CPSC 532W - Homework 3

Xiaoxuan Liang - 48131163

Public GitHub repo: https://github.com/Xiaoxuan1121/CPSC532W/tree/main/a3

1. Program 1: the program has been run for 50000 iterations for each method, and the results are:

   (a) Importance Sampling:

```
Importance sampling for 1.daphne took 28.951397 seconds

posterior mean of mu in 1.daphne using Importance Sampling is 7.34036922454834

posterior variance of mu in 1.daphne using Importance Sampling is 0.8557209372520447
```

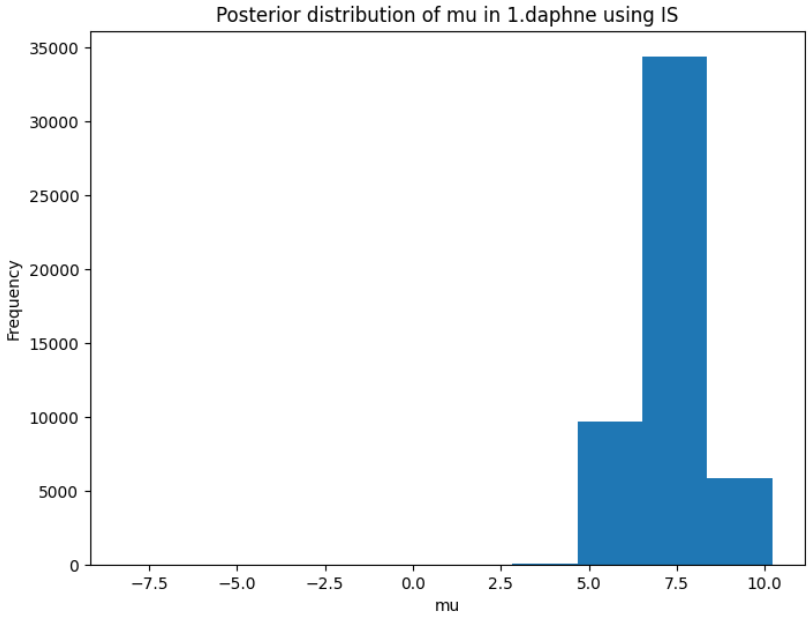


Figure 1: Posterior distrbution for mu for 1.daphne using Importance Sampling

(b) MH Gibbs:

```
Gibbs sampling for 1.daphne took 70.520023 seconds

posterior mean of mu in 1.daphne using Gibbs is 7.228082656860352

posterior variance of mu in 1.daphne using Gibbs is 0.8902580142021179
```

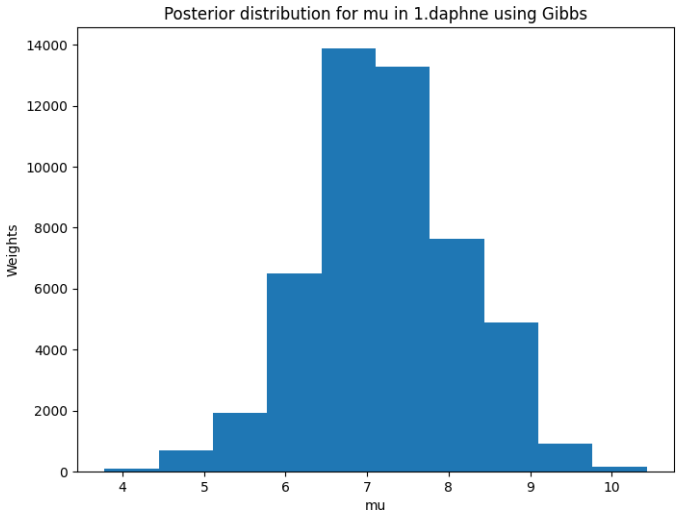


Figure 2: Posterior distrbution for mu for 1.daphne using MH Gibbs Sampling

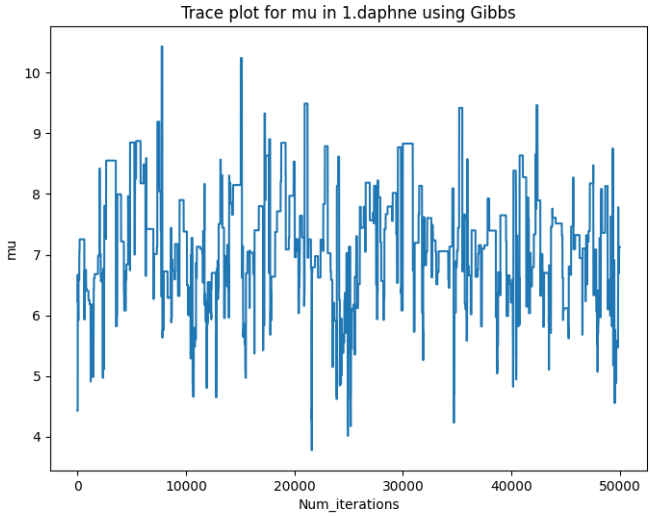


Figure 3: Trace plot for mu for 1.daphne using MH Gibbs Sampling

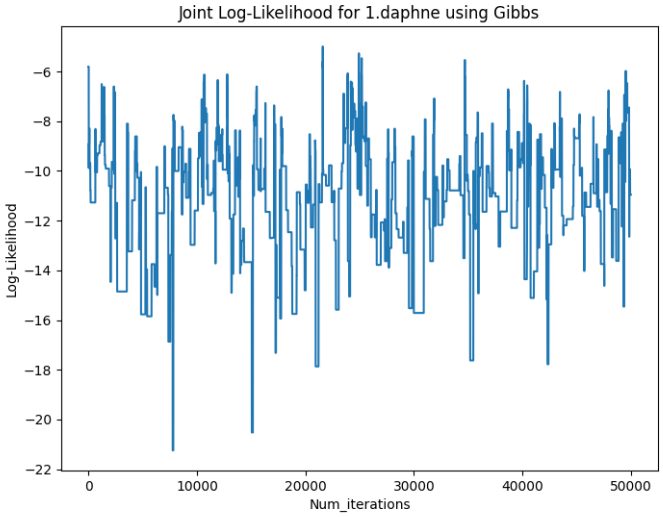


Figure 4: Joint loglikelihood plot for mu for 1.daphne using MH Gibbs Sampling

(c) HMC:

```
Hamiltonian monte carlo for 1.daphne took 479.338874 seconds

posterior mean of mu in 1.daphne using HMC is 7.254238605499268

posterior variance of mu in 1.daphne using HMC is 0.8068938255310059
```

