Repetisjon/oppgaver i BED-1304 Python-Lab

Markus J. Aase

Forelesning 9 - Simulering

Informasjon

Til nå har vi gått igjennom flere ting i Python-lab. Det inkluderer temaer som:

- Python basics
- Funksjoner (Innebygde funksjoner og de vi definerer selv)
- Lister, Dictionary og Tuples
- Logikk og løkker (`if/else`-setninger, `for`-løkker og `while`-løkker).
- Pakker som `Pandas`, `NumPy`, `Matplotlib` og `SymPy`.

Simulering er det siste temaet vi skal gjennomgå i BED-1304, og krever at vi har kontroll på de tidligere temaene gjennomgått i kurset.

Dette dokumentet er lagd for å repetere og teste forståelsen av kjernepensum i **BED-1304 Python-Lab**. Husk at dette er et supplement til forelesning og seminar.

God koding!

Introduksjon til simulering

Hva er simulering?

Simulering er en metode for å modellere og analysere komplekse prosesser ved å etterligne dem i datamaskinen. I stedet for å løse matematiske modeller eksakt, kan vi bruke simulering til å utforske hvordan et system oppfører seg under ulike betingelser.

Med simulering kan vi blant annet:

- Generere tilfeldige tall for å modellere usikkerhet.
- Bruke simulering for å estimere sannsynligheter og forventede utfall.
- Undersøke økonomiske prosesser, for eksempel prisutvikling over tid.

Dette gir oss et kraftig verktøy for å forstå systemer som er for kompliserte til å løses eksakt med penn, papir eller algebra. Gjennom simulering kombinerer vi programmeringsverktøy som `for`-løkker, `while`-løkker, lister, `dictionaries`, samt biblioteker som `NumPy` og `pandas`.

Eksempel: Simulering av terningkast

Et klassisk, og relativt enkelt eksempel i simulering er å bruke terningkast. I matematikk og statistikk bruker vi ofte terninger for å forstå prosesser, men terninger kan faktisk lære oss ganske mye vi kan ha nytte av i den virkelige verden også!

Anta at du vil gjøre N kast med terningen, hvor $N = 5$, $N = 100$ og $N = 10\,000$. Da kan du kanskje lure på, hva blir gjennomsnittsverdien når du kaster terningen ulike antall ganger.

Vi starter med å skrive en pseudokode.

Pseudokode

```
# Pseudokode for terningkast-simulering
1. Velg hvor mange kast vi skal gjøre (N)
2. For hvert kast:
   a. Generer et tilfeldig tall mellom 1 og 6
   b. Lagre utfallet
3. Regn ut gjennomsnittet av alle kastene
4. Returner gjennomsnittet
```

Python-kode

```python
import random

def simuler_terningkast(N):
    kast = []
    for i in range(N):
        verdi = random.randint(1, 6)  # tilfeldig tall mellom 1 og 6
        kast.append(verdi)
    gjennomsnitt = sum(kast) / len(kast)
    return gjennomsnitt

# Test for ulike antall kast
print("5 kast:", simuler_terningkast(5))
print("100 kast:", simuler_terningkast(100))
print("10 000 kast:", simuler_terningkast(10000))
```

Observasjoner

- Her vil vi få ulike svar hver gang, fordi vi har det elementet `random.randint(a, b)` en funksjon som gir et tilbake et tilfeldig tall mellom a og b, i dette tilfelle 1 og 6.
- Når vi bare kaster 5 ganger, kan gjennomsnittet variere mye (stor usikkerhet).
- Med 100 kast blir gjennomsnittet mer stabilt.
- Med 10 000 kast nærmer gjennomsnittet seg den teoretiske verdien 3.5.
- Legg merke til at vi **kan** bruke `numpy` sin random funksjon: `np.random.randint(1, 6+1)`, men da må vi ha +1 fordi `numpy` versjonen **ikke** er inklusiv det siste tallet man gir funksjonen. Bruker vi `random.randint(1, 6)`, trenger vi **ikke** +1 fordi `random` versjonen **er** inklusiv, og tar derfor med tallet 6.

La oss visualisere det vi har gjort ved hjelp av et histogram i matplotlib.

