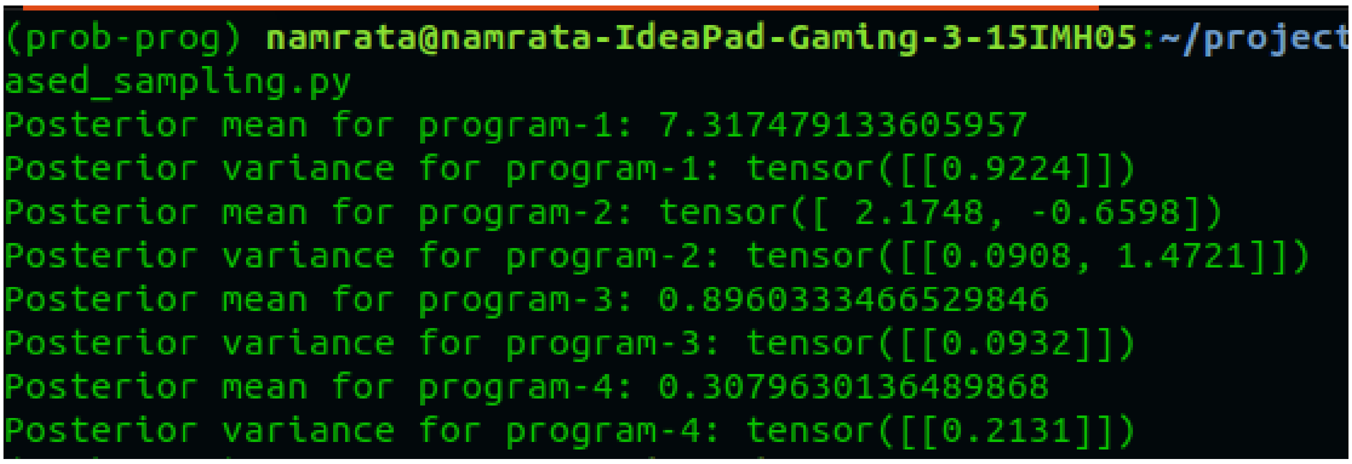
CPSC532W: Probabilistic Programming, Homework 3

Namrata Deka

My code for this homework can be found here: https://github.com/namratadeka/cpsc532W/tree/main/HW3.

1. Importance Sampling



```
(prob-prog) namrata@namrata-IdeaPad-Gaming-3-15IMH05:~/project
ased_sampling.py
Posterior mean for program-1: 7.317479133605957
Posterior variance for program-1: tensor([[0.9224]])
Posterior mean for program-2: tensor([ 2.1748, -0.6598])
Posterior variance for program-2: tensor([[0.0908, 1.4721]])
Posterior mean for program-3: 0.8960333466529846
Posterior variance for program-3: tensor([[0.0932]])
Posterior mean for program-4: 0.3079630136489868
Posterior variance for program-4: tensor([[0.2131]])
```

