Homework 3

Kevin Yang - 50244152

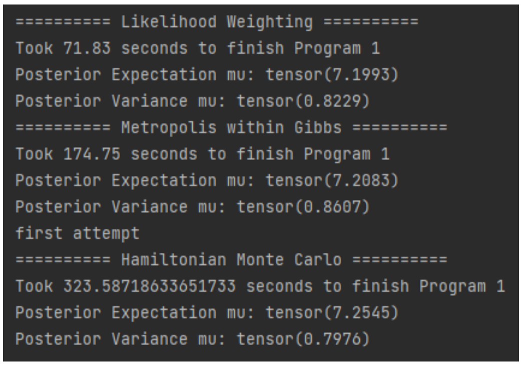
October 27, 2021

Link to repo:

https://github.com/keviny2/CPSC532W-Assignments/tree/main/FOPPL

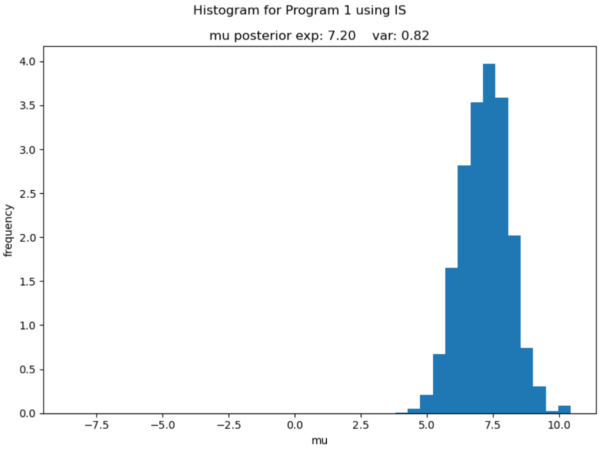
1 Program 1

1.1 Wall-Clock Time

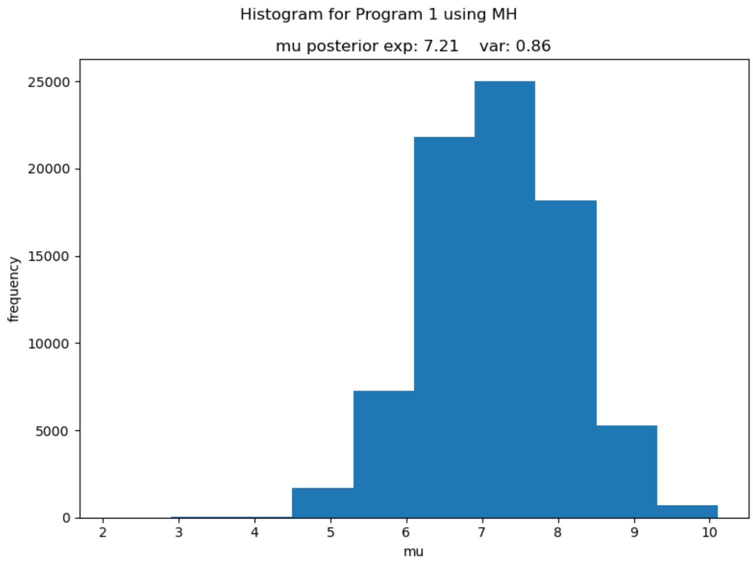


```
========== Likelihood Weighting ==========
Took 71.83 seconds to finish Program 1
Posterior Expectation mu: tensor(7.1993)
Posterior Variance mu: tensor(0.8229)
========== Metropolis within Gibbs ==========
Took 174.75 seconds to finish Program 1
Posterior Expectation mu: tensor(7.2083)
Posterior Variance mu: tensor(0.8607)
first attempt
========== Hamiltonian Monte Carlo ==========
Took 323.58718633651733 seconds to finish Program 1
Posterior Expectation mu: tensor(7.2545)
Posterior Variance mu: tensor(0.7976)
```

