

Spredning av sykdommer – Et matematisk blikk

av Imran Ali

imala015@osloskolen.no

Mot slutten av året 2019 ble det oppdaget en sykdom spredt av koronavirus, kalt Covid-19, i Wuhan provinsen i Kina. Viruset var antatt først å ha blitt overført fra dyr til mennesker, den spredte seg etter hvert til andre deler av verden. Sykdommen fører til, for de fleste tilfeller, milde symptomer. For noen få kan det føre til lungebetennelse og organsvikt, som i verste fall kan lede til død. I noen land var tiltak som sosial distansering gjennomført senere enn nødvendig. Dette førte til at veldig mange mennesker spredte sykdommen. Helseinstitusjoner hadde vansker med å håndtere mange smittede samtidig, ofte grunnet mangel på medisinskutstyr og sykehusplasser. For å hjelpe smittede som trengte hjelp med å puste, behovet for ventilatorer (maskiner som hjelper mennesker med å puste ved å pumpe oksygen) og annet nødvendig medisinsk utstyr var absolutt nødvendig. Mangel av slik utstyr førte til høy dødsrate blant de smittede.

Det viser seg at tiltak som sosial distansering, hvis gjennomført tidlig nok, minker smitteraten (det vil si hvor fort en smittsom sykdom sprer seg) og sykehus har dermed nok tid til å skaffe seg medisinsk utstyr for å håndtere større andel syke mennesker. Vi skal her studere hvordan matematiske modeller for sykdommer kan benyttes til å kartlegge hvor raskt et virus kan spre seg i en populasjon. Dette er informasjon som er viktig for regjeringer å kunne planlegge tiltak for å hindre at helsevesenet kan bli overbelastet. For å forstå slike modeller trenger vi å ha en viss forståelse for noe matematikk og programmering. Noe av matematikken vi skal først se på handler om funksjoner og den deriverte av en funksjon.

Funksjoner

I matematikk er en funksjon en relasjon mellom par av verdier $(x, f(x))$, der for hver x -verdi har vi kun en funksjonsverdi $f(x)$. Enklere sagt, tar en funksjon et tall og gjør noe med det tallet. En funksjon som kan skrives som $f(x) = ax + b$ kaller vi for lineær funksjon. Vi skal se på noen eksempler på hvordan vi kan tegne slike funksjoner i et koordinatsystem.

Eksempel: $f(x) = 2x + 4$

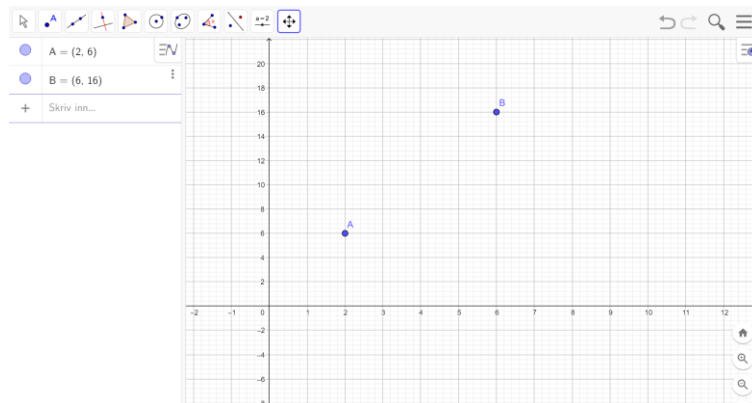
For å kunne tegne slike funksjoner trenger vi minimum to punkter, dvs. to par med $(x, f(x))$. Hvis vi for eksempel setter inn verdien 2 for x , får vi da

$$f(2) = 2 \cdot 2 + 4 = 8$$

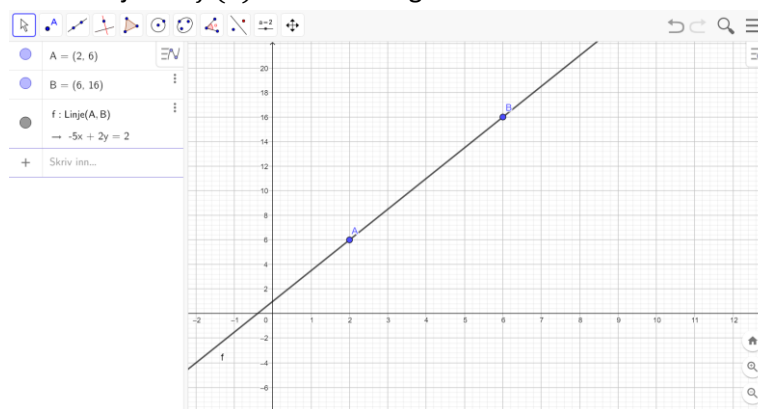
Tilsvarende, hvis vi nå setter f.eks. verdien 6 for x :

$$f(6) = 2 \cdot 6 + 4 = 16$$

Da har vi to punkter eller koordinater $(2, 8)$ og $(6, 16)$. Dette forteller oss at punktet $(2, 8)$ ligger 2 plasser fra origo (et spesielt punkt i koordinatsystem $(0, 0)$) i første aksene (x -aksen) og 8 plasser vertikalt langs y -aksen. Hvis vi plasserer disse to punktene i et koordinatsystem, vil det se slik ut





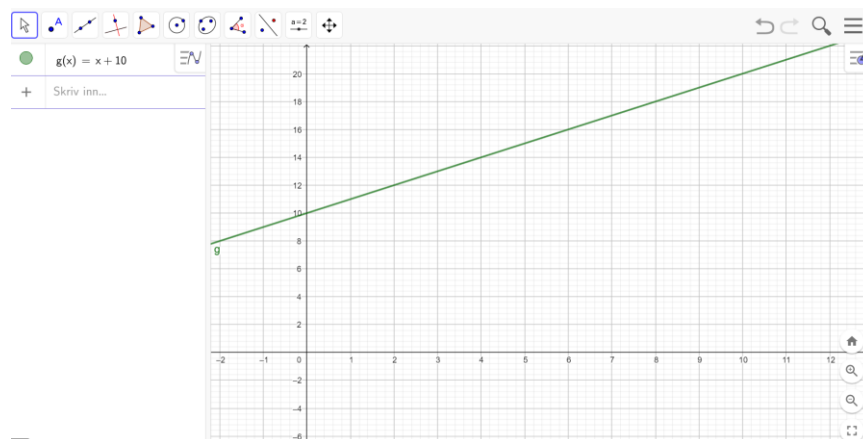
Her har vi tegnet punktene i et koordinatsystem i GeoGebra. For å tegne grafen til $f(x) = 2x + 4$, må det lages en linje som går gjennom punkt A og punkt B. Den rette linja vil da beskrive funksjonen $f(x) = 2x + 4$ grafisk.



Eksempel: $g(x) = x + 10$

Det finnes flere måter vi kan tegne grafen på. Hvis vi bruker en program som GeoGebra, så er det mulig å skrive funksjonen i algebrafeltet og da vil grafen lages av GeoGebra automatisk.

Aksene kan justeres ved å bruke «Flytt knappen»  og «Flytt grafikkfeltet knappen» . På denne måten kan det vises akkurat det som trengs for å tolke grafen. For å nullstille aksene, da kan hjem-knappen trykkes nederst til høyre på skjermen.

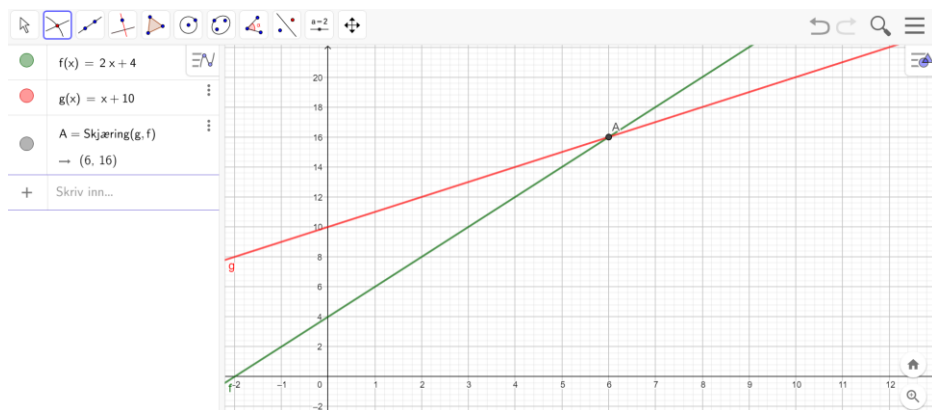


Eksempel: Skjæringspunkter

Vi skal nå tegne $f(x) = 2x + 4$ og $g(x) = x + 10$ i samme koordinatsystem og finne hvor de skjærer hverandre, eller hvor de deler samme par av $(x, f(x))$ koordinater. I matematikk kalles dette å finne **skjæringspunkt** (noen funksjoner kan ha flere). Først begynner vi med å tegne grafene i GeoGebra:



Hvis vi deretter velger «Skjæring mellom to objekter» i punktmenyen, kan vi trykke på begge grafene og da vil et punkt plasseres der grafene skjærer hverandre:



Leser vi av fra algebrafeltet, ser vi at koordinat for skjæringspunktet er (6, 16). Det vil si at grafene skjærer hverandre i $x = 6$ og de har funksjonsverdien 16. For å bekrefte at dette stemmer, setter vi x verdien i begge funksjonen:

$$f(6) = 2 \cdot 6 + 4 = 16$$



$$g(6) = 6 + 10 = 16$$

Dermed har vi fått bekreftet at det faktisk stemmer.

