Radioaktivitet

Jon Bjarne Bø

Innholdsfortegnelse

Informasjon om ressursen

https://youtu.be/tN8MVvRfSWo

Info til elev

Dette dokumentet er en læringsressurs til deg i arbeid med utforskning av begrepet radioaktivitet i naturfag og/eller fysikk. Oppgavene du møter på her vil inneholde Python-kode som du selv må gjøre endringer på for å kunne svare på oppgavene og utforske ulike fenomener. Når du har gjennomført oppgavene er målet at du har brukt programmering som et utforskende verktøy innenfor dette temaet, og dermed fått en dypere forståelse av hva nuklider og radioaktiv stråling er, blitt kjent med de ulike typene radioaktiv stråling, fått et forhold til begrepet halvveringstid og i tillegg blitt litt kjent med ulike henfallskjeder.
Utgangspunktet for denne utforskingen er du kan noe grunnleggende programmering i Python fra før, men selv om du har veldig lite forkunnskaper vil dette opplegget kunne være en inngang til å bruke programmering i faget. Fokuset for oppgavene er å lære mer og å utforske radioaktivitet som tema, og ikke spesifikt på programmeringstekniske begreper.

Relevante kompetansemål i videregående skole

Naturfag
Mål for opplæringen er at eleven skal kunne:

- vurdere og lage programmer som modellerer naturfaglige fenomener
- utforske og beskrive elektromagnetisk og ioniserende stråling, og vurdere informasjon om stråling og helseeffekter av ulike strålingstyper

Fysikk 2
Mål for opplæringen er at eleven skal kunne:

- bruke numeriske metoder og programmering til å utforske og modellere fysiske fenomener
- utforske og analysere en selvvalgt teoretisk eller praktisk problemstilling i fysikk, og presentere viktige prinsipper, sammenhenger og konsekvenser

Installasjon av nødvendige biblioteker

Oppgavene under forutsetter at du har installert Python på din egen maskin. Om du ikke har det kan du få hjelp av lærer, eller evt. installere det selv. Den enkleste måten å installere og jobbe med Python på er å kaste ned og bruke Thonny.

Før vi kan komme i gang med oppgavene trenger vi et bibliotek kalt radioactivedecay installert. For å installere biblioteker i Python åpner vi terminalen på PCen vår. Enten kan du åpne Command Prompt på windows, eller terminalen om du har mac. Bruker du Spyder, VS Code eller en annen IDE som har en terminal integrert kan du også heller bruke den integrerte terminalen.

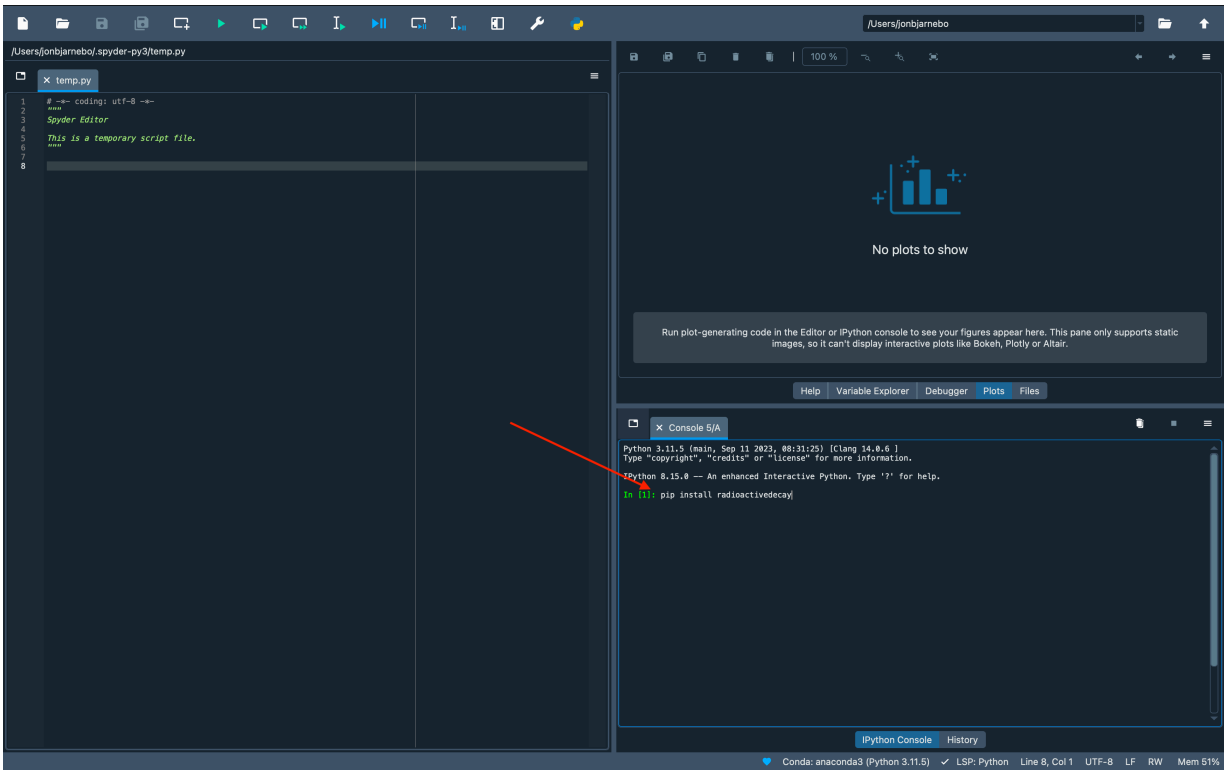
Når du har åpnet terminalen limer du inn linjen under:

```
pip install radioactivedecay
```

Under viser jeg fremgangsmåten for å installere dette biblioteket i Spyder og i Thonny (fordi det er de mest brukte IDE-ene i skolen).

Installasjon av radioactivedecay i Spyder

Åpne Spyder og finn den integrerte terminalen. Den ligger vanligvis nede i venstre hjørne slik som i bildet under. Skriv inn `pip install radioactivedecay` i terminalen og press **enter**.



Figur 1: Skjermbilde fra Spyder

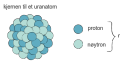
Når disse bibliotekene nå er installert er straks på tide å jobbe med oppgaver, men først litt teori.

Atomkjerner, protoner, nøytroner og isotoper

For å kunne arbeide med oppgaver må vi ha litt grunnleggende forståelse for hva radioaktivitet er, og da må vi først se på noen begreper. Begrepene vi skal ta for oss først er *nuklide*, *atom*, *atomnummer*, *nukleontall*, *grunnstoff*, *isotop*.

Et **radioaktivt stoff** har ustabile **atomkjerner** som sender ut stråling. For å kunne beskrive radioaktivitet, må vi derfor kunne litt mer om atomkjerner.

Alle atomer har en kjerne som består av **protoner** og **nøytroner**. (Et unntak er kjernen til vanlig hydrogen som består av ett proton og ingen nøytroner.)



Figur 6

Protoner er positivt ladet, mens **nøytroner** er nøytrale. Protoner og nøytroner har fellesbetegnelsen **nukleoner**.

Rundt kjernen svirrer elektronene. De er ikke vist i figuren til høyre fordi de i denne målestokken vil befinne seg flere hundre meter fra kjernen.

For å få en liten forståelse for noen flere begreper skal du nå få gjøre en liten øvelse hentet fra viten.no:

I tillegg til at vi skiller mellom ulike grunnstoff finnes det også flere varianter av samme grunnstoff. Disse kalles **isotoper** av grunnstoffet. Under ser vi to isotoper av grunnstoffet uran. Ser du hva som er forskjellig?

Figur 7: To isotoper av Uran

Om du kikker litt godt på bildet over vil du se at nukleontallet til de to isotopene er ulikt. Isotoper av samme grunnstoff har altså ulikt antall nøytroner i kjernen. Det er alltid antall nøytroner som varierer i de ulike isotopene siden grunnstoffet er det samme (og kun er bestemt av antall protoner i kjernen).

For å skille isotopene til et grunnstoff fra hverandre, tar vi med nukleontallet som en del av grunnstoffnavnet slik som vist i figuren til høyre.

Figur Kobolt-60

Vi tar ofte med nukleontallet i navnet når vi oppgir isotopen, og navnet til koboltisotopen med 60 nukleoner som vist i Figur 8 blir da Kobolt-60.

Nå har vi trukket frem begrepet **isotop** og definert det som ulike variasjoner av et grunnstoff. Kobolt-60 er altså en isotop av grunnstoffet kobolt. Vi kan også kalle Kobolt-60 for en **nuklide**.

Nuklider

En **nuklide** er et atom med et bestemt antall protoner og nøytroner i atomkjernen. Alle nuklider med samme protontall, det vil si med samme atomnummer, hører til samme grunnstoff. Nuklider med samme antall protoner, men forskjellig antall nøytroner, er ulike isotoper av det samme grunnstoffet. Slike isotoper kan være stabile eller radioaktive avhengig av antall nøytroner i kjernen.

Ulike typer radioaktiv stråling

De fleste atomene rundt oss er stabile og endrer seg ikke over tid. Radioaktive stoffer består av atomer med **ustabile atomkjerner**. For å bli mer stabile, vil kjernene før eller senere sende ut stråling. Når dette skjer, avgir kjernene energi. Vi sier at stoffet **henfaller**. Radioaktive atomkjerner kan sende ut tre typer stråling: **alfastråling**, **betastråling** og **gammastråling**.

Alfastråling

Når en atomkjerne sender ut alfastråling sendes det ut noe vi kaller en alfapartikkel, som er en heliumkjerne (altså en kjerne bestående av to nøytroner og to protoner). Når dette skjer vil jo også atomkjernen som sender ut denne strålingen endres slik at den får to færre nøytroner og to færre protoner i kjernen. Se på animasjonen under og forsøk å finn ut hvilken Thoriumisotop som dannes når Uran-238 sender ut alfastråling.

