Stochastic Simulation - Coursework 2023

This assignment has two parts and graded over **100 marks**. Some general remarks:

- The assignment is due on **11 December 2023, 1PM GMT**, to be submitted via Blackboard (see the instructions on the course website).
- You should use this .ipynb file as a skeleton and you should submit a PDF report. Prepare the IPython notebook and export it as a PDF. If you can't export your notebook as PDF, then you can export it as HTML and then use the "Print" feature in browser (Chrome: File -> Print) and choose "Save as PDF".
- Your PDF should be no longer than 20 pages. But please be concise.
- You can reuse the code from the course material but note that this coursework also requires novel implementations. Try to personalise your code in order to avoid having problems with plagiarism checks. You can use Python's functions for sampling random variables of all distributions of your choice.
- **Please comment your code properly.**

Let us start our code.

In [ ]:
```
import matplotlib.pyplot as plt
import numpy as np

rng = np.random.default_rng(36) # You can change this.
```

Q1: Model Selection via Perfect Monte Carlo (40 marks)

Consider the following probabilistic model

$$p(x) = \mathcal{N}(x; 5, 0.01),$$
$$p(y_i|x) = \mathcal{N}(y_i; \theta x, 0.05),$$

for $i = 1, \ldots, T$ where y_i are conditionally independent given x. You are given a dataset (see it on Blackboard) denoted here as $y_{1:T}$ for $T = 100$. As defined in the course, we can find the marginal likelihood as

$$p_\theta(y_{1:T}) = \int p_\theta(y_{1:T}|x)p(x)\mathrm{d}x,$$

where we have left θ-dependence in the notation to emphasise that the marginal likelihood is a function of θ.

Given the samples from prior $p(x)$, we can identify our test function above as $\varphi(x) = p_\theta(y_{1:T}|x)$.

(i) The first step is to write a log-likelihood function of $y_{1:T}$, i.e., $p_\theta(y_{1:T}|x)$. Note that, this is the joint likelihood of conditionally i.i.d. observations y_i given x. This

function should take input the data set vector `y` as loaded from `y_perfect_mc.txt` below, θ (scalar), and x (scalar), and `sig` (likelihood variance which is given as 0.05 in the question but leave it as a variable) values to evaluate the log-likelihood. Note that log-likelihood will be a **sum** in this case, over individual log-likelihoods. **(10 marks)**

In [ ]:
```
# put the dataset in the same folder as this notebook
# the following line loads y_perfect_mc.txt
y = np.loadtxt('y_perfect_mc.txt')
y = np.array(y, dtype=np.float64)

# fill in your function below.

def log_likelihood(y, x, theta, sig): # fill in the arguments
    return np.sum(-np.log(np.sqrt(2*np.pi*sig))
                  - (y - theta*x)**2/(2*sig))

# uncomment and evaluate your likelihood (do not remove)
print(log_likelihood(y, 1, 1, 1))
print(log_likelihood(y, 1, 1, 0.1))
print(log_likelihood(y, -1, 2, 0.5))
```

```
-9898.905478066723
-98046.88084613332
-28974.21408410412
```

(ii) Write a logsumexp function. Let $\mathbf{v}$ be a vector of log-quantities and assume we need to compute $\log \sum_{i=1}^{N} \exp(v_i)$ where $\mathbf{v} = (v_1, \ldots, v_N)$. This function is given as

$$\log \sum_{i=1}^{N} \exp(v_i) = \log \sum_{i=1}^{N} \exp(v_i - v_{\max}) + v_{\max},$$

where $v_{\max} = \max_{i=1,\ldots,N} v_i$. Implement this as a function which takes a vector of log-values and returns the log of the sum of exponentials of the input values. **(10 marks)**

In [ ]:
```
def logsumexp(v):
    vmax = np.max(v) # find maximum of v
    return np.log(np.sum(np.exp(v - vmax))) + vmax # return sum

# uncomment and evaluate your logsumexp function (do not remove)
print(logsumexp(np.array([1, 2, 3])))
print(logsumexp(np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])))
print(logsumexp(np.array([5, 6, 9, 12])))
```

```
3.4076059644443806
10.45862974442671
12.051811977232925
```

(iii) Now we are at the stage of implementing the log marginal likelihood estimator. Inspect your estimator as described in Part (i). Its particular form is not implementable without using the trick you have coded in Part (iii). Now, implement a function that returns the **log** of the MC estimator you derived in Part (i). This function will take in

- `y` dataset vector
- θ parameter (scalar)
- `x_samples` (`np.array` vector) which are N Monte Carlo samples.
- a variance (scalar) variable `sig` for the joint log likelihood $p_\theta(y_{1:T}|x)$ that will be used in `log_likelihood` function (we will set this to 0.05 as given in the question).