Oppgave 1:

- a) Fyll verditablellen for funksjonen $f(x) = 5x - 5$.

x	f(x)
-1	$f(-1) = 5 \cdot (-1) - 5 = -5 - 5 = -10$
0	
1	
2	

- b) Tegn funksjonen $f(x)$ for hånd og i GeoGebra.
- c) Tegn funksjonen $f(x) = 5x - 5$ og $g(x) = -4x + 20$ i samme koordinatsystem.
Obs! Husk å velg en passende fremvisning av funksjonen. Bruk «Flytt knappen»  og «Flytt grafikkfeltet knappen»  for å få til dette.
- d) Finn skjæringspunktet til funksjonene $f(x) = 5x - 5$ og $g(x) = -4x + 20$.

Derivasjon

Den deriverte av en funksjon viser hvor mye en funksjon forandrer seg i løpet av et steg i den første aksene. Vi skriver ofte den deriverte av en funksjon $f(x)$ som $f'(x)$ eller $\frac{df}{dx}$. Apostrofen til $f'(x)$ sier at det er første deriverte av $f(x)$. Dobbeltderiverte kan vi da skrive som $f''(x)$, det vil si med to apostrofer, og så videre. I eksemplene vi har sett på hittil, har vi sett på lineære funksjoner av typen $g(x) = ax + b$. Den deriverte av slike funksjoner gir oss $g'(x) = a$. Vi kaller a for **stigningstallet** til lineære funksjoner.

Eksempel: Finn stigningstallet til $f(x) = 2x - 5$ og $g(x) = -x - 5$.

Stigningstallet blir da tallet som står foran x:

$$f'(x) = 2$$

$$g'(x) = -1$$

Det finnes mange regler for den deriverte av en funksjon. Vi skal forholde oss til så kalte polynomfunksjoner, dvs. funksjoner som kan skrives som polynomer: $f(x) = ax^2 + bx + c$. Dette er et annengrads polynom fordi det høyeste tallet til eksponenten til x er 2. Da ser vi også at lineære funksjoner er førstegrads polynomer fordi x er opphøyet med 1.

Regel 1: Derivasjon av konstanter (tall)

Den deriverte av et tall er alltid 0.

Regel 2: Derivasjon av polynomer

Hvis $f(x) = ax^n$, der a er en konstant og n er et reelt tall, ulik null, da er den deriverte av funksjonen f

$$\frac{df}{dx} = a \cdot n \cdot x^{n-1}.$$

Eksempel: Finn den deriverte til funksjonen $f(x) = x^4$.

$$f'(x) = 4 \cdot x^{4-1} = 4x^3$$

Eksempel: Finn den deriverte til funksjonen $f(x) = 2x^3 + 5$.

$$f'(x) = 2 \cdot 3 \cdot x^{3-1} + 0 = 6x^2$$

Eksempel: Finn den deriverte til funksjonen $f(x) = 2x^4 + 5x^2$.

$$f'(x) = 2 \cdot 4 \cdot x^{4-1} + 5 \cdot 2 \cdot x^{2-1} = 8x^3 + 10x$$

Oppgave 2: Finn den deriverte til følgende funksjoner

- a) $f(x) = 2x + 4$
- b) $f(x) = x^2$
- c) $f(x) = 6x^4$
- d) $f(x) = x^{99}$
- e) $f(x) = 2x^4 + 2x^2 + 3$ (Hint: Bruk reglene på hvert ledd)
- f) Finn den dobbelt deriverte til $f(x) = 6x^4$. (Hint: deriver svaret du fikk i d) en gang til)

Numerisk derivasjon

Vi har nå sett at vi kan derivere funksjoner ved hjelp av ulike regler. Men for å kunne gjøre det på en datamaskin må vi gjøre det på et annet vis. På en datamaskin er det nemlig ikke mulig å representere funksjoner med uendelig mange punkter. Vi velger isteden et gitt antall punkter og prøver heller å finne funksjonsverdier for de deriverte som ligger nær. Å gjøre dette kaller vi for *numerisk derivasjon*. Det finnes mange ulike metoder for å tilnærme funksjonen fra dens deriverte. Her skal vi lære om den såkalte «forward Euler» metoden, også kalt Euler metoden – oppkalt etter matematikeren [Leonhard Euler](#).

Euler metoden:

For funksjonen $f(x)$ kan vi skrive dens deriverte som

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i}$$

der $x_{i+1} - x_i$ er avstanden mellom to x verdier og $f(x_{i+1})$ og $f(x_i)$ er funksjonsverdiene i x_{i+1} og x_i respektivt.

Med en slik metode vil det alltid være et feil som følger for hvert ledd med x_i verdier vi regner ut $f'(x_i)$. Så lenge vi lar avstanden mellom x_i 'ene -og intervallet der vi regner ut funksjonsverdiene være «liten nok», kan vi anta at også feilen er liten.

Eksempel: Bruk Euler metoden på ligningen $f'(x) = 4x^3$.

Hvis vi nå tar utgangspunkt i dette uttrykket og ser på det som en ligning, kan vi finne en numerisk løsning for hvordan funksjonen $f(x)$ oppfører seg. La oss nå erstatte x med x_i og $f'(x)$ med uttrykket fra Euler metoden.

$$\frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i} = 4x_i^3$$

Vi prøver nå å løse dette uttrykket som en ligning der $f(x_{i+1})$ er ukjent, da får vi at

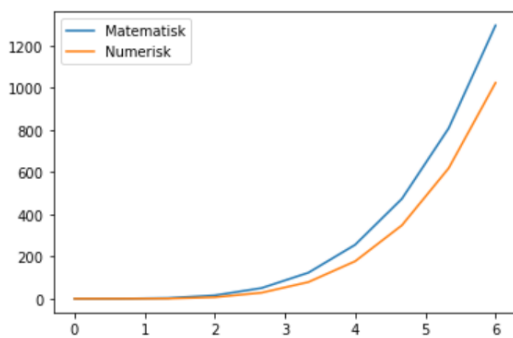
$$f(x_{i+1}) - f(x_i) = 4x_i^3 \cdot (x_{i+1} - x_i)$$

$$f(x_{i+1}) = f(x_i) + 4x_i^3 \cdot (x_{i+1} - x_i)$$

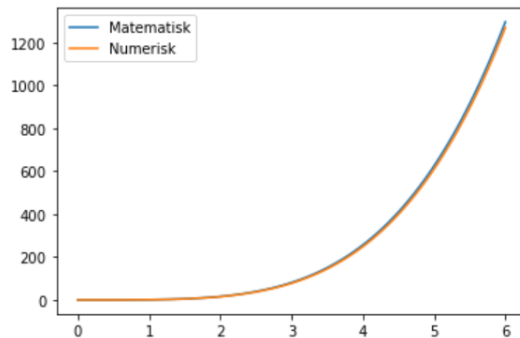
Hvis vi antar at det er alltid lik avstand mellom x_i 'ene, kan vi la $x_{i+1} - x_i = h$, da ender vi til slutt med uttrykket:

$$f(x_{i+1}) = f(x_i) + 4 \cdot h \cdot x_i^3.$$

Vi fant i et tidligere eksempel at den deriverte av $f(x) = x^4$ var $f'(x) = 4x^3$. La oss bruke dette resultatet for å sammenligne det numeriske resultatet med den faktiske løsningen $f(x) = x^4$:



10 x-verdier



100 x-verdier

Vi kan se at når vi regner ut for få x -verdier, sliter Euler metoden med å «ta igjen» den *sanne løsningen*. Men når vi øker antall x -verdier, da virker Euler metoden så godt at det er nesten ikke mulig å skille mellom den matematiske løsningen og den numeriske.