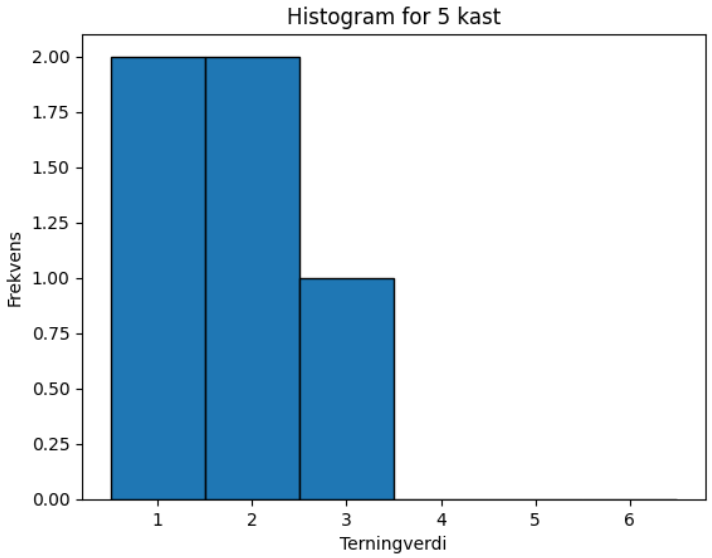
Python-kode

```
import matplotlib.pyplot as plt

# Vi bruker funksjonen definert over
kast_5 = simuler_terningkast(5)

# Plot 5 kast
plt.hist(kast_5, bins=range(1,8), edgecolor="black", align="left")
plt.xticks(range(1,7))
plt.title("Histogram for 5 kast")
plt.xlabel("Terningverdi")
plt.ylabel("Frekvens")
plt.show()
```

Da får vi noe som dette ut.



Figur 1: Plott for histogram av 5 kast.

Her ser vi at vi ikke nødvendigvis vil få alle mulige terningverdier. For det kan fort være at etter 5 kast, så har vi aldri fått en sekser.

OBS: Hvis du prøver å kjøre koden over selv, vil du ikke nødvnedigvis få samme plott - det kommer av funksjonen `simuler_terningkast(N)` har en *tilfeldig* komponent, altså `verdi = random.randint(1, 6)` - som tilfeldig gir ut et tall mellom 1 og 6.

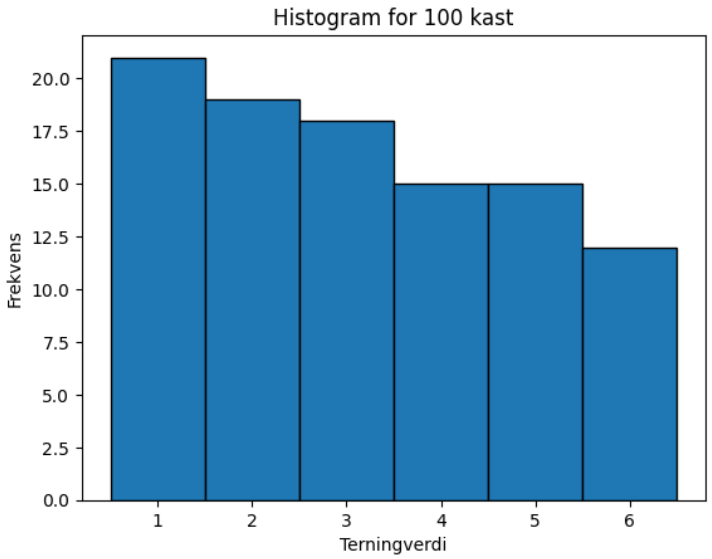
Derfor går vi videre å plotter histogram for 100 terningkast, og ser hva vi får da.

```
import matplotlib.pyplot as plt

kast_100 = simuler_terningkast(100)

# Plot 100 kast
plt.hist(kast_100, bins=range(1,8), edgecolor="black", align="left")
plt.xticks(range(1,7))
plt.title("Histogram for 100 kast")
plt.xlabel("Terningverdi")
plt.ylabel("Frekvens")
plt.show()
```

Da får vi noe som dette ut.



Figur 2: Plott for histogram av 100 kast.

Nå ser vi at frekvensen ser litt annerledes ut, vi har fått over alle de ulike verdiene mer enn 10 ganger. Men vi ser at vi har fått 1 og 2 flest ganger. Dette skyldes også tilfeldigheter, som følge av funksjonen vi har brukt.