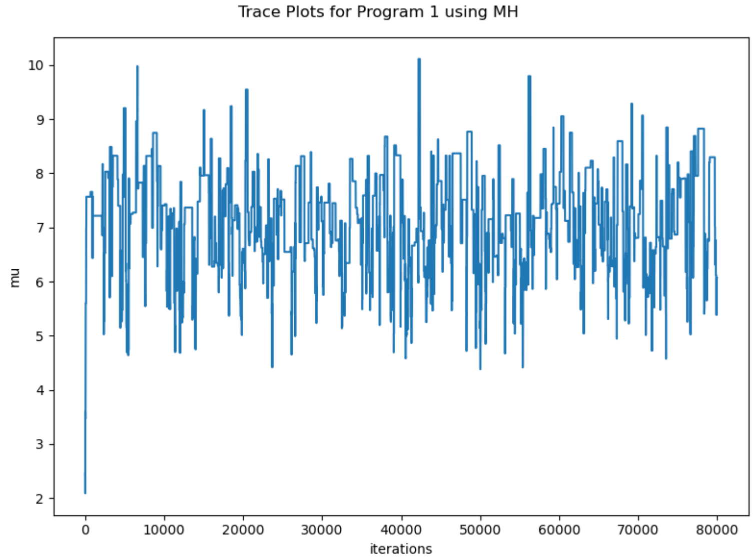
1.2 IS

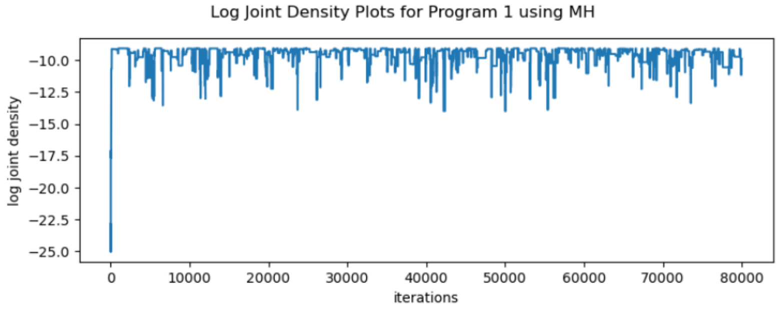


1.3 MH-Gibbs

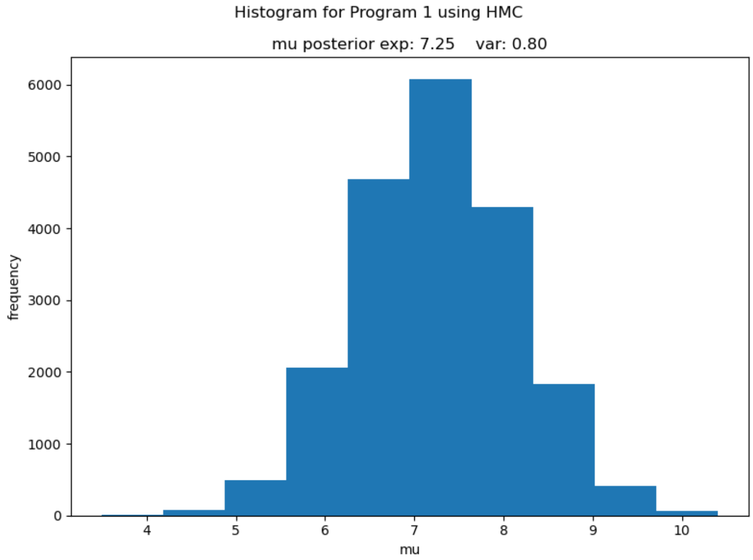

Histogram for Program 1 using MH

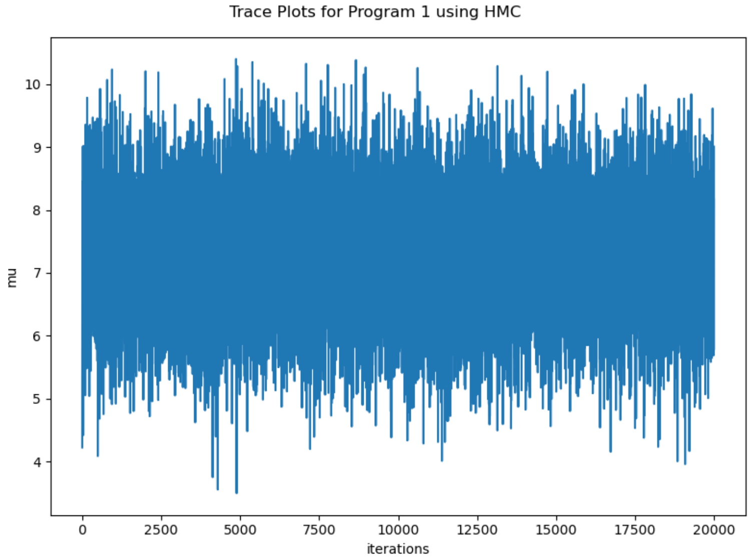
mu posterior exp: 7.21   var: 0.86

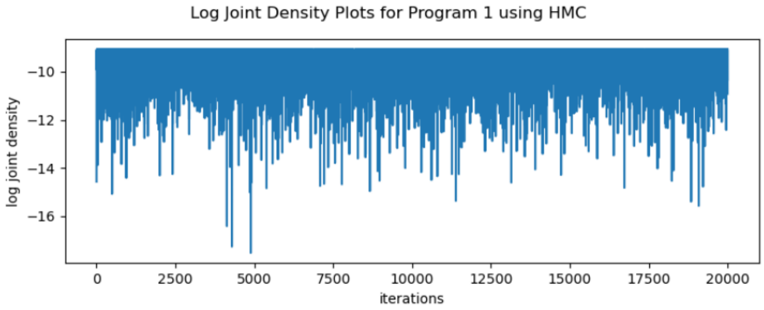

Trace Plots for Program 1 using MH


Log Joint Density Plots for Program 1 using MH

1.4 HMC

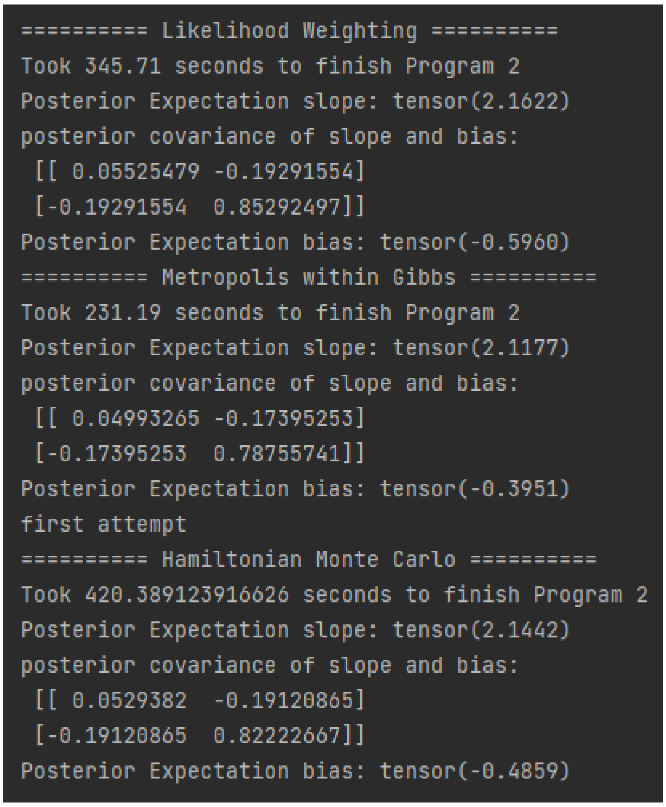

Histogram for Program 1 using HMC
mu posterior exp: 7.25 var: 0.80
frequency
0
1000
2000
3000
4000
5000
6000
4
5
6
7
8
9
10
mu


Trace Plots for Program 1 using HMC
mu
4
5
6
7
8
9
10
0
2500
5000
7500
10000
12500
15000
17500
20000
iterations


Log Joint Density Plots for Program 1 using HMC
log joint density
−10
−12
−14
−16
0
2500
5000
7500
10000
12500
15000
17500
20000
iterations

