Pythonmatte

Programmering i matematikk programfag på vgs

Torodd F. Ottestad

Innhald

Om boka

Denne boka inneheld ulike måtar ein kan nytta programmering på i matematikk (programfag på vgs).

Eg nyttar Python som programmeringsspråk gjennom heile boka .

OBS

Boka er under utvikling og vert oppdatert med ujamne mellomrom. Sist oppdatert: {{today}}

For dei spesielt interesserte er boka laga med Quarto. For å lære meir om Quarto-bøker kan ein kikka her.

Om meg

Her kan du lesa meir om meg

Logo: Programmer icons created by juicy_fish - Flaticon

Part I

Sannsyn og simulering (S1/S2)

I det følgjande kapittelet skal me sjå på korleis me kan simulera ulike stokastiske forsøk i Python. Me ser på alt frå enkle simuleringar i uniforme modellar, samansette forsøk, og vidare binomiske, hypergeometriske og normalfordelte forsøk. Me ser òg på hypotesetesting i Python.

1 Terningar og intro til simulering

Ein fin stad å starta med simulering er med terningar Her er sannsynet *uniformt* (det er like sannsynleg å få 2 som 5), og dei ulike utfalla er heiltal.

Det første som må gjerast er å gjera i stand "trekkaren" vår. Eg bruker her ein tilfeldighets-generator frå `NumPy` (dokumentasjon her).

```
import numpy as np
rng = np.random.default_rng()
```

Når me no har klargjort generatoren kan me bruka den innebygde `integers`-funksjonen for å trilla ein terning.

> **!** Merk
>
> Dei to linjene med kode over **må** vera med i programmet for at det skal funka. I mange av døma i boka er ikkje desse to linjene med i alle kodesnuttane.

1.1 Ein terning

```
terning = rng.integers(1, 7)
print(terning)
```

```
6
```

> **!** Merk
>
> Her er verdien `terning` eit heiltal (*integer*) **større eller lik** 1 og **mindre enn** 7. Sidan det er heiltal me trekk er dermed
>
> $$\text{terning} \in \{1, 2, 3, 4, 5, 6\}$$

For å trilla fleire terningar kan me anten bruka løkker:

```python
for i in range(10):
    print(rng.integers(1, 7))
```

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

eller så kan me legga inn eit argument `size` i `integers`. Då blir output ein array (ein form for liste) med `size` terningar:

```python
terningar = rng.integers(1, 7, size=10)
print(terningar)
```

```
[2 3 4 6 1 1 1 1 6 6]
```

No har me det me treng for å kunna simulera eit stokastisk forsøk og estimera sannsyn ut frå simuleringa. Til dømes kan me prøva å finna ut av kor sannunleg det er å trilla 5 eller 6 på ein terning:

```python
N = 1000000 # tal simuleringar

terningar = rng.integers(1, 7, size=N)

gunstige = sum(terningar >= 5)

sannsyn = gunstige / N

print(f"Sannsynet for 5 eller 6 er {sannsyn:.4f}")
```

```
Sannsynet for 5 eller 6 er 0.3336
```

Forklaring: `gunstige = sum(terningar >= 5)`

For å forstå denne ser me på eit døme:

```
array = np.array([1, 2, 3, 4, 5, 6])

større_enn_3 = array > 3

print(array)
print(større_enn_3)
print(sum(større_enn_3))
```

```
[1 2 3 4 5 6]
[False False False  True  True  True]
3
```

Altså gjer me om verdiar til `True` eller `False`. Python reknar `True` som 1 og `False` som 0. Når me då summerer alle elementa i `array` får me antall `True` i arrayen.

1.2 Fleire terningar

Spørsmål som "Kva er sannsynet for at produktet av to terningar er 8 eller mindre" er fint å finna svar på ved hjelp av simulering:

```
N = 1000000

terning1 = rng.integers(1, 7, size=N)
terning2 = rng.integers(1, 7, size=N)

produkt = terning1 * terning2
gunstige = sum(produkt <= 8)
sannsyn = gunstige / N

print(f"Sannsynet er {sannsyn:.4f}")
```

```
Sannsynet er 0.4452
```

Forklaring: `produkt = terning1 * terning2`

Kodelinja finn produktet av element på samme plass i dei to arrayane. Sjekk dømet:

```
a = np.array([1, 2, 3, 4, 5])
b = np.array([6, 7, 8, 9, 10])

c = a * b

print(c)
```

```
[ 6 14 24 36 50]
```

