1 Q1

1.1

$$\begin{aligned} p(y) &= \mathcal{N}(y; 0, 2) \\ &= \frac{1}{2\sqrt{\pi}} e^{-\frac{y^2}{4}} \end{aligned}$$

$$\begin{aligned} p(9) &= \frac{1}{2\sqrt{\pi}} e^{-\frac{81}{4}} \\ &\approx 4.53 \times 10^{-10} \end{aligned}$$

1.2

We use the test function $p(y = 9 \mid x)$ and the estimator

$$\hat{\phi}^N \coloneqq \frac{1}{N} \sum_{i=1}^{N} p(y = 9 \mid X_i)$$

, where X_i are i.i.d. samples from $p(x)$

Array of simulated outputs = [4.49e-15, 4.39e-13, 1.33e-11, 3.52e-11, 2.57e-10]

The RAE for MC are [0.999990091401412, 0.9990311963862334, 0.9707366901001838, 0.9222107519259465, 0.43201889916565817] for N = [10, 100, 1000, 10000, 100000]

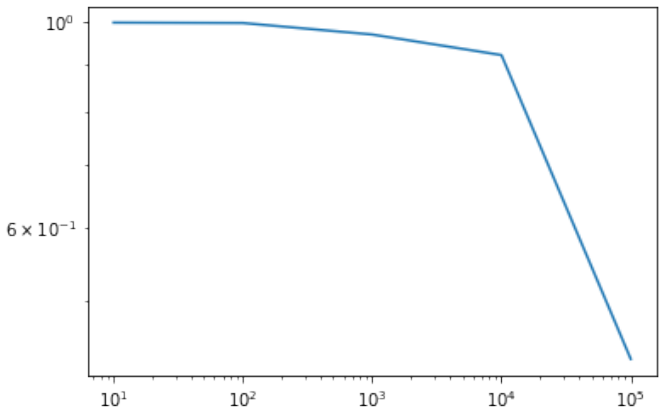


Figure 1: RAE for MC

1.3

The estimator is

$$\hat{\phi}_{IS}^{N} \coloneqq \frac{1}{N}\sum_{i=1}^{N} p(y=9 \mid X_i)\frac{p(X_i)}{q(X_i)}$$

, where X_i are i.i.d. samples from $q(x)$

Array of simulated outputs = [2.434927491105582e-10, 3.700687577412485e-10, 4.595926702830628e-10, 4.4601345753794163e-10, 4.4874102341308115e-10]

The RAE for IS are [0.4622824346554595, 0.18275812257411417, 0.014942139417032818, 0.015045534726342846, 0.0090221106734527 1] for N = [10, 100, 1000, 10000, 100000]

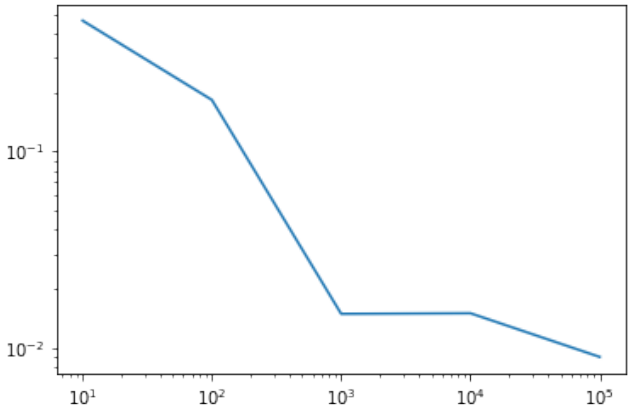
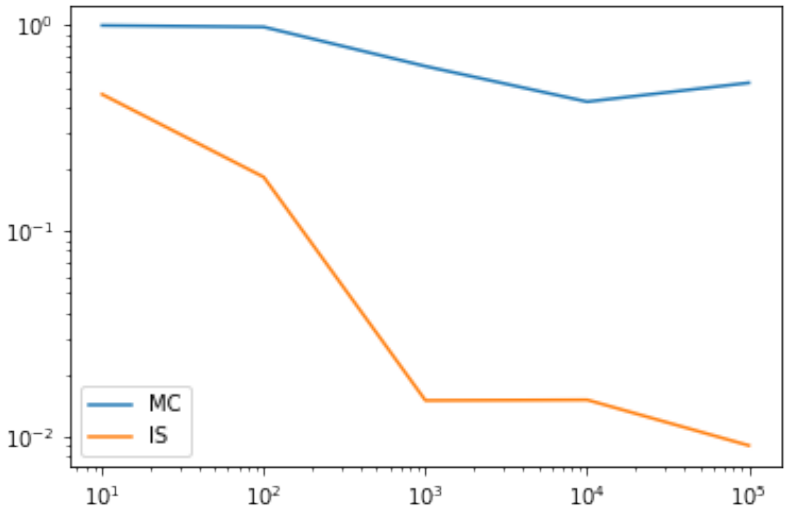


Figure 2: RAE for IS

1.4

We compare the log-log plots of the RAE.