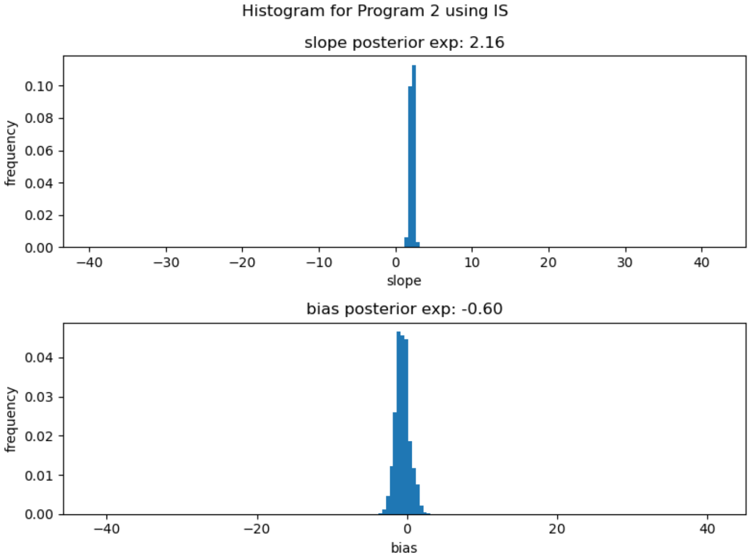
2 Program 2

2.1 Wall-Clock Time

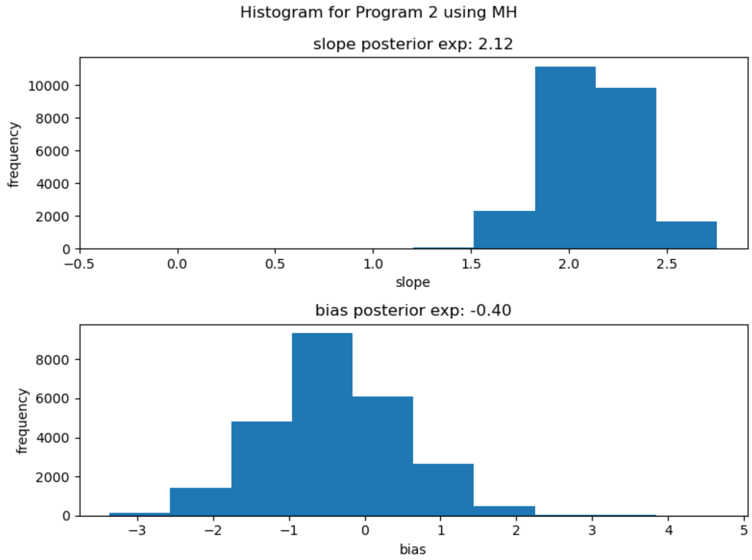


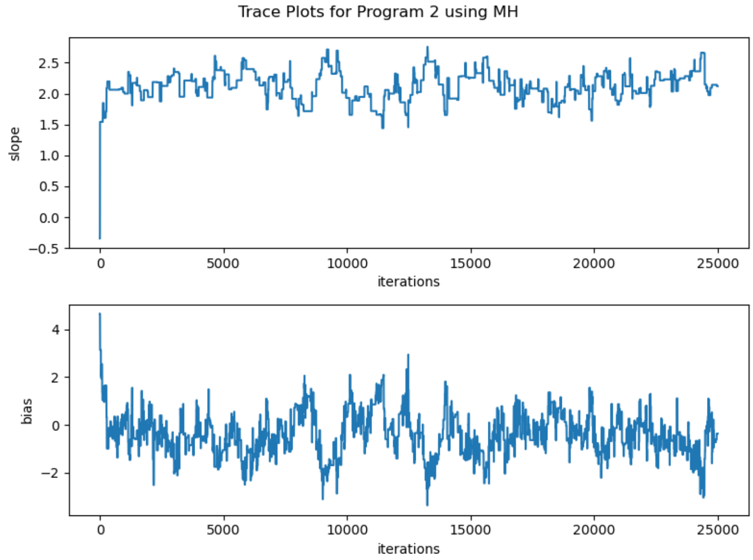
```
========== Likelihood Weighting ==========
Took 345.71 seconds to finish Program 2
Posterior Expectation slope: tensor(2.1622)
posterior covariance of slope and bias:
 [[ 0.05525479 -0.19291554]
 [-0.19291554  0.85292497]]
Posterior Expectation bias: tensor(-0.5960)
========== Metropolis within Gibbs ==========
Took 231.19 seconds to finish Program 2
Posterior Expectation slope: tensor(2.1177)
posterior covariance of slope and bias:
 [[ 0.04993265 -0.17395253]
 [-0.17395253  0.78755741]]
Posterior Expectation bias: tensor(-0.3951)
first attempt
========== Hamiltonian Monte Carlo ==========
Took 420.389123916626 seconds to finish Program 2
Posterior Expectation slope: tensor(2.1442)
posterior covariance of slope and bias:
 [[ 0.0529382  -0.19120865]
 [-0.19120865  0.82222667]]
Posterior Expectation bias: tensor(-0.4859)
```

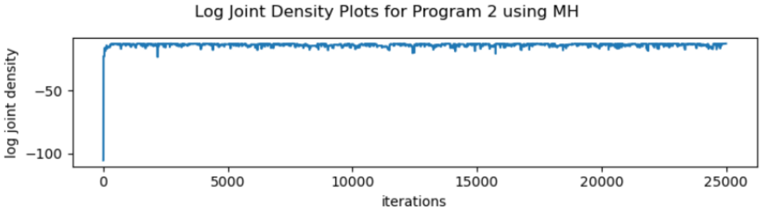
2.2 IS



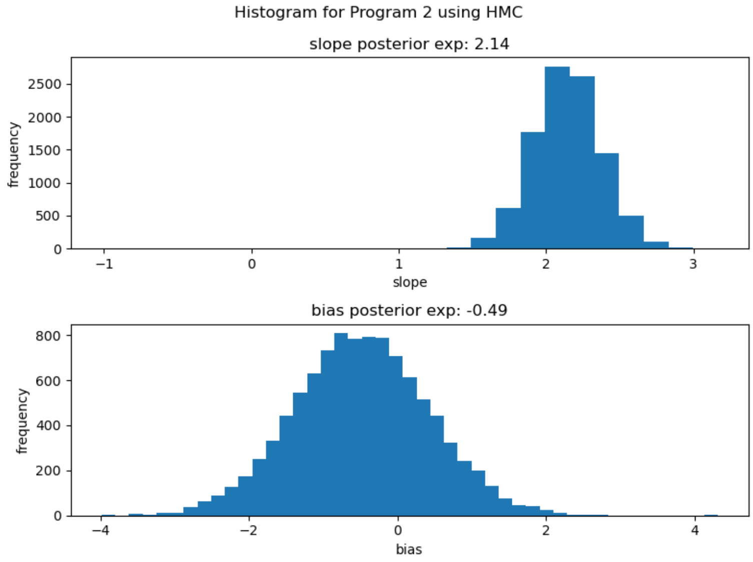
2.3 MH-Gibbs

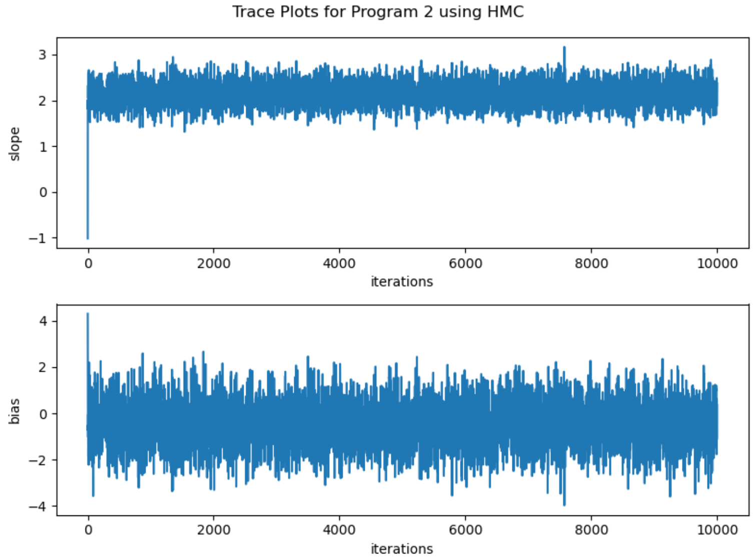

Histogram for Program 2 using MH
slope posterior exp: 2.12
frequency
10000
8000
6000
4000
2000
0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
slope
bias posterior exp: -0.40
frequency
8000
6000
4000
2000
0
−3
−2
−1
0
1
2
3
4
5
bias

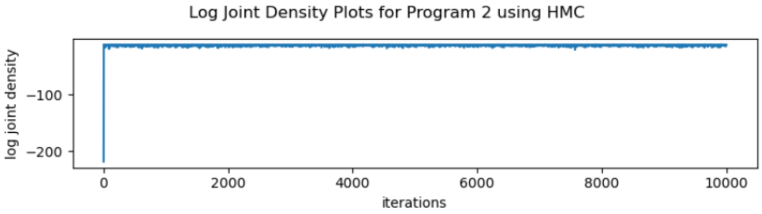

Trace Plots for Program 2 using MH
slope
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
0
5000
10000
15000
20000
25000
iterations
bias
4
2
0
−2
0
5000
10000
15000
20000
25000
iterations


Log Joint Density Plots for Program 2 using MH
log joint density
−50
−100
0
5000
10000
15000
20000
25000
iterations

2.4 HMC

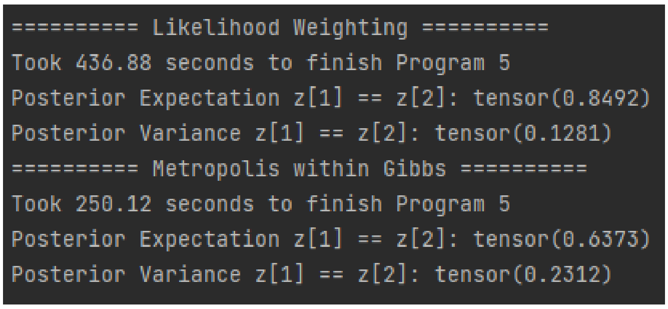

Histogram for Program 2 using HMC
slope posterior exp: 2.14
frequency
2500
2000
1500
1000
500
0
−1
0
1
2
3
slope
bias posterior exp: -0.49
frequency
800
600
400
200
0
−4
−2
0
2
4
bias


Trace Plots for Program 2 using HMC
slope
3
2
1
0
−1
0
2000
4000
6000
8000
10000
iterations
bias
4
2
0
−2
−4
0
2000
4000
6000
8000
10000
iterations


Log Joint Density Plots for Program 2 using HMC
log joint density
−100
−200
0
2000
4000
6000
8000
10000
iterations

