HW3

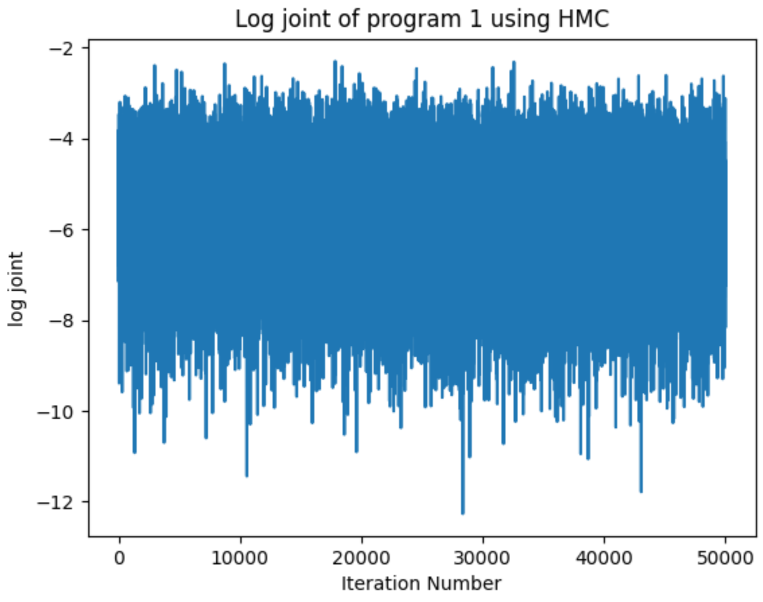
Justice Sefas

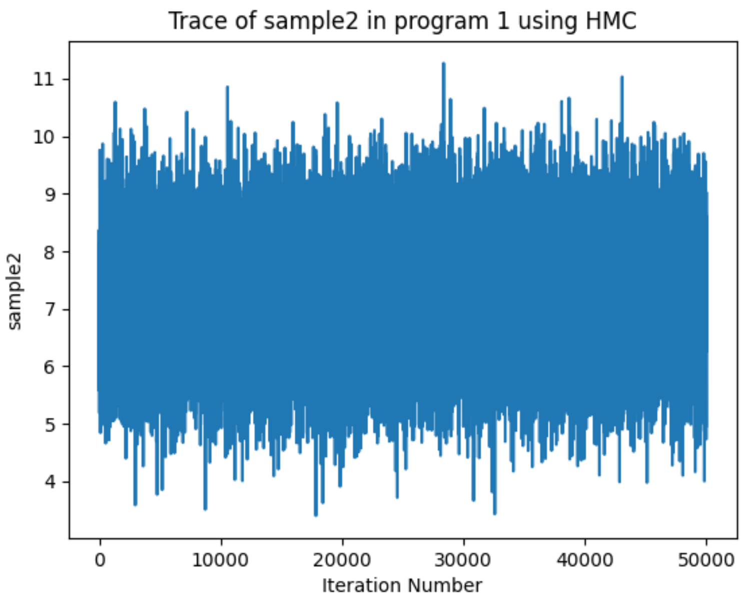
October 28, 2021

Program 1

HMC

Plots





Run time (s)

