Pythonprogrammering med Tekna

Tilfeldige dartkast, π estimering og numerisk integrasjon

kodeskolen simula

kodeskolen@simula.no

I dette kompendiet skal vi estimere π ved å kaste tilfeldige dart mot en blink. Deretter ser vi hvordan vi kan generalisere denne koden til å estimere verdien til et integral. Spesifikt, skal vi bruke Monte-Carlo integrasjon som er en av de mest intuitive formene for numerisk integrasjon og som er en grunnstein i en rekke fagfelt, fra statistikk til økonomi og beregningsorientert kjemi og medisin.

Innhold

1 Monte Carlo: Krig, kasino og integrasjon

Monte Carlo metoden er kanskje en av de nyttigste metodene innen anvendt matematikk. Den lar oss løse integral og differensiallikninger som vi ellers ikke kunne tenke oss, og i dag finner vi denne metoden overalt rundt oss.

Men først, hvorfor er en metode, eller *algoritme*, som brukes så mye kalt opp etter et kasino i Monaco? Monte Carlo metoden startet som en amerikansk militærhemmelighet og ble funnet på for å lage atomvåpen på 40-tallet. Forskerne som arbeidet med Manhatten prosjektet hadde nemlig noen integral de måtte løse for å finne ut hvordan nøytron sprer seg i kjærnefysisk materiale. Disse integralene var så vanskelige å løse at selv de smarteste hodene i verden ikke klarte å løse dem!

Løsningen på problemet var ikke åpenbart på den tiden, men en av forskerne på Los Alamos National Laboratory kom på løsningen mens han var syk og la kabal. Mens Stanisław Ulam la kabal funderte han: *Hvor stor er egentlig sannsynligheten for at denne kabalen går opp?*. Ulam satte seg ned og fant frem matematikken, men uansett hva han prøvde kom han ingen vei. Så fikk han en idè *kan jeg ikke bare prøve mange ganger og telle hvor ofte den går opp?*. Det var denne ideen som også skulle bli løsningen på nøytronspredingsproblemet.

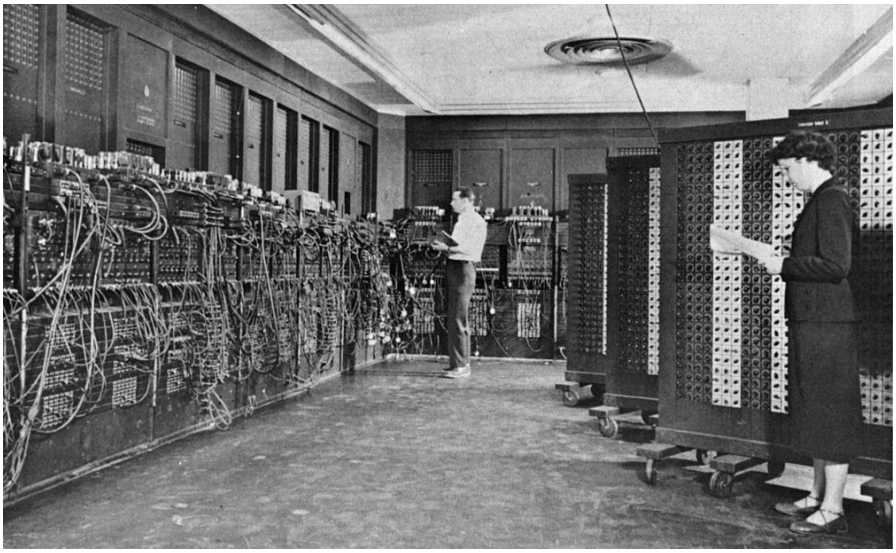
Ikke lenge etter at Ulam la denne kabalen så fortalte han om ideen til sjefen sin, John von Neumann, en av historiens skarpeste matematikere[1]. De diskuterte hvordan de kunne simulere ulike måter et enkelt nøytron beveger seg, og når man ikke viste helt hva det skulle gjøre seg så valgte man det tilfeldig. Hvis vi gjentar det mange ganger for enkeltnøytron, får vi til slutt vite hvordan en stor samling med nøytron sprer seg.

I dag virker kanskje ikke denne metoden veldig nyskapende. På den tiden derimot, datamaskiner var akkurat funnet på og det å løse avanserte matematiske problem med tilfeldige simuleringer var en bissar idè. Tenk på kabal-eksempelet; orker du legge en kabal tusenvis av ganger for å se sannsynligheten for å vinne? Det gjorde ikke matematikerne på den tiden heller. Det som var revolusjonerende med Ulam sin idè var å bruke datamaskiner for å gjenta forsøket mange ganger.

Ulam og von Neumann samarbeidet om prosjektet og fikk programmert verdens første universelle datamaskin[2], ENIAC. Å programmere denne var ikke helt enkelt, man måtte nemlig tegne kretstegninger på papir og be programmererne ved

[1] På Wikipedia kan dere lese om von Neumann https://en.wikipedia.org/wiki/John_von_Neumann

[2] Med universell datamaskin mener vi en Turing komplett datamaskin.



Figur 1: To programmerere som arbeider med ENIAC datamaskinen. (Bilde av amerikanske hæren, offentlig domene)

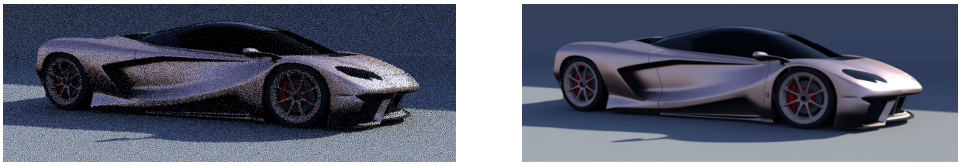
universitetet i Pennsylvania om å manuelt trekke ledninger og skru på brytere. Nedenfor kan vi se et bilde av to programmerere ved universitetet og en del av et dataprogram som ble kjørt på ENIAC datamaskinen.

Siden Monte Carlo metoden ble brukt til å forske på atomvåpen var alt rundt den selvfølgelig hemmelig. Metoden trengte et kodeord, slik at de kunne snakke om den uten at de rundt dem forsto hva de pratet om. Etter å prate med en kollega, Nicholas Metropolis, så foreslo han navnet Monte Carlo, et verdenskjent kasino i Monaco som Metropolis' onkel alltid lånte penger for å spille på. Navnet passet bra — kasino handler tross alt om tilfeldighet, og kodeordet forble i bruk selv etter at metoden ble offentliggjort.

I dag ser vi Monte Carlo metoden overalt. I stråleterapi brukes det for å regne ut hvordan en pasient bør bestråles for å minimere stråling til friskt vev. Vi vet ikke nøyaktig hvordan en pasienter kommer til å bevege seg, så vi simulerer bestråling mange ganger med forskjellig bevegelse, og ser hvor mye stråling som gis til friskt vev og hvor mye som gis til kreftsvulsten.

Monte Carlo metoden er også kommet til dataspill. I 2019 lanserte Nvidia nye grafikkort som kan simulere hvordan lys sprer seg i et landskap, såkalt *ray tracing*. For å simulere dette må man løse mange integral for hver eneste pixel på skjermen[3], og hvis vi skulle løst disse integralene nøyaktig hadde dataspillet aldri kunnet kjørt

[3]Spesifikt så må vi integrere over vinklene som lyset kommer fra



Figur 2: Her ser vi to simuleringer av lys som treffer en bil. Begge simuleringene brukte Monte Carlo metoden for å løse integral, men den til venstre fikk bare simulert noen få lysstråler per integral og det til høyre fikk simulert mange lysstråler per integral. (Modeller av Yasutoshi Mori, CC-BY lisens)

i sanntid. Derfor brukes heller Monte Carlo metoden for å gi et raskt estimat av hvordan lyset sprer seg, også blir resultatet av denne simuleringen kjørt igjennom en støyfjerningsalgoritme for å lage et fint bilde. I figur 2 kan vi se et eksempel om hvordan kvaliteten på en simulering endres avhengig av hvor mange tilfeldige forsøk man bruker for å tilnærme integralet.

2 Å estimere π

Vi vet at tallet π er omtrentlig lik 3.14, men, hvordan kom vi egentlig frem til det? Vel, det er mange måter vi kan komme frem til veriden av π på, og i dette kompendiet skal vi se hvordan vi kan finne fram til det ved å kaste dart!

2.1 Finne et uttrykk for π

I denne delen av kompendiet skal vi se hvorfor det fungerer å finne π på denne måten.