Hint: Notice that the log of the MC estimator of the marginal likelihood takes the form

$$\log \frac{1}{N} \sum_{i=1}^{N} p_\theta(y_{1:T}|x^{(i)}),$$

as given in the question. You have to use $p_\theta(y_{1:T}|x^{(i)}) = \exp(\log p_\theta(y_{1:T}|x^{(i)}))$ to get a `logsumexp` structure, i.e., $\log$ and sum (over particles) and $\exp$ of $\log p_\theta(y_{1:T}|x^{(i)})$ where $i = 1, \ldots, N$ and $x^{(i)}$ are the N Monte Carlo samples (do **not** forget $1/N$ term too). Therefore, now use the function of Part (i) to compute $\log p_\theta(y_{1:T}|x^{(i)})$ for every $i = 1, \ldots, N$ and Part (ii) `logsumexp` these values to compute the estimate of log marginal likelihood. **(10 marks)**

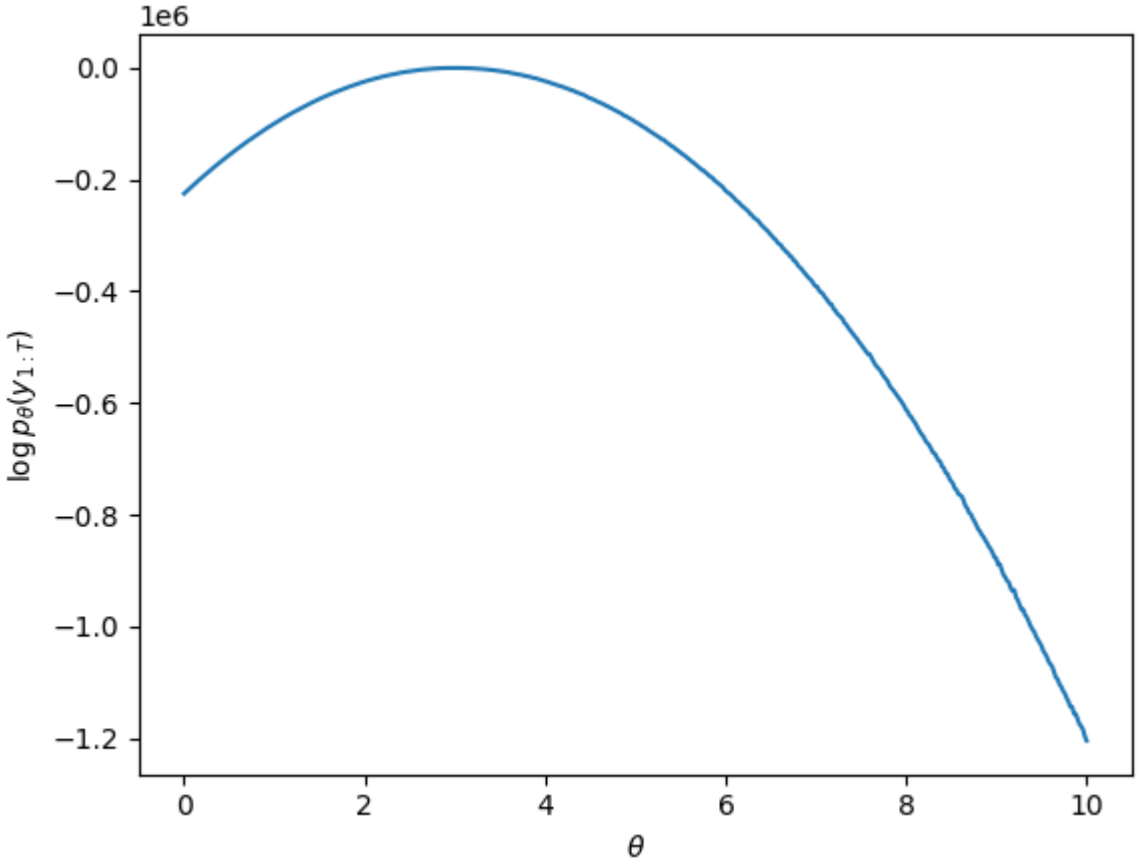
In [ ]:
```
def log_marginal_likelihood(y, theta, x_samples, sig): # fill in the argu
    N = len(x_samples)
    log_likelihoods = np.zeros(N) # initialise log likelihoods
    # for each sample, set corresponding log likelihood
    for i, x in enumerate(x_samples):
        log_likelihoods[i] = log_likelihood(y, x, theta, sig)
        # return logsumexp of likelihoods and return with constant
    return logsumexp(log_likelihoods) - np.log(N)

# uncomment and evaluate your marginal likelihood (do not remove)

print(log_marginal_likelihood(y, 1, np.array([-1, 1]), 1))
print(log_marginal_likelihood(y, 1, np.array([-1, 1]), 0.1))
print(log_marginal_likelihood(y, 2, np.array([-1, 1]), 0.5))

# note that the above test code takes 2 dimensional array instead of N pa
```

```
-9899.598625247283
-98047.57399331388
-16970.96811085916
```