350.9575340747833

Mean

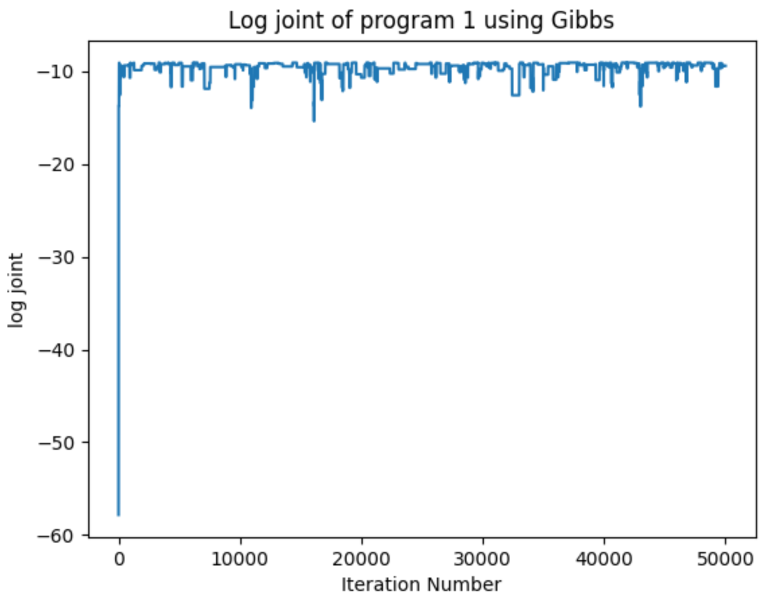
7.23814616

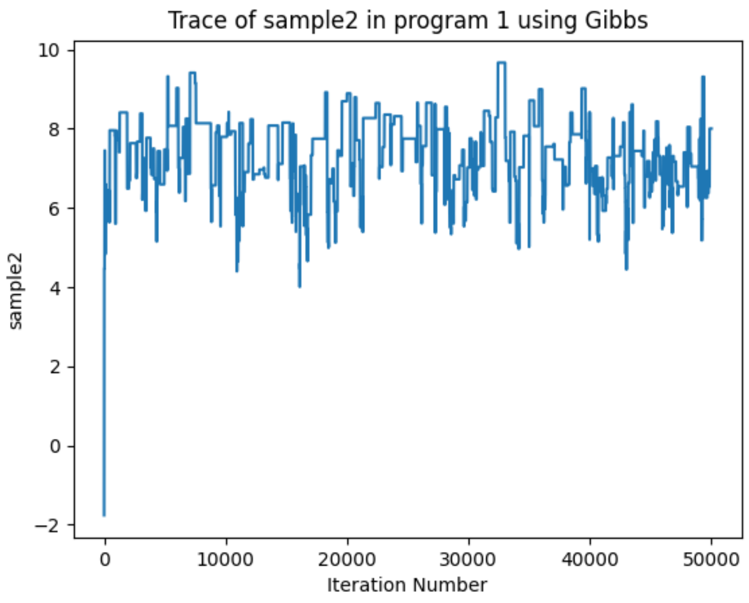
Variance

0.82865091

Gibbs

Plots


Log joint of program 1 using Gibbs
log joint
−10
−20
−30
−40
−50
−60
0
10000
20000
30000
40000
50000
Iteration Number


Trace of sample2 in program 1 using Gibbs
sample2
10
8
6
4
2
0
−2
0
10000
20000
30000
40000
50000
Iteration Number

Run time (s)