Dette kan kanskje virke litt overveldende i starten. Her lønner det seg å øve seg med oppgaver. Først skal vi se på et nytt eksempel, så skal du få prøve deg selv på å bruke Euler metoden.

Eksempel: Bruk Euler metoden til finne et uttrykk for ligningen $f'(x) = 6x^2$.

Først så erstatter vi x med x_i og $f'(x)$ med $\frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i}$:

$$\frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i} = 6x_i^2$$

Så løser vi for den ukjente $f(x_{i+1})$. For å gjøre det må gange begge sider og likhetstegnet med $x_{i+1} - x_i$. Hvis vi gjør dette, ender vi opp med uttrykket

$$f(x_{i+1}) - f(x_i) = 6x_i^2 \cdot (x_{i+1} - x_i)$$

$$f(x_{i+1}) = f(x_i) + 6x_i^2 \cdot (x_{i+1} - x_i)$$

Vi husker fra forrige eksempel at hvis vi antar at avstanden mellom $x_{i+1} - x_i$ er lik, kan vi velge å sette $x_{i+1} - x_i = h$.

$$f(x_{i+1}) = f(x_i) + 6h \cdot x_i^2$$

Da er vi ferdig og vi har klart å lage et uttrykk for ligningen ved hjelp av Euler metoden.

Oppgave 3: Følg de siste to eksemplene og finn et uttrykk for følgende ligninger

- a) $f'(x) = 5x^4$
- b) $f'(x) = 24x^3$
- c) $f'(x) = 5x^4 + 24x^3$

Kortere notasjon: For å spare plass når vi uttrykker slike ligninger numerisk er det fint å kunne skrive det på en mer kompakt måte. La $f(x_i) = f_i$, da kan vi skrive det siste eksempelet på følgende måte:

$$f_{i+1} = f_i + 6h \cdot x_i^2$$

På denne måten er det lettere å håndtere den forrige oppgaven.

Oppgave 4: Gjør forrige oppgave på nytt, denne gangen skal du bruke den kortere notasjon.

Oppgave 5: Du skal nå tegne løsningen til $f'(x) = 5x^4$ og sammenligne tegningen med Euler metoden. Løsningen til $f'(x) = 5x^4$ viser seg å være $f(x) = x^5$.

a) Fyll ferdig verditabellen når vi lar $h = 0.5$, x ligger i intervallet $[0, 5]$ og $f(x_0) = 0$.

x_i	f_{i+1}
$x_0 = 0$	$f_1 = f_0 + 5h \cdot x_0^4 = 0 + 5 \cdot 0.5 \cdot 0^4 = 0$
$x_1 = 0.5$	$f_2 = f_1 + 5h \cdot x_1^4 = 0 + 5 \cdot 0.5 \cdot 0.5^4 = 0.15625$
$x_2 = 1$	$f_3 = f_2 + 5h \cdot x_2^4 = 0.15625 + 5 \cdot 0.5 \cdot 1^4 = 2.65625$
$x_3 = 1.5$	$f_4 = f_3 + 5h \cdot x_3^4 = 2.65625 + 5 \cdot 0.5 \cdot 1.5^4 = 15.3125$
$x_4 = 2$	$f_5 = f_4 + 5h \cdot x_4^4 = 15.3125 + 5 \cdot 0.5 \cdot 2^4 = 55.3125$
$x_5 = 2.5$	
$x_6 = 3$	
$x_7 = 3.5$	
$x_8 = 4$	
$x_9 = 4.5$	
$x_{10} = 5$	

b) Fyll verditabellen for $f(x) = x^5$

x	$f(x)$
$x = 0$	$f(0) = 0^5 = 0$
$x = 0.5$	$f(0) = 0.5^5 = 0.03125$
$x = 1$	$f(0) = 1^5 = 1$

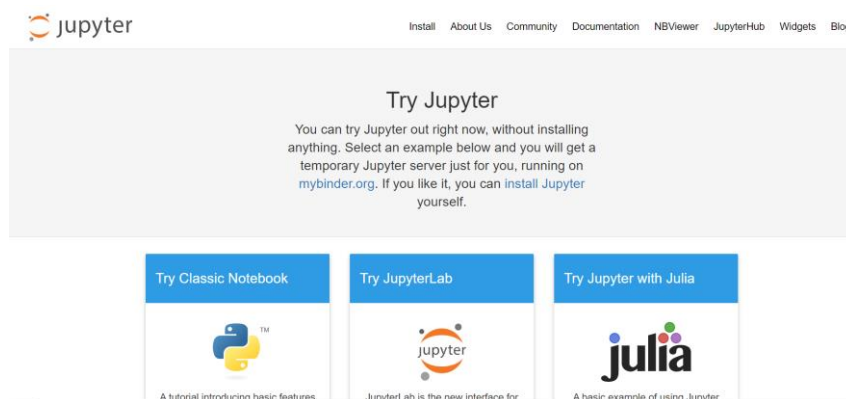
$x = 1.5$	
$x = 2$	
$x = 2.5$	
$x = 3$	
$x = 3.5$	
$x = 4$	
$x = 4.5$	
$x = 5$	

- c) Tegn for hånd i et koordinatsystem den numeriske løsningen og den matematiske løsningen, der x ligger i intervallet $[0, 5]$.

Jupyter

Hittil har vi brukt verditabell for å finne verdier til f_i , det vil si funksjonsverdi $f(x_i)$ til å kunne tegne grafer for hånd. Dette er en jobb datamaskin er veldig flink til å gjøre. Derfor skal vi nå se hvordan vi kan bruke programmering til å la datamaskinen utføre beregningene og vi kan isteden da ha søkelys på å tolke grafer den lager for oss. For å få til dette skal vi bruke programmeringsspråket Python. Alt arbeid vi skal gjøre skal skje gjennom en nettleser, slik kan du da slippe å installere ting på din egen datamaskin eller iPad!

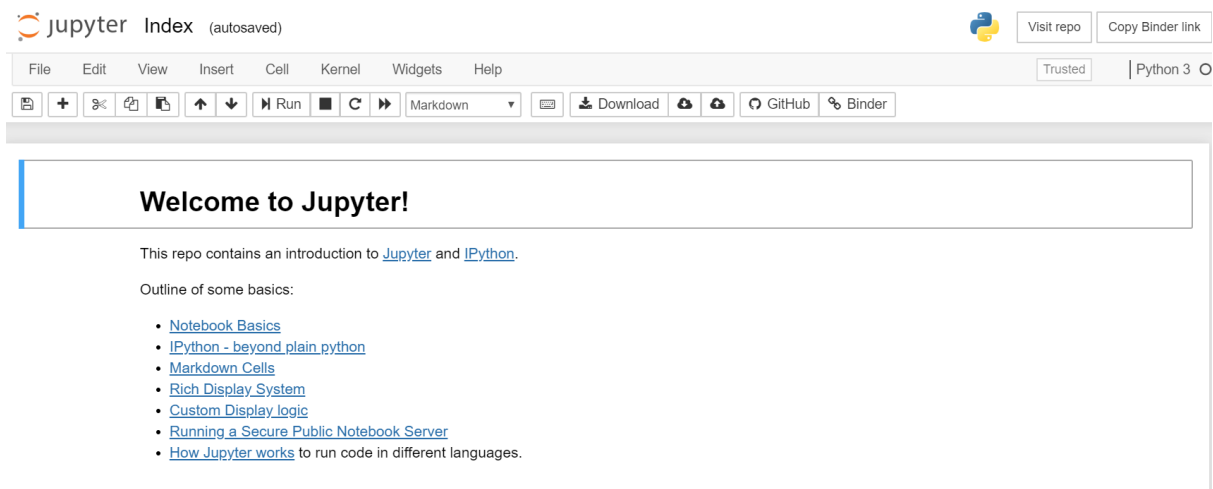
La oss først besøke følgende nettside: <https://jupyter.org/try>



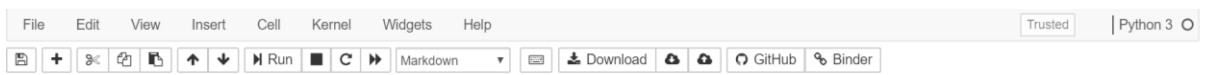
Vi skal kun forholde oss til «Try Classic Notebook». Trykk på lenken. Når du gjør det vil du få følgende på din skjerm






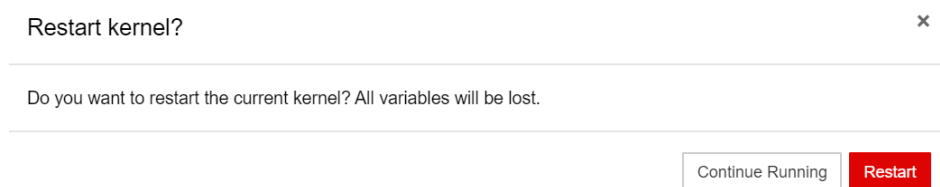
Nå må du vente til du ser «Welcome to Jupyter!» på skjermen.