(iv) We will now try to find the most likely θ. For this part, you will run your `log_marginal_likelihood` function for a range of θ values. Note that, for every θ value, you need to sample N new samples from the prior (do not reuse the same samples). Compute your estimator of the $\log \hat{\pi}^N_{\mathrm{MC}} \approx \log p_\theta(y_{1:T})$ for θ-range given below. Plot the log marginal likelihood estimator as a function of θ. **(5 marks)**

In [ ]:
```
sig = 0.05
sig_prior = 0.01
mu_prior = 5.0

N = 1000
```

```
theta_range = np.linspace(0, 10, 500)
log_ml_list = np.array([]) # you can use np.append to add elements to thi

# fill in your code here
for theta in theta_range:
    x_samples = rng.normal(mu_prior, sig_prior, N)
    log_ml_list = np.append(log_ml_list, log_marginal_likelihood(y,
                            theta, x_samples, sig))
# uncomment and plot your results (do not remove)

plt.plot(theta_range, log_ml_list)
plt.xlabel(r'$\theta$')
plt.ylabel(r'$\log p_\theta(y_{1:T})$')
plt.show()
```



(v) Now you have `log_ml_list` variable that corresponds to marginal likelihood values in `theta_range`. Find the θ value that gives the maximum value in this list and provide your final estimate of most likely θ. **(5 marks)**

In [ ]:
```
# You code goes here
theta_est = theta_range[np.argmax(log_ml_list)]
# find index of maximum in log_ml_list and find theta value corresponding

# print your theta estimate, e.g.:
print(theta_est)
```

```
3.006012024048096
```

Q2: Posterior sampling (35 marks)

In this question, we will perform posterior sampling for the following model

$$p(x) \propto \exp(-x_1^2/10 - x_2^2/10 - 2(x_2 - x_1^2)^2),$$
$$p(y|x) = \mathcal{N}(y; Hx, 0.1)$$

where $H = [0, 1]$. In this exercise, we assume that we have observed $y = 2$ and would like to implement a few sampling methods.

Before starting this exercise, please try to understand how the posterior density should look like. The discussion we had during the lecture about Exercise 6.2 (see Panopto if you have not attended) should help you here to understand the posterior density. Note though quantities and various details are **different** here. You should have a good idea about the posterior density before starting this exercise to be able to set the hyperparameters such as the chain-length, proposal noise, and the step-size.

In [ ]:
```
y = np.array([2.0])
sig_lik = 0.1
H = np.array([0, 1])
```

(i) In what follows, you will have to code the log-prior and log-likelihood functions. Do **not** use any library, code the log densities directly. **(5 marks)**

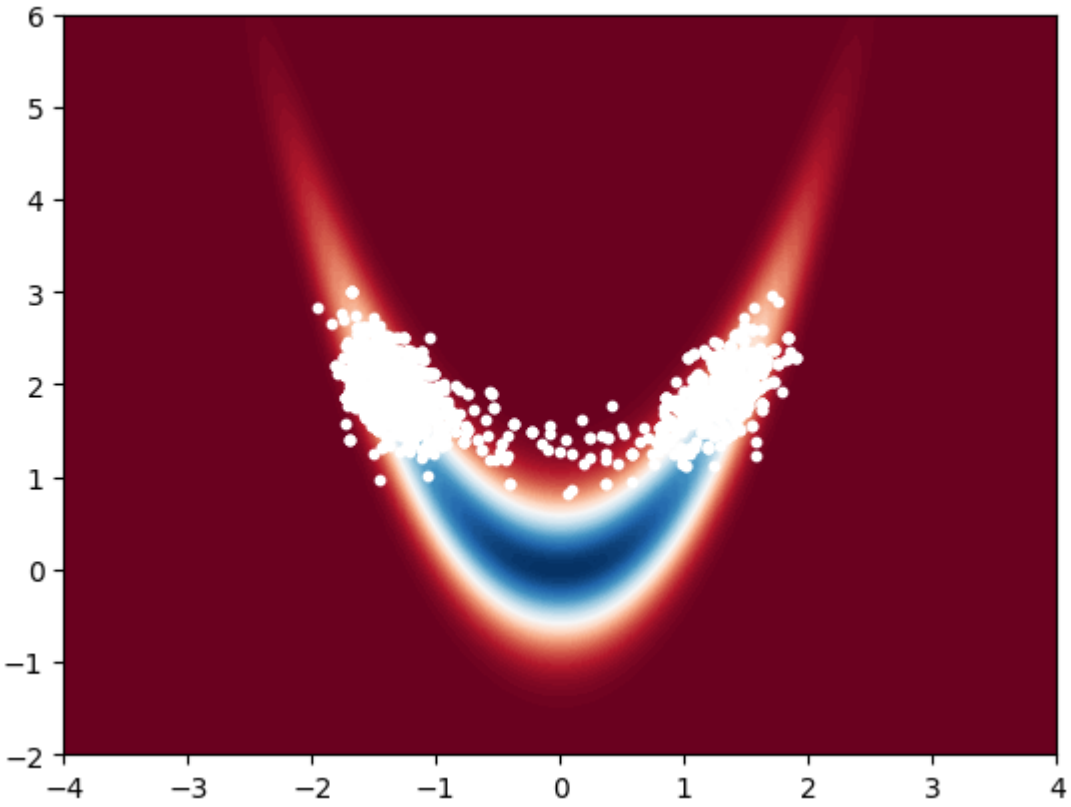
In [ ]:
```
def prior(x): # code banana density for visualisation purposes
    return np.exp(-x[0]**2/10 - x[1]**2/10 - 2*(x[1] - x[0]**2)**2)

def log_prior(x): # fill in the arguments
    return -x[0]**2/10 - x[1]**2/10 - 2*(x[1] - x[0]**2)**2

def log_likelihood(y, x, sig_lik): # fill in the arguments
    H = np.array([0, 1])
    return - np.log(np.sqrt(2*np.pi*sig_lik)) - np.sum(
        (y - H@x)**2)/(2*sig_lik)

# uncomment below and evaluate your prior and likelihood (do not remove)
print(log_prior([0, 1]))
print(log_likelihood(y, np.array([0, 1]), sig_lik))
```

```
-2.1
-4.76764598670765
```