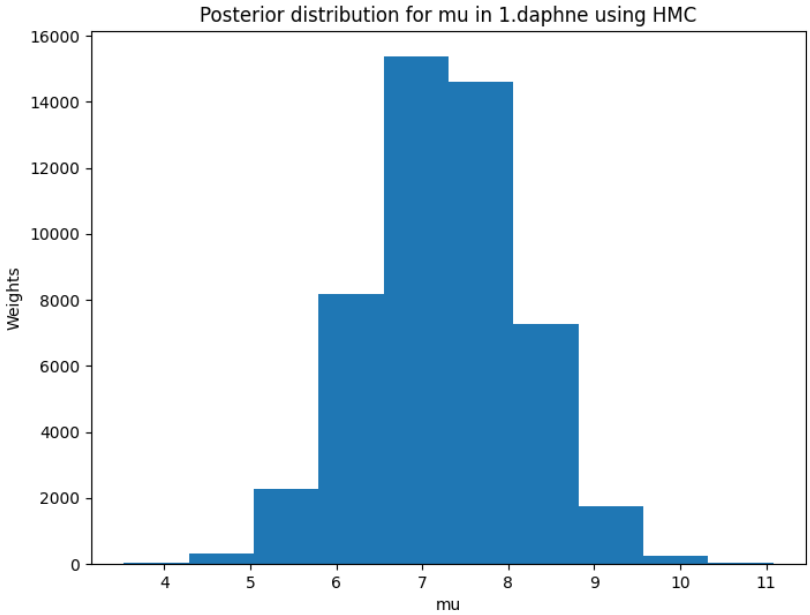


Figure 5: Posterior distrbution for mu for 1.daphne using HMC

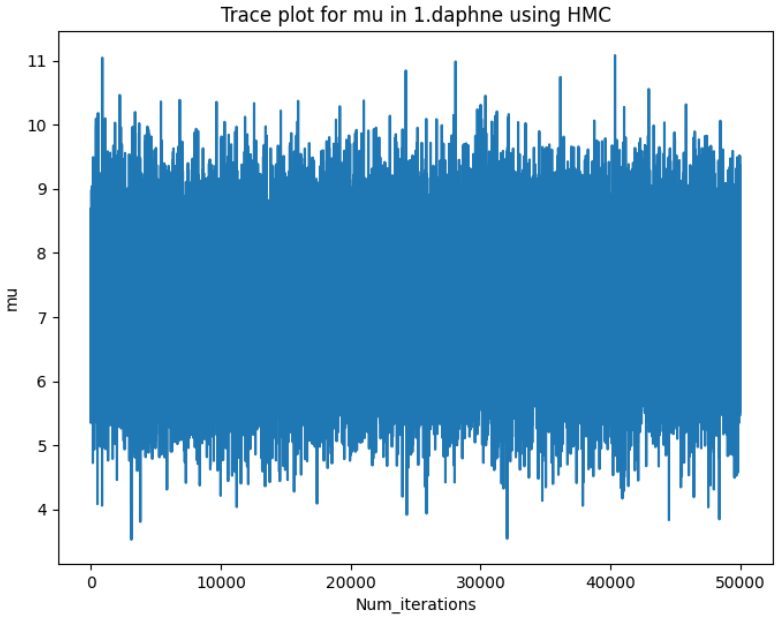


Figure 6: Trace plot for mu for 1.daphne using HMC

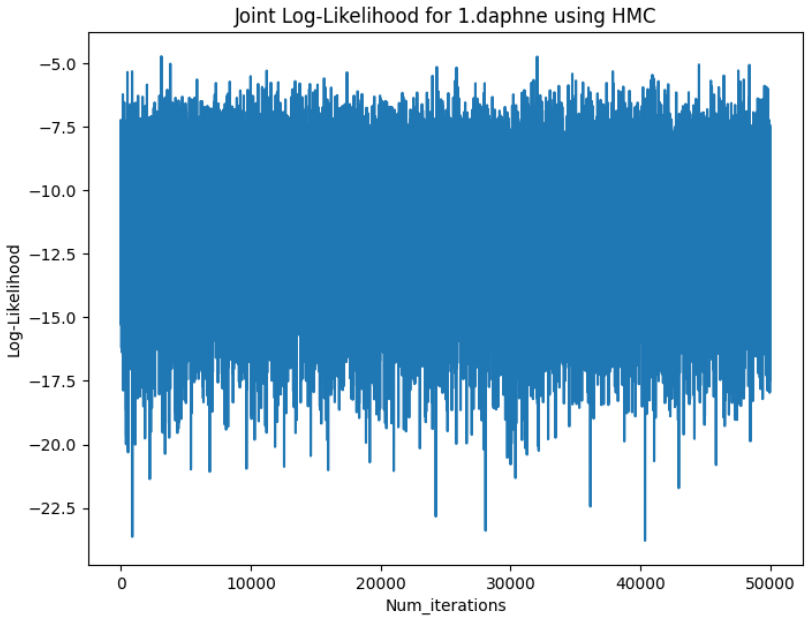
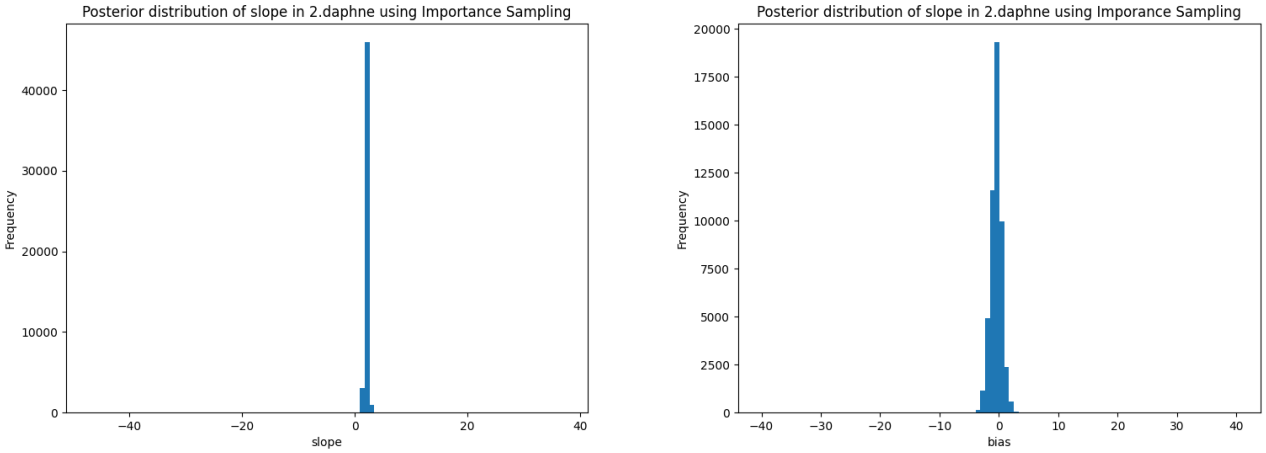


Figure 7: joint loglikelihood plot for mu for 1.daphne using HMC

2. Program 2: First two methods were ran 50000 iterations, HMC is ran 15000 iterations

(a) Importance Sampling

```
Importance sampling for 2.daphne took 67.366311 seconds

posterior means of slope and bias in 2.daphne using Importance Sampling are tensor([ 2.1342, -0.4396])

posterior covariance of slopa and bias in 2.daphne using Importance Sampling is [[ 0.04997719 -0.18395454]
 [-0.18395454  0.83638414]]
```



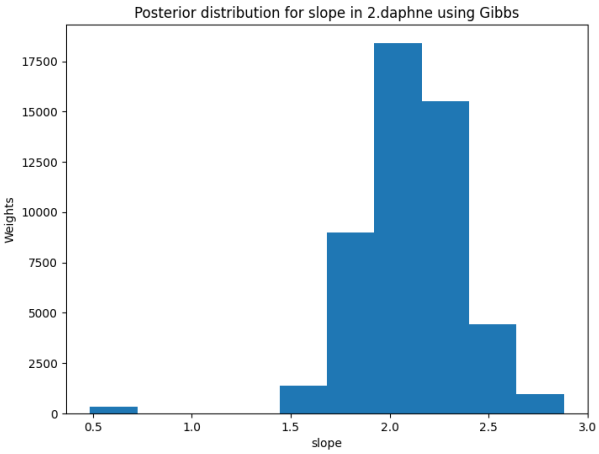
(a) Samples from the posterior for slope

(b) Samples from the posterior for bias

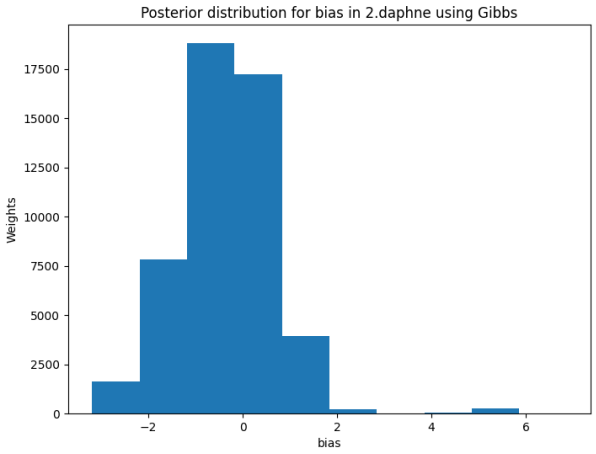
Figure 8: Posterior distribution for slope and bias for 2.daphne using Importance Sampling

