Coursework 2

Johannes Hübers,
CID 02358380

1 Main part

1.1 Q1

1.

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

Thus

$$p(y = 9) = \frac{1}{2\sqrt{\pi} e^{81/4}} \approx 4.528 \cdot 10^{-10}$$

2. The test function is

$$\varphi(x) = p(y = 9|x),$$

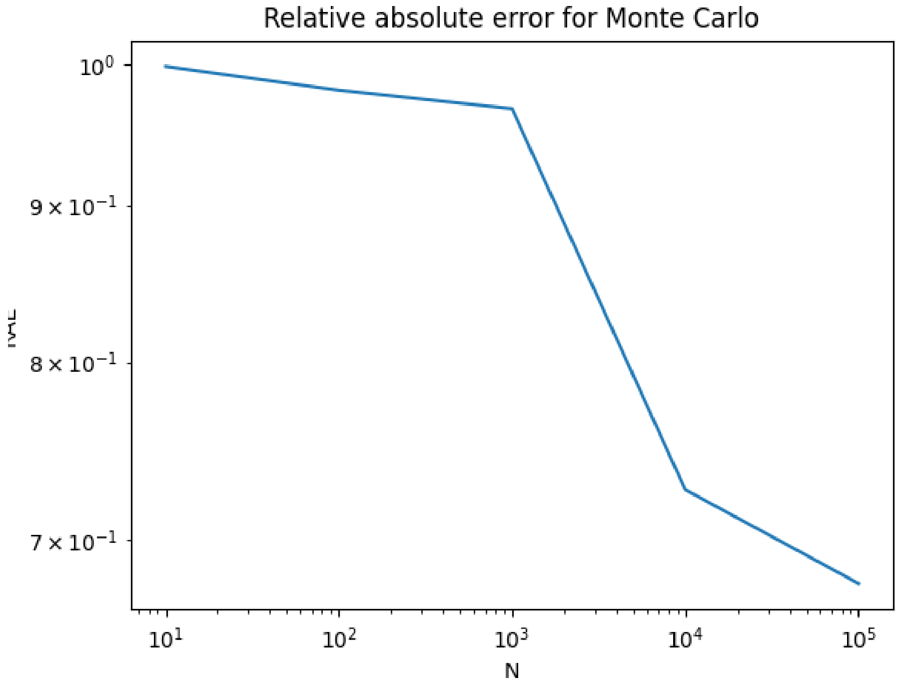
the likelihood function for $y = 9$. The estimator used is Monte Carlo integration,

$$\hat{\varphi}^N_{MC} = \frac{1}{N} \sum_{i=1}^{N} \varphi(X_i)$$

where the X_i are sampled from the distribution given by $p(x)$. In other words, Monte Carlo integration gives the average value of φ over N samples of the distribution. For large N this should be reasonably close to the actual integral

$$\bar{\varphi} = \int \varphi(x) p(x) dx.$$

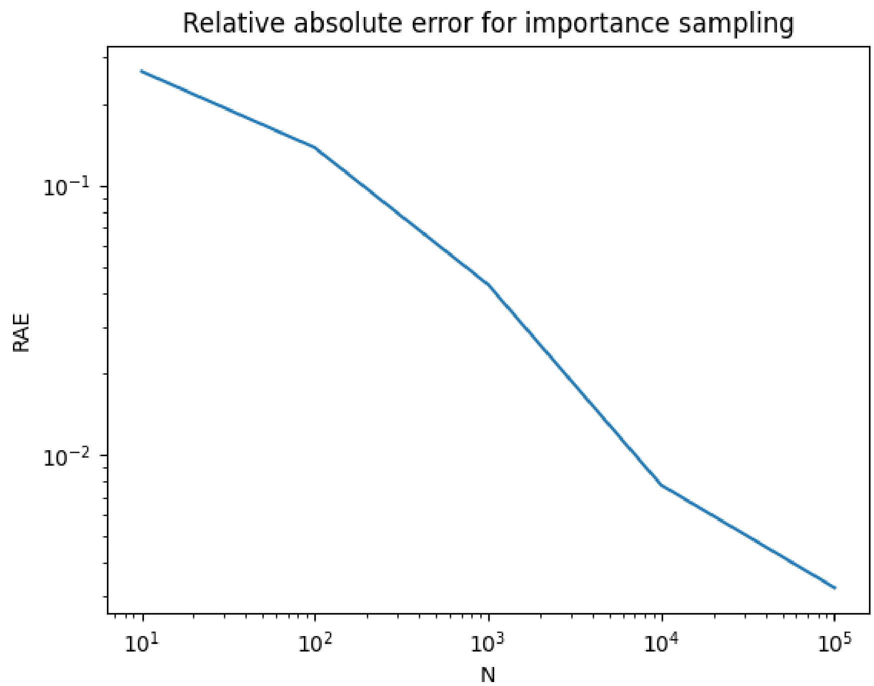
For Monte Carlo the relative absolute errors for $N = 10, 100, 1000, 10000, 100000$ were approximately $0.999, 0.981, 0.968, 0.727, 0.677$. For importance sampling they were approximately $0.264, 0.138, 0.043, 0.008, 0.003$. Clearly importance sampling was more accurate.



