Stochastic Simulation Coursework 2

tmp120 01854740

November 2022

1 Importance sampling for marginal likelihoods

1.

$p(x) = N(x; 0, 1)$ and $p(y|x) = N(y; x, 1)$
so $p(y) = \int p(y|x)p(x) = \int N(y; x, 1)N(x; 0, 1) = N(y; 0, 1 + 1) = N(y; 0, 2) = \frac{1}{\sqrt{2}\sqrt{2\pi}} e^{-\frac{y^2}{4}}$

```
import numpy as np
def p(y):
    return 1/(np.sqrt(4*np.pi))*np.exp(-y**2/4)
p(9)
```

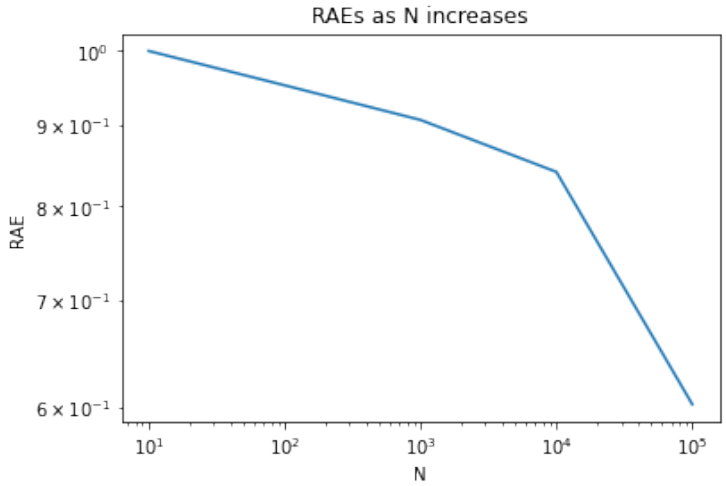
Hence $p(y = 9) = 4.53 * 10^{-10}$

2.

$p(y = 9) = \int (y = 9|x)p(x)dx$
Let the test function $\phi(x) = p(y = 9|x)$. Then the estimator $\hat{p}(y = 9) = \frac{1}{N} \sum_{i=1}^{N} p(y = 9|X_i)$ where $X_i \sim p(x)$.
For $N = 10, 100, 1000, 10000, 100000$, the MC estimator gives: 5.92e-17, 2.17e-11, 4.26e-11, 7.21e-11 and 1.80e-10.
So the RAEs have the following values: 1.00, 0.95, 0.91, 0.84, 0.60.
Note we obtain variation in the results every time we rerun the code though the RAE tends to decrease as N is increased.

```
def pygivenx(y,x):
    return 1/(np.sqrt(2*np.pi))*np.exp(-(y-x)**2/2)
N=10
x_s = np.zeros(N)
for i in range(N):
    x_s[i] = np.random.normal(0,1)
p_hat = (1/N) * np.sum(pygivenx(9,x_s))
RAE10 = abs(p_hat-py(9))/abs(py(9))
N=100
x_s = np.zeros(N)
```

```
for i in range(N):
    x_s[i] = np.random.normal(0,1)
p_hat = (1/N) * np.sum(pygivenx(9,x_s))
RAE100 = abs(p_hat-py(9))/abs(py(9))
N=1000
x_s = np.zeros(N)
for i in range(N):
    x_s[i] = np.random.normal(0,1)
p_hat = (1/N) * np.sum(pygivenx(9,x_s))
RAE1000 = abs(p_hat-py(9))/abs(py(9))
N=10000
x_s = np.zeros(N)
for i in range(N):
    x_s[i] = np.random.normal(0,1)
p_hat = (1/N) * np.sum(pygivenx(9,x_s))
RAE10000 = abs(p_hat-py(9))/abs(py(9))
N=100000
x_s = np.zeros(N)
for i in range(N):
    x_s[i] = np.random.normal(0,1)
p_hat = (1/N) * np.sum(pygivenx(9,x_s))
RAE100000=abs(p_hat-py(9))/abs(py(9))
import matplotlib.pyplot as plt
plt.loglog([10,100,1000,10000,100000],[RAE10,RAE100,RAE1000,
                                       RAE10000,RAE100000])
plt.xlabel("N")
plt.ylabel("RAE")
plt.title("RAEs as N increases")
```



