempelet skal vi se på hvordan vi kan bruke dette til å bestemme høyden $h(t)$ som en funksjon av tiden.

Vi antar at ved starten ($t = 0$), så er væskeoverflaten en høyde h_0 over åpningen i karet. Vi trekker så ut en propp, slik at væsken strømmer ut av åpningen. I starten vil væsken strømme raskt ut, men hastigheten vil avta etter hvert som høyden blir mindre. Ved hjelp av Torricellis lov kan vi sette opp en differensiallikning som beskriver problemet.

Anta at karet vi ser på er en sylinder med grunnflate A, og at åpningen i bunnen av karet har areal B. Volumet av væsken, når hastigheten er $v(t)$, som slipper ut av karet i løpet av et lite tidsrom, Δt, er omtrent $Bv(t)\Delta t$. Samtidig som væske strømmer ut av karet, vil volumet som er igjen i karet avta fra $Ah(t)$ til $Ah(t + \Delta t)$. Forskjellen her må altså være

$$Ah(t) - Ah(t + \Delta t) \approx Bv(t)\Delta t.$$

Hvis vi nå deler på Δt og lar $\Delta t \to 0$, får vi

$$\lim_{t \to 0} \frac{Ah(t) - A(h - \Delta t)}{\Delta t} = Bv(t).$$

Venstresiden av likningen over, gjenkjenner vi som den deriverte av $-Ah(t)$, og vi har dermed at

$$-Ah'(t) = Bv(t).$$

Løser vi så for $h'(t)$, får vi til slutt at Torricellis lov gir oss

$$h'(t) = -\frac{B}{A}\sqrt{2gh(t)}.$$

Dette er en førsteordens differensiallikning, og vi ser at den er på formen som i likning 4.0.1, men med $h'(t)$ i stedet for $y'(x)$ og $-\frac{B}{A}\sqrt{2gh}$ i stedet for $f(x, y)$.

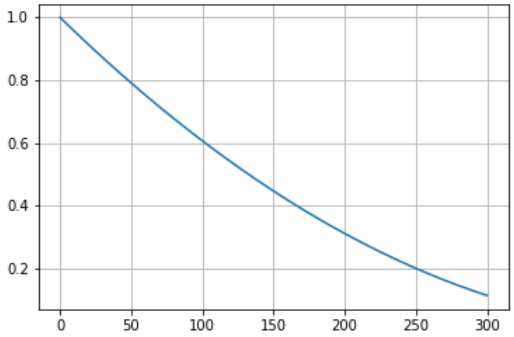
Når vi nå skal løse dette problemet, må vi spesifisere størrelsene som inngår. Vi kan for eksempel la grunnflaten til sylinder ha areal $A = 1$ m^2, og åpningen ha areal $B = 10$ cm$^2 = 0.001$ m$^2 = 10^{-3}$ m$^2 =$. Videre kan vi la starthøyden på vannflaten være $h_0 = 1$ m. Da ser likningen vår ut slik

$$h'(t) = -0.001\sqrt{2 \cdot 9.81 \cdot h(t)}.$$

For å løse problemet numerisk med Eulers metode, må vi også velge en skrittlengde, Δt. Her blir valget litt tilfeldig, så vi velger $\Delta t = 0.1$, og ser an situasjonen underveis. Hvis vi må regne ut veldig mange trinn før vannstanden synker noe særlig, øker vi skrittlengden, og dersom karet tømmes bare i løpet av noen få steg, så minker vi skrittlengden. I koden under har vi brukt $\Delta t = 0.1$:

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

# Definer oppdelingen av t aksen
n = 3000
t = np.linspace(0, 300, num = n+1)
delta_t = t[1] - t[0]

# Definer initialverdien
h = np.zeros(n+1)
h[0] = 1

for i in range(n):
 h[i+1] = ((-10**(-3))*np.sqrt(2*9.81*h[i]))*delta_t + h[i]

plt.plot(t,h)
```

Figuren denne koden genererer, ser slik ut:



Test deg selv:
Kan du skrive en kode som finner ut når karet er halvtomt?

4.2.1 Feilestimat for Eulers metode

Dersom vi bruker Eulers metode for å løse en differensiallikning med initialbetingelse, og vi også vet at løsningen er to ganger deriverbar, så kan vi sammenlikne de numeriske beregningene med løsningsfunksjonens tangent, og bruke feilestimatet fra teorien for Taylor-polynomer. Dette er bare ment som et uformelt feilestimat, men det går an å presisere denne utregningen.

Vi ser altså på differensiallikningen

$$y' = f(x, y),$$

med initialbetingelse $y(x_0) = y_0$. Anta at y'' eksisterer og er kontinuerlig og begrenset nær x_0. Da har vi at

$$y(x) = y_0 + y'(x_0)(x - x_0) + \frac{y''(c)}{2}(x - x_0)^2 = y_0 + f(x_0, y_0)(x - x_0) + \frac{y''(c)}{2}(x - x_0)^2,$$

der c er et tall mellom x og x_0. Dersom vi lar $x = x_0 + \Delta x$, der Δx er skrittlengden i Eulers metode, har vi

$$y(x_0 + \Delta x) = y_0 + y'(x_0) \cdot \Delta x + \frac{y''(c)}{2} \cdot (\Delta x)^2 = y_0 + f(x_0, y_0) \cdot \Delta x + \frac{y''(c)}{2} \cdot (\Delta x)^2.$$

Fra før vet vi at tilnærmingen fra Eulers metode er

$$y(x_0 + \Delta x) \approx y_0 + f(x_0, y_0) \cdot \Delta x,$$

og dermed får vi at feilen vi gjør er den samme som feilen fra Taylorpolynomet:

$$y(x_0 + \Delta x) - y_1 = \frac{y''(c)}{2}(\Delta x)^2 = \text{en konstant} \cdot (\Delta x)^2.$$

Altså får vi for hvert skritt vi tar med metoden, en feil på: konstant $\cdot \Delta x^2$. Dersom vi skal løse likningen på intervallet $x \in [a, b]$ må vi ta omtrent $(b - a)/\Delta x$ steg så kan vi tenke oss at feilene "adderes opp til"

$$\underbrace{\text{konstant} \cdot (\Delta x)^2 + \text{konstant} \cdot (\Delta x)^2 + \ldots + \text{konstant} \cdot (\Delta x)^2}_{(b-a)/\Delta x \text{ ganger}} = \text{konstant} \cdot \Delta x$$

Dette er ikke en presis forklaring, (se for eksempel [3] for en detaljert utledning). Feilen metoden gjør *globalt* er av størrelsesorden Δx, som betyr at dersom vi ønsker én desimals bedre nøyaktighet på den numeriske løsningen, må steglengden være 10 ganger kortere. Dette krever ganske mye utregning, for ganske lite forbedring. I tillegg er metoden slik at det vil innføres systematiske feil dersom den dobbeltderiverte har konstant fortegn. Metoden har derimot den fordelen at den er mer intuitiv, og enklere å kode enn andre løsningsmetoder. Senere i dette kapittelet skal vi se på en annen metode for å løse differensiallikninger som er mer nøyaktig.

4.3 Eulers metode for systemer

I denne seksjonen skal vi se på hvordan vi kan løse systemer av differensiallikninger ved hjelp av Eulers metode. Det meste av stoffet her er basert på boken [1].

Et system av differensiallikninger kan for eksempel se slik ut:

$$y' = f(t, y, z),$$
$$z' = g(t, y, z),$$

med initialbetingelsene $y(t_0) = y_0$ og $z(t_0) = z_0$. For å løse dette systemet ved hjelp av Eulers metode, starter vi, på samme måte som i seksjon 4.2, med å definere en steglengde, Δt, og setter $t_n = t_0 + n\Delta t$ (legg merke til at vi her kaller den uavhengige variabelen for t i stedet for x. Dette har ingen betydning, og vi kunne like gjerne brukt x). Vi ønsker å finne tilnærmede verdier for $y(t_n)$ og $z(t_n)$, og kaller disse y_n og z_n. Vi regner ut neste tilnærmingsverdiene ved å bruke en lineær approksimasjon slik som tidligere. Altså, de neste tilnærmingsverdiene kan beregnes ved hjelp av

$$y_{n+1} = y_n + f(t_n, y_n, z_n)\Delta t,$$
$$z_{n+1} = z_n + g(t_n, y_n, z_n)\Delta t.$$

Eksempel 4.3.1 ((Eksempel 12.7.14 i [1])):
La oss se på systemet

$$y' = 2ty + z,$$
$$z' = y^{-2} + e^{-t},$$

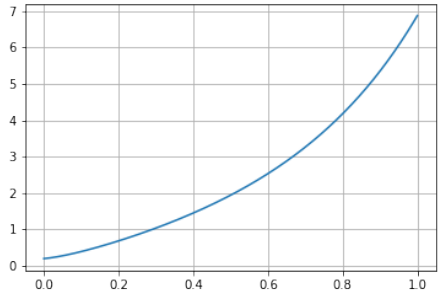
med initialbetingelsene $y(0) = 0.2$ og $z(0) = 1$. Vi skal her se på løsningen i intervallet $t \in [0, 1]$.

Følgende kodesnutt

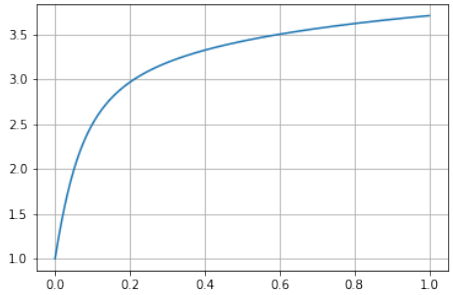
```
import numpy as np
import matplotlib.pyplot as plt

n = 1000 # Antall steg vi skal ta

# Del inn t-aksen og bestem delta_t
t = np.linspace(0,1,num = n+1)
delta_t = t[1]-t[0]

# Definer losningsvektorene
y = np.zeros(n+1)
z = np.zeros(n+1)

# Sett initialbetingelsene
y[0] = 0.2
z[0] = 1

for i in range(n):
 f = 2*t[i]*y[i] + z[i]
 g = y[i]**(-2) + np.exp(-t[i])

 y[i+1] = f*delta_t + y[i]
 z[i+1] = g*delta_t + z[i]