Vi må nå først bli litt kjent med menyen:

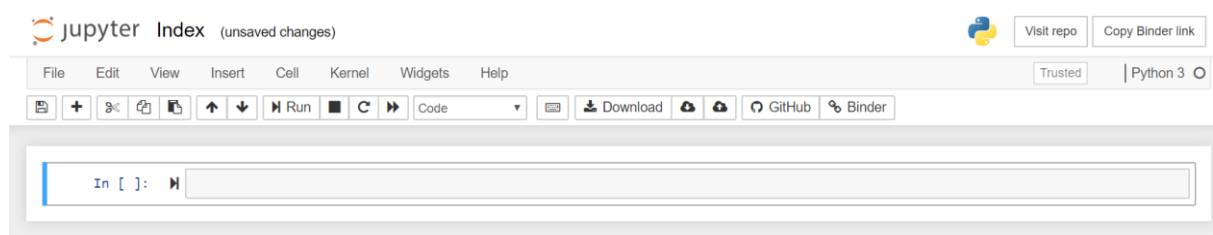


Her er det tre knapper som er viktig å være klar over hva de gjør. Først **saksknappen** . Saksknappen fjerner celler. Celler er der vi skriver vår kode, i mindre ruter. Når vi har skrevet noe kode i en celle, trenger vi å kjøre den eller teste den. For å gjøre det skal du da bruke **kjør-knappen** (eller run) . Noen ganger kan nettsiden virke som om den sitter fast og koden din vil ikke kjøre. For å fikse dette kan det være lurt å trykke på **restart-knappen** .



Dessverre vil det hende iblant at selv restart-knappen heller ikke hjelper til å løse problemet med å kjøre koden. Ofte skjer dette ved å være inaktiv for lenge. Da blir du nødt til å gå tilbake nettsiden <https://jupyter.org/try> og starte på nytt. Hvis du er redd for å miste koden din, så kan det være lurt at du lagrer det et sted, for eksempel i en wordfil mens du jobber underveis. Et siste alternativ, hvis det er problemer med «Try Classic Notebook» er å bruke «Try JupyterLab». Denne videoen viser hvordan du kan da skrive din kode: <https://vimeo.com/415492421>

Da er vi nesten klare til å starte! Begynn med å trykke på saksknappen til alt tekst under er borte fra skjermen. Du skal kun stå igjen med en tom celle



Verdier, variabler og regneoperasjoner

Oppgave 6: I den tomme cellen skriv følgende

```
a = 5.2
b = 4.8
print(a + b)
```

Bruk linjeskift etter hver setning. Når du har skrevet koden inn i cellen, trykk på «Run» knappen for å kjøre din første skript. Pass på at du slutter parenteser til print.

Det du gjorde i denne oppgaven kalles tilordning av *verdier* til *variabler*. Verdier kan være tall, bokstaver eller tekst. Variabler er et sted i datamaskinen sin hukommelse der den lagrer en plass for verdien. Et sted i en datamaskin har det blitt laget plass for variabel *a* med tallverdi 5.2.

Vi skal ikke gå i detaljer på hvordan en datamaskin kan lagre tall i sin hukommelse, men en ting som er viktig å være klar over er at i motsetning til den reelle tallinja, kan ikke en datamaskin klare å representere desimaltall der det kreves uendelig mange desimaler.

Eksempel:

Tallet pi

```
3.1415926535897932384626433832795028841971693993
75105820974944592307816406286208998628034825342
1170679.....
```

har uendelig mange desimaler som bare fortsetter og fortsetter. Dette tallet kan vi aldri representere eksakt i en datamaskin. Isteden lagrer datamaskinen kun et visst antall siffer i sin hukommelse.

I Python brukes `*`, `/`, `+` og `-` som de fire regnearter til å gange, dele, addere og subtrahere. Videre brukes `**` for å lage potenser. For å bruke f.eks. kvadratrøtter må vi ty biblioteker i Python.

Bruk av biblioteker

Ofte, når vi skal lage våre egne *skripter* (korte kodesnutt) eller store programmer, så finnes det mange *biblioteker* andre mennesker har allerede. I disse bibliotekene vil vi kunne finne smarte og gode løsninger som vi kan ta med i vår kode. På denne måten så slipper vi å finne opp hjulet på nytt.

Oppgave 7: I en tom celle skriv følgende og deretter «Run»

```
from math import pi, sqrt
print(pi)
print(sqrt(100))
```

Obs! Du må passe på at du har akkurat riktig antall parenteser. Ellers vil ikke din kode kjøre.

La oss se på denne oppgaven steg for steg. Første setning

```
from math import pi, sqrt
```

Vi har hentet to ting fra biblioteket som heter **math**. Den første er en konstant som heter *pi*, mens den andre er kvadratrot *funksjon* som finner røtter til reelle positive tall.

```
print(pi)
print(sqrt(100))
```

I Python er funksjon en kodesnutt som tar imot en variabel og gjør noe med den. I dette tilfellet tar **sqrt** imot tall verdien 100 og gir tilbake kvadratroten av 100, som er 10. En måte du kan se at du bruker en funksjon i Python er ved å se på parenteser som følger rett etter en bokstav eller et ord. For eksempel ved å se på

```
print(pi)
```

så kan vi da vite at **print** er en funksjon i Python som tar imot noe og printer det på skjermen. I dette tilfellet printer den ut tallverdien til konstanten pi.

Feilmelding og «bugs»

En ting du må kanskje venne deg til vi lager kode, det vil mest sannsynlig alltid ved en eller annen tidspunkt introduseres feil. Noen ganger vil du få en melding fra Python om at «dette går ikke, her er feilen». Eller andre ganger vil ikke Python klage og du vil isteden få rare resultater fra din kode. Det sist nevnte er det vi kaller for bugs. Det kan kanskje hende at du har allerede opplevd dette underveis i denne gjennomgangen? Vi skal bare ta en kjapp titt på feilmeldinger i Python og prøve å se om det kan gi oss pekepinn på hva som har gått galt.

Oppgave 8: I en tom celle skriv følgende og deretter «Run»

```
from math import sqrt
print(sqrt(-1))
```

Beskriv hva som blir nevnt i feilmeldingen når du trykker «Run» og hva du faktisk tror er feilen.

Vi tar denne oppgaven også sammen. Men før vi fortsetter, så er det lurt at du prøver selv først før du leser dette.

Når du trykker «Run» så kommer følgende feilmelding

```
In [1]: from math import sqrt
        print(sqrt(-1))

-----
ValueError                                Traceback (most recent call last)
<ipython-input-1-a7e58f78be2f> in <module>
      1 from math import sqrt
----> 2 print(sqrt(-1))

ValueError: math domain error
```

Hvis du husker tilbake til hva som ble nevnt, så tar funksjonen **sqrt** kun positive reelle tallverdier. Siden vi forsøker å finne kvadratroten av et negativt tall så ender vi opp med å få en feilmelding i Python fra funksjonen. Den prøver å fortelle oss at det er ikke mulig å finne roten av et negativt tall.

Kvadratrøtter av negative tall finnes ikke i den reelle tallinja. Men derimot finnes de i det komplekse tallplanet. Så hvis du faktisk hadde lyst til å regne ut kvadratrøtter til negative tall, så kan du faktisk få det til. Vi kan nå da hente kvadratfunksjon fra et annet bibliotek, **cmath**:

```
In [1]: from math import sqrt
print(sqrt(-1))

-----
ValueError                                Traceback (most recent call last)
<ipython-input-1-a7e58f78be2f> in <module>
      1 from math import sqrt
----> 2 print(sqrt(-1))

ValueError: math domain error

In [2]: from cmath import sqrt
print(sqrt(-1))

1j
```

I *celle 2* har vi hentet **sqrt** fra biblioteket **cmath** og den klarer faktisk å regne ut kvadratroten til tallet minus 1. Det svaret den gir, 1j, er et såkalt kompleks tall eller imaginært tall.