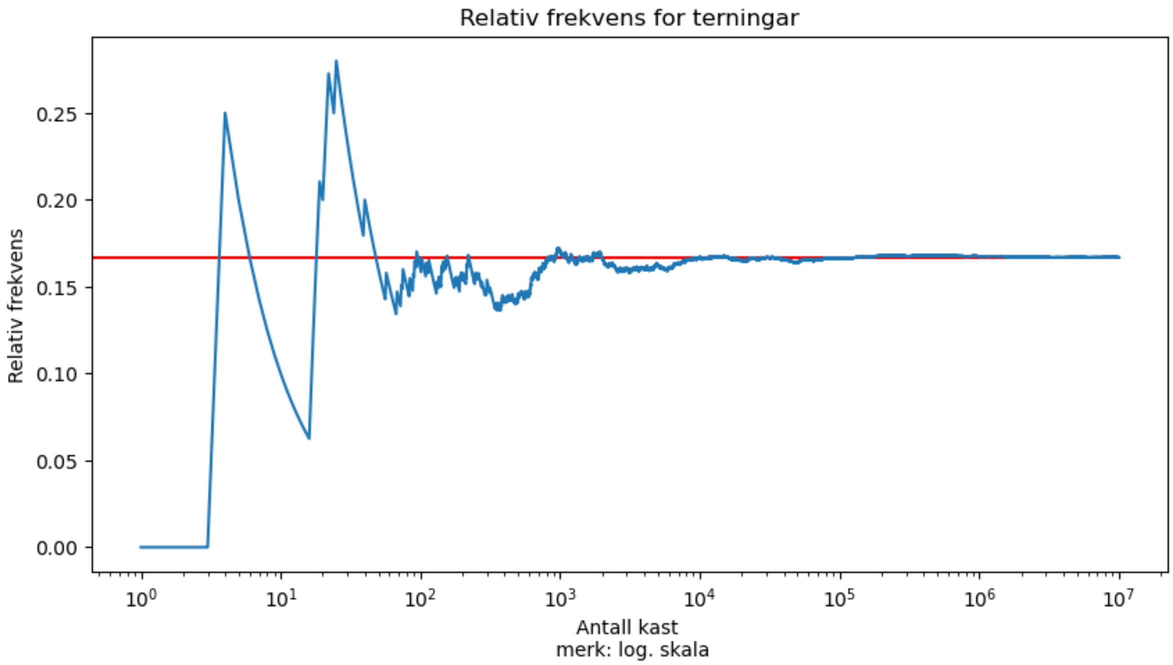
$1 \cdot 6 = 6$ og $2 \cdot 7 = 14$...

1.3 Nøyaktighet

Sjekkar kva som skjer når me triller fleire og fleire terningar (eller ein terning fleire gongar). For å visa samanhengen plottar me resultatet. I dømet ser me på sannsynet for å trilla 4 på ein terning.

```
import matplotlib.pyplot as plt

# antall kast
N = 10000000

# triller terningar
terningar = rng.integers(1, 7, size=N)

# finn den kumulative summen av terningar som er lik 4
kumulativ_sum = np.cumsum(terningar == 4)

# lager "x-akse" frå 1 til N
x = np.arange(1, N + 1)

# finn relativ frekvens
rel_frekvens = kumulativ_sum / x

plt.figure(figsize=(10, 5))                       # lagar ein figur med 10x5 mål
plt.hlines(1/6, 0, N, color="red")                # teiknar ein linje med farge "red" for d
```

```python
plt.plot(x, rel_frekvens)                          # plottar x-akse og y-akse
plt.xscale("log")                                  # logaritmisk x-akse
plt.xlabel("Antall kast \n merk: log. skala")      # namn på x-aksen
plt.ylabel("Relativ frekvens")                     # namn på y-aksen
plt.title("Relativ frekvens for terningar")        # tittel på figur
plt.show()
```



Her ser me at di fleire kast me gjennomfører, di nærare kjem den relative frekvensen den teoretiske verdien for å trilla ein firar på vanleg terning.

$$P(\text{firar}) = \frac{1}{6} \approx 0.167$$

2 Samansette forsøk - choice

No skal me sjå på korleis me kan simulera eit enkelt samansett forsøk. Oppgåva me skal sjå på er denne:

> I ei skål ligg det 7 banan-twist og 3 daim-twist. Kva er sannsynet for at me får banan når me trekk ut to bitar frå skåla. (Både med og utan tilbakelegging)

2.1 Teoretisk sannsyn

Først kan me sjå på kva det teoretiske sannsynet er for desse to. Ofte når me bruker simulering er det fordi det er vanskeleg å finna svaret ved rekning, men i dette dømet er det ikkje så vanskeleg.

Med tilbakelegging

$$P(\mathrm{BB}) = \frac{7}{10} \cdot \frac{7}{10} = \frac{49}{100} = 0.490$$

Utan tilbekelegging

$$P(\mathrm{BB}) = \frac{7}{10} \cdot \frac{6}{9} = \frac{42}{90} \approx 0.467$$

2.2 Simulering av twist-trekket

```python
# importerar og lagar ein random generator
from numpy.random import default_rng
rng = default_rng()

# antall simuleringar
N = 1000000

# lagar liste med twistskåla
twistskål = ["Banan"]*7 + ["Daim"]*3

```

```
# skriv ut twistskåla
print(twistskål)
```

```
['Banan', 'Banan', 'Banan', 'Banan', 'Banan', 'Banan', 'Banan', 'Daim', 'Daim', 'Daim']
```

2.2.1 Med tilbakelegging

```
BB = 0

for i in range(N):
    twist = rng.choice(twistskål, size = 2)
    if twist[0] == "Banan" and twist[1] == "Banan":
        BB += 1

rel_frek = BB/N

print(f"Sannsynet for at me trekk to banantwist er {rel_frek}")
```

```
Sannsynet for at me trekk to banantwist er 0.489409
```

Tips

For ein litt meir elegant kode *kan* me droppa if-setningen i løkka vår. Dette kan me gjera ved å gjera ein boolsk variabel (`True` eller `False`) om til eit heiltal. Då blir `True = 1` og `False = 0`

```
BB = 0

for i in range(N):
    twist = rng.choice(twistskål, size = 2)
    BB += int(twist[0] == "Banan" and twist[1] == "Banan")

rel_frek = BB/N

print(f"Sannsynet for at me trekk to banantwist er {rel_frek}")
```

```
Sannsynet for at me trekk to banantwist er 0.489524
```

Me kan sjå kor langt unna den teoretiske verdien me kjem:

```python
feil = abs(rel_frek - 49/100)
print(f"Feilen blir {round(feil, 6)} når me gjer {N} simuleringar")
```

```
Feilen blir 0.000476 når me gjer 1000000 simuleringar
```

Dersom me vil ha eit enno meir nøyaktig resultat kan me gjera fleire simuleringar, dette kjem me litt attende til seinare. Merk at programmet vil fort ta ganske lang tid å køyra etter kvart som talet på simuleringar aukar.