# Plott losningene i to forskjellige figurer
plt.figure(1)
plt.grid()
plt.plot(t,y)
```

```
plt.figure(2)
plt.grid()
plt.plot(t, z)
```

gir oss da de følgende plottene



4.3.1a: Løsningen for $y' = 2ty + z$.



4.3.1b: Løsning for $z' = y^{-2} + e^{-t}$.

4.3.1 Høyereordens differensiallikninger som systemer av førsteordens likninger

I dette kompendiet har vi kun sett på differensiallikninger av første orden. Nå skal vi se på hvordan vi kan bruke teorien så langt på differensiallikninger av høyere orden, ved å gjøre disse om til systemer av førsteordens likninger.

Vi illustrerer framgangsmåten med et eksempel:

Eksempel 4.3.2 ((Eksempel 12.7.15 i [1]))**:**
Gjør følgende andreordens differensiallikning om til et system av førsteordens differensiallikninger:

$$y'' + ty' + 2ty = \sin t. \tag{4.3.1}$$

Det første vi gjør er å introdusere en ny variabel, for eksempel z, og setter denne lik den deriverte av den opprinnelige funksjonen y; $z = y'$. Da har vi at $z' = y''$. Setter vi dette inn i likning (4.3.1), får vi

$$z' + tz + 2ty = \sin t.$$

Dermed har vi nå

$$\begin{aligned} y' &= z, \\ z' &= -tz - 2ty + \sin t, \end{aligned}$$

som er et system av førsteordens differensiallikninger. Nå kan vi bruke teorien vi har lært om Eulers metode for system av differensiallikninger for å finne løsningen, y. Når vi koder dette, vil det være nok å plotte løsningen for bare y, siden det er denne som er den opprinnelige funksjonen vi ønsker å finne. Koden under løser dette problemet med initialbetingelsene $y(0) = 2$ og $y'(0) = 3$ ved hjelp av Eulers metode.

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

```
n = 1000 # Antall steg vi skal ta

# Del inn t-aksen og bestem delta_t
t = np.linspace(0,10,num = n+1)
delta_t = t[1]-t[0]

# Definer losningsvektorene
y = np.zeros(n+1)
z = np.zeros(n+1)

# Sett initialbetingelsene
y[0] = 2
z[0] = 3

for i in range(n):
y[i+1] = z[i]*delta_t + y[i]
z[i+1] = (-t[i]*z[i]-2*t[i]*y[i] + np.sin(t[i]))*delta_t + z[i]


# Plott losningene i to forskjellige figurer
plt.figure(1)
plt.grid()
plt.plot(t,y)
```

Eksempel 4.3.3 (Svigninger (Eksempel 9.9.3 i [2]))**:**
La oss ta for oss eksempel 9.9.3 i læreboken. Vi ser for oss et lodd med masse M, som henger i en fjær med stivhet k. Vi antar at loddet blir trukket nedover og så blir sluppet. Formelen som beskriver hvordan loddets posisjon, x, varierer med tiden, t, kan beskrives av funksjonen

$$Mx'' + \alpha x' + kx = 0,$$

hvor α representerer luftmotstanden. Ved teorem 9.9.1 i læreboken kan vi finne en analytisk løsning til denne differensiallikningen. Skal vi derimot løse likningen numerisk ved hjelp av Eulers metode, kan vi innføre $y = x'$, og skrive differensiallikningen som et system:

$$\begin{aligned} x' &= y \\ y' &= -\frac{\alpha}{M}y - \frac{k}{M}x \end{aligned}$$

Vi tar nå utgangspunkt i at $M = 1$, $\alpha = 2$ og $k = 1$. I tillegg har vi initialbetingelsene $x(0) = 1$ og $x'(0) = 0$. Koden under løser dette problemet.

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

# Definer antall delintervaller og del opp t-aksen
n = 2000
t = np.linspace(0,20, num = n+1)
# Finn steglengden
delta_t = t[1]-t[0]

# Initialiser losningsvektorene
x = np.zeros(n+1)
y = np.zeros(n+1)
```

```
# Sett initialbetingelsene
x[0] = 1
y[0] = 0

# Definer konstantene
M = 1
alpha = 2
k = 1

# Los vha Eulers metode
for i in range(n):
  x[i+1] = x[i] + y[i]*delta_t
  y[i+1] = y[i] - ((alpha/M)*y[i] + (k/M)*x[i])*delta_t

plt.plot(t,x)
```

Test deg selv:
Kjør koden med verdier $\alpha \in [-3, 3]$. Hva skjer?

Vi kan si følgende om ulike verdier for α. Etter testene du nettopp har kjørt la du kanskje merke til:

- $\alpha = 0$: i dette tilfellet vil den analytiske løsningen være en svigning med konstant utslag. Loddet vil svinge med samme utslag for all framtid. Koden over viser imidlertid kanskje ikke det samme, og dette skyldes feilen i Eulers metode. Prøv å redusere steglengden og øke antall punkter for å få dette tydeligere fram.
- $0 < \alpha < 2$: gir en dempet svigning, der utslaget blir mindre og mindre.
- $\alpha > 2$: representerer en fjær med så mye friksjon at det egentlig ikke gir noen svigning (tenk på en rusten fjær).
- $-2 < \alpha < 0$: gir en funksjon som svinger med større og større utslag.
- $\alpha < -2$: gir en funksjon som går mot minus uendelig. Både dette tilfellet og det forrige representerer ikke egentlige svigninger, fordi det må *tilføres* ekstra energi for å få denne oppførselen.

4.4 Runge-Kutta metoder

Det finnes veldig mange numeriske metoder som kalles for Runge-Kutta metoder. Den vi skal presentere her kalles en *fjerdeordens metode* fordi feilen til metoden er i størrelsesorden h^4, der h er steglengden.

Fjerdeordens Runge-Kutta:

Løsningen av initialverdiproblemet

$$y' = f(x, y), \qquad y(x_0) = y_0,$$

kan løses numerisk ved

$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4),$$

hvor h er steglengden, og

$$\begin{aligned} k_1 &= f(x_n, y_n), \\ k_2 &= f(x_n + \tfrac{h}{2}, y_n + \tfrac{h}{2}k_1), \\ k_3 &= f(x_n + \tfrac{h}{2}, y_n + \tfrac{h}{2}k_2), \\ k_4 &= f(x_n + h, y_n + hk_3), \end{aligned}$$

Da er $y_n \approx y(x_0 + nh)$, med en feil mindre enn: konstant $\cdot\, h^4$.

Vi viser hvordan dette kan kodes ved å se tilbake på eksempelet 4.2.1 som vi presenterte i seksjon 4.2.

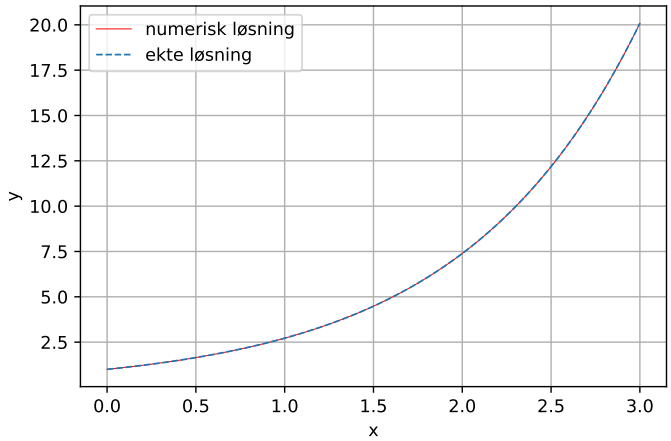
Eksempel 4.4.1:
La

$$y' = y,$$

og $y(0) = 1$. Som sist er den eksakte løsningen til dette problemet $y = e^x$. En fjerdeordens Runge-Kutta metode for å løse dette initialverdiproblemet kan kodes slik:

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

n = 100

x = np.linspace(0,3, num=n+1)
h = x[1]-x[0]
y_ekte = np.exp(x)

y = np.zeros(len(x))
y[0] = 1

for i in range(n):
  k1 = y[i]
  k2 = y[i] + (h/2)* k1
  k3 = y[i] + (h/2)* k2
  k4 = y[i] + h*k3

  y[i+1] = y[i] + (h/6)*(k1 + 2*k2 + 2*k3 + k4)

plt.plot(x,y, color='red', linewidth=0.5)
plt.plot(x,y_true, ls='--', linewidth=1)
plt.grid()
```

Plottet som koden genererer ser slik ut:



Figur 4.4.1

Dersom vi printer ut tilnærmingen for $x = 3$, får vi at metoden gir $y = 20.085536526494263$, mens den eksakte løsningen fremdeles er $y = 20.085536923187668$. Sammenligner vi de numeriske løsningene fra dette eksempelet og eksempel 4.2.1, ser vi at feilen er mindre for Runge-Kutta enn for Euler. Dette ser vi også tydelig fra plottet over. Der det var lett å skille den eksakte og den numeriske løsningen når vi brukte Eulers metode, er det vanskelig å skille de to i plottet over. Dette er også som forventet siden denne Runge-Kuttametoden er av orden h^4, mens Eulers metode er av orden h.

Sammenlignet med Eulers metode, er Runge-Kutta metoder kanskje vanskeligere å huske. En fordel med metoden er at den gir grunnlag for kode som det ikke er så komplisert å endre på for å bruke på andre funksjoner.

Kapittel 5

Lineær algebra

I dette kapittelet skal vi kort gå gjennom noen funksjoner vi kan bruke i Python når vi jobber med vektorer og matriser. Mer utfyllende informasjon kan dere finne her: `https://numpy.org/doc/stable/user/quickstart.html`.

5.1 Vektorer

Å definere vektorer kan vi gjøre ved hjelp av pakken *numpy*. I kodesnutten under definerer vi en vektor $x = [1, 2, 3, 4]$:

```
import numpy as np

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

En viktig ting å notere seg her, er at Python starter å telle på 0. Det vil si at det første elementet i en vektor får vi ved å be Python gi oss element nr. 0. Bygger vi videre på kodesnutten over, kan vi printe det første elementet i vektoren, og det vil se slik ut:

```
import numpy as np

x = np.array([1,2,3,4])
print(x[0])
```

som sagt, vil Python her gi oss det første elementet i x, altså 1.

5.2 Matriser

Matriser kan defineres på en lignende måte som vektorene:

```
import numpy as np

M = np.array([[1,2],[3,4]])
print(M)
```

Resultatet av kodesnutten over er:

```
[[1,2]
```

`[3,4]]`.

Ønsker vi å hente ut et element av en matrise, må vi presisere plasseringen til elementet ved hjelp av radnummeret og kolonnenummeret det sitter på. Igjen er det viktig å huske på at første rad og kolonne vil være nr. 0. Det vil si at dersom vi ønsker å hente ut elementet 3 av matrisen M over, gjør vi dette slik:

```
import numpy as np

M = np.array([[1,2],[3,4]])
print(M[1,0])
```

Dersom vi ønsker å hente ut hele rader eller kolonner fra matrisen, gjøres dette ved å angi enten rad- eller kolonnenummeret og kolon i den andre indeksen. For eksempel vil kodesnutten under gi oss hele rad nr. 2 i matrisen M.

```
print(M[1,:])
```

Ønsker vi å hente ut for eksempel kolonne nr. 1 kan vi bruke

```
print(M[:,0])
```

Noen ofte brukte matriser

Noen matriser som vi bruker mye, er det egne funksjoner for å lage i numpy. Dette gjelder matriser der alle elementene er null, matriser der alle elementene er 1 og identitetsmatrisen. Ønsker vi å lage en 5×5-matrise der alle elementene er 0, kan dette gjøres på følgende måte:

```
np.zeros([5,5])
```

Tilsvarende kan en 5×5-matrise der alle elementene er 1 defineres slik:

```
np.ones([5,5])
```

og til slutt kan vi definere en 5×5-identitetsmatrise med følgende kommando

```
np.eye(5)
```

Hvis vi ønsker å endre en verdi på et element i en vektor eller matrise, spesifiserer vi det elementet vi vil endre, og angir den nye verdien med likhetstegn. Hvis vi for eksempel vil endre elementet i rad 2, kolonne 2 i matrisen vår, M, over, kan vi skrive:

```
M[1,1] = 7
```

hvis vi nå printer M, ser vi at elementet har fått verdien 7.