(ii) Next, implement **the random walk Metropolis algorithm (RWMH)** for this target. Set an appropriate chain length, proposal variance, and `burnin` value. Plot a scatter-plot with your samples (see the visualisation function below). Use log-densities only. **(10 marks)**

In [ ]:
```
# fill in your code here
N = 2000
sig_q = 0.1
x_RW = np.zeros((N, 2))
# initialise normal and uniform variables
W = rng.normal(0, 1, (N, 2))
u = rng.uniform(0, 1, N)
```

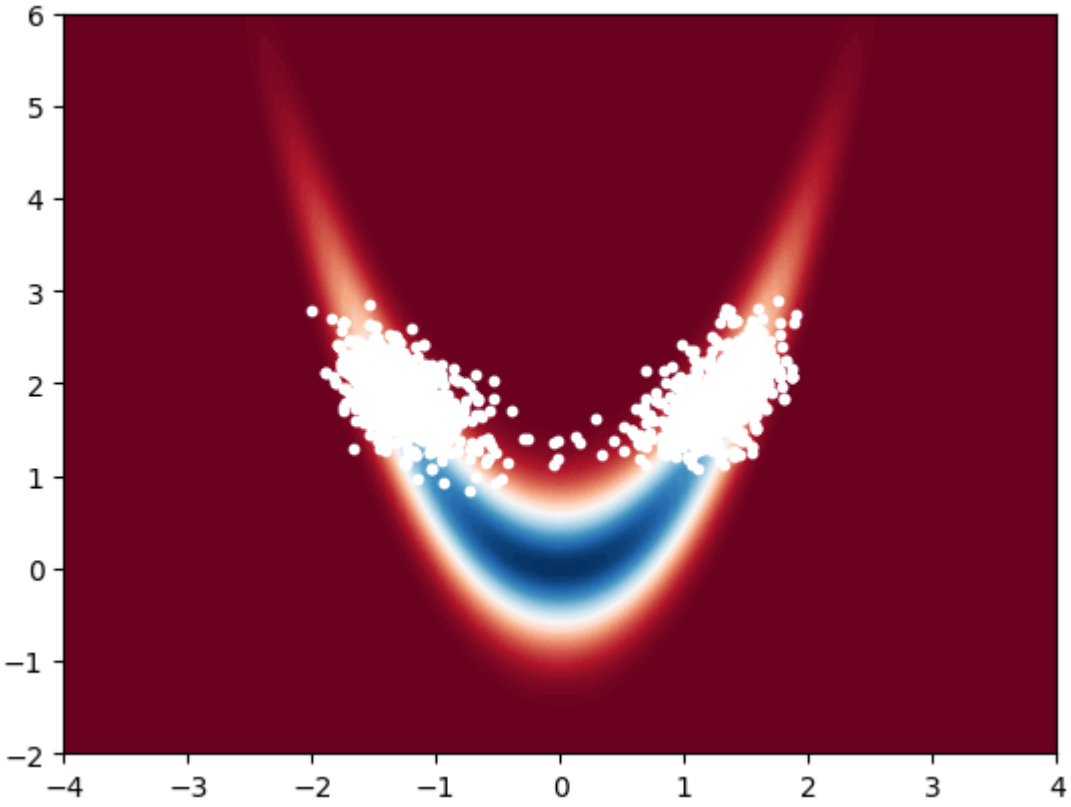
```
x_RW[0, :] = np.array([0, 0]) # initialise initial conditions

for n in range(1, N):
    # take sample with appropriate mean and variance
    x_sample = x_RW[n-1, :] + np.sqrt(sig_q) * W[n-1]
    # accept with acceptance ratio
    if np.log(u[n-1]) < log_prior(x_sample) + log_likelihood(
        y, x_sample, sig_lik) - log_prior(x_RW[n-1, :]) - log_likelihood(
            y, x_RW[n-1, :], sig_lik):
        x_RW[n, :] = x_sample
    # otherwise reuse old sample
    else:
        x_RW[n, :] = x_RW[n-1, :]

burnin = 50
# uncomment and plot your results (do not remove)
x_bb = np.linspace(-4, 4, 100)
y_bb = np.linspace(-2, 6, 100)
X_bb , Y_bb = np.meshgrid(x_bb , y_bb)
Z_bb = np.zeros((100 , 100))
for i in range(100):
    for j in range(100):
        Z_bb[i, j] = prior([X_bb[i, j], Y_bb[i, j]])
plt.contourf(X_bb , Y_bb , Z_bb , 100 , cmap='RdBu')
plt.scatter(x_RW[burnin:, 0], x_RW[burnin:, 1], s=10 , c='white')
plt.show()

# Note that above x vector (your samples) is assumed to be (N, 2).
# It does not have to be this way (You can change the name of the variabl
# i.e., If your x vector is (2, N), then use
#plt.scatter(x[0, :], x[1, :], s=10 , c='white')
# instead of
# plt.scatter(x[:, 0], x[:, 1], s=10 , c='white')
# in the above code.
```



