CPSC 532W Assignment 3

Ali Seyfi - 97446637

Here is the link to the repository:
https://github.com/aliseyfi75/Probabilistic-Programming/tree/master/Assignment_3

1 Importance Sampling

Write IS sampler that consume the output produced by the Daphne compiler and the evaluators you wrote in completing HW 2.

1.1 Code

Provide code snippets that document critical aspects of your implementation sufficient to allow us to quickly determine whether or not you individually completed the assignment.

1.1.1 primitives

```python
baseprimitives = {
    '+': lambda x: x[0] + x[1],
    '-': lambda x: x[0] - x[1],
    '*': lambda x: x[0] * x[1],
    '/': lambda x: x[0] / x[1],
    '>': lambda x: x[0] > x[1],
    '>=': lambda x: x[0] >= x[1],
    '<': lambda x: x[0] < x[1],
    '<=': lambda x: x[0] <= x[1],
    '==': lambda x: x[0] == x[1],
    '=': lambda x: torch.tensor([1]) if x[0] == x[1] else torch.tensor([0]),
    'and': lambda x: x[0] and x[1],
    'or': lambda x: x[0] or x[1],
    'sqrt': lambda x: torch.sqrt(x[0]),
    'exp': lambda x: torch.exp(x[0]),
    'log': lambda x: torch.log(x[0]),
    'vector': vector,
    'list': list,
    'get': get,
    'put': put,
    'hash-map': hash_map,
    'first': lambda x: x[0][0],
    'last': lambda x: x[0][-1],
    'nth': lambda x: x[0][int(x[1].item())],
    'second': lambda x: x[0][1],
    'rest': lambda x: x[0][1:],
    'append': append,
    'cons': lambda x: append([x[1],x[0]]),
    'conj': append,
    'mat-add': lambda x: x[0] + x[1],
    'mat-mul': lambda x: torch.matmul(x[0], x[1]),
    'mat-transpose': lambda x: x[0].T,
    'mat-tanh': lambda x: x[0].tanh(),
    'mat-repmat': lambda x: x[0].repeat((int(x[1].item()), int(x[2].item())))
}
```

Listing 1: primitives.py - Base primitives

```python
def vector(x):
    try:
        vector = torch.stack(x)
    except:
        vector = x
    return vector

def list(x):
    try:
        list = torch.stack(x)
    except:
        list = x
    return list

def get(x):
    if type(x[0]) == dict:
        value = x[0][x[1].item()]
    else:
        value = x[0][x[1].long()]
    return value

def put(x):
    if type(x[0]) == dict:
        x[0][x[1].item()] = x[2]
    else:
        x[0][x[1].long()] = x[2]
    return x[0]

def hash_map(x):
    keys = x[::2]
    value = x[1::2]
    new_keys = []
    for key in keys:
        try:
            new_keys.append(key.item())
        except:
            new_keys.append(key)
    result = dict(zip(new_keys, value))
    return result

def append(x):
    first = x[0]
    second = x[1]

    if first == 'vector':
        first = torch.tensor([])
    elif first.dim() == 0:
        first = first.unsqueeze(0)
    if second == 'vector':
        second = torch.tensor([])
    if second.dim() == 0:
        second = second.unsqueeze(0)
    return torch.cat((first, second))
```

Listing 2: primitives.py - Functions

```python
class Dist:
    def __init__(self, name, distribution, num_par, *par):
        self.name = name
        self.distribution = distribution
        self.num_par = num_par
        self.pars = []
        for i in range(num_par):
            self.pars.append(par[i])

    def sample(self):
        return self.distribution.sample()

    def log_prob(self, c):
        return self.distribution.log_prob(c)

class normal(Dist):
    def __init__(self, pars):
        mean = pars[0]
        var = pars[1]
        super().__init__('normal', distributions.Normal(mean, var), 2, mean, var)

class beta(Dist):
    def __init__(self, pars):
        alpha = pars[0]
        betta = pars[1]
        super().__init__('beta', distributions.Beta(alpha, betta), 2, alpha, betta)

class exponential(Dist):
    def __init__(self, par):
        lamda = par[0]
        super().__init__('exponential', distributions.Exponential(lamda), 1, lamda)

class uniform(Dist):
    def __init__(self, pars):
        a, b = pars[0], pars[1]
        super().__init__('uniform', distributions.Uniform(a, b), 2, a, b)

class discrete(Dist):
    def __init__(self, pars):
        prob = pars[0]
        super().__init__('discrete', distributions.Categorical(prob), 0)

class bernoulli(Dist):
    def __init__(self, pars):
        p = pars[0]
        super().__init__('bernoulli', distributions.Bernoulli(p), 1, p)

class gamma(Dist):
    def __init__(self, pars):
        alpha, beta = pars[0], pars[1]
        super().__init__('gamma', distributions.Gamma(alpha, betta), 2, alpha, betta)

class dirichlet(Dist):
    def __init__(self, pars):
        super().__init__('dirichlet', distributions.Dirichlet(*pars), len(pars), *pars)

class dirac(Dist):
    def __init__(self, value):
        mean = value[0]
        mean = torch.clip(mean, -1e5, 1e5)
        var = torch.tensor(1e-3)
        super().__init__('normal', distributions.Normal(mean, var), 2, mean, var)
```

Listing 3: primitives.py - Distributions

```python
distlist = {
    'normal' : normal,
    'beta' : beta,
    'exponential' : exponential,
    'uniform' : uniform,
    'discrete' : discrete,
    'bernoulli': bernoulli,
    'gamma': gamma,
    'dirichlet': dirichlet,
    'flip': bernoulli,
    'dirac': dirac
}
```

Listing 4: primitives.py - distlist

1.1.2 evaluate program

```python
def evaluate_program(ast, sigma={}):
    """Evaluate a program as desugared by daphne, generate a sample from the prior
    Args:
        ast: json FOPPL program
    Returns: sample from the prior of ast
    """
    funcs = {}
    final_ast = ast
    if isinstance(ast, list):
        if isinstance(ast[0], list):
            if ast[0][0] == 'defn':
                for statement in ast:
                    if statement[0]== 'defn':
                        funcs[statement[1]] = (statement[1], statement[2], statement[3])
                        final_ast = final_ast[1:]
                    else:
                        result, sigma = eval(statement, sigma, {}, funcs)
            if final_ast[0][0] != 'defn':
                result, sigma = eval(final_ast[0], sigma, {}, funcs)
        else:
            result, sigma = eval(ast, sigma, {}, funcs)
    else:
        result, sigma = eval(ast, sigma, {}, funcs)
    if sigma == {}:
        results = result
    else:
        results = [result, sigma]
    return results
```

Listing 5: evaluation_based_sampling.py - evaluate_program

1.1.3 eval

```python
def eval(x, sigma, l, funcs):
    "Evaluate an expression in an environment."
    if isinstance(x, list) and len(x) == 1:
        x = x[0]
    if not isinstance(x, list):
        if isinstance(x, int) or isinstance(x, float):
            result = torch.tensor(x, dtype=float)
        elif x in baseprimitives or torch.is_tensor(x) or x in funcs or x in distlist:
            result = x
        else:
            result = l[x]
    elif x[0] == 'if':
        cond_result, sigma = eval(x[1], sigma, l, funcs)
        if cond_result:
            result, sigma = eval(x[2], sigma, l, funcs)
        else:
            result, sigma = eval(x[3], sigma, l, funcs)
    elif x[0] == 'let':
        name, exp = x[1]
        result, sigma = eval(exp, sigma, l, funcs)
        l[name]= result
        return eval(x[2], sigma, l, funcs)
    elif x[0] == 'sample':
        dist, sigma = eval(x[1], sigma, l, funcs)
        result = dist.sample()
    elif x[0] == 'observe':
        dist, sigma = eval(x[1], sigma, l, funcs)
        while isinstance(dist, list):
            dist, sigma = eval(dist, sigma, l, funcs)
        result, sigma = eval(x[2], sigma, l, funcs)
        try:
            sigma['logW'] = sigma['logW'] + dist.log_prob(result)
        except:
            pass
    else:
        statements = []
        for expression in x:
            statement, sigma = eval(expression, sigma, l, funcs)
            statements.append(statement)

        first_statemnt, other_statements = statements[0], statements[1:]
        if first_statemnt in baseprimitives:
            result = baseprimitives[first_statemnt](other_statements)
        elif first_statemnt in distlist:
            result = distlist[first_statemnt](other_statements)

        elif first_statemnt in funcs:
            _, variables, process = funcs[first_statemnt]
            assignment = {key:value for key, value in zip(variables, other_statements)}
            result, sigma = eval(process, sigma, {**l, **assignment}, funcs)
        else:
            result = torch.tensor(statements)
    return result, sigma
```