Vi begynner med å finne et uttrykk for π: Vi vet at arealet (A_{sirkel}) til en sirkel er lik

$$A_{\text{sirkel}} = \pi r^2, \tag{1}$$

hvor r er radiusen til sirkelen. Det betyr at hvis vi vet at sirkelen har radius lik 1, så får vi

$$A_{\text{sirkel}} = \pi 1^2 = \pi. \tag{2}$$

Altså er pi det samme som arealet til en enhetssirkel! For å finne π trenger vi derfor kun finne arealet til en sirkel med radius lik 1.

Det neste spørsmålet vi må stille oss nå er: hvordan kan vi finne arealet til en sirkel? Dette gjør vi ved å kaste dart.

La oss se hvordan. Se for deg at vi har en sirkulær blink med radius lik 1 og en firkant som akkurat når ut til kanten av blinken, litt som i figur 3b. Siden diameteren til sirkelen er lik to (dobbelt så lang som radiusen), så må sidelengdene på dette kvadratet også være lik 2. Altså blir arealet til kvadratet lik 4.

Neste steg er å kaste blink et tilfeldig sted på denne blinken. Hva er da sannsynligheten for å treffe inne i sirkelen, altså, hva er

$$P(\text{treffe sirkel})? \tag{3}$$

Vel, vi at sannsynligheten for at noe stemmer er antall gunstige utfall delt på antall mulige utfall, i dette tilfellet blir det

$$P(\text{treffe sirkel}) = \frac{\text{gunstig areal}}{\text{mulig areal}}. \tag{4}$$

Det område som vi kan treffe er kvadratet, det mulige arealet er arealet til kvadratet (A_{kvadrat} og det område vi ønsker å treffe er inni sirkelen, så det gunstige arealet er arealet til sirkelen (A_{sirkel}). Dette gir følgende likning

$$P(\text{treffe sirkel}) = \frac{A_{\text{sirkel}}}{A_{\text{kvadrat}}}, \tag{5}$$

og siden vi vet at arealet til sirkelen er lik π og arealet til kvadratet er lik 4, får vi denne sammenhengen:

$$P(\text{treffe sirkel}) = \frac{\pi}{4}. \tag{6}$$

Vi vet altså at sannsynligheten for å treffe inne i sirkelen er lik $\frac{\pi}{4}$, så hvis vi klarer å måle sannsynligheten for å treffe inne i sirkelen får kan vi løse likningen med hensyn på π og få

$$\pi = 4P(\text{treffe sirkel}). \tag{7}$$

Men hvordan kan vi egentlig finne denne sannsynligheten? Vel, vi vet at sannsynligheten er gitt ved

$$P(\text{treffe sirkel}) = \frac{\text{antall gunstige}}{\text{antall mulige}}, \tag{8}$$

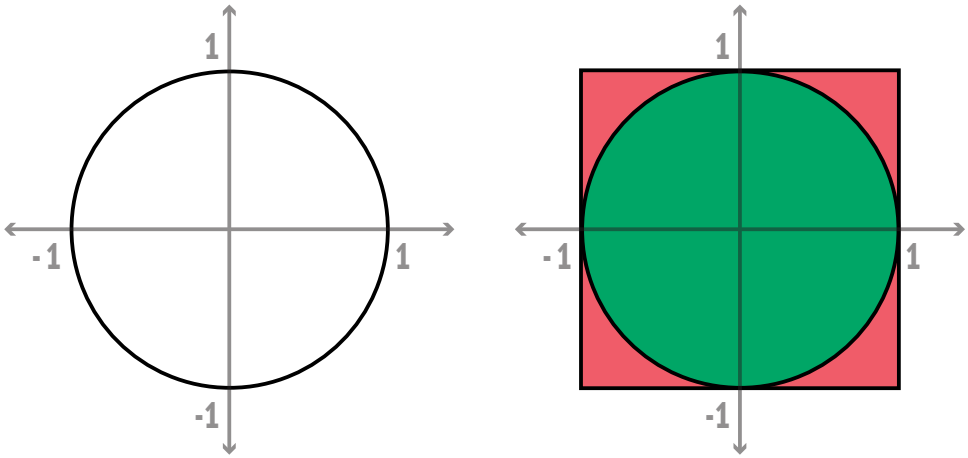
så hvis vi kaster dart på blinken veldig mange ganger og teller opp antall treff inni sirkelen og antall treff inni firkanten kan vi estimere hva sannsynligheten er! Da får vi denne likningen

$$P(\text{treffe sirkel}) \approx \frac{\text{antall darts inne i sirkelen}}{\text{antall darts inne i kvadratet}}, \tag{9}$$

så da er det bare å lage en blink og finne frem dartpilene våre!

2.2 Frakoblet Monte Carlo

Som nevnt tidligere, så ble slike Monte Carlo metoder populært som følge av at datamaskiner kan gjennomføre mange simuleringer utrolig raskt. Men det forsøket vi har beskrevet her er faktisk fult mulig å gjennomføre uten en datamaskin. Du kan foreksempel kaste en fysisk dart mot en blink så tilfeldig du klarer og deretter telle antall treff. Et litt enklere alternativ er å kaste tørkede erter inn i en kvadratformet boks med en sirkel tegnet opp på bunnen.



(a) Enhetssirkel **(b)** Enhetssirkel med omskrevet firkant

Figur 3: 3a viser en illustrasjon av enhetssirkelen med x- og y-aksen tegnet inn. 3b hviser enhetssirkelen sammen men den omskrevne firkanten. Grønn farge markerer det område som vi ser på som "treff" og rød farge markerer "bom".

For å se et eksempel på frakoblet Monte Carlo med dartkasting kan du se på den denne fine videoen: https://www.youtube.com/watch?v=M34TO71SKGk, hvor PhysicsGirl og Veritasium regner ut π slik vi har beskrevet det her. figur 4 viser et skjermbilde fra videoen.

Figur 4: PhysicsGirl og Veritasium regner ut π

3 Kaste dart med Python

Det blir fort slitsomt om vi skal kaste hundrevis av dart, så la oss derfor heller lage et dataprogram som gjør det for oss! Hvordan kan vi kaste tilfeldig dart på datamaskinen?

3.1 Kaste en dart

For hver dartpil trenger vi kun vite hvor denne pila traff i blinken, altså trenger vi en tilfeldig x- og y-koordinat. Siden blinken har sidelengde lik 2 og er sentrert i origo, trenger vi derfor en tilfeldig x-koordinat mellom -1 og 1 og en tilfeldig y-koordinat mellom -1 og 1.

For å få et tilfeldig koordinat mellom -1 og 1 kan vi bruke `uniform(-1, 1)` funksjonen som ligger i `pylab`-biblioteket. Denne funksjonen tar inn to tall og gir et tilfeldig desimaltall som ligger mellom de to tallene. La oss nå bruke denne funksjonen til å lage en ny funksjon, `kast_dart` som simulerer et dart-kast:

```
1  from pylab import uniform
2
3  def kast_dart():
4      x_koordinat = uniform(-1, 1)
5      y_koordinat = uniform(-1, 1)
6      return x_koordinat, y_koordinat
```

På første linje i dette programmet så importerer vi `uniform` fra `pylab`-bibliteket biblioteket. Deretter definerer vi `kast_dart` funksjonen vår, som simulerer et dart-kast som treffer kvadratet vårt. Denne funksjonen tar ingen input, hver gang den blir kalt så henter den to tilfeldige tall, hvert mellom -1 og 1. Disse tallene representerer x- og y-koordinatene hvor dart-pila traff kvadratet, og blir returnert av funksjonen.

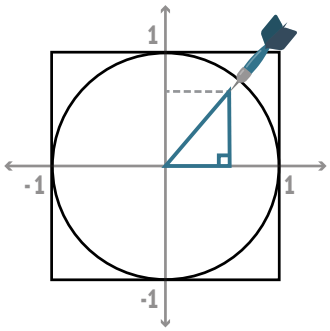
La oss teste dart-simuleringen ved å kalle på funksjonen og skrive ut koordinatene til terminalen.

```
9  dart_x, dart_y = kast_dart()
10 print(dart_x, dart_y)
```

```
-0.9288543323341905 0.7618020937850156
```

Hvis vi kjører programmet mange ganger, ser vi at koordinatene holder seg innenfor (-1, 1), men de blir forskjellige hver gang. Vi har altså klart å simulere tilfeldige dartkast!

3.2 Traff vi blinken?



Figur 5: Dartkast.