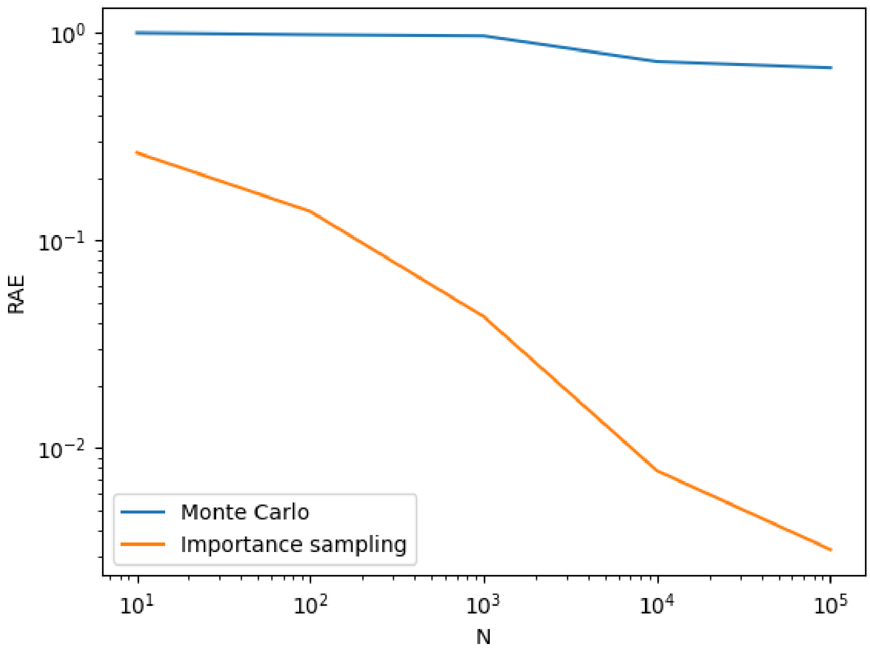
We see that the relative absolute error remains very high (above 67%) even after 100000 iterations. This is due to $y = 9$ being such an improbable event (more precisely, having very low density).

3. The IS estimator works by using a proposal $q(x)$ from which we sample many times. For each sample X_i, the value $\varphi(X_i)$ is weighted by $w_i = p(X_i)/q(X_i)$ and then the weighted sum is divided by the number of samples to find an average value of φ:

$$\hat{\varphi}_{IS}^N = \frac{1}{N} \sum_{i=1}^{N} w_i \varphi(X_i)$$



4. Plotting the RAE for Monte Carlo and importance sampling together, we see that importance sampling is much more accurate than Monte Carlo integration. This is because we sample x from the proposal $q(x)$ which is centered around 6, rather than 0 as is $p(x)$. 6 is of course much closer to 9 than 0 is, thus, given that the likelihood is centered around x, we get a more accurate estimate.