Listing 6: evaluation_based_sampling.py - evaluate_program

1.1.4 likelihood weighting

```
def likelihood_weighting(L, exp):
    sigma = {'logW':0}
    results_temp , sigma_temp = evaluate_program(exp, sigma)
    n_params = 1
    if results_temp.dim() != 0:
        n_params = len(results_temp)
    results = torch.zeros((n_params, L))
    weights = []
    for l in range(L):
        sigma = {'logW':0}
        results_temp , sigma_temp = evaluate_program(exp, sigma)
        results[:,l] = results_temp
        weights.append(sigma_temp['logW'])
    return results, torch.tensor(weights)
```

Listing 7: evaluation_based_sampling.py - likelihood_weighting

1.1.5 expectation calculator

```
def expectation_calculator(results, log_weights, func, *args):
    weights = torch.exp(log_weights)
    func_result = func(results, *args)
    return torch.sum(weights*func_result, dim=1) / torch.sum(weights)
```

Listing 8: evaluation_based_sampling.py - expectation_calculator

1.2 Results

I draw 10^5 samples for each task and the results are in the following:

1.2.0.1 Task 1

Time of drawing samples: **16.51 seconds**
Posterior mean of mu is: **7.2514**
Posterior variance of mu is: **0.8652**

1.2.0.2 Task 2

Time of drawing samples: **145.30 seconds**
Posterior mean of slope is: **1.9222**
Posterior variance of slope is: **0.0237**
Posterior mean of bias is: **0.9856**
Posterior variance of bias is: **0.6657**
Posterior covariance matrix of slope and bias: $\begin{bmatrix} 3.6949 & 1.8946 \\ 1.8946 & 0.9715 \end{bmatrix}$

1.2.0.3 Task 3

Time of drawing samples: **94.24 seconds**
Posterior mean of probability that the first and second datapoint are in the same cluster is: **0.7517**
Posterior variance of probability that the first and second datapoint are in the same cluster is: **0.1866**

1.2.0.4 Task 4

Time of drawing samples: **31.27 seconds**
Posterior mean of probability that it is raining: **0.3195**
Posterior variance of probability that it is raining: **0.2174**

1.2.0.5 Task 5

Time of drawing samples: **17.77 seconds**
Posterior marginal mean of x is: **4.0185**
Posterior marginal variance of x is: **0.4771**
Posterior marginal mean of y is: **2.9814**
Posterior marginal variance of y is: **0.4771**

1.2.1 Histograms

1.2.1.1 Task 1

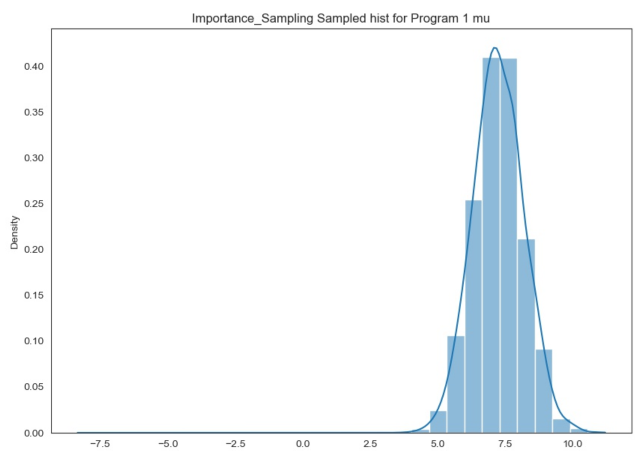
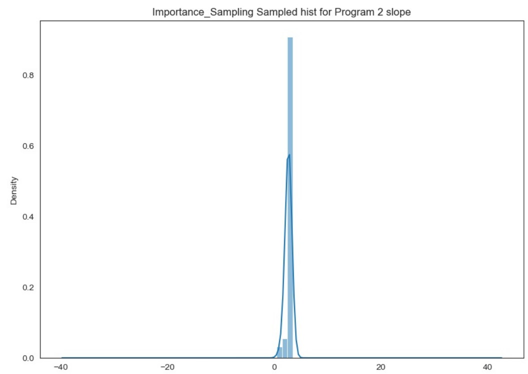


Figure 1: Histogram of posterior distribution of mu

1.2.1.2 Task 2



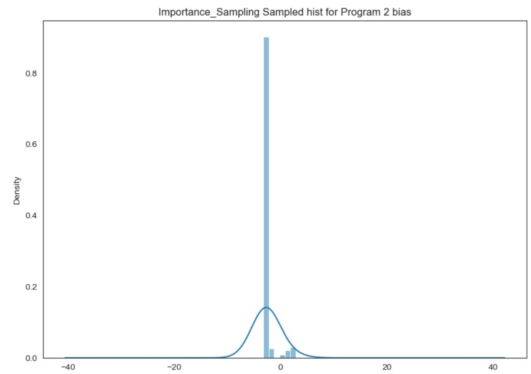


Figure 2: Histogram of posterior distribution of slope Figure 3: Histogram of posterior distribution of bias