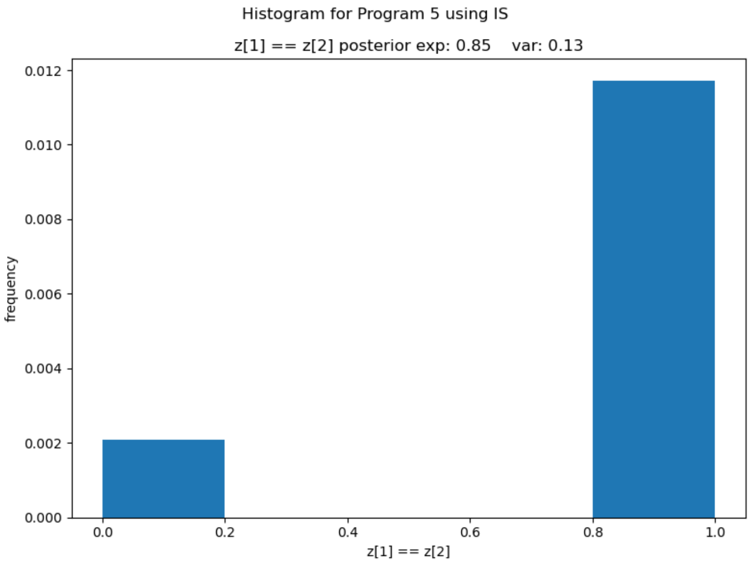
3 Program 3

3.1 Wall-Clock Time

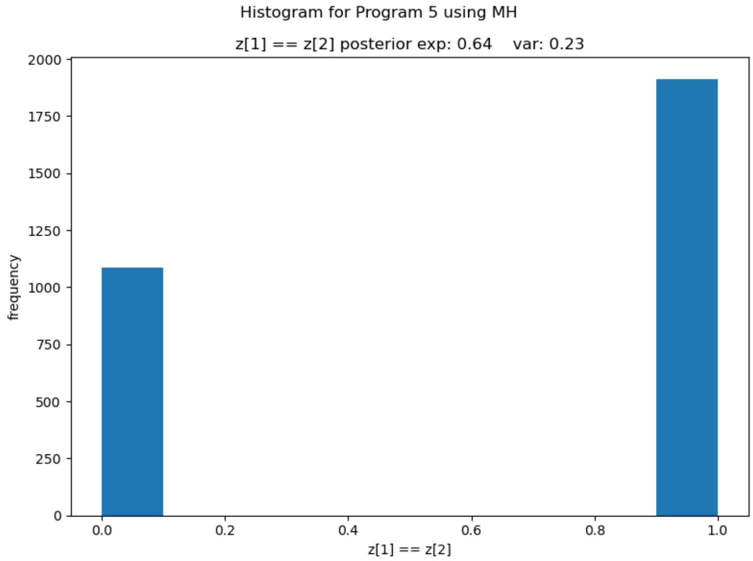


```
========== Likelihood Weighting ==========
Took 436.88 seconds to finish Program 5
Posterior Expectation z[1] == z[2]: tensor(0.8492)
Posterior Variance z[1] == z[2]: tensor(0.1281)
========== Metropolis within Gibbs ==========
Took 250.12 seconds to finish Program 5
Posterior Expectation z[1] == z[2]: tensor(0.6373)
Posterior Variance z[1] == z[2]: tensor(0.2312)
```

3.2 IS

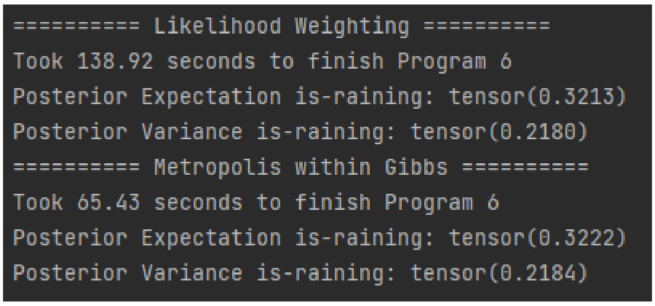


3.3 MH-Gibbs



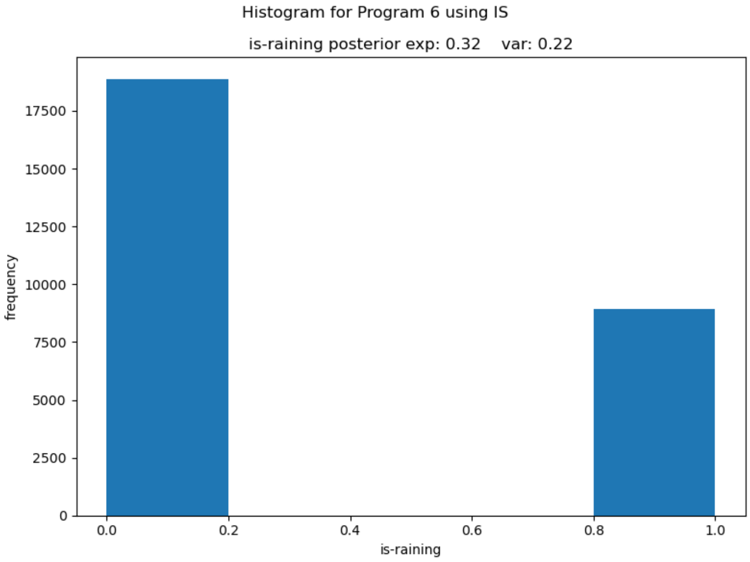
4 Program 4

4.1 Wall-Clock Time

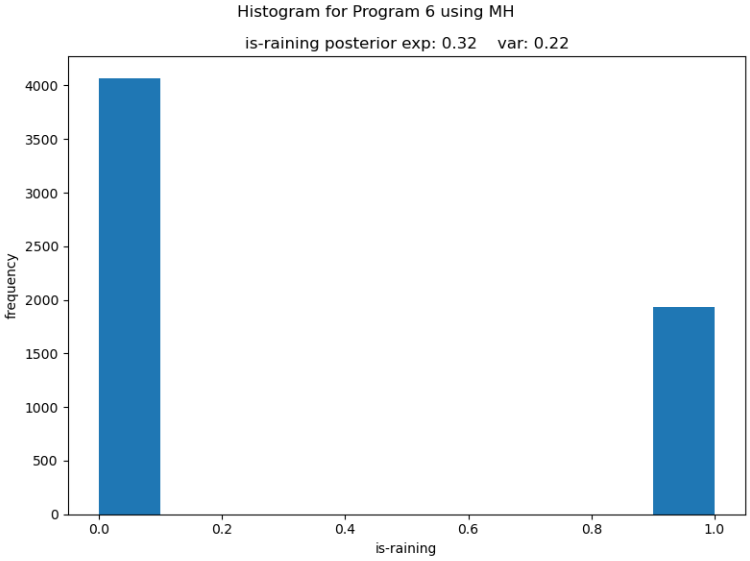


```
========== Likelihood Weighting ==========
Took 138.92 seconds to finish Program 6
Posterior Expectation is-raining: tensor(0.3213)
Posterior Variance is-raining: tensor(0.2180)
========== Metropolis within Gibbs ==========
Took 65.43 seconds to finish Program 6
Posterior Expectation is-raining: tensor(0.3222)
Posterior Variance is-raining: tensor(0.2184)
```