2.2.2 Utan tilbakelegging

Forskjellen blir ikkje stor her. Det einaste me gjer er å legga til `replace = False` som argument i `choice`-funksjonen

```python
BB = 0

for i in range(N):
    twist = rng.choice(twistskål, size = 2, replace = False)
    if twist[0] == "Banan" and twist[1] == "Banan":
        BB += 1

rel_frek = BB/N

print(f"Sannsynet for at me trekk to banantwist er {rel_frek}")
```

```
Sannsynet for at me trekk to banantwist er 0.466383
```

```python
feil = abs(rel_frek - 42/90)
print(f"Feilen blir {round(feil, 6)} når me gjer {N} simuleringar")
```

```
Feilen blir 0.000284 når me gjer 1000000 simuleringar
```

2.3 Ikkje uniforme sannsynsmodellar

Dette dømet me har sett på er eit døme på ein ikkje-uniform sannsynsmodell, sidan sannsynet for Banan og Daim ikkje er det same. I starten laga me ei liste med alle twistane i skåla. I dette dømet er det praktisk, sidan me har eit lite utfallsrom (banan og daim) og kontroll på kor mange det er av kvar.

Av og til kan det vera nyttig å definera ikkje-uniforme sannsynsmodellar på ein litt anna måte.

```python
twistar = ["Banan", "Daim"]
sannsyn = [7/10, 3/10]

to_twist = rng.choice(twistar, size = 2, p = sannsyn)

print(f"Me trekk {to_twist}")
```

```
Me trekk ['Banan' 'Daim']
```

Dette kan brukast viss me veit utfallsrommet og sannsynet for kvart av utfalla, ikkje nødvendigvis antallet. F.eks. blodtype hos tilfeldige personar i befolkningen.

3 Simulering av ulike fordelingar

3.1 Binomisk sannsyn

3.2 Hypergeometrisk sannsyn

3.3 Normalfordelt sannsyn

4 Hypotesetesting

```
print("Kjem etterkvart")
```

```
Kjem etterkvart
```

Part II

Følgjer og rekker (S2/R2)

I dette kapittelet skal me sjå på korleis me kan bruka Python til å arbeida med følgher og rekker.

Ei talfølgje er ein serie tal. Dei kjem på ulike formar. Eit døme på ei talfølgje er:

```python
for i in range(1, 6):
    print(i, end=",")

print("...")
```

```
1,2,3,4,5,...
```

Dette kjenner me att som dei 5 første naturlege tala, $\mathbb{N}$, og er eit enkelt døme på ei talfølgje. Kvart av tala i følgja kallar me for *ledd*. Det første leddet vert kalla a_1 medan det n-te leddet vert kalla a_n.

Vidare i kapittelet skal me sjå på ulike typer talfølgjer, eksplisitte og rekursive funksjonar for å finna ledd i talfølgjer. Vidare ser me på ulike typer rekker. Før me mot slutten ser på nokre døme på praktisk bruk av følgjer og rekker (lån og sparing).

5 Følgjer

På førre side såg me eit kjapt døme på ei talfølgje,

$$1, 2, 3, 4, 5, \ldots$$

Av denne talfølgja ser me to ting. For det første er ho uendeleg, sidan det ikkje er definert nokon ende, berre "…". Me ser og at det er ein fast differanse mellom kvart av ledda. Talfølgjer med fast differanse mellom ledda kallar me *aritmetiske* talfølgjer.

5.1 Aritmetiske følgjer

Eit anna døme på ei aritmetisk talfølgje er denne

$$3, 7, 11, 15, \ldots$$

Her ser me at det første leddet $a_1 = 3$ og at differansen $d = 4$.

At det er ein fast differanse mellom kvart av ledda betyr at dersom me skal finna eit ledd (a_n), må me ta leddet før (a_{n-1}) og legga til differansen (d). Dermed får me

$$a_n = a_{n-1} + d$$

Frå dømet over ser me at det stemmer, $7 = 3 + 4$ og $11 = 7 + 4$ osb.

5.1.1 Rekursiv formel for ledd n

Samanhengen over kan me nytta for å finna a_n rekursivt. Rekursjon handlar om gjentaking, så tanken er at me kan gjenta formelen for å finna ledd n. Me kan laga ein funksjon som kun tek utgangspunkt i opplysningen om at $a_n = a_{n-1} + d$.

```
def a(n):
    return a(n-1) + 4

```

```
print(a(4))
```

```
RecursionError: maximum recursion depth exceeded
```

Men om du prøver å køyra denne koden vil du få ein feil:

    RecursionError: maximum recursion depth exceeded

Om du ser på koden, ser du kanskje kva som er problemet?

Det let seg løysa om me definerer ein rekursjonsbotn (i dette tilfellet a_1). Me prøver på nytt:

```
def a(n):
    if n == 1:
        return 3
    else:
        return a(n-1) + 4

print(a(4))
```

```
15
```

Denne funksjonen kan me bruka for å t.d. skriva ut dei 10 første ledda i følgja:

```
for i in range(1, 11):
    print(a(i), end = ", ")

print("...")
```

```
3, 7, 11, 15, 19, 23, 27, 31, 35, 39, ...
```

Ulempen med denne rekursive funksjonen er at han må gjennom *alle* dei føregåande ledda for kvart ledd du bruker han for å finna. Så om du skal finna $a_1$000 vil formelen finna alle ledda før. Og på nytt når du ser etter $a_1$001... Det kan fort bli både tidkrevjande og tungvint, sjølv for datamaskina.