1.2.1.3 Task 3

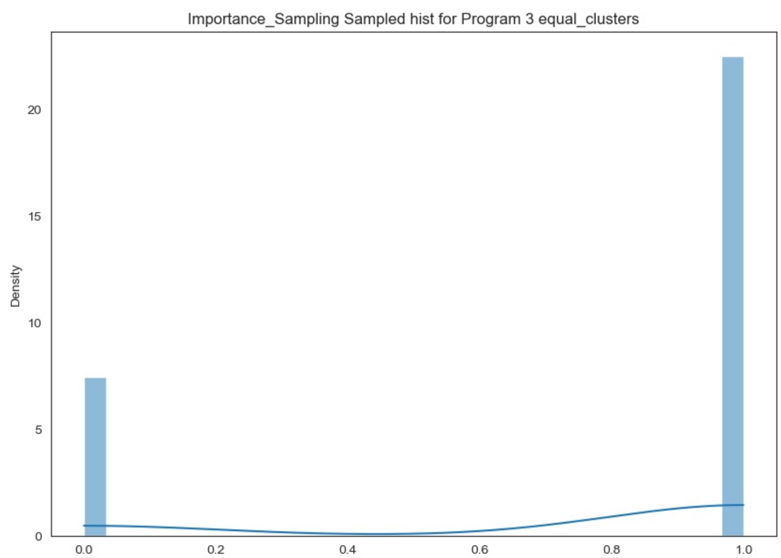


Figure 4: Histogram of posterior distribution of being in same cluster

1.2.1.4 Task 4

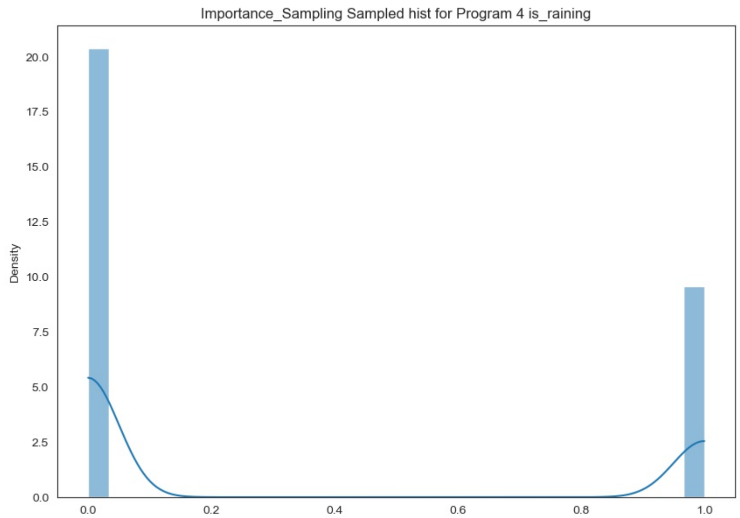


Figure 5: Histogram of posterior distribution of is_raining

1.2.1.5 Task 5

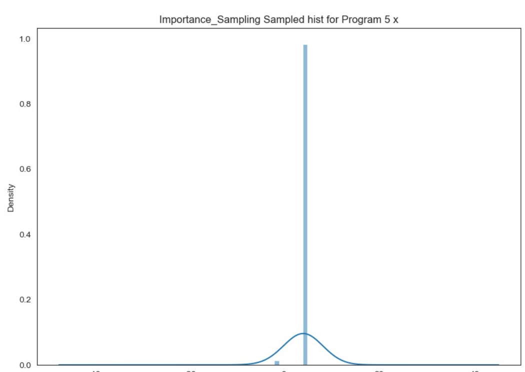


Figure 6: Histogram of posterior distribution of x

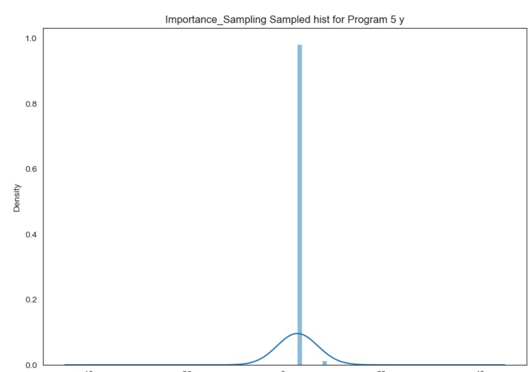


Figure 7: Histogram of posterior distribution of y

2 Trick of task 5

In order to get a good answer in this task, I have approximated the Dirac distribution with a normal distribution with the mean equal to the center of Dirac distribution with a really low variance (in case of Importance Sampling and MH within Gibbs, 10^{-5} gave me good results, and for HMC I got good results with variance equal to 10^{-3}).

3 Graphical based sampling code

3.0.1 primitives

Primitives are same as 3.0.1

3.0.2 topological sort

```python
def topological_sort(graph):
    nodes = graph[1]['V']
    edges = graph[1]['A']
    is_visited = dict.fromkeys(nodes, False)
    node_stack = []
    node_order_reverse = []
    for node in nodes:
        if not is_visited[node]:
            node_stack.append((node, False))
        while len(node_stack) > 0:
            node, flag = node_stack.pop()
            if flag:
                node_order_reverse.append(node)
                continue
            is_visited[node] = True
            node_stack.append((node, True))
            if node not in edges:
                continue
            children = edges[node]
            for child in children:
                if not is_visited[child]:
                    node_stack.append((child, False))
    return node_order_reverse[::-1]
```

Listing 9: graph_based_sampling.py - topological_sort

3.0.3 environment

```python
env = {**baseprimitives, **distlist}
```

Listing 10: graph_based_sampling.py - environment

3.0.4 deterministic eval

```python
def deterministic_eval(exp):
    "Evaluation function for the deterministic target language of the graph based
    representation."
    if isinstance(exp, list):
        if exp[0] == 'hash-map':
            exp = ['hash-map'] + [value for expression in exp[1:] for value in expression]
    return evaluate_program(exp)
```

Listing 11: graph_based_sampling.py - deterministic_eval

3.0.5 value substitution

```python
def value_subs(expressions, variables):
    if isinstance(expressions, list):
        result = []
        for expression in expressions:
            result.append(value_subs(expression, variables))
    else:
        if expressions in variables:
            result = variables[expressions]
        else:
            result = expressions
    return result
```

Listing 12: graph_based_sampling.py - value_subs

3.0.6 sample from joint

```python
def sample_from_joint(graph, var=False):
    "This function does ancestral sampling starting from the prior."
    node_order = topological_sort(graph)
    results = {}
    for node in node_order:
        first_statement, *other_statements = graph[1]['P'].get(node)
        if first_statement == 'sample*':
            dist = deterministic_eval(value_subs(other_statements, results))
            result = dist.sample()
        if first_statement == 'observe*':
            result = deterministic_eval(graph[1]['Y'].get(node))
        results[node] = result

    if var:
        return results
    else:
        return deterministic_eval(value_subs(graph[2], results))
```

Listing 13: graph_based_sampling.py - sample_from_joint

4 MH within Gibbs

Write MH within Gibbs sampler that consume the output produced by the Daphne compiler and the evaluators you wrote in completing HW 2.

4.1 Code

Provide code snippets that document critical aspects of your implementation sufficient to allow us to quickly determine whether or not you individually completed the assignment.

4.1.1 MH within Gibbs sampling

```python
def mh_within_gibbs_sampling(graph, num_samples):

    _, unobserved_variables = extract_variables(graph)
    _, free_variables_inverse = extract_free_variables(graph)

    values = [sample_from_joint(graph, var=True)]
    for _ in range(num_samples):
        values.append(gibbs_step(graph[1]['P'], unobserved_variables, values[-1],
    free_variables_inverse))

    sample_temp = deterministic_eval(value_subs(graph[2], values[0]))
    n_params = 1
    if sample_temp.dim() != 0:
        n_params = len(sample_temp)
    samples = torch.zeros(n_params, num_samples+1)

    for idx, value in enumerate(values):
        sample = deterministic_eval(value_subs(graph[2], value))
        samples[:, idx] = sample
    return samples, values
```

Listing 14: graph_based_sampling.py - mh_within_gibbs_sampling

4.1.2 extract variables

```python
def extract_variables(graph):
    observed_variables = []
    for node in graph[1]['V']:
        if graph[1]['P'].get(node)[0] == 'observe*':
            observed_variables.append(node)
    unobserved_variables = [v for v in graph[1]['V'] if v not in observed_variables]
    return observed_variables, unobserved_variables
```

Listing 15: graph_based_sampling.py - extract_variables

4.1.3 extender

```python
def extender(l):
    if isinstance(l, list):
        return sum([extender(e) for e in l], [])
    else:
        return [l]
```

Listing 16: graph_based_sampling.py - extender

4.1.4 extract free variables

```
def extract_free_variables(graph):
    free_variables = {}
    for node in graph[1]['V']:
        expressions = extender(graph[1]['P'].get(node)[1])
        for expression in expressions:
            if expression != node:
                if expression in graph[1]['V']:
                    if node in free_variables:
                        free_variables[node].append(expression)
                    else:
                        free_variables[node] = [expression]
    free_var_inverse = {}
    for node in graph[1]['V']:
        for variable in free_variables:
            if node in free_variables[variable]:
                if node not in free_var_inverse:
                    free_var_inverse[node] = []
                free_var_inverse[node].append(variable)
    return free_variables, free_var_inverse
```

Listing 17: graph_based_sampling.py - extract_free_variables

4.1.5 Gibbs step

```
def gibbs_step(p, unobserved_variables, value, free_var_inverse):
    for selected_variable in unobserved_variables:
        q = deterministic_eval(value_subs(p[selected_variable][1], value))
        value_new = value.copy()
        value_new[selected_variable] = q.sample()
        alpha = mh_accept(p, selected_variable, value_new, value, free_var_inverse)
        if alpha > torch.rand(1):
            value = value_new
    return value
```

Listing 18: graph_based_sampling.py - Gibbs_step

4.1.6 MH accept

```
def mh_accept(p, selected_variable, value_new, value_old, free_var_inverse):
    q_new = deterministic_eval(value_subs(p[selected_variable][1], value_new))
    q_old = deterministic_eval(value_subs(p[selected_variable][1], value_old))

    log_q_new = q_new.log_prob(value_old[selected_variable])
    log_q_old = q_old.log_prob(value_new[selected_variable])

    log_alpha = log_q_new - log_q_old

    Vx = free_var_inverse[selected_variable] + [selected_variable]
    for v in Vx:
        log_alpha += deterministic_eval(value_subs(p[v][1], value_new)).log_prob(value_new[v
    ])
        log_alpha -= deterministic_eval(value_subs(p[v][1], value_old)).log_prob(value_old[v
    ])
    log_alpha = torch.clip(log_alpha, max=0)
    return torch.exp(log_alpha)
```