(b) MH Gibbs

```
Gibbs sampling for 2.daphne took 173.793085 seconds

posterior means of slope and bias in 2.daphne using MH Gibbs Sampling is tensor([ 2.1813, -0.6248])

posterior covariance of slope and bias in 2.daphne using MH Gibbs Sampling is [[ 0.07555684 -0.26053815]
 [-0.26053815  1.07662599]]
```

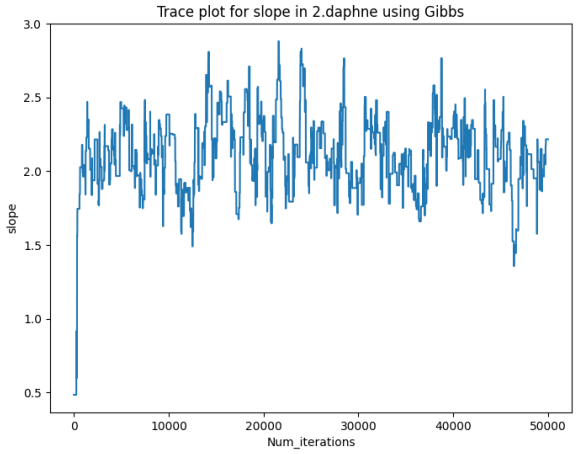


(a) Samples from the posterior for slope

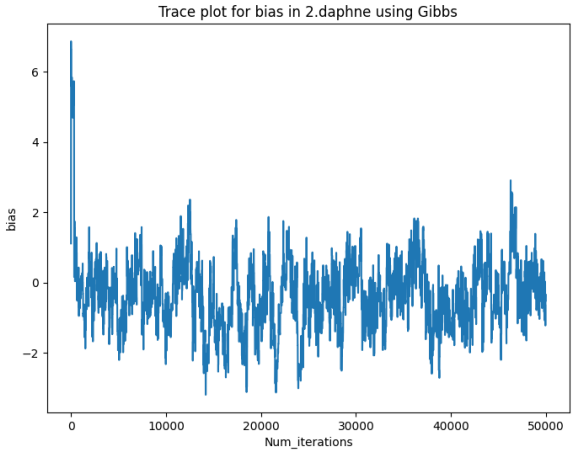


(b) Samples from the posterior for bias

Figure 9: Posterior distribution for slope and bias for 2.daphne using MH Gibbs Sampling

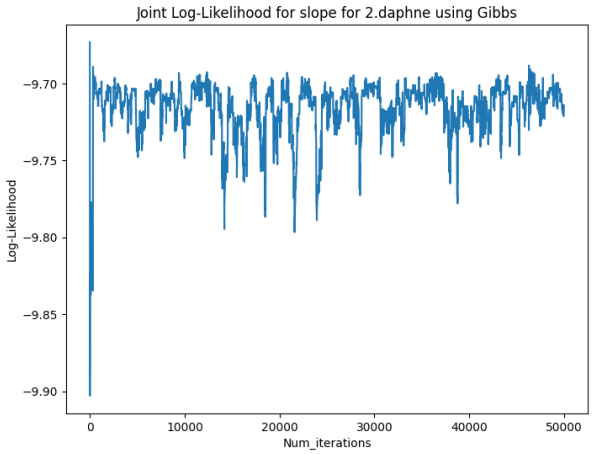


(a) Samples from the posterior for slope

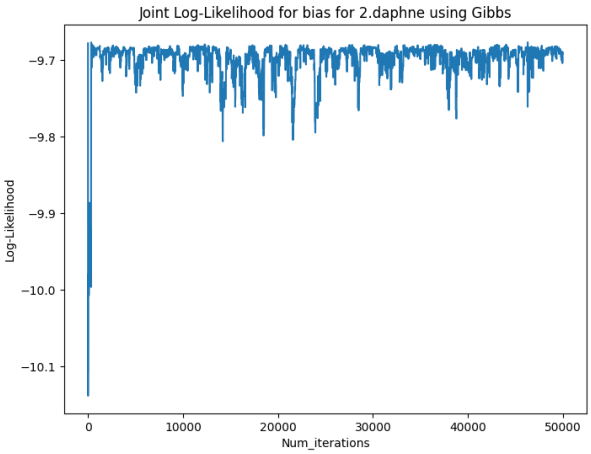


(b) Samples from the posterior for bias

Figure 10: Trace plots for slope and bias for 2.daphne using MH Gibbs Sampling



(a) Samples from the posterior for slope



(b) Samples from the posterior for bias

Figure 11: Joint Log-likelihood plots for slope and bias for 2.daphne using MH Gibbs Sampling

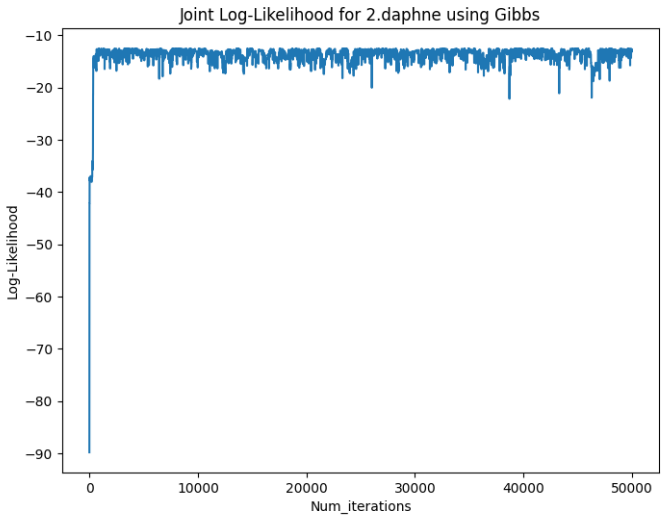
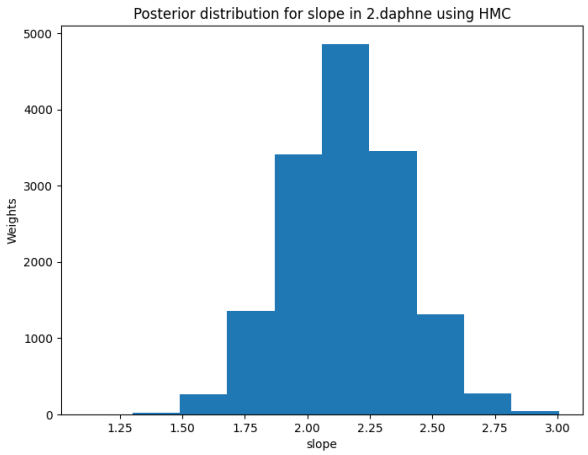


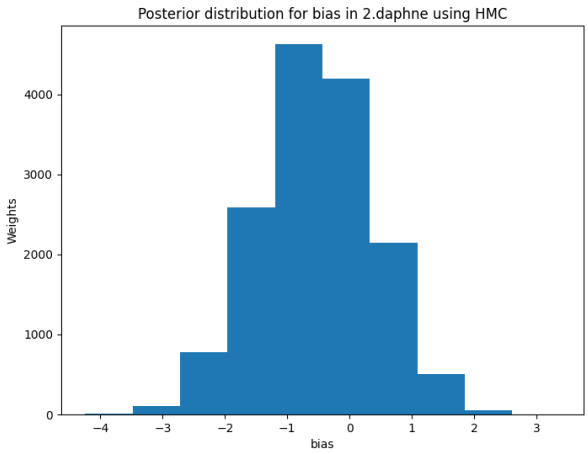
Figure 12: Joint Log-likelihood plots for 2.daphne using MH Gibbs Sampling