1.2 Q2

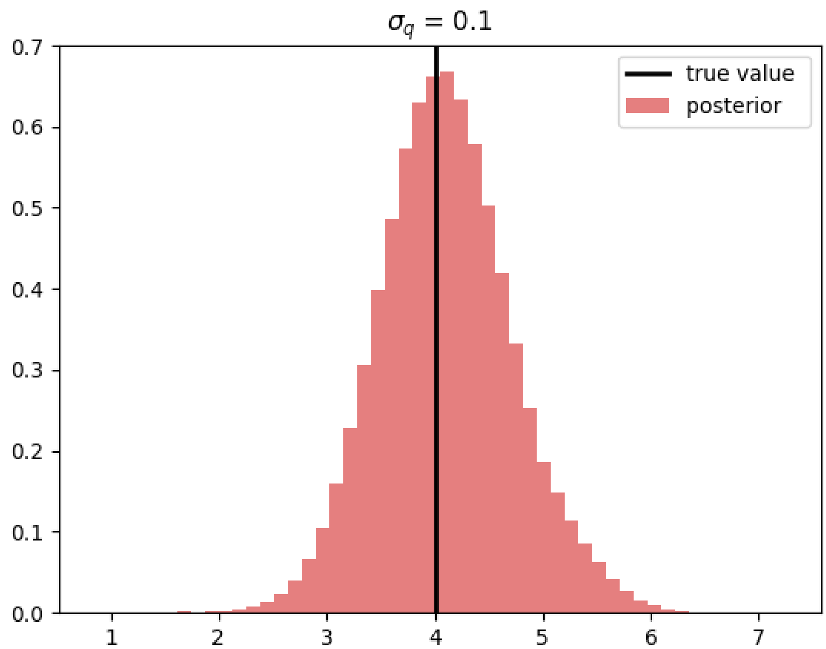
1. Using the symmetry $q(x|x') = q(x'|x)$ and Bayes' rule,

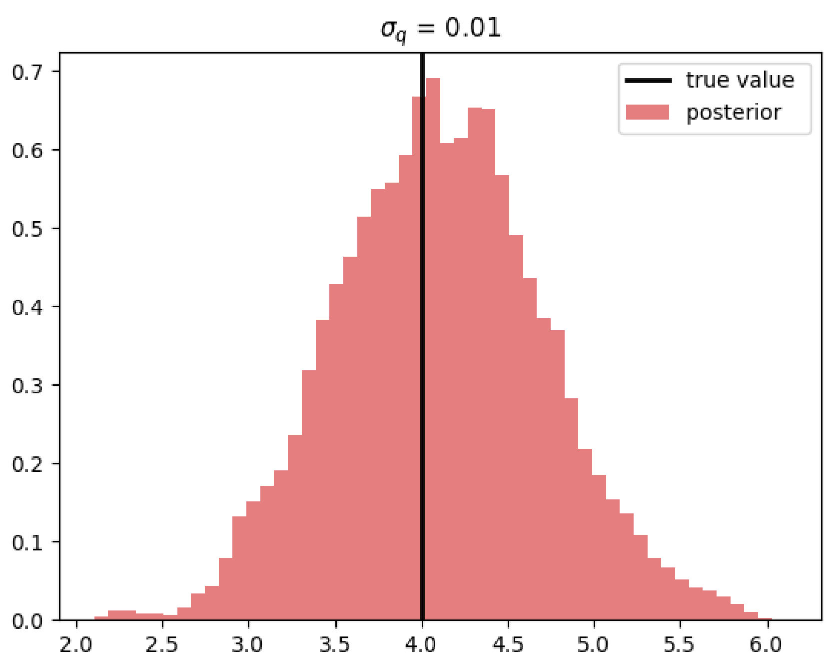
$$\begin{aligned} r(x,x') &= \frac{\bar{p}_*(x')q(x|x')}{\bar{p}_*(x)q(x'|x)} \\ &= \frac{p(x'|y_{1:3}, s_{1:3})}{p(x|y_{1:3}, s_{1:3})} \\ &= \frac{p(x')p(y_{1:3}|x', s_{1:3})/p(y_{1:3})}{p(x)p(y_{1:3}|x, s_{1:3})/p(y_{1:3})} \\ &= \frac{p(x')p(y_1|x', s_1)p(y_2|x', s_2)p(y_3|x', s_3)}{p(x)p(y_1|x, s_1)p(y_2|x, s_2)p(y_3|x, s_3)} \\ &= \frac{e^{-(x'-\mu_x)^2/(2\sigma_x^2)}e^{-((y_1-\|x'-s_1\|)^2+(y_2-\|x'-s_2\|)^2+(y_3-\|x'-s_3\|)^2)/(2\sigma_y^2)}}{e^{-(x-\mu_x)^2/(2\sigma_x^2)}e^{-((y_1-\|x-s_1\|)^2+(y_2-\|x-s_2\|)^2+(y_3-\|x-s_3\|)^2)/(2\sigma_y^2)}} \\ &= e^{((x-\mu_x)^2-(x'-\mu_x)^2)/(2\sigma_x^2)}e^{(\sum_{i=1}^3(y_i-\|x-s_i\|)^2-(y_i-\|x'-s_i\|)^2)/(2\sigma_y^2)} \end{aligned}$$