4.2 IS

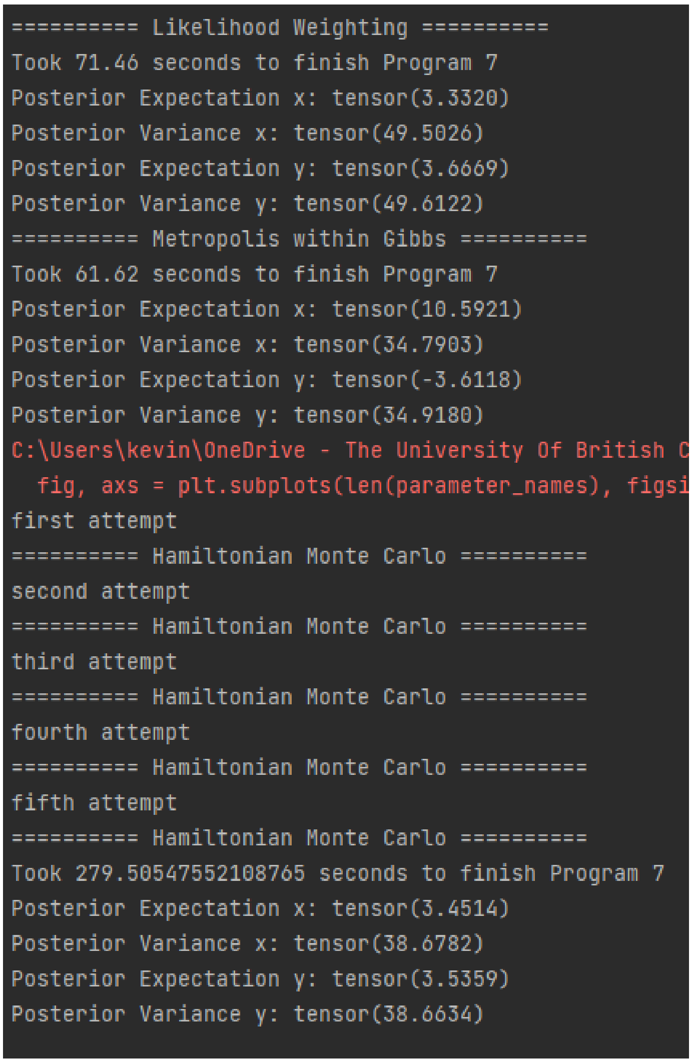


4.3 MH-Gibbs



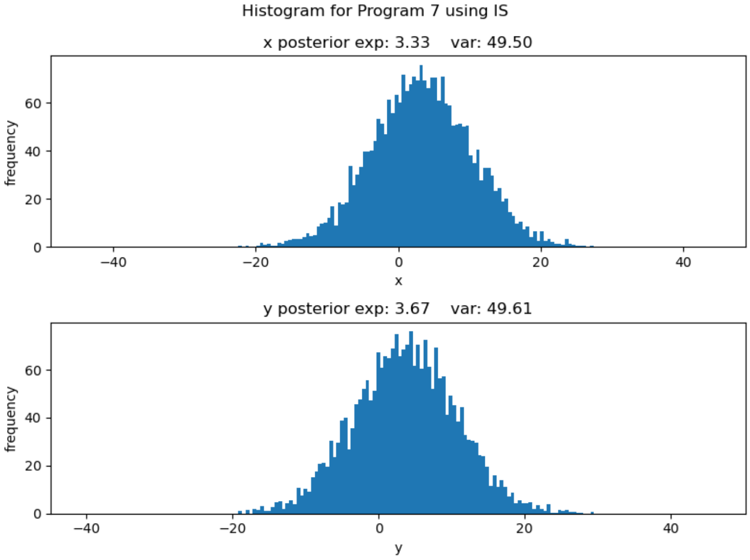
5 Program 5

5.1 Wall-Clock Time

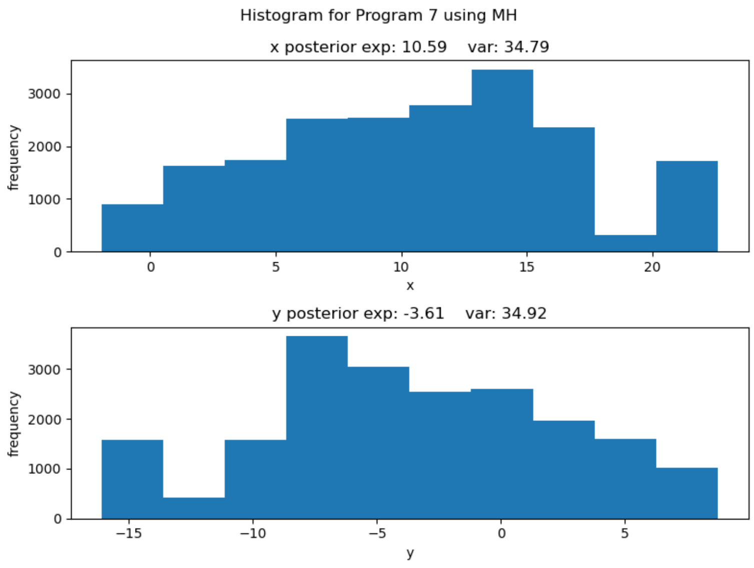


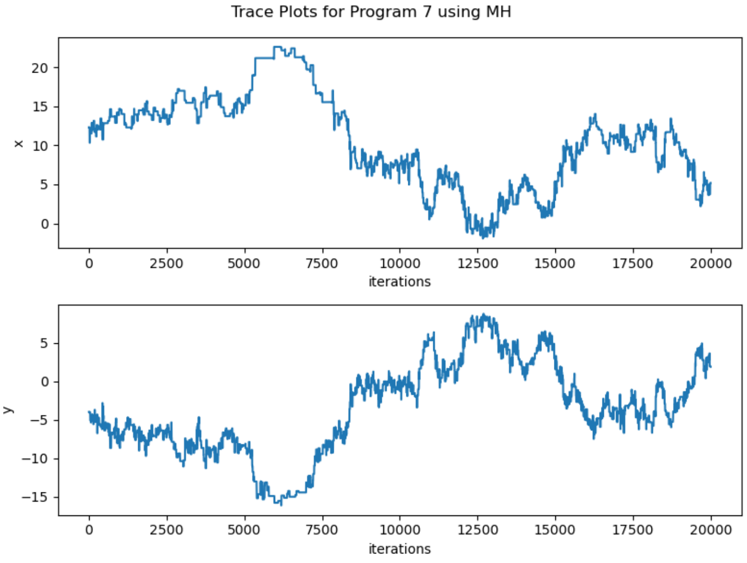
```
========== Likelihood Weighting ==========
Took 71.46 seconds to finish Program 7
Posterior Expectation x: tensor(3.3320)
Posterior Variance x: tensor(49.5026)
Posterior Expectation y: tensor(3.6669)
Posterior Variance y: tensor(49.6122)
========== Metropolis within Gibbs ==========
Took 61.62 seconds to finish Program 7
Posterior Expectation x: tensor(10.5921)
Posterior Variance x: tensor(34.7903)
Posterior Expectation y: tensor(-3.6118)
Posterior Variance y: tensor(34.9180)
C:\Users\kevin\OneDrive - The University Of British C
  fig, axs = plt.subplots(len(parameter_names), figsi
first attempt
========== Hamiltonian Monte Carlo ==========
second attempt
========== Hamiltonian Monte Carlo ==========
third attempt
========== Hamiltonian Monte Carlo ==========
fourth attempt
========== Hamiltonian Monte Carlo ==========
fifth attempt
========== Hamiltonian Monte Carlo ==========
Took 279.50547552108765 seconds to finish Program 7
Posterior Expectation x: tensor(3.4514)
Posterior Variance x: tensor(38.6782)
Posterior Expectation y: tensor(3.5359)
Posterior Variance y: tensor(38.6634)
```