Listing 19: graph_based_sampling.py - MH_accept

4.2 Results

I draw 10^5 samples for each task and the results are in the following:

4.2.0.1 Task 1

Time of drawing samples: **54.98 seconds**
Posterior mean of mu is: **7.2882**
Posterior variance of mu is: **0.8270**

4.2.0.2 Task 2

Time of drawing samples: **220.52 seconds**
Posterior mean of slope is: **2.1574**
Posterior variance of slope is: **0.0597**
Posterior mean of bias is: **-0.5397**
Posterior variance of bias is: **0.8999**
Posterior covariance matrix of slope and bias: $\begin{bmatrix} 0.0751 & -0.2611 \\ -0.2611 & 1.0883 \end{bmatrix}$

4.2.0.3 Task 3

This time I draw 10^4 samples. Time of drawing samples: **190.16 seconds**
Posterior mean of probability that the first and second datapoint are in the same cluster is: **0.7508**
Posterior variance of probability that the first and second datapoint are in the same cluster is: **0.1871**

4.2.0.4 Task 4

Time of drawing samples: **207.88 seconds**
Posterior mean of probability that it is raining: **0.3216**
Posterior variance of probability that it is raining: **0.2182**

4.2.0.5 Task 5

Time of drawing samples: **84.22 seconds**
Posterior marginal mean of x is: **-4.0088**
Posterior marginal variance of x is: **3.5686e-05**
Posterior marginal mean of y is: **11.0087**
Posterior marginal variance of y is: **2.1461e-04**