I dette scenarioet hadde vi kanskje gjort det dårlig i terningspillet Yatzy – eller kanskje vi likevel hadde vært heldige og fått liten eller stor straight?

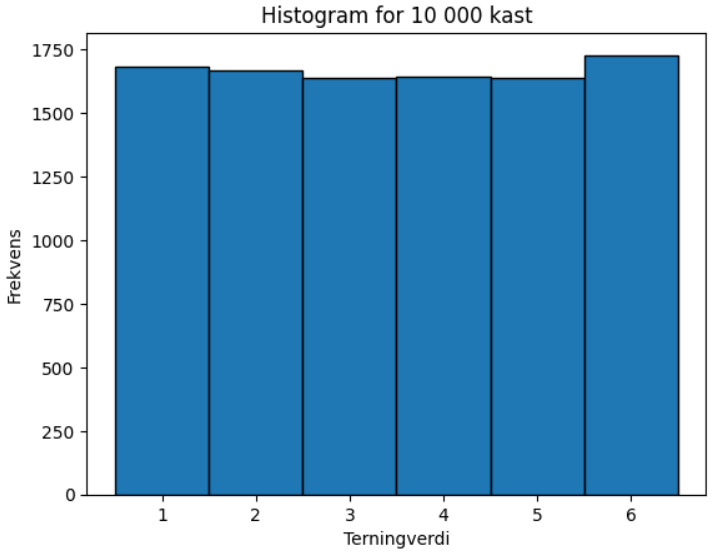
Hvordan ser histogrammet ut hvis vi øker antall kast til 10 000? La oss sjekke det ut.

```
import matplotlib.pyplot as plt

kast_10000 = simuler_terningkast(10000)

# Plot 10 000 kast
plt.hist(kast_10000, bins=range(1,8), edgecolor="black", align="left")
plt.xticks(range(1,7))
plt.title("Histogram for 10000 kast")
plt.xlabel("Terningverdi")
plt.ylabel("Frekvens")
plt.show()
```

Da får vi ut noe som dette.



Figur 3: Plott for histogram av 10000 kast.

Da ser vi at vi (cirka) får alle terningverdiene like ofte (altså, ved samme frekvens). Dette er et interessant perspektiv fordi det viser **loven om store tall**, i praksis. Når vi kaster terningen få ganger, vil utfallet variere mye, og noen tall kan dukke opp flere ganger enn andre. Men når vi kaster terningen mange ganger (her 10 000 kast), vil frekvensen av hver terningverdi bli ganske lik. Sagt med andre ord, da nærmer frekvensen av hver terningverdi seg den teoretiske sannsynligheten på 1/6, nemlig at det er *en sjettedels sjanse* for å få hvilken som helst verdi.

- Dette illustrerer hvordan tilfeldige prosesser *jevner seg ut* når vi har mange observasjoner.
- Monte Carlo-simuleringer bygger nettopp på denne ideen: ved å generere mange tilfeldige scenarioer kan vi estimere forventede verdier og sannsynligheter.
- Det gir oss også en intuitiv forståelse av hvorfor det er nyttig å simulere økonomiske prosesser, spill og andre systemer som er for komplekse til å løses eksakt.

Med andre ord, selv om enkeltkast er helt tilfeldige, får vi med et stort antall kast et *stabilt og forutsigbart mønster* – noe som er kjernen i simulering og stokastisk analyse.

Oppgaver

Flervalgsoppgaver – Simulering i Python

1. Hva menes med simulering i denne sammenhengen?

   (a) Å tegne grafer over funksjoner
   (b) Å etterligne virkelige prosesser på datamaskinen ved hjelp av tilfeldige tall
   (c) Å beregne eksakte matematiske løsninger
   (d) Å løse ligninger symbolsk med SymPy

2. Hvilken Python-funksjon brukes i eksemplet til å generere tilfeldige tall mellom 1 og 6?

   (a) `np.random.choice(6)`
   (b) `random.random()`
   (c) `random.randint(1, 6)`
   (d) `random.uniform(1, 6)`

3. Hva skjer når vi øker antall terningkast fra 5 til 10 000?

   (a) Resultatene blir mer tilfeldige
   (b) Vi får færre ulike verdier
   (c) Histogrammet blir mer skjevt
   (d) Frekvensene jevner seg ut og nærmer seg sannsynligheten 1/6 for hver verdi