(c) HMC

```
Hamiltonian monte carlo for 2.daphne took 253.446195 seconds

posterior means of slope and bias in 2.daphne using HMC is tensor([ 2.1535, -0.5259], grad_fn=<MeanBackward1>)

posterior covariance of slope and bias in 2.daphne using HMC is [[ 0.05315861 -0.1921821 ]
 [-0.1921821   0.83151631]]
```

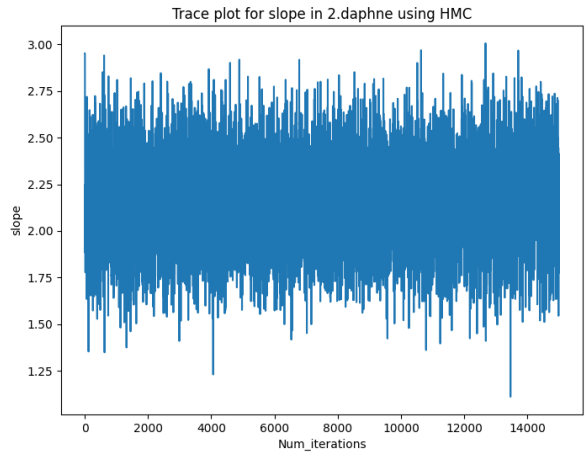


(a) Samples from the posterior for slope

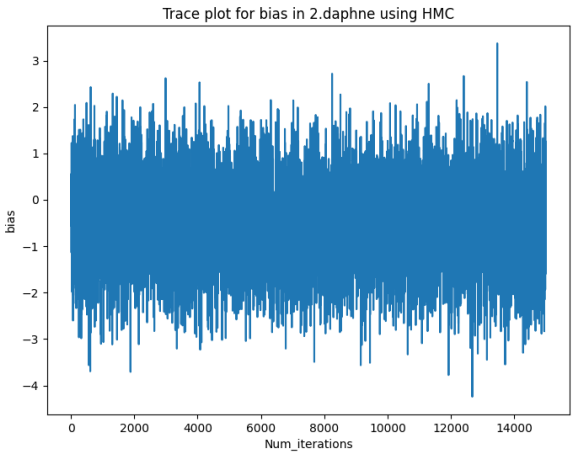


(b) Samples from the posterior for bias

Figure 13: Posterior distribution for slope and bias for 2.daphne using HMC Sampling

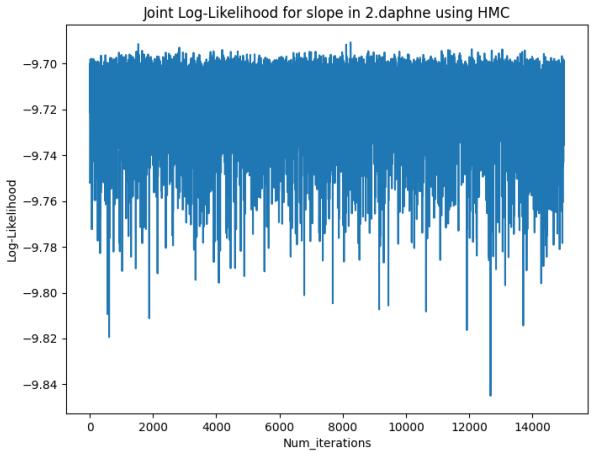


(a) Samples from the posterior for slope

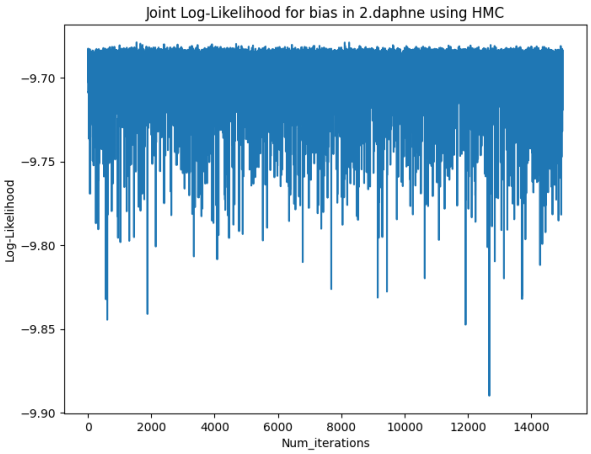


(b) Samples from the posterior for bias

Figure 14: Trace plots for slope and bias for 2.daphne using HMC Sampling



(a) Samples from the posterior for slope



(b) Samples from the posterior for bias

Figure 15: Joint Log-likelihood plots for slope and bias for 2.daphne using HMC Sampling

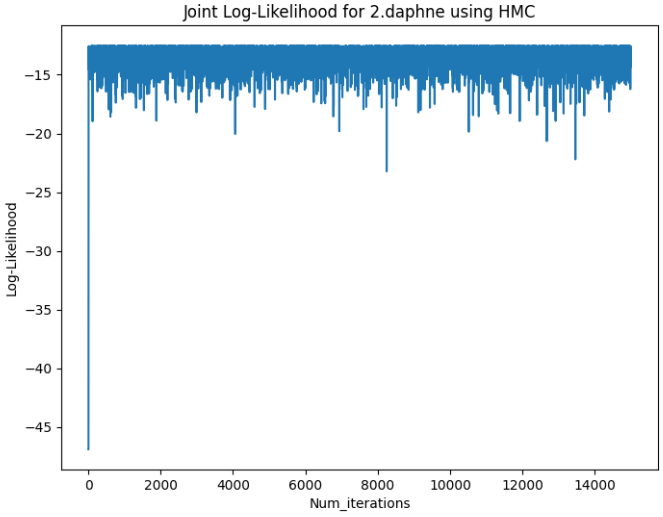


Figure 16: Joint Log-likelihood plots for 2.daphne using HMC Sampling

3. Program 3: Importance Sampling was ran 20000 iterations, and Gibbs sampling was ran 10000 iterations.

   (a) Importance Sampling

```
Importance sampling for 3.daphne took 51.753358 seconds

posterior probability (mean) that the first and second datapoint are
in the same cluster using Important sampling is 0.6138077974319458
```

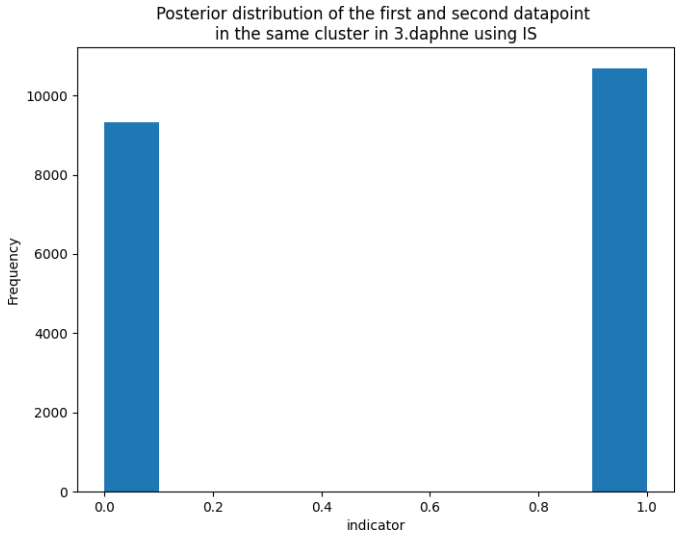


Figure 17: Posterior distribution for probability for 3.daphne using Importance Sampling

(b) MH Gibbs Sampling

```
Gibbs sampling for 3.daphne took 295.438750 seconds

posterior probability that the first and second datapoint are
in the same cluster using MH Gibbs is 0.70660001039505
```