4.2.1 Histograms

4.2.1.1 Task 1

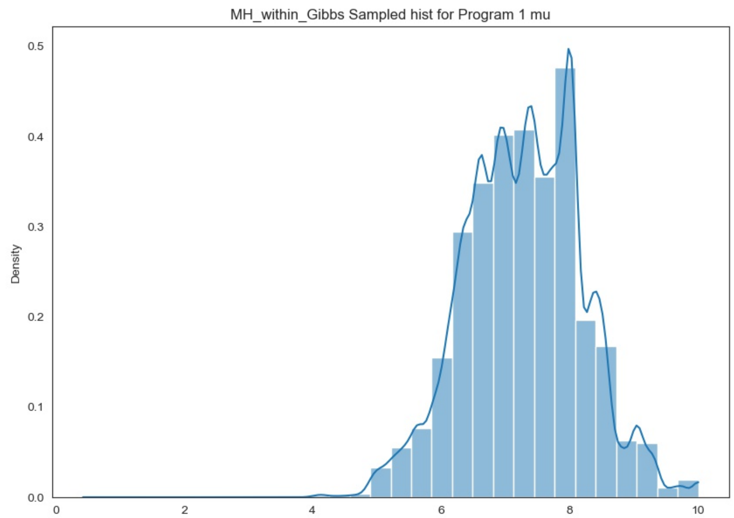


Figure 8: Histogram of posterior distribution of mu

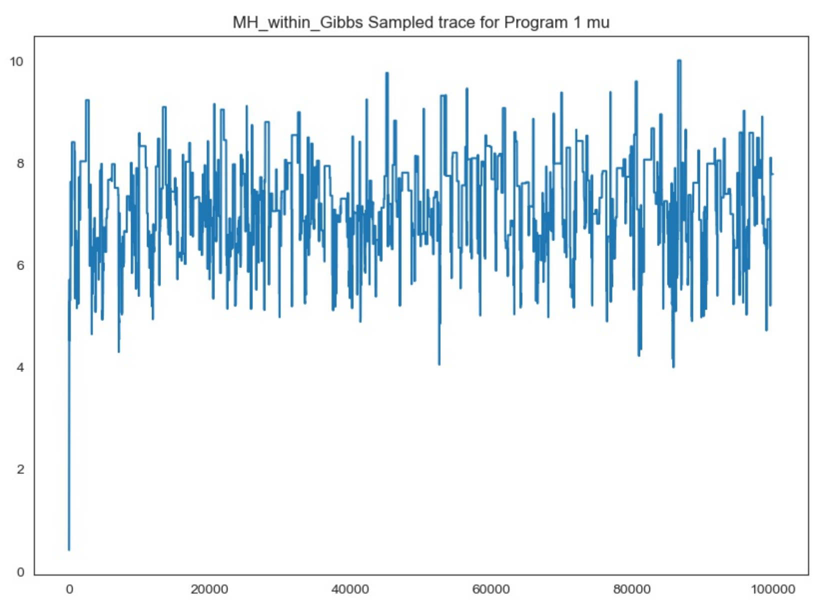


Figure 9: Sample trace plots of mu

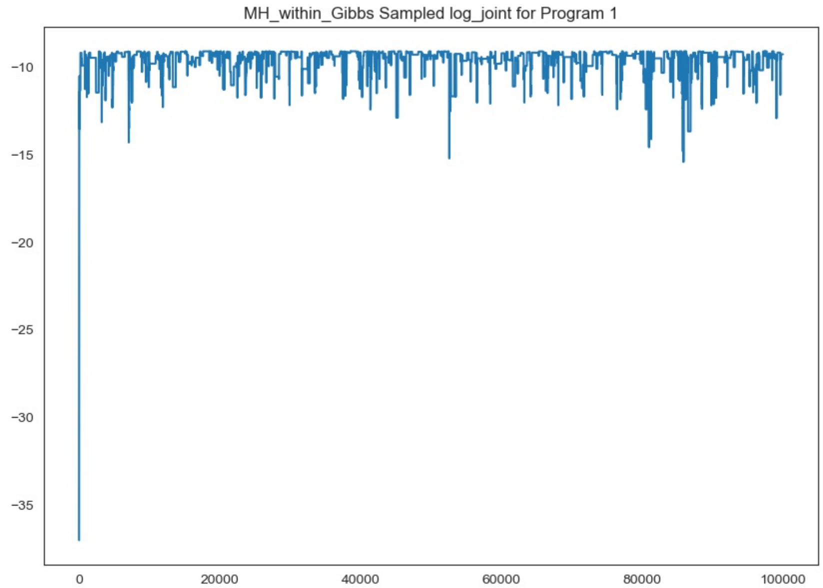


Figure 10: Joint log likelihood

4.2.1.2 Task 2

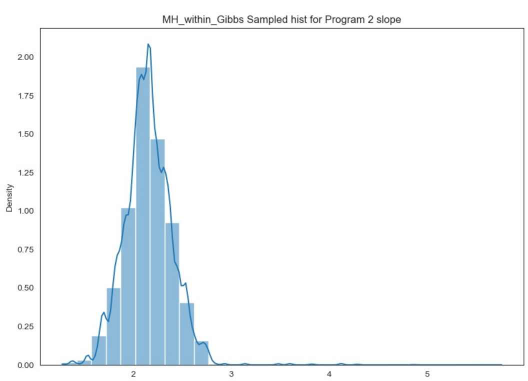


Figure 11: Histogram of posterior distribution of slope

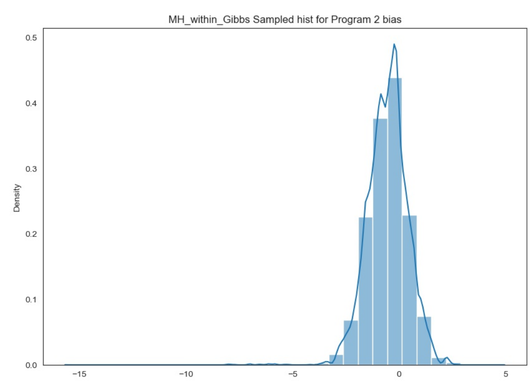


Figure 12: Histogram of posterior distribution of bias

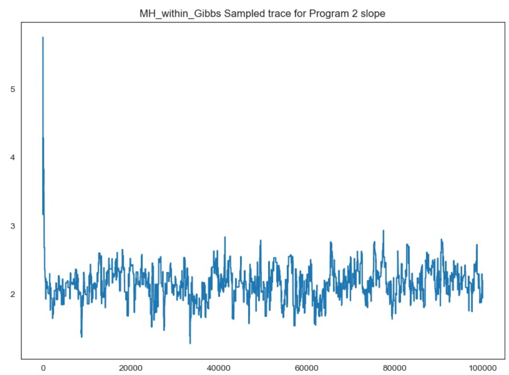


Figure 13: Sample trace plots of slope

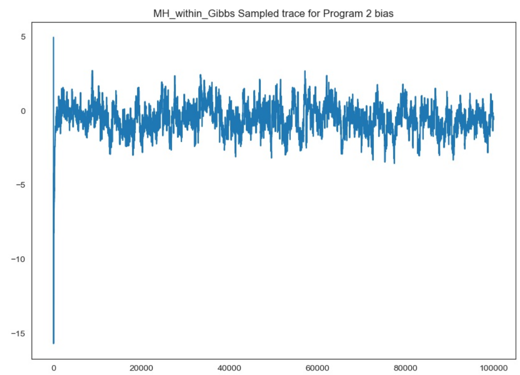


Figure 14: Sample trace plots of bias

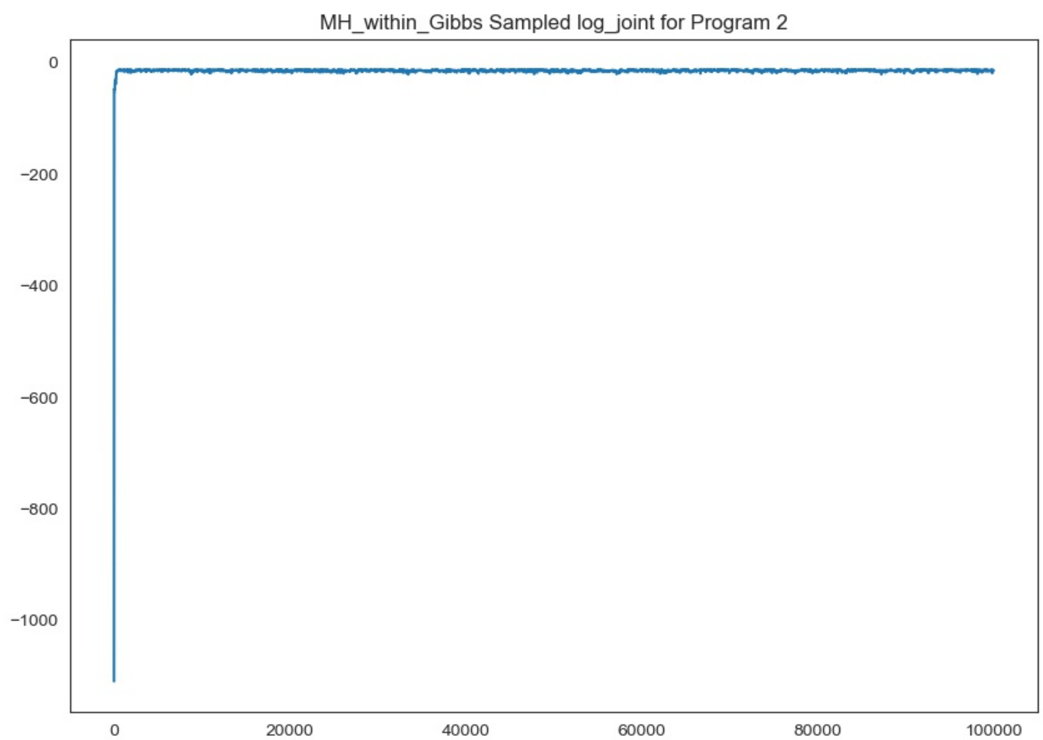


Figure 15: Joint log likelihood

4.2.1.3 Task 3

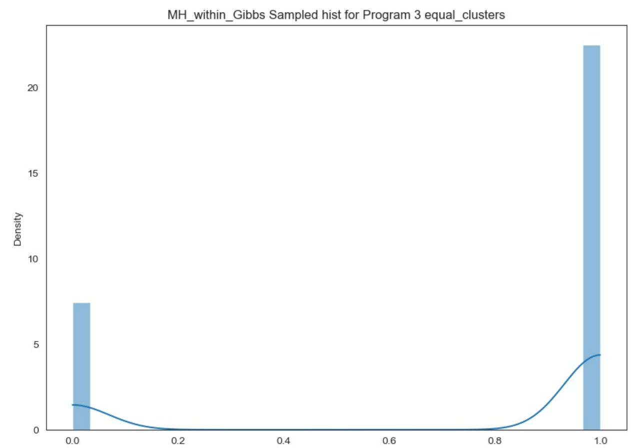


Figure 16: Histogram of posterior distribution of being in same cluster

4.2.1.4 Task 4

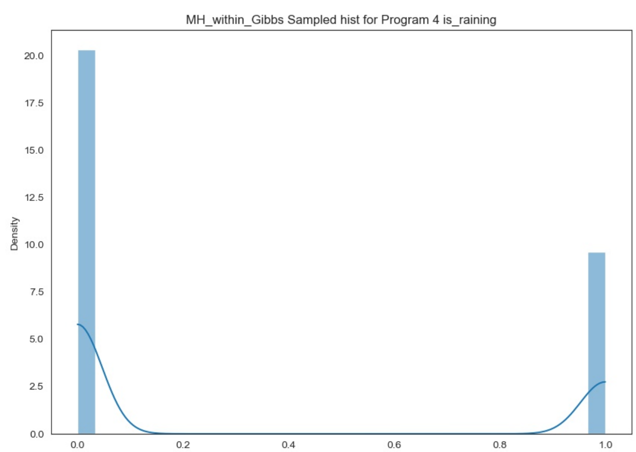


Figure 17: Histogram of posterior distribution of is_raining

4.2.1.5 Task 5

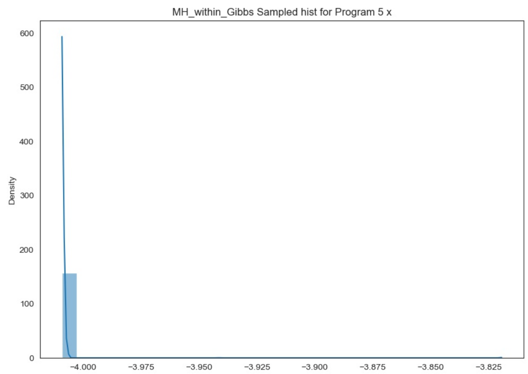


Figure 18: Histogram of posterior distribution of x

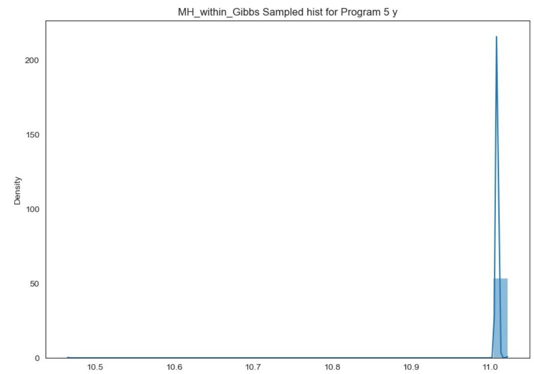


Figure 19: Histogram of posterior distribution of y

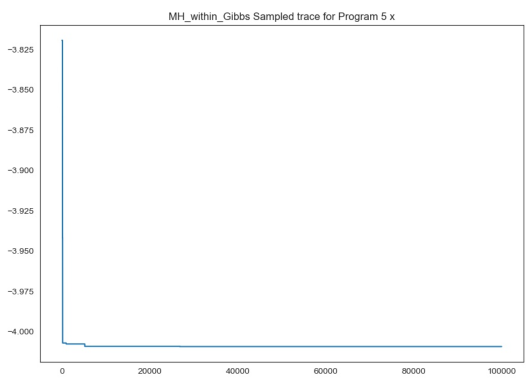


Figure 20: Sample trace plots of slope

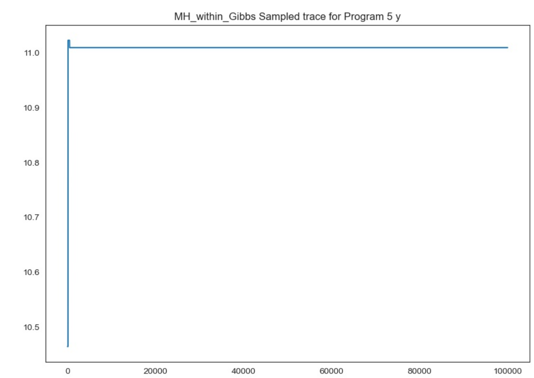


Figure 21: Sample trace plots of bias

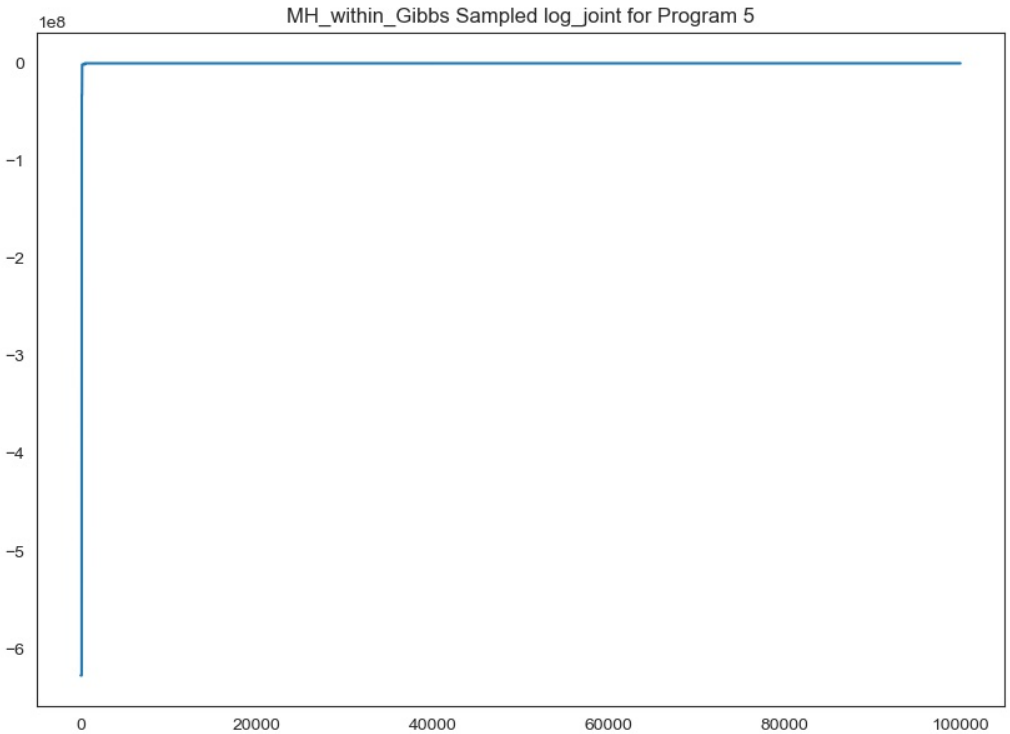


Figure 22: Joint log likelihood

5 Hamiltonian Monte Carlo

5.1 Code

5.1.1 HMC

```python
def hmc(graph, num_samples=1000, num_leapfrog_steps=10, epsilon=0.1, M=None):
    list_observed_variables, list_unobserved_variables = extract_variables(graph)
    initial_variable_values = sample_from_joint(graph, var=True)

    observed_variables = {}
    unobserved_variables = {}
    for variable in initial_variable_values:
        if variable in list_observed_variables:
            observed_variables[variable] = initial_variable_values[variable]
        else:
            unobserved_variables[variable] = initial_variable_values[variable]
            if not torch.is_tensor(unobserved_variables[variable]):
                unobserved_variables[variable] = torch.tensor(unobserved_variables[variable], dtype=torch.float64)
            else:
                unobserved_variables[variable] = unobserved_variables[variable].type(torch.float64)
            unobserved_variables[variable].requires_grad = True

    if M is None:
        M = torch.eye(len(list_unobserved_variables))

    M_inverse = torch.inverse(M)
    P = graph[1]['P']
    samples = []

    normal_generator = torch.distributions.MultivariateNormal(torch.zeros(len(M)), M)
    for _ in range(num_samples):
        r = normal_generator.sample()
        new_unobserved_variables, new_r = leapfrog(P, num_leapfrog_steps, epsilon, copy.deepcopy(unobserved_variables), observed_variables, r)
        u = torch.rand(1)
        current_energy = energy(P, M_inverse, unobserved_variables, observed_variables, r)
        new_energy = energy(P, M_inverse, new_unobserved_variables, observed_variables, new_r)

        energy_diff = current_energy - new_energy
        energy_diff_clip = torch.clip(energy_diff, max=0)
        if u < torch.exp(energy_diff_clip):
            unobserved_variables = new_unobserved_variables