Neste steg i simuleringen vår blir å sjekke om vi traff blinken. Vi trenger altså å sjekke om vi er innenfor enhetssirkelen. For at et punkt skal være innenfor enhetssirkelen må avstanden mellom origo og punktet være mindre enn 1.

Vi ser av figur 5 at vi kan tegne opp en rettvinklet trekant slik at avstanden til origo er gitt ved hypotenusen til trekanten og x- og y-koordinatene til dartkastet tilsvarer første og andre katet. Da kan vi bruke Pytagoras-setning til å finne avstanden.

Pytagoras setning sier at:

$$\text{Hypotenus} = \sqrt{\text{Katet}_1^2 + \text{Katet}_2^2} \tag{10}$$

Dette kan vi skrive som en funksjon i Python:

```
9  def pytagoras(katet1, katet2):
10     return sqrt(katet1**2 + katet2**2)
```

NB: for at denne koden skal kjøre må du huske å importere `sqrt` i starten av programmet ditt. Dette kan du gjøre med følgende linje:

```
1  from math import sqrt
```

Nå kan vi bruke `pytagoras` funksjonen til å lage en betingelse for om darten traff inne i enhetssirkelen. Dette kan vi bruke til å lage en funksjon som sjekker om vi traff eller ikke:

```
12 def traff_blinken(dart_x, dart_y):
13     avstand_origo = pythagoras(dart_x, dart_y)
14     if avstand_origo < 1:
15         return True
16     else:
```

```
        return False
```

For å gjøre koden enda mer komprimert kan vi legge merke til at den kan skrives om slik:

```
def traff_blinken(dart_x, dart_y):
    avstand_origo = pytagoras(dart_x, dart_y)
    return avstand_origo < 1
```

Dette blir det samme fordi betingelsen `avstand_origo < 1` enten har verdien `True` eller `False` avhengig av om avstanden er mindre enn 1 eller ikke. Så hvis vi returnerer `avstand_origo < 1` får vi den oppførselen vi ønsker oss: en funksjon som returerer `True` dersom darten er innenfor sirkelen og `False` hvis ikke.

La oss teste funksjonen ved å kalle på den slik:

```
dart_x, dart_y = kast_dart()
print(dart_x, dart_y)
print(traff_blinken(dart_x, dart_y))
```

```
0.6194445857786388 -0.11045470776043587
True
```

Hvis du kjører koden mange ganger ser du at den av og til treffer innenfor og av og til utenfor blinken. Vi har altså et program som både kan simulere et dartkast og sjekke om det er innenfor eller utenfor blinken. I neste seksjon skal vi se på hvordan vi kan simulere mange dart-kast og bruke simuleringen vår til å estimere π.

Den fullstendige koden så langt finner du under. Prøv å gå igjennom hver linje og overbevis deg selv om at du forstår hva den gjør før du går videre:

```
from math import sqrt
from pylab import uniform

def kast_dart():
    x_koordinat = uniform(-1, 1)
    y_koordinat = uniform(-1, 1)
    return x_koordinat, y_koordinat

def pytagoras(katet1, katet2):
    return sqrt(katet1**2 + katet2**2)
```

```
11
12 def traff_blinken(dart_x, dart_y):
13     avstand_origo = pytagoras(dart_x, dart_y)
14     return avstand_origo < 1
15
16 dart_x, dart_y = kast_dart()
17 print(dart_x, dart_y)
18 print(traff_blinken(dart_x, dart_y))
```

```
0.5185238714452942 0.08294549466037027
True
```

4 Estimere π med dartkast

4.1 Estimere sannsynlighet for treff

Nå har vi et program som kan kaste en dart og avgjøre om den har truffet innenfor enhetssirkelen eller ikke. For å estimere sannsynligheten for treff trenger vi å gjøre mange slike kast og så telle opp antall treff. Begynn med å opprette et program som du kaller `pi_estimering.py`. I starten av dette programmet skal du lime inn funksjonene `kast_dart`, `pytagoras` og `traff_blinken(dart_x, dart_y)` som du lagde i avsnitt 3 (se side 9 til 12). Husk også å importere `sqrt` og `uniform`. Starten på programmet skal altså se slik ut:

```
1  from math import sqrt
2  from pylab import uniform
3
4  def kast_dart():
5      x_koordinat = uniform(-1, 1)
6      y_koordinat = uniform(-1, 1)
7      return x_koordinat, y_koordinat
8
9  def pytagoras(katet1, katet2):
10     return sqrt(katet1**2 + katet2**2)
11
12 def traff_blinken(dart_x, dart_y):
13     avstand_origo = pytagoras(dart_x, dart_y)
14     return avstand_origo < 1
```

Vi begynner med å lage en variabel, `antall_kast` som skal bestemme hvor mange kast vi ønsker å gjøre. Til og begynne med kan vi sette den til 10

```
16 antall_kast = 10
```

For å holde orden på hvor mange ganger vi har truffet trenger vi en *tellevariabel*, `antall_treff`. Før vi har kastet en enste dart er antall treff null:

```
17 antall_treff = 0
```

Vi ønsker å simulere flere kast. Vi skal altså gjenta samme handling flere ganger, hvilket betyr at vi bør bruke en `for`-løkke. Inne i løkka kan vi bruke koden du skrev i avsnitt 3 (se side 9 til 12) til å kaste en tilfeldig dart og sjekke om den traff

blinken. Dersom vi treffer blinken ønsker vi å øke telleveriabelen, `antall_treff`, med 1.

```
19 for kast in range(antall_kast):
20     # Kast en pil
21     dart_x, dart_y = kast_dart()
22     # Sjekk treff
23     if traff_blinken(dart_x, dart_y):
24         antall_treff += 1
```

Linje 24 passer på at vi hele tiden oppdaterer hvor mange treff vi har fått med 1 dersom darten treffer inni blinken.

Nå har vi alt vi trenger for å estimere sannsynligheten. Vi husker fra avsnitt 2.1 (se side 7) at sannsynligheten kan estimeres ved:

$$P(\text{treffe sirkel}) \approx \frac{\text{antall darts inne i sirkelen}}{\text{antall darts inne i kvadratet}}, \tag{11}$$

Med Python-kode blir det:

```
26 estimert_sannsynlighet = antall_treff/antall_kast
```

4.2 Finne estimatet for π

Vi har altså funnet et estimat for hvor sannsynlig det er å treffe inne i sirkelen når du kaster en dart. I seksjon avsnitt 2.1 fant vi også et uttrykk for π basert på dette estimatet (se side 7) :

$$\pi = 4P(\text{treffe sirkel}). \tag{12}$$

Dette kan vi skrive i Python slik:

```
27 estimert_pi = 4*estimert_sannsynlighet
```

Til slutt kan vi skrive ut resultatene av eksperimentet på en leselig måte:

```
29 print(f'Antall kast: {antall_kast}')
30 print(f'Antall treff: {antall_treff}')
```

```
print(f'Sannsynlighet for treff: {estimert_sannsynlighet}')
print(f'Estimert pi: {estimert_pi}')
```

```
Antall kast: 10
Antall treff: 8
Sannsynlighet for treff: 0.8
Estimert pi: 3.2
```

Prøv å kjøre programmet flere ganger og se hva estimatet av π blir. Synes du det er et godt estimat? Nå estimerer vi sannsynligheten for treff med kun 10 kast. Hvis vi skal få et mer nøyaktig estimat kan vi øke antall kast, for eksempel til 10 000. Hvis vi gjør et mer nøyaktig estimat av sannsynligheten vil også estimatet av π bli mer nøyaktig. Her er et output fra programmet med 10 000 kast.

```
Antall kast: 10000
Antall treff: 7857
Sannsynlighet for treff: 0.7857
Estimert pi: 3.1428
```

Vi ser at dette er er rett med 2 desimalers nøyaktighet! Du kan også øke antall kast enda mer, men programmet vil da bli tregere. Det er en av de store fordelene med såkalte *numeriske* metoder: Selv om du aldri kan få helt nøyaktig estimat, kan du nesten alltid få så nøyaktig som du trenger, hvis du bare er villig til å vente.