Alfastråling

Alfastråling er en strøm av alfapartikler som består av to protoner og to nøytroner. Et grunnstoff som avgir alfastråling, blir omdannet til et nytt grunnstoff. Atomnummeret blir redusert med to og nukleontallet med fire.

Betastråling

Når en atomkjerne sender ut betastråling sendes det ut noe vi kaller en betapartikkel, som er et elektron i høy fart. Du lurer muligens på hvordan dette kan sendes ut fra kjernen som kun består av protoner og nøytroner? Gå gjennom den korte animasjonen under og finn det ut!

I animasjonen over ser vi at atomkjernen som sender ut betastråling ender opp med et ekstra proton i kjernen og dermed også blir til et isotop av et annet grunnstoff.

Betastråling

Betastråling er en strøm av elektroner. Et grunnstoff som avgir betastråling, blir omdannet til et nytt grunnstoff. Atomnummeret blir ett større, men nukleontallet endrer seg ikke.

Ikke pensum, men kanskje interessant?

En betapartikkel kan også være positivt ladet. Da kalles partikkelen for en **positron** og er antipartikkelen til elektronet. Det vil si at den har samme masse som elektronet, men motsatt ladning. Dette skjer når det er et proton som deles og blir da til et nøytron og et positron.

Gammastråling

Gammastråling er elektromagnetisk stråling og altså ikke partikkelstråling slik som alfa- og betastråling. Ved utsendelse av gammastråling endres ikke sammensetningen av kjernen annet enn at den blir mer stabil.

Gammastråling

Gammastråling er elektromagnetisk stråling. Den blir sendt ut av atomkjerner som har sendt ut alfa- eller betastråling, men som fortsatt har et overskudd av energi.

Oppgaver

Nå som vi har sett litt på de ulike typene radioaktiv stråling skal vi bli litt mer kjent med ulike isotoper ved å bruke blant annet programmering. For å kunne finne ulike grunnstoffer og atomnummer kan det være nyttig å også bli litt kjent med periodesystemet.

Det viktigste vi skal kunne hente ut av periodesystemet i oppgavene er atomnummer og grunnstoff. Resten er mindre relevant for oss nå, og vi går derfor ikke noe mer inn på oppbyggningen til periodesystemet.

Oppgave 1

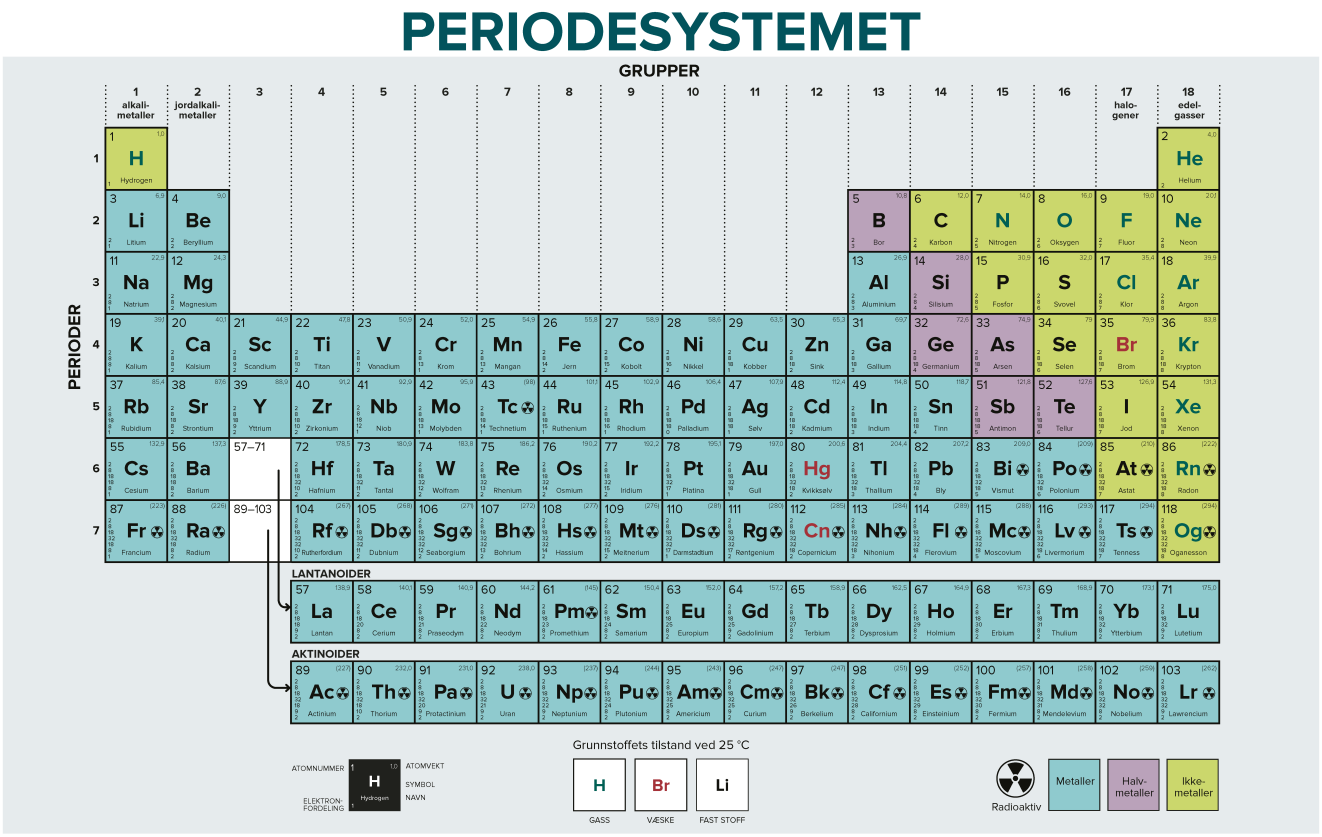
Finn atomnummeret til Radon-222.

Oppgave 2

Kopier koden under og noter ned hva som skjer.

```
import radioactivedecay as rd

nuklide = rd.Nuclide("Rn-222")

print(nuklide.A)
print(nuklide.Z)
```

Oppgave 3



Figur 9: Periodesystemet hentet fra Senit i 2024

Tabell 1: Tabell over noen radioaktive nuklider for ulike grunnstoff

Radioaktive nuklider
Uran-238
Thorium-234
Plutonium-239
Karbon-14
Cesium-137

Basert på det du fant ut i oppgave 2. Utforsk noen andre radioaktive isotoper for andre grunnstoffer fra tabellen til høyre. Undersøk om det mønsteret du fant også gjelder for disse.

Oppgave 4

Vi skal nå utvide koden litt for å få et innblikk i hvilken strålingstype som blir sendt ut fra de ulike nuklidene i Tabell 1.

I det forrige programmet du skrev (i oppgave 2) legg til følgende kodelinje:

```
strålingstype = nuklide.decay_modes()
print(strålingstype)
```

Om du har gjort alt riktig nå bør du få printet ut [‘ ’] som betyr at Radon-222 sender ut alfastråling.

Du skal nå gjøre endringer i programmet slik at du kan finne ut hvilken type stråling som sendes ut fra nuklidene i Tabell 1.

Ulike typer radioaktiv stråling

Som du vil se etter hvert som du tester ut ulike nuklider så er noen tilfeller hvor man får ulike typer radioaktiv stråling. Selv om vi har dekket de fleste av disse typene tidligere er det noe vil vil møte på her som kan være greit å vite om:

- **SF**: Står for *Spontaneous Fission*, eller *spontan fissjon* på norsk. Dette innebærer at tunge atomkjerner splittes til mindre atomkjerner spontant uten ytre påvirkning og dermed frigjør energi. Dette er ikke noe vi trenger å tenke på i dette opplegget, men søk gjerne opp spontan fissjon om du vil undersøke dette mer på egenhånd.
- **IT**: Står for *Isomeric Transition* som på norsk kan oversettes til *isomerisk overgang*. Dette er når man har en nuklide som er i eksitert tilstand og som deretter går over til en mer stabil tilstand (som oftest ved å sende ut gammastråling).

Henfallskjeder

I denne inndelingen skal vi se på noe som kalles for henfallskjeder, som henger tett sammen med det vi nettopp har arbeidet med. Ta derfor vare på programmene du har laget og erfaringene du har gjort deg så langt.

Introduksjon

Vi kan nå legge til en kodelinje til for å se på hvilke av de ulike strålingstypene (for de nuklidene som har flere) som er mest sannsynlige. Dette kan vi gjøre ved å legge inn linjen under.

```
sannsynlighet = nuklide.branching_fractions()
print(sannsynlighet)
```

Om vi kjører denne koden nå vil programmet skrive ut sannsynlighetene for de ulike strålingstypene til nukliden vi gir som input (tall mellom 0 og 1).

La oss nå skrive om programkoden litt slik at den blir litt mer kompakt og inkludere linjen over også. Under ser du koden sammen med outputen (kopier den gjerne inn selv der du skriver kode).

```
1  import radioactivedecay as rd
2
3  nuklidenavn = "Pb-210"
4  nuklide = rd.Nuclide(nuklidenavn)
5  atomnummer = nuklide.Z
6  nukleontall = nuklide.A
7  strålingstype = nuklide.decay_modes()
8  sannsynlighet = nuklide.branching_fractions()
9
10 print(f"{nuklidenavn} har atomnummer {atomnummer} og nukleontall {nukleontall}.")
11 print(f"{nuklidenavn} sender ut {strålingstype} stråling.")
12 print(f"Sannsynligheten for de ulike stråletypene er {sannsynlighet}.")
```

```
Pb-210 har atomnummer 82 og nukleontall 210.
Pb-210 sender ut [' -', ' '] stråling.
Sannsynligheten for de ulike stråletypene er [1.0, 1.9e-08].
```

Som vi ser i outputen over sender nukliden Pb-210 (som er en blyisotop) ut både alfastråling og betastråling. Dette kan vi se når programmet skriver ut [**' -', ' '**]. For å se hvilke strålingstype som er mest vanlig kan vi se på det som blir skrevet ut fra linje 12: [**1.0, 1.9e-08**]. Verdiene står i rekkefølge (altså er den første sannsynligheten i lista til den første stråletypen i den andre lista). Dette betyr at nesten all radioaktiv stråling er betastråling, mens en veldig liten del av strålingen fra Pb-210 er alfastråling ($1.9 \cdot 10^{-8}$, altså kun 0.0000019%).