Figure 1: Importance Sampling Posterior Means and Variances

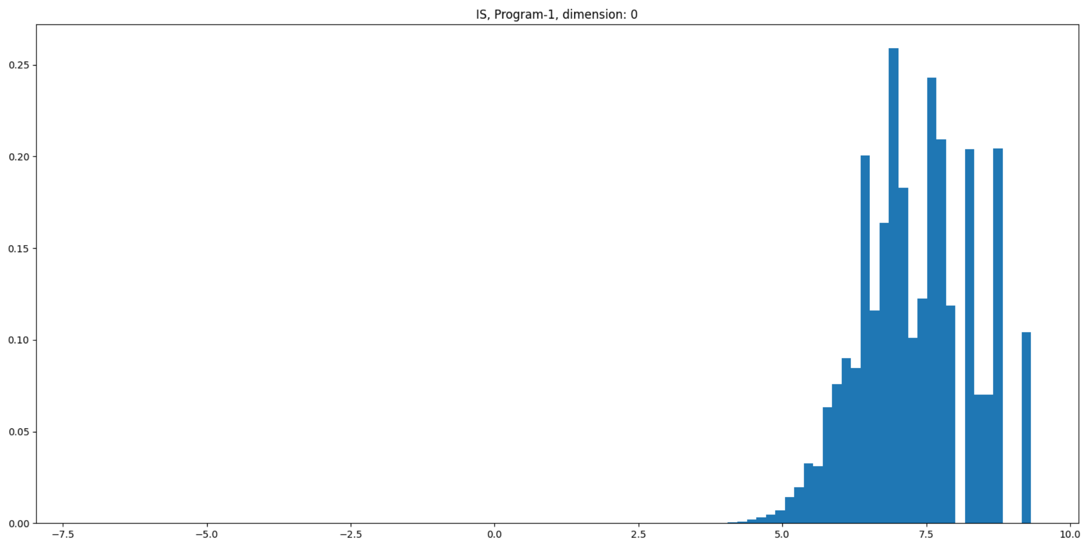


Figure 2: Weighted histogram for samples from program-1

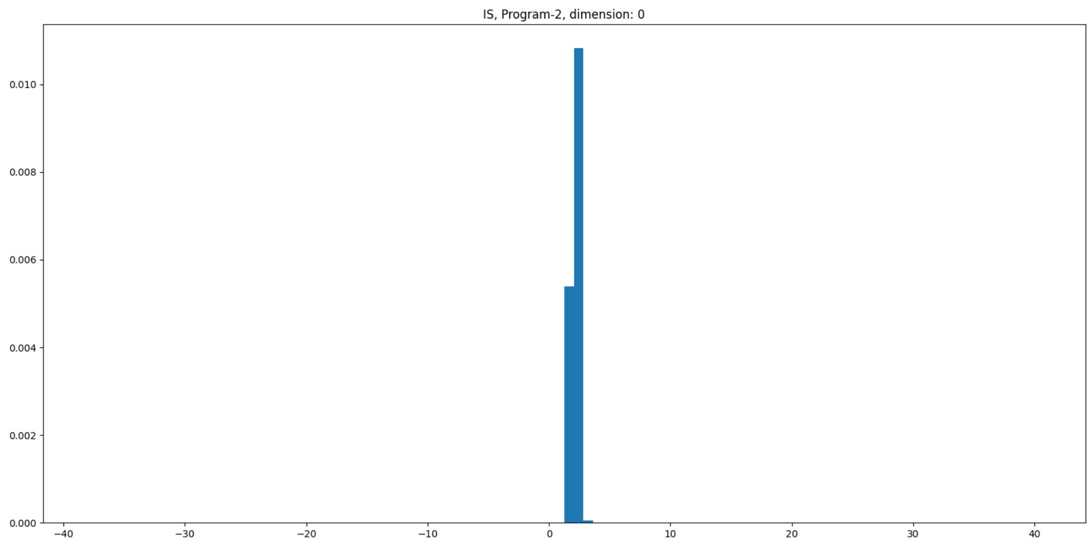


Figure 3: Weighted histogram for slope samples from program-2

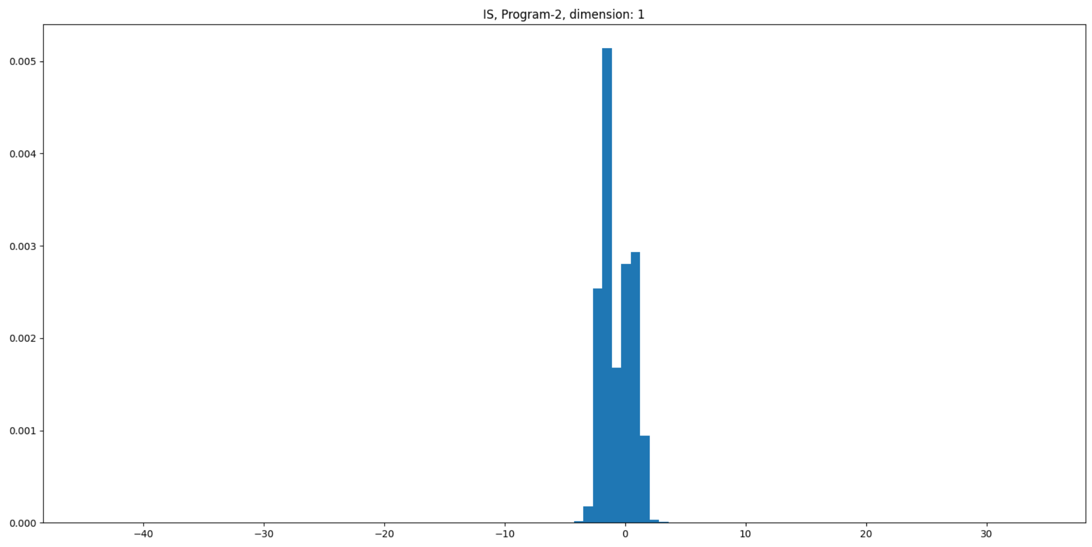


Figure 4: Weighted histogram for bias samples from program-2

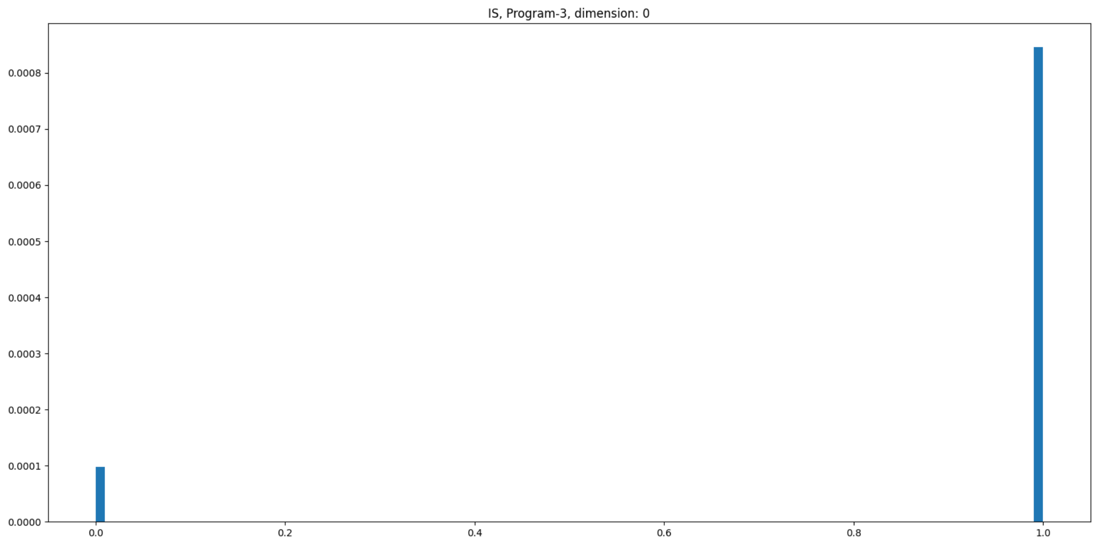


Figure 5: Weighted histogram for samples from program-3

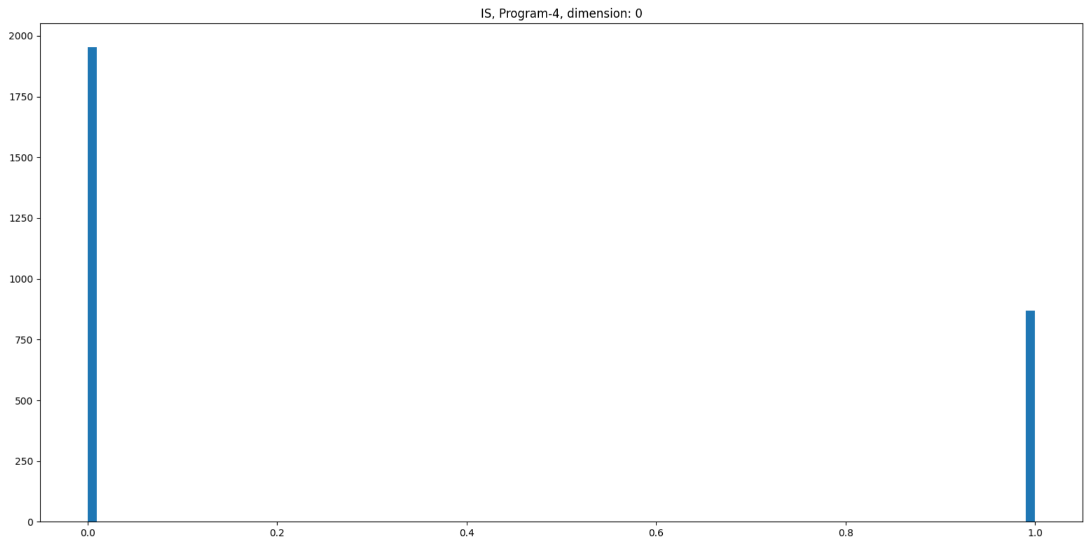
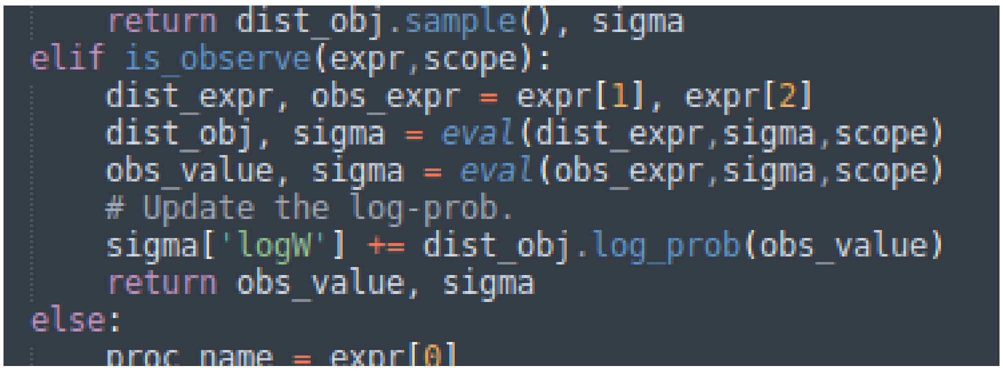
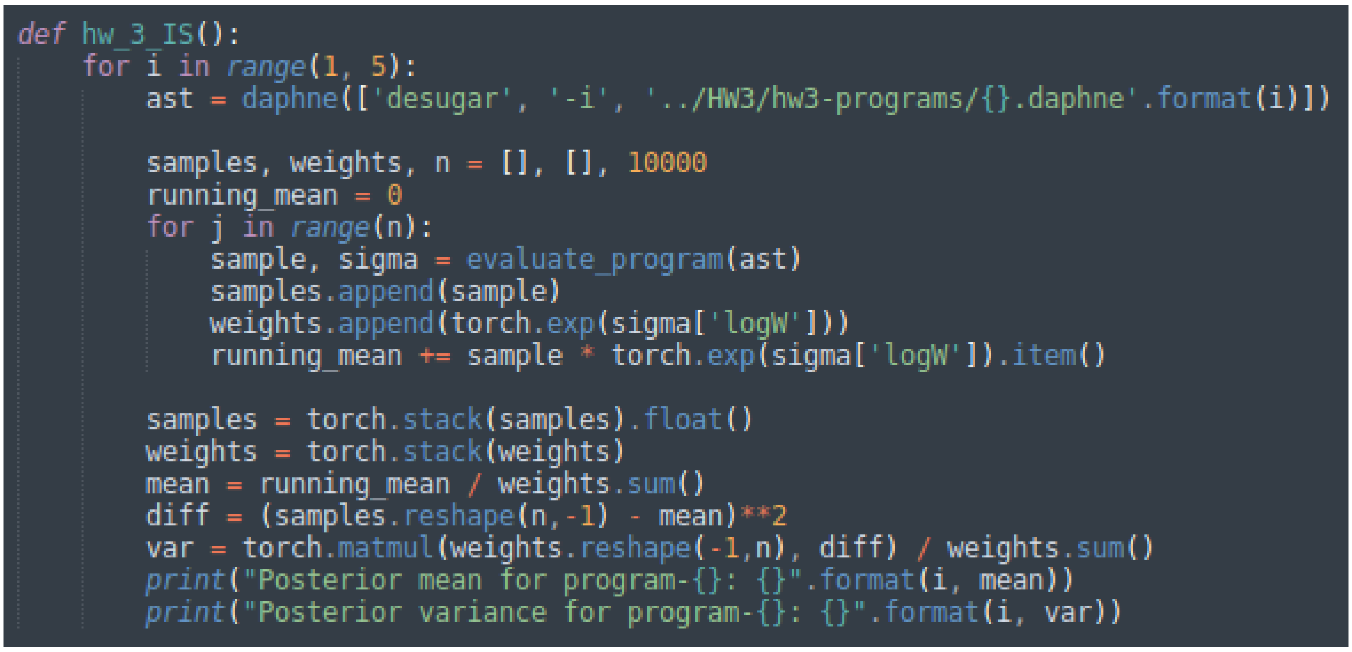