5.3 Vektor-matriseoperasjoner

Numpy-pakken inneholder også innebygde funksjoner som kan være nyttige for oss når vi jobber med vektorer og matriser. Mer om disse funksjonene kan dere lese om her: `https://numpy.org/doc/stable/reference/routines.linalg.html`. Noen funksjoner som vi kan få bruk for i dette kurset er:

i) `np.transpose(M)`. Denne funksjonen gir oss den transponerte matrisen av M.

ii) `np.dot(a,b)`: Denne funksjonen regner ut prikkproduktet (skalarproduktet) av vektorene a og b.

iii) `np.matmul(M,N)`: Denne funksjonen regner ut matriseproduktet av to matriser. Den kan også regne ut matrise-vektorprodukter. Vi vet fra teorien at $a^T M$ ikke nødvendigvis gir en vektor med de samme elementene som Ma, og avhengig av hvilket produkt vi ønsker, må vi spesifisere argumentene i funksjonen riktig. `np.matmul(a,M)` vil gi oss produktet $a^T M$, mens `np.matmul(M,a)` vil gi oss produktet Ma.

iv) `np.linalg.det(M)`: Denne funksjonen finner determinanten til M.

v) `np.linalg.inv(M)`: Denne funksjonen finner den inverse matrisen til M.

vi) `np.linalg.solve(M,a)` løser likningssystemet $Mx = a$ for x.

Eksempel 5.3.1 (En fordel med koding)**:**
Nå skal vi se på et eksempel, som, selv om matrisen og vektorene fremdeles er små, viser en fordel med å bruke koding.

Se gjennom eksempelet på side 335 i læreboken. Dette er et eksempel på studium av populasjonsvekst. Anta nå at vi har to aldersgrupper:

1. De under 25 år
2. De fra 25 år opptil 50 år

Hvis vi nå lar $F_1 = 1.8$, $F_2 = 0.1$ og $p_1 = 0.9$, får vi likningssystemet:

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 1.8 & 0.1 \\ 0.9 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

Vi antar nå at etter en tidsperiode på $T = 25$ år, så er tilstanden $y_1 = 216$ og $y_2 = 99$. Ved hjelp av likningssystemet kan vi finne ut hva tilstanden var ved start. Til dette kan vi bruke innsettingsmetoden eller Gausseliminasjon, og løsningene kan vi finne relativt enkelt, siden likningssystemet vårt er så lite.

Men hva om vi ønsker å undersøke en finere inndeling av aldersgruppene? I stedet for å la det være 25 års forskjell, lar vi det nå være 10 år. Da får vi denne inndelingen:

1. De under 10 år
2. De fra 10 år opptil 20 år
3. De fra 20 år opptil 30 år
4. De fra 30 år opptil 40 år
5. De fra 40 år opptil 50 år.

Vi tenker oss at $F_1 = 0$, $F_2 = 0.2$, $F_3 = 1.2$, $F_4 = 0.7$, $F_5 = 0.1$ og $p_1 = 0.8$, $p_2 = 0.8$, $p_3 = 0.9$, $p_4 = 0.8$. Etter en periode på $T = 10$ år er antallet i hver aldersgruppe gitt ved

$$y = \begin{bmatrix} 295 \\ 112 \\ 144 \\ 84 \\ 108 \end{bmatrix}.$$

Hvis vi nå spør oss om hva antallet i de ulike aldersgruppene var ved start, kan vi raskt se at vi kommer til å bruke en del lengre tid enn i sted dersom vi skal løse dette for hånd. Fremdeles er likningssystemet vårt veldig lite, men ved hjelp av funksjonen `np.linalg.solve(M,y)` i numpy-pakken får vi svaret umiddelbart, og vi har spart oss for mye tid og regning.

Når vi allerede på et så lite eksempel kan se nytten av å bruke koding, tenk deg hvordan situasjonen er dersom vi har *store* systemer vi vil løse. For eksempel kan matrisene være av størrelse én million $\times$ én million, og det blir praktisk talt umulig for oss å løse det for hånd.

```
import numpy as np

F = [0, 0.2, 1.2, 0.7, 0.1]
p = [0.8, 0.8, 0.9, 0.8]

M = np.array([
    [F[0], F[1], F[2], F[3], F[4]],
    [p[0], 0,    0,    0,    0],
    [0,    p[1], 0,    0,    0],
    [0,    0,    p[2], 0,    0],
    [0,    0,    0,    p[3], 0]
    ])

y = np.array([295, 112, 144, 84, 108])

x = np.linalg.solve(M,y)

# Losningen er da:
print(x)
```

Svaret blir da:

```
[140.          180.          93.33333333 135.          525.        ]
```

Kapittel 6

Optimering og parameterestimering

6.1 Optimering - finne minimum

Innen matematikken er optimering å lete etter de optimale parameterene til en funksjon. Med optimalt mener vi da de verdiene som gir oss funksjonen sin minimumsverdi. Vi antar da at funksjonen har et lokalt minimumspunkt. Dette er selvsagt ikke tilfelle for alle funksjoner, men vil ofte være tilfelle i aktuelle anvendelser. F.eks. er det naturlig å søke etter et minimum av en kostnadsfunksjon (innen økonomi), en energifunksjon (i fysikk), eller reaksjons-hatighetsfunskjson (i kjemi).

Fra før vet vi at vi kan finne minimum av en funksjon ved å sette opp en ligning med den deriverte

$$f'(x) = 0.$$

Dett gir oss stasjonære punkt. Det kan altså være et minimum, et maksimum eller et vendepunkt. For å finne ut om vi har et minimumspunkt må vi se på den andrederiverte. Vi skal i dette kapittelet se på en numerisk løsningsmetode som som kan brukes til å finne minimum av funksjoner. Metoden kalles 'gradient descent'[1] og fungerer både for funksjoner av en variabel og funksjoner av flere variable. Vi bruker ofte notasjonen $f(x)$ for funksjoner av en variabel og $F(\mathbf{x})$ for funksjoner av to eller flere variable. Generelt kan problemet vi skal løse formuleres som

$$\min_{\mathbf{x}} F(\mathbf{x}) \tag{6.1.1}$$

Vi sier da gjerne at funksjonen F er en 'kost-funksjon' som skal minimeres (jf. anvendelser innen økonomi). Det matematiske problemet vårt er å finne en verdi av $\mathbf{x}$ som gjør F så liten som mulig.

Hvorfor ser vi bare etter minimumspunkter og ikke maksimumspunkter?

Vi skal lage en løsningslgoritme for å finne minimumspunkter. Hvis vi i stede ser etter maks av $f(x)$, kan vi lage en ny funksjon $g(x) = -f(x)$ og finne minimum av denne. Da trenger vi ikke endre noe på algoritmen vår.

6.1.1 'Gradient descent' metoden

Gradient descent-metdoen har som utgangspunkt at gradienten til en funksjon i et punkt alltid peker i retningen der funksjonen endrer seg mest. Om går et steg i motsatt retning, vil vi derfor alltid gå mot en lavere verdi av kostfunksjonen. Dette kan vi fortsette med helt til vi finner minimumspunktet.

Gitt et vilkårlig valgt startpunkt $\mathbf{x_0}$, er metoden definert som følger:

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \gamma \nabla F(\mathbf{x}_n). \tag{6.1.2}$$

[1] Det er ikke etablert noe god norsk oversettelse

Vi oppdaterer altså $\mathbf{x}$ med den forrige verdien og et steg i retningen til $-\nabla F$. Vi justerer lengden på steget med parameteren γ. Hvor stor γ skal være vil variere fra problem til problem. I praksis må vi prøve oss frem. Om vi velger γ for stor, vil ikke metoden fungere og om vi velger γ veldig liten vil trenge veldig mange steg for å nå frem til et minimum. Det er også mulig å regne ut den 'ideelle' steglengden i hvert steg, men det skal vi ikke komme inn på i dette kompendiet.

6.1.2 Finne minimum i 1 dimensjon ($f(x)$, er en funksjon av en variabel).

Algoritmen fungerer for funksjoner med både en og flere variable. Vi skal nå se på et eksempel i 1 dimensjon, altså med en funksjon $f(x)$ av en enkelt variabel. I denne sammeneng vil gradienten være den deriverte $\nabla f(x) = f'(x)$.

Koden under løser optimeringsproblemet $\min_x f(x)$ der $f(x) = (x-1)^2$ og $f'(x) = 2(x-1)$. Kostfunksjonen $f(x)$ er plottet i figur 6.1.1

```
[language=Python,numbers=none]
#definerer uttrykket til gradienten
def f_derivert(x):
    return 2*(x - 1)

n = 10 #velger antall steg metoden skal ta
x0 = 4 #setter en (vilkårlig) startverdi
gamma = .7# setter en steglengde

#begynner optimeringen
x = x0
for i in range(n):
    x =  x - gamma*f_derivert(x)

print(x) #skriver ut resultatet
```

Koden vil skrive ut `1.0003145728`, altså en løsning som er riktig til og med det 3 desimalet. Om vi hadde brukt flere steg, ville løsningen blitt enda mere nøyaktig.

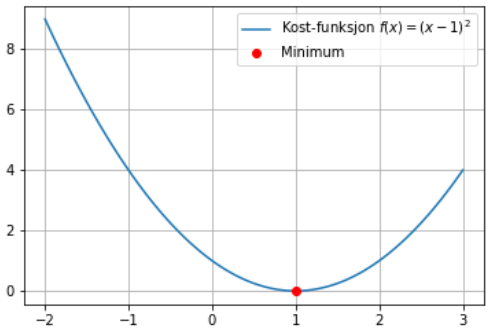
Når vi søker en løsning x av minimeringsproblemet $\min_x f(x)$, så leter vi frem og tilbake langs x-aksen. Vi beveger oss altså bare i en 'dimensjon' og retning i denne sammenheng betyr ganske enkelt høyre eller venstre. Gitt et startpunkt x_0 er altså spørsmålet om vi skal bevege oss mot høyre eller mot venstre og hvor langt vi skal gå.

Hvor mange steg skal vi gå?