> Verdien som står først i lista er alltid den verdien som er mest sannsynlig. Står f.eks. β^- først i lista er det betastråling som er mest sannsynlig at kommer til å skje. Om vi kun ønsker den mest sannsynlige verdien kan vi da skrive `nuklide.decay_modes()[0]` for å kun få den mest sannsynlige reaksjonen i stede for å få en komplett liste med alternativer.

Utforskning

Du skal nå utforske litt mer rundt hva som skjer når Uran-238 sender ut stråling.

a) Ta utgangspunkt i programkoden fra eksempelet over og gjør tilsvarende med Uran-238.

b) Finn ut hvilket nytt grunnstoff som blir dannet **om det som er mest sannsynlig skjer**. Bruk gjerne periodesystemet fra Figur 9 til å finne det nye grunnstoffet. Regn deg også frem til hvilken isotop av dette grunnstoffet det er vi får og skriv ned denne nye nukliden.

c) Gjenta oppgave *a* og *b* for den nye nukliden.

d) Når du nå har fått enda en ny nuklide skal du gjenta prosessen en siste gang.

I oppgaven over ender vi opp med en liten rekke av fire ulike nuklider. Hvilke grunnstoff og hvilke isotoper av grunnstoffet er avhengig av typen radioaktiv stråling, og begynnelsesnukliden vår. Om du fikk til oppgaven over ender du opp med variant av det som vi kaller for en henfallskjede.

Henfallskjeden vi nå har fått svarer til den som vises i animasjonen under som du nå skal gjøre:

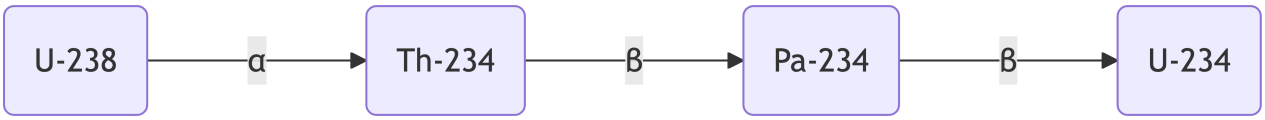
Det er også mulig å finne disse *datternuklidene* direkte ved å bruke en funksjon i *radioactivedecay*-biblioteket kalt *progeny*. Denne funksjonen oppgir hvilken nuklide vår opprinnelige nuklide blir omdannet til. Under ser du et eksempel på dette.

```
import radioactivedecay as rd

nuklidenavn = "U-238"
nuklide = rd.Nuclide(nuklidenavn)
print(nuklide.progeny())
```

```
['Th-234', 'SF']
```

Om vi nå kun er ute etter det mest sannsynlige utfallet kan vi velge å kun bruke første verdien i lista. Da printer vi slik istede: `print(nuklide.progeny()[0])`.

Når vi har gjort tre ganger med disse ulike nuklidene vi får kan vi evt. tegne inn dette i et lite diagram, f.eks. noe som dette her:

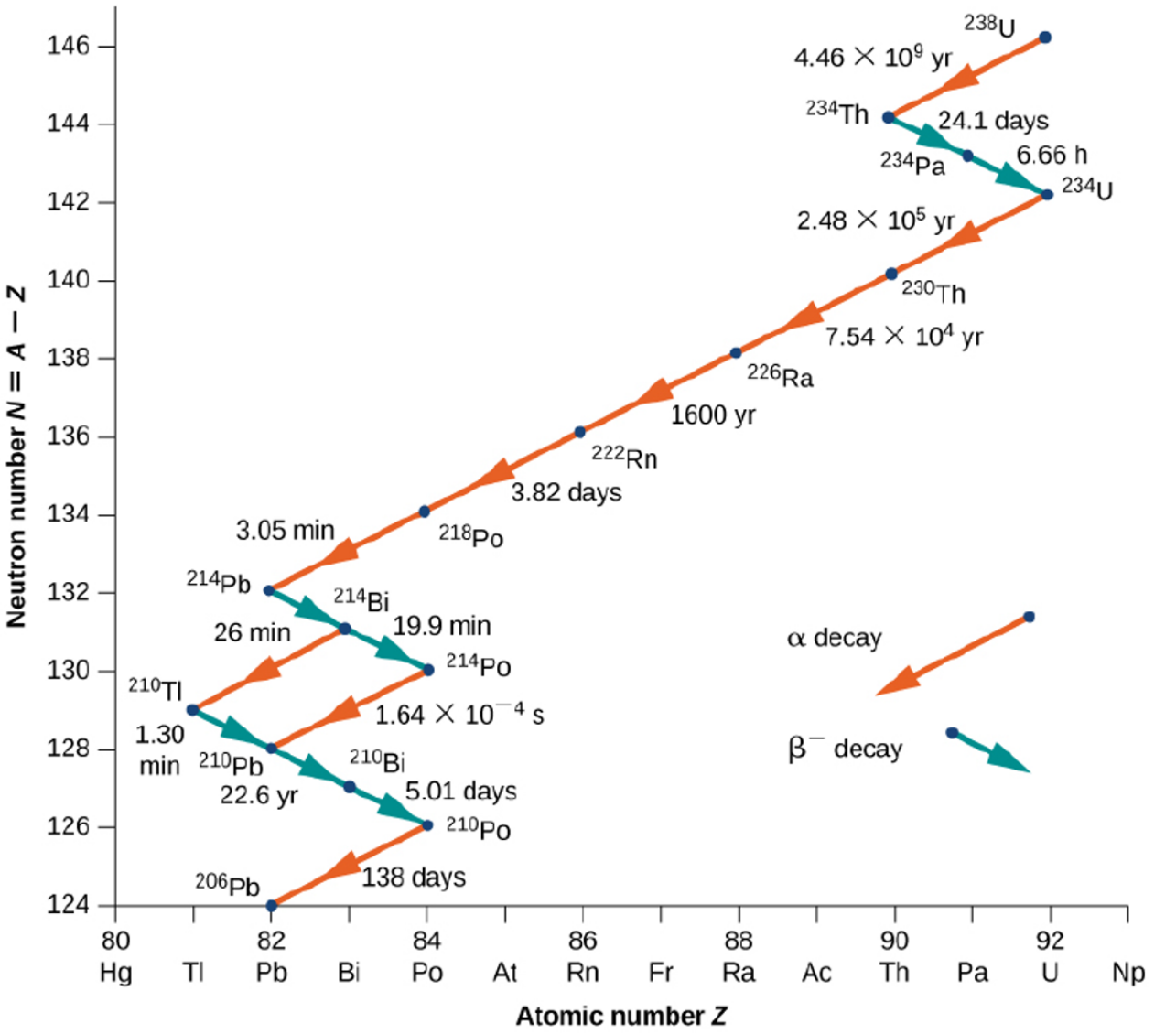


Lage henfallskjeder

Som vi nå har sett er det mulig å lage henfallskjeder ved å undersøke hvilken stråletype som sendes ut av en nuklide og deretter regne seg frem til resultatet av at strålingen sendes ut, finne dette nye grunnstoffet i en tabell og til slutt tegne opp en slik henfallskjede systematisk. Noen henfallskjeder kan være ganske lange, mens andre kan være kortere. Et eksempel på en slik henfallskjede som kan være ganske lang er den som kalles *uran-radium-serien* og som starter med U-238. Under ser du et bilde av denne serien.

Som vi kan se av bildet over ender denne serien opp som stabilt bly, men siden halveringstiden til Uran-238 er så lang tar denne serien lang tid. Vi blir med andre ord ikke tomme for Uran-238 med det første.

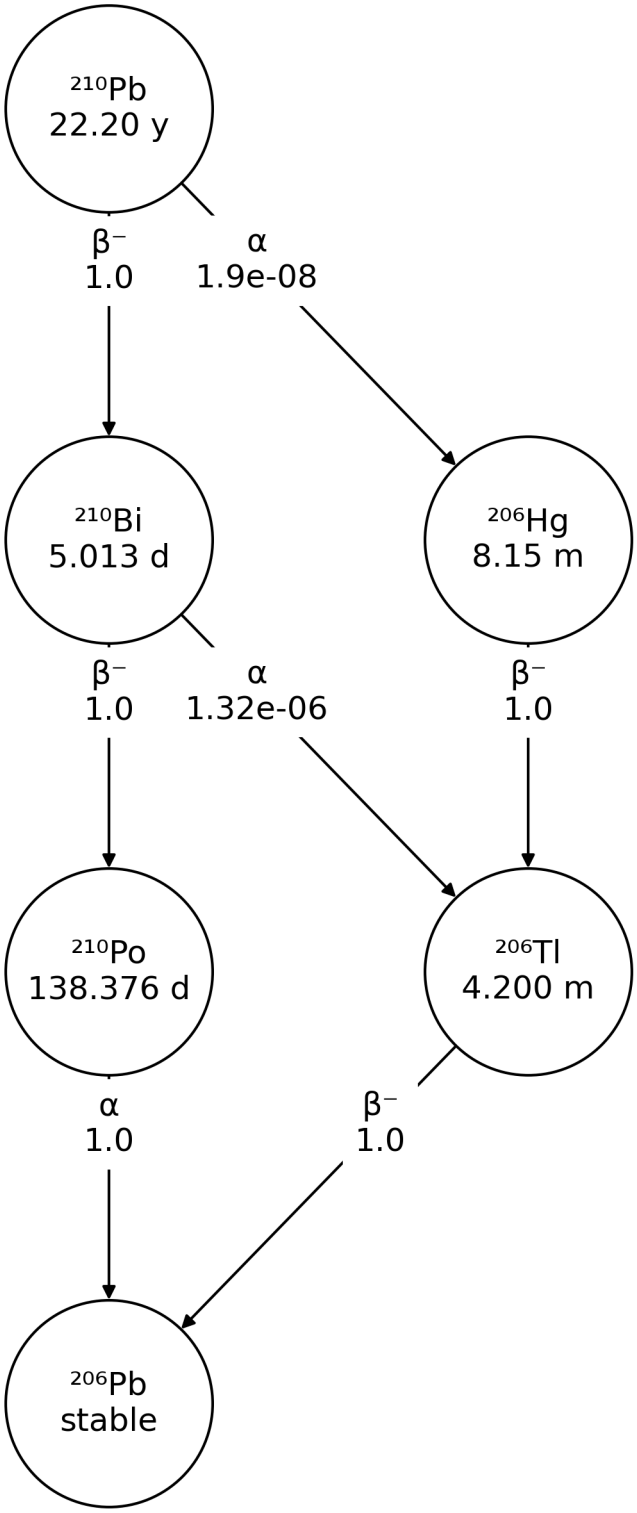
Vi kan også lage slike henfallskjeder med hjelp av programmering. Dette er ganske enkelt å gjøre i Python og krever lite kode.