Figure 6: Weighted histogram for samples from program-4



```
        return dist_obj.sample(), sigma
    elif is_observe(expr,scope):
        dist_expr, obs_expr = expr[1], expr[2]
        dist_obj, sigma = eval(dist_expr,sigma,scope)
        obs_value, sigma = eval(obs_expr,sigma,scope)
        # Update the log-prob.
        sigma['logW'] += dist_obj.log_prob(obs_value)
        return obs_value, sigma
    else:
        proc_name = expr[0]
```

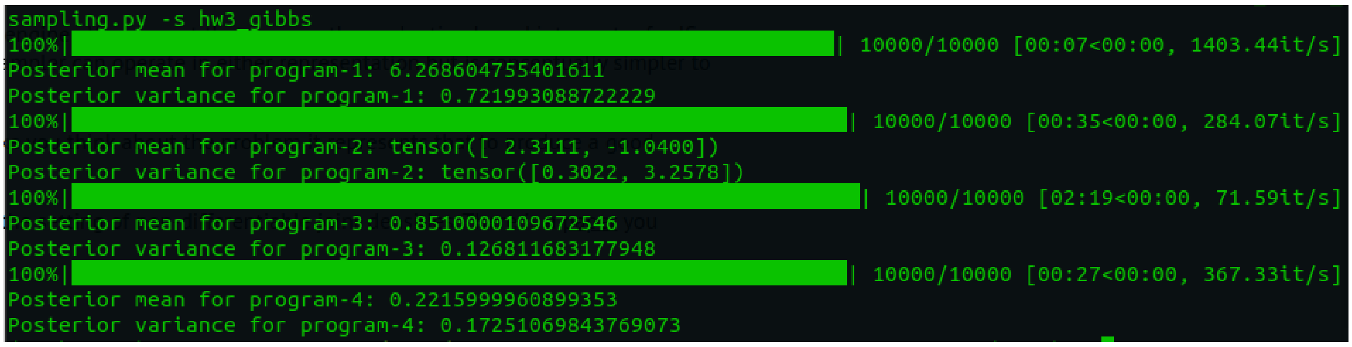
Figure 7: Updating sigma on encountering ”observe”.



```
def hw_3_IS():
    for i in range(1, 5):
        ast = daphne(['desugar', '-i', '../HW3/hw3-programs/{}.daphne'.format(i)])

        samples, weights, n = [], [], 10000
        running_mean = 0
        for j in range(n):
            sample, sigma = evaluate_program(ast)
            samples.append(sample)
            weights.append(torch.exp(sigma['logW']))
            running_mean += sample * torch.exp(sigma['logW']).item()

        samples = torch.stack(samples).float()
        weights = torch.stack(weights)
        mean = running_mean / weights.sum()
        diff = (samples.reshape(n,-1) - mean)**2
        var = torch.matmul(weights.reshape(-1,n), diff) / weights.sum()
        print("Posterior mean for program-{}: {}".format(i, mean))
        print("Posterior variance for program-{}: {}".format(i, var))
```

Figure 8: Computing weighted expectations.

2. MH within Gibbs Sampling



```
sampling.py -s hw3_gibbs
100%|████████████████████████████████████████| 10000/10000 [00:07<00:00, 1403.44it/s]
Posterior mean for program-1: 6.268604755401611
Posterior variance for program-1: 0.721993088722229
100%|████████████████████████████████████████| 10000/10000 [00:35<00:00, 284.07it/s]
Posterior mean for program-2: tensor([ 2.3111, -1.0400])
Posterior variance for program-2: tensor([0.3022, 3.2578])
100%|████████████████████████████████████████| 10000/10000 [02:19<00:00, 71.59it/s]
Posterior mean for program-3: 0.8510000109672546
Posterior variance for program-3: 0.126811683177948
100%|████████████████████████████████████████| 10000/10000 [00:27<00:00, 367.33it/s]
Posterior mean for program-4: 0.2215999960899353
Posterior variance for program-4: 0.17251069843769073
```

Figure 9: Posterior means and variances for MH within Gibbs.

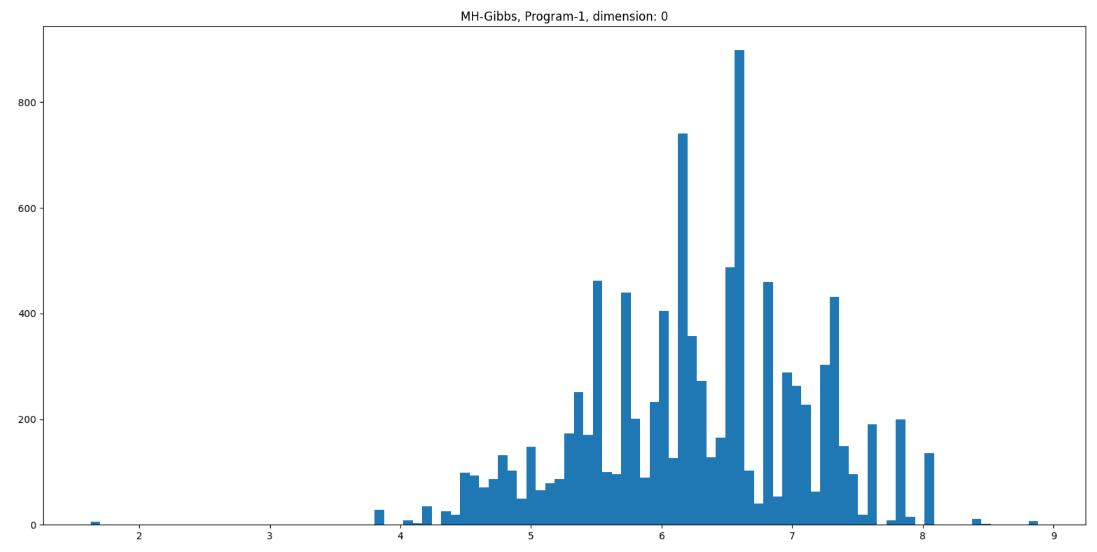


Figure 10: Histogram for samples from Program-1.