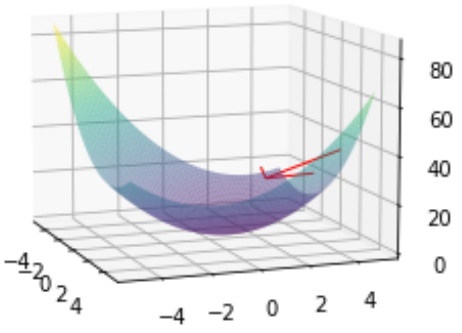
Det er umulig å si på forhånd. Det vil både være avhengig av hvor stor vi velger steglengden, hvor vi starter og, ikke minst, hvordan funksjonen ser ut. Det lureste er derfor å legge inn et stopp-kriterium i koden. Altså en betingelse som sier definerer hva som er en "god nok" løsning. I et optimeringsproblem vil det være flere mulige stoppkriterier:

1. Gradienten (eller nomen til gradienten) er liten, $\|\nabla F\| < tol$, der tol er en liten verdi. Vi vet at i minimumspunktet er gradienten 0. Det kan imidlertid ta veldig lang tid å komme dit (Fordi vi går steg som er av samme størrelesorden som gradienten. Når gradienten begynner å bli liten og kostfunksjonen flater ut går vi derfor veldig små steg).
2. Kostfunksjonen begynner å bli større. Dette er enkelt å teste ved å teste om $F(\mathbf{x_{i+1}}) > F(\mathbf{x_i})$
3. Kostfunksjonen endrer seg veldig lite. Altså om $F(\mathbf{x_i}) - F(\mathbf{x_{i+1}}) < tol$, der tol er et lite tall.

Legg merke til at ingen av kriteriene ovenfor kan garantere at vi faktisk er i et minimum. Vi kan f.eks. tenke oss at vi har en funksjon som flater veldig ut i et område der den *nesten* er konstant, før den synker videre.



6.1.1a: 1-dimensjonal kostfunksjon $f(x) = (x-1)^2$.



6.1.1b: 2-dimensjonal kostfunksjon $F(x,y) = x^2 + (y-1)^2 + xy$.

Figur 6.1.1: I det en-dimensjonale tilfelle (a) vil retningen til gradienten være det samme som fortegnet til den deriverte, altså pluss eller minus. I det to-dimensjonale tilfellet (2) vil gradienten være en vektor med to komponenter. Gradienten i et enkelt punkt er tegnet inn i rødt (men den peker i negativ retning).

6.1.3 Finne minimum i 2 dimensjoner (F, er en funksjon av to eller flere variable).

Det er litt lettere å forstå prinsippet bak gradient descent- metoden med et eksempel med to variable. Da gir gradienten en mer naturlig retning (ikke bare høyre og venstre). I eksempelet under skal vi løse optimeringsproblemet

$$\min_{x,y} \; x^2 + (y-1)^2 + xy \tag{6.1.3}$$

Dette problemet kan vi også løse analytisk og vil da få løsningen

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} -\frac{2}{3} \\ \frac{4}{3} \end{bmatrix}.$$

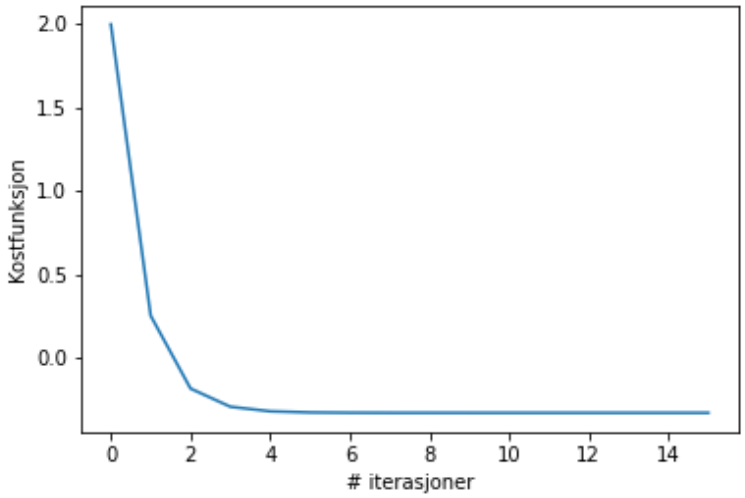
La kostfunksjonen hete $F(x,y)$. I figur 6.1.1 ser vi et plot av funksjonen $F(x,y)$. Fra plottet kan vi også se at den analytiske løsningen av minimumspunktet ser ut til å være riktig.

Vi skal nå løse minimeringsproblemet med gradient descent metoden. Først partiellderiverer vi og finner gradienten

$$\nabla F(x,y) = \begin{bmatrix} 2x + y \\ 2y + x - 2 \end{bmatrix}$$

Så må vi endre litt på koden fra 1-D eksempelet for å få den til å fungere for funksjoner av 2 variable. I tillegg legger vi inn den aktuelle gradienten.

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

#DEFINERER FUNKSJONER OG AKTUELLE VARIABLE

# velger max antall iterasjoner
n = 100

# Definerer kostfunksjonen
def F(x,y):
    val = np.array([x**2 + (y-1)**2 + x*y ])
    return val
```



Figur 6.1.2: Utviklingen av kostfunksjonen i eksempelet med $\min_{x,y}\ x^2 + (y-1)^2 + xy$.

```
# Definerer gradientfunksjonen
def grad_F(x,y):
     grad = np.array([2*x + y,2*y + x-2])
     return grad

#Setter startverdiene (vilkårlig)
x0 = np.array([1])
y0 = np.array([1])

#setter toleranserverdien (for normen av gradienten)
tol = 10**-4

#setter steglengden
gamma = .5;

#BEGYNNER OPTIMERINGEN
x = x0 #initiallisterer x og y
y = y0

cost = F(x,y) #beregner kostfunksjonen

for i in range(n):
   x, y = [x,y] - gamma * grad_F(x,y) #oppdaterer x og y
   cost = np.append(cost, F(x,y))
   if np.linalg.norm(grad_F(x,y)) < tol:
       break

print(x,y) # skriver ut svaret
plt.plot(cost) # plotter kostfunksjonen
```

Selv om vi bare trenger selve løsningen $x = -0.66666$, $y = 1.33328$, så er det praktisk å plotte utviklingen av kostfunksjonen i tillegg. Det kan hjelpe oss til å se om optimeringen har konvergert til en løsning, eller om vi må endre på noen av parameterene våre. Plottet av som returneres av koden over er vist i figur 6.1.2.

Test koden over med $\gamma = 0.6$ og $\gamma = 0.8$. Som nevnt tidligere, kan det være vanskelig å vite hvor stor steglengde vi bør ta. Hvis vi ikke skal utvide algoritmen vår til å finne steglengden automatisk (vha linjesøk), så må vi rett og slett prøve oss frem.

Det er ikke vanskelig å skrive om koden i eksempelet over slik at den fungerer for en `numpy.array` av vilkårlig størrelse. Det vil imidlertid være vanskelig å plotte. For å studere utviklingen av optimeringen er det fortsatt mulig å se på kostfunksjonen eller normen til gradienten. Normen er alltid en skalar, uansett hvor lang vektoren er.

Test deg selv:

Skriv om eksempelet over til en python funksjon som tar inn et gradient-uttrykk av vilkårlig størrelse og returner minimumspunktet. Hvilke verdier kan det være praktisk å ta inn i funksjonen annet enn gradienten?

6.2 Sammenligne data og modell - Regresjon og kurvetilpassing

I svært mange situasjoner har vi observasjoner eller målinger i tillegg til de matematiske modellene eller antakelsene våre. Fra et vitenskapelig standpunkt er dette svært nyttig. Vi kan da sammenligne observasjoner eller målinger med modellen og argumentere for at vi i stor/liten grad har samsvar mellom teori og observasjon. Samtidig er mange modeller ganske generelle. De sier bare noe om hva slags sammenhenger vi forventer å finne i observasjonene, men de inneholder en del ukjente parametre som gir os detaljene i sammenhengen. Vi kan da bruke observasjonene våre til å bestemme disse ukjente parametrene. Dette gjøres ved å sette opp et optimeringsproblem.

La oss si at $y_1, y_2 \cdots y_n$ er verdien av observasjonen vår i tidspunktene $t_1, t_2 \cdots t_n$. Samtidig har vi en modell $u(t)$. Denne modellen er egentlig bare en funksjon. Funksjonen inneholder imidlertid noen parametre $a_1, a_2, \cdots a_k$. Optimeringsproblem består da i å finne parametere a_i slik at kurven til funksjonen $u(t)$ ligner mest mulig på observasjonene i tidspunktene $t_1, t_2 \cdots t_n$. I praksis løser vi

$$\min_{\mathbf{a}} \sum (U_i(\mathbf{a}) - y_i)^2 \tag{6.2.1}$$

Vi har her laget en ny notasjon for modellen. $U_i(\mathbf{a})$ er nå modellen $u(t)$ evaluert i punktet t_i med parametere $\mathbf{a}$. Dette gjør vi fordi vi ønsker å optimere over en funksjon som har parametrene som variable, ikke tiden t eller x. Dette vil bli tydeligere i de konkrete eksemplene under.

6.2.1 Det lineære tilfellet - lineær regresjon

Hvis vi antar at observasjonene våre skal endre seg lineært med tiden, vil modellen vår være gitt av utrykket for en rett linje

$$u(t) = a_1 t + a_0.$$

De to parametrene i denne modellen er stigningstallet a_1 og konstantleddet a_0. Optimeringsproblemet vårt blir altså å finne disse to verdiene for et gitt sett av observasjoner y_i. Minimeringsproblemet (6.2.1) blir da:

$$\min_{a_0, a_1} \sum_i (a_1 t_i + a_0 - y_i)^2 \tag{6.2.2}$$

Dette problemet kalles lineær regresjon har en analytisk løsning. Vi trenger altså ikke løse det med gradient descent-algoritmen fra avsnitt 6.1.1, men kan finne en eksakt formel:

$$a_1 = \frac{\sum_i (t_i y_i) - n\bar{t}\bar{y}}{\sum_i (t_i - \bar{t})^2} \tag{6.2.3}$$

$$a_0 = \bar{y} - a_1 \bar{t}, \tag{6.2.4}$$

der $\bar{t}$ og $\bar{y}$ er gjennomsnittsverdiene til henholdsvis observasjonstidspunktene og observasjonsverdiene. Uttryket for a_1 kan også skrives som $a_1 = \frac{s_{ty}}{s_{tt}}$, der s_{tt} er variansen til t-verdiene og s_{ty} er kovariansen til y- og t-verdiene. Formelen er utledet under, i avsnitt 6.2.3.

6.2.2 Eksempel - temperaturdata

Vi skal nå ta utgangspunkt i temperatur-dataene fra Svalbard lufthavn. Dette er de samme dataene som vi lastet og plottet i avsnitt 1.6. Temperaturen varierer naturlig fra måned til måned og det er også en naturlig variasjon fra år til år. Noen år er ekstra kalde mens noen er varmere. Hvis vi ønsker å se om det er en økende eller synkende trend over lengre tid kan vi tilpasse dataene til en lineær modell v.h.a. regresjon. Vi skal i dette eksempelet lage en regresonsfunksjon vha `numpy` [2] og anvende den på månedstemperaturene fra januar 2000 til desember 2019.