(iii) Now implement **Metropolis-adjusted Langevin algorithm**. For this, you will need to code the gradient of the density and use it in the proposal as described in the lecture notes. Set an appropriate chain length, step-size, and `burnin` value. Plot a scatter-plot with your samples (see the visualisation function below). Use log-densities only. **(10 marks)**

In [ ]:
```
def grad_log_prior(x): # fill in the arguments
    return np.array([-x[0]/5 + 8 * x[0] * (x[1] - x[0]**2),
                     -x[1]/5 - 4 * (x[1] - x[0]**2)])

def grad_log_likelihood(x, y, sig_lik): # fill in the arguments
    return H * (y - H@x)/sig_lik

def log_MALA_kernel(x_sample, x, gamma, y): # fill in the arguments
    return - np.linalg.norm(x_sample - x - gamma * (grad_log_prior(
        x) + grad_log_likelihood(x, y, sig_lik)))**2 /(
            4 * gamma) - np.log(4 * np.pi * gamma)

gam = 0.04

# fill in your code here
N = 2000
x_L = np.zeros((N, 2))
x_L[0, :] = np.array([0, 0])

sig_L = np.sqrt(2 * gam)
# initialise normal and uniform random variables
W = rng.normal(0, 1, (N, 2))
u = rng.uniform(0, 1, N)

for n in range(1, N):
    # take sample from MALA proposal
    x_sample = x_L[n-1, :] + gam * (grad_log_prior(x_L[n-1, :])
         + grad_log_likelihood(x_L[n-1, :], y, sig_lik)) + sig_L * W[n-1]
    # accept with acceptance ratio
    if np.log(u[n-1]) < log_prior(x_sample) + log_likelihood(
                        y, x_sample, sig_lik) + log_MALA_kernel(
                        x_L[n-1, :], x_sample, gam, y) - log_likelihood(
                        y, x_L[n-1, :], sig_lik) - log_prior(
                        x_L[n-1, :]) - log_MALA_kernel(
                        x_sample, x_L[n-1, :], gam, y):
        x_L[n, :] = x_sample
    else: # otherwise reuse previous sample
        x_L[n, :] = x_L[n-1, :]



burnin = 50
# uncomment and plot your results (do not remove)
x_bb = np.linspace(-4, 4, 100)
y_bb = np.linspace(-2, 6, 100)
X_bb , Y_bb = np.meshgrid(x_bb , y_bb)
Z_bb = np.zeros((100 , 100))
for i in range(100):
    for j in range(100):
        Z_bb[i, j] = prior([X_bb[i, j], Y_bb[i, j]])
plt.contourf(X_bb , Y_bb , Z_bb , 100 , cmap='RdBu')
plt.scatter(x_L[burnin:, 0], x_L[burnin:, 1], s=10 , c='white')
```