5.1.2 Eksplisitt formel for ledd n

Me kan sjå om me finn ein eksplisitt måte å finna a_n på (altså finna direkte, utan rekursjon). Me ser på dømet igjen.

$$3, 7, 11, 15, ...$$

Me ser at

$$a_2 = 7 = 3 + 4$$
$$a_3 = 11 = 7 + 4 = 3 + 4 + 4$$
$$a_4 = 15 = 11 + 4 = 7 + 4 + 4 = 3 + 4 + 4 + 4$$

Altså er

$$a_n = a_1 + (n - 1)d$$

som me kan programmera som

```
def a(n):
    return 3 + (n-1)*4

print(a(4))
```

```
15
```

Og igjen kan me skriva ut dei ti første ledda:f

```
for i in range(1, 11):
    print(a(i), end=", ")

print("...")
```

```
3, 7, 11, 15, 19, 23, 27, 31, 35, 39, ...
```

6 Rekker

```
a = "test"
```

Part III

Funksjonar (S1/S2/R1/R2)

- plotting
- kurvetilpassing / modellering
- derivasjon
- integrasjon

7 Plotting

7.1 Funksjonar med delt forskrift

Nokre funksjonar kan ha ulik definisjon på ulike intervall. Desse kallar me funksjonar med delt forskrift. Me ser på funksjonen

$$f(x) = \begin{cases} x^2 & \text{for } x \leq 0 \\ x & \text{for } x > 0 \end{cases}$$

Denne kan definerast med `numpy.piecewise()`.

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

Startar med å definera funksjonen. Det gjer me med

```python
numpy.piecewise(x, condlist, funclist)
```

der `condlist` er ei liste med betingelse/intervall og `funclist` funksjonane i same rekkefølge

```python
def f(x):
    return np.piecewise(x, [x <= 0, x > 0], [lambda x: x**2, lambda x: x])
```

Kva er `lambda`?

`lambda x:` er kortversjonen av

```python
def g(x):
    return x**2
```

Dermed kan me heller definera funksjonen slik:

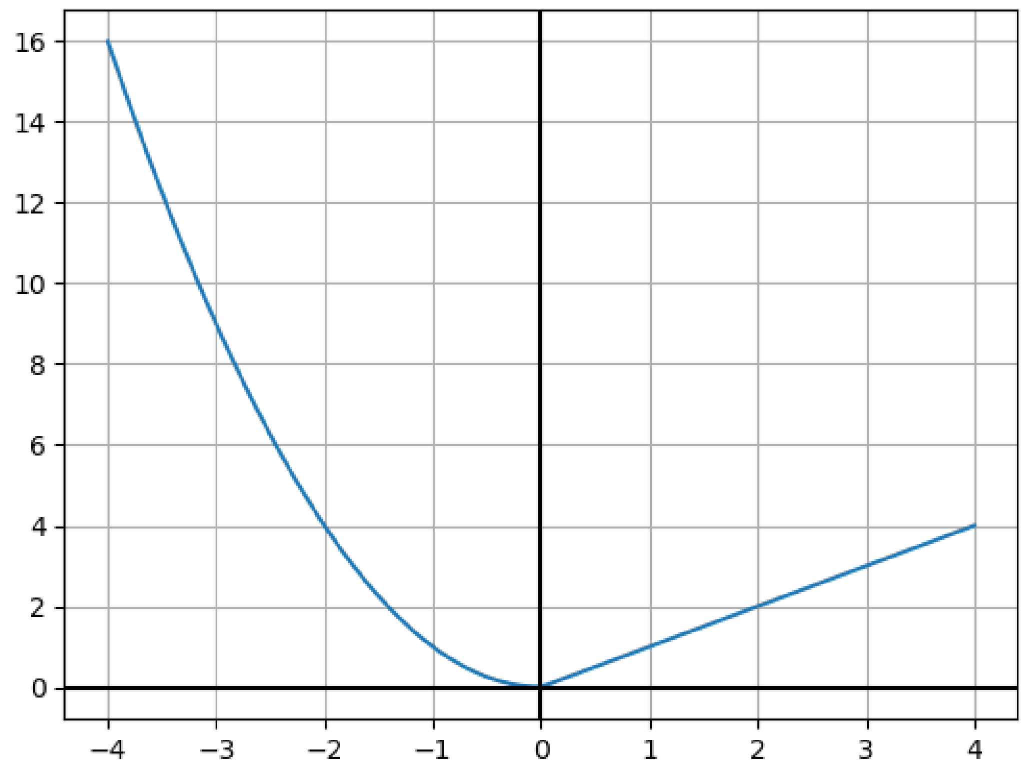
```python
x = np.linspace(-3,2,5)
g = lambda x: x**2

y = g(x)
```

Lagar ein array med 100 x-verdiar og finn vidare y-verdiane med funksjonen me definerte.

```python
x = np.linspace(-4,4,100)
y = f(x)
```

Plottar grafen:

```python
plt.plot(x, y)

# pynt
plt.axhline(0, color="black")
plt.axvline(0, color="black")
plt.grid()
plt.show()
```