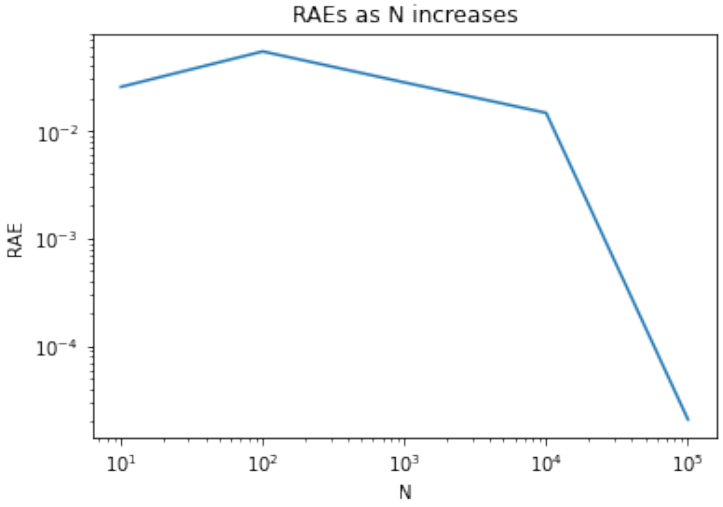
3.

We have the proposal $q(x) = N(x; 6, 1)$ and $p(x) = N(x; 0, 1)$.
So we sample $X_i \sim q(x)$ for $i = 1$ to N.
Let $w_i = \frac{p(X_i)}{q(X_i)}$ and as before $\phi(x) = p(y|x)$.
Now $\hat{p}_{IS}(y = 9) = \frac{1}{N} \sum_{i=1}^{N} w_i p(y = 9|X_i)$ with the X_i from $q(x)$.

The IS estimator for $N = 10, 100, 1000, 10000, 100000$ gives: 4.41e-10, 4.77e-10, 4.40e-10, 4.59e-10 and 4.53e-10
Hence, the RAE has values: 0.025, 0.054, 0.028, 0.015 and 2.09e-05.
When we rerun the code there is variation but generally a decrease in RAE as N increases.