We see that the graph for IS is consistently below that of MC, which shows that IS is a better estimator. We also see that the RAE for IS decreases a lot faster than MC on the plot, which means IS gives faster convergence to true value.

One possible explanation of this to note that the "mass" of the function $p(y = 9 \mid x)$ is concentrated near $x = 9$. Under the original distribution p used in MC, where we sample with mean 0, we are unlikely to sample from this region, leading to poor results. On the other hand, using IS, we sample with mean 6, making us more likely to reach the region.

2 Q2

2.1

In the model, we assume each sensor s_i can detect the distance of the object from the sensor ($||x - s_i||$) plus some Gaussian noise (independent of the location) with zero mean and standard deviation σ_y. We also assume a $\mathcal{N}(\mu_X, \sigma_X^2)$ prior for the location of the object.

We now briefly describe the MH algorithm: Pick a starting point x_0 and a proposal density $q(x \mid y)$. We proceed recursively: at step n, we sample a proposal X' from $q(x \mid x_{n-1})$ and compute the acceptance ratio as defined below, then accept the sample with probability $\min(1, r(x_{n-1}, X'))$. If we accept, we set $x_n \coloneqq X'$. Otherwise, $x_n \coloneqq x_{n-1}$.

We can set a burnin period and discard the first few samples for better convergence behaviour.

We consider the acceptance ratio

$$r(x, x') = \frac{\bar{p}_*(x')q(x \mid x')}{\bar{p}_*(x)q(x' \mid x)}$$

where $\bar{p}_*(x)$ is defined such that $p(x \mid y_{1:3}, s_{1:3}) \propto \bar{p}_*(x)$. We use the following choice of $\bar{p}_*(x)$

$$\begin{aligned}\bar{p}_*(x) &= p(y_{1:3} \mid x, s_{1:3})p(x|s_{1:3}) \\ &= p(x)\prod_{i=1}^{3} p(y_i \mid x, s_{1:3})\end{aligned}$$

, where the last equality is because the y_i are independent given x (and x is independent of s_i).

Now we show that the proposal is indeed symmetric so they cancel in the acceptance ratio:

$$\begin{aligned}q(x' \mid x) &= \frac{1}{\sigma_q\sqrt{2\pi}} \exp\left\{-\frac{1}{2}\left(\frac{x' - x}{\sigma_q}\right)^2\right\} \\ &= \frac{1}{\sigma_q\sqrt{2\pi}} \exp\left\{-\frac{1}{2}\left(\frac{x - x'}{\sigma_q}\right)^2\right\} \\ &= q(x \mid x')\end{aligned}$$

We compute the acceptance ratio, noting that for some constant C,

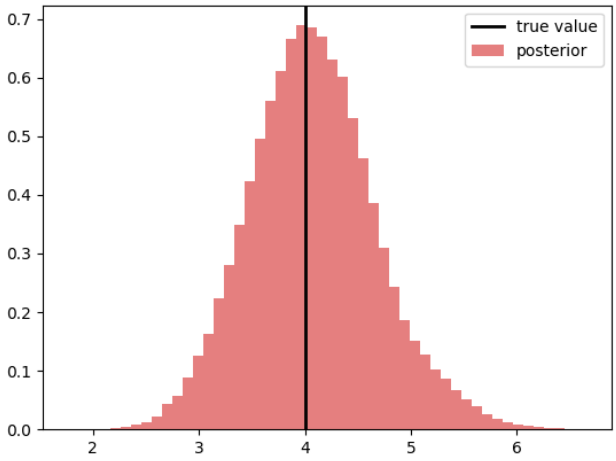
$$p(x)\prod_{i=1}^{3} p(y_i \mid x, s_{1:3}) = C \exp\left\{-\frac{1}{2\sigma_y^2}\sum_{i=1}^{3}(y_i - |x - s_i|)^2 - \frac{1}{2\sigma_x^2}x^2\right\}$$

$$\begin{aligned}r(x, x') &= \frac{\bar{p}_*(x')}{\bar{p}_*(x)} \\ &= \exp\left\{\frac{1}{2\sigma_y^2}\sum_{i=1}^{3}\left((y_i - |x - s_i|)^2 - (y_i - |x' - s_i|)^2\right) + \frac{1}{2\sigma_x^2}(x^2 - x'^2)\right\}\end{aligned}$$