45.406129598617554

Mean

7.17261325

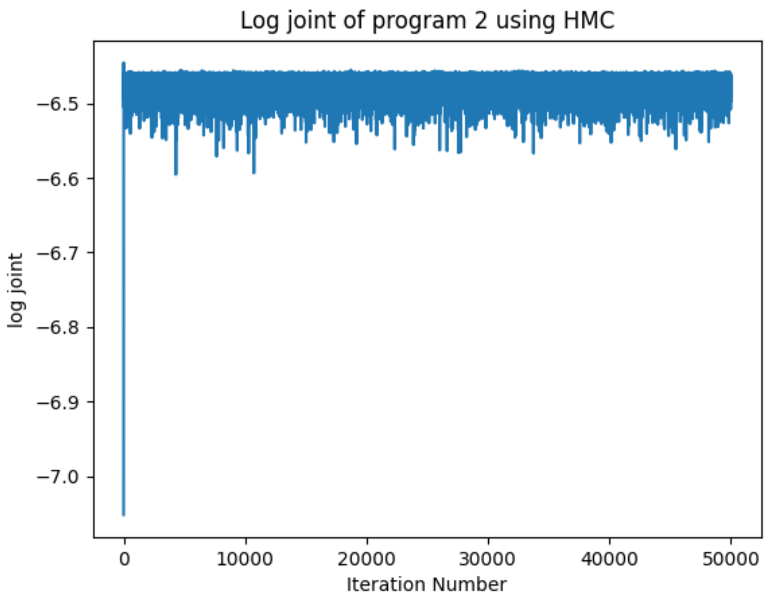
Variance

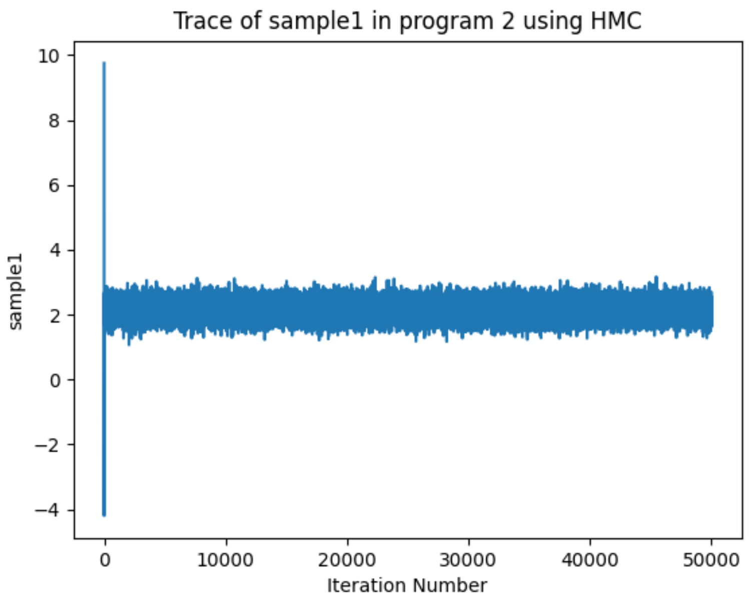
8.32507872e-01

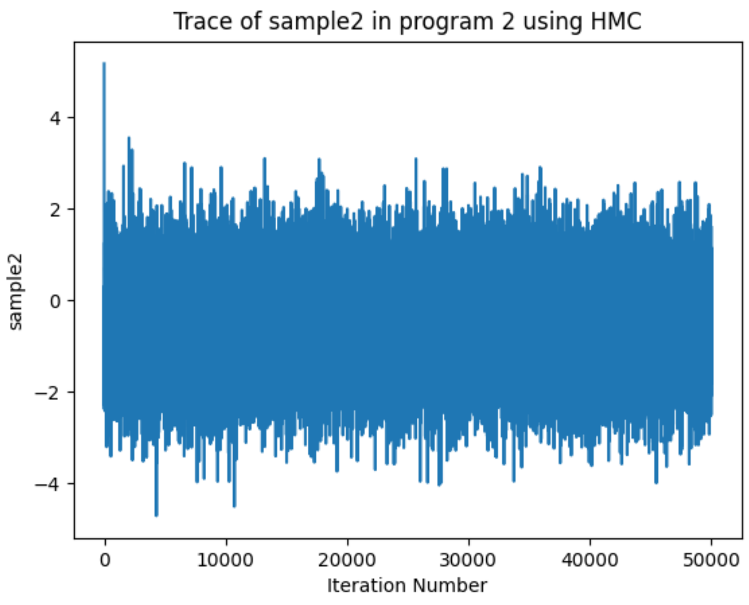
Importance Sampling

Program 2

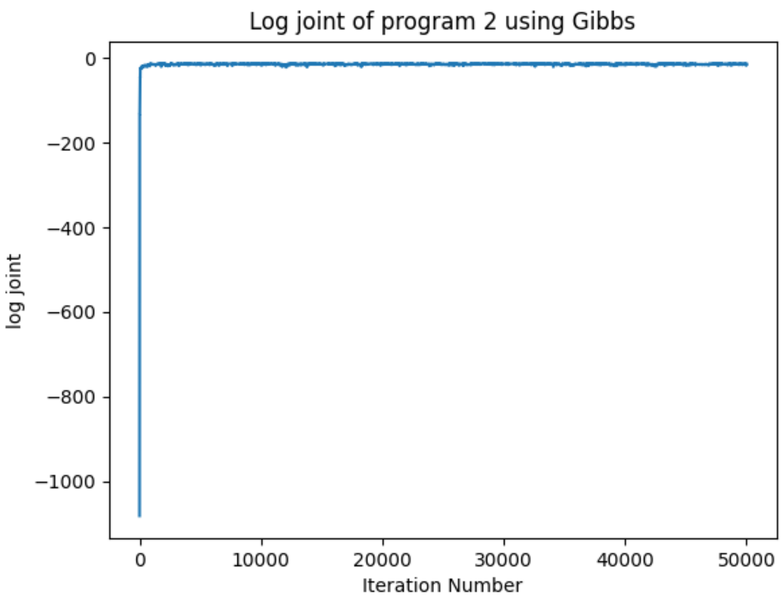
HMC