```
#FORTSETTER FRA EKSEMPELET FRA KAP 1
y = tempr[1216:1456]

n = np.size(y)

t = np.linspace(0,1,n)

y_bar = np.mean(y)
t_bar = np.mean(t)

a_1 = (np.sum(t*y)-n*t_bar*y_bar)/np.sum((t-t_bar)**2)
a_0 = y_bar - a_1*t_bar

def reg_line(a_0,a_1,t):
    return a_1*t+a_0

temp_plot = plt.subplot()


plt.plot(tidspunkt[1216:1456],y,'r')
plt.plot(tidspunkt[1216:1456],reg_line(a_0,a_1,t))

#endrer makeringen på x-aksen
temp_plot.xaxis.set_major_locator(ticker.IndexLocator(base=12,offset=0))

#roterer markeringen på x-aksen 70 grader slik at den blir lettere å lese
temp_plot.tick_params(labelrotation=70)
```

Plottet som produseres av koden over er vist i figur 6.2.1. Her bruker vi en lineær modell for temperaturen. Samtidig vet vi at månedstemperaturen oppfører seg svært ikke-lineært og svinger med årstidene. Den lineære kurven vil derfor aldri ligne spesielt mye på de observerte dataene. Vi er imidlertid i stand til å plukke ut den lineære trenden i dataene. Dette kan være svært nyttig.

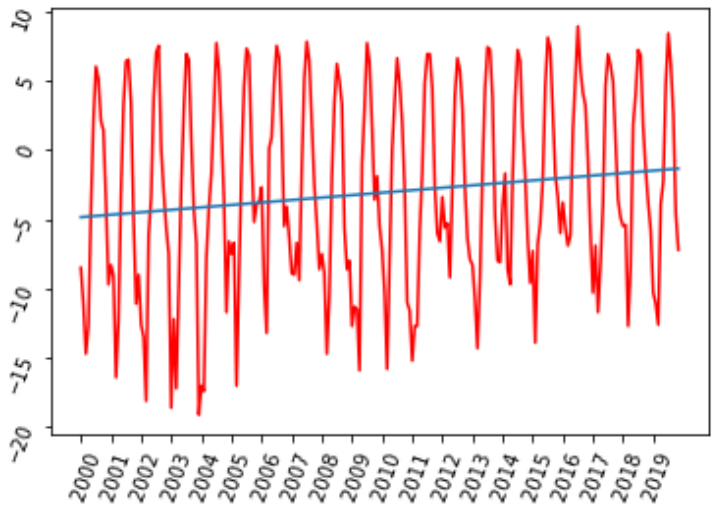
Lineær regresjon kan både sees på som prosessen å tilpasse en rett linje til et sett av punkter, men også som et parameterestimeringsproblem der vi leter etter stigningstallet og konstantleddet i en lineær modell.

6.2.3 Utledning av regresjonsformelen

Vi vil i dette avsnittet utlede regresjonsformelen i ligning (6.2.3) og ligning (6.2.3). Dette er analytiske løsningen av minimeringsproblemet for tilpassing av en rett linje til et sett av datapunkter.

Vi setter først opp de partieltderiverte av minste-kvadrat kostfunksjonen. Legg merke til at vi deriverer

[2] Det finnes en rekke andre pakker som er laget spesielt for å håndtere ulike former for regresjon og datanalyse. Vi vil i dette kompendiet fokusere på konseptet regresjon som et optimeringsproblem og ikke komme inn på funksjonalitet i andre pakker enn `numpy`. Den interesserte leser kan f.eks. slå opp i dokumentasjonen til pakken `pandas`



Figur 6.2.1: Målt temperatur ved Svalbard lufthavn fra januar 2000 til desember 2019. Den lineære trenden i observasjonene er lagt på i blå. Dette er et eksempel på lineær regresjon.

med hensyn på a_0 og a_1:

$$\begin{aligned}\frac{\partial}{\partial a_1}\sum_i (a_0 t_i + a_1 - y_i)^2 &= 2\sum_i t_i(a_0 t_i + a_1 - y_i)\\ \frac{\partial}{\partial a_0}\sum_i (a_0 t_i + a_1 - y_i)^2 &= 2\sum_i (a_0 t_i + a_1 - y_i)\end{aligned}$$

Setter hver av ligningene lik 0 og deler opp summene,

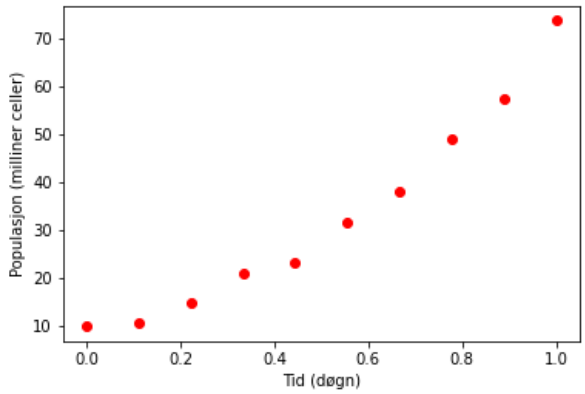
$$\begin{aligned}a_0\sum_i t_i^2 + a_1\sum_i t_i - \sum_i t_i y_i &= 0\\ a_0\sum_i t_i + n a_1 - \sum_i y_i &= 0\end{aligned}$$

Faktoren 2 blir dividert vekk etter at uttrykket er satt lik 0. Dette er et ligningssystem av to ukjente (a_0 og a_1) og to ligninger. Det kan løses på en rekke måter. Vi kan rydde i uttrykket ved å bruke definisjonen av gjennomsnitt $\bar{y} = \frac{1}{n}\sum_i y_i$ og $\bar{t} = \frac{1}{n}\sum_i t_i$. Deler vi den andre ligningen på n og løser for a_0. Vi får vi den andre delen av regresjonsformelen gitt i ligning (6.2.4). Setter vi dette inn i den øverste ligningen og løser for a_1 får vi første del av regresjonsformelen gitt i ligning (6.2.3).

6.2.4 Støy, måleunøyaktighet og lokale variasjoner

Som vi så i eksempelet over kan det være at observasjonene varierer mye mer enn det det er rom for i modellen. Dette kan være fordi modellen vår er litt for enkel, eller fordi vi ønsker å bruke en modell som ikke fokuserer på detaljer på fin skala.

De fleste observasjoner vil også være litt unøyaktige. Det kan være fordi måleredskapet er unøyaktig, eller fordi vi måler noe litt annet enn det vi faktisk ønsker. Selv om parameterne i f.eks. den lineære modellen kan bestemmes entydig med bare to observasjonspunkter vil det i p.g.a. måleunøyaktigheten være nyttig med flere observasjoner. Hvis vi har få observasjoner, vil eventuelle feil i disse observasjonene ha en stor innvirkning på optimeringsproblemet. Minste-kvadrat-formuleringen gjør at vi kan håndtere unøyaktige observasjoner, så lenge det er mange av dem og så lenge feilen har 0 som forventningsverdi, (altså at det er like sannsynlig at observasjonen er for lav som at den er for høy).



Tid (døgn)	Populasjon
0	9,8059
0,1111	10,3501
0,2222	14,7566
0,3333	20,8319
0,4444	23,2521
0,5556	31,3383
0,6667	38,0607
0,7778	48,8139
0,8889	57,2060
1	73,6929

Figur 6.2.2: Målinger av populasjonstørrelse i et kontrollert eksperiment med bakterievekst

6.2.5 Det ikke-lineære tilfellet

Vi skal nå se på situasjonen der vi forventer at det er en ikke-lineær utvikling av observasjonene våre. Vi så bla.a. eksempler på dette i forbindelse med differensialligningen i kapittel 4. I disse situasjonene er det ikke gitt at det finnes en enkel analytisk løsning på optimeringsproblemet (6.2.1). Dette problemet må da løses vha en numerisk optimeringsmetode, f.eks. gradient descent-metoden.

Vi trenger da gradienten til $F(\mathbf{a}) = \sum_i (U_i(\mathbf{a}) - y_i)^2$. Legg merke til at F er en funksjon av parametervektoren $\mathbf{a}$ og at vi partiellderiverer med hensyn på elementene i $\mathbf{a}$.

$$\nabla F(\mathbf{a}) = \begin{bmatrix} \frac{\partial F}{\partial a_1} \\ \frac{\partial F}{\partial a_2} \\ \vdots \\ \frac{\partial F}{\partial a_n} \end{bmatrix} = \begin{bmatrix} 2\sum_i (U_i(\mathbf{a}) - y_i)\frac{\partial U_i}{\partial a_1} \\ 2\sum_i (U_i(\mathbf{a}) - y_i)\frac{\partial U_i}{\partial a_2} \\ \vdots \\ 2\sum_i (U_i(\mathbf{a}) - y_i)\frac{\partial U_i}{\partial a_n} \end{bmatrix} = 2\sum_i (U_i(\mathbf{a}) - y_i)\nabla U_i(\mathbf{a}) \tag{6.2.5}$$

Vi har her derivert hver enkelt ledd i summen og brukt kjerneregelen. Vi ser at resultatet er avhengig av gradienten til modellen U_i. Vi må altså kunne derivere modellen med hensyn på modellparameterne for å gjøre optimeringen.

6.2.6 Eksempel - modell for bakterievekst

I en svært enkel modell for bakterievekst antar vi at den gjennomsnittlige delingstiden for cellene er konstant i tiden. Dette gjør at vekstraten til populasjonen er proporsjonal med størrelsen på populasjonen. Som vi har sett tidligere kan dette beskrives med differensialligningen

$$y'(t) = a_1 y(t).$$

For en kort tidsperiode kan dette være en rimelig modell. Den analytiske løsningen er gitt ved

$$y(t) = a_0 e^{a_1 t},$$

men uten noe mere informasjon vil vi ikke kunne bestemme parameterene a_0 og a_1.

Fra et kontrollert eksperiment er vi i stand til å måle størrelsen på en bakteriepopulasjon gjennom det første døgnet. Siden det er to parametre, vil det være tilstrekkelig å gjøre to målinger for å bestemme parameterne (og vi vil da ha en enkel analytisk løsning). Vi regner med at det er noe måleusikkerhet og gjør derfor 10 målinger jevnt fordelt i tid. De eksperimentelle resultatene er oppsummert i figur 6.2.2. Videre estimerer vi parameterne v.h.a. en miste kvadrat formulering (6.2.1).

Det vil være to måter å løse dette parameterestimeringsproblemet. Vi skal først se på den mest generelle, som vil fungere uavhengig av modell. Lengre nede går vi igjennom et triks som er nyttig for akkurat denne type modeller med eksponensiell vekst.

Løsning med gradient descent

Modellen vi skal sette inn i optimeringsproblemet bygger på den analytiske løsningen av differensialligningen, men har altså $\mathbf{a} = [a_0, a_1]$ som variable.

$$U_i(\mathbf{a}) = a_0 e^{a_1 t_i}$$

$$F(\mathbf{a}) = \sum_i (a_0 e^{a_1 t_i} - y(t_i))^2$$

$$\nabla U_i = \begin{bmatrix} e^{a_1 t_i} \\ t_i a_0 e^{a_1 t_i} \end{bmatrix}$$

En enkel kode for å bestemme modell-parameterne er gitt her:

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

# LASTER INN DATA
# Det ville her vært naturlig å laste målinger fra en csv-fil
# I dette eksempelet blir verdiene gitt dirkete som en array
y = np.array([9.8059,   10.3501,   14.7566,   20.8319,   23.2521, \
              31.3383,   38.0607,   48.8139,   57.2060,   73.6929])
t = np.linspace(0,1,10)  # 10 tidspunlkter mellom 0 og 1

#DEFINERER FUNKSJONER OG AKTUELLE VARIABLE
n = 100000 # velger max antall iterasjoner

# Definerer kostfunksjonen
def F(a,t,y):
    cost = np.sum((a[0] * np.exp(a[1]*t)-y)**2);
    return cost

# Definerer gradientfunksjonen
def grad_F(a,t,y):
      grad = 2* np.sum( (a[0] *np.exp(a[1]*t) - y) \
      * [np.exp(a[1]*t), t*a[0]*np.exp(a[1]*t) ],1);
      return grad

#Setter startverdiene (vilkårlig)
p0 = np.array([1,1])

tol = 10**-5 #setter toleranserverdien (for normen av gradienten)

#setter steglengden
gamma = .00002;

#BEGYNNER OPTIMERINEGN
p = p0 #initialliserer x og y

cost = F(p,t,y) #beregner kostfunksjonen

for i in range(n):
    p = p - gamma * grad_F(p,t,y) #oppdaterer x og y
    cost = np.append(cost, F(p,t,y))
    if np.absolute(cost[i-1]-cost[i]) < tol:
```

```
        break

print("a_0 = ",  p[0], ", a_1 = ", p[1]) # skriver ut svaret

#PLOTTER UTVIKLINGEN AV KOSTFUNKSJONEN
fig = plt.figure()
ax = fig.add_subplot(1,1,1)
plt.plot(cost)
plt.ylabel('Kostfunksjon')
plt.xlabel('# iterasjoner')
ax.set_yscale('log') #Benytter logaritmisk skala
```