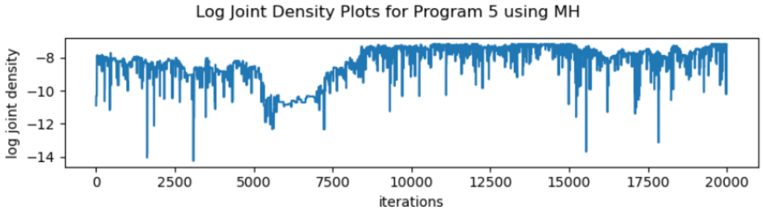
5.2 IS


Histogram for Program 7 using IS
x posterior exp: 3.33 var: 49.50
frequency
0
20
40
60
−40
−20
0
20
40
x
y posterior exp: 3.67 var: 49.61
frequency
0
20
40
60
−40
−20
0
20
40
y

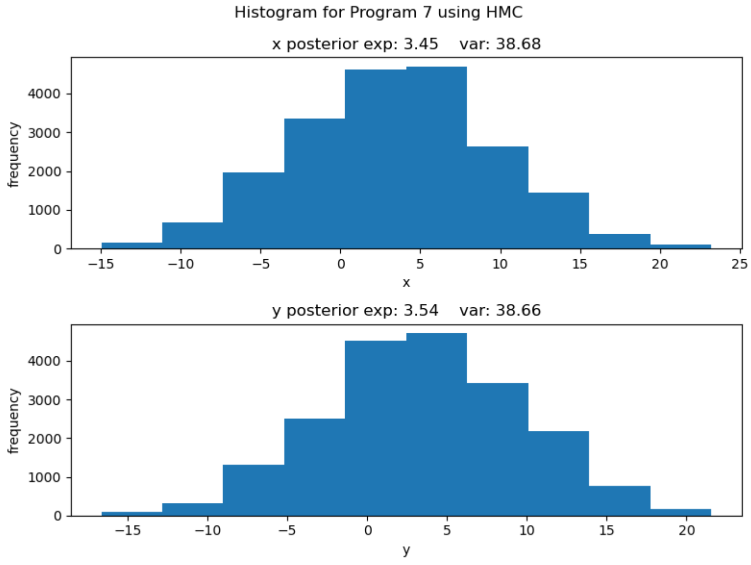
5.3 MH-Gibbs


Histogram for Program 7 using MH
x posterior exp: 10.59 var: 34.79
frequency
0
1000
2000
3000
0
5
10
15
20
x
y posterior exp: -3.61 var: 34.92
frequency
0
1000
2000
3000
−15
−10
−5
0
5
y

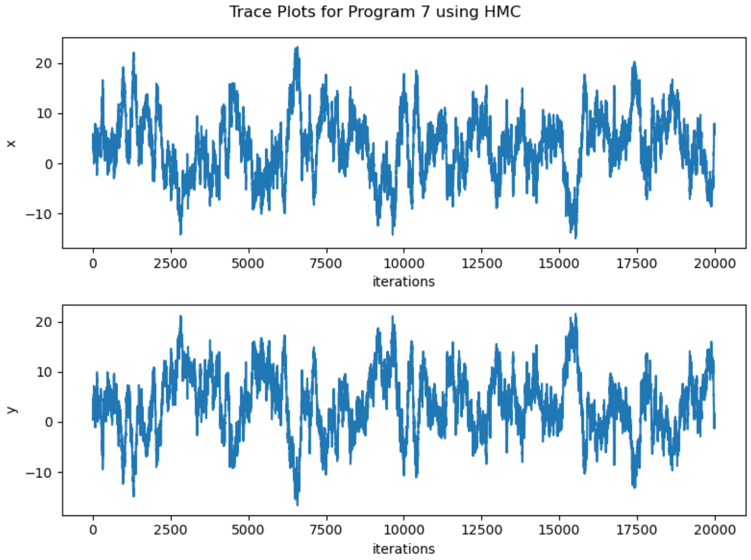

Trace Plots for Program 7 using MH
x
0
5
10
15
20
0
2500
5000
7500
10000
12500
15000
17500
20000
iterations
y
−15
−10
−5
0
5
0
2500
5000
7500
10000
12500
15000
17500
20000
iterations


Log Joint Density Plots for Program 5 using MH
log joint density
−8
−10
−12
−14
0
2500
5000
7500
10000
12500
15000
17500
20000
iterations

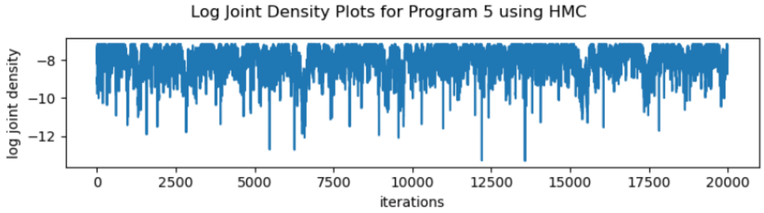
5.4 HMC



Histogram for Program 7 using HMC
x posterior exp: 3.45 var: 38.68
frequency
0
1000
2000
3000
4000
−15
−10
−5
0
5
10
15
20
25
x
y posterior exp: 3.54 var: 38.66
frequency
0
1000
2000
3000
4000
−15
−10
−5
0
5
10
15
20
y

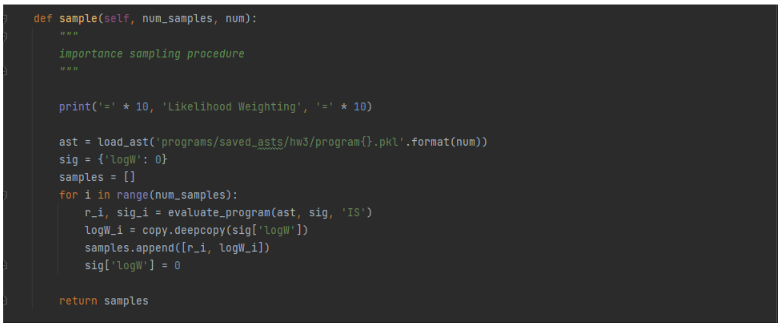


Trace Plots for Program 7 using HMC
x
20
10
0
−10
0
2500
5000
7500
10000
12500
15000
17500
20000
iterations
y
20
10
0
−10
0
2500
5000
7500
10000
12500
15000
17500
20000
iterations

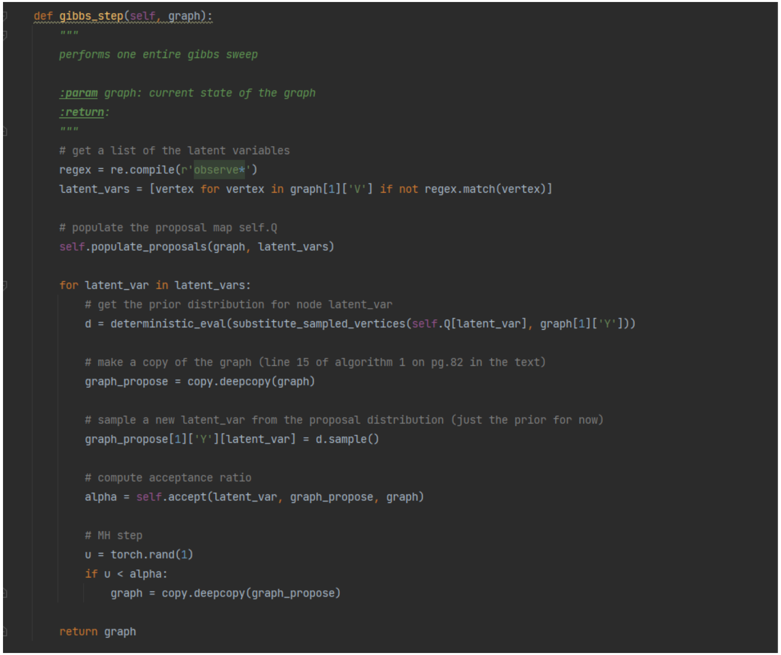


Log Joint Density Plots for Program 5 using HMC
log joint density
−8
−10
−12
0
2500
5000
7500
10000
12500
15000
17500
20000
iterations