4. Hva er den teoretiske forventningsverdien (gjennomsnittet) for en rettferdig seks-sidet terning?

   (a) 3
   (b) 3.5
   (c) 4
   (d) 6

5. Hvorfor får vi ikke alltid samme resultat når vi kjører koden på nytt?

   (a) Fordi koden er feil
   (b) Fordi Python lagrer tilfeldige tall midlertidig
   (c) Fordi funksjonen `random.randint()` genererer tilfeldige tall
   (d) Fordi løkka ikke itererer riktig

6. Hva illustrerer loven om store tall"?

   (a) At tilfeldige tall blir mer uforutsigbare jo flere vi genererer
   (b) At summen av tilfeldige tall alltid er konstant
   (c) At gjennomsnittet av mange tilfeldige utfall nærmer seg forventningsverdien
   (d) At sannsynligheten for alle utfall øker med flere forsøk

7. Hva viser histogrammet med 10 000 kast best?

   (a) At 6 kommer sjeldnere enn 1
   (b) At terningen er urettferdig

(c) At alle verdier forekommer omtrent like ofte

(d) At tilfeldigheter forsvinner helt

8. Hva er hovedforskjellen mellom `random.randint(1,6)` og `np.random.randint(1,6)`?

   (a) Ingen forskjell – de gir alltid samme resultat

   (b) Begge ekskluderer 6

   (c) `random.randint()` gir flyttall, mens `np.random.randint()` gir heltall

   (d) `random.randint(1,6)` inkluderer 6, mens `np.random.randint(1,6)` ekskluderer 6

9. Hva er en fordel med simulering sammenlignet med eksakte beregninger?

   (a) Den er alltid mer presis

   (b) Den kan modellere komplekse prosesser som ikke har enkle matematiske løsninger

   (c) Den krever ingen programmering

   (d) Den fjerner tilfeldigheter

10. Hvordan henger Monte Carlo-metoden sammen med terningkast-eksemplet?

    (a) Den brukes bare til spill

    (b) Den beregner kun sannsynligheter for 6-ere

    (c) Den bygger på idéen om å bruke mange tilfeldige utfall for å estimere et forventet resultat

    (d) Den eliminerer behovet for tilfeldige tall

Forklaring av kode

I denne oppgaven skal du lese og forklare hva følgende to kodeeksempler gjør. Bruk egne ord til å beskrive formålet med koden, hvordan den fungerer steg for steg, og hva vi forventer som resultat.

(a) Simulering av myntkast

```python
import random

def simuler_myntkast(N):
    resultat = []
    for i in range(N):
        kast = random.choice(["Kron", "Mynt"])
        resultat.append(kast)
    sannsynlighet_kron = resultat.count("Kron") / N
    return sannsynlighet_kron

print(simuler_myntkast(1000))

```

Forklar hva koden gjør, og hva tallet som skrives ut representerer. Hva forventer du at resultatet blir dersom du øker antall kast fra 10 til 10 000?

(b) Simulering av køsystem

```python
import random

def simuler_kø(antall_kunder):
    ventetider = []
    for i in range(antall_kunder):
        service_tid = random.uniform(2, 8)  # minutter per kunde
        ventetider.append(service_tid)
    gjennomsnitt = sum(ventetider) / len(ventetider)
    return gjennomsnitt

print("Gjennomsnittlig ventetid:", simuler_kø(500))

```

Forklar hva koden simulerer og hvordan tilfeldighetene brukes her. Hva forteller resultatet, og hvordan kan vi bruke simulering til å forbedre køsystemet?

Skrive kode selv - En utfordring

Nå skal du prøve å kombinere flere av de temaene vi har jobbet med i kurset. Du skal skrive din **egen simulering**, som bruker tilfeldige tall til å utforske et økonomisk fenomen.

Oppgave: Simulering av aksjepris over tid

Anta at en aksje starter med en pris på 100 kroner. Hver dag kan prisen enten *stige* eller *synke* med en viss prosentandel. Vi antar at den daglige prosentvise endringen er tilfeldig og følger en normalfordeling med forventning 0 og standardavvik 1%.

- Bruk `numpy` til å generere tilfeldige prosentvise endringer for et gitt antall dager (for eksempel 252 dager, som tilsvarer ett år med børsdager).
- Simuler prisutviklingen dag for dag, og lagre resultatene i en liste eller NumPy-array.
- Visualiser prisutviklingen ved hjelp av `matplotlib`.