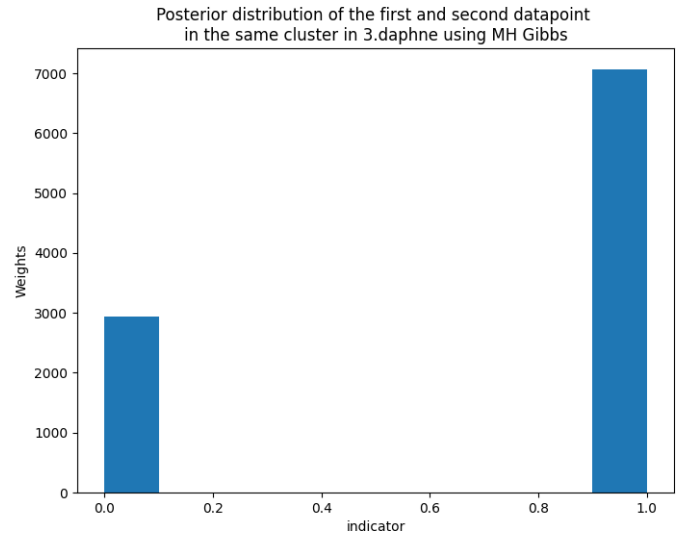


Figure 18: Posterior distribution for probability for 3.daphne using MH Gibbs Sampling

4. Program 4: Both methods were ran 50000 iterations

(a) Importance Sampling

```
Importance sampling for 4.daphne took 33.953193 seconds

posterior probability of raining in 4.daphne using Importance sampling is 0.3193276524543762
```

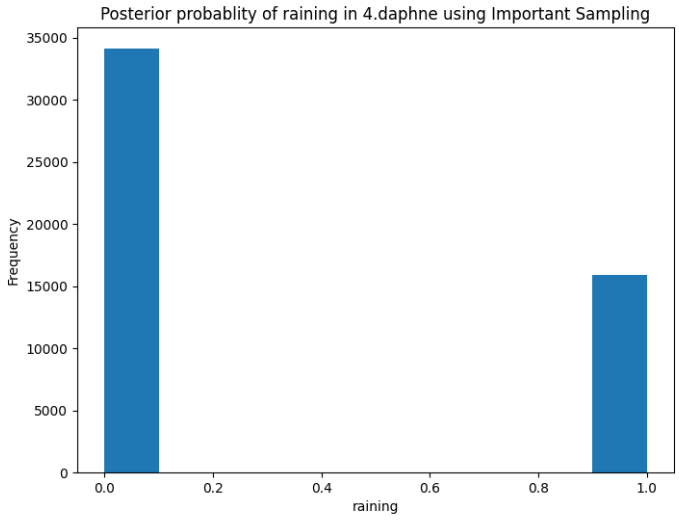


Figure 19: Posterior distribution for raining for 4.daphne using Importance Sampling

(b) MH Gibb Sampling

```
Gibbs sampling for 4.daphne took 212.917732 seconds

posterior probability of raining in 4.daphne using MH Gibbs is 0.3207400143146515
```

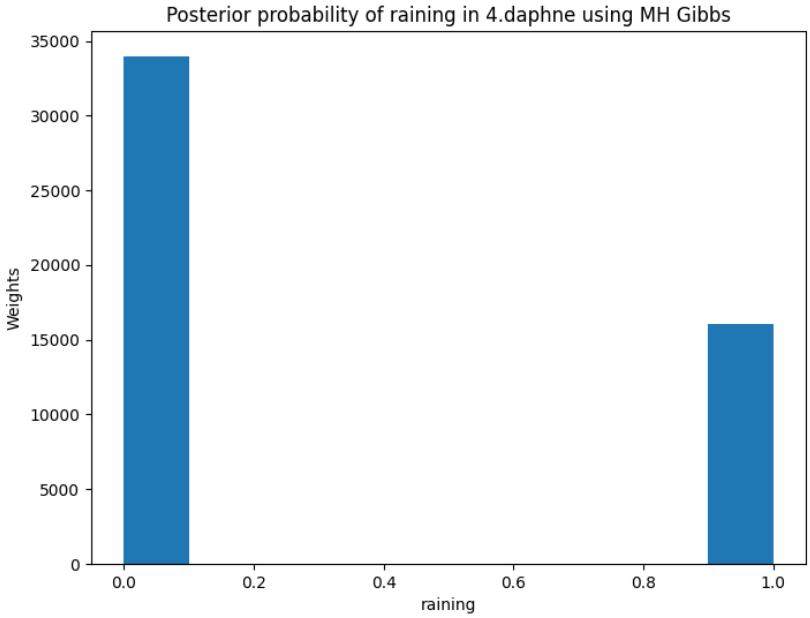


Figure 20: Posterior distribution of raining for 4.daphne using Importance Sampling

5. The caveat of directly applying Dirac distribution curing sampling process is that the likelihood goes to infinity at the center of the distribution and 0 anywhere else other than the center of the distribution, such that it is impossible to land into the region that the probability is positive. Therefore, a function is proposed to approximate Dirac distribution, which is

$$f(x) = z \exp(-(x-c)^4)$$

where c indicates the center of this function and z is a constant. It has the similar shape as what Dirac function looks like, and the highest likelihood also occurs at the center, which is several times of the likelihood corresponding to anywhere else. More important, this function is differentiable everywhere and also integrable from $-\infty$ to ∞:

$$\int_{-\infty}^{\infty} z \exp(-(x-c)^8 \, dx = 2z\Gamma\left(\frac{5}{4}\right) z \Rightarrow z = \frac{1}{2\Gamma(5/4)} \text{ such that the integral is equal to 1}$$

Results analysis:

(a) Importance Sampling seems giving a good result. Both posterior means for x and y are around 3.5 as x and y have the same priors and they are added to approximately 7.

(b) Gibbs Sampling seems giving a not bad result. The sum of posterior means of x and y is 7 but they are not around 3.5. It makes sure that posterior means are on the line $x + y = 7$. This may due to the order that we sample x and y: we sample for x and first and update y based on the resulted value of x.

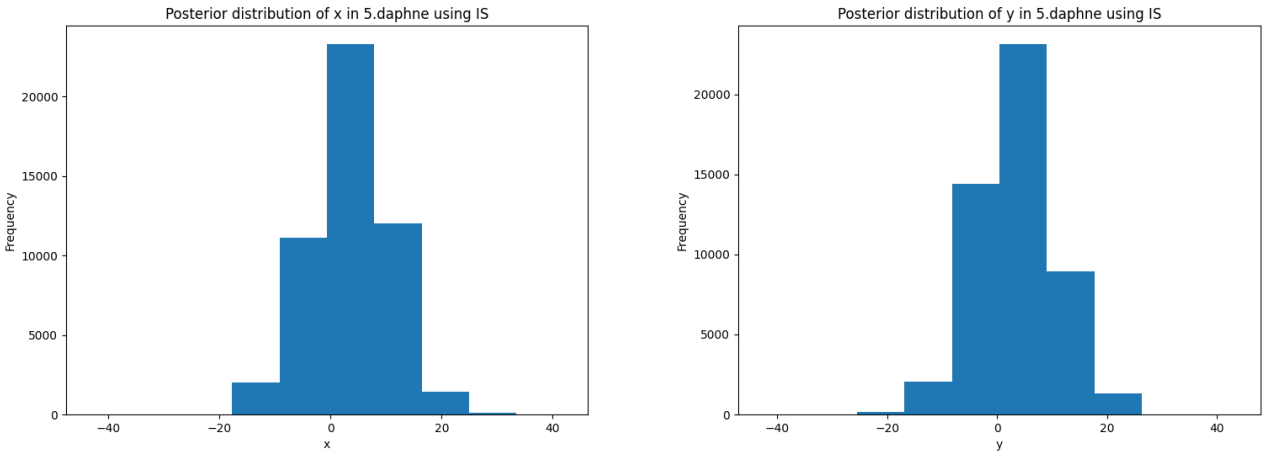
(c) HMC gives a bad result and it's not efficient. The gradient could easily go to infinity and samples in the

whole process get stuck at some numbers forever so we have to rerun it until the gradients stay at the range of finite numbers with luck. It's achievable but not reliable to calculate posterior means in this way.