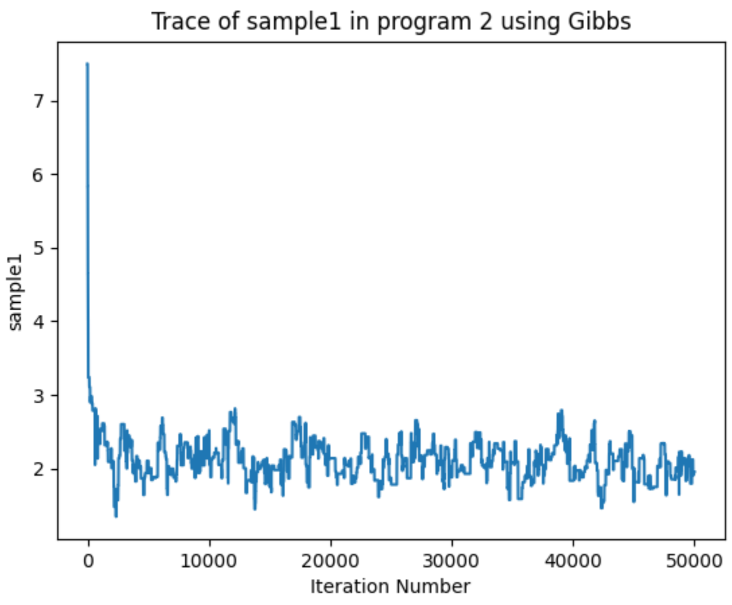

Trace of sample1 in program 2 using HMC
sample1
Iteration Number
10
8
6
4
2
0
−2
−4
0
10000
20000
30000
40000
50000


Trace of sample2 in program 2 using HMC
sample2
Iteration Number
4
2
0
−2
−4
0
10000
20000
30000
40000
50000