Flott! Da har vi et python program som kan estimere *pi* fra tilfeldige dart kast. Det fulle programmet vi har laget frem til nå er gitt under. Gå gjerne igjennom programmet linje for linje og pass på at du forstår alt som skjer før du går videre til neste seksjon:

```
from math import sqrt
from pylab import uniform

def kast_dart():
    x_koordinat = uniform(-1, 1)
    y_koordinat = uniform(-1, 1)
    return x_koordinat, y_koordinat

def pytagoras(katet1, katet2):
    return sqrt(katet1**2 + katet2**2)

```

```
def traff_blinken(dart_x, dart_y):
    avstand_origo = pytagoras(dart_x, dart_y)
    return avstand_origo < 1

antall_kast = 100000
antall_treff = 0

for kast in range(antall_kast):
    # Kast en pil
    dart_x, dart_y = kast_dart()
    # Sjekk treff
    if traff_blinken(dart_x, dart_y):
        antall_treff += 1

estimert_sannsynlighet = antall_treff/antall_kast
estimert_pi = 4*estimert_sannsynlighet

print(f'Antall kast: {antall_kast}')
print(f'Antall treff: {antall_treff}')
print(f'Sannsynlighet for treff: {estimert_sannsynlighet}'
    )
print(f'Estimert pi: {estimert_pi}')
```

5 Visualisering

I denne seksjonen skal vi kan visualisere dartkastene våre i et plot. Visualisering er nyttig fordi det ofte gir en dypere forståelse av hva som skjer i et program, både for programmereren og for andre som skal bruke programmet siden. I denne seksjonene skal vi modifisere programmet, `pi_estimering.py` som vi skrev i avsnitt 4 (se side side 15 for fullstendig program).

5.1 Tegne kvadratet

Det første vi må gjøre er å tegne opp kvadratet. Kvadratet skal ha hjørner i punktene (-1, -1), (-1, 1), (1, 1) og (1, -1). I Python lages plot ved at man sender inn punkter som python så trekker streker igjennom. For å få en firkant slik vi ønsker må vi altså trekke strek fra (-1, -1) til (-1, 1) til (1, 1) til (1, -1) og tilslutt tilbake til (-1, -1).

I python sender vi inn punktene som en liste med x-koordinater og en liste med y-koordinater. For å plotte en firkant kan vi altså bruke følgende kode:

```
1 # importer alt fra pylab
2 from pylab import *
3 # lagre x og y-koordinater i lister
4 x = [-1, -1,  1,  1, -1]
5 y = [-1,  1,  1, -1, -1]
6 # plot firkant med linjeplot
7 plot(x, y)
```

NB: Denne koden vil kun lage plottet. Den vil ikke *vise* noe plot uten en `show()` kommando på slutten.

For å gjøre koden mer oversiktelig putter vi disse kommandoene inn i en egen funksjon, `tegn_kvadrat`. La oss også spesifisere at vi vil ha en sort firkant. Plasser funksjonsdefinisjonen inn i `pi_estimering.py` programmet ditt oppe i koden sammen med funksjonene der.

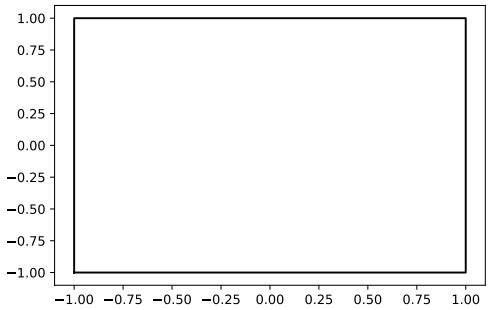
```
16 def tegn_kvadrat():
17     # Tegn et kvadrat med sidelengde lik 2
18     # Kvadratet er gitt ved å tegne strek fra
19     # (-1, -1) til (-1, 1) til (1, 1) til (1, -1) og
         tilbake til (-1, -1)
```

```
20      x = array([-1, -1,  1,  1, -1])
21      y = array([-1,  1,  1, -1, -1])
22      plot(x, y, 'black')
```

Nå kan vi bruke denne funksjonen i `pi_estimering.py` -programmet vårt fra tidligere. Kall på `tegn_kvadrat()` etterfulgt av `show()` helt på slutten av `pi_estimering.py`.

```
46  # Tegn kvadratet
47  tegn_kvadrat()
48  # Vis figuren
49  show()
```

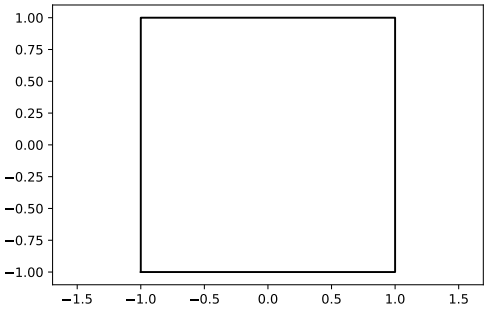
Dette gir oss følgende figur:



Figur 6: Plot av firkant

Legger du merke til noe merkelig med tegningen av kvadratet? Det ser jo ikke veldig kvadratisk ut? Det kommer av at aksene i plottet ikke er likt skalert. For å sørge for at kvadratet ser kvadratisk ut må vi tvinge plottet til å ha lik skalering av aksene. Dette kan vi gjøre med å kallet på `axis('equal')` før vi viser frem plottet med `show()`.

```
46  # Tegn kvadratet
47  tegn_kvadrat()
48  # For å gjøre figuren pen setter vi at lengdene på aksene
       skal være lik
49  axis('equal')
50  # Vis figuren
51  show()
```



Figur 7: Kvadratisk plot

5.2 Tegne sirkelen

Det neste vi må gjøre er å tegne blinken vår, en enhetssirkel. La oss først tenke på hvordan plotting fungerer i Python. Vi har par med (x, y) koordinater og trekker en strek mellom hver etterfølgende koordinat. Altså trenger vi en måte å beskrive punkter på enhetssirkelen som ligger ved sidenav hverandre. For å få til dette kan vi bruke parameterfremstillingen til en sirkel. Parameterfremstillingen til en enhetssirkel er gitt ved

$$x = \cos(\theta), \tag{13}$$
$$y = \sin(\theta), \tag{14}$$

hvor θ er en vinkel (i radianer) som går fra 0 til 2π. Dette kan vi bruke i Python, vi kan opprette en array, `vinkler`, som har verdier mellom 0 og 2π, og bruke den for å opprette to nye arrays, `sirkel_x` og `sirkel_y`, som inneholder x- og y-koordinatene til punktene på sirkelen. Da kan vi sende inn `sirkel_x` og `sirkel_y` til `plot` for å tegne opp sirkelen. La oss lage en funksjon som gjør nettopp dette:

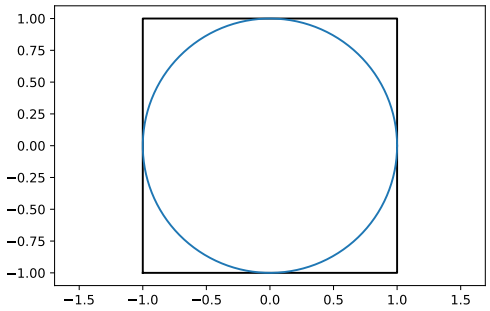
```
24 def tegn_blink():
25     # Tegn tom blink med radius 1
26
27     # Opprett array som inneholder vinkler mellom
28     # 0 og 2pi
29     vinkler = arange(0, 2*pi , 0.001)
30     # cos(theta) og sin(theta) gir første og andre
31     # koordinat for punkter på en enhetssirkel
32     sirkel_x = cos(vinkler)
33     sirkel_y = sin(vinkler)
34     plot(sirkel_x, sirkel_y)
```

NB: For at denne koden skal fungere, må du importere `pi`, `cos` and `sin`. Det er kanskje litt ironisk å importere `pi` i et program som handler om å *estimere* π, men som du har sett er ikke denne funksjonen nødvendig for å gjøre selve estimatet. Vi tegner kun en sirkel for å gjøre visualiseringen penere. Dersom du har lyst til å unngå å bruke π kan du enten droppe helt å tegne sirkelen eller eventuelt erstatte 2π i koden med noe som antageligvis er større, for eksempel 10.