Figur 10: Uranserien

Under ser vi et program som tegner en henfallskjede hvor vi starter med den ustabile bly-isotopen Pb-210.

```
import radioactivedecay as rd
import matplotlib.pyplot as plt

nuklidenavn = "Pb-210"
nuklide = rd.Nuclide(nuklidenavn)
fig, ax = nuklide.plot(label_pos=0.35)
plt.show()
```


210Pb
22.20 y
β−
1.0
α
1.9e-08
210Bi
5.013 d
206Hg
8.15 m
β−
1.0
α
1.32e-06
β−
1.0
210Po
138.376 d
206Tl
4.200 m
α
1.0
β−
1.0
206Pb
stable

Utforskningsoppgave om henfallskjeder

Det finnes 4 store henfallskjeder/serier:

- *uran-radium-serien* (starter med U-238)
- *thoriumserien* (starter med Th-232)
- *uran-actinium-serien* (starter med U-235)
- *neptuniumserien* (starter med Np-237)

Uran-radium-serien, *thoriumserien* og *uran-actinium-serien* finnes alle i naturen i dag, men det gjør ikke *neptuniumserien*. På samme måte som i programmeringseksempelet over kan vi nå lage henfallskjeder også for disse store seriene.

Oppgave: Lag en henfallskjede for alle fire seriene og forsøk å finne ut hvorfor ikke neptuniumserien finnes i naturen basert på henfallskjedene du lager.

For estetikere og spesielt interesserte

Det er fult mulig å gjøre diagrammene litt penere. Dette er beskrevet i dokumentasjonen til radioactivedecay, men kan være litt kronglete og vanskelig å finne. Jeg har derfor valgt noen av de tilpassingene jeg mener er nyttigst og lagt dem inn i en liten kodesnutt med kommentarer som du kan kopiere og lime inn om du ønsker å utforske litt selv rundt det estetiske ved disse henfallskjedene. Koden ser du ved å utvide på knappen "Kode" under. Om du ønsker å se litt mer på dokumentasjonen til det underliggende biblioteker kan du finne det her.

```
import radioactivedecay as rd

nuklide = rd.Nuclide("Rn-222")

kwargs_draw = {
    "font_size": 12,  # Tekststørrelsen i nodene
    "node_size": 6000,  # Størrelsen på nodene
    "node_color": "#FFDDC1",  # Bakgrunnsfargen til nodene
    "edge_color": "#FF5733",  # Fargen til pilene mellom nodene
    "font_color": "#000000",  # Fargen på teksten i noden
    "edgecolors": "#000000",  # Fargen på kantlinjen til noden
    "node_shape": "o",  # Bestem formen på noden (må være en av disse: so^>v<dph8)
    "linewidths": 1,  # Bestem størrelsen på kantlinjen rundt nodene
    "arrowsize": 20,  # Størrelsen på pilene
}

kwargs_edge_labels = {
    "font_size": 12,  # Tekststørrelse til teksten utenfor nodene
    "bbox": {
        "boxstyle": "round",
        "fc": "none",
        "ec": "none",
    },  # Konfigurering av boksene utenfor nodene
    "rotate": False,  # Rotasjon/ikke rotasjon av teksten utenfor nodene
}

fig, ax = nuklide.plot(
    label_pos=0.36, kwargs_draw=kwargs_draw, kwargs_edge_labels=kwargs_edge_labels
)
```

Halveringstid

For en gitt atomkjerne er det alltid en fast sannsynlighet for at en radioaktiv omdanning vil skje. Dette betyr at vi aldri kan forutsi når en reaksjon vil skje for hver enkelt atomkjerne. Men dersom vi har mange atomkjerner, vet vi at en fast prosent av dem kommer til å bli omdannet. Dette er grunnen til at vi snakker om radioaktive stoffers halveringstid, tida det tar før halvparten av atomkjernene er omdannet.

Halveringstid

Halveringstida til et stoff er tida det tar før halvparten av atomkjernene er omdannet.

Halveringstida til en bestemt atomkjerne er alltid den samme, og vi kan ikke påvirke den. Den kan variere fra mindre enn en milliarddels sekund til mange ganger universets alder!

Vi bruker ofte symbolet $T_{1/2}$ om halveringstid. Om vi skal representere en mengde som er igjen av et stoff eller aktiviteten til stoffet over tid kan vi representere dette med en eksponentialfunksjon hvor vekstfaktoren er $\frac{1}{2}$ fordi verdiene halveres hver gang stoffmengden halveres. Vi kan da bruke funksjonen

$$f(x) = a \cdot (\frac{1}{2})^{\frac{t}{T_{1/2}}}$$

der a er verdien vi starter med (ofte er dette enten registrert startaktivitet eller masse), t er tiden vi måler (verdien på x-aksen) og $T_{1/2}$ er halveringstiden til stoffet.

Teoretisk halveringstid

Alle radioaktive stoffer har som tidligere nevnt en halveringstid. Denne kan vi slå opp for å finne i ulike tabeller. Under ser du en oversikt over noen utvalgte nuklider med oppgitt halveringstid.

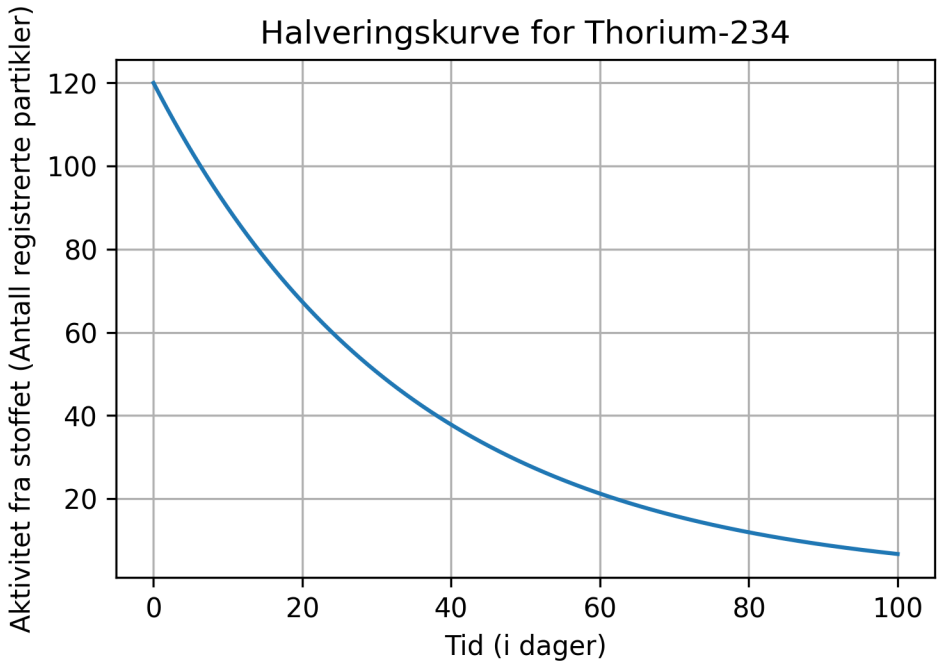
Tabell 2: En oversikt over ulike halveringstider til noen isotoper

Isotop	Halveringstid
Uran-238	4,5 milliarder år
Plutonium-239	24 000 år
Karbon-14	5730 år
Cesium-137	30 år
Thorium-234	24 dager
Radon-222	4 dager
Astat-211	7,2 timer
Polonium-214	0,00016 sekunder

Vi skal først se på et program som plotter en halveringskurve for Thorium-234 med utgangspunktet i data fra Tabell 2.