For loop (løkker i Python)

Når du trenger å gjenta en operasjon flere ganger kan en for-løkke (for loop på engelsk) den perfekte verktøy for å få gjort en slik oppgave. La oss se på et eksempel der vi forsøker å regne ut ulike funksjonsverdier for $f(x) = 2x - 3$ i Python. Først la oss lage en verditabell for funksjonen:

x	$f(x) = 2x - 3$
x = 0	$f(0) = 2 \cdot 0 - 3 = -3$
x = 1	$f(1) = 2 \cdot 1 - 3 = -1$
x = 2	$f(2) = 2 \cdot 2 - 3 = 1$
x = 3	$f(3) = 2 \cdot 3 - 3 = 3$
x = 4	$f(4) = 2 \cdot 4 - 3 = 5$
x = 5	$f(5) = 2 \cdot 5 - 3 = 7$
x = 6	$f(2) = 2 \cdot 6 - 3 = 9$

La oss nå se litt nærmere på denne tabellen. Legg merke til at det eneste tallet som forandrer seg for alle funksjonsverdier vi regner ut er plassen til x. Den blir byttet ut med tallet til kolonnen til venstre. Vi gjør i grunn samme utregning igjen og igjen. Slike oppgaver er akkurat hva for-løkker er laget for å utføre. Dette er den perfekte jobben for en datamaskin.

Strukturen til en for løkke kan beskrives som:

for element i liste:

<gjør noe>

En liste i Python kan lages ved å bruke hakeparenteser:

[0,1,2,3,4,5,6]

Denne lista består av verdiene fra første kolonne i verditabellen. Hvis ser på strukturen til en for-løkke så kan vi nå bytte ut uttrykkene med følgende:

```
for x in [0,1,2,3,4,5,6]:
    print(x)
```

Dette er en fungerende forløykke. Legg merke til at print funksjonen har et innrykk. Python er et såkalt innrykk språk. I Python deles kode opp etter innrykk. I Jupyter får du automatisk innrykk etter at du taster kolon og trykker linjeskift.

Oppgave 9:

- a) I en tom celle lag for-løykka

```
for x in [0,1,2,3,4,5,6]:  
    print(x)
```

Hva kommer ut?

- b) Nå skal du printe ut funksjonsverdiene istedenfor. I en tom celle lag forløykka, men print isteden funksjonsverdier for $2x - 3$. Pass på at du bruker gange tegn $*$ mellom 2 og x, dvs. `print(2*x - 3)`.

range: Siden det er veldig vanlig å kjøre for-løykker over et intervall med heltall, finnes det en rask og enkel måte å gjøre det på:

```
range(start, slutt, steg)
```

Dette sier til funksjonen range at den skal lage en liste fra en start verdi til en sluttverdi med et gitt steg mellom hvert heltall. Hvis det ikke blir gitt en start verdi, da antas det automatisk at start verdien er 0. Og hvis det heller ikke gis et steg, da antas det at steg er 1 mellom hvert tall. Hvis vi nå lager for-løykka igjen med range, da blir den

```
for x in range(0, 7, 1):  
    print(x)
```

Legg merke til hvis vi setter sluttverdien til å være 6 da vil range ha siste verdi 5. Hvis vi velger 7 som sluttverdi, da vil til og med 6 være inkludert.

Legg merke til at

```
for x in range(7, 1):  
    print(x)
```

og

```
for x in range(7):  
    print(x)
```

gir akkurat samme resultat:

```
for x in range(7):  
    print(x)
```

```
0  
1  
2  
3  
4  
5  
6
```

Oppgave 10: Lag en forløykke med range og print ut funksjonsverdier for $2x - 3$ for x fra 0 til og med 6 med steg 1.

Oppgave 11: Lag en forløkke med range og print ut funksjonsverdier for $x^2 - 2$ for x fra -2 til og med 2 med steg 1. (For å opphøye x i 2 kan det skrives som x^{**2} i Python.)

Plotting (graftegning)

Foreløpig har vi klart å printe ut verdiene til en funksjon. Noen ganger trenger vi en visualisering for å lettere forstå og tolke funksjoner. For å lage slike visualiseringer trenger vi å ta vare på de verdiene vi printer ut. Men vi trenger mange flere verdier hvis vi skal kunne lage grafer av funksjoner. For å få til dette trenger vi å bruke noen biblioteker hvor alle verktøyene vi trenger er tilgjengelig.

Numpy: Dette biblioteket vil hjelpe oss å ta vare på verdiene vi lager med forløkker som vi senere kan bruke for å lage gode visualiseringer. Disse verdiene blir lagret i såkalte «**array**» (utt. eng.).

Matplotlib: Dette biblioteket bruker vi for verdier vi tar vare på av numpy array's til å visualisere de i grafer.

Når vi skal lage grafer av funksjoner, så trenger vi også x-verdier som ligger mellom heltallene. På denne måten kan vi sikre oss at funksjoner som vi visualiserer ser mer «mykere» ut. For å få til dette bruker vi funksjonen **linspace** som ligner på range, men den gir oss også desimaltall. Den brukes på følgende måte: `linspace(start, slutt, antall)`

```
from numpy import linspace
a = linspace(0, 8, 10)
print(a)
```

Dette gir følgende utskrift:

```
[0.         0.88888889 1.77777778 2.66666667 3.55555556
 4.44444444 5.33333333 6.22222222 7.11111111 8.]
```

Dermed ser vi at **linspace** lager 10 verdier, med lik avstand imellom verdiene, fra 0 og til og med 8. I motsetning til **range**, inkluderer **linspace** til og med sluttverdien.

Oppgave 12: Lek litt med følgende kode. Prøv å kjør koden for ulike n verdier mellom 10 og 1000. Du trenger å kun endre n verdien i linje 2 og da vil automatisk n verdien få tilordnet den nye verdien i linje tre.

```
from numpy import linspace
n = 10
a = linspace(0, 8, n)
print(a)
```

Obs! Pass på hvis du kopier koden herfra at det ikke er noen innrykk når du limer inn i Jupyter.

Oppgave 13: Bruk **linspace** til å lage følgende intervaller med passende antall verdier. Husk at du kan importere kvadratroten og pi fra **math** biblioteket:

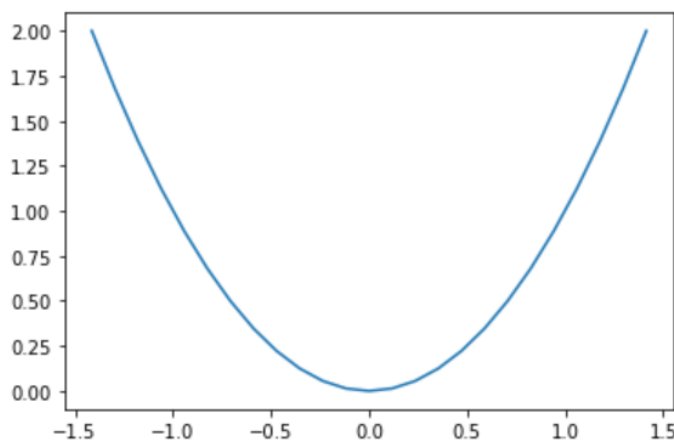
```
from math import sqrt, pi
a)  $x \in [0, 2\pi]$ 
```

b) $x \in [-\sqrt{2}, \sqrt{2}]$

Nå er vi klare til å lage noen fine plotter! La oss begynne å se hvordan vi kan lage en graf for funksjonen $f(x) = x^2$, for $x \in [-\sqrt{2}, \sqrt{2}]$.

```
from numpy import linspace, sqrt
from matplotlib.pyplot import plot, show
```

```
x = linspace(-sqrt(2), sqrt(2), 25)
plot(x, x**2)
show()
```



Som navnene tilsier, **plot** funksjonen lager grafen og **show** funksjonen gjør grafen synlig på skjermen. I `plot(x, x**2)` er første argumentet `x` verdiene til x-aksen, og andre argumentet `x**2` funksjonsverdiene for $f(x) = x^2$. To gange-tegn brukes, som vi har sett tidligere, for å fortelle Python at vi vil opphøye `x` med en eksponent. I `linspace(-sqrt(2), sqrt(2), 25)` ber vi funksjonen lage 25 verdier med lik avstand mellom $-\sqrt{2}$ og $\sqrt{2}$. Legg også merke til at vi ikke har brukt **math** biblioteket til å hente kvadratrots funksjonen **sqrt**. Isteden viser det seg at **numpy** har også kvadratrots funksjonen `sqrt`. Dette er selvfølgelig helt det samme. Faktisk har numpy også tallet **pi** tilgjengelig.