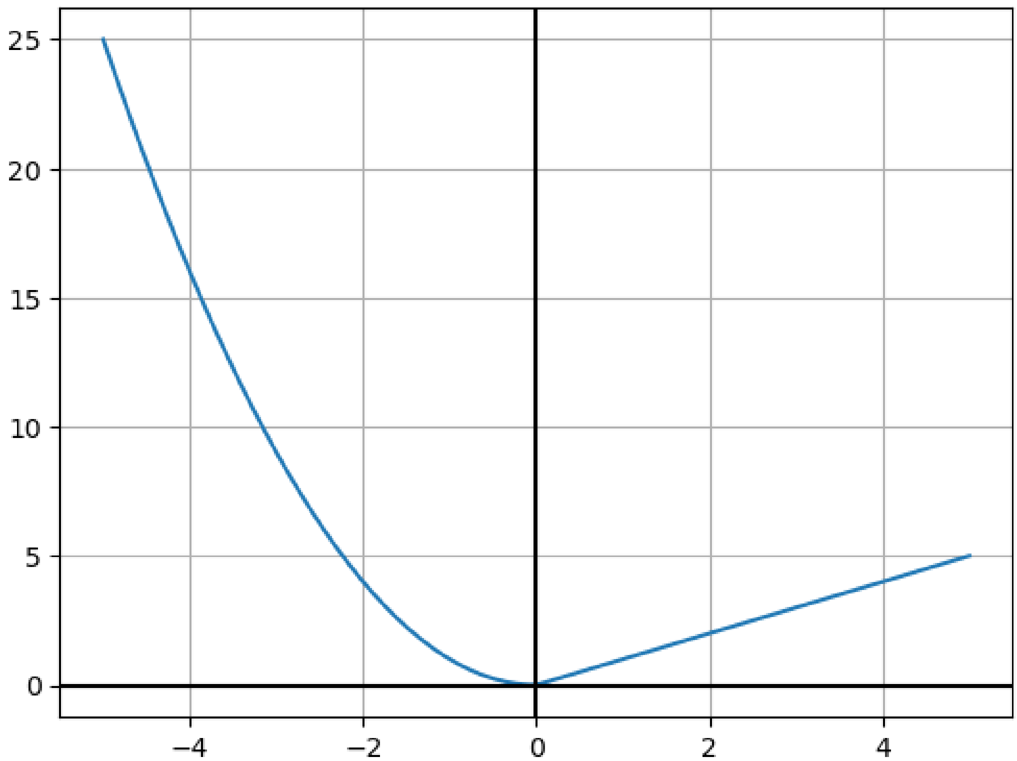


7.1.1 Alternativ: `if/else`

Ein annan måte dette kan gjerast på er å bruka betingelsar og løkker. Dette kan me gjera slik:

```
# definerer funksjonen
def f(x):
    if x <= 0:
        return x**2
    else:
        return x

# lager x-verdiar
x_verdiar = np.linspace(-5, 5, 100)

# rekner ut y-verdiane
y_verdiar = [f(x) for x in x_verdiar]

```

```
# plottar
plt.plot(x_verdiar, y_verdiar)

# pynt
plt.axhline(0, color="black")
plt.axvline(0, color="black")
plt.grid()
plt.show()
```



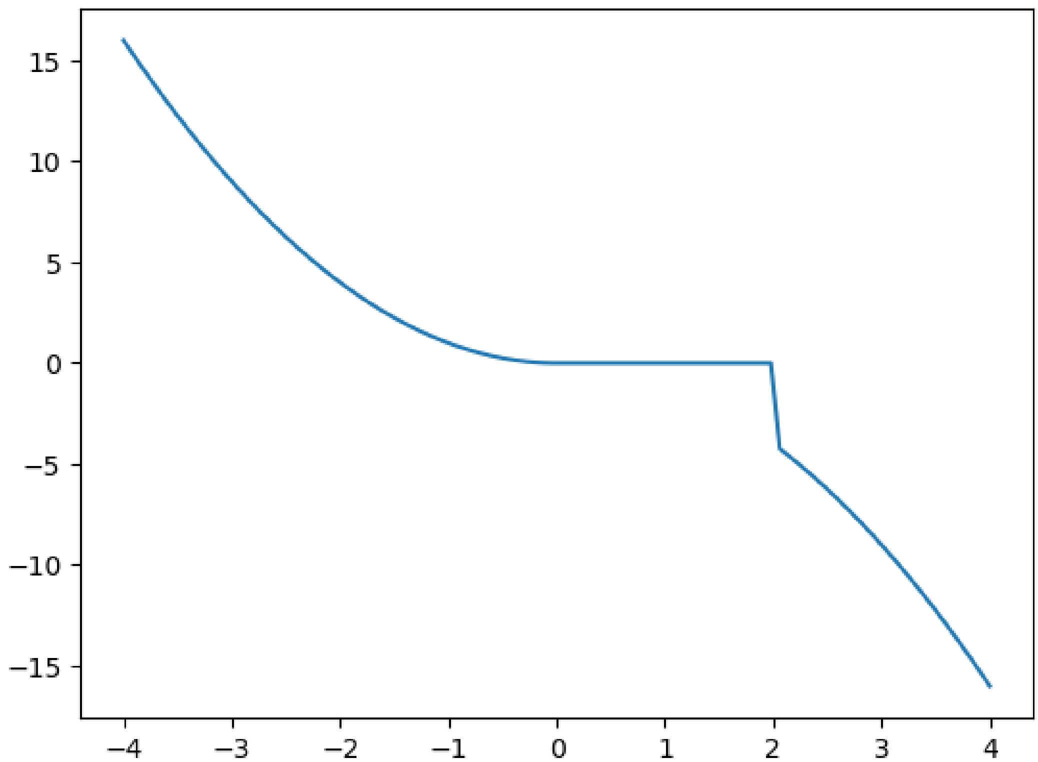
7.1.2 Diskontinuerlege funksjonar

Framgangsmåten over med `piecewise` kan brukast for funksjonar som er definert for alle x-verdiar mellom nederste og øverste del av definisjonsmengda. Om ein har ein funksjon med delt definisjonsmengde som td.

$$h(x) = \begin{cases} x^2 & \text{for } x \leq 0 \\ -x^2 & \text{for } x \geq 2 \end{cases}$$

må ein gjera tilpassingar. Prøver først måten me gjorde det over:

```
# definerer funksjonen
def h(x):
    return np.piecewise(x, [x <= 0, x >= 2], [lambda x: x**2, lambda x: -x**2])


# finn x og y
x = np.linspace(-4, 4, 100)
y = h(x)

# plottar
plt.plot(x, y)
plt.show()
```