Liste 1 halveringstid.py

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


# Definerer en funksjon som regner ut ny verdi etter en gitt tid har gått
def halvering(tid, halveringstid, start_aktivitet):
    return start_aktivitet * 0.5 ** (tid / halveringstid)


tid = 100  # Valgt tid (i dager)
halveringstid = 24  # Halveringstid til Thorium-234 (i dager)
aktivitet = 120  # Stoffets aktivitet i starten (en fiktiv startverdi)

t_verdier = np.linspace(0, tid, 1000)

plt.plot(t_verdier, halvering(t_verdier, halveringstid, aktivitet))
plt.title("Halveringskurve for Thorium-234")

plt.xlabel("Tid (i dager)")
plt.ylabel("Aktivitet fra stoffet (Antall registrerte partikler)")
plt.grid(True)
plt.show()
```



Figur 11: Enkel halveringsgraf for Thorium-234

Over ser du en halveringsgraf. Om vi går inn i Tabell 2 og leter frem Thorium-234 vil vi se at halveringstiden er 24 dager. Om vi så går inn på 24 på x-aksen vår i plottet over og leser av verdien vil vi få 60, som er halvparten av det vi startet med.

Oppgave 5

I denne oppgaven skal du kopiere koden over og endre på noen av verdiene for å gjøre deg kjent med hvordan koden fungerer.

a) Endre på variabelen "aktivitet". Sett den til en mye høyere verdi og en mye lavere verdi. Hva skjer?

b) Endre på variabelen "halveringstid" og se hva som skjer når du stiller denne inn til å være mindre og større.

c) Endre den simulerte tiden (variabelen som heter "tid") til å være kun 20 dager. Hva skjer? Ser dette bra ut og hvorfor skjer dette?

Som du muligens så når du arbeidet med forrige oppgave blir aksene automatisk skalert i Python, noe som kan føre til noen misforståelser når vi skal lese grafene. For å få Python til å alltid vise akseverdiene fra 0 og opp til de verdiene som grafene viser kan vi legge til følgende to linjer med kode:

```
plt.ylim(0, aktivitet + 1)
plt.xlim(0, tid + 1)
```

Disse to linjene må plasseres et eller annet sted mellom `plt.plot()` og `plt.show()` i koden.

Tips for den spesielt interesserte

Det er også mulig å legge inn en liten kodesnutt slik at halveringsverdiene og tidspunktene blir synlige som punkter i plottet. En måte å gjøre dette på kan være å legge til koden under:

```
if tid > halveringstid:
    for i in range(int(tid)//int(halveringstid)):
        t_halv = halveringstid*(i+1)
        if t_halv < tid:
            t_halveringsverdier.append(t_halv)
            halveringsverdier.append(halvering(t_halv, halveringstid, aktivitet))
    plt.plot(t_halveringsverdier, halveringsverdier, ".")
    for x, y in zip(t_halveringsverdier, halveringsverdier):
        plt.text(x, y, f"({round(x, 2)}, {round(y, 2)})")
```

Oppgave 6

Lag en halveringsgraf for en annen radioaktiv nuklide fra Tabell 2. Husk å endre på benevninger på x-aksen slik at denne stemmer med den teoretiske verdien. Kontroller grafisk om kurven kan se riktig ut basert på det du vet om halveringstiden. *Ønsker du en liten utfordring kan du også implementere tipset over i koden din.*

Hente halveringstider i Python

Vi kan også bruke biblioteket *radioactivedecay* til å hente ut data om halveringstid til en nuklide slik vi har brukt det til andre formål tidligere. Om vi vil hente ut data om halveringstid kan vi bruke funksjonen som kalles for `half_life()` slik:

```
import radioactivedecay as rd

nuklidenavn = "Th-234"
nuklide = rd.Nuclide(nuklidenavn)

halveringstid = nuklide.half_life("d")
print(halveringstid)
```

```
24.1
```

Som vi ser over printer på programmet mitt halveringstiden og jeg kan nå bruke denne i programmet vi lagde tidligere for å plotte halveringsgrafen.

Oppgave 7

Kopier programkoden over og fjern "d" i `half_life()`-funksjonen. Hva skjer?
Prøv nå å bytte ut "d" med f.eks. "m" eller "y". Hva skjer nå?
Forsøk å komme med en forklaring på dette.

Oppgave 8: En liten utfordring

Om du har kontroll på dette nå kan en liten utfordring være å implementere det du kan om *radioactivedecay*-biblioteket slik at du lager et program som plotter en halveringsgraf med utgangspunkt i halveringstiden du får fra *radioactivedecay*-biblioteket.

Som du muligens ser over må vi prøve oss litt frem her for å finne riktig "input" i funksjonen. Dette er siden funksjonen som standard oppgir halveringstiden i sekunder. For å slippe å regne det om selv kan vi gi funksjonen en input. De mest nyttige inputene er listet opp under:

```
nuklide.half_life("readable")  # Oppgir halveringstid som en lesbar tekst
nuklide.half_life("m")  # Oppgir halveringstid i minutter
nuklide.half_life("h")  # Oppgir halveringstid i timer
nuklide.half_life("d")  # Oppgir halveringstid i dager
nuklide.half_life("y")  # Oppgir halveringstid i år
```

Bruker vi "readable" vil vi få et tall sammen med en egnet enhet:

```
nuklide.half_life("readable")
```

```
'24.10 d'
```

Finne enheten

Om vi kun ønsker å hente ut enheten kan vi bruke *slicing*-metoder i Python:

```
nuklide.half_life("readable").split(" ")[-1]
```

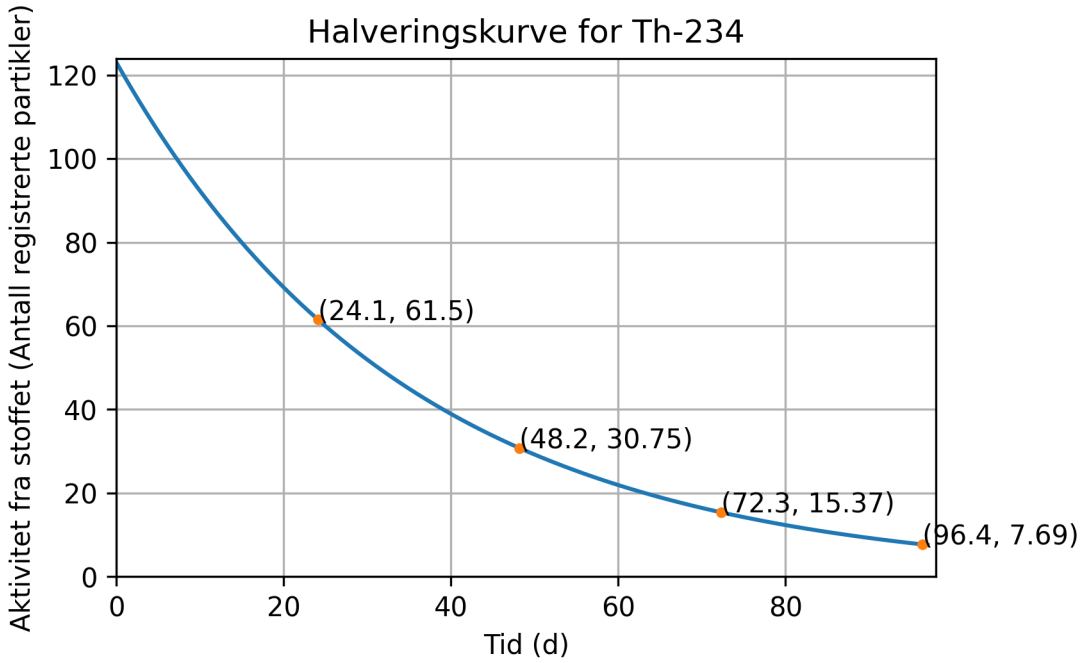
```
'd'
```

Oppsummert så har vi nå sett en del på teoretisk halveringstid, og om vi implementerer alt vi har gjennomgått over vil vi ha et program som kan lage grafen under ved å hente halveringstider direkte fra *radioactivedecay*-biblioteket.

Om du har stått fast eller har noen feil i koden din, kan du nå åpne "Kode"-knappen under (mellom denne teksten og grafen) og kopiere koden for å se om det også fungerer fint hos deg.

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

nuklide = rd.Nuclide("Th-234")

# Definerer en funksjon som regner ut ny verdi etter en gitt tid har gått
def halvering(tid, halveringstid, start_aktivitet):
    return start_aktivitet * 0.5 ** (tid / halveringstid)

# Henter ut enheten til halveringstiden
enhet_på_aksen = nuklide.half_life("readable").split(" ")[-1]

halveringstid = nuklide.half_life(enhet_på_aksen)

# Ønsket simuleringstid settes til 4 * halveringstiden (rundet opp):
tid = np.ceil(halveringstid * 4)

# Stoffets aktivitet i starten (en bestemt/målt startverdi)
aktivitet = 123

t_verdier = np.linspace(0, tid, 1000)

plt.plot(t_verdier, halvering(t_verdier, halveringstid, aktivitet))
plt.title(f"Halveringskurve for {nuklide.nuclide}")

halveringsverdier = []
t_halveringsverdier = []

# Koden mellom denne kommentaren og neste kommentar er for å generere punkter i grafen hver gang
if tid > halveringstid:
    for i in range(int(tid) // int(halveringstid)):
        t_halv = halveringstid * (i + 1)
        if t_halv < tid:
```

```
            t_halveringsverdier.append(t_halv)
            halveringsverdier.append(halvering(t_halv, halveringstid, aktivitet))
    plt.plot(t_halveringsverdier, halveringsverdier, ".")
    for x, y in zip(t_halveringsverdier, halveringsverdier):
        plt.text(x, y, f"({round(x, 2)}, {round(y, 2)})")
# Koden over kan fint sløyfes om man ikke ønsker disse halveringspunktene synlige i grafen.

plt.ylim(0, aktivitet + 1)
plt.xlim(0, tid + 1)
plt.xlabel(f"Tid ({enhet_på_aksen})")
plt.ylabel("Aktivitet fra stoffet (Antall registrerte partikler)")
plt.grid(True)
plt.show()
```



Figur 12: Halveringsgraf for Thorium-234 over 97 dager med punkter for hver halvering

Halveringstid i det virkelige liv (Forsøk)

I virkeligheten er det sjelden slik at man får fine glatte grafer med data om man velger å gjennomføre eksperimenter for å måle halveringstider. Denne tilfeldigheten som spiller inn for desintegrasjonen (omdanningen) som vi har sett på teoretisk er vanskelig å erfare ute å gjennomføre et eksperiment. Derfor skal vi nå gjennomføre et eksperiment hvor målet vårt er å finne halveringstiden til en nuklide kalt Ba-137m.

Om dere skal gjennomføre forsøket på labben selv ligger forsøksbeskrivelsen tilgjengelig her. Om dere ikke har utstyr, eller bare skal være litt mer effektive og ikke gjennomføre det selv kan du lese videre og ta utgangspunkt i en video og data fra et forsøk som allerede er blitt gjennomført.