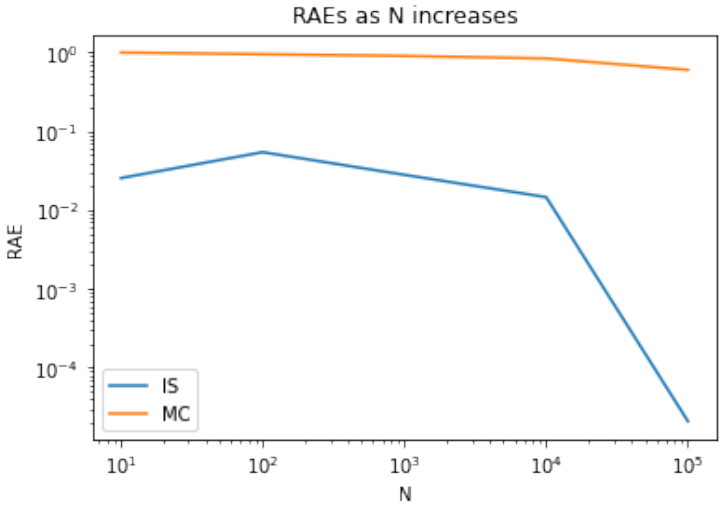
```
def q(x):
    return 1/(np.sqrt(2*np.pi))*np.exp(-(x-6)**2/2)
def p(x):
    return  1/(np.sqrt(2*np.pi))*np.exp(-x**2/2)
def w(x):
    return p(x)/q(x)
N=10
x_s = np.zeros(N)
weights = np.zeros(N)
for i in range(N):
    x_s[i] = np.random.normal(6,1)
    weights[i] = w(x_s[i])
p_IS = (1/N) * np.sum(weights * pygivenx(9,x_s))
RAE10_IS = abs(p_IS-py(9))/abs(py(9))
N=100
x_s = np.zeros(N)
weights = np.zeros(N)
for i in range(N):
    x_s[i] = np.random.normal(6,1)
    weights[i] = w(x_s[i])
p_IS = (1/N) * np.sum(weights * pygivenx(9,x_s))
RAE100_IS = abs(p_IS-py(9))/abs(py(9))
N=1000
x_s = np.zeros(N)
weights = np.zeros(N)
for i in range(N):
    x_s[i] = np.random.normal(6,1)
    weights[i] = w(x_s[i])
p_IS = (1/N) * np.sum(weights * pygivenx(9,x_s))
RAE1000_IS = abs(p_IS-py(9))/abs(py(9))
N=10000
x_s = np.zeros(N)
weights = np.zeros(N)
for i in range(N):
    x_s[i] = np.random.normal(6,1)
    weights[i] = w(x_s[i])
p_IS = (1/N) * np.sum(weights * pygivenx(9,x_s))
RAE10000_IS = abs(p_IS-py(9))/abs(py(9))
N=100000
x_s = np.zeros(N)
weights = np.zeros(N)
for i in range(N):
    x_s[i] = np.random.normal(6,1)
    weights[i] = w(x_s[i])
p_IS = (1/N) * np.sum(weights * pygivenx(9,x_s))
RAE100000_IS = abs(p_IS-py(9))/abs(py(9))
plt.loglog([10,100,1000,10000,100000],[RAE10_IS,RAE100_IS,
                                       RAE1000_IS,RAE10000_IS,
                                       RAE100000_IS])
```

```
plt.xlabel("N")
plt.ylabel("RAE")
plt.title("RAEs as N increases")
```



4.

The IS estimator is much more accurate than MC for all values of N and increasingly so as N increases. Whilst the RAE of the MC starts at about 1 and decreases to a bit above 0.5 when N=100000, the RAE of the IS estimator decreases almost to order of 10^{-5} i.e. converges to the true value quickly whilst MC is just about within an order of 10 for N=100000.



2 Metropolis-Hastings for 1d source localisation

1.

The model has prior $p(x) = N(x; \mu_x, \sigma_x^2)$. The hidden object at location x in 1D (we aim to sample $p(x|y_{1:3}, s_{1:3})$) is observed via three sensors at s_i and so $||x - s_i||$ is used to measure the distance of each sensor to x. The likelihoods are $N(y_i; ||x-s_i||, \sigma_y^2)$ where the y_i are the data readings from the sensors, which each have noise and we use the proposal $q(x'|x) = N(x'; x, \sigma_q^2)$ for Metropolis Hastings.
$p(x|y_{1:3}, s_{1:3})$ is proportional to $p(y_1, y_2, y_3|x, s_1, s_2, s_3)p(x)$ and by conditional independence $p(y_1, y_2, y_3|x, s_1, s_2, s_3)p(x) = p(x)\prod_i^3 p(y_i|x, s_i)$.
So we set the unnormalised posterior density $\hat{p}_\star(x) = p(x)\prod_i^3 p(y_i|x, s_i)$.
The acceptance ratio $r(x, x') = \frac{\hat{p}_\star(x')q(x|x')}{\hat{p}_\star(x)q(x'|x)}$

But $N(x; x', \sigma_q) = N(x'; x, \sigma_q) = \frac{1}{\sigma_q\sqrt{2\pi}} e^{-\frac{(x'-x)^2}{2\sigma_q^2}}$ so these terms cancel as $(x' - x)^2 = (x - x')^2$.
The rest of the fraction results in:
$r(x, x') = e^{-\frac{1}{2\sigma_x^2}((x'-\mu_x)^2-(x-\mu_x)^2)-\sum_{i=1}^3 \frac{1}{2\sigma_x^2}((y_i-||x'-s_i||)^2-(y_i-||x-s_i||)^2)}$.