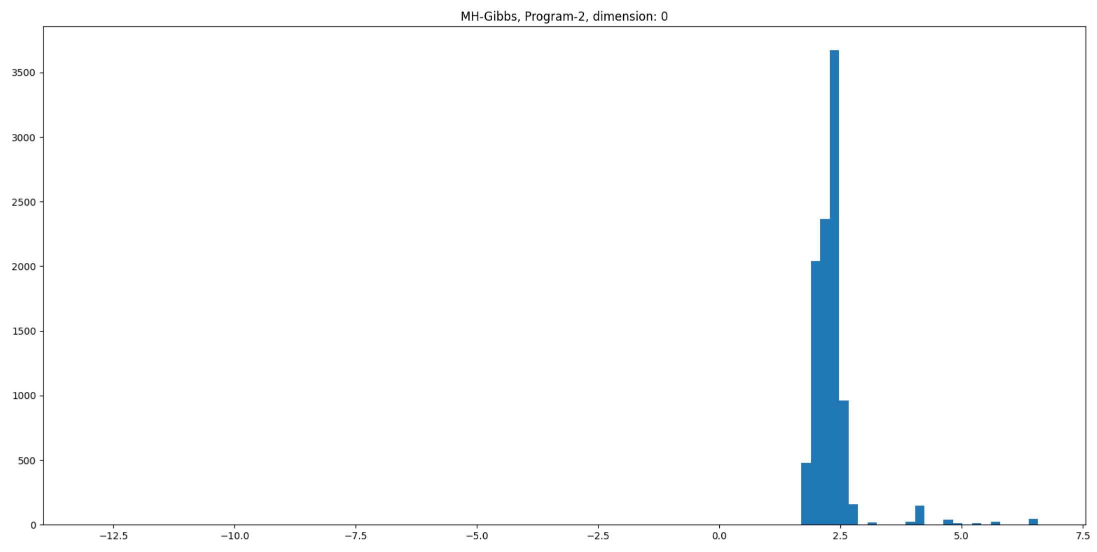


Figure 11: Histogram for slope samples from Program-2.

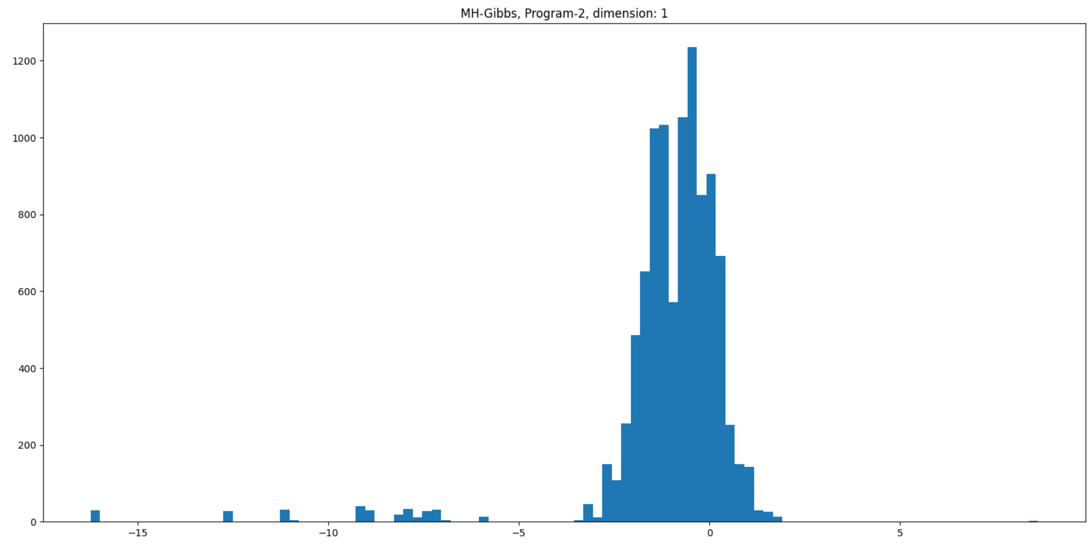


Figure 12: Histogram for bias samples from Program-2.

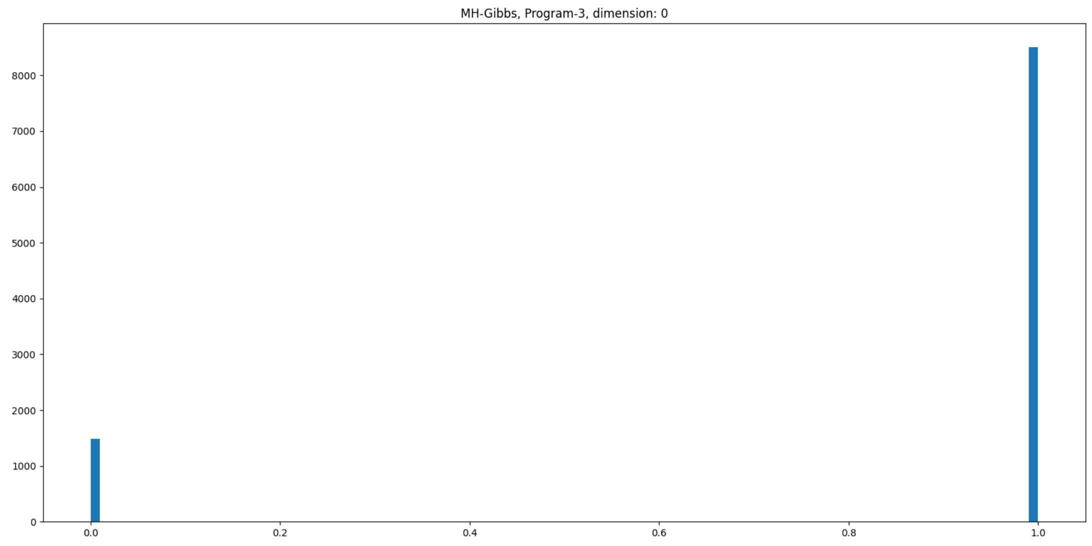


Figure 13: Histogram for samples from Program-3.