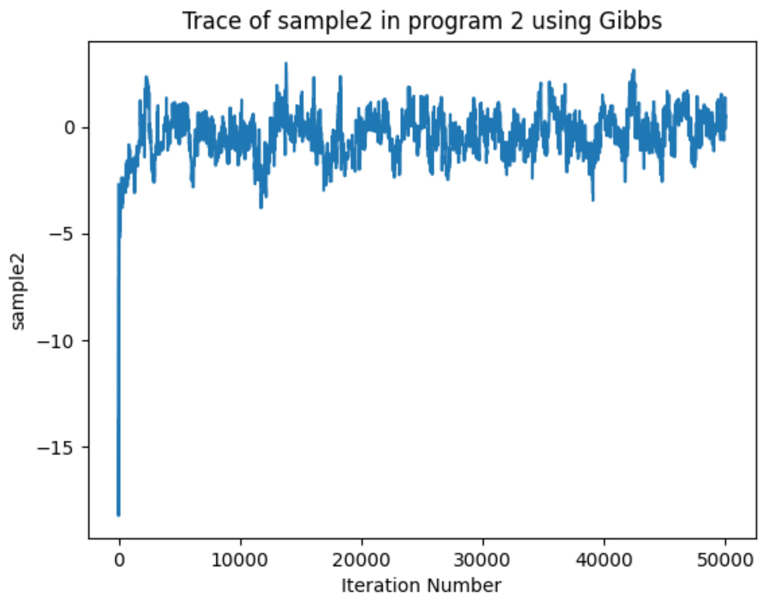
Gibbs



Log joint of program 2 using Gibbs
log joint
0
−200
−400
−600
−800
−1000
0
10000
20000
30000
40000
50000
Iteration Number



Trace of sample1 in program 2 using Gibbs
sample1
7
6
5
4
3
2
0
10000
20000
30000
40000
50000
Iteration Number



Run time (s)

186.18388175964355

Mean

$$\begin{bmatrix} 2.11975777 & -0.41269691 \end{bmatrix}$$

Importance Sampling

Run time (s)

55.23421025276184

Mean

$$\begin{bmatrix} 2.1270 & -0.4365 \end{bmatrix}$$

Covariance

$$\begin{bmatrix} 0.05088208 & -0.18123494 \\ -0.18123494 & 0.82379968 \end{bmatrix}$$

Program 3

Gibbs

Importance

Run time (s)