Oppgave 14: Lag graf av følgende funksjoner med passende antall punkter i **linspace**.

- a) $f(x) = x^3$, for $x \in [-2, 2]$.
- b) $f(x) = x^4 - x^2$, for $x \in [-\sqrt{2}, \sqrt{2}]$.
- c) $f(x) = -x^4 + x^3 + x^2 - x$, for $x \in [-\sqrt{2}, 1.82]$.

Implementasjon av Euler metode

Vi skal nå ta en titt på følgende ligning:

$$f'(x) = x, \text{ for } x \in [0, 5],$$

$$f(0) = 5.$$

Vi bruker først Euler metoden på ligningen

$$\frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i} = x_i$$

$$f(x_{i+1}) - f(x_i) = x_i \cdot (x_{i+1} - x_i)$$

$$f(x_{i+1}) = f(x_i) + x_i \cdot (x_{i+1} - x_i)$$

Lar vi $h = x_{i+1} - x_i$, og bruker **den korte notasjonen**, så får vi at

$$f_{i+1} = f_i + h \cdot x_i$$

Hittil har vi laget tabeller for å finne verdiene til funksjonene og laget grafen for hånd. Dette er den perfekte jobben for datamaskinen. Nå skal vi gjøre dette med Python.

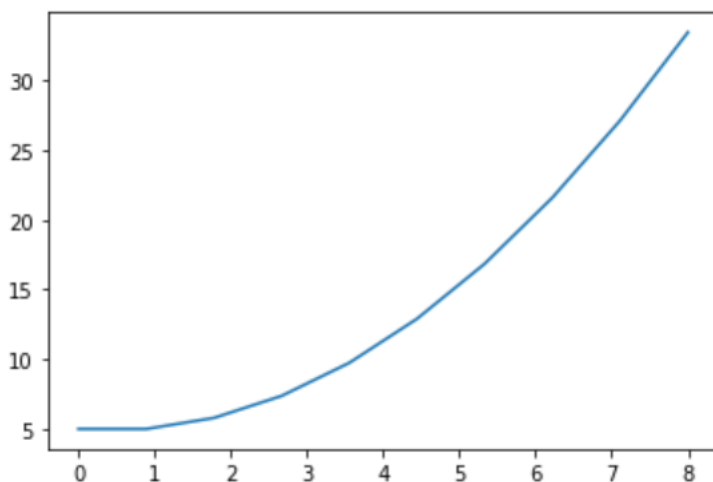
```
from numpy import linspace, zeros
from matplotlib.pyplot import plot, show
```

```
n = 10
x = linspace(0, 8, n)
h = x[1] - x[0]
```

```
f = zeros(n)
f[0] = 5
```

```
for i in range(0, n-1):
    f[i+1] = f[i] + h*x[i]
```

```
plot(x, f)
show()
```



La oss se litt nærmere på dette skriptet.

```
n = 10
x = linspace(0, 8, n)
h = x[1] - x[0]
```

Her har vi satt 10 verdier mellom 0 og 8 med lik avstand. Vi har satt vår h til å være $x_1 - x_0$. Vi bruker hakeparenteser fordi x er en liste med tall, og 0 er første plass i lista, mens 1 er neste plass i lista.

```
f = zeros(n)
f[0] = 5
```

Her har vi brukt en ny funksjon **zeros** fra biblioteket **numpy**. Dette lager en tom liste som har n elementer med tallverdi 0. I dette tilfelle er n lik 10 og derfor har denne lista ti 0 verdier. Hensikten vår med å lage denne lista er å ta vare på alle verdiene vi lager i vår forløkke. Som første verdi setter

vi den til å være 5, slik det vi har fått vite i oppgaven, nemlig at $f(0) = 5$. Dette kalles en **start betingelse** eller **initialverdi**. Neste steg er å kjøre Forward-Euler eller Euler metoden i forløkka:

$$f[i+1] = f[i] + h \cdot x[i]$$

Legg merke til hvor nært dette ligner på vår utregning:

$$f_{i+1} = f_i + h \cdot x_i$$

Dette er en av styrkene til programmeringsspråket Python. Vi sier at den har en nærmest matematisk skrivestil eller å gå fra matematikk til kode er relativt enkelt. Dette er noe veldig få programmeringsspråk som klarer.

La oss studere problemstillingen litt nærmere og se om den løsningen vi har fått numerisk er nøyaktig. Vår ligning var

$$f'(x) = x, \text{ for } x \in [0, 8], \\ f(0) = 5.$$

Slike ligninger, hvor den deriverte er involvert, kalles for **differensiallikninger**. Siden det er kun den første deriverte her, kalles denne ligningen for **førsteordens differensiallikning**. For å se om koden vår gir en god tilnærming for $f(x)$ skal vi sammenligne den numeriske løsningen med den eksakte løsningen

$$f(x) = \frac{1}{2}x^2 + 5$$

Vi kan se at dette passer helt perfekt med vår ligning fordi

- 1) Hvis vi deriverer $f(x)$ ser vi at vi ender opp med:

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

- 2) Hvis vi finner funksjonsverdien for $f(x)$ når $x = 0$ og den er 5, da har vi vist at løsningen stemmer:

$$f(0) = \frac{1}{2} \cdot 0^2 + 5 = 5.$$

Der ser vi at vi har endt opp tilbake til vår problemstilling. La oss nå undersøke hvor god tilnærming vår numerisk løsning har for ulike antall x verdier eller når vi bytter n med ulike verdier.

```
from numpy import linspace, zeros
from matplotlib.pyplot import plot, show, legend

n = 10
x = linspace(0, 8, n)
h = x[1] - x[0]

f = zeros(n)
f[0] = 5
f_losn = 0.5*x**2 + 5

for i in range(0, n-1):
    f[i+1] = f[i] + h*x[i]

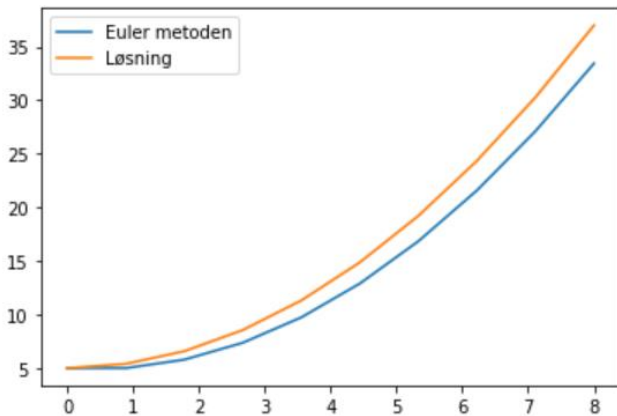
plot(x, f, x, f_losn)
```

```
legend(['Euler metoden', 'Løsning'])
show()
```

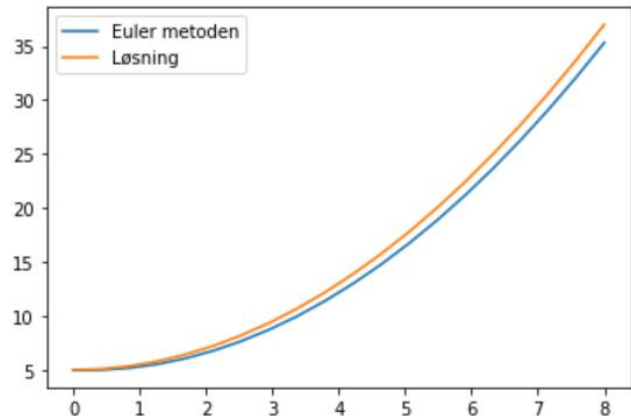
Vi kan se at når vi kjører skriptet (se grafene under) for ulike antall x verdier (bestemt av variabel **n**) begynner Euler metoden å komme nærmere den sanne løsningen $f(x) = \frac{1}{2}x^2 + 5$. Denne oppførselen hvor den numeriske løsningen nærmer seg den eksakte løsningen når vi øker n kalles **konvergering**. Vi sier at den numeriske løsningen **konvergerer** til den eksakte løsningen når n øker. For å lage disse grafer ble skriptet kjørt med de oppgitte n verdiene.