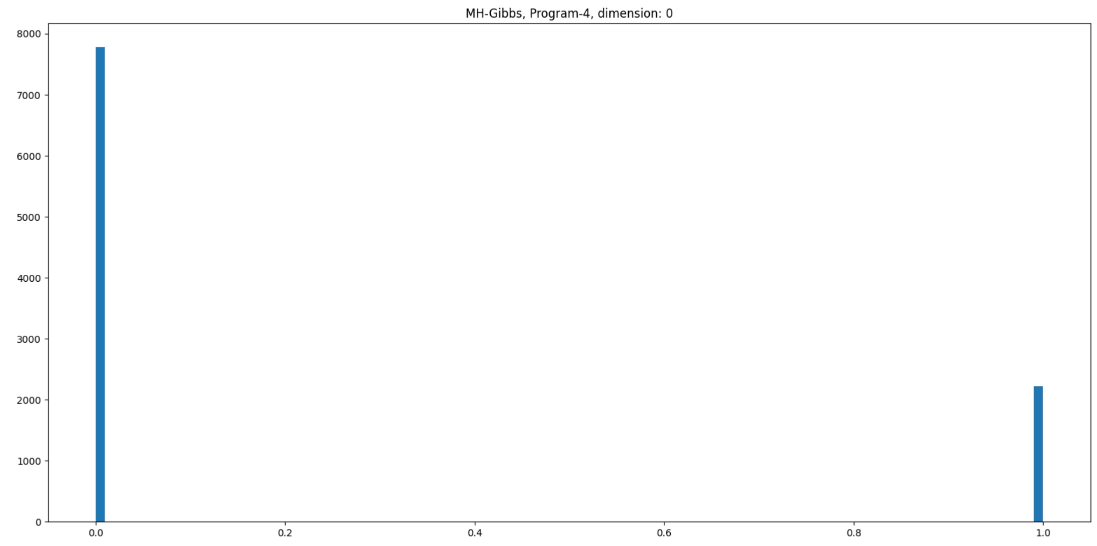
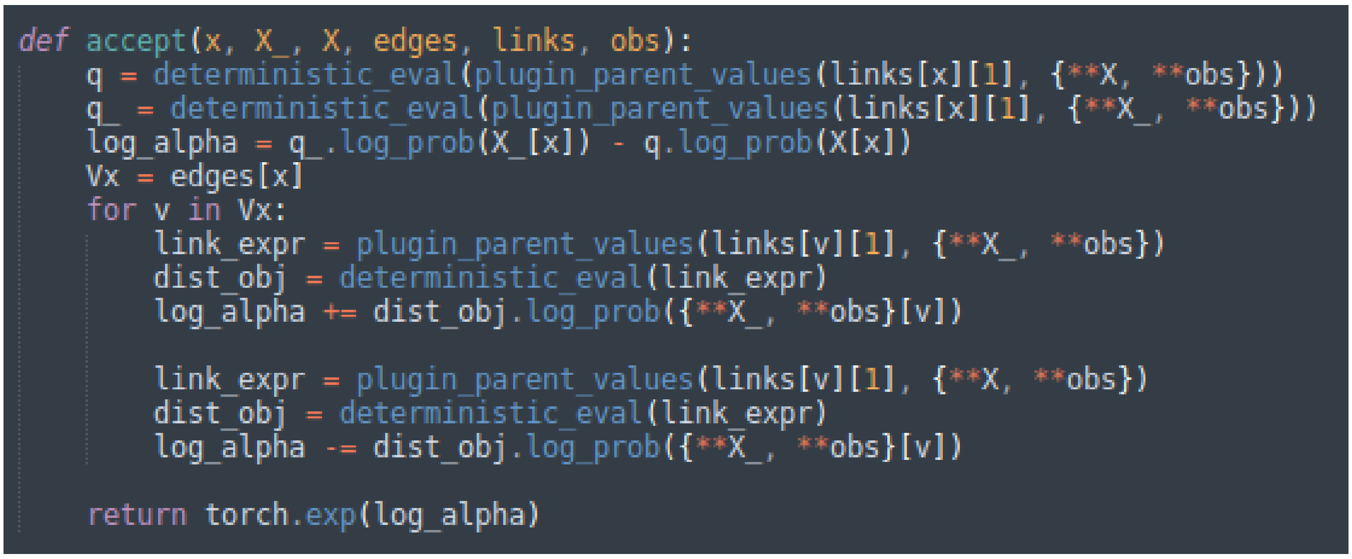
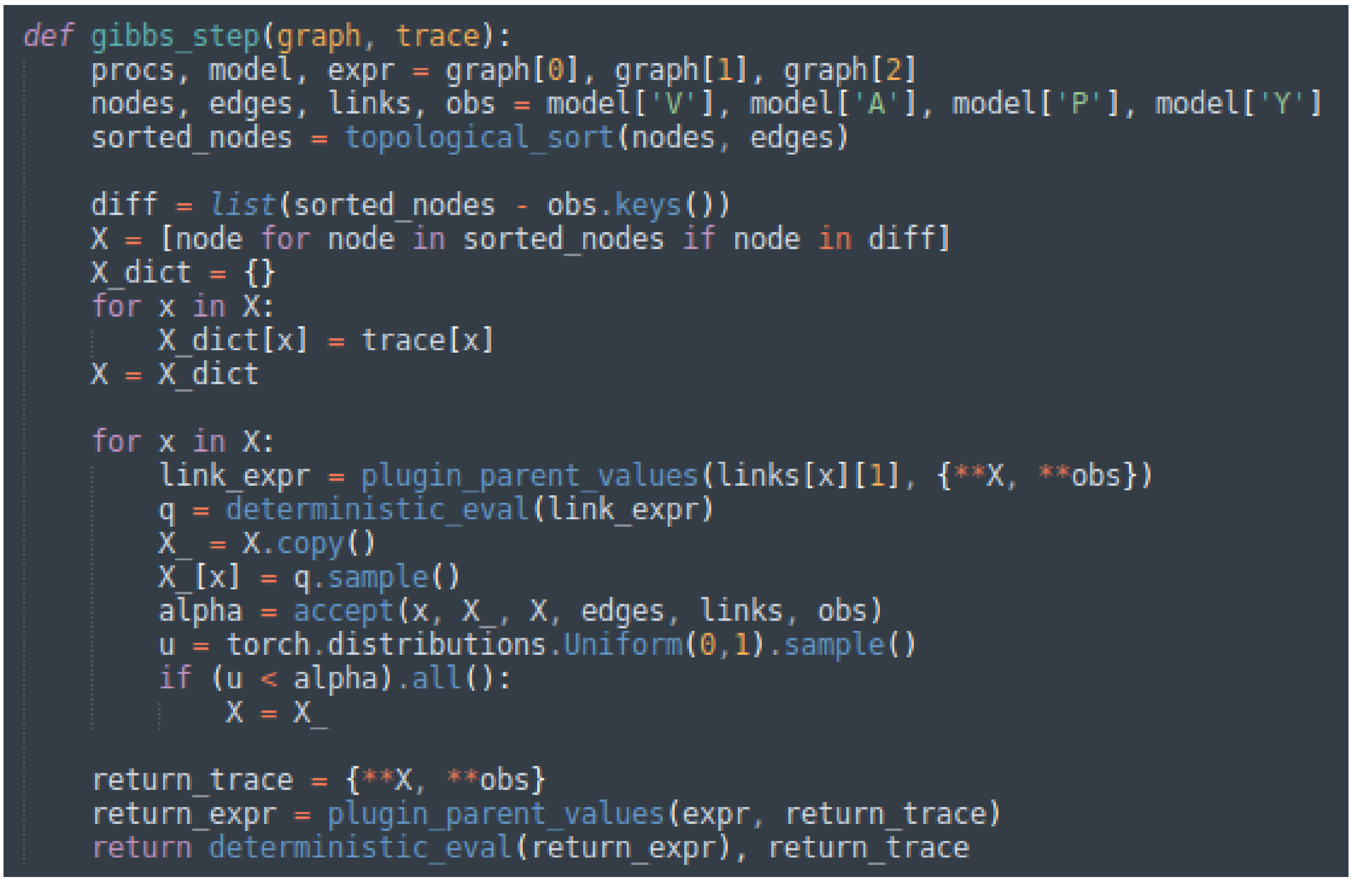


Figure 14: Histogram for samples from Program-4.



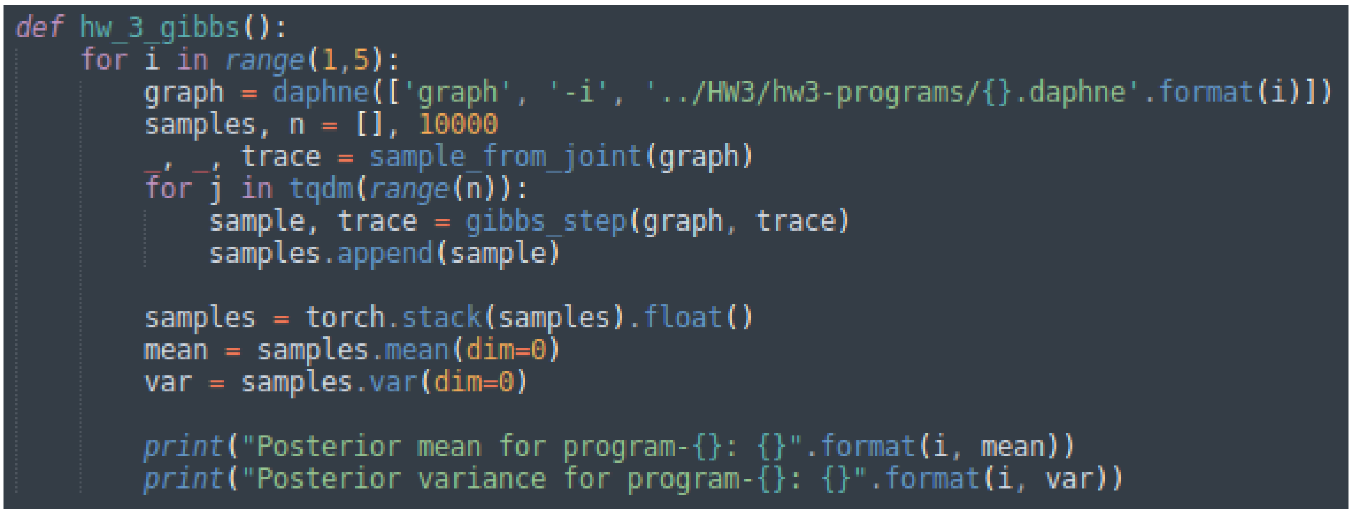
```
def accept(x, X_, X, edges, links, obs):
    q = deterministic_eval(plugin_parent_values(links[x][1], {**X, **obs}))
    q_ = deterministic_eval(plugin_parent_values(links[x][1], {**X_, **obs}))
    log_alpha = q_.log_prob(X_[x]) - q.log_prob(X[x])
    Vx = edges[x]
    for v in Vx:
        link_expr = plugin_parent_values(links[v][1], {**X_, **obs})
        dist_obj = deterministic_eval(link_expr)
        log_alpha += dist_obj.log_prob({**X_, **obs}[v])

        link_expr = plugin_parent_values(links[v][1], {**X, **obs})
        dist_obj = deterministic_eval(link_expr)
        log_alpha -= dist_obj.log_prob({**X_, **obs}[v])

    return torch.exp(log_alpha)
```