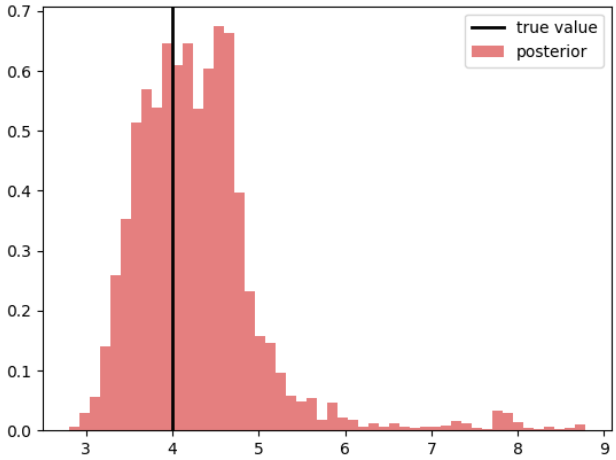
2.2 Implementation

We set $N = 200000$

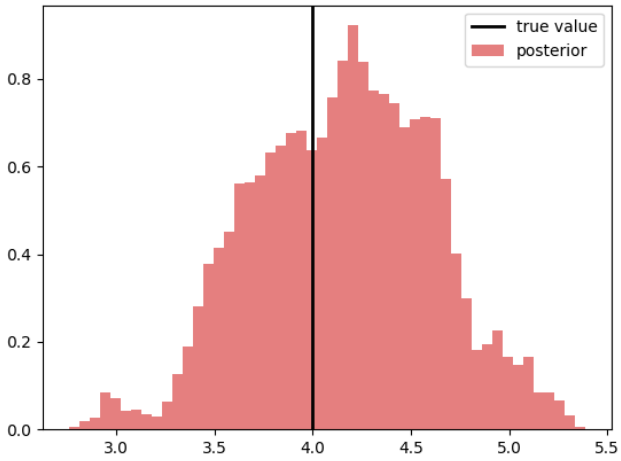
2.2.1 $\sigma_q = 0.1$, **burnin**= 1000



2.2.2 $\sigma_q = 0.01$, **burnin**= 1000



2.2.3 $\sigma_q = 0.01$, **burnin**= 50000

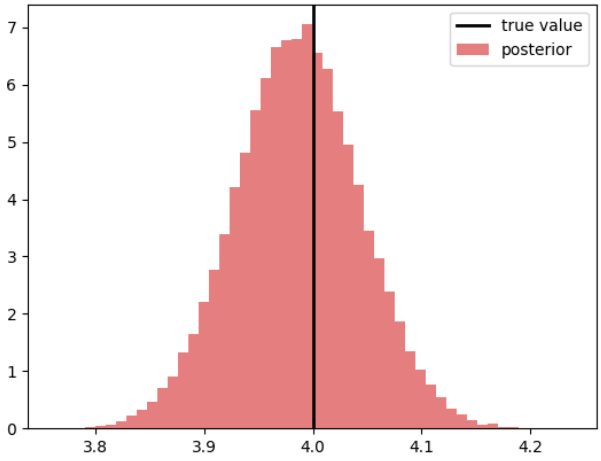


2.2.4 Difference

Firstly, we compare the burnin difference. Choosing burnin to be 1000 worked well for $\sigma_q = 0.1$ as we see the symmetry and "smoothness" of the graph. In contrast, using the same value of burnin gave a long tail on the right for $\sigma_q = 0.01$. This may be because smaller σ_q leads to less exploration and hence slower convergence, so more samples need to discarded and hence higher burnin is needed.

Next, we note that there are is quite a bit of uneveness for $\sigma_q = 0.01$. This may be because the sample have a higher chance to "concentrate" locally as opposed to exploring somewhere else due to the small σ_q.

2.3 Use new data



We see that the histogram is much narrower than the previous ones. This is because we have reduced σ_y greatly, meaning that our sensors are more accurate. Also, the observed data is much more closer to the "true" values 5, 2, and 1. These two gives us more certainty about the true location, hence the histogram is spikier.