I dette eksempelet har vi brukt optimering til å løse et parameterestimeringsproblem. Parameterne a_0 og a_1 er egenskaper ved bakteriekulturen miljøet de var i som ikke lar seg observere direkte.

Løsning v.h.a. eksponensiell regresjon

Akkurat dette eksempelet, med en modell for eksponensiell vekst, kan vha en liten omformulering løses med det analytiske utrykket for lineær regresjon i avsnitt 6.2.1. Trikset består ganske enkelt i å arbeide med logaritmen til observasjonene. Vi vil da forvente å få en lineær modell som passer inn i det lineære regresjonsutrykket

$$\begin{aligned} y(t) &= a_0 e^{a_1 t} \\ \ln y(t) &= \ln\left(a_0 e^{a_1 t}\right) = \ln a_0 + a_1 t \end{aligned}$$

I siste linje har vi altså en lineær modell. Legg merke til at den lineære modellen har $\ln a_0$ som parameter. Om vi kaller resultatet fra ligning (6.2.4) for A_0, så har vi at $\ln a_0 = A_0$ og $a_0 = e^{A_0}$. Parameteren a_1, trenger vi ikke transformere.

Diskusjon

Vil vi få eksakt likt resultat med de to løsningsmetodene (den optimeringsbaserte og den som benyttet lineær regresjon)? Hvorfor ikke?

6.2.7 Mer avanserte modeller

Det generelle rammeverket lar seg ganske enkelt utvide til mere kompliserte modeller med mange parametre. Vi vil imidlertid fortsatt trenge en analytisk modell og gradienten til modellen. I kapittelet om differensialligninger, så vi på modeller som bare hadde en numerisk, og ikke (kjent) analytisk løsning. Det vil også være mulig å bruke disse modellene til parameterestimering, men det krever litt andre algoritmer. Noen av disse algoritmene benytter en endelig differanse som tilnærming til de partielt-deriverte i gradienten.

Gradient descent metoden kan være svært treg for problemer av den typen vi så i eksempelt over. Særlig om de ulike parameterne i modellen har ulike roller og ulik størrelsesorden. Hvis det er få parametre spiller dette ikke så stor rolle. Hvis det er mange parametre (flere hundre eller flere tusen), vil dette imidlertid bli et problem. Det finnes flere spesielt tilpassede pakker for å gjøre både denne og andre typer optimering på en effektiv måte[3]

[3] Se f.eks. `scipy.optimize`

Kapittel 7

Hypotesetesting

Vi har tidligere sett hvordan vi kan laste inn observasjoner og målinger fra en data-fil (i kapittel 1) og hvordan vi analyserer dataene eller tilpasser en modellen vår etter dem (i kapittel 6). Samtidig vet vi at alle observasjoner og målinger er forbundet med noe usikkerhet eller unøyaktighet. Hvordan kan vi vite at analysen vi gjør, eller modellparametrene vi har beregnet ikke er ødelegges av denne usikkerheten eller unøyaktigheten. I eksempelet med temperaturer på Svalbard lufthavn i kapittel 1, så vi at middeltemperaturen mellom 1930 og 1950 var 2.48 grader lavere enn middeltemperaturen mellom 2000 og 2020. Var dette et resultat av en reel trend, eller var det en tilfeldighet forårsaket av variasjonen og unøyaktigheten i observasjonene? Til å avgjøre slike spørsmål trenger vi statistiske verktøy som hypotesetesting og konfidensintervall.

7.1 Konseptet hypotesetesting

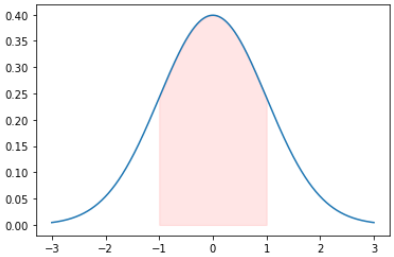
Utgangspunktet vårt ved hypotesetesting er at vi anerkjenner at det er knyttet usikkerhet til målingene våre og/eller at det er andre ukjente faktorer som påvirker det vi måler på en tilfeldig måte. Om vi for eksempel er interesset i en fenomen som vi forventer at skal ha en konstant verdi, så vil vi likevel observere en viss variasjon. Derfor velger vi å gjøre flere målinger/observasjoner (fordelt i tid eller rom) og beregne middelverdien av disse. Vi sier gjerne vi gjør et utvalg.

La oss si at det vi er interessert i å måle egentlig har verdien μ. Denne verdien er ukjent for oss og vi ønsker å finne den ved å gjøre flere observasjoner. Hvis vi betegner n observasjoner som y_i, $i = 1, 2 \cdots n$, så sier vi at vi estimerer μ vha middelverdien til observasjonene $\hat{y}$. Spørsmålet blir da i hvilken grad vi kan stole på dette estimatet. For å vurdere dette på en systematisk måte angriper vi problemet baklengs. Vi gjør først en antakelse om hva vi tror μ skal være. Gitt at antakelsen vår er sann, hvor sannsynlig vil det da være å observere de n observasjonene vi har gjort? Denne sannsynligheten kan også beregnes. Hvis sannsynligheten for å gjøre en disse n observasjonene er svært liten, så forkaster vi antakelsen om μ. Vi tror da at μ er noe annet. Hva vet vi imidlertid ikke. Om sannsynligheten for å gjøre disse n observasjonene er stor, så slår vi oss til ro med at antakelsen vår om μ er sann. Denne måten å tenke på kalles hypotesetesting. Dette kan også generaliseres til å gjelde for de fleste tilfeller der noe skal estimeres vha observasjoner.

Formelt sier vi at vi setter opp en *nullhypotese* og en *alternativ hypotese*.

Eksempel med temperatur

La oss si at vi antar at middeltemperaturen i juni måned er omtrent konstant fra år til år (f.eks. på 20°C). Dette er vår hypotese. Det vil si at hvis plukker ut to tilfeldig valgte år, så forventer vi at junitemperaturen er omtrent den samme. Samtidig vet vi at dette varierer litt fra år til år. Noen somre er litt kaldere og noen er litt varmere. Det er ikke helt umulig at vi har en rekord-sommer"med svært høye temperaturer 10 år på rad. Det er imidlertid ganske usannsynlig. Hvor mange rekord-sommere"må vi observere før vi forkaster hypotesen vår om at junitemperaturen er omtrent 20°C?



Figur 7.2.1: Standard normalfordeling med $\mu = 0$ og $\sigma = 1$. Toppen av fordelingen ligger ved $\mu = 0$. Om vi gjør en observasjon er det altså høyest sannsynlighet for at denne er μ. Det skraverte området går fra $-\sigma$ til σ. Dette utgjør 68.1% av arealet. Altså er det 68.1% sannsynlighet for å observere en verdi +/- ett standardavvik. Andre versjoner av normalfordelingen får vi ved å velge en annen μ og en annen σ. Dette kan gi oss en bredere eller smalere kurve med et annet toppunkt.

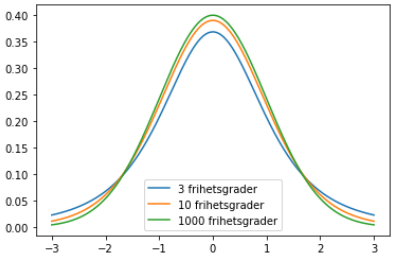
For å utføre en hypotesetest trenger vi først noen matematiske betrakninger om hvordan observasjonene våre varierer tilfeldig. Dette kan beskrives vha en sansynlighetsfordeling med en tilhørende sansynlighets-tetthetsfunksjon. Vi skal se på normalfordelingen og t-fordelingen.

7.2 Normalfordeling

I mange situasjoner observerer vi noe som varierer av en sammensatt grunn. Noe variasjon kan stamme fra egenarten til fenomenet, noe stammer fra en ukjent påvirkning fra andre fenomener og noe variasjon stammer gjerne fra usikkerheten i måleinstrumentet vårt. En typisk egenskap ved observasjoner som varierer av samensatte årsaker er at de får en normalfordeling. Årsaken til dette kan beskrives vha sentralgrenseteoremet. Vi skal ikke gå inn på detaljert teori knyttet til dette temaet i dette kompendiet. Vi konstaterer imidlertid at med mindre vi har spesiell grunn til å tro noe annet, så regner vi med at observasjonene våre varier med en tilnærmet normalfordeling. Normalfordelingen har flere godt kjente egenskaper. Den er f.eks. symetrisk rundt sin egen forventningsverdi, μ, og den streker seg uendelig langt i begge retninger. Det er altså mest sannsynlig at vi gjør observasjoner i nærheten av forventningsverdien, men det finnes en svært liten sannsynlighet for å observere både et vilkårlig stort og vilkårlig lite tall. Fordelingen kan beskrives vha forventningsverdien μ og et standardavvik σ. En standard normalfordeling (med $\mu = 0$ og $\sigma = 1$) er vist i figur 7.2.1.