        samples.append(unobserved_variables)


    sample_temp = deterministic_eval(value_subs(graph[2], samples[0]))
    n_params = 1
    if sample_temp.dim() != 0:
        n_params = len(sample_temp)
    final_samples = torch.zeros(n_params, num_samples)

    for idx, sample in enumerate(samples):
        final_sample = deterministic_eval(value_subs(graph[2], sample))
        final_samples[:, idx] = final_sample

    return final_samples, samples
```

Listing 20: graph_based_sampling.py - hmc

5.1.2 energy

```
def energy(P, M_inverse, unobserved_variables, observed_variables, r):
    K = torch.matmul(r, torch.matmul(M_inverse, r)) * 0.5

    U = 0

    all_variables = {**observed_variables, **unobserved_variables}
    for variable in all_variables:
        U = U - deterministic_eval(value_subs(P[variable][1], {**unobserved_variables, **observed_variables})).log_prob(all_variables[variable])

    return K + U
```

Listing 21: graph_based_sampling.py - energy

5.1.3 leapfrog

```
def leapfrog(P, num_leapfrog_steps, epsilon, unobserved_variables, observed_variables, r):
    r_half = r - 0.5*epsilon*grad_energy(P, unobserved_variables, observed_variables)
    new_unobserved_variables = unobserved_variables
    for _ in range(num_leapfrog_steps):
        new_unobserved_variables = detach_and_add_dict_vector(new_unobserved_variables, epsilon*r_half)
        r_half = r_half - epsilon*grad_energy(P, new_unobserved_variables, observed_variables)
    final_unobserved_variables = detach_and_add_dict_vector(new_unobserved_variables, epsilon*r_half)
    final_r = r_half - 0.5*epsilon*grad_energy(P, final_unobserved_variables, observed_variables)
    return final_unobserved_variables, final_r
```

Listing 22: graph_based_sampling.py - leapfrog

5.1.4 detach dictionary and add vector

```
def detach_and_add_dict_vector(dictionary, vector):
    new_dictionary = {}
    for i, key in enumerate(list(dictionary.keys())):
        new_dictionary[key] = dictionary[key].detach() + vector[i]
        new_dictionary[key].requires_grad = True
    return new_dictionary
```

Listing 23: graph_based_sampling.py - detach_and_add_dict_vector

5.1.5 grad energy

```
def grad_energy(P, unobserved_variables, observed_variables):
    U = 0
    for variable in observed_variables:
        U -= deterministic_eval(value_subs(P[variable][1], {**unobserved_variables, **observed_variables})).log_prob(observed_variables[variable])
    U.backward()

    U_gradients = torch.zeros(len(unobserved_variables))
    for i, key in enumerate(list(unobserved_variables.keys())):
        U_gradients[i] = unobserved_variables[key].grad
    return U_gradients
```

Listing 24: graph_based_sampling.py - grad_energy

5.2 Results

I draw 10^4 samples for each task and the results are in the following:

5.2.0.1 Task 1

Time of drawing samples: **34.80 seconds**
Posterior mean of mu is: **7.3272**
Posterior variance of mu is: **0.8059**

5.2.0.2 Task 2

Time of drawing samples: **104.60 seconds**
Posterior mean of slope is: **2.1118**
Posterior variance of slope is: **0.1792**
Posterior mean of bias is: **-0.5026**
Posterior variance of bias is: **0.8677**
Posterior covariance matrix of slope and bias: $\begin{bmatrix} 0.1792 & -0.2515 \\ -0.2515 & 0.8678 \end{bmatrix}$