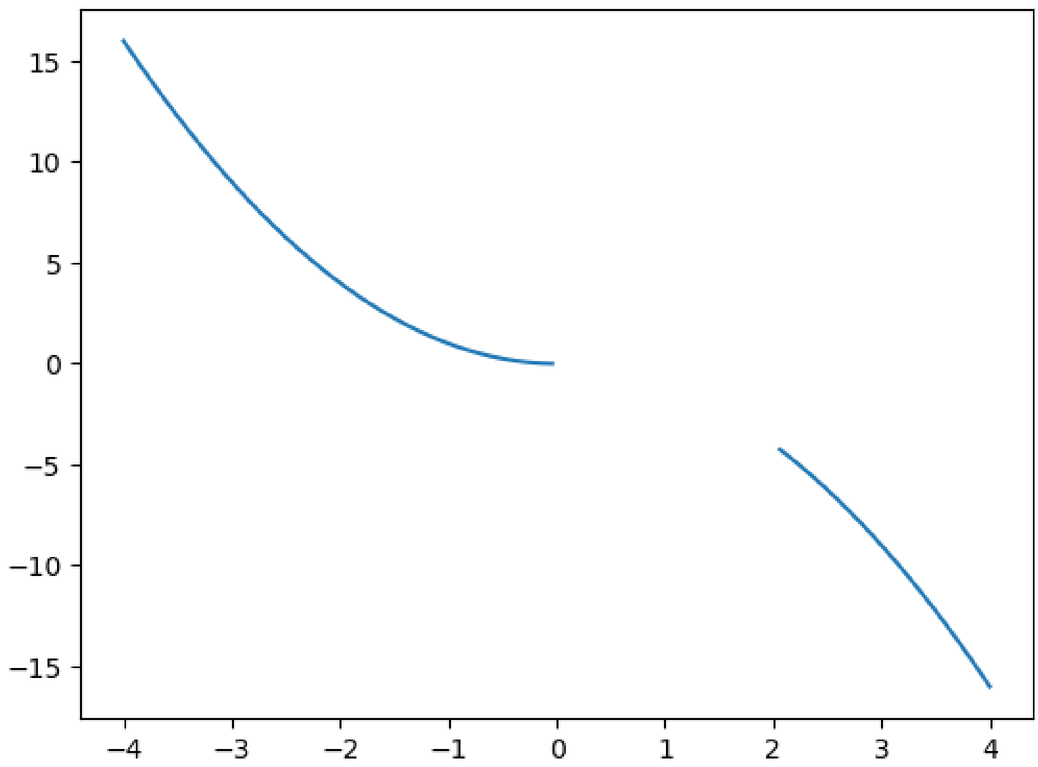


Her viser utfordringa med denne typen funksjonar. I staden for å teikna to kurver som ikkje heng saman, vert funksjonsverdien 0 når $x \in \langle 0, 2 \rangle$. I `piecewise` sin dokumentasjon finn me dette:

The output is the same shape and type as x and is found by calling the functions in funclist on the appropriate portions of x, as defined by the boolean arrays in condlist. **Portions not covered by any condition have a default value of 0**.

Måten å løysa det på er å definera kva som skal skje i intervallet der funksjonen ikkje er definert:

```
# definerer funksjonen (også mellom 0 og 2)
def h(x):
    return np.piecewise(x, [x <= 0, (x > 0) & (x < 2), x >= 2], [lambda x: x**2, np.nan, l


# finn x og y
x = np.linspace(-4, 4, 100)
y = h(x)

# plottar
plt.plot(x, y)
plt.show()
```



Her er intervallet `(x > 0) & (x < 2)` definert ved funksjonen `np.nan` (*not a number*). På denne måten unngår me at funksjonsverdien vert sett til 0 automatisk i mellomrommet mellom dei to intervalla som utgjer definisjonsmengda.

8 Derivasjon

```
print("...")
```

...

9 Integrasjon

9.1 Approksimering av integral

Det er fleire måtar me kan approksimera bestemte integral numerisk. I GeoGebra finn me funksjonane `SumUnder` og `SumOver` som gjev oss summen av arealet til n rektangel mellom a og b på x-aksen. Rektangla er litt for store eller litt for små, som følgje av at den øverste sida ligg under eller over funksjonen. `SumUnder` vil dermed gje eit resultat som er litt mindre enn det faktiske resultatet, medan `SumOver` vil gje eit litt for stort resultat.

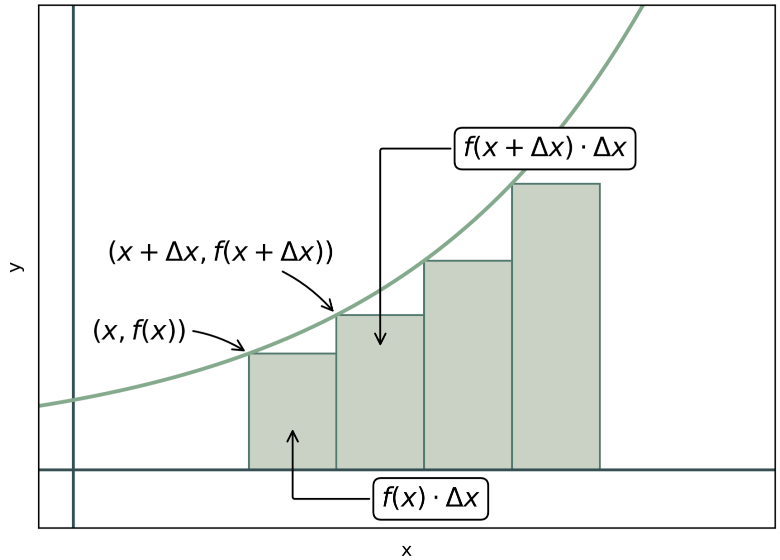
Animasjonen under viser korleis nøyaktigheita til `SumUnder` aukar etter kvart som antall rektangel vert større. Med fleire rektangel vert arealet som ikkje vert dekka mindre og mindre.