For å tegne opp sirkelen sammen med firkanten kaller du på `tegn_blink()` før show i `pi_estimering.py`

```
46 # Tegn kvadratet
47 tegn_kvadrat()
```

```
48 # For å gjøre figuren pen setter vi at lengdene på aksene
      skal være lik
49 axis('equal')
50 # tegn blinken
51 tegn_blink()
52 # Vis figuren
53 show()
```



Figur 8: Sirkel inni kvadrat

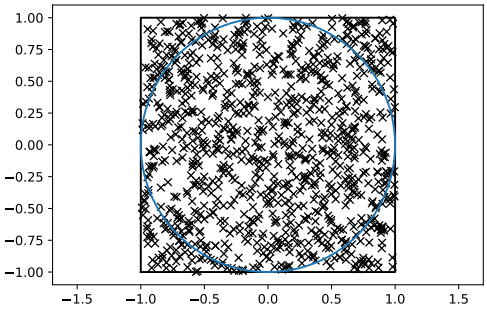
5.3 Tegne dartkastene

Nå har vi et program som tegner kvadratet og enhetssirkelen. Det vi mangler er å tegne selve dartkastene. Dette kan vi gjøre ved å kalle på plot inne i kaste-løkka vår og sende inn x-koordinaten og y-koordinaten til dartkastet. Vi må finne hvilken linje vi kastet darten i `pi_estimering.py` og legge til en plottekommando på linja under som plotter et sort krysss i posisjonen til dartkastet:

```
43     dart_x, dart_y = kast_dart()
44     plot(dart_x, dart_y, 'kx')
```

Vi sender inn `'kx'` for å gi beskjed om at vi vil ha et sort kryss. `k` er kort for black/sort og `x` betyr at vi vil ha et kryss. Du kan for eksempel bytte ut `x` med `*` for å få stjerner i stedenfor. **NB:** Det kan være en god ide å ikke ha flere kast enn 1000 når du plotter siden plotting gjør programmet tregere. Prøv derfor å endre `antall_kast` til `1000` eller mindre før du kjører programmet.

Dette gir følgende plot:



Figur 9: Visualisering av dartkast

Vi ser at det ikke er så lett å se blinken under alle de sorte kryssene. I tillegg hadde det vært fint å få frem visuelt hvilket dartkast som traff og hvilke som bomma.

For å gjøre det kan vi flytte plotte-kommandoen inn i `if`-blokka og be om grønne kryss (`'gx'`) dersom darten traff inne i blinken, og røde kryss hvis ikke (`'rx'`).

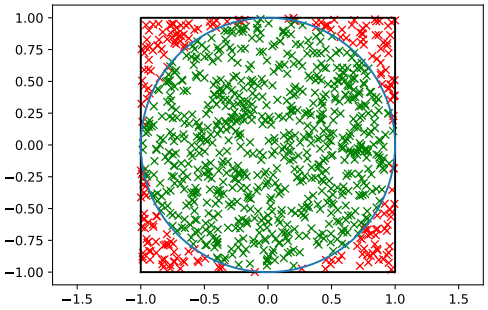
Prøv selv først og sammenlign med koden under:

```
from math import sqrt
from pylab import *

def kast_dart():
    x_koordinat = uniform(-1, 1)
    y_koordinat = uniform(-1, 1)
    return x_koordinat, y_koordinat

def pytagoras(katet1, katet2):
    return sqrt(katet1**2 + katet2**2)

def traff_blinken(dart_x, dart_y):
    avstand_origo = pytagoras(dart_x, dart_y)
    return avstand_origo < 1

def tegn_kvadrat():
    # Tegn et kvadrat med sidelengde lik 2
    # Kvadratet er gitt ved å tegne strek fra
    # (-1, -1) til (-1, 1) til (1, 1) til (1, -1) og
       tilbake til (-1, -1)
    x = array([-1, -1,  1,  1, -1])
```

```
    y = array([-1,  1,  1, -1, -1])
    plot(x, y, 'black')

def tegn_blink():
    # Tegn tom blink med radius 1

    # Opprett array som inneholder vinkler mellom
    # 0 og 2pi
    vinkler = arange(0, 2*pi , 0.001)
    # cos(theta) og sin(theta) gir første og andre
    # koordinat for punkter på en enhetssirkel
    sirkel_x = cos(vinkler)
    sirkel_y = sin(vinkler)
    plot(sirkel_x, sirkel_y)

antall_kast = 1000
antall_treff = 0

for kast in range(antall_kast):
    # Kast en pil
    dart_x, dart_y = kast_dart()

    # Sjekk treff
    if traff_blinken(dart_x, dart_y):
        antall_treff += 1

        #Tegn grønt darttreff
        plot(dart_x, dart_y, 'gx')
    else:
        #Tegn rødt darttreff
        plot(dart_x, dart_y, 'rx')

estimert_sannsynlighet = antall_treff/antall_kast
estimert_pi = 4*estimert_sannsynlighet

print(f'Antall kast: {antall_kast}')
print(f'Antall treff: {antall_treff}')
print(f'Sannsynlighet for treff: {estimert_sannsynlighet}'
   )
print(f'Estimert pi: {estimert_pi}')
```

```
60
61 # Tegn kvadratet
62 tegn_kvadrat()
63 # For å gjøre figuren pen setter vi at lengdene på aksene
     skal være lik
64 axis('equal')
65 # tegn blinken
66 tegn_blink()
67 # Vis figuren
68 show()
```

Koden gir et plott som dette:



Figur 10: Visualisering av treff og bom

Gratulerer! Du har nå en helt egen π estimator med visualisering av Monte Carlo metoden!

A BONUSUTVIDELSE: Integrere funksjoner med Monte Carlo metoden

A.1 Modifisere π estimeringen til å estimere integral

I denne delen av kompendiet skal vi se litt på hvilke endringer du kan gjøre med π-estimering-programmet ditt for å finne integralet av en funksjon. Å integrere en funksjon kan nemlig også gjøres med dartkast på blink i en boks. Det eneste vi må endre på er hva som er "blinken", hva som er "boksen" og hva det vil si å "treffe".

Som eksempel vil vi prøve å løse følgende integral:

$$\int_0^3 x^3 - 3x^2 + x + 5dx \tag{15}$$

Vi begynner med å kopiere koden fra avsnitt 5.3 på side 22 inn i en ny fil, `monte_carlo_integrasjon.py`

Det første vi kan gjøre er å skrive inn funksjonen som en funksjon i Python:

```
32 def f(x):
33     return x**3 - 3*x**2 + x + 5
```

(Denne funksjonen kan du for eksempel plassere under de andre funksjonene i `pi_estimering.py`)

La oss også fjerne linjene som regner og skriver ut π, siden det ikke lenger er det vi prøver på.

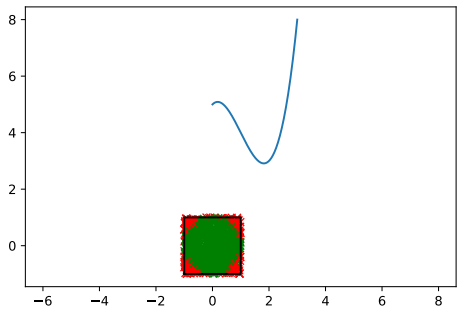
Så kan vi endre `tegn_blink` funksjonen til å tegne grafen til `f` som blinken vår i stedet for enhetssirkelen. Da må vi bestemme grenseverdiene til funksjonen. Hvilke x-koordinater vil vi tegne den for? Vi ønsker å regne ut integralet mellom 0 og 3, så la oss definere variablene `start_x=0` og `slutt_=3`.

Da kan vi bruke `arange(start_x, slutt_x, 0.001)` til å lage en *array* av 1001 x-verdier mellom 0 og 3. `tegn_blink` funksjonen skal altså se slik ut:

```
25 def tegn_blink():
26     # Tegn opp grensa til "blinken".
27     # Når vi integrer en funksjon vil det tilsvare
          funksjonskurven
```

```
    x = arange(start_x, slutt_x, 0.001)
    y = f(x)
    plot(x, y)
```

Vi har ikke gjort noen endringer i selve estimeringen enda, men la oss kjøre koden og se hvordan plottet ser ut:



Her ser vi at funksjonskurven er langt utenfor "boksen" vi "kaster dart" i. Vi må altså endre område til denne boksen. Hvilket område ønsker vi at den skal dekke?

Vel, vi ønsker å estimere integralet fra likning (15) altså arealet under grafen til funksjonen. Det betyr at boksen vi kaster dart i, må inkludere dette arealet.

Minste y-verdi er altså 0 og y-verdi må være stor nok til at grafen ikke "stikker utenfor". En måte å sørge for det på er å sette den til å være lik høyeste y-verdi innenfor området vi ser på. For å gjøre det må vi først lage en array med `y_verdier` for x mellom 0 og 3 og så bruke den innebygde `max`-funksjonen til å hente ut den største av y-verdier. `start_x` og `slutt_x` blir fortsatt grenseverdiene 0 og 3. Vi har dermed fire variabler som definerer området til boksen:

```
antall_kast = 1000
antall_treff = 0

start_x = 0 # der vi starter integralet
slutt_x = 3 # der vi slutter integralet

x_verdier = arange(start_x, slutt_x, 0.01) # hent ut noen
   x-verdier
y_verdier = f(x_verdier) # hent ut tilhørende y-verdier

```

```
49 start_y = 0 # boksen skal starte på 0
50 slutt_y = max(y_verdier) # For å ivære sikker på å
      inkludere hele funkjsonsgrafen
```

La oss nå modifisere `tegn_kvadrat()` og `kast_dart()` til å bruke disse variablene.