5.2.0.3 Task 5

Time of drawing samples: **294.33 seconds**
Posterior marginal mean of x is: **-8.8936**
Posterior marginal variance of x is: **-2.2888e-05**
Posterior marginal mean of y is: **13.7359**
Posterior marginal variance of y is: **1.0681e-04**

5.2.1 Histograms

5.2.1.1 Task 1

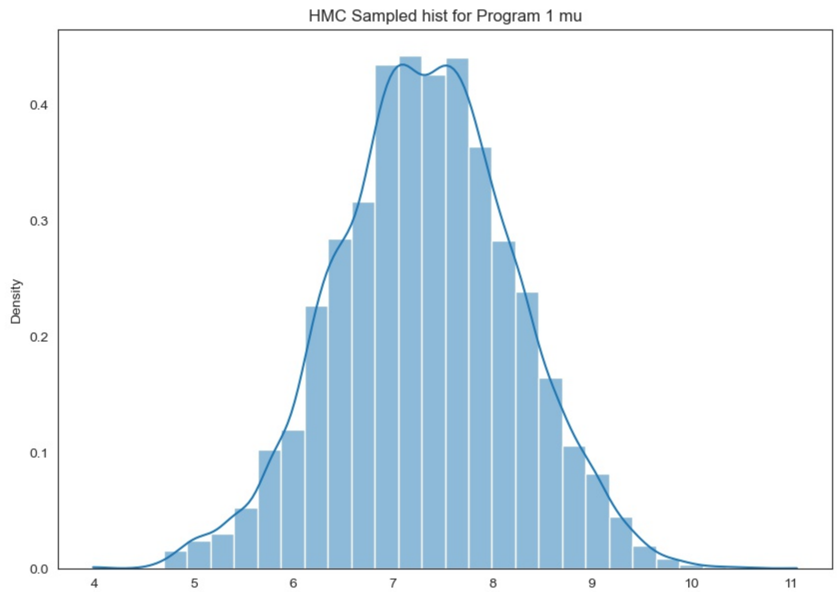


Figure 23: Histogram of posterior distribution of mu

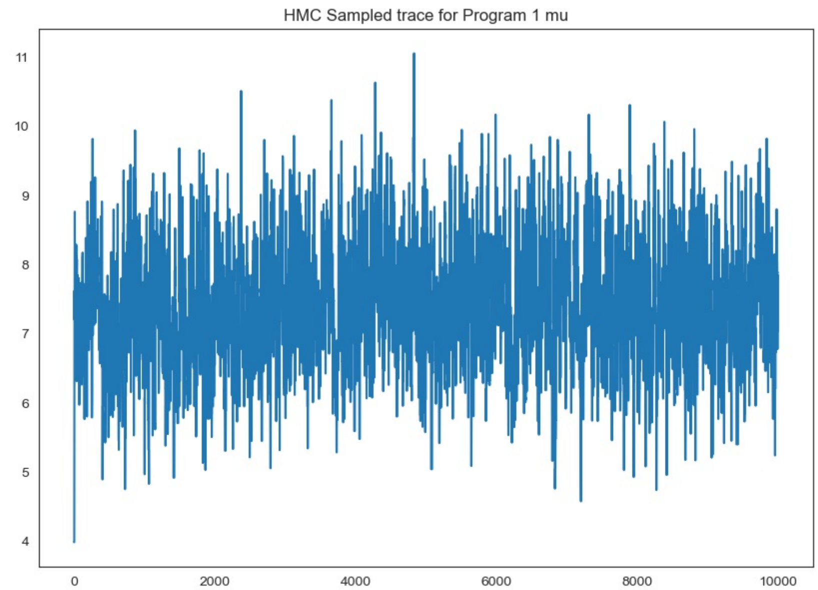


Figure 24: Sample trace plots of mu

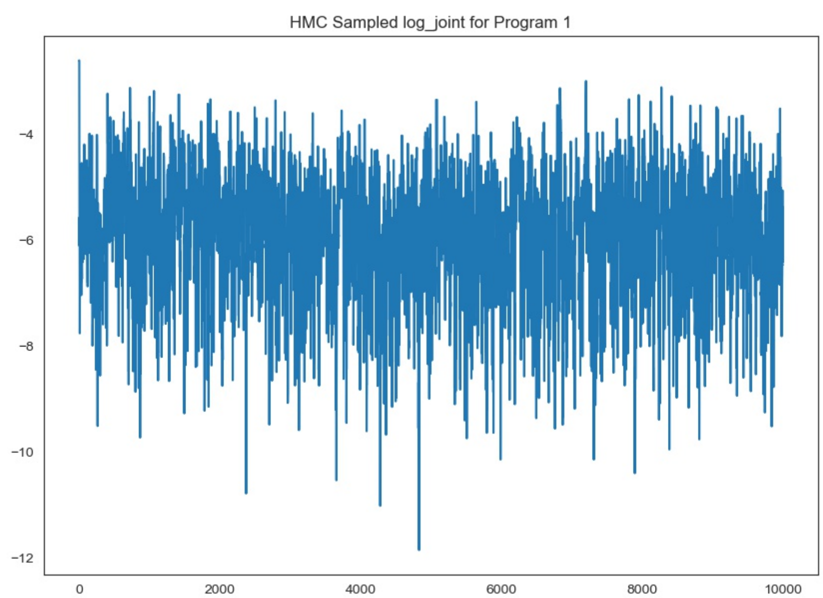


Figure 25: Joint log likelihood

5.2.1.2 Task 2

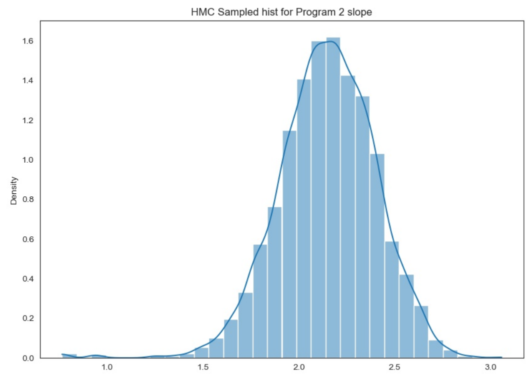


Figure 26: Histogram of posterior distribution of slope

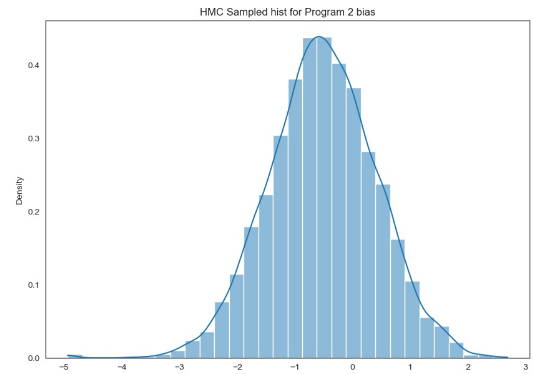