Før vi går videre, trenger vi å ta et raskt blikk på den nye koden i skriptet. Nemlig følgende linjer:

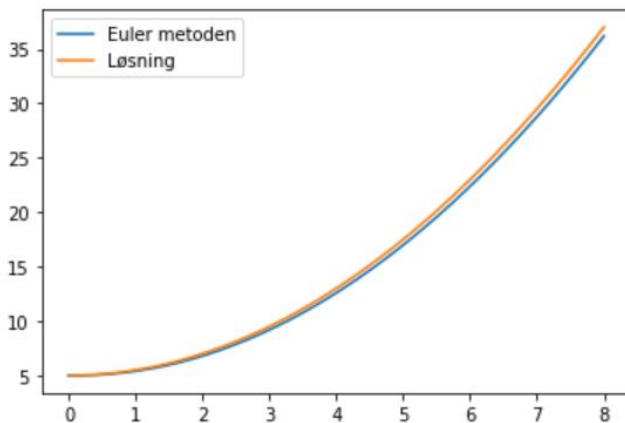
```
f_losn = 0.5*x**2 + 5
og
plot(x, f, x, f_losn)
legend(['Euler metoden', 'Løsning'])
```



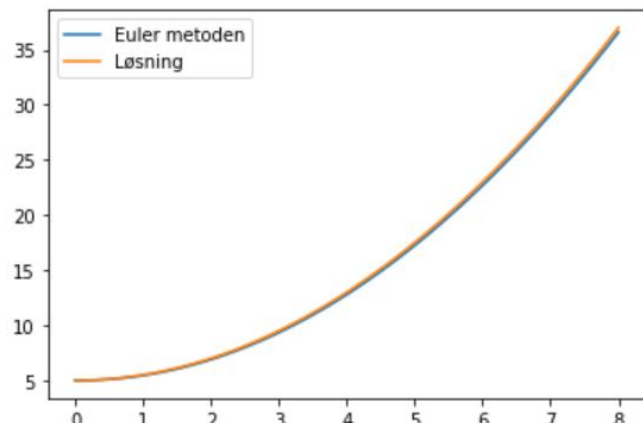
For n = 10



For n = 20



For n = 40



For n = 80

Den øverste linja her er funksjonen vår eller den eksakte løsningen. For å lage de nye grafene med den numeriske og den eksakte, har vi lagt inn i samme grafvindu ved å skrive

```
plot(x, f, x, f_losn)
```

I **plot** funksjonen har vi inkludert begge grafene. Funksjonen tar lager grafene for samme x- og y-akse, men med `f` (numeriske) og `f_losn` (eksakte). Til slutt har vi en ny funksjon **legend** fra **matplotlib**. Denne funksjonen har som oppgave å sette navn på våre grafer slik du kan se fra bildene.

Oppgave 5: I denne oppgaven skal du løse følgende problemstilling

$$f'(x) = x^2, \text{ for } x \in [0, 5],$$
$$f(0) = -2.$$

- a) Bruk Euler metoden og skriv et uttrykk for $f(x_{i+1})$.
- b) Vis at følgende er en eksakt løsning av ligningen

$$f(x) = \frac{1}{3}x^3 - 2.$$

- c) Lag en skript for å løse problemstillingen ved hjelp av Euler metoden.
(Obs! Hvis du har tenkt å kopiere koden herfra, pass på når du kopierer at det ikke er noen innrykk, med unntak av forløkka. Da må innrykk ikke være for stor. For å være sikker hent innholdet i forløkka opp til kolon og trykk linjeskift ved kolon for å få det ned igjen i forløkka.)
- d) Vis grafisk den numeriske løsningen og den eksakte løsningen i samme grafvindu.
- e) Kommenter om nøyaktigheten av den numeriske løsningen.

Oppgave 16: Gitt følgende problemstilling

$$f'(x) = x^5 - x^2, \text{ for } x \in [-1, 4],$$
$$f(0) = -2.$$

- a) Bruk Euler metoden og skriv et uttrykk for $f(x_{i+1})$.
- b) Vis at følgende er en eksakt løsning av ligningen

$$f(x) = \frac{1}{6}x^6 - \frac{1}{3}x^3 - 2.$$

- c) Lag en skript for å løse problemstillingen ved hjelp av Euler metoden.
- d) Vis grafisk den numeriske løsningen og den eksakte løsningen i samme grafvindu.
- e) Kommenter om nøyaktigheten av den numeriske løsningen.

SIR modellen – En modell for utvikling av en epidemi

Hittil har vi innført mange nye matematiske temaer og vi har lært litt programmering underveis. Dette har vært nødvendig fordi vårt mål siden starten har vært å kunne modellere spredning av sykdommer. For å nå dette målet har vi vært innom temaer som *funksjoner*, *derivasjon*, *Euler metoden* og til slutt *programmering* for å kunne løse såkalte *differensiallikninger*. Nå er vi endelig klare til å snakke om modellen vi skal bruke for å forstå bedre hvordan et virus kan spre seg i en gitt populasjon. Faktisk kan modellen brukes i andre sammenhenger også, f.eks. for å studere smitte mellom oppdrettsfisk i en merd; der disse fiskene lever i kunstig høye tettheter, hvor dette da gir grunnlag for høy smitterate.



Bildet er hentet fra SNL (<https://snl.no/fiskeoppdrett>)

Problemstillingen er som følger. For $t \in [0, T]$

$$\mathbf{S}'(t) = -\frac{a}{N}\mathbf{S}(t) \cdot \mathbf{I}(t)$$

$$\mathbf{I}'(t) = \frac{a}{N}\mathbf{S}(t) \cdot \mathbf{I}(t) - b\mathbf{I}(t)$$

$$\mathbf{R}'(t) = b\mathbf{I}(t)$$

S står for **susceptible** på engelsk, og innebærer da at det er de som kan bli smittet eller med andre ord friske. **I** står for **infected** på engelsk og innebærer da at det er de som er smittet og kan spre sykdommen videre. **R** står for **removed** på engelsk og innebærer da at det er de som var smittet, men som nå har blitt fjernet fra blant de smittede, ved at de enten har blitt friske og immune, eller har død. Vi kan nå også se hvorfor modellen kalles for SIR, nemlig det er et akronym (susceptible, infected, removed).

Videre bruker vi **t** for **tiden**, **a** er **smitteraten** og er en prosent for hvor sannsynlig det er at noen blir smittet. **N** er en **fast tallstørrelse på populasjonen**: $N = S(t) + I(t) + R(t)$, og **b** er **raten for å bli fjernet fra blant de smittede eller dødsraten**.

Før vi fortsetter videre er det lurt å ikke la seg selv bli overveldet av disse likningene. Vi har tidligere kun løst en ligning om gangen. Nå derimot skal vi løse tre likninger samtidig! Heldigvis viser det seg at det er ikke en så stor forskjell. Vi ender opp med bare noen ekstra linjer med kode!

Vi starter med å bruke Euler metoden på problemstillingen. Siden vi ikke bruker x nå, så er det ikke en veldig stor forandring. Vi bytter våre x 'er ut med t 'er. Første likningen blir da

$$\frac{S(t_{i+1}) - S(t_i)}{t_{i+1} - t_i} = -\frac{a}{N}S(t_i) \cdot I(t_i)$$

Løser vi dette for den ukjente $S(t_{i+1})$ får vi da at

$$S(t_{i+1}) - S(t_i) = -\frac{a}{N}S(t_i) \cdot I(t_i) \cdot (t_{i+1} - t_i)$$

$$S(t_{i+1}) = S(t_i) - \frac{a}{N}S(t_i) \cdot I(t_i) \cdot (t_{i+1} - t_i)$$

La nå $h = (t_{i+1} - t_i)$, da får vi at

$$S(t_{i+1}) = S(t_i) - h \frac{a}{N} S(t_i) \cdot I(t_i)$$

Vi kan til slutt rydde uttrykket ved å samle fellesfaktor $S(t_i)$ for de leddene der den opptrer flere ganger:

$$S(t_{i+1}) = S(t_i) \cdot \left(1 - \frac{a \cdot h}{N} I(t_i)\right)$$

Vi kan gange inn $S(t_i)$ i parentesen og da vil vi ende opp uttrykket vi hadde før vi ryddet sammen uttrykket. Til slutt kan vi også bruke **kort notasjon** til å forminske uttrykket:

$$S_{i+1} = S_i \cdot \left(1 - \frac{a \cdot h}{N} I_i\right)$$

Oppgave 17:

- Bruk Euler metoden på likningen for $I'(t)$ og lag uttrykk for I_{i+1} .
- Bruk Euler metoden på likningen for $R'(t)$ og lag uttrykk for R_{i+1} .