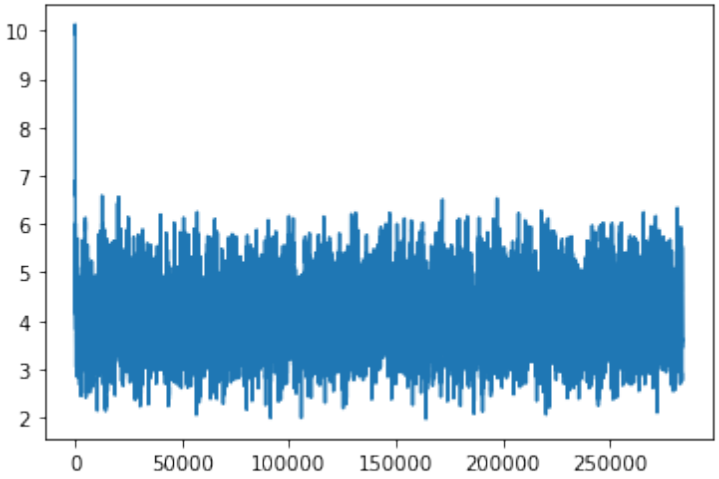
The algorithm is as follows:
Set N and X_0. For n in the range(1,N), set $X' \sim q(x'|X_{n-1}) = N(x'; x_{n-1}, \sigma_q^2)$.
Still within the loop, sample $U \sim Unif(0, 1)$ and accept $X_n = X'$ if $U <= r(X_{n-1}, X')$.
Otherwise, set $X_n = X_{n-1}$.
Finally, discard the first burnin samples.

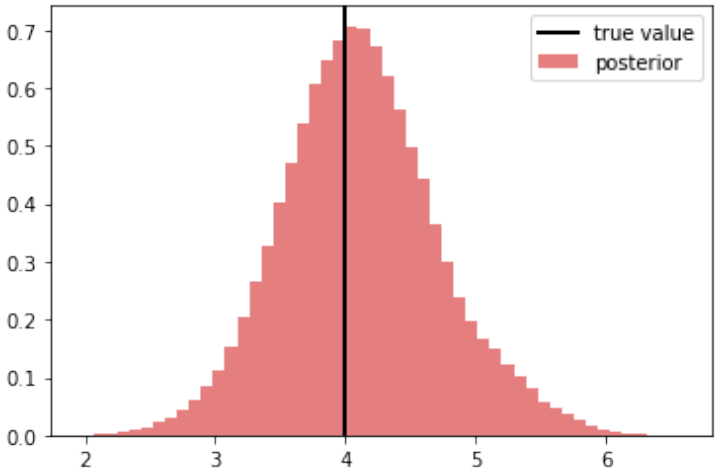
2.

```
s0 = -1
s1 = 2
s2 = 5
sigma_y = 1
sigma_x = 10
miu_x = 0
x_true = 4
y0 = 4.44
y1 = 2.51
y2 = 0.73
x0 = 10
N = 300000
sigma_q = 0.1
def q(x_dash,x,sigma = sigma_q):
    return 1/(sigma*np.sqrt(2*np.pi))*np.exp(-(x_dash-x)**2/(2*
                                        sigma**2))
def p(x, miu=miu_x, sigma=sigma_x):
    return 1/(sigma*np.sqrt(2*np.pi))*np.exp(-(x-miu)**2/(2*sigma**
```