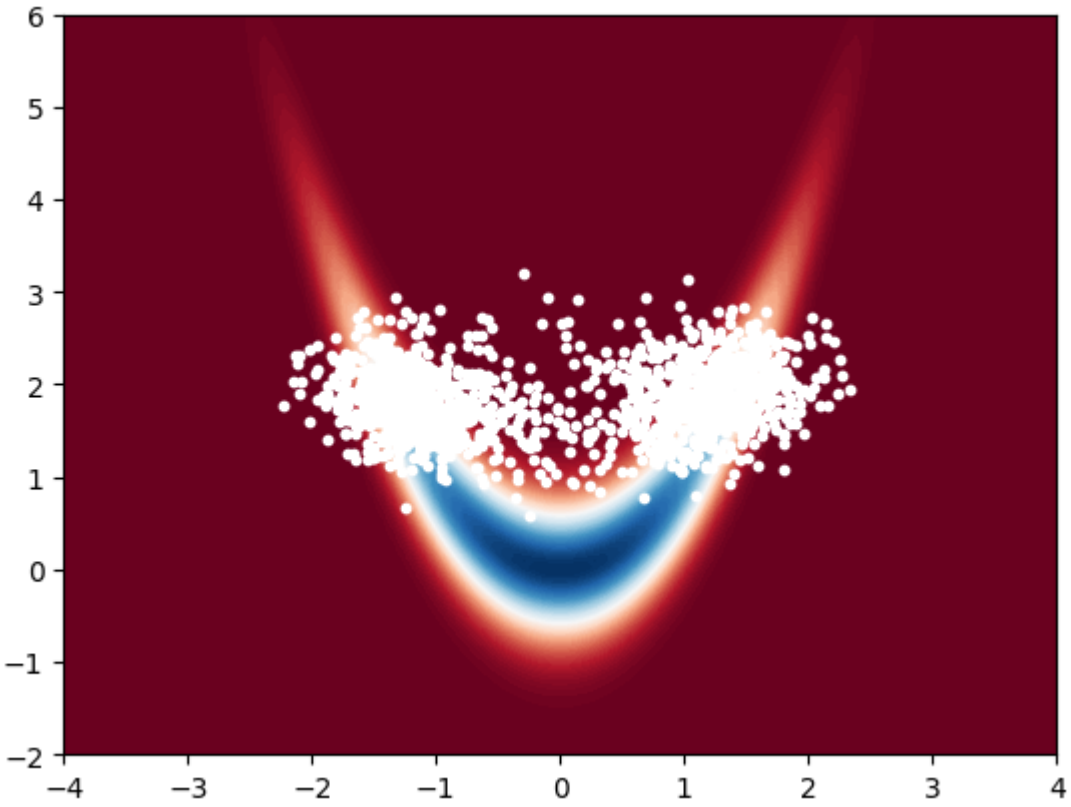
```
plt.show()

# Note that above x vector (your samples) is assumed to be (N, 2).
# It does not have to be this way (You can change the name of the variabl
# i.e., If your x vector is (2, N), then use
# plt.scatter(x[0, :], x[1, :], s=10 , c='white')
# instead of
# plt.scatter(x[:, 0], x[:, 1], s=10 , c='white')
# in the above code.
```



(iv) Next, implement **unadjusted Langevin algorithm**. For this, you will need to code the gradient of the density and use it in the proposal as described in the lecture notes. Set an appropriate chain length, step-size, and `burnin` value. Plot a scatter-plot with your samples (see the visualisation function below). Use log-densities only. **(10 marks)**

In [ ]:
```
# fill in your code here
N = 2000
gam = 0.04
x_ULA = np.zeros((N, 2))

x_ULA[0, :] = np.array([0, 0])

W = rng.normal(0, 1, (N, 2)) # initialise normal random variables

for n in range(1, N):
    x_ULA[n, :] = x_ULA[n-1, :] + gam * (grad_log_prior(x_ULA[n-1, :])
         + grad_log_likelihood(x_ULA[n-1, :], y, sig_lik)) + np.sqrt(
             2 * gam) * W[n-1] # update using SDE formula


burnin = 500 # large burnin as it converges slowly
# uncomment and plot your results (do not remove)
```

```
x_bb = np.linspace(-4, 4, 100)
y_bb = np.linspace(-2, 6, 100)
X_bb , Y_bb = np.meshgrid(x_bb , y_bb)
Z_bb = np.zeros((100 , 100))
for i in range(100):
    for j in range(100):
        Z_bb[i, j] = prior([X_bb[i, j], Y_bb[i, j]])
plt.contourf(X_bb , Y_bb , Z_bb , 100 , cmap='RdBu')
plt.scatter(x_ULA[burnin:, 0], x_ULA[burnin:, 1], s=10 , c='white')
plt.show()

# Note that above x vector (your samples) is assumed to be (N, 2).
# It does not have to be this way (You can change the name of the variabl
# i.e., If your x vector is (2, N), then use
# plt.scatter(x[0, :], x[1, :], s=10 , c='white')
# instead of
# plt.scatter(x[:, 0], x[:, 1], s=10 , c='white')
# in the above code.
```



In this question, you will first derive a Gibbs sampler by deriving full conditionals. Then we will describe a method to estimate marginal likelihoods using Gibbs output (and you will be asked to implement the said method given the description).