Figure 15: Accept function for MH within Gibbs



```python
def gibbs_step(graph, trace):
    procs, model, expr = graph[0], graph[1], graph[2]
    nodes, edges, links, obs = model['V'], model['A'], model['P'], model['Y']
    sorted_nodes = topological_sort(nodes, edges)

    diff = list(sorted_nodes - obs.keys())
    X = [node for node in sorted_nodes if node in diff]
    X_dict = {}
    for x in X:
        X_dict[x] = trace[x]
    X = X_dict

    for x in X:
        link_expr = plugin_parent_values(links[x][1], {**X, **obs})
        q = deterministic_eval(link_expr)
        X_ = X.copy()
        X_[x] = q.sample()
        alpha = accept(x, X_, X, edges, links, obs)
        u = torch.distributions.Uniform(0,1).sample()
        if (u < alpha).all():
            X = X_

    return_trace = {**X, **obs}
    return_expr = plugin_parent_values(expr, return_trace)
    return deterministic_eval(return_expr), return_trace
```

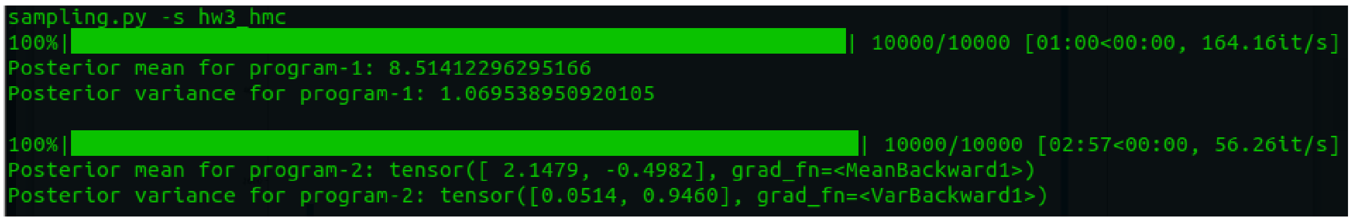
Figure 16: Step function for MH within Gibbs



```python
def hw_3_gibbs():
    for i in range(1,5):
        graph = daphne(['graph', '-i', '../HW3/hw3-programs/{}.daphne'.format(i)])
        samples, n = [], 10000
        _, _, trace = sample_from_joint(graph)
        for j in tqdm(range(n)):
            sample, trace = gibbs_step(graph, trace)
            samples.append(sample)

        samples = torch.stack(samples).float()
        mean = samples.mean(dim=0)
        var = samples.var(dim=0)

        print("Posterior mean for program-{}: {}".format(i, mean))
        print("Posterior variance for program-{}: {}".format(i, var))
```

Figure 17: Gibbs Sampler

3. Hamiltonian Monte Carlo Sampling



```
sampling.py -s hw3_hmc
100%|████████████████████████████████████████| 10000/10000 [01:00<00:00, 164.16it/s]
Posterior mean for program-1: 8.51412296295166
Posterior variance for program-1: 1.069538950920105

100%|████████████████████████████████████████| 10000/10000 [02:57<00:00, 56.26it/s]
Posterior mean for program-2: tensor([ 2.1479, -0.4982], grad_fn=<MeanBackward1>)
Posterior variance for program-2: tensor([0.0514, 0.9460], grad_fn=<VarBackward1>)
```

Figure 18: Posterior means and variances for HMC sampling.

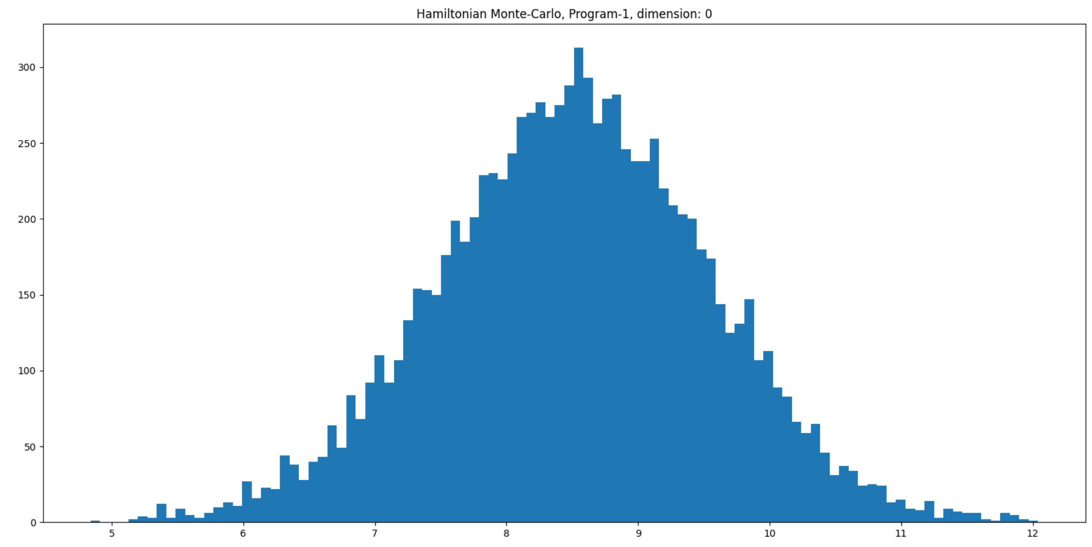


Figure 19: Histogram for samples from Program-1.

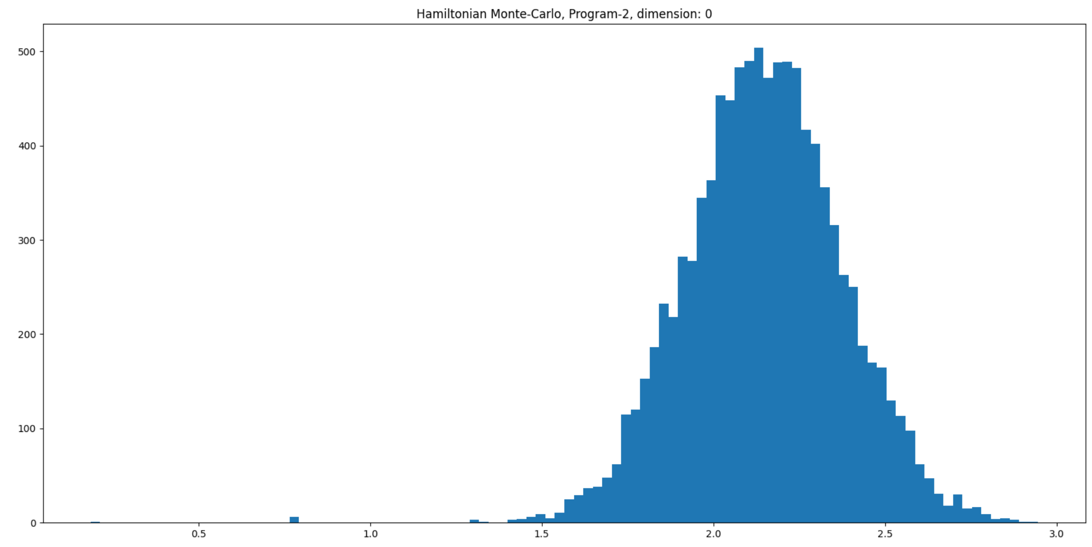


Figure 20: Histogram for slope samples from Program-2.