`tegn_kvadrat()` skal nå plotte en firkant mellom $(x_{\text{start}}, y_{\text{start}})$, $(x_{\text{start}}, y_{\text{slutt}})$, $(x_{\text{slutt}}, y_{\text{slutt}})$ og $(x_{\text{slutt}}, y_{\text{start}})$ Koden blir altså:

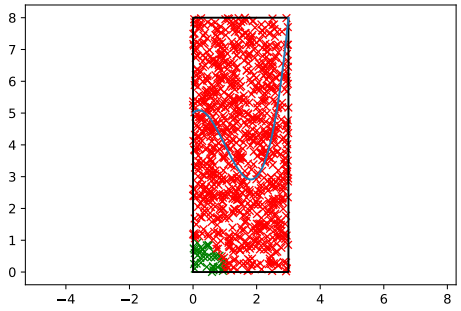
```
15 def tegn_kvadrat():
16     # Tegn en boks vi skal regne integral inni
17     # boksen er gitt ved å tegne strek
18     # fra (start_x, start_y) til (start_x, slutt_y)
19     # til (slutt_x, slutt_y) til (slutt_x, start_y)
20     # og tilbake til (start_x, start_y)
21     x = array([start_x, start_x,  slutt_x,  slutt_x,
          start_x])
22     y = array([start_y, slutt_y,  slutt_y, start_y,
          start_y])
23     plot(x, y, 'black')
```

Det er sikkert lurt å endre navnet fra `tegn_kvadrat` til noe mer passende som `tegn_boks`, men for å gjøre ting så enkelt som mulig, og unngå å introdusere bugs nå mens vi jobber med funksjonaliteten til programmet, venter vi med å endre funksjonsnavn til slutt.

Nå har vi endret tegningen av boksen. For å endre kaste-området må vi også endre `kast_dart` til å kaste dart inne i boksen. Vi ser at `kast_dart` trekker x-koordinat mellom -1 og 1. Nå ønsker vi istedet at den skal trekke mellom `start_x` og `slutt_x`. Tilsvarende ønsker vi at y-koorinaten skal være et tilfeldig tall mellom `start_y` og `slutt_y` Den oppdaterte `kast_dart()` funksjonen blir da slik:

```
4 def kast_dart():
5     x_koordinat = uniform(start_x, slutt_x)
6     y_koordinat = uniform(start_y, slutt_y)
7     return x_koordinat, y_koordinat
```

Før vi går videre er det lurt å kjøre koden vi har skrevet så langt å se hvordan plottet ser ut:



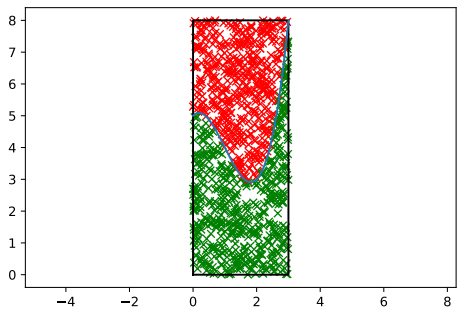
Her ser vi at dartene og boksen ser rett ut. Men "treff-området" er fortsatt knyttet til enhetssirkelen. For å fikse det, må vi oppdatere `traff_blinken` funksjonen.

For å gjøre det må vi spørre oss selv hva et "treff" er? Hva vil det si å "treffe"? I dette tilfelle ønsker vi å finne arealet under grafen. Et "treff" skjer altså dersom "darten" havner under funksjonsgrafen. Det betyr at `y_dart` er *mindre* enn `f(x_dart)`.

Den oppdaterte `traff_blinken` koden blir slik:

```
12 def traff_blinken(dart_x, dart_y):
13     return dart_y < f(dart_x)
```

La oss nå kjøre koden igjen og se hva vi får:



Nå stemmer treffområdet også! Vi har altså et program som estimerer sannsynligheten for treff under grafen. Hvordan kan vi gjøre om til å finne integralet?

Vi har en boks som vi kaster dart i, så arealet til det grønne området må være mindre enn arealet til hele den boksen. Derfor er et naturlig utgangspunkt å lage en funksjon som regner ut arelaet til denne boksen. Vi vet at arealet til boksen er gitt ved

$$A_{\text{boks}} = \text{lengde} \times \text{høyde}, \tag{16}$$

hvor lengden er gitt ved $x_{\text{start}} - x_{\text{slutt}}$ og høyden er gitt ved starty, y_{slutt}. Hvis vi skriver dette med Python får vi denne funksjonen:

```
35 def boks_areal(start_x, slutt_x, start_y, slutt_y):
36     boks_lengde = slutt_x - start_x
37     boks_høyde = slutt_y - start_y
38     return boks_lengde*boks_høyde
```

Det neste steget er å finne arealet under funksjonen, A_f. Vi vet at sannsynligheten for å treffe under funksjonen er gitt ved

$$P\left(\text{treff under } f\right) = \frac{A_f}{A_{\text{boks}}}, \tag{17}$$

hvilket betyr at vi kan finne arelaet under funksjonen ved å gange sammen treff-sannsynligheten med arealet til boksen. Setter vi sammen dette får vi altså at

$$\int_{x_{\text{start}}}^{x_{\text{slutt}}} f(x)dx = A_f = A_{\text{boks}} \times P\left(\text{treff under } f\right). \tag{18}$$

Vi kan nå legge til den koden på bunnen av `monte_carlo_integrasjon.py` programmet vårt slik:

```
67 estimert_sannsynlighet = antall_treff/antall_kast
68
69 total_areal = boks_areal(start_x, slutt_x, start_y,
       slutt_y)
70 integral = estimert_sannsynlighet*total_areal
71
72 print(f'Antall kast: {antall_kast}')
73 print(f'Antall treff: {antall_treff}')
```

```
print(f'Sannsynlighet for treff: {estimert_sannsynlighet}')
print(f'Integralet er: {integral}')
```

```
Antall kast: 1000
Antall treff: 540
Sannsynlighet for treff: 0.54
Integralet er: 12.96
```

Vi kan dobbeltsjekke programmet ved å sammenligne med en analytisk løsning av integralet. I dette tilfelle er løsningen **12,75**.

Estimatet vårt er altså ganske nærme, men ikke helt nøyaktig. Om det er nøyaktig nok avhenger av hva vi skal bruke svaret til. Prøv å øke antall kast og se hvor nøyaktig du kan få integral estimatet (**NB:** Plotting gjør programmet ganske mye tregere, så hvis du vil ha et veldig høyt antall kast må du kanskje fjerne eller kommentere vekk plottefunksjonaliteten).

Nå har vi kode som regner ut integralet av en hvilken som helst funksjon[4]! Prøv å endre `start_x` og `start_y` og se hva som skjer. Du kan også prøve å endre hvilken funksjon du integrerer over!

Full kode:

```
from math import sqrt
from pylab import *

def kast_dart():
    x_koordinat = uniform(start_x, slutt_x)
    y_koordinat = uniform(start_y, slutt_y)
    return x_koordinat, y_koordinat

def pytagoras(katet1, katet2):
    return sqrt(katet1**2 + katet2**2)

def traff_blinken(dart_x, dart_y):
    return dart_y < f(dart_x)

def tegn_kvadrat():
```