Consider the following probabilistic model

$$\begin{aligned} p(x_1) &= \mathcal{N}(x_1; \mu_1, \sigma_1^2), \\ p(x_2) &= \mathcal{N}(x_2; \mu_2, \sigma_2^2), \\ p(y|x_1, x_2) &= \mathcal{N}(y; x_1 + x_2, \sigma_y^2), \end{aligned}$$

where y is a scalar observation and x_1, x_2 are latent variables. This is a simple model where we observe a sum of two random variables and want to construct possible values of x_1, x_2 given the observation y.

(i) Derive the Gibbs sampler for this model, by deriving full conditionals $p(x_1|x_2, y)$ and $p(x_2|x_1, y)$ (You can use Example 3.2 but note that this case is different). **(10 marks)**

Using Bayes' Rule and assuming independence of latent variables x_1 and x_2:

$$\begin{aligned} p(x_1|x_2, y) &= \frac{p(y, x_1, x_2)}{p(x_2, y)} \\ &= \frac{p(y|x_1, x_2)p(x_1, x_2)}{p(y, x_2)} \\ &\propto p(y|x_1, x_2)p(x_1)p(x_2) \\ &\propto p(y|x_1, x_2)p(x_1) \end{aligned}$$

Similarly, for x_2, we find:

$$p(x_2|x_1, y) \propto p(y|x_1, x_2)p(x_2)$$

Expanding these densities we find:

$$\begin{aligned} p(x_1|x_2, y) &\propto p(y|x_1, x_2)p(x_1) \\ &\propto \exp\left\{ -\frac{(y - (x_1 + x_2))^2}{2\sigma_y^2} \right\} \exp\left\{ -\frac{(x_1 - \mu_1)^2}{2\sigma_1^2} \right\} \\ &= \exp\left\{ -\frac{\sigma_1^2(y - (x_1 + x_2))^2 + \sigma_y^2(x_1 - \mu_1)^2}{2\sigma_1^2\sigma_y^2} \right\} \\ &\propto \exp\left\{ -\frac{(\sigma_1^2 + \sigma_y^2)x_1^2 + (2\sigma_1^2 x_2 - 2\sigma_1^2 y - 2\sigma_y^2\mu_1)x_1}{2\sigma_1^2\sigma_y^2} \right\} \\ &\propto \exp\left\{ -\frac{(\sigma_1^2 + \sigma_y^2)\left(x_1 - \frac{\sigma_1^2 y + \sigma_y^2\mu_1 - \sigma_1^2 x_2}{\sigma_1^2 + \sigma_y^2}\right)^2}{2\sigma_1^2\sigma_y^2} \right\} \end{aligned}$$

This can be recognised as a Gaussian:

$$\mu_{p_1} = \frac{\sigma_1^2(y - x_2) + \sigma_y^2\mu_1}{\sigma_1^2 + \sigma_y^2}$$

$$\sigma_{p_1} = \frac{\sigma_1^2\sigma_y^2}{\sigma_1^2 + \sigma_y^2}$$

Similarly, we can recognise the density $p(x_2|x_1, y)$ as a Gaussian with:

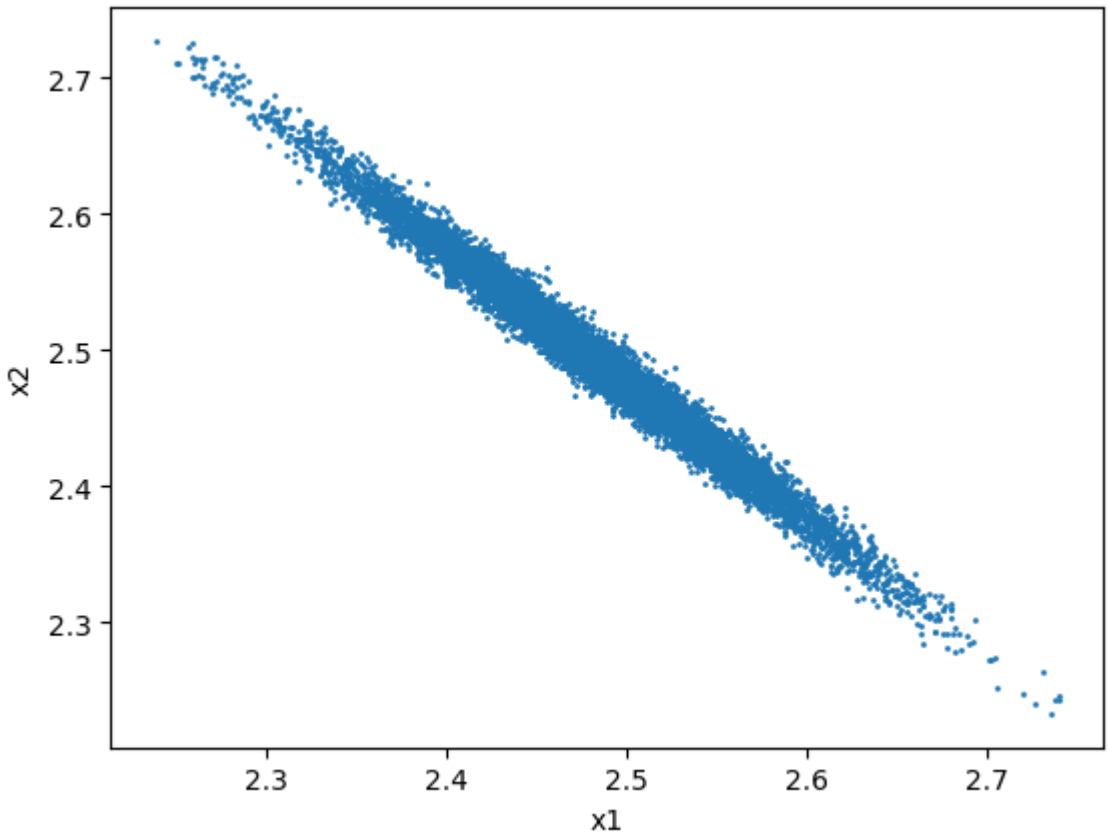
$$\mu_{p_2} = \frac{\sigma_2^2(y - x_1) + \sigma_y^2\mu_2}{\sigma_2^2 + \sigma_y^2}$$

$$\sigma_{p_2} = \frac{\sigma_2^2\sigma_y^2}{\sigma_2^2 + \sigma_y^2}$$

(ii) Let us set $y = 5$, $\mu_1 = 0$, $\mu_2 = 0$, $\sigma_1 = 0.1$, $\sigma_2 = 0.1$, and $\sigma_y = 0.01$.

Implement the Gibbs sampler you derived in Part (i). Set an appropriate chain length and `burnin` value. Plot a scatter plot of your samples (see the visualisation function below). Discuss the result: Why does the posterior look like this? **(15 marks)**

In [ ]:
```
y = 5

mu1 = 0
mu2 = 0
sig1 = 0.1
sig2 = 0.1

sig_y = 0.01

# fill in your code here for Gibbs sampling
# initialise chain and initial conditions
N = 10000
x_gibbs = np.zeros((N, 2))
x_gibbs[0, :] = np.array([0, 0])
# fix variances as they don't change
sig_p_1 = sig1*sig_y/np.sqrt(sig1**2 + sig_y**2)
sig_p_2 = sig2*sig_y/np.sqrt(sig2**2 + sig_y**2)

W = rng.normal(0, 1, (N, 2))

for n in range(1, N):
    x_gibbs[n, 0] = (sig1**2*(y - x_gibbs[n-1, 1]) + sig_y**2*mu1)/(
        sig1**2 + sig_y**2) + sig_p_1*W[n-1, 0] # update x1 with derived
    x_gibbs[n, 1] = (sig2**2*(y - x_gibbs[n, 0]) + sig_y**2*mu2)/(
        sig2**2 + sig_y**2) + sig_p_2*W[n-1, 1] # update x2 with derived

burnin = 200
# uncomment and plot your results (do not remove)
plt.scatter(x_gibbs[burnin:, 0], x_gibbs[burnin:, 1], s=1)
plt.xlabel("x1")
plt.ylabel("x2")
plt.show()
```



From the figure, we can see the samples form around the line $x_1 + x_2 = 5$. The variance orthogonal to this line is small as opposed to the variance along the line, which is clearly much larger. We can see that the samples are centered around the point $(x_1, x_2) = (2.5, 2.5)$. This comes as a result of the likelihood function, which gives rise to the expression $y = x_1 + x_2 + \sigma_y W$ where $W \sim \mathcal{N}(0, 1)$ and $y = 5$. This can be rearranged as $x_1 + x_2 = 5 - \sigma_y W$. This rearrangement makes it clear that the samples lie along this line, and that the "noise" perpendicular to the line comes from the value σ_y. The spread of the samples along the line can be noticed to be a result of σ_1^2 in the x_1 direction, and σ_2^2 in the x_2 direction.