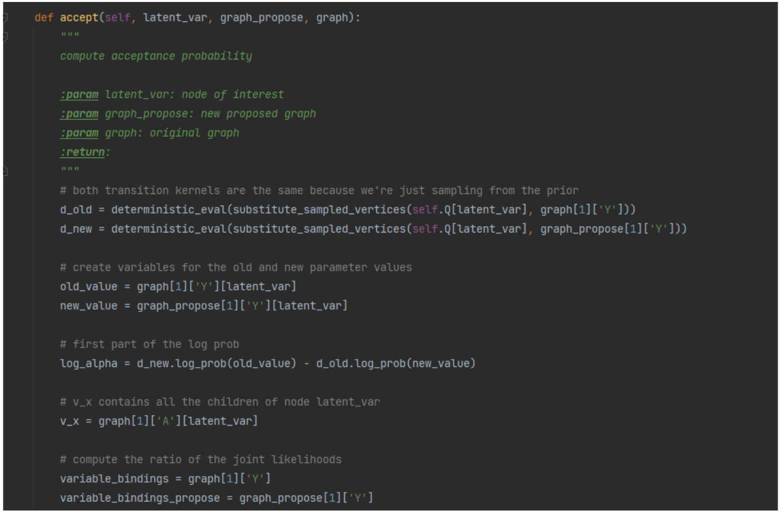
6 Code Snippets



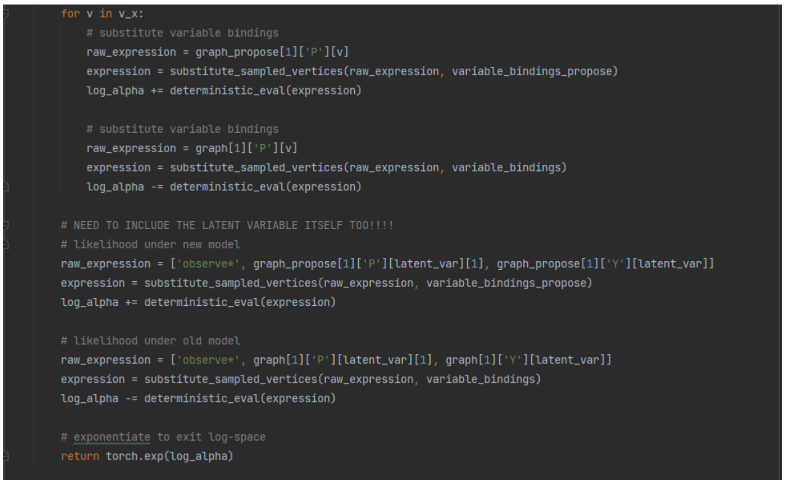
```python
def sample(self, num_samples, num):
    """
    importance sampling procedure
    """

    print('=' * 10, 'Likelihood Weighting', '=' * 10)

    ast = load_ast('programs/saved_asts/hw3/program{}.pkl'.format(num))
    sig = {'logW': 0}
    samples = []
    for i in range(num_samples):
        r_i, sig_i = evaluate_program(ast, sig, 'IS')
        logW_i = copy.deepcopy(sig['logW'])
        samples.append([r_i, logW_i])
        sig['logW'] = 0

    return samples
```



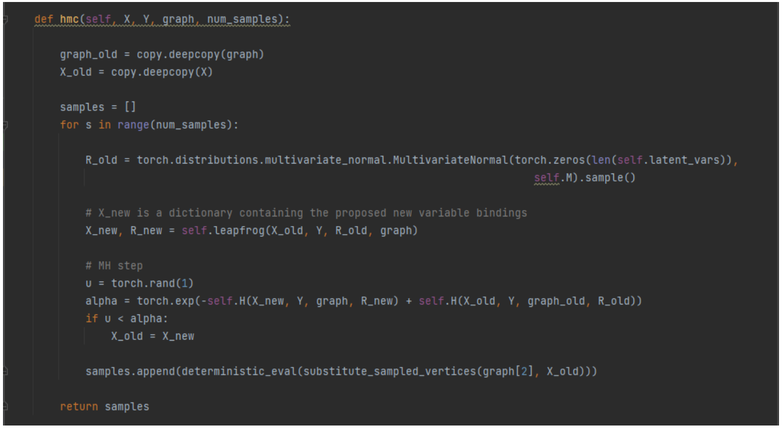
```python
def gibbs_step(self, graph):
    """
    performs one entire gibbs sweep

    :param graph: current state of the graph
    :return:
    """
    # get a list of the latent variables
    regex = re.compile(r'observe*')
    latent_vars = [vertex for vertex in graph[1]['V'] if not regex.match(vertex)]

    # populate the proposal map self.Q
    self.populate_proposals(graph, latent_vars)

    for latent_var in latent_vars:
        # get the prior distribution for node latent_var
        d = deterministic_eval(substitute_sampled_vertices(self.Q[latent_var], graph[1]['Y']))

        # make a copy of the graph (line 15 of algorithm 1 on pg.82 in the text)
        graph_propose = copy.deepcopy(graph)

        # sample a new latent_var from the proposal distribution (just the prior for now)
        graph_propose[1]['Y'][latent_var] = d.sample()

        # compute acceptance ratio
        alpha = self.accept(latent_var, graph_propose, graph)

        # MH step
        u = torch.rand(1)
        if u < alpha:
            graph = copy.deepcopy(graph_propose)

    return graph
```



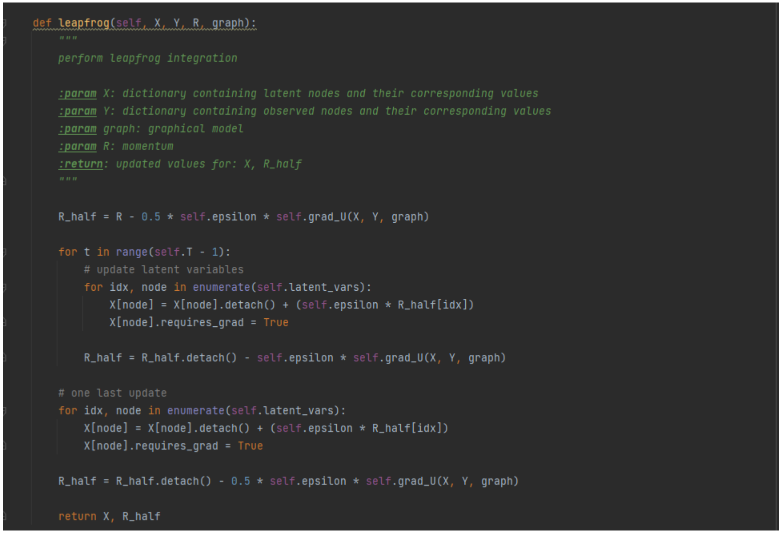
```python
def accept(self, latent_var, graph_propose, graph):
    """
    compute acceptance probability

    :param latent_var: node of interest
    :param graph_propose: new proposed graph
    :param graph: original graph
    :return:
    """
    # both transition kernels are the same because we're just sampling from the prior
    d_old = deterministic_eval(substitute_sampled_vertices(self.Q[latent_var], graph[1]['Y']))
    d_new = deterministic_eval(substitute_sampled_vertices(self.Q[latent_var], graph_propose[1]['Y']))

    # create variables for the old and new parameter values
    old_value = graph[1]['Y'][latent_var]
    new_value = graph_propose[1]['Y'][latent_var]

    # first part of the log prob
    log_alpha = d_new.log_prob(old_value) - d_old.log_prob(new_value)

    # v_x contains all the children of node latent_var
    v_x = graph[1]['A'][latent_var]

    # compute the ratio of the joint likelihoods
    variable_bindings = graph[1]['Y']
    variable_bindings_propose = graph_propose[1]['Y']
```