Utvidelse: Kjør simuleringen for flere ulike scenarioer, for eksempel 10 ulike aksjer eller 10 ulike mulige utfall av samme aksje. Plott alle på samme figur, slik at du kan sammenligne hvordan tilfeldighetene påvirker utviklingen.

Tips:

- Bruk `np.random.normal(0, 0.01, antall_dager)` for å generere prosentvise endringer.
- En enkel måte å beregne ny pris på er:

$$P_{t+1} = P_t \times (1 + r_t)$$

der r_t er den prosentvise endringen dag t.

- Du kan bruke løkke (`for`-løkke) eller vektoroperasjoner i `NumPy`.

Refleksjonsspørsmål

- Hva skjer med variasjonen i prisene når du øker antall dager i simuleringen?
- Hva representerer de ulike prisbanene du ser i plottet?
- Hvordan kan en slik simulering være nyttig for økonomer eller investorer?

Ekstra utfordring: Legg til en beregning som estimerer forventet avkastning og risiko (standardavvik) basert på simuleringen din. Hvordan endres resultatene dersom du øker eller reduserer standardavviket i endringsraten?

Løsningsforslag

Løsningsforslag – Multiple Choice

1. b) Å etterligne virkelige prosesser på datamaskinen ved hjelp av tilfeldige tall
2. c) `random.randint(1, 6)`
3. d) Frekvensene jevner seg ut og nærmer seg sannsynligheten 1/6 for hver verdi
4. b) 3.5
5. c) Fordi funksjonen `random.randint()` genererer tilfeldige tall
6. c) At gjennomsnittet av mange tilfeldige utfall nærmer seg forventningsverdien
7. c) At alle verdier forekommer omtrent like ofte
8. d) `random.randint(1,6)` inkluderer 6, mens `np.random.randint(1,6)` ekskluderer 6
9. b) Den kan modellere komplekse prosesser som ikke har enkle matematiske løsninger
10. c) Den bygger på idéen om å bruke mange tilfeldige utfall for å estimere et forventet resultat

Løsningsforslag: Forklaring av kode

(a) Simulering av myntkast

Koden definerer en funksjon `simuler_myntkast(N)` som simulerer N myntkast. For hvert kast velges tilfeldig mellom 'Kron' og 'Mynt' ved hjelp av `random.choice()`. Deretter beregnes andelen av kastene som ble 'Kron' ved å telle antall forekomster og dele på totalt antall kast.

- Resultatet som skrives ut er en verdi mellom 0 og 1 – sannsynligheten (estimatet) for å få Kron".
- For små verdier av N (f.eks. 10 kast) kan andelen variere mye, men for store N (f.eks. 10 000 kast) vil resultatet nærme seg 0.5.
- Dette illustrerer loven om store tall – gjennomsnittet stabiliserer seg når antallet forsøk øker.

(b) Simulering av køsystem

Koden simulerer ventetiden til et gitt antall kunder i en kø. For hver kunde genereres en tilfeldig *servicetid* mellom 2 og 8 minutter med `random.uniform(2, 8)`. Disse tidene lagres i listen `ventetider`, og gjennomsnittlig servicetid beregnes ved å ta summen delt på antall kunder.

- Resultatet som skrives ut er den gjennomsnittlige ventetiden per kunde.
- Ved å øke `antall_kunder`, får vi et mer stabilt estimat for gjennomsnittlig ventetid.
- Dette er et eksempel på en enkel Monte Carlo-simulering som kan brukes til å modellere og forbedre køsystemer, for eksempel ved å vurdere effekten av kortere eller lengre servicetider.

Løsningsforslag: Skrive kode selv - En utfordring

```python
# Løsningsforslag: Simulering av aksjeprisbaner (vektorisert)
import numpy as np
import matplotlib.pyplot as plt

# Parametre
start_pris = 100.0      # startpris i kroner
antall_dager = 252      # handelsdager i ett år
antall_simuleringer = 50  # hvor mange prisbaner vi simulerer
sigma = 0.01            # daglig standardavvik (1%)
mu = 0.0                # daglig forventet endring (0%)
seed = 42               # for replikerbarhet (kan fjernes)
np.random.seed(seed)

### Her er det mange løsninger ###
### Skriv din løsning under    ###
```