Demonstrasjonsforsøk av halveringstid til Ba-137m

Se videoen under enten felles i klassen eller individuelt (hør med lærer).

https://youtu.be/L-VmdbxcY80?si=VQVouxo7CYtnOaOv

Dataene fra forsøket kan lastes ned som en .csv-fil her. Om du heller vil kopiere dataene i tabellformat så kan du også se i tabellen til høyre for og lese av verdiene manuelt.

Under er tabellen med dataene i lesbart format.

Tabell 3: Data fra demonstrasjonsforsøket

Time (s)	Geiger Counts (counts/sample)	Time (s)	Geiger Counts (counts/sample)	Time (s)	Geiger Counts (counts/sample)
10	682	210	187	410	78
20	545	220	197	420	78
30	517	230	211	430	91
40	541	240	176	440	66
50	421	250	164	450	90
60	414	260	144	460	81
70	387	270	173	470	70
80	383	280	147	480	64
90	332	290	137	490	60
100	365	300	124	500	64
110	322	310	123	510	55
120	315	320	117	520	52
130	262	330	130	530	49
140	277	340	116	540	51
150	283	350	121	550	39
160	260	360	138	560	60
170	245	370	93	570	50
180	212	380	106	580	53
190	240	390	96	590	50
200	227	400	87	600	35

Plotting av halveringsgrafer basert på data fra csv-filer

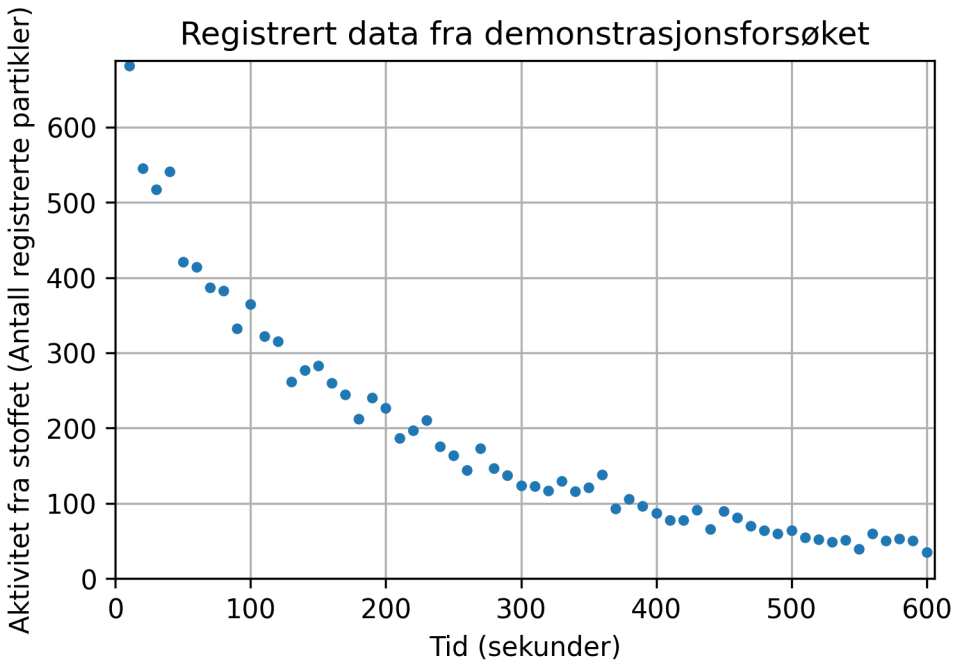
I denne delen skal vi se på hvordan vi kan plotte grafer basert på data fra faktiske gjennomførte forsøk i Python. Etter vi har sett på dette skal vi bruke det vi har lært tidligere til å plotte den simuerte grafen sammen med de faktiske punktene i samme koordinatsystem for å kunne sammenlikne teori med praksis. Utgangspunktet for punktene som vi plotter i denne delen er data fra forsøket over, men om du har gjennomført forsøket selv bruker du dine egne data. Da vil også punktene naturligvis være forskjellige fra de som blir vist i grafene.

Csv-filen som er utgangspunktet for punktene vi snart skal plotte kan lastes ned her om du ønsker å bruke samme data. Dette er data fra demonstrasjonsforsøket over og som er presentert i Tabell 3.

> **! Programmeringskunnskaper**
>
> Utgangspunktet for denne delen av plottingen er at du har kjennskap til og tidligere har brukt *matplotlib*-biblioteket i Python.
> Du kan fremdeles få mye igjen for å fullføre oppgavene under uavhengig av om du har kjennskap til dette fra før, men jeg kommer ikke til å forklare mye rundt disse bibliotekene i dette dokumentet.

Vi kan plotte dataene fra forsøket i Python ved å bruke *matplotlib*-biblioteket. Et forslag til program med resultert graf kan sees under.

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

# Laster inn datafilen
data = np.loadtxt(
    "Halveringstid-ba-137m-data.csv", # Filnavnet til datafilen
    dtype=int,
    skiprows=1, # teller ikke med øverste rad fra dokumentet
    delimiter=";" # Skilletegnet som blir brukt mellom dataene
)

tid = data[:, 0]  # Skiller ut alle tidsverdiene
aktivitet = data[:, 1]  # Aktiviteten registrert av geigertelleren

plt.title("Registrert data fra demonstrasjonsforsøket")
plt.plot(tid, aktivitet, ".")  # Plotter datapunktene

plt.ylim(0, aktivitet[0] * 1.01)
plt.xlim(0, tid[-1] * 1.01)

plt.xlabel(f"Tid (sekunder)")
plt.ylabel("Aktivitet fra stoffet (Antall registrerte partikler)")

plt.grid(True)
plt.show()
```



Figur 13: Enkel halveringsgraf for Ba-137m basert på data fra demonstrasjonsforsøket

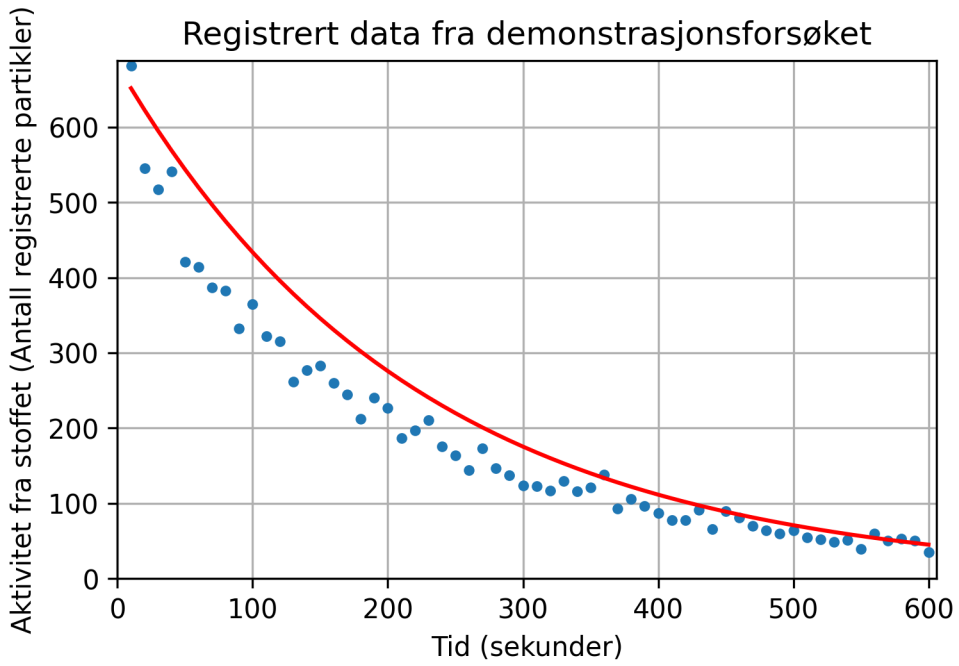
Nå som vi har en representasjon av forsøksdataene kan vi lage en simulert graf over forventet kurve basert på teori. For å slippe å lete frem halveringstiden til Ba-137m kan vi bruke *radioactivedecay*-biblioteket slik vi har sett eksempler på tidligere og sammenlikne grafene.

Oppgave 9

Kopier koden over og bruk det du har lært om tidligere til å plotte inn den teoretiske halveringsgrafen for *Ba-137m* sammen med punktene.
For å plotte grafen sammen med punktene kan du bare skrive inn en ny `plot()`-funksjon etter linje 16 i programkoden over (så lenge den skrives inn før `plt.show()`). Grafen bør se slik ut som i grafen under.

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

data = np.loadtxt(
    "Halveringstid-ba-137m-data.csv", dtype=int, skiprows=1, delimiter=";"
)

tid = data[:, 0]  # Skiller ut alle tidsverdiene
aktivitet = data[:, 1]  # Aktiviteten registrert av geigertelleren


# Definerer en funksjon som regner ut ny verdi etter en gitt tid har gått
def halvering(tid, halveringstid, start_aktivitet):
```

```
    return start_aktivitet * 0.5 ** (tid / halveringstid)

nuklidenavn = "Ba-137m"
nuklide = rd.Nuclide(nuklidenavn)
halveringstid = nuklide.half_life("s")  # Halveringstiden til Ba-137m i sekunder
start_aktivitet = aktivitet[0]  # Setter startaktiviteten til å være den samme som i forsøket

plt.title("Registrert data fra demonstrasjonsforsøket")
plt.plot(tid, aktivitet, ".")  # Plotter datapunktene
plt.plot(tid, halvering(tid, halveringstid, start_aktivitet), "-", color="red")

plt.ylim(0, aktivitet[0] * 1.01)
plt.xlim(0, tid[-1] * 1.01)

plt.xlabel(f"Tid (sekunder)")
plt.ylabel("Aktivitet fra stoffet (Antall registrerte partikler)")

plt.grid(True)
plt.show()
```



Figur 14: Teoretisk halveringsgraf for Ba-137m sammenliknet med datapunktene fra demonstrasjonsforsøket

Selv om det kan se ut som at den teoretiske modellen passer dårlig til de observerte datapunktene er det ikke helt sikkert at det trenger å være slik. Om vi ser godt etter kan det se ut som at datapunktene synker på samme måte, men at den teoretiske grafen kun ligger litt høyere. For å justere på høyden til denne simulerte teoretiske grafen kan vi endre på verdien til `start_aktiviteti` koden over.