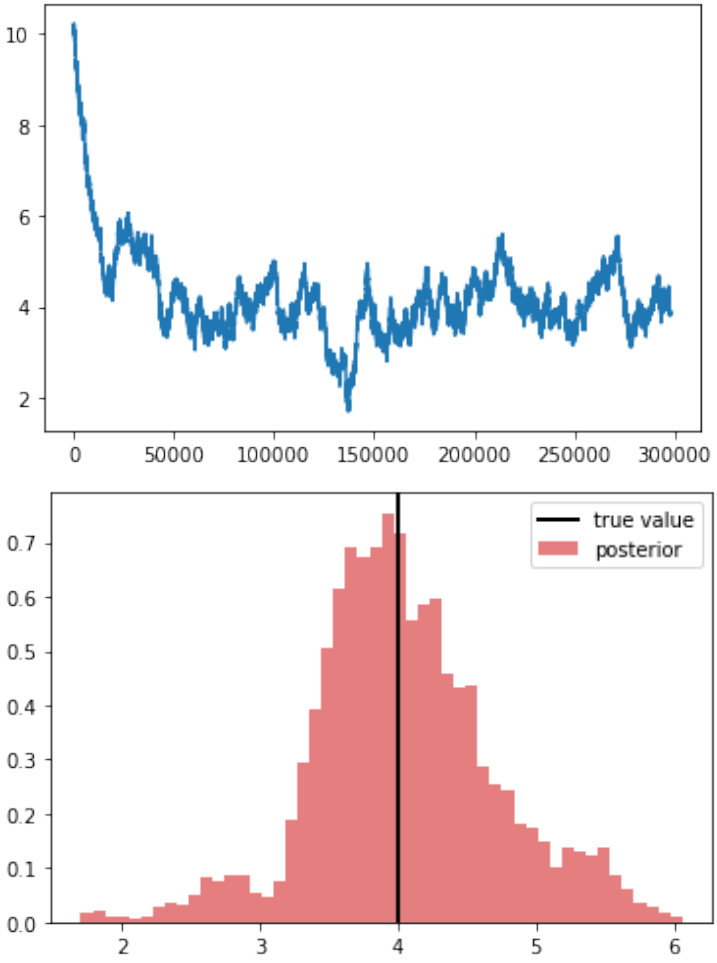
```
                                        2))
def pylikelihood(yi,x,si,sigma=sigma_y):
    return 1/(sigma*np.sqrt(2*np.pi))*np.exp(-(yi-np.linalg.norm(x-
                                        si))**2/(2*sigma**2))
def p_star(x):
    return p(x)*pylikelihood(y0,x,s0)*pylikelihood(y1,x,s1)*
                                        pylikelihood(y2,x,s2)
def r(x_dash,x):
    return (p_star(x_dash)*q(x,x_dash))/(p_star(x)*q(x_dash,x))
xs = np.array([])
x = x0
for i in range(N):
    x_dash = np.random.normal(x,sigma_q)
    u = np.random.uniform(0,1)
    if u <= r(x_dash,x):
        xs = np.append(xs,x_dash)
        x = x_dash
burnin = 120
plt.plot(xs)
#with sigma_y = 1
plt.clf()
# plot the true value vertically
plt.axvline(x_true , color='k', label='true value', linewidth=2)
plt.hist(xs[burnin:N], bins=50 , density=True , label='posterior',
                                    alpha=0.5, color=[0.8, 0, 0])
plt.legend()
plt.show ()

sigma_q = 0.01
xs = np.array([])
x = x0
for i in range(N):
    x_dash = np.random.normal(x,sigma_q)
    u = np.random.uniform(0,1)
    if u <= r(x_dash,x):
        xs = np.append(xs,x_dash)
        x = x_dash
burnin = 20000
plt.plot(xs)
plt.clf()
# plot the true value vertically
plt.axvline(x_true , color='k', label='true value', linewidth=2)
plt.hist(xs[burnin:N], bins=50 , density=True , label='posterior',
                                    alpha=0.5, color=[0.8, 0, 0])
plt.legend()
plt.show ()
```

When σ_q is 0.1, we obtain the following path and histogram:





When σ_q is 0.01, we get:



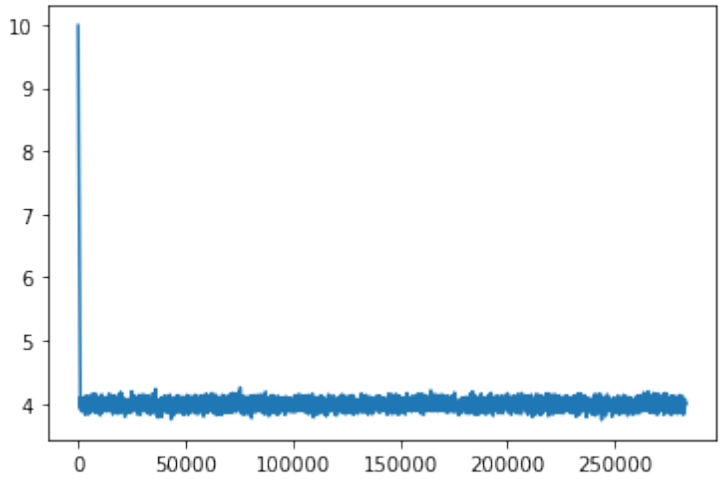
When the σ_q is decreased from 0.1 to 0.01, we need to set the burnin period much higher (e.g. 20000 vs 100) as the path takes much longer to get near the true value, as the jumps are smaller between each step. Also, for 0.1 the histogram resembles a very smooth normal distribution with mean at 4, whilst for 0.01 the histogram is not so symmetric as the path less consistently covers all the values around 4 as it has smaller jumps.

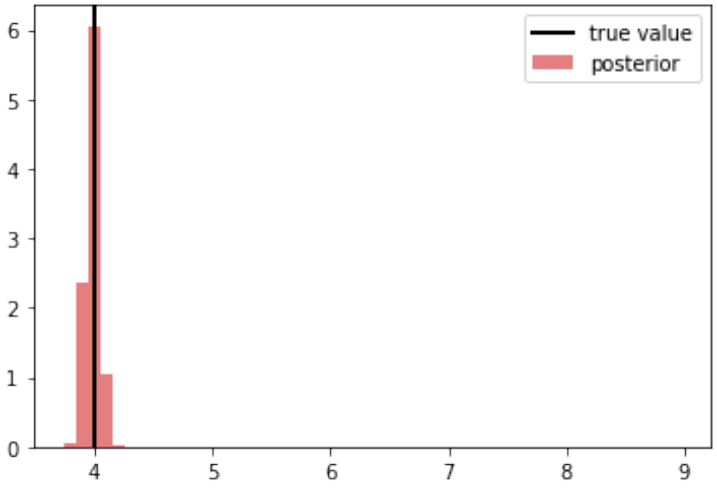
3.

```
sigma_y = 0.1
sigma_x = 10
miu_x = 0
x_true = 4
y0 = 5.01
```

```
y1 = 1.97
y2 = 1.02
x0 = 10
def rnew(x_dash,x):
    return np.exp(-1/(2*sigma_x**2)*((x_dash-miu_x)**2-(x-miu_x)**2
                                     )-1/(2*sigma_y**2)*((y0-abs(
                                     x_dash-s0))**2-(y0-abs(x-s0))
                                     **2)-1/(2*sigma_y**2)*((y1-
                                     abs(x_dash-s1))**2-(y1-abs(x-
                                     s1))**2)-1/(2*sigma_y**2)*((
                                     y2-abs(x_dash-s2))**2-(y2-abs
                                     (x-s2))**2))
xs = np.array([])
x = x0
for i in range(N):
    x_dash = np.random.normal(x,sigma_q)
    u = np.random.uniform(0,1)
    if u <= rnew(x_dash,x):
        xs = np.append(xs,x_dash)
        x = x_dash
plt.plot(xs)
plt.clf()
# plot the true value vertically
plt.axvline(x_true , color='k', label='true value', linewidth=2)
plt.hist(xs[burnin:N], bins=50 , density=True , label='posterior',
                                  alpha=0.5, color=[0.8, 0, 0])
plt.legend()
plt.show ()
```

Now with $\sigma_y = 0.1$ and $y_0 = 5.01, y_1 = 1.97, y_2 = 1.02$:





Reducing the variance of the likelihood to 0.1 results in the posterior having a much smaller variance, as we can see in the histogram, values are much more concentrated around 4, the true value. This because reducing the assumed variance of the input data reduces the variance in the acceptance ratio and hence of which values are accepted as samples. Changing the initial y values has no visible effect.