Figure 27: Histogram of posterior distribution of bias

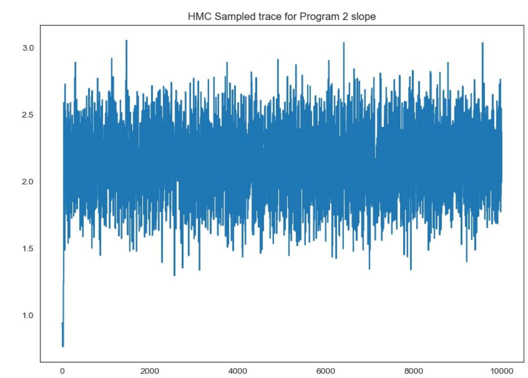


Figure 28: Sample trace plots of slope

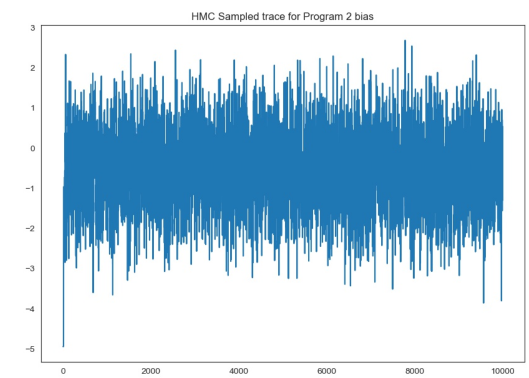


Figure 29: Sample trace plots of bias

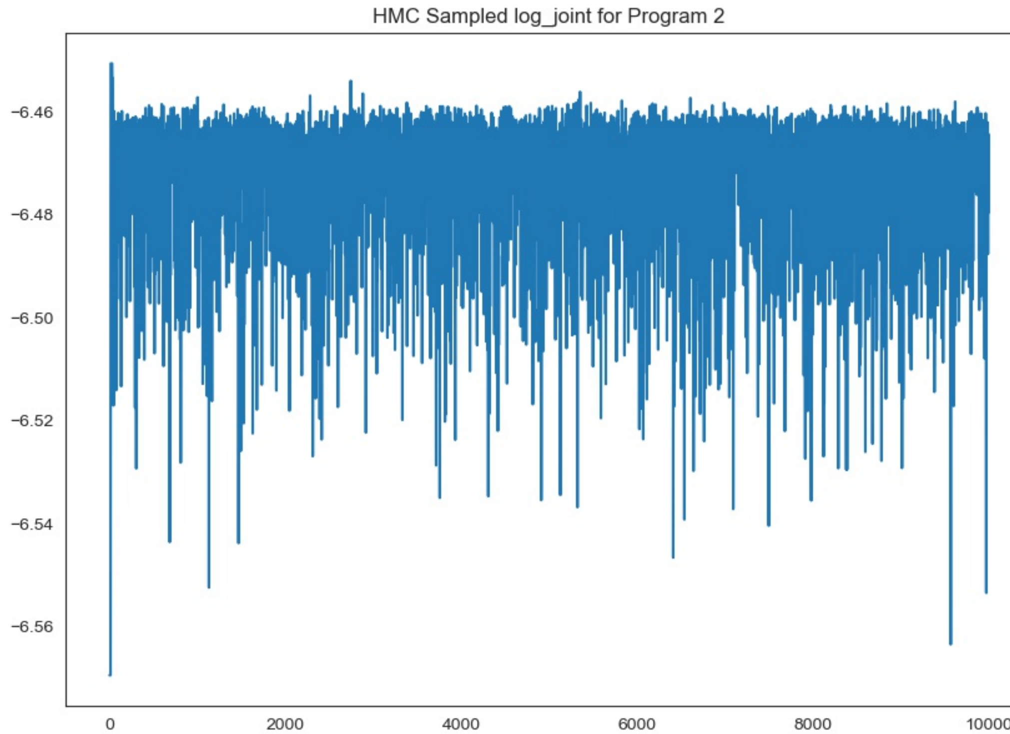


Figure 30: Joint log likelihood

5.2.1.3 Task 5

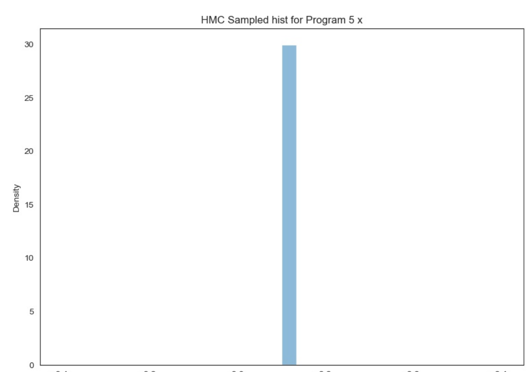


Figure 31: Histogram of posterior distribution of x

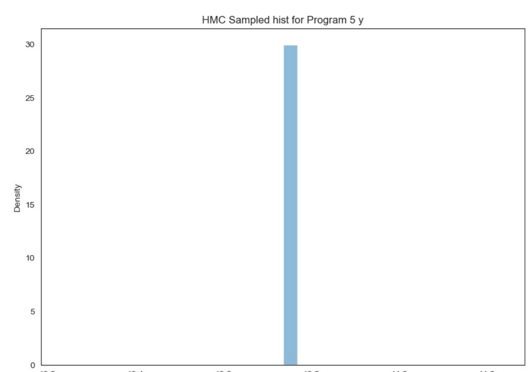


Figure 32: Histogram of posterior distribution of y

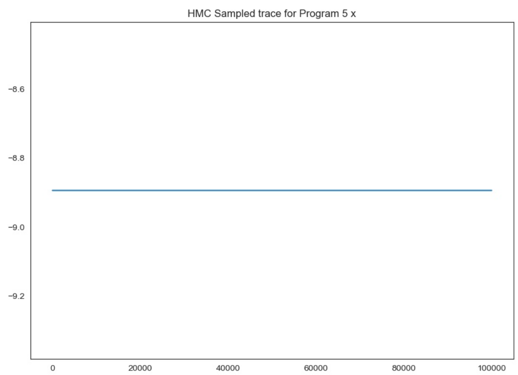


Figure 33: Sample trace plots of slope

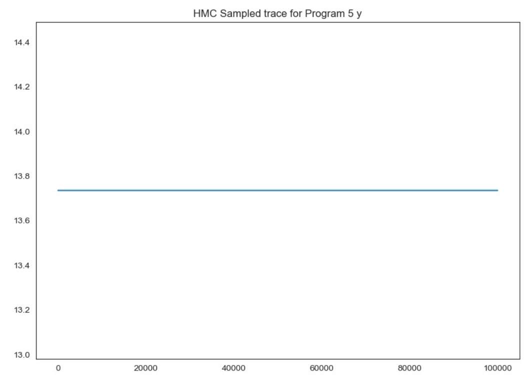


Figure 34: Sample trace plots of bias

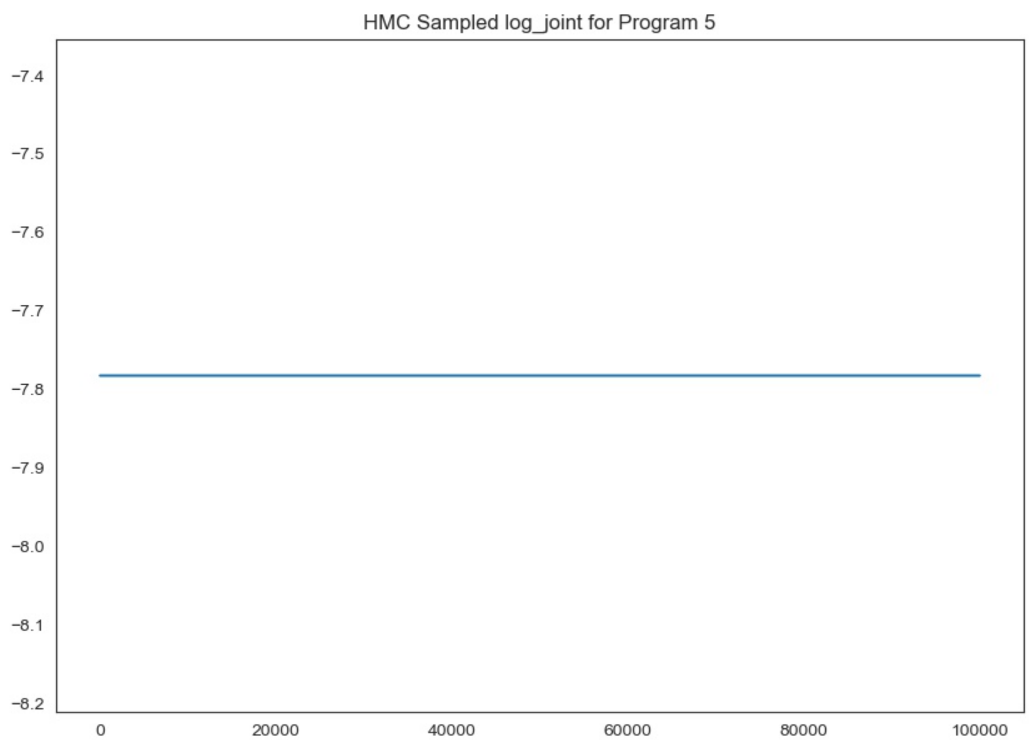


Figure 35: Joint log likelihood