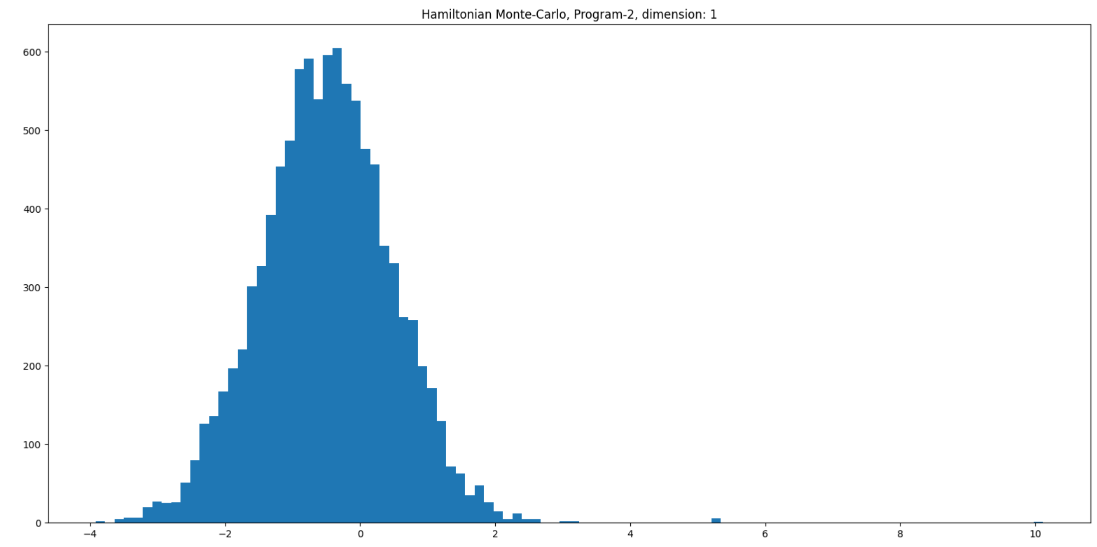
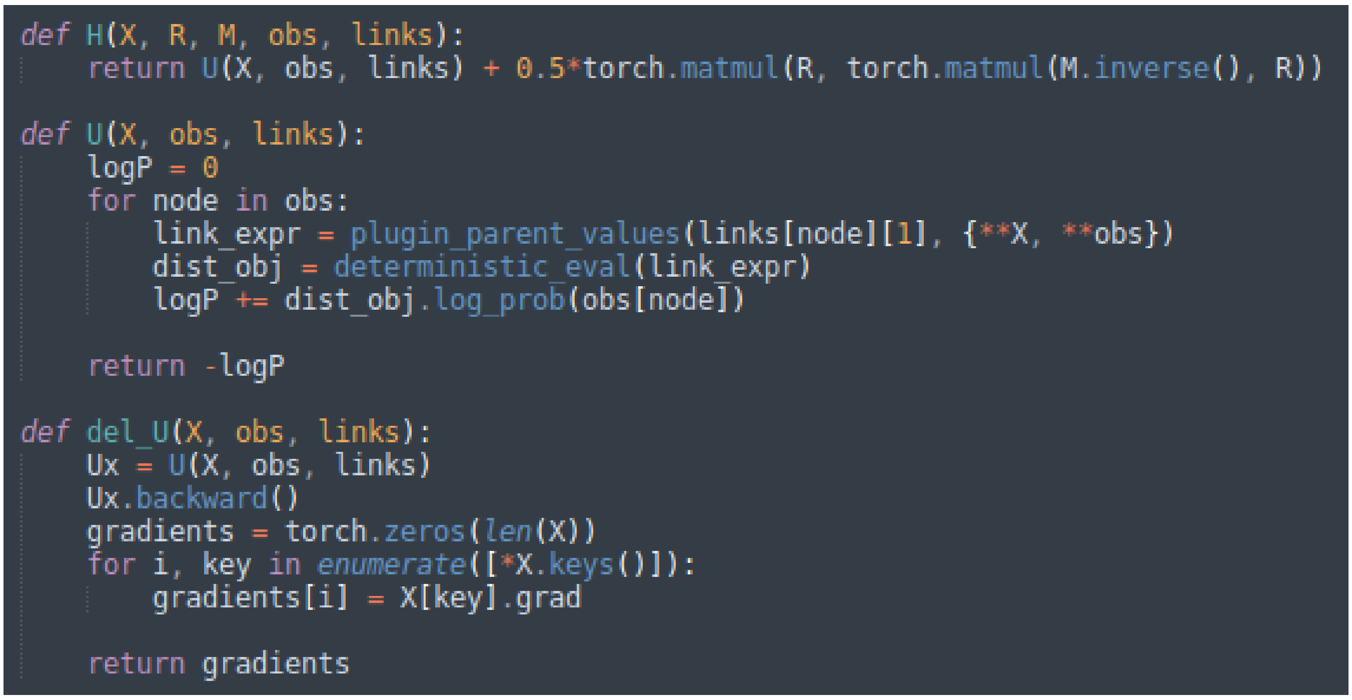
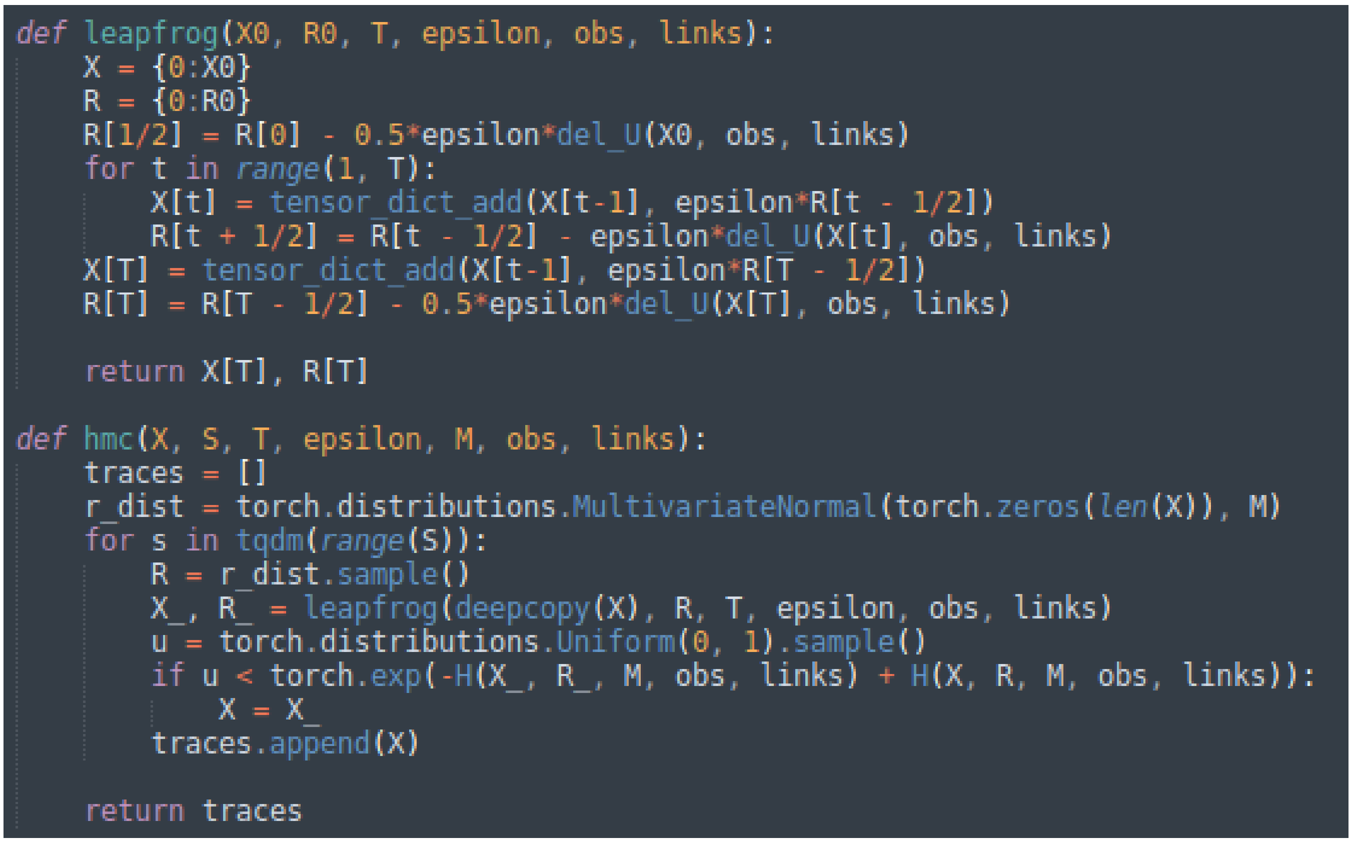


Figure 21: Histogram for bias samples from Program-2.