[4]Dette stemmer ikke helt og i tillegg A.3 snakker vi litt om utfordringer ved negative funksjoner og diskuterer mulige løsninger

```
16      # Tegn en boks vi skal regne integral inni
17      # boksen er gitt ved å tegne strek
18      # fra (start_x, start_y) til (start_x, slutt_y)
19      # til (slutt_x, slutt_y) til (slutt_x, start_y)
20      # og tilbake til (start_x, start_y)
21      x = array([start_x, start_x,  slutt_x,  slutt_x,
            start_x])
22      y = array([start_y, slutt_y,  slutt_y, start_y,
            start_y])
23      plot(x, y, 'black')
24
25  def tegn_blink():
26      # Tegn opp grensa til "blinken".
27      # Når vi integrer en funksjon vil det tilsvare
            funksjonskurven
28      x = arange(start_x, slutt_x, 0.001)
29      y = f(x)
30      plot(x, y)
31
32  def f(x):
33      return x**3 - 3*x**2 + x + 5
34
35  def boks_areal(start_x, slutt_x, start_y, slutt_y):
36      boks_lengde = slutt_x - start_x
37      boks_høyde = slutt_y - start_y
38      return boks_lengde*boks_høyde
39
40  antall_kast = 1000
41  antall_treff = 0
42
43  start_x = 0 # der vi starter integralet
44  slutt_x = 3 # der vi slutter integralet
45
46  x_verdier = arange(start_x, slutt_x, 0.01) # hent ut noen
        x-verdier
47  y_verdier = f(x_verdier) # hent ut tilhørende y-verdier
48
49  start_y = 0 # boksen skal starte på 0
50  slutt_y = max(y_verdier) # For å ivære sikker på å
        inkludere hele funkjsonsgrafen
```

```
51
52
53  for kast in range(antall_kast):
54      # Kast en pil
55      dart_x, dart_y = kast_dart()
56
57      # Sjekk treff
58      if traff_blinken(dart_x, dart_y):
59          antall_treff += 1
60
61          #Tegn grønt darttreff
62          plot(dart_x, dart_y, 'gx')
63      else:
64          #Tegn rødt darttreff
65          plot(dart_x, dart_y, 'rx')
66
67  estimert_sannsynlighet = antall_treff/antall_kast
68
69  total_areal = boks_areal(start_x, slutt_x, start_y,
        slutt_y)
70  integral = estimert_sannsynlighet*total_areal
71
72  print(f'Antall kast: {antall_kast}')
73  print(f'Antall treff: {antall_treff}')
74  print(f'Sannsynlighet for treff: {estimert_sannsynlighet}'
        )
75  print(f'Integralet er: {integral}')
76  # Tegn kvadratet
77  tegn_kvadrat()
78  # For å gjøre figuren pen setter vi at lengdene på aksene
        skal være lik
79  axis('equal')
80  # tegn blinken
81  tegn_blink()
82  # Vis figuren
83  show()
```

A.2 Ryddigere funksjonsnavn

Noen av funksjonsnavnene passer ikke helt inn i programmet lenger, og før vi gir oss kan det være lurt å endre dem. For eksempel kan `tegn_kvadrat` og `tegn_blink` kanskje heller hete `tegn_boks` og `tegn_graf` siden vi ikke lenger tegner et kvadrat og en blink men heller en funksjonsgraf med en boks rundt. `traff_blinken` kan for eksempel omdøpes til `traff_integral`. Dette er bare noen forslag og det beste er om du selv går igjennom koden og tenker gjennom hva hver funksjon gjør og hva et fornuftig navn på det kan være.

Her er den fullstendige koden med oppdaterte funksjonsnavn:

```
from pylab import *

def kast_dart():
    x_koordinat = uniform(start_x, slutt_x)
    y_koordinat = uniform(start_y, slutt_y)
    return x_koordinat, y_koordinat

def traff_integral(dart_x, dart_y):
    return dart_y < f(dart_x)

def tegn_boks():
    # Tegn en boks vi skal regne integral inni
    # boksen er gitt ved å tegne strek
    # fra (start_x, start_y) til (start_x, slutt_y)
    # til (slutt_x, slutt_y) til (slutt_x, start_y)
    # og tilbake til (start_x, start_y)
    x = array([start_x, start_x,  slutt_x,  slutt_x,
       start_x])
    y = array([start_y, slutt_y,  slutt_y, start_y,
       start_y])
    plot(x, y, 'black')

def tegn_graf():
    # Tegn opp grensa til "blinken".
    # Når vi integrer en funksjon vil det tilsvare
       funksjonskurven
    x = arange(start_x, slutt_x, 0.001)
    y = f(x)
    plot(x, y)
```

```
27
28  def f(x):
29      return x**3 - 3*x**2 + x + 5
30
31  def boks_areal(start_x, slutt_x, start_y, slutt_y):
32      boks_lengde = slutt_x - start_x
33      boks_høyde = slutt_y - start_y
34      return boks_lengde*boks_høyde
35
36  antall_kast = 1000
37  antall_treff = 0
38
39  start_x = 0 # der vi starter integralet
40  slutt_x = 3 # der vi slutter integralet
41
42  x_verdier = arange(start_x, slutt_x, 0.01) # hent ut noen
       x-verdier
43  y_verdier = f(x_verdier) # hent ut tilhørende y-verdier
44
45  start_y = 0 # boksen skal starte på 0
46  slutt_y = max(y_verdier) # For å ivære sikker på å
       inkludere hele funkjsonsgrafen
47
48
49  for kast in range(antall_kast):
50      # Kast en pil
51      dart_x, dart_y = kast_dart()
52
53      # Sjekk treff
54      if traff_integral(dart_x, dart_y):
55          antall_treff += 1
56
57          #Tegn grønt darttreff
58          plot(dart_x, dart_y, 'gx')
59      else:
60          #Tegn rødt darttreff
61          plot(dart_x, dart_y, 'rx')
62
63  estimert_sannsynlighet = antall_treff/antall_kast
64
```

```
65 total_areal = boks_areal(start_x, slutt_x, start_y,
       slutt_y)
66 integral = estimert_sannsynlighet*total_areal
67
68 print(f'Antall kast: {antall_kast}')
69 print(f'Antall treff: {antall_treff}')
70 print(f'Sannsynlighet for treff: {estimert_sannsynlighet}'
       )
71 print(f'Integralet er: {integral}')
72 # Tegn boks
73 tegn_boks()
74 # For å gjøre figuren pen setter vi at lengdene på aksene
       skal være lik
75 axis('equal')
76 # tegn graf
77 tegn_graf()
78 # Vis figuren
79 show()
```

A.3 Hva gjør vi når funksjonen er negativ?

Koden vi har skrevet her kan løse veldig mange integral. Den kan faktisk brukes for å tilnærme integral som er umulig å løse for hånd (slik som integralet av $e^{(x^2)}$)! Men det er også noen integral vi ikke kan løse med koden vi har her, nemlig integral av funksjoner med negative verdier. Hvis du har en slik funksjon er det tre løsninger.

Den første løsningen er å bruke den følgende sammenhengen:

$$\int_{x_{\text{start}}}^{x_{\text{stop}}} f(x)dx = \int_{x_{\text{start}}}^{x_{\text{stop}}} (f(x)+c)dx - \int_{x_{\text{start}}}^{x_{\text{stop}}} cdx, \tag{19}$$

for en konstant c snn at $f(x)+c$ er positiv og løse problemene

$$\int_{x_{\text{start}}}^{x_{\text{stop}}} (f(x)+c)dx, \tag{20}$$

og

$$\int_{x_{\text{start}}}^{x_{\text{stop}}} cdx. \tag{21}$$

Det siste problemet kan vi løse analytisk, det er gitt ved

$$\int_{x_{\text{start}}}^{x_{\text{stop}}} c dx = c\left(x_{\text{stop}} - x_{\text{start}}\right), \tag{22}$$

og problemet i likning (20) kan vi løse med Monte Carlo metoden.

Alternativt kan vi finne alle områdene hvor funksjonen krysser x-aksen og løse integralene separat for hvert område hvor funksjonen kun er på en side av x-aksen. Ulempen med dette er at vi må vite alle punktene hvor funksjonen vår skifter fortegn, og det er ikke nødvendigvis enkelt.

Den siste løsningen er å bruke en annen numerisk metode, slik som *trapesmetoden* eller *Simpsons metode*. Fordelen med disse er at de virker for alle integral med en variabel, mens ulempen er at de, i motsetning til Monte Carlo metoden, ikke virker hvis vi har integral med mange variabler.

B BONUSUTVIDELSE: Visualisere nøyaktigheten

Så langt har vi regnet ut et estimat for π, men vi vet jo ikke hvor nøyaktig dette estimatet er. Dette er problematisk. Tenk deg for eksempel at du skal styre en satellitt som ligger ca 50000km over jordoverflaten. Da beveger satelitten seg i en bane som har en radius på ca 55000km. Hvis vi vil flytte denne satelitten til andre siden av jorda, så må vi flytte den $\pi \times 55000$km, mens hvis vi har estimert π til å være 3.31, så vil satellitten bomme med over 7000km! I dette tilfellet må vi altså være trygge på at vi har estimert π nøyaktig.

Den enkleste måten å vite hvor nøyaktig vi har estimert π på er å gjenta π forsøket mange ganger og telle hva π blir hver gang. For å gjøre det kan vi for eksempel bruke enda en `for`-løkke for å gjenta forsøket mange ganger og lagre verdien for π i en array eller liste hver iterasjon. I dette eksempelet tar vi utgangspunkt i `pi_estimering.py` scriptet du finner på side 15 i kompendiet.