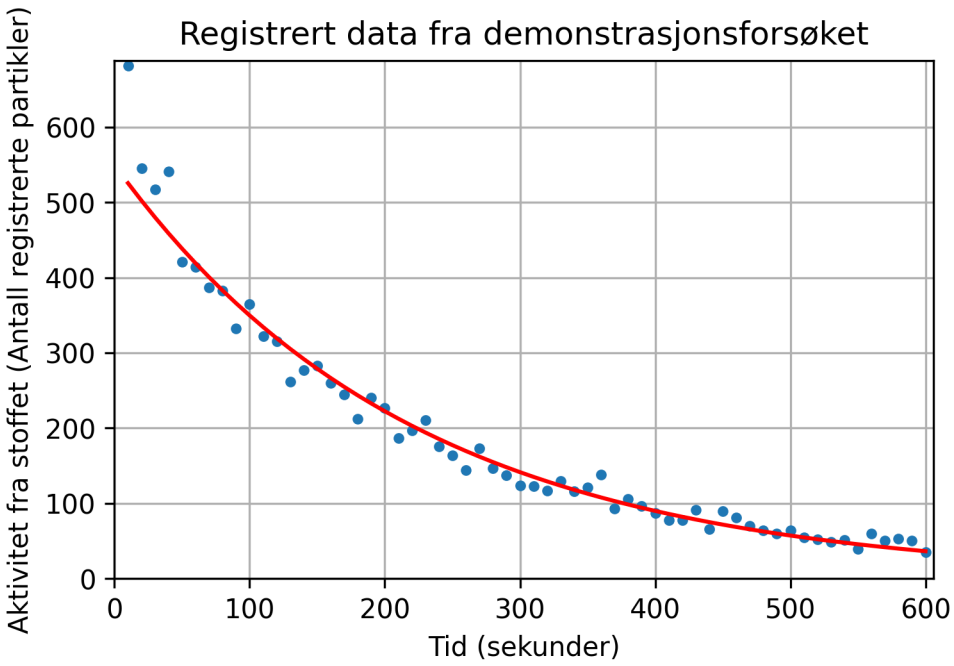
Oppgave 10

Endre på verdien til `start_aktivitet`-variabelen slik at den teoretiske grafen passer best mulig til datapunktene. Skriv inn eksakte verdier f.eks. 400 eller 700.

a) Prøv deg frem til en verdi som gjør at grafen passer godt (hvis det er mulig)
b) Hva forteller dette oss om:

- simuleringen?
- datasettet/målingene?

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

data = np.loadtxt(
    "Halveringstid-ba-137m-data.csv", dtype=int, skiprows=1, delimiter=";"
)

tid = data[:, 0]  # Skiller ut alle tidsverdiene
aktivitet = data[:, 1]  # Aktiviteten registrert av geigertelleren


# Definerer en funksjon som regner ut ny verdi etter en gitt tid har gått
def halvering(tid, halveringstid, start_aktivitet):
    return start_aktivitet * 0.5 ** (tid / halveringstid)

nuklidenavn = "Ba-137m"
nuklide = rd.Nuclide(nuklidenavn)
halveringstid = nuklide.half_life("s")  # Halveringstiden til Ba-137m i sekunder
start_aktivitet = 550  # Velger en ny tilpasset startaktivitet

plt.title("Registrert data fra demonstrasjonsforsøket")
plt.plot(tid, aktivitet, ".")  # Plotter datapunktene
plt.plot(tid, halvering(tid, halveringstid, start_aktivitet), "-", color="red")

plt.ylim(0, aktivitet[0] * 1.01)
plt.xlim(0, tid[-1] * 1.01)

plt.xlabel(f"Tid (sekunder)")
plt.ylabel("Aktivitet fra stoffet (Antall registrerte partikler)")

plt.grid(True)
plt.show()
```



Figur 15: Teoretisk halveringsgraf for Ba-137m med justert startaktivitet sammenliknet med datapunktene fra demonstrasjonsforsøket

Om vi justerer tilstrekkelig på startaktiviteten vil vi se at den teoretiske modellen faktisk ser ut til å stemme ganske godt med de fleste punktene med unntak av de første 4. Selv om vi nå har en graf som passer ganske godt har vi ikke en faktisk regresjonsmodell som kan hjelpe oss med å estimere halveringstiden. Vi kan altså pr. nå ikke si så mye annet enn at teorien ser ut til å stemme ganske greit og da kan vi muligens bare slå opp den teoretiske halveringsverdien og si oss fornøyd med det?

Om vi ønsker å finne en konkret matematisk modell som beskriver datapunktene vil derfor det enkleste være å gjennomføre en regresjon i Geogebra.

Oppgave 11

a) Legg inn datapunktene i Geogebra og bruk regresjonsanalyse til å finne en god modell til punktene.

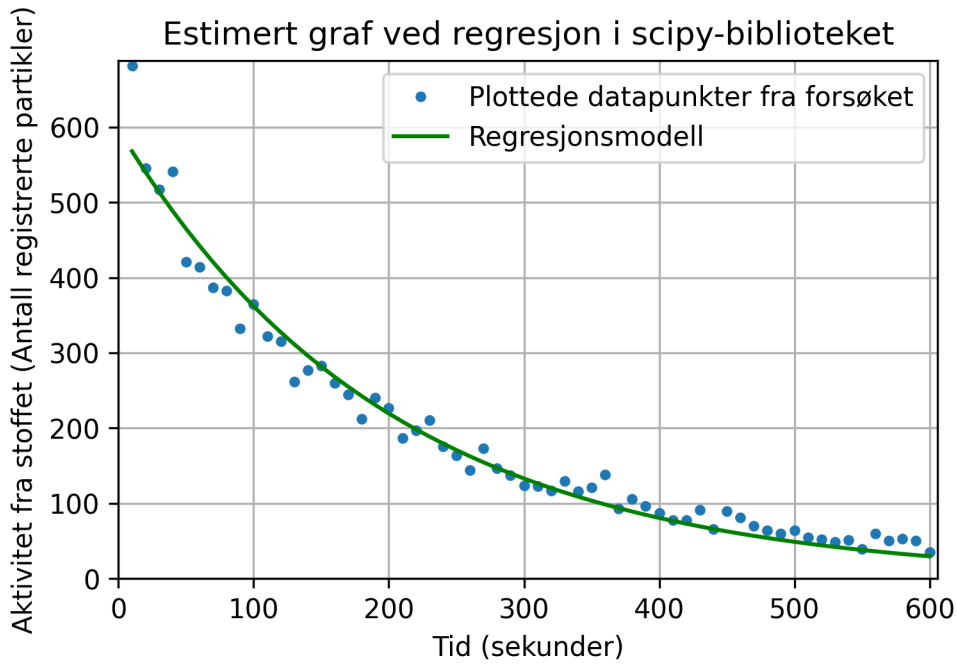
b) Legg inn regresjonsmodellen som du fikk i Geogebra i python-scriptet ditt og sammenlikn med datapunktene og den teoretiske modellen.

Regresjon i Python (Ekstramateriale for interesserte)

Det er også mulig å gjøre regresjon i Python. Dette er litt mer komplisert og krever et bibliotek kalt scipy for best optimaliserte funksjon. Under ser du et eksempel som viser oss hvordan vi kunne gått frem for å finne et estimat for en godt tilpasset eksponentialfunksjon. For å lese om dette i litt mer detalj kan man se noen gode eksempler på denne nettsiden, eller se i dokumentasjonen til scipy.

Når vi skal bruke dette biblioteket er det ikke mange ekstra linjene for å få en regresjonsmodell, men det kan være litt vanskelig å forstå hva som gir hva. Vi ser først på det endelige resultatet av modellen vår før jeg forklarer i litt mer detalj.

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


def halvering(tid, halveringstid, start_aktivitet):
    return start_aktivitet * 0.5 ** (tid / halveringstid)


p0 = (130, 1000)
parametre, cov = curve_fit(halvering, tid, aktivitet, p0)
halveringstid_est, start_aktivitet_est = parametre

plt.title("Estimert graf ved regresjon i scipy-biblioteket")
plt.plot(
    tid, aktivitet, ".", label="Plottede datapunkter fra forsøket"
)  # Plotter datapunktene
plt.plot(
    tid,
    halvering(tid, halveringstid_est, start_aktivitet_est),
    "-",
    color="green",
    label="Regresjonsmodell",
)
plt.legend()
plt.ylim(0, aktivitet[0] * 1.01)
plt.xlim(0, tid[-1] * 1.01)
plt.xlabel(f"Tid (sekunder)")
plt.ylabel("Aktivitet fra stoffet (Antall registrerte partikler)")
plt.grid(True)
plt.show()

print(f"Estimert halveringstid = {halveringstid_est:.2f}")
print(f"Faktisk halveringstid = {halveringstid}")
```




```
Estimert halveringstid = 138.45
Faktisk halveringstid = 153.12
```

I koden over (om du utvider koden over grafen) ser du hva som skal til for å lage en enkel regresjonsmodell basert på dataene våre. Her er det i utgangspunktet 3 linjer som skiller seg ut og som jeg skal forklare litt rundt.

```
p0 = (130, 1000)
```

I koden over bestemmer vi noen verdier som skal være grove forslag til verdiene vi antar regresjonsmodellen skal ha. I vårt tilfelle har vi funksjonen *halvering* som trenger to ukjente verdier: *halveringstid* og *start_aktivitet*. Disse to verdiene er det vi nå gjetter på. Som nevnt holder det med et veldig grovt estimat, så over gjetter jeg på halveringstiden 130 sekunder og startverdi på 1000. Dette er jo langt unna noe vi får i modellen, men det er kun valgt for å illustrere at dette estimatet kan være veldig grovt og likevel fungere.

```
parametre, cov = curve_fit(halvering, tid, aktivitet, p0)
```

I den neste linjen skjer det meste av "magien". Her bruker vi *curve_fit*-funksjonen fra *scipy*-biblioteket til å estimere en god modell for dataene våre. Vi deler opp resultatene av denne funksjonen i to nye variabler. Den første variabelen kalt *parametre* vil inneholde alle verdiene til regresjonsmodellen, mens den andre variabelen *cov* inneholder data som kan fortelle oss noe om nøyaktigheten til modellen (dette går jeg ikke mer inn på her, men det kan leses mer om i kildene over).

curve_fit-funksjonen trenger selve funksjonen vi ønsker at den skal finne verdier til, x-verdiene og y-verdiene som denne funksjonen skal tilpasses til og til slutt p0 som er vårt estimat for de riktige verdiene.

```
halveringstid_est, start_aktivitet_est = parametre
```

I denne siste linjen deler vi opp variabelen *parametra* inn i de respektive variablene som skal fungere som input i den nye regresjonsmodellen vår. Hadde vi hatt flere inputverdier i funksjonen måtte vi også hatt flere variabler i denne linjen.