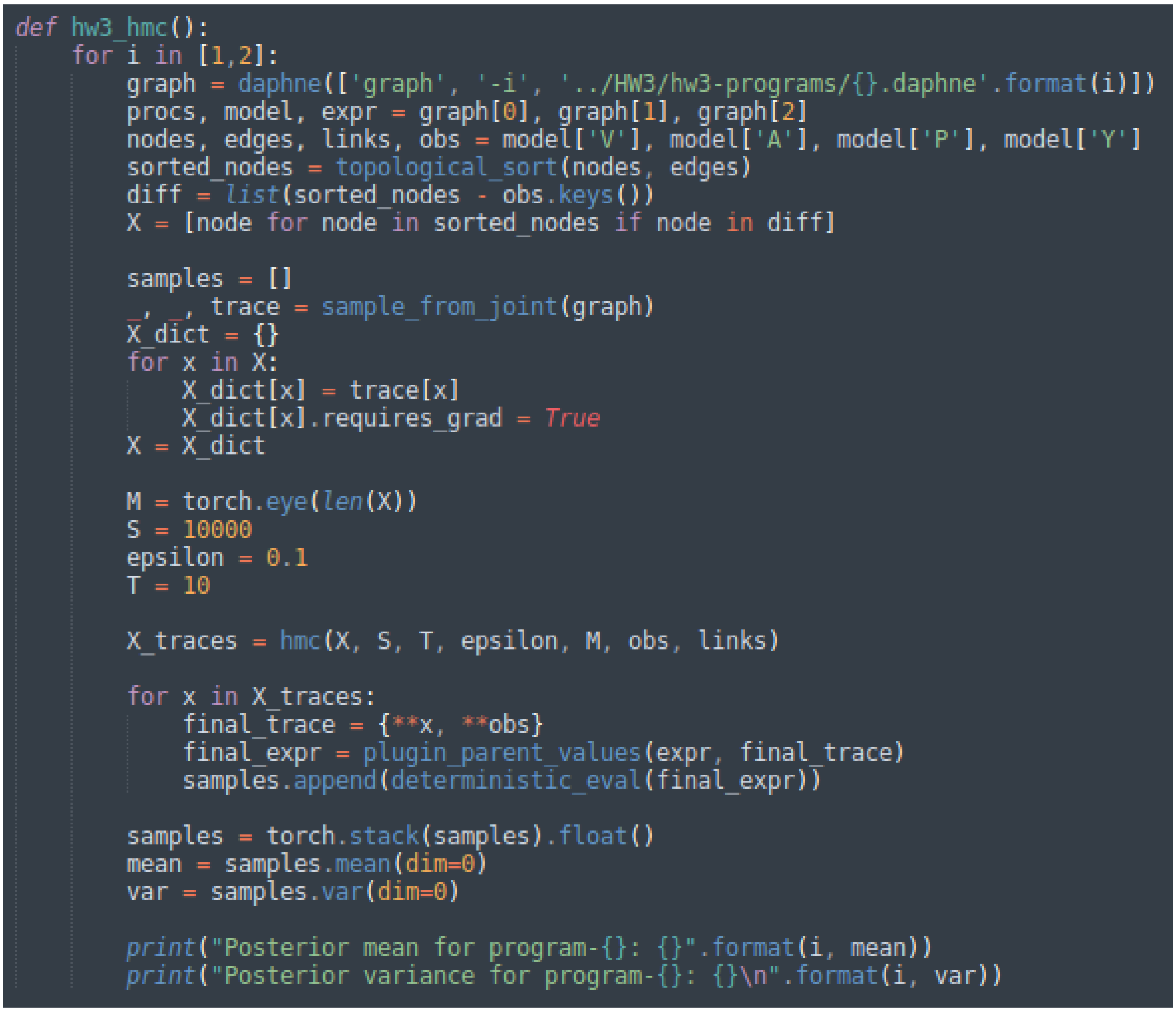
```
def H(X, R, M, obs, links):
    return U(X, obs, links) + 0.5*torch.matmul(R, torch.matmul(M.inverse(), R))

def U(X, obs, links):
    logP = 0
    for node in obs:
        link_expr = plugin_parent_values(links[node][1], {**X, **obs})
        dist_obj = deterministic_eval(link_expr)
        logP += dist_obj.log_prob(obs[node])

    return -logP

def del_U(X, obs, links):
    Ux = U(X, obs, links)
    Ux.backward()
    gradients = torch.zeros(len(X))
    for i, key in enumerate([*X.keys()]):
        gradients[i] = X[key].grad

    return gradients
```

Figure 22: H, U and ∆U functions for HMC.



```python
def leapfrog(X0, R0, T, epsilon, obs, links):
    X = {0:X0}
    R = {0:R0}
    R[1/2] = R[0] - 0.5*epsilon*del_U(X0, obs, links)
    for t in range(1, T):
        X[t] = tensor_dict_add(X[t-1], epsilon*R[t - 1/2])
        R[t + 1/2] = R[t - 1/2] - epsilon*del_U(X[t], obs, links)
    X[T] = tensor_dict_add(X[t-1], epsilon*R[T - 1/2])
    R[T] = R[T - 1/2] - 0.5*epsilon*del_U(X[T], obs, links)

    return X[T], R[T]

def hmc(X, S, T, epsilon, M, obs, links):
    traces = []
    r_dist = torch.distributions.MultivariateNormal(torch.zeros(len(X)), M)
    for s in tqdm(range(S)):
        R = r_dist.sample()
        X_, R_ = leapfrog(deepcopy(X), R, T, epsilon, obs, links)
        u = torch.distributions.Uniform(0, 1).sample()
        if u < torch.exp(-H(X_, R_, M, obs, links) + H(X, R, M, obs, links)):
            X = X_
        traces.append(X)

    return traces
```

Figure 23: Leapfrog function for HMC.



```
def hw3_hmc():
    for i in [1,2]:
        graph = daphne(['graph', '-i', '../HW3/hw3-programs/{}.daphne'.format(i)])
        procs, model, expr = graph[0], graph[1], graph[2]
        nodes, edges, links, obs = model['V'], model['A'], model['P'], model['Y']
        sorted_nodes = topological_sort(nodes, edges)
        diff = list(sorted_nodes - obs.keys())
        X = [node for node in sorted_nodes if node in diff]

        samples = []
        _, _, trace = sample_from_joint(graph)
        X_dict = {}
        for x in X:
            X_dict[x] = trace[x]
            X_dict[x].requires_grad = True
        X = X_dict

        M = torch.eye(len(X))
        S = 10000
        epsilon = 0.1
        T = 10

        X_traces = hmc(X, S, T, epsilon, M, obs, links)

        for x in X_traces:
            final_trace = {**x, **obs}
            final_expr = plugin_parent_values(expr, final_trace)
            samples.append(deterministic_eval(final_expr))

        samples = torch.stack(samples).float()
        mean = samples.mean(dim=0)
        var = samples.var(dim=0)

        print("Posterior mean for program-{}: {}".format(i, mean))
        print("Posterior variance for program-{}: {}\n".format(i, var))
```

Figure 24: Hamiltonian Monte-Carlo Sampler.