A Q1 core

```
from math import fsum
import scipy.stats
import numpy as np
import matplotlib.pyplot as plt
import random

class Question_One:
    def __init__(self, arr):
        np.random.seed(123)
        self.true_val = scipy.stats.norm(0, 2**0.5).pdf(9)
        self.n_arr = arr

    def test_function(self, x, y):
        return scipy.stats.norm(x, 1).pdf(y)

    def mc_estimator(self, N):

        x_arr = np.random.normal(0, 1, size=N)
        test_arr = self.test_function(x_arr, 9)
        return np.sum(test_arr) / N

    def weight(self, x):
        return scipy.stats.norm(0, 1).pdf(x) / scipy.stats.norm(6,
                                            1).pdf(x)

    def is_estimator(self, N):
        x_arr = np.random.normal(6, 1, size=N)
        weighted_test_arr = self.test_function(x_arr, 9) * self.
                                            weight(x_arr)
        return sum(weighted_test_arr) / N

    def output_arr(self, func):
        self.out = [func(i) for i in self.n_arr]
        return self.out

    def _compute_RAE(self, arr):
        return [abs(1 - i / self.true_val) for i in arr]

    def RAE_arr(self, func, arr):
        out = self.output_arr(func)
        print(f"Array of simulated outputs = {out} \n")
        return self._compute_RAE(out)
```

B Q1 plots

```
N_arr = [10**i for i in range(1, 6)]
Q1 = Question_One(N_arr)

MC_RAE = Q1.RAE_arr(Q1.mc_estimator, N_arr)
print(f"The RAE for MC are {MC_RAE} for N = {N_arr}")
plt.loglog(N_arr, MC_RAE)
plt.savefig("CW2 Q1 plot1.png")
```

```
IS_RAE = Q1.RAE_arr(Q1.is_estimator, N_arr)
print(f"The RAE for IS are {IS_RAE} for N = {N_arr}")
plt.loglog(N_arr, IS_RAE)
plt.savefig("CW2 Q2 plot2.png")

plt.loglog(N_arr, MC_RAE, label="MC")
plt.loglog(N_arr, IS_RAE, label="IS")
plt.legend()
plt.savefig("CW2 Q1 plot3.png")
```

C Q2 core

```
from math import exp
import scipy.stats
import numpy as np
import matplotlib.pyplot as plt
import random
%matplotlib qt
class Question_Two:
    def __init__(self, data, x0, sd_q, sd_y, sd_x):
        self.sensors = [-1, 2, 5]
        self.x_true = 4

        self.data = data
        self.x0 = x0
        self.sd_q = sd_q
        self.sd_y = sd_y
        self.sd_x = sd_x

    def rejection_statistic(self, x):
        squared_diff = sum(((y - abs(x - s))**2 for y, s in zip(
                                        self.data, self.sensors))
                                        )
        return -squared_diff / (2 * self.sd_y**2) - x**2 / (2 *
                                        self.sd_x**2)

    def mh_sampling(self, N):
        prev = self.x0
        samples = [prev]
        for i in range(N):
            proposal = np.random.normal(prev, self.sd_q)
            numerator = self.rejection_statistic(proposal)
            denominator = self.rejection_statistic(prev)
            a = exp(self.rejection_statistic(proposal) - self.
                                            rejection_statistic(
                                            prev))
            if random.random() < a:
                prev = proposal
            samples.append(prev)
        return samples

    def plot_samples(self, x_s, burnin, N):
        plt.clf ()
```

```
        # plot the true value vertically
        plt.axvline(self.x_true , color='k', label='true value',
                                            linewidth=2)
        plt.hist(x_s[burnin:N], bins=50 , density=True , label='
                                            posterior',
        alpha=0.5, color=[0.8, 0, 0])
        plt.legend()
        plt.show ()

    def sample_and_plot(self, N, burnin):
        arr = self.mh_sampling(N)
        self.plot_samples(arr, burnin, N)
```

D Q2 plots

```
N = 200000

# sd_q is 0.1; burnin = 1000

Q2 = Question_Two([4.44, 2.51, 0.73], 10, 0.1, 1, 10)
Q2.sample_and_plot(N, 1000)

# sd_q is 0.01; burnin = 1000
Q2 = Question_Two([4.44, 2.51, 0.73], 10, 0.01, 1, 10)
Q2.sample_and_plot(N, 1000)

# sd_q is 0.01; burnin = 50000
Q2 = Question_Two([4.44, 2.51, 0.73], 10, 0.01, 1, 10)
Q2.sample_and_plot(N, 50000)

#  New data
Q2 = Question_Two([5.01, 1.97, 1.02], 10, 0.1, 0.1, 10)
Q2.sample_and_plot(N, 10000)
```