Given a distribution we want to sample from, the Metropolis–Hastings algorithm produces a Markov transition kernel for which this is the stationary distribution. For this it uses a local proposal $q(x'|x)$ to sample the next sample x' which is then either accepted (with probability $r(x,x')$) or rejected. In case of rejection, the preceding sample x is repeated.

2. The values for `burnin` were chosen by counting until x' first fell below $x_{\text{true}} = 4$. This led to `burnin` = 427 for $\sigma_q = 0.1$ and `burnin` = 27009 (significantly larger) for $\sigma_q = 0.01$. Running the program several times there seemed to be a relationship like `burnin` $\propto \sigma_q^{-2}$.

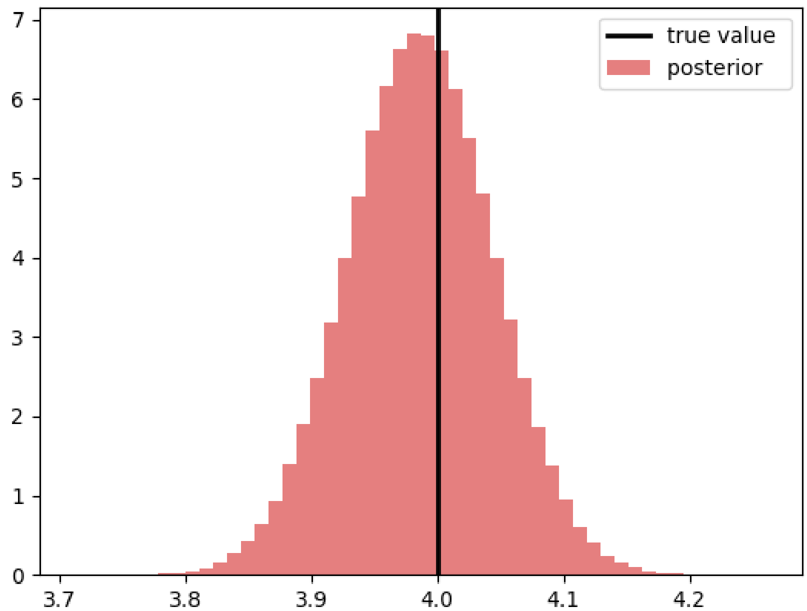
Here are the histograms ($N = 1000000$) for the samples:





3. Decreasing σ_y means the sensors are more accurate. This explains why the histogram for $\sigma_y = 0.1$ is much less wide than for $\sigma_y = 1$. The more accurate sensors mean that the estimate for x will be more accurate as well.

Again, `burnin` was chosen as the first index of iteration for which x' was below $x_{\text{true}} = 4$. For this particular execution of the program as in the histogram ($N = 1000000$) this was `burnin` = 143.