Om vi sammenlikner resultatene fra regresjonsmodellen ser vi at den ikke treffer særlig bra. Hadde vi gjennomført regresjonen i geogebra ville modellen vært mye bedre, så hva kan vi gjøre for å få denne enda bedre til?

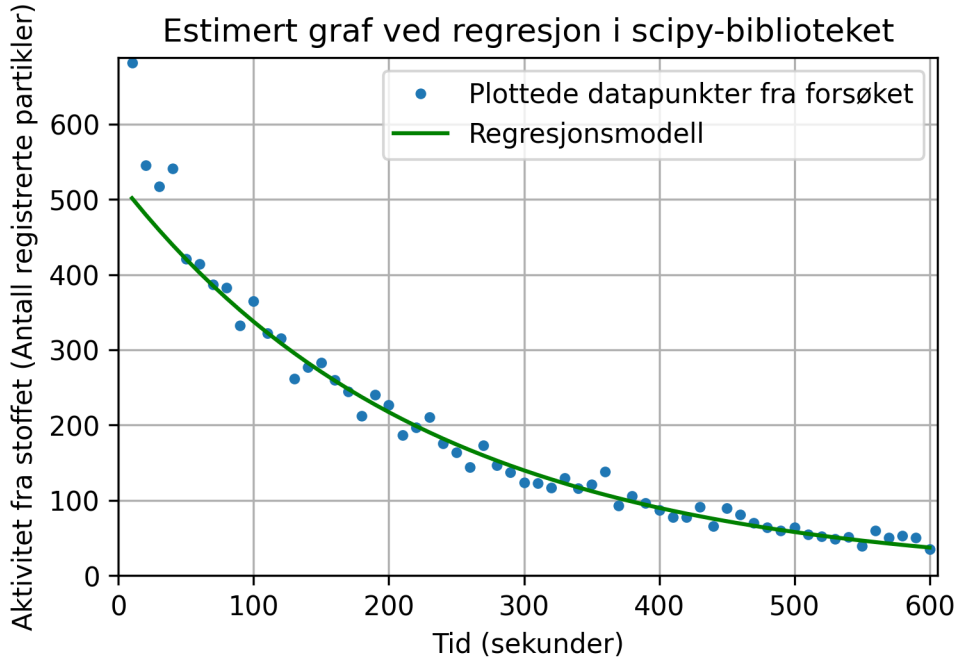
Her kan du utforske litt selv om du ønsker, men det enkleste for å få modellen til å passe bedre med de fleste av punktene er å "kutte bort" de første 4 registrerte punktene fra modellen. Dette kan vi enkelt gjøre ved å endre inputen til *curve_fit*-funksjonen slik:

```
parametre, cov = curve_fit(halvering, tid[4:], aktivitet[4:], p0)
```

Om vi endrer denne linjen med kode, men beholder alt slik det var i forrige eksempel ender vi opp med følgende modell:

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


def halvering(tid, halveringstid, start_aktivitet):
    return start_aktivitet * 0.5 ** (tid / halveringstid)


p0 = (130, 1000)
parametre, cov = curve_fit(halvering, tid[4:], aktivitet[4:], p0)
halveringstid_est, start_aktivitet_est = parametre

plt.title("Estimert graf ved regresjon i scipy-biblioteket")
plt.plot(
    tid, aktivitet, ".", label="Plottede datapunkter fra forsøket"
)  # Plotter datapunktene
plt.plot(
    tid,
    halvering(tid, halveringstid_est, start_aktivitet_est),
    "-",
    color="green",
    label="Regresjonsmodell",
)
plt.legend()
plt.ylim(0, aktivitet[0] * 1.01)
plt.xlim(0, tid[-1] * 1.01)
plt.xlabel(f"Tid (sekunder)")
plt.ylabel("Aktivitet fra stoffet (Antall registrerte partikler)")
plt.grid(True)
plt.show()
```

```
print(f"Estimert halveringstid = {halveringstid_est:.2f}")
print(f"Faktisk halveringstid = {halveringstid}")
```




```
Estimert halveringstid = 157.36
Faktisk halveringstid = 153.12
```

Nå ser jo den estimerte verdien mye bedre ut når man sammenlikner med faktisk halveringstid, men hvordan vet vi om denne modellen er god nok? Det er mulig å jobbe videre med *scipy*-biblioteket her, men det gjør vi ikke her. Det enkleste vi kan gjøre når vi vil sammenlikne en graf er å bruke Geogebra som et støtteverktøy. Om vi sammenlikner med en regresjonsgraf i Geogebra, og ser på hvordan denne er sammenliknet med kurven vi fikk gir det oss en indikasjon på om vi hat fått det greit til.

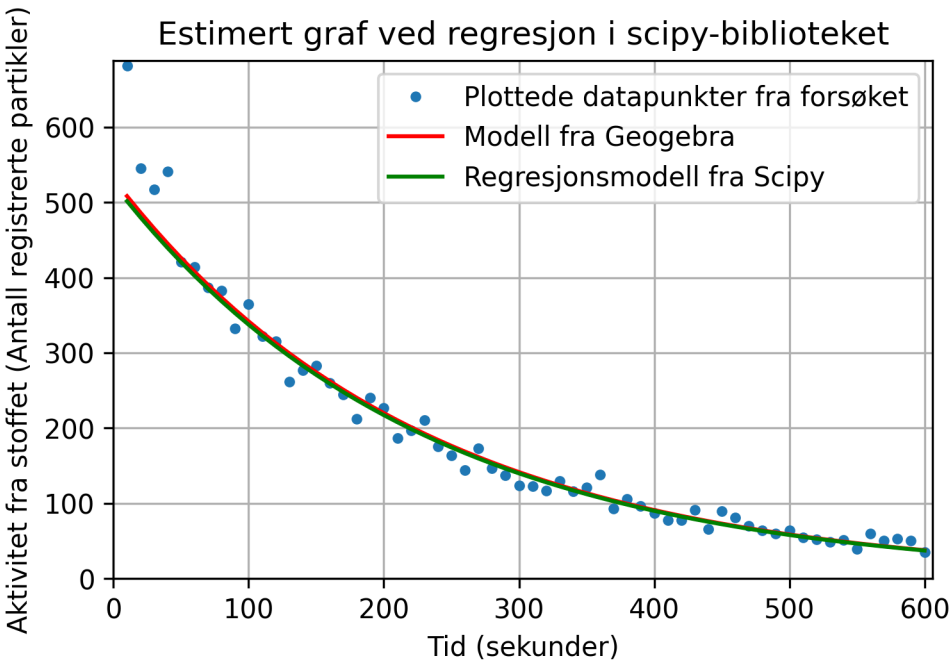
Når jeg selv la inn alle punktene i Geogebra og gjennomførte regresjon, fikk jeg følgende eksponentielle modell for dataene så denne modellen slik ut:

$$f(x) = 531.339 \cdot 0.9956^x$$

Under har jeg tegnet grafen til denne modellen inn sammen med modellen fra *scipy*.

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


def halvering(tid, halveringstid, start_aktivitet):
    return start_aktivitet * 0.5 ** (tid / halveringstid)
```

```
def regresjon_ggb(tid, a, b):
    return a * b**tid


p0 = (130, 1000)
parametre, cov = curve_fit(halvering, tid[4:], aktivitet[4:], p0)
halveringstid_est, start_aktivitet_est = parametre

plt.title("Estimert graf ved regresjon i scipy-biblioteket")
plt.plot(
    tid, aktivitet, ".", label="Plottede datapunkter fra forsøket"
)  # Plotter datapunktene
plt.plot(
    tid,
    regresjon_ggb(tid, 531.339, 0.9956),
    "-",
    color="red",
    label="Modell fra Geogebra",
)
plt.plot(
    tid,
    halvering(tid, halveringstid_est, start_aktivitet_est),
    "-",
    color="green",
    label="Regresjonsmodell fra Scipy",
)
plt.legend()
plt.ylim(0, aktivitet[0] * 1.01)
plt.xlim(0, tid[-1] * 1.01)
plt.xlabel(f"Tid (sekunder)")
plt.ylabel("Aktivitet fra stoffet (Antall registrerte partikler)")
plt.grid(True)
plt.show()
```


Estimert graf ved regresjon i scipy-biblioteket
Plottede datapunkter fra forsøket
Modell fra Geogebra
Regresjonsmodell fra Scipy
Aktivitet fra stoffet (Antall registrerte partikler)
Tid (sekunder)
0
100
200
300
400
500
600