Det første vi gjør er å opprette en variabel som beskriver hvor mange forskjellige π-estimat vi ønsker, og kaller den `antall_estimat`.

```
16 # Vi vil generere mange pi-estimat
17 antall_estimat = 1000
```

For å lagre de forskjellige estimatene våre til π bruker vi en array, `pi_verdier`, derfor starter vi programmet med å importere `zeros` funksjonen fra `pylab`.

```
2 from pylab import uniform, zeros
```

Så bruker vi `zeros` funksjonen vi akkurat importerte til å opprette en `array`-variabel som vi kaller for `alle_pi_estimat`. I denne variabelen skal vi lagre alle de forskjellige estimatene våre på verdien til π.

```
16 # Vi vil generere mange pi-estimat
17 antall_estimat = 1000
18 alle_pi_estimat = zeros(antall_estimat)
```

Den siste biten av estimeringen er å flytte simuleringskoden inn i en `for`-løkke slik at vi får mange estimat. Siden vi skal estimere π mange ganger kan det også være lurt å kaste relativt få dart hver gang. Derfor setter vi `antall_kast =1000`.

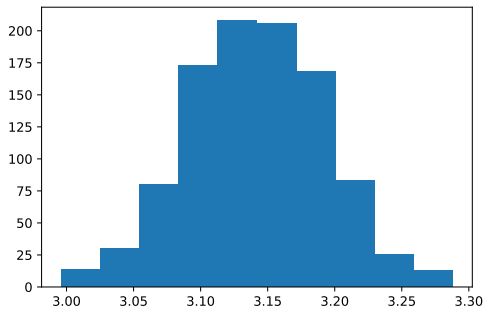
```
29 for estimat_nummer in range(antall_estimat):
30     antall_kast = 1000
31     antall_treff = 0
```

```
32      for kast in range(antall_kast):
33          # Kast en pil
34          dart_x, dart_y = kast_dart()
35          # Sjekk treff
36          if traff_blinken(dart_x, dart_y):
37              antall_treff += 1
38
39      estimert_sannsynlighet = antall_treff/antall_kast
40      estimert_pi = 4*estimert_sannsynlighet
41
42      alle_pi_estimat[estimat_nummer] = estimert_pi
```

Etter at vi har lagret alle estimatene av π kan vi måle nøyaktigheten til estimatet ved å bruke et histogram. I Python, kan vi bruke `hist` funksjonen i `pylab` for å tegne et histogram. La oss modifisere `pi_estimering.py` slik at vi tegner histogrammet du ser i figur 11.

```
2  from pylab import uniform, zeros, hist, show
```

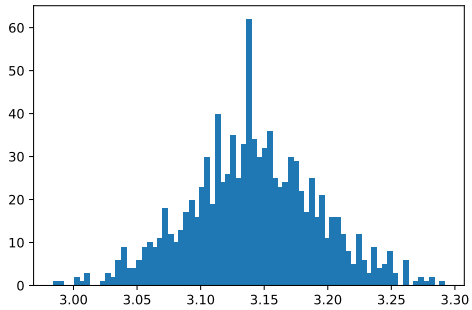
```
35 # Så viser vi frem estimatene våre i et histogram
36 hist(alle_pi_estimat)
37 show()
```



Figur 11: Histogram over π estimat

Hvis vi kjører programmet nå får vi et fint histogram, men det bygd opp av veldig få stolper. Siden vi har estimert π veldig mange ganger kan vi jo ha mange flere stolper i histogrammet vårt. For å endre antallet stolper i histogrammet kan vi gi `hist` funksjonen vår to input-variabler: arrayet med alle π estimatene og antallet

søyler vi ønsker. La oss lage et histogram med 75 søyler. Hvis vi gjør det ser koden slik ut og histogrammet vi får kan du se i figur 12.



Figur 12: Endelig histogram over π estimat. Histogrammet forteller hvor stor spredning det er på estimatene hvilket gir oss innsikt i hvor nøyaktig estimatet er. I tillegg ser vi at toppen av histogrammet ligger ganske nærme den faktiske verdien av π

```
39 # Og skriver ut gjennomsnittestimatet til brukeren
40 endelig_pi_estimat = mean(alle_pi_estimat)
```

Men, når vi skriver ut estimatet vårt på π, så bruker vi jo bare det estimatet vårt. Her kan vi få til noe bedre! For å få til det bruker vi isteden gjennomsnittet av alle de 1000 π estimatene våre. For å få til det må vi importere `mean` funksjonen i `pylab`.

```
2 from pylab import uniform, zeros, hist, show, mean
```

Deretter kan vi regne ut gjennomsnittet av alle π estimatene våre og lagre de i en variabel `endelig_pi_estimat` som vi skriver ut til brukeren.

```
39 # Og skriver ut gjennomsnittestimatet til brukeren
40 endelig_pi_estimat = mean(alle_pi_estimat)
41 print(f'Estimert pi: {endelig_pi_estimat}')
```

Nå har vi laget et program som regner ut π ved å kaste mange dart-piler og visualiserer hvor nøyaktig dette estimatet er! Du kan se hele `pi_estimering_med_usikkerhet.py`-programmet her:

```
1 from math import sqrt
2 from pylab import uniform, zeros, hist, show, mean
3
```

```
def kast_dart():
    x_koordinat = uniform(-1, 1)
    y_koordinat = uniform(-1, 1)
    return x_koordinat, y_koordinat

def pytagoras(katet1, katet2):
    return sqrt(katet1**2 + katet2**2)

def traff_blinken(dart_x, dart_y):
    avstand_origo = pytagoras(dart_x, dart_y)
    return avstand_origo < 1

# Vi vil generere mange pi-estimat
antall_estimat = 1000
alle_pi_estimat = zeros(antall_estimat)

for estimat_nummer in range(antall_estimat):
    antall_kast = 1000
    antall_treff = 0
    for kast in range(antall_kast):
        # Kast en pil
        dart_x, dart_y = kast_dart()
        # Sjekk treff
        if traff_blinken(dart_x, dart_y):
            antall_treff += 1

    estimert_sannsynlighet = antall_treff/antall_kast
    estimert_pi = 4*estimert_sannsynlighet

    alle_pi_estimat[estimat_nummer] = estimert_pi

# Så viser vi frem estimatene våre i et histogram med 75 s
  øyler
hist(alle_pi_estimat, 75)
show()

# Og skriver ut gjennomsnittestimatet til brukeren
endelig_pi_estimat = mean(alle_pi_estimat)
print(f'Estimert pi: {endelig_pi_estimat}')
```

C Relevante kompetansemål

I dette kompendiet vil vi gå igjennom estimering av pi med Monte-Carlo-metoden fra grunnen av steg for steg. Det bør nevnes at man alternativt kan tillpasse vanskelighetsgraden med å gi biter av koden og/eller matematikken ferdig løst.

Dette prosjektet kombinerer algoritmisk tenking, geometri i planet, sannynlighetsestimering og numerisk integrasjon. Det rører derfor ved flere forskjellige kompetansemål. Det betyr at man kan tilpasse hvilke kompetansemål man vil rette seg mot ved å endre hvilke biter av matematikken/koden elever må løse selv og hvilke som gis ferdig utfylt. Under følger et utplukk relevante kompetansemål.

1T

- formulere og løyse problem ved hjelp av algoritmisk tenking, ulike problemløysingsstrategiar, digitale verktøy og programmering
- identifisere variable storleikar i ulike situasjonar, setje opp formlar og utforske desse ved hjelp av digitale verktøy
- modellere situasjonar knytte til ulike tema, drøfte, presentere og forklare resultata og argumentere for om modellane er gyldige
- bruke trigonometri til å analysere og løyse samansette teoretiske og praktiske problem med lengder, vinklar og areal

S1

- bruke digitale verktøy til å simulere og utforske utfall i stokastiske forsøk, og forstå begrepet stokastiske variabler

S2

- simulere utfall i, utforske og tolke ulike statistiske fordelinger, og gi eksempler på reelle anvendelser av disse fordelingene

R1

- forstå begrepet vektor og regneregler for vektorer i planet, og bruke vektorer til å beregne ulike størrelser i planet

R2

- utvikle algoritmer for å beregne integraler numerisk, og bruke programmering til å utføre algoritmene
- utforske egenskaper ved radianer og trigonometriske funksjoner og identiteter og anvende disse egenskapene til å løse praktiske problemer (for å tegne sirkel bruker vi radianer)