Den enklaste approksimasjonen er å finna venstre- eller høgresum. Med venstresum skal kvar av rektangla ha høgde slik at hjørnet øverst til venstre ligg på funksjonen. Høgresum finn me på same måte, men med hjørnet øverst til høgre på funksjonen. Algoritmen vert ganske lik for begge. Me ser på venstresum først.

9.1.1 Venstresum

Me skal finna summen av n rektangel mellom a og b som er slik at hjørnet øverst til venstre på kvart rektangel ligg på funksjonen. Breidda til rektangla kallar me dx (Δx på figuren).

```python
def venstresum(f, a, b, n):
    # finn bredden
    dx = (b-a)/n

    # startverdiar
    x = a
    sum_venstre = 0

    # finn arealet av kvart rektangel og legg arealet til totalen
    for i in range(n):
        rektangel = f(x)*dx
        sum_venstre += rektangel
        x += dx
```



Figur 9.1: Venstresum

```

    # returnerer totalverdien
    return sum_venstre
```

Tester algoritmen på funksjonen

$$f(x) = x^3 + 2x + 3$$

Prøver å finna ein omtrentleg verdi for arealet under $f(x)$ mellom $x = 2$ og $x = 5$. Prøver med $n = 100$.

```
def f(x):
    return x**3 + 2*x + 3


print(f"Venstresum: {venstresum(f, 2, 5, 100):.3f}")
```

```
Venstresum: 180.410
```

9.1.2 Høgresum

Me skal finna summen av n rektangel mellom a og b som er slik at hjørnet **øverst til høgre** på kvart rektangel ligg på funksjonen. Breidda til rektangla kallar me dx. Funksjonen er heilt lik som i `venstresum()` men me endrar `rektangel` til `f(x+dx)*dx` slik at me reknar høgda på høgresida av rektangelet. Sjå Figur 9.1

```
def høgresum(f, a, b, n):
    dx = (b-a)/n

    x = a
    sum_høgre = 0

    for i in range(n):
        rektangel = f(x+dx)*dx
        sum_høgre += rektangel
        x += dx

    return sum_høgre
```

Testar på samme funksjon og intervall som tidlegare:

```
print(f"Høgresum: {høgresum(f, 2, 5, 100):.3f}")
```

```
Høgresum: 184.100
```

Med det kan me anta at arealet ligg ein stad mellom 180,41 og 184,10. Med andre ord

$$180.41 \leq \int_2^5 f(x)\,dx \leq 184.10$$

Om me aukar talet på rektangel vil me få ein betre approksimasjon:

```
print(f"Venstresum: {venstresum(f, 2, 5, 1000):.3f}")
print(f"Høgresum: {høgresum(f, 2, 5, 1000):.3f}")
```

```
Venstresum: 182.066
Høgresum: 182.435
```

9.1.3 Sum under og sum over

Me kan laga algoritmar som fungerer på same måte som GeoGebra sine tidlegare nemnde `SumUnder` og `SumOver`. Me tek utgangspunkt i samme algoritme som tidlegare, men no må me sjekka kva for ei av sidene som er kortast (for `SumUnder`) eller lengst (for `SumOver`).

Sum under først:

```
def sumunder(f, a, b, n):
    dx = (b-a)/n

    x = a
    sum_under = 0

    # finn den kortaste sida og bruker den som høgde
    for i in range(n):
        if f(x) <= f(x+dx):
            rektangel = f(x)*dx
        else:
            rektangel = f(x+dx)*dx

        sum_under += rektangel
        x += dx

    return sum_under
```

Sum over blir heilt anaalogt, men me snur ulikskapen:

```
def sumover(f, a, b, n):
    dx = (b-a)/n

    x = a
    sum_over = 0

    for i in range(n):
        if f(x) >= f(x+dx):
            rektangel = f(x)*dx
        else:
            rektangel = f(x+dx)*dx

        sum_over += rektangel
        x += dx
```

```

    return sum_over
```

Tester på funksjonen og intervallet frå tidlegare:

```
print(f"Sum under: {sumunder(f, 2, 5, 100):.3f}")
print(f"Sum over: {sumover(f, 2, 5, 100):.3f}")
```

```
Sum under: 180.410
Sum over: 184.100
```

Kva for ein av desse som fungerer best vil avhenga av funksjonen. Tenk gjerne litt på kva type funksjonar dei ulike passar godt eller dårleg til.