Før du leser videre, er det lurt at du forsøker selv å finne uttrykkene for I_{i+1} og R_{i+1} . Vi har nesten nådd målet vårt. Vi skal nå iverksette disse likningene i Python!

SIR modell – implementasjon i Python

Vi skal nå lage skripten til å kunne simulere disse likningene. Dette kalles å “**implementere**” eller iverksette. For å løse disse likningene, trenger vi noen start verdier. La oss anta at smitteraten, a , er 40% eller 0.4 som tallverdi. Det vil si vi antar at det er en veldig høy smitterate. Dette kan for eksempel skyldes at sykdommen har ikke blitt oppdaget ennå og den blir spredt uten at individer i en populasjon er klar over at de er syke. Andre faktorer kan også være at styresmaktene har ikke innført strenge tiltak som f.eks. karantene, isolasjon, spredning av info om håndvask og sosial distansering. Dødsraten eller sjansen til å danne immunitet, b , er 3% eller 0.03 som tallverdi. Vi skal kjøre simuleringen for $t \in [0, 100]$, der t er antall dager. For å begrense simuleringen, antar vi populasjonen er på 1000, f.eks. en liten by. I denne byen er det 997 friske individer, 3 smittede individer og ingen døde individer eller individer som har dannet immunitet ennå. Slik ser da vår problemstilling ut:

For $t \in [0, 100]$, $S_0 = 997$, $I_0 = 3$, $R_0 = 0$, $a = 0.4$, $b = 0.03$, og $N = 1000$, der

$$S_{i+1} = S_i \cdot \left(1 - \frac{a \cdot h}{N} \cdot I_i\right)$$

$$I_{i+1} = I_i \cdot \left(1 - b \cdot h + \frac{a \cdot h}{N} \cdot S_i\right)$$

$$R_{i+1} = R_i + b \cdot h \cdot I_i$$

Og her følger koden:

```
from matplotlib.pyplot import plot, show, legend
from numpy import zeros, linspace

a = 0.4
b = 0.03

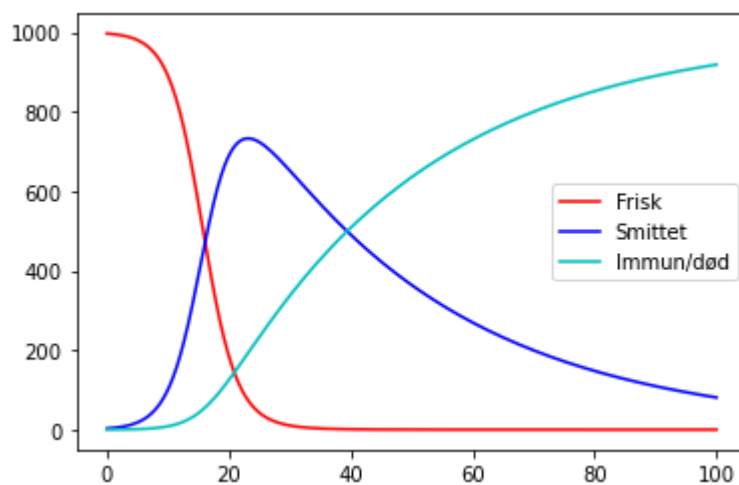
n = 1000
S = zeros(n)
I = zeros(n)
R = zeros(n)

S[0] = 997
I[0] = 3
R[0] = 0
N = S[0] + I[0] + R[0]

T = 100
t = linspace(0, T, n)
h = t[1] - t[0]

for i in range(0, n - 1):
    S[i+1] = S[i]*(1 - (a*h/N)*I[i])
    I[i+1] = I[i]*(1 - b*h + (a*h/N)*S[i])
    R[i+1] = R[i] + b*h*I[i]

plot(t, S, 'r', t, I, 'b', t, R, 'c')
legend(['Frisk', 'Smittet', 'Immun/død'])
show()
```



Kjørt med parametre:

$$a = 0.4, b = 0.03, S_0 = 997, I_0 = 3, R_0 = 0, T = 100$$

Først trenger vi å kommentere om følgende kode

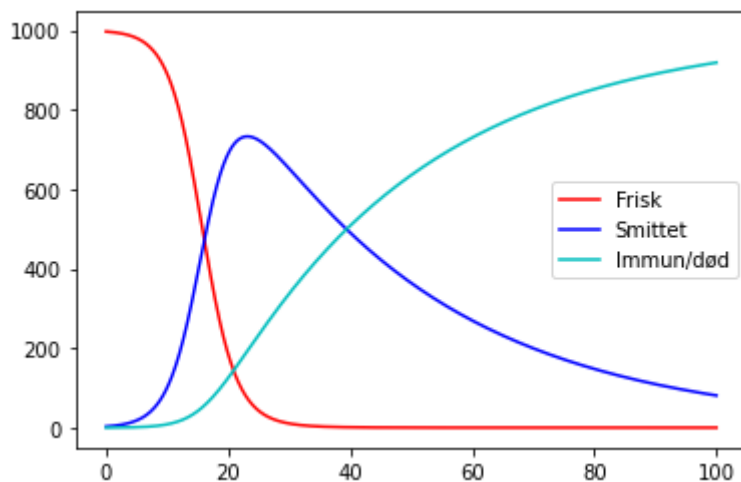
```
plot(t, S, 'r', t, I, 'b', t, R, 'c')
```

Ved hjelp av denne linja, klarer Python å fremstille alle tre grafer i samme vindu. Alle grafene vi har plottet har en felles argument, nemlig t. Det vil si vi plotter alle grafene over tidsintervallet fra [0, 100]. Vi har inkludert tre sett med argumenter til plot funksjonen, der hver sett med argumenter tilhører hver graf, slik vist over. Grafene blir laget etter følgende argumenter:

- Første grafen blir bestemt av argumentene t, S, 'r', der S er friske over tid og 'r' står for «red» eller rød farge.
- Andre grafen blir bestemt av argumentene t, I, 'b', der I er smittede over tid og 'b' står for «blue» eller blå farge.
- Tredje grafen blir bestemt av argumentene t, I, 'c', der R er døde/immune over tid og 'c' står for «cyan» eller cyan farge.

Analyse av grafene

La oss ta en nærmere titt på grafen og se hva vi kan trekke ut av informasjon



I denne kjøringen av vår skript har vi valgt å sette smitterate til 40% eller $a = 0.4$. Vi ser at da øker antall smittede innen de første 20 dagene drastisk. Dette er en situasjon som kan skape store utfordringer for styresmakter å håndtere. Hvis en slik situasjon skulle inntreffe et land, ville helsevesenet i det landet hatt store utfordringer med å håndtere så mange smittede samtidig, og da spesielt smittede som kan være i kritisk tilstand. Vi vil helst unngå at slikt inntreffer og vi må se hvordan vi kan klare å få grafer som kan skape en mer optimistisk situasjon. Vi vil helst ha en situasjon der sykehus ikke sprenger i kapasitet og sliter med å kunne ta imot pasienter, eller at de får en bølge av pasienter men de er ikke utrustet til å ta imot så mange samtidig.

Oppgave 18:

Bruk koden fra skriptet og prøv å juster parametere. Kommenter grafene som blir produsert. Husk å tilbakestille parametere etter hver deloppgave. Koden som du starter med som utgangspunkt blir kalt opprinnelig kode.

- a) Sett $S_0 = 950$ og $R_0 = 50$. Sammenlign grafene fra den opprinnelige koden. Hva betyr det når vi setter $R_0 = 50$ som start verdi? Legg spesielt merke til hvordan grafen over smittede skiller seg fra den opprinnelige ved $t = 20$.
- b) Start på nytt. Sett $b = 0.1$. Forklar hva b er. Sammenlign grafene og kommenter hvordan grafene skiller seg den opprinnelige.
- c) Start på nytt. Sett $a = 0.3$. Forklar hva a er. Sammenlign grafene og kommenter hvordan grafene skiller seg den opprinnelige. Gjenta for $a = 0.2$ og $a = 0.1$. Sammenlign og kommenter.
- d) **(Valgfritt)** Les følgende artikkel på wikipedia:
https://en.wikipedia.org/wiki/Flatten_the_curve
Artikkelen kan oversettes til norsk gjennom nettsiden google translate. Lim inn nettsiden og sett språk fra engelsk til norsk.
- e) Lag en kort rapport for myndigheter med dine anbefalinger, basert på dine funn fra disse simuleringer. Bruk all kunnskap du har tilegnet her for å rettferdiggjøre dine funn. Husk å være saklig.