7.3 t-fordeling

Selv om vi antar at fenomenene vi studerer følger en normalfordeling, er det en annen sannsynlighets-fordeling vi skal fokusere mest på i dette kapittelet, t-fordelingen. t-fordelingen brukes når vi skal gjøre observasjoner av noe som er normalfordelt. Å gjøre observasjoner blir det samme som å plukke ut et utvalg. La oss si vi gjør n observasjoner. Disse observasjonene kan vi bergene middelverdien av. Hvor sannsynlig er det at den middelverdien vi beregner er den samme som μ, den egentlige forventnings-verdien til den underliggende normalfordelingen? Med en viss sannsynlighet, er det jo mulig at vi f.eks. bare plukker ut verdier som er lavere enn μ. Det er imidlertid mer sannsynlig at vi plukker ut verdier som har en middelverdi svært nærme μ. Sannsynlighetsfordelingen til den estimerte middelverdien følger t-fordelingen. Denne fordelingen vil endre seg hvis vi endrer på utvalgstørrelsen. Hvis utvalget er veldig stort forventer vi at fordelingen vil være nesten lik normalfordelingen. Hvis utvalget er lite forventer vi at forelingen er bredere. Hvis vi f.eks. gjør et utvalg på 3, er det ikke veldig usannsynlig at alle verdier er lavere enn μ. I den andre enden vil det være mulig å vise at t-fordelingen går mot nomalfordlingen når utvalget er uendelig stort. Noen versjoner av t-fordelingen er vist i figur 7.3.1. Utvalgstørrelsen betegnes som antall frihetsgrader i fordelingen.



Figur 7.3.1: t-fordeling med 3, 10 og 1000 frihetsgrader. Versjonen med 1000 frihetsgrader vil være tilnærmet lik normalfordelingen. For lavere antall frihetsgrader har fordelingen litt lavere topp og litt tykkere haler.

Frihetsgrader

Antallet frihetsgrader er en mengde som angir antall observasjoner som inngår i bergeningen minus antall parametre som skal beregnes. I de fleste tilfeller skal vi bergene *en* parameter, middelverdien, fra n observasjoner. Antall frihetsgrader blir da $n-1$. I eksempelet til slutt i dette kapittelet har vi bruk for frihetsgradene i en estimering av de to parameterne i en lineær regresjon. I dette tilfellet vil antall frihetsgrader være $n-2$, siden vi estimerer to parametere. Betraktningen om antallet frihetsgrader er viktigst hvis antallet observasjoner et lavt. I tilfellet med $n=1$ vil antall frihetsgrader være 0. Vi kan fortsatt representere et estimat for f.eks. middelverdien, men vi har ingen formening om denne ene observasjonen representerer den sanne μ på en god måte. For å si noe om dette må vi ha minst en observasjon til. I tilfellet med regresjon må vi ha to observasjoner for å kunne definere en linje (fordi en lineær funksjon er avhengig av to parametre). Vi må imidlertid ha minst en til for å ha en mening om denne linjen representerer den sanne underliggende trenden (vi ser da etter om det tredje punktet også ligger på linjen). Har vi mange fler punkter, har vi en bedre formening om linjen er riktig.

7.4 t-test

En t-test er ment som et verktøy for å beregne signifikansen av et utvalg. Selv om det fenomenet vi er interessert i antas å ha en normalfordeling, så vet vi gjerne ikke hvordan denne normalfordelingen ser ut. Altså hva μ og σ er. For å si noe om dette må vi gjøre et utvalg y_i (f.eks. ved å ta noen målinger). Vi lager et estimat av μ ved å beregne middelverdien av utvalget, $\hat{y}$. Den enkleste form for t-test tar utgangspunkt i t-fordelingen og vurderer hvor sannsynlig det er at vi observerer denne middelverdien *g*itt at den egentlige middelverdien er μ_0. Om det er svært usannsynlig å gjøre et slik utvalg for denne μ_0-verdien, kan vi trekke konklusjonen om at den egentlige μ verdien sannsynligvis ikke er μ_0.

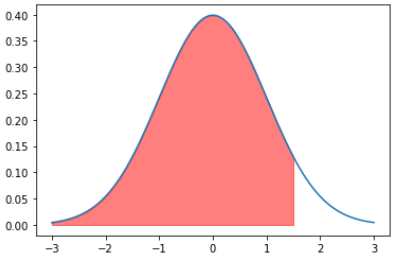
Formulert som en hypotesetest blir dette:
Nullhypotese: Middelverdien μ er μ_0
Alternativ hypotese: Middelverdien μ er ulik μ_0.

7.5 p-verdi, signifikans og signifikansnivå

Fra en sansynlighetstetthetsfunksjon, slik som i figur 7.3.1, kan vi lese av hvor sannsynlig det er å å gjøre en observasjon med en gitt verdi. Det enkleste tilfellet er om vi faktisk kjenner μ og σ. Dette er enten godt studerte situasjoner, eller situasjoner der vi har en god matematisk modell (som feks. summen av mange terningkast o.l.). Hvis vi skal gjøre en enkelt observasjon, y, kan vi skrive sannsynligheten for at denne observasjonen er mindre enn en verdi y_1 som

$$P(y < y_1).$$

Sansynligheten beregnes som integralet av sansynlighetstetthetsfunksjonen fra $-\infty$ til y_1. Altså arealet under kurven frem til y_1. Dette illustreres i figur 7.5.1.



Figur 7.5.1: Sansynligheten $P(y < 1.5)$. Sansynligheten er gitt som arealet av det skraverte området.

Når vi jobber med hypotesetesting og t-tester, ser vi egentlig baklengs på problemet. Vi gjør først observasjoner, og spør oss etterpå hvor sannsynlig det var at vi fikk nettopp disse målingene, gitt den hypotesen vi satte opp. Denne sannsynligheten betegnes med en *p-verdi*. En høy p-verdi betyr at det er høy sannsynlighet får å få disse observasjonene. Det tyder altså på at hypotesen vår er riktig. En lav p-verdi betyr at sannsynligheten for å få disse observasjonene er lav. Det kan altså tyde på at hypotesen vår er feil og at vi bør gå for alternativ-hypotesen. Hvor vi setter grensen α mellom høy og lav p-verdi, betegnes som vårt *signifikansnivå*. Begrepet signifikans beskriver i hvor stor grad vi forventer at et resultatet er tilfeldig. Alternativet er da at resultatet ikke er tilfeldig, men beskriver en reel sammenheng eller forskjell. Det er vanlig å sette α til f.eks. 0.95 eller 0.99. Med $\alpha = 0.95$ sier vi at vi godtar resultater som er 95% sannsynlige eller mer, gitt vår hypotese. Disse resultatene gir oss altså ingen grunn til å forkaste null-hypotesen. For vi et resultat som er mindre sannsynlig enn $1 - \alpha$, altså 5%, bestemmer vi oss for å forkaste null-hypotesen. Det er imidlertid viktig å merke seg at det fortsatt er mulighet for at nullhypotesen er riktig og at observasjonen vår bare var en tilfeldighet. Om vi skal bli enda mere siker, må vi øke signifikansnivået til f.eks. $\alpha = 0.99$. Helt sikrere blir vi imidlertid aldri. Det er viktig å merke seg at p-verdien ikke er sannsynligheten for at null-hypotesen er riktig, men altså sannsynligheten for å gjøre de observasjonene vi har gjort, gitt at nullhypotesen er sann.

7.6 Ensidige og tosidige tester

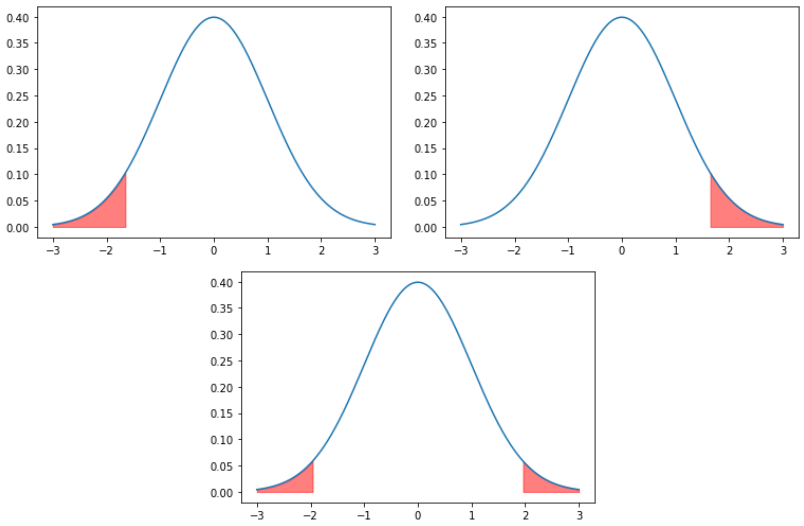
Til nå har vi i hovedsak fokusert på null-hypotesen. Null-hypotesen inneholder alltid en likhet, altså hva vi tror μ, (eller den parameteren vi ønsker å måle), er. Alternativ-hypotesen kan naturlig formuleres på tre ulike måter som alle er en kontrast til nullhypotesen.
Nullhypotese: $\mu = \mu_0$

1. Alternativ hypotese: $\mu \neq \mu_0$
2. Alternativ hypotese: $\mu > \mu_0$
3. Alternativ hypotese: $\mu < \mu_0$

Hvilke alternativ hypotese vi velger vil være avhengig av situasjonen vi skal skal undersøke. Den første alternativ hypotesen, $\mu \neq \mu_0$, gir oss en tosidig test. Med det mener vi at både utvalg som er signifikant mindre enn μ_0 og utvalg som er signifikant større enn μ_0 fører til at vi forkaster null-hypotesen. De to siste alternative hypotesene gir oss ensidige tester. Forskjellen på de tre tilfellene er illustrert i figur 7.6.1. De skarverte feltene utgjør i dette eksempelet 5% av arealet under kurven (altså signifikansnivå $\alpha = 0.95$). En t-verdi som havner innenfor de skraverte feltene tilsier at vi velger å forkaste null-hypotesen. Legg merke til at det at testen er en- eller tosidig spiller inn på hvordan vi går fra t-verdi til p-verdi. Standard tabeller for t-verdier er satt opp for ensidige tester, så om testen vår er tosidig, må vi gange p-verdien vi beregner med 2.

7.7 Ulike versjoner at t-testen

Avhengig av situasjonen vi er i kan vi formulere t-testen på ulike måter. De forskjellige formuleringene gir oss et uttrykk for t-verdien til testen. Denne t-verdien må igjen legges inn i t-fordelingen for å få den



Figur 7.6.1: De skreverte feltene tilsvarer områder der vi forkaster nullhypotesen med tre ulike tilfeller av alternativhypotesen. Det første tilfellet tilsvarer alternativ-hypotese $\mu < \mu_0$, det andre tilsvarer alternativ-hypotese $\mu > \mu_0$ og det siste tilsvarer $\mu \neq \mu_0$ som gir oss en tosidig test.

aktuelle p-verdien. Formelen for å gå fra t-verdi til p-verdi er ganske komplisert. Tradisjonelt har man derfor slått dette opp i en tabell. I praksis er denne tabellen implementert i `Python`[1].

7.7.1 Et-utvalgs t-test

Et-utvalgs t-test er den grunnleggende ideen beskrevet i avsnittet over. Testen brukes når vi gjør et utvalg av størrelse n for å undersøke om en populasjon har middelverdi μ_0. Aktuell nullhypotese er: $\mu = \mu_0$

t-verdien beregnes med følgende uttrykk:

$$t = \frac{\hat{y} - \mu_0}{\hat{\sigma}/\sqrt{n},}$$

der $\hat{\sigma}$ er standardavviket i utvalget vårt.