73.02027440071106

Probability

0.8041344881057739

Variance

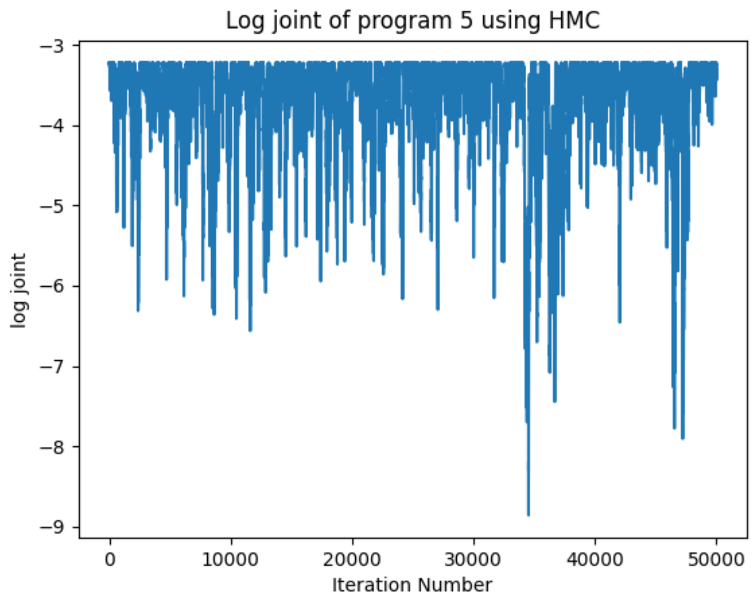
0.15750189558993366

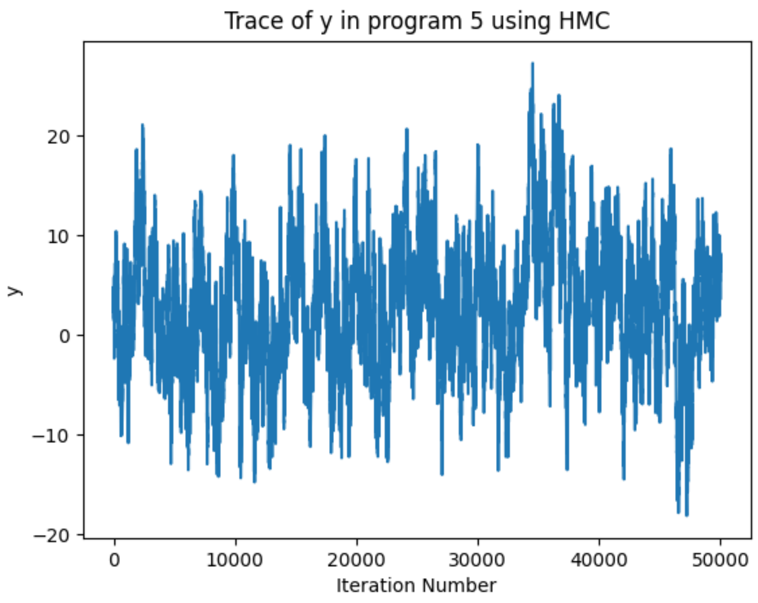
Program 4

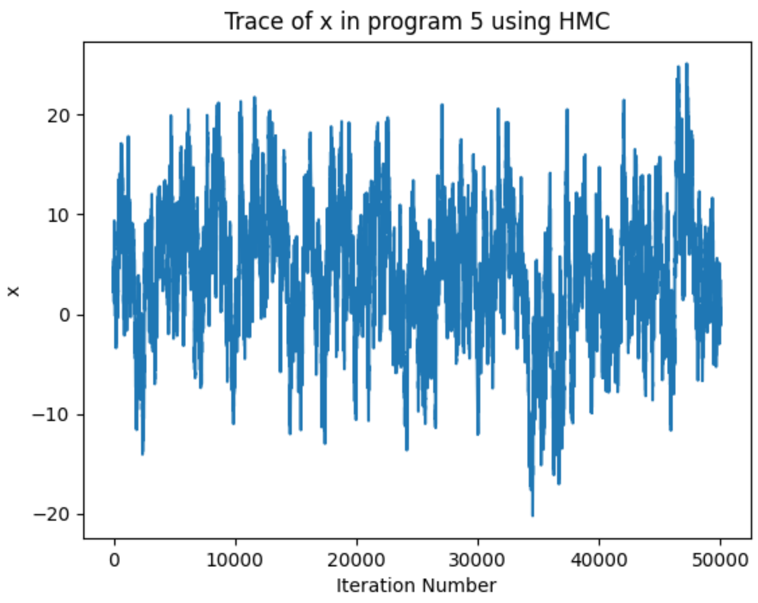
Gibbs

Program 5

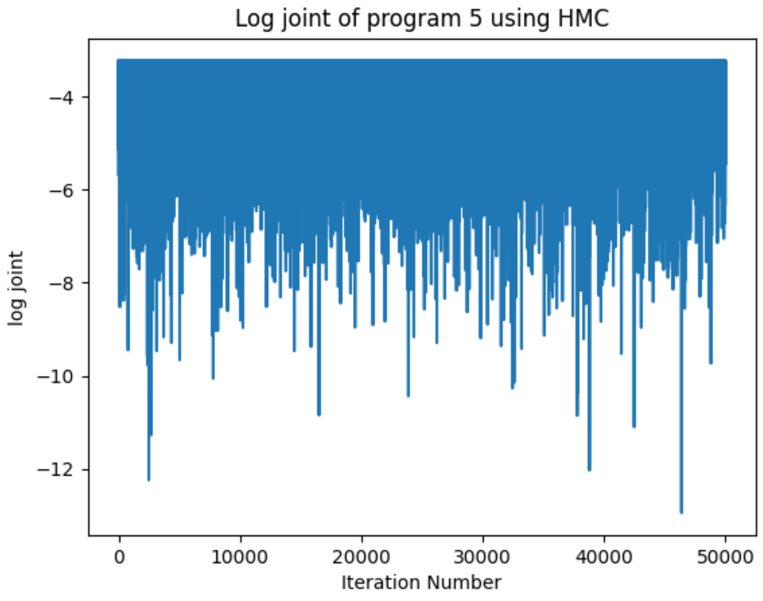
HMC