2 Appendix with code

2.1 Code for Q1

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

def phi(x):
    y = 9
    return 1/np.sqrt(2*np.pi) * np.exp(-((y-x)**2)/2)

def p(x):
    return 1/np.sqrt(2*np.pi) * np.exp(-1/2 * x**2)

def q(x):
    return 1/np.sqrt(2*np.pi) * np.exp(-1/2 * (x-6)**2)

phi_bar = 1/(2*np.sqrt(np.pi) * np.exp(81/4))
```

```
MC_estimates = []
MC_RAE = []
IC_estimates = []
IC_RAE = []
samples = np.random.normal(0,1,100000)
samples2 = np.random.normal(6,1,N)
for N in [10,100,1000,10000,100000]:
    estimate_MC = np.sum(phi(samples[0:N]))/N
    MC_estimates.append(estimate_MC)
    MC_RAE.append(abs(estimate_MC-phi_bar)/abs(phi_bar))

    estimate_IC = np.sum((p(samples2)/q(samples2)*phi(samples2))[0:N])/N
    IC_estimates.append(estimate_IC)
    IC_RAE.append(abs(estimate_IC-phi_bar)/abs(phi_bar))

print("phi_bar =", phi_bar)
print()

print("Monte Carlo")
print(MC_estimates)
print(MC_RAE)
print()

print("Importance sampling")
print(IC_estimates)
print(IC_RAE)

# plots
plt.plot([10,100,1000,10000,100000],MC_RAE)
plt.loglog()
plt.xlabel("N")
plt.ylabel("RAE")
plt.title("Relative absolute error for Monte Carlo")
plt.show()

plt.plot([10,100,1000,10000,100000],IC_RAE)
plt.loglog()
plt.xlabel("N")
plt.ylabel("RAE")
plt.title("Relative absolute error for importance sampling")
plt.show()

plt.plot([10,100,1000,10000,100000],MC_RAE,label="Monte Carlo")
plt.plot([10,100,1000,10000,100000],IC_RAE,label="Importance sampling")
plt.loglog()
plt.xlabel("N")
plt.ylabel("RAE")
plt.legend()
plt.show()
```

2.2 Code for Q2

2.

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

def alpha(x,x_prime):
    sigma_y = 1
    mu_x = 0
    sigma_x = 10
    y = [4.44, 2.51, 0.73]
    s = [-1, 2, 5]

    sum = 0
    for i in range(3):
        sum += ((y[i]-abs(x-s[i]))**2 - (y[i]-abs(x_prime-s[i]))**2)/(2*
                                                    sigma_y**2)
    return np.exp(((x-mu_x)**2 - (x_prime-mu_x)**2) / (2*sigma_x**2) +
                                                sum)

x_true = 4
N = 1000000
for sigma_q in [0.1,0.01]:
    x_0 = 10
    x_list = [x_0]

    burnin = -1
    for n in range(N):
        x = x_list[n]
        x_prime = np.random.normal(x,sigma_q)

        if x_prime < 4 and burnin == -1:
            print(n)
            burnin = n

        if alpha(x,x_prime) >= np.random.uniform():
            x_list.append(x_prime)
        else:
            x_list.append(x)

    plt.clf()
    plt.axvline (x_true, color='k', label='true value ', linewidth=2)
    plt.hist(x_list[burnin:N], bins=50 , density=True , label='posterior
                                        ', alpha=0.5, color=[0.8, 0, 0])
    plt.legend()
    plt.title("$\sigma_q$ = "+str(sigma_q))
    plt.show()
```

3.

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

```python
def alpha(x,x_prime):
    sigma_y = 0.1
    mu_x = 0
    sigma_x = 10
    y = [5.01,1.97,1.02]
    s = [-1, 2, 5]
    sum = 0
    for i in range(3):
        sum += ((y[i]-abs(x-s[i]))**2 - (y[i]-abs(x_prime-s[i]))**2)/(2*
                                                    sigma_y**2)
    return np.exp(((x-mu_x)**2 - (x_prime-mu_x)**2) / (2*sigma_x**2) +
                                                sum)

x_true = 4
N = 1000000
for sigma_q in [0.1]:
    x_0 = 10
    x_list = [x_0]

    burnin = -1
    for n in range(N):
        x = x_list[n]
        x_prime = np.random.normal(x,sigma_q)

        if x_prime < 4 and burnin == -1:
            print(n)
            burnin = n

        if alpha(x,x_prime) >= np.random.uniform():
            x_list.append(x_prime)
        else:
            x_list.append(x)

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