First two methods were ran 50000 iterations, and HMC was ran 10000 iterations.

(a) Importance Sampling

```
Importance sampling for 5.daphne took 14.455134 seconds

posterior mean of x in 5.daphne using Importance Sampling is 3.4984302520751953

posterior mean of y in 5.daphne using Importance Sampling is 3.4777727127075195

posterior variance of x in 5.daphne using Importance Sampling is 48.890724182128906

posterior variance of y in 5.daphne using Importance Sampling is 48.88740158081055
```



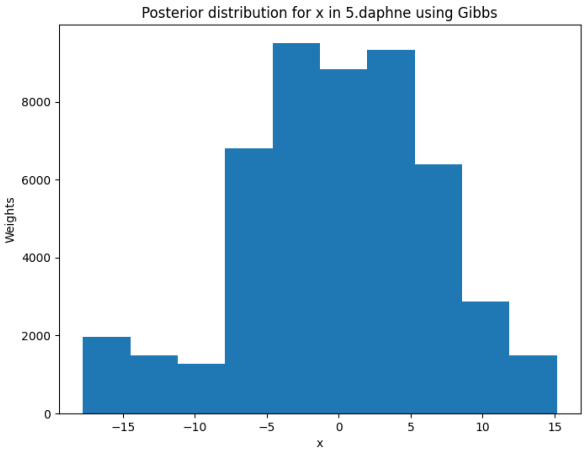
(a) Samples from the prior for slope

(b) Samples from the prior for bias

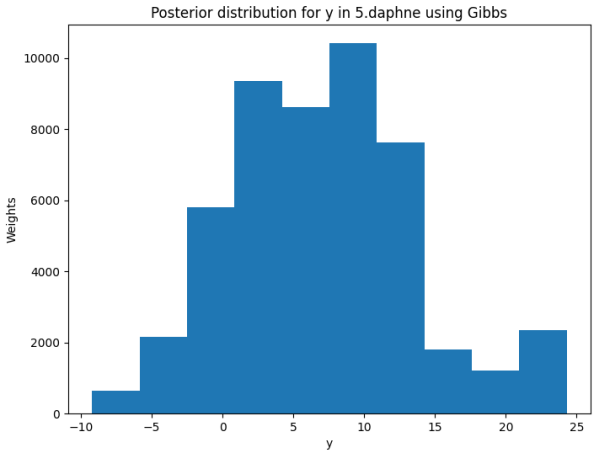
Figure 21: Posterior distribution for x and y for 5.daphne using Importance Sampling

(b) MH Gibbs Sampling

```
Gibbs sampling for 5.daphne took 60.365833 seconds

posterior mean of x in 5.daphne using Gibbs is -0.08612194657325745

posterior mean of y in 5.daphne using Gibbs is 7.060629844665527

posterior variance of x in 5.daphne using Gibbs is 43.51240158081055

posterior variance of y in 5.daphne using Gibbs is 43.29920196533203
```

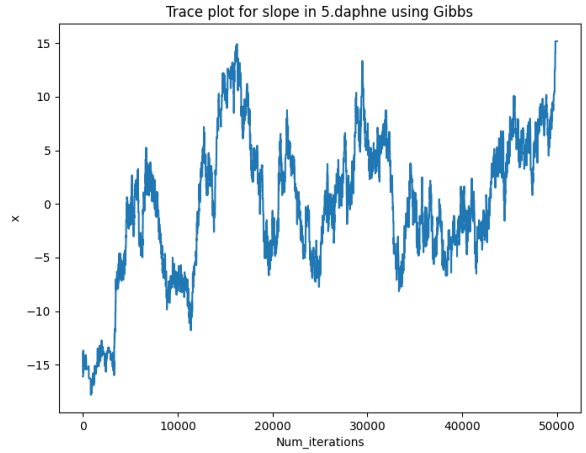


(a) Samples from the posterior for slope

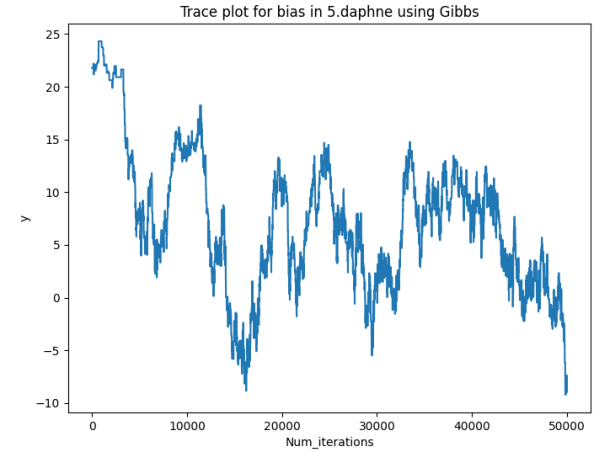


(b) Samples from the posterior for bias

Figure 22: Posterior distribution for x and y for 5.daphne using MH Gibbs Sampling

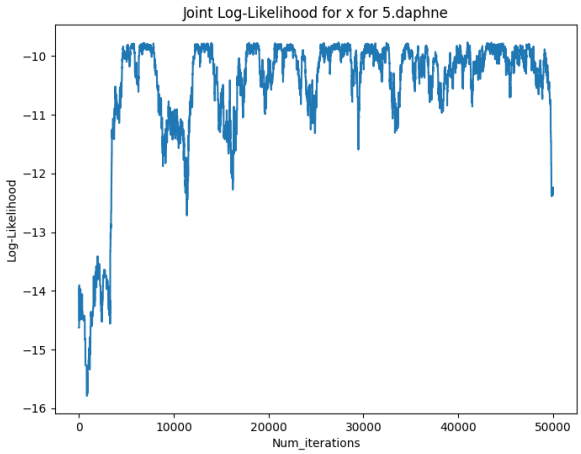


(a) Samples from the posterior for x

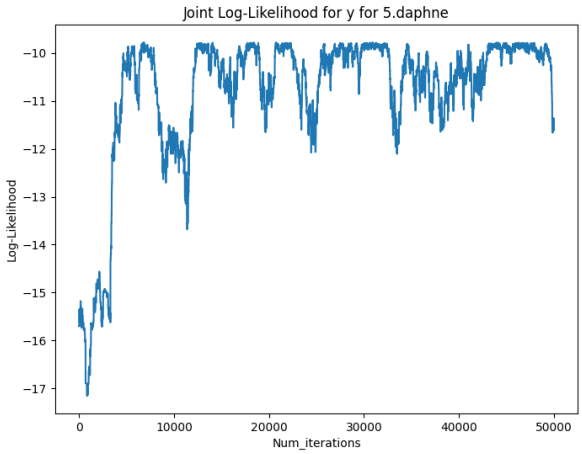


(b) Samples from the posterior for y

Figure 23: Trace plots for x and y for 5.daphne using MH Gibbs Sampling



(a) Samples from the posterior for x



(b) Samples from the posterior for y

Figure 24: Joint Log-likelihood plots for x and y for 5.daphne using MH Gibbs Sampling