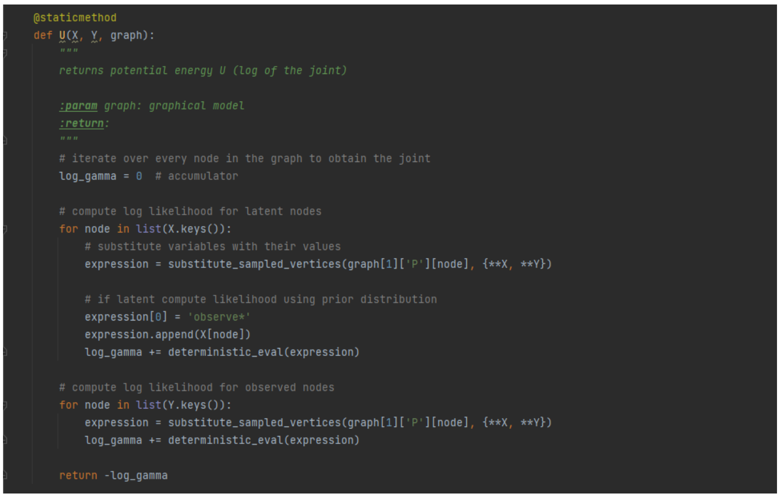
```
        for v in v_x:
            # substitute variable bindings
            raw_expression = graph_propose[1]['P'][v]
            expression = substitute_sampled_vertices(raw_expression, variable_bindings_propose)
            log_alpha += deterministic_eval(expression)

            # substitute variable bindings
            raw_expression = graph[1]['P'][v]
            expression = substitute_sampled_vertices(raw_expression, variable_bindings)
            log_alpha -= deterministic_eval(expression)

        # NEED TO INCLUDE THE LATENT VARIABLE ITSELF TOO!!!!
        # likelihood under new model
        raw_expression = ['observe*', graph_propose[1]['P'][latent_var][1], graph_propose[1]['Y'][latent_var]]
        expression = substitute_sampled_vertices(raw_expression, variable_bindings_propose)
        log_alpha += deterministic_eval(expression)

        # likelihood under old model
        raw_expression = ['observe*', graph[1]['P'][latent_var][1], graph[1]['Y'][latent_var]]
        expression = substitute_sampled_vertices(raw_expression, variable_bindings)
        log_alpha -= deterministic_eval(expression)

        # exponentiate to exit log-space
        return torch.exp(log_alpha)
```



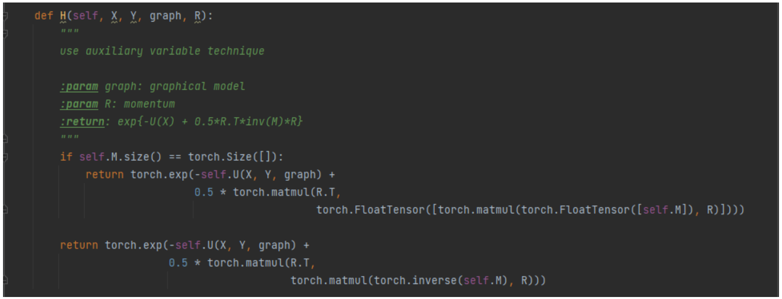
```
    def hmc(self, X, Y, graph, num_samples):

        graph_old = copy.deepcopy(graph)
        X_old = copy.deepcopy(X)

        samples = []
        for s in range(num_samples):

            R_old = torch.distributions.multivariate_normal.MultivariateNormal(torch.zeros(len(self.latent_vars)),
                                                                                  self.M).sample()

            # X_new is a dictionary containing the proposed new variable bindings
            X_new, R_new = self.leapfrog(X_old, Y, R_old, graph)

            # MH step
            u = torch.rand(1)
            alpha = torch.exp(-self.H(X_new, Y, graph, R_new) + self.H(X_old, Y, graph_old, R_old))
            if u < alpha:
                X_old = X_new

            samples.append(deterministic_eval(substitute_sampled_vertices(graph[2], X_old)))

        return samples
```



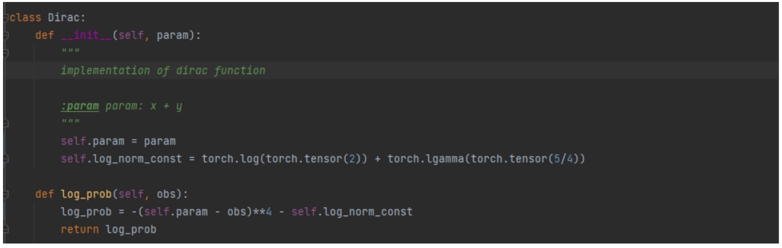
```
    def leapfrog(self, X, Y, R, graph):
        """
        perform leapfrog integration

        :param X: dictionary containing latent nodes and their corresponding values
        :param Y: dictionary containing observed nodes and their corresponding values
        :param graph: graphical model
        :param R: momentum
        :return: updated values for: X, R_half
        """

        R_half = R - 0.5 * self.epsilon * self.grad_U(X, Y, graph)

        for t in range(self.T - 1):
            # update latent variables
            for idx, node in enumerate(self.latent_vars):
                X[node] = X[node].detach() + (self.epsilon * R_half[idx])
                X[node].requires_grad = True

            R_half = R_half.detach() - self.epsilon * self.grad_U(X, Y, graph)

        # one last update
        for idx, node in enumerate(self.latent_vars):
            X[node] = X[node].detach() + (self.epsilon * R_half[idx])
            X[node].requires_grad = True

        R_half = R_half.detach() - 0.5 * self.epsilon * self.grad_U(X, Y, graph)

        return X, R_half
```



```python
    @staticmethod
    def U(X, Y, graph):
        """
        returns potential energy U (log of the joint)

        :param graph: graphical model
        :return:
        """
        # iterate over every node in the graph to obtain the joint
        log_gamma = 0  # accumulator

        # compute log likelihood for latent nodes
        for node in list(X.keys()):
            # substitute variables with their values
            expression = substitute_sampled_vertices(graph[1]['P'][node], {**X, **Y})

            # if latent compute likelihood using prior distribution
            expression[0] = 'observe*'
            expression.append(X[node])
            log_gamma += deterministic_eval(expression)

        # compute log likelihood for observed nodes
        for node in list(Y.keys()):
            expression = substitute_sampled_vertices(graph[1]['P'][node], {**X, **Y})
            log_gamma += deterministic_eval(expression)

        return -log_gamma
```



```python
    def H(self, X, Y, graph, R):
        """
        use auxiliary variable technique

        :param graph: graphical model
        :param R: momentum
        :return: exp{-U(X) + 0.5*R.T*inv(M)*R}
        """
        if self.M.size() == torch.Size([]):
            return torch.exp(-self.U(X, Y, graph) +
                             0.5 * torch.matmul(R.T,
                                                torch.FloatTensor([torch.matmul(torch.FloatTensor([self.M]), R)])))

        return torch.exp(-self.U(X, Y, graph) +
                         0.5 * torch.matmul(R.T,
                                            torch.matmul(torch.inverse(self.M), R)))
```



```python
class Dirac:
    def __init__(self, param):
        """
        implementation of dirac function

        :param param: x + y
        """
        self.param = param
        self.log_norm_const = torch.log(torch.tensor(2)) + torch.lgamma(torch.tensor(5/4))

    def log_prob(self, obs):
        log_prob = -(self.param - obs)**4 - self.log_norm_const
        return log_prob
```