7.7.2 To-utvalgs t-test

Som navnet sier tar denne testen utgangspunkt i to utvalg. Vi antar at utvalgene er av samme størrelse n. Spørsmålet vi ønsker å besvare er om de to utvalgene har samme middelverdi. Aktuell nullhypotese er da: $\mu_1 = \mu_2$

Hvis vi benevner den estimerte middelverdien av de to utvalgene som $\hat{y}^1$ og $\hat{y}^2$, så bergenes t-verdien som:

$$t = \frac{\hat{y}^1 - \hat{y}^2}{s_k\sqrt{2/n}},$$

der s_k er et kombinerte standavviket gitt som

$$s_k = \sqrt{\frac{\hat{\sigma}^2_{y^1} + \hat{\sigma}^2_{y^2}}{2}}.$$

[1]Denne funksjonaliteten ligger bl.a. i pakken `scipy.stats`.

7.7.3 Test av stigningstall i regresjon

Det vil ofte være nyttig å vurdere om et stigningstall fra en regresjonsberegning er signifikant ulik 0. Dette er en form for et-utvalgs t-test, der vi i stede for å teste for middelverdi tester parameteren a_1 i regresjonsuttrykket er 0. For å forenkle uttrykket litt kaller vi det reelle, ukjente stigningstallet for β og det estimerte stigningstallet for $\hat{\beta}$. Vi har tidligere jobbet med y som en funksjon av tiden, men for å unngå sammenblanding med t-verdier, ser vi nå på y som en funksjon av x. Vi har altså gjort regresjonen vår med observasjoner y_i i punkter x_i.
Aktuell nullhypotese er: $\beta = \beta_0$

Uttrykket for t-verdien er gitt som:

$$t = \frac{\hat{\beta} - \beta_0}{SE_{\widehat{\beta}}}.$$

$SE_{\widehat{\beta}}$ er standard-feilen for estimeringen av β og estimeres med:

$$SE_{\widehat{\beta}} = \frac{\sqrt{\frac{1}{n-2}\sum_{i=1}^{n}\left(y_i - \widehat{y}_i\right)^2}}{\sqrt{\sum_{i=1}^{n}\left(x_i - \widehat{x}\right)^2}}$$

Antall frihetsgrader: Antall observasjoner minus antall parametre som skal estimeres, altså $n - 2$.

7.8 Kriterier for bruk av t-test

Hovedkriteriet som må være oppfylt for at t-testene som er beskrevet over skal være gyldige er at $\hat{y}$ må være normalfordelt. Altså: om vi gjør veldig mange forskjellige grupper av utvalg av samme størrelse, så vil middelverdien av disse gruppene følge en normalfordeling.

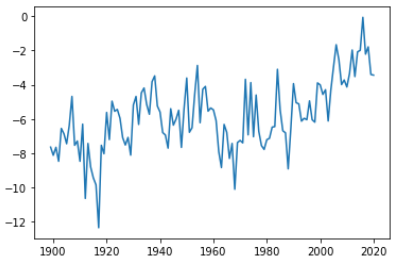
For to-utvalgs tester har vi også at:

- De to utvalgene er normalfordelte og vi forventer at de har like standardavvik.
- De to utvalgene er like store
- Det er ikke overlapp i de to utvalgene.

7.9 Eksempel - temperatur på Svalbard lufthavn I

I dette eksempelet tar vi utgangspunkt i temperaturdataene fra Svalbard Lufthavn som vi lastet inn i kapittel 1. Vi skal dele observasjonene i to like store deler, den første fra 1899 til 1959 og den andre delen fra 1960 til 2020. For å forenkle behandlingen av temperaturdataene skal vi først gå fra gjennomsnittlig månedstemperatur til gjennomsnittlig årstemperatur. Dette blir gjort i scriptet under.

```
"Fortsetter fra script i Kap. 1 ... Finner antall år tidsserien dekker ved å dele på 12. Siden
serien ikke starter å slutter samme måned, runder vi av  nedover med np.floor()"
n = np.floor(tempr.size/12)
n = int(n) #Verdien må defineres som int for å kunne brukes som en indeks

offset = 4 #Tidsserien starter i september, men vi vil starte i januar

aarlig_middel = np.zeros(n)

for i in range(n):
    et_enkelt_aar = tempr[i*12+offset:i*12+offset+12] #plukker ut 12 mnd.
    aarlig_middel[i] = np.mean(et_enkelt_aar) #bergener middelverdien

aarstall = np.arange(1899.,2021.) #definerer aarstallene som en array
plt.plot(aarstall,aarlig_middel) #plotter den nye tidsserien
```



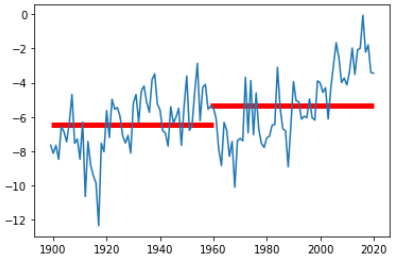
Figur 7.9.1: Årlig middeltemperatur på Svalbard Lufthavn.

Scriptet returnerer plottet i figur 7.9.1

Vi ønsker å undersøke om middeltemperaturen for den andre perioden er høyere enn middeltemperaturen i den første. Dette gir oss en ensidig, to-utvalgs t-test. Vi setter $\alpha = 0.95$

Nullhypotese: Middeltemperaturen i perioden 1899 til 1959 er lik middeltemperaturen i perioden 1960 til 2020.
Alternativ hypotese: Middeltemperaturen i perioden 1960 til 2020 er høyere enn middeltemperaturen i perioden 1899 til 1959
Hypotesetesten blir utført i scriptet under:

```
"""
Fortsetter fra scriptet over...

#laster en pakke som vi trenger for å finne p-verdier
from scipy import stats

Vi skal nå dele tidsserien i to og teste om den andre halvdelen har
signifikant hoyere middelverdi enn den forste
"""
tempr1 = aarlig_middel[0:n//2] # henter ut data forste halvdel
# Bruker // i stede for / slik at resultatet blir int
m1 = np.mean(tempr1) # beregner gjennomsnittet

#plotter gjennomsnittet som en linje
plt.hlines(m1,aarstall[0],aarstall[n//2],colors='r',linewidth=5)


tempr2 = aarlig_middel[n//2:] # henter ut data fra andre halvdel
m2 = np.mean(tempr2) # beregner gjennomsnittet
plt.hlines(m2,aarstall[n//2],aarstall[-1],colors='r',linewidth=5)

# lengden av tidsseriene (der er like lage)
N = tempr1.size

# beregner variansen for de to tidsseriene
var_temp1 = tempr1.var(ddof=1)
var_temp2 = tempr2.var(ddof=1)

#beregner t-verdien
s = np.sqrt((var_temp1 + var_temp2)/2)
t = (m1 - m2)/(s*np.sqrt(2/N))

# setter antall frihetsgrader
df = 2*N - 1
```



Figur 7.9.2: Svalbard Lufthavn. Årlig middeltemperatur i perioden 1899 til 1959 og perioden 1960 til 2020

```
#finner p-verdien tilhørende den beregnede t-verien
#Dette tilsvarer å slå opp i en tabell
p =  stats.t.cdf(t,df=df)

print("t = " + str(t))
print("p = " + str(p))


# Vi kan også sammenligne med den innebygde t-testen i pakken scipy.stats
# Deler p verdien pa to da testen returnerer verdi for tosidig test.
t2, p2 = stats.ttest_ind(tempr1,tempr2)
print("t = " + str(t2))
print("p = " + str(p2/2))
```

Scriptet returnerer plottet i figur 7.9.2 og verdiene:

```
t = -3.2420575305732013
p = 0.0007682873292877338
t = -3.2420575305732013
p = 0.0007682873292877338
```

Vi har her beregnet p og t-verdien to ganger. Dette er bare for å bekrefte at utregningene vår med `numpy` samsvarer med de forhåndsimplementerte metodene i `scipy.stats`.

Vi ser at $p < 0.05$ hvilket betyr at vi forkaster nullhypotesen.

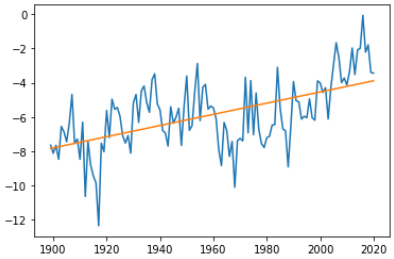
7.10 Eksempel - temperatur på Svalbard lufthavn II

En alternativ tilnærming til testen i forrige eksempel er å utføre en regresjon på hele tidsserien, for så å teste om stigningstallet er signifikant større enn 0. Vi lar fortsatt $\alpha = 0.95$.

Nullhypotese: Regresjonslinjen har stigningstall 0
Alternativ hypotese: Regresjonslinjen har stigningstall > 0.

Hypotesetesten blir utført i scriptet under:

```
#fortsetter fra scriptet over
#lager en regresjonslinje
y = aarlig_middel
n = np.size(y)
t = aarstall
y_bar = np.mean(y)
t_bar = np.mean(t)
a_1 = (np.sum(t*y)-n*t_bar*y_bar)/np.sum((t-t_bar)**2)
a_0 = y_bar - a_1*t_bar

def reg_line(a_0 ,a_1 ,t):
    return a_1*t+a_0
```



Figur 7.10.1: Svalbard Lufthavn. Årlig middeltemperatur i perioden 1899 til 2020 med regresjonslinje

```
plt.plot(t,reg_line(a_0,a_1,t))

#beregner standard-feil og t-verdi
SE = np.sqrt((1/(n-2))*np.sum( (y-y_bar)**2  ))/np.sqrt(np.sum( (t-
    t_bar)**2 ))

t_verdi = a_1/SE

df = n-2 #frihetsgrader

p =  1-stats.t.cdf(t_verdi,df=df)
print("t = " + str(t_verdi))
print("p = " + str(p/2))
```

Scriptet returnerer plottet i figur 7.10.1. og verdiene:

```
t = 6.206608313480839
p = 2.0026829083974462e-09
```

Siden $p < 0.05$ forkaster vi nullhypotesen. Vi legger merke til at p er svært liten. Det er altså svært usannsynlig at regresjonslinjen vår er positiv ved en tilfeldighet.

Bibliografi

[1] Martin G. Gulbrandsen, Johannes Kleppe, Tore A. Kro, and Jon Eivind Vatne. *Matematikk for ingeniørfag - med numeriske beregninger*. Gyldendal Akademisk, 1 edition, 2013.

[2] Tor Gulliksen, Amir M. Hashemi, and Arne Hole. *Matematikk i praksis*. Universitetsforlaget, 6 edition, 2013.

[3] Timothy Sauer. *Numerical Analysis*. Pearson, 2 edition, 2014.

[4] Jon Eivind Vatne. Numeriske metoder i MATLAB - Tillegg til pensum i MAT102. 2017.

[5] Øyvind Nordli, Rajmund Przybylak, Astrid E.J. Ogilvie, and Ketil Isaksen. Long-term temperature trends and variability on spitsbergen: the extended svalbard airport temperature series, 1898–2012. *Polar Research*, 33(1):21349, 2014.