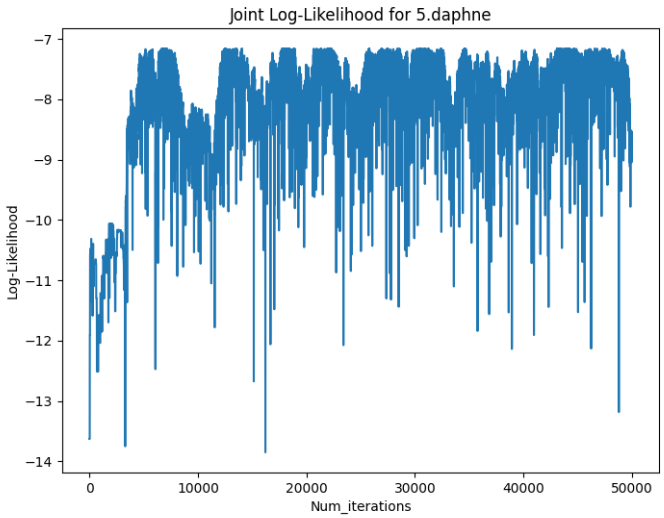
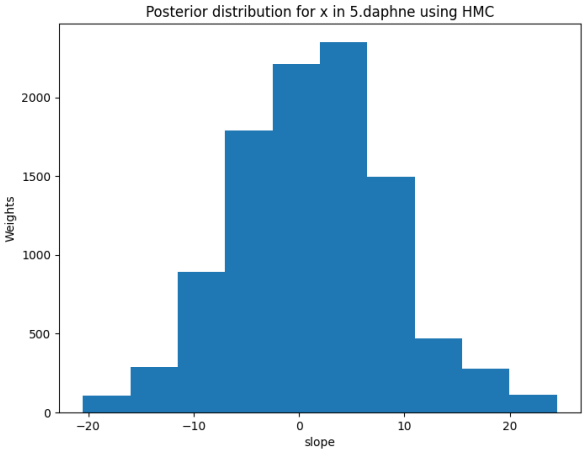


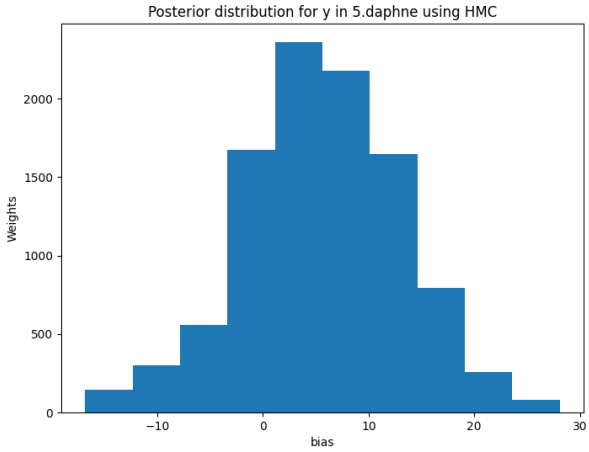
Figure 25: Joint Log-likelihood plots for 5.daphne using MH Gibbs Sampling

(c) HMC

```
Hamiltonian monte carlo for 5.daphne took 64.211889 seconds

posterior mean of x in 5.daphne using HMC is 1.3545114994049072

posterior mean of y in 5.daphne using HMC is 5.6317267417907715

posterior variance of x in 5.daphne using HMC is 55.87028121948242

posterior variance of y in 5.daphne using HMC is 55.87904357910156
```

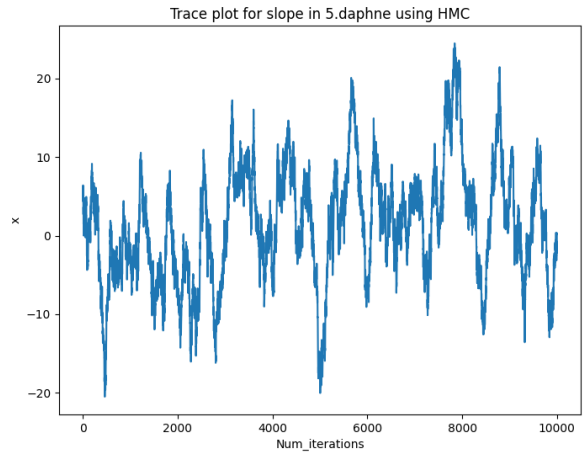


(a) Samples from the posterior for x

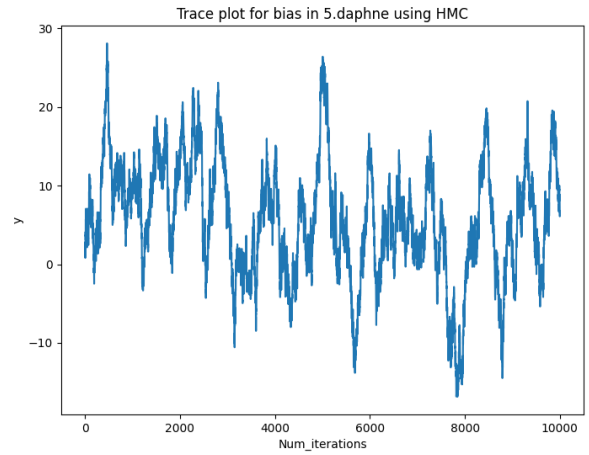


(b) Samples from the posterior for y

Figure 26: Posterior distribution for slope and bias for 5.daphne using HMC Sampling

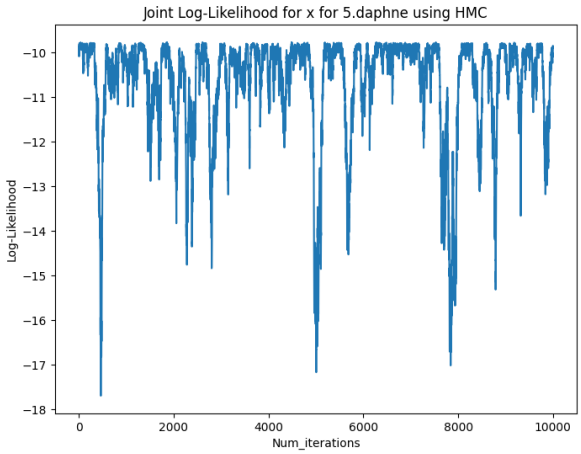


(a) Samples from the posterior for x

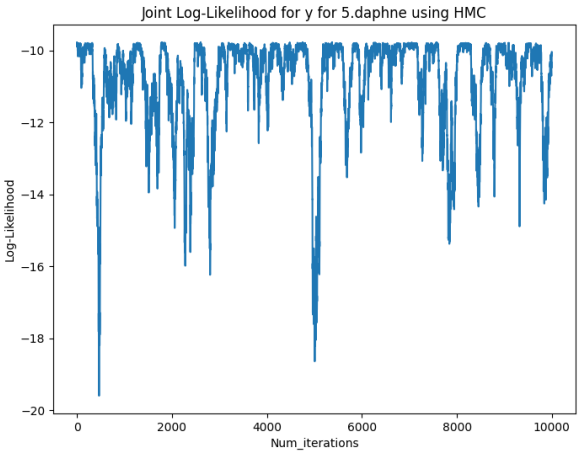


(b) Samples from the posterior for y

Figure 27: Trace plots for x and y for 5.daphne using HMC Sampling



(a) Samples from the prior for x



(b) Samples from the prior for y

Figure 28: Joint Log-likelihood plots for x and y for 5.daphne using HMC Sampling