9.1.4 Trapesmetoden

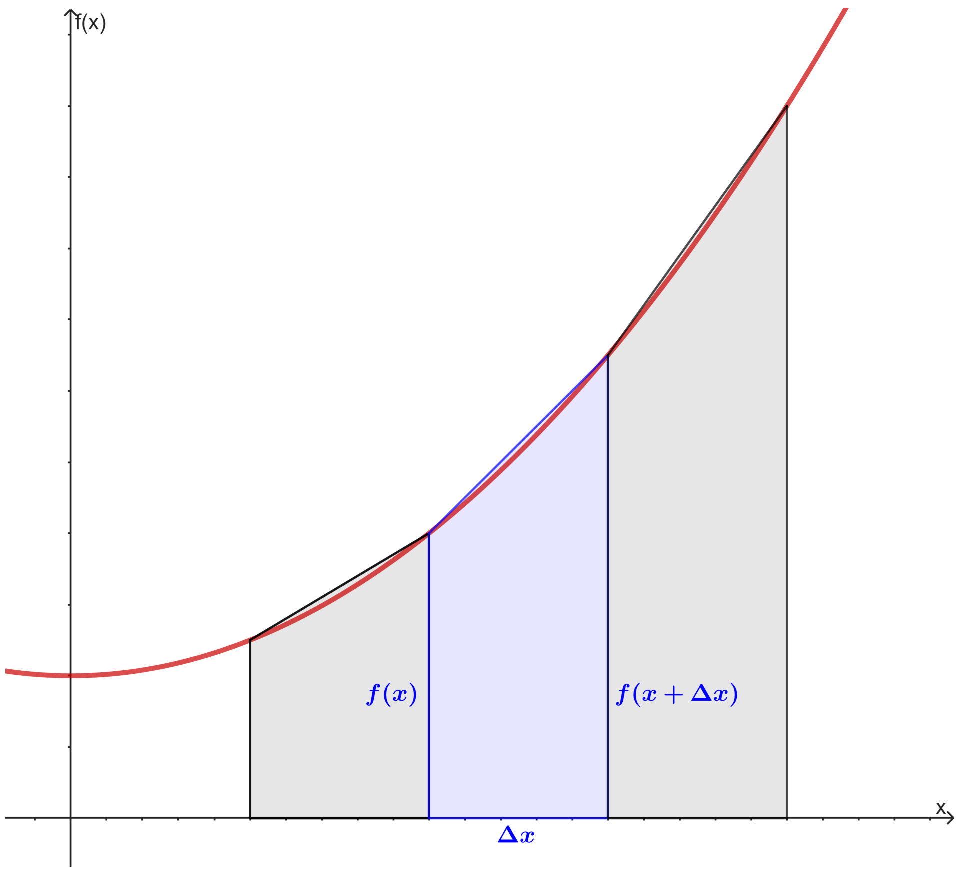
Ein veldig effektiv måte å approksimera arealet under grafen på er å laga trapes framfor rektangel. Høgda på trapeset vert `dx` medan dei to parallelle sidene vert `f(x)`og `f(x+dx)`.

```
def trapesmetoden(f, a, b, n):
    dx = (b-a)/n

    x = a
    sum_trapes = 0

    for i in range(n):
        trapes = ((f(x)+f(x+dx))*dx)/2
        sum_trapes += trapes
        x += dx

    return sum_trapes

# tester med n=100
print(f"Trapesmetode: {trapesmetoden(f, 2, 5, 100):.3f}")
```

```
Trapesmetode: 182.255
```

Me reknar ut det bestemte integralet

$$\int_2^5 x^3 + 2x + 3\,dx = \frac{729}{4} = 182.25$$



Figur 9.2: Trapesmetoden

Her ser me at me kjem nærare svaret med 100 trapes enn med 1000 rektangel, så trapesmetoden er mykje meir nøyaktig.

9.2 Symbolsk integrasjon med `SymPy`

(her kjem litt om korleis ein kan integrera symbolsk med `sympy` / CAS i python)

A Tips til programmeringa

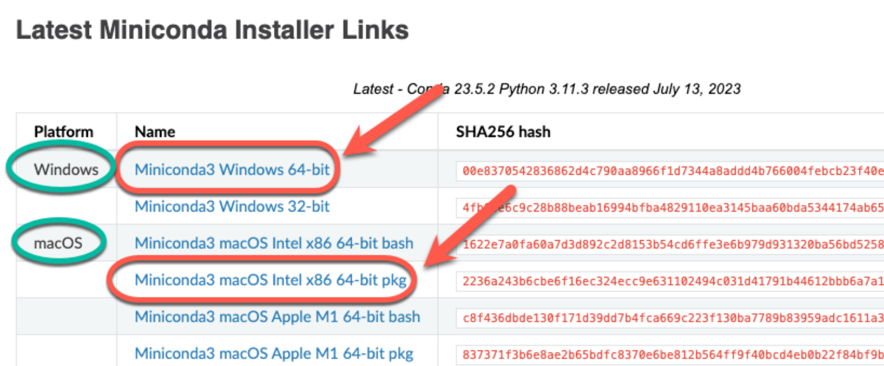
Det er mange måtar ein kan skrive og kompilere Pythonkode på. Beste tips for programmering i matematikk er å bruka Jupyter-notatbøker. Dette er filer (`.ipynb`) der ein kan kombinera små snuttar med Pythonkode og tekst (i `markdown`-format). DEnne boka er stort sett basert på jupyter-filer. Dette er eit døme på korleis ei slik fil ser ut.

A.1 Jupyter lab

Eit godt verktøy for å laga, endra og køyra jupyterfiler er Jupyter Lab. Dette programmet kan installerast på mange måtar. Mitt tips er å installera det gjennom *Miniconda*. Dokumentasjonen til programmet finn du her

A.2 Miniconda

1) Gå inn på https://docs.conda.io/en/latest/miniconda.html



2) Last ned den nyaste installasjonsfila
3) Installer miniconda ved å køyra fila.
4) Opne **terminal** på Mac og **Anaconda Prompt** på Windows. (På Mac kan du opna spotlight og søka etter programmet, på Windows kan du søka i start-menyen).
5) I terminal/Anaconda Prompt skriv du desse kodelinjene (linje for linje)

```
conda config --add channels conda-forge
conda config --set channel_priority strict
conda update -n base -c defaults conda
conda install pandas matplotlib jupyterlab ipympl xarray python=3.11
conda install scipy sympy
```

6) Når du skal bruke Jupyter lab opnar du terminal/Anaconda Prompt og skriv inn

```
jupyter lab
```