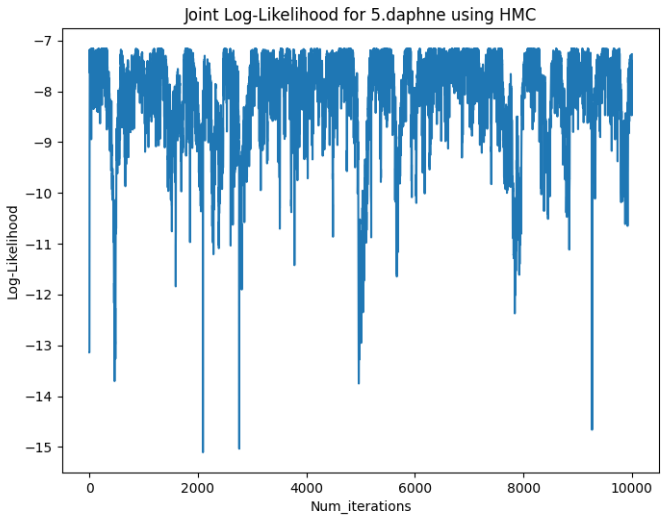


Figure 29: Joint Log-likelihood plots for 5.daphne using HMC Sampling

6. Code Snippets:

```
elif ast[0] == 'observe':
    dist, sigma = evaluate_variable(ast[1], variables_dict, functions_dict, sigma)
    observation, sigma = evaluate_variable(ast[2], variables_dict, functions_dict, sigma)
    if type(observation) is not torch.Tensor:
        observation = torch.tensor(float(observation))
    sigma['logW'] = sigma['logW'] + dist.log_prob(observation)
    return observation, sigma
```

```
def importance_Sampling(ast, num_samples):
    samples = []
    Ws = []
    for i in range(num_samples):
        sample, sigma = evaluate_program(ast)
        samples.append(sample)
        W = torch.exp(sigma['logW'])
        Ws.append(W)

    samples = torch.stack(samples)
    Ws = torch.stack(Ws)

    return samples, Ws


def importance_Sampling_mean(samples, Ws, num_samples):
    unnormlized_mean = 0
    for i in range(num_samples):
        unnormlized_mean += samples[i] * Ws[i]
    mean = unnormlized_mean / sum(Ws)

    return mean

def importance_Sampling_variance(samples, Ws, mean):
    variance = torch.matmul(((samples - mean)**2).T, Ws) / sum(Ws)

    return variance

def importance_Sampling_Covariance(samples, Ws):
    covariance = np.cov(samples.numpy().T, aweights = Ws.numpy(), ddof = 0)

    return covariance
```

Figure 30: Code for Importance Sampling

```
def accept(x, cal_X, cal_X_prime, edges, links):
    q = deterministic_eval(evaluate(links[x][1], cal_X))
    q_prime = deterministic_eval(evaluate(links[x][1], cal_X_prime))
    log_alpha = q_prime.log_prob(cal_X[x]) - q.log_prob(cal_X_prime[x])
    free_variables = edges[x].copy()
    free_variables.append(x)
    for v in free_variables:
        if type(cal_X_prime[v]) is bool:
            cal_X_prime[v] = torch.tensor(float(cal_X_prime[v]))
        if type(cal_X[v]) is bool:
            cal_X[v] = torch.tensor(float(cal_X[v]))
        log_alpha += deterministic_eval(evaluate(links[v][1], cal_X_prime)).log_prob(cal_X_prime[v])
        log_alpha -= deterministic_eval(evaluate(links[v][1], cal_X)).log_prob(cal_X[v])

    alpha = torch.exp(log_alpha)
    return alpha

def gibbs_Step(X, cal_X, edges, links):
    for x in X:
        q = deterministic_eval(evaluate(links[x][1], cal_X))
        cal_X_prime = cal_X.copy()
        cal_X_prime[x] = q.sample()
        alpha = accept(x, cal_X, cal_X_prime, edges, links)
        u = dist.uniform.Uniform(0, 1).sample()
        if u < alpha:
            cal_X = cal_X_prime
    return cal_X

def gibbs(X, cal_X, num_samples, edges, links, returnings):
    samples = []
    cal_Xs = []
    for i in range(num_samples):
        # print(i)
        cal_X_prime = gibbs_Step(X, cal_X, edges, links)
        sample = deterministic_eval(evaluate(returnings, cal_X_prime))
        samples.append(sample)
        cal_X = cal_X_prime
        cal_Xs.append(cal_X)
    samples = torch.stack(samples)
    return samples, cal_Xs
```

Figure 31: Code for MH gibbs sampling Part I

```
def joint_log_likelihood(vertices, links, variables_dict_set, num_samples):
    logPs = []
    for i in range(num_samples):
        logP = 0
        variables_dict = variables_dict_set[i]
        for vertex in vertices:
            logP += deterministic_eval(evaluate(links[vertex][1], variables_dict)).log_prob(variables_dict[vertex]).float()
        logPs.append(logP)
    return logPs

def latent_observed(links):
    latent = []
    observed = []
    for variable in links:
        if links[variable][0] == 'sample*':
            latent.append(variable)

        elif links[variable][0] == 'observe*':
            observed.append(variable)
    return latent, observed
```

Figure 32: Code for MH gibbs sampling Part II

```
def H(cal_X, cal_Y, links, R, M):
    U_p = U(cal_X, cal_Y, links)
    K = 0.5 * torch.matmul(R.T, torch.matmul(torch.inverse(M), R))
    return U_p + K

def U(cal_X, cal_Y, links):
    logP = 0
    for observe in cal_Y:
        logP += deterministic_eval(evaluate(links[observe][1], {**cal_X, **cal_Y})).log_prob(cal_Y[observe])

    for latent in cal_X:
        logP += deterministic_eval(evaluate(links[latent][1], {**cal_X, **cal_Y})).log_prob(cal_X[latent])

    return -logP

def grad_U(cal_X, cal_Y, links):
    U_cal_X = U(cal_X, cal_Y, links)
    U_cal_X.backward()
    grads = torch.zeros(len(cal_X))
    i = 0
    for latent in cal_X:
        grads[i] = (cal_X[latent].grad)
        if torch.isinf(grads[i]):
            grads[i] = torch.tensor(2**63 - 1)
        if torch.isneginf(grads[i]):
            grads[i] = torch.tensor(-2**63)
        i += 1
    return grads
```

Figure 33: Code for HMC sampling part I

```
def leapfrog(cal_X0, cal_R0, T, epsilon, cal_Y, links):
    cal_X = {0 : cal_X0}
    cal_R = {0 : cal_R0}
    cal_R[0.5] = cal_R0 - 0.5 * epsilon * grad_U(cal_X0, cal_Y, links)
    for t in range(1, T):
        cal_X[t] = add_dict(cal_X[t - 1], epsilon * cal_R[t - 0.5])
        cal_R[t + 0.5] = cal_R[t - 0.5] - epsilon * grad_U(cal_X[t], cal_Y, links)
    cal_X[T] = add_dict(cal_X[T - 1], epsilon * cal_R[T - 0.5])
    cal_R[T] = cal_R[T - 0.5] - 0.5 * epsilon * grad_U(cal_X[T], cal_Y, links)
    return cal_X[T], cal_R[T]


def add_dict(cal_inside, cal_R):
    dicts = {}
    keys = list(cal_inside.keys())
    for i in range(len(keys)):
        dicts[keys[i]] = cal_inside[keys[i]].detach() + cal_R[i]
        dicts[keys[i]].requires_grad = True

    return dicts


def HMC(cal_X, S, T, epsilon, M, cal_Y, links):
    samples = []
    cal_XYs = []
    for s in range(1, S + 1):
        R = dist.multivariate_normal.MultivariateNormal(torch.zeros(len(cal_X)), M).sample()
        cal_X_prime, R_prime = leapfrog(copy.deepcopy(cal_X), R, T, epsilon, cal_Y, links)
        u = dist.uniform.Uniform(0, 1).sample()
        if u < torch.exp(-H(cal_X_prime, cal_Y, links, R_prime, M) + H(cal_X, cal_Y, links, R, M)):
            cal_X = cal_X_prime
        samples.append(cal_X)
        cal_XYs.append({**cal_X, **cal_Y})

    return samples, cal_XYs
```

Figure 34: Code for HMC sampling part II

```
def joint_log_likelihood_HMC(vertices, links, variables_dict_set, num_samples):
    logPs = []
    for i in range(num_samples):
        logP = 0
        variables_dict = variables_dict_set[i]
        for vertex in vertices:
            logP += deterministic_eval(evaluate(links[vertex][1], variables_dict)).log_prob(variables_dict[vertex])
        logPs.append(logP)
    logPs = torch.stack(logPs).detach()

    return logPs
```

Figure 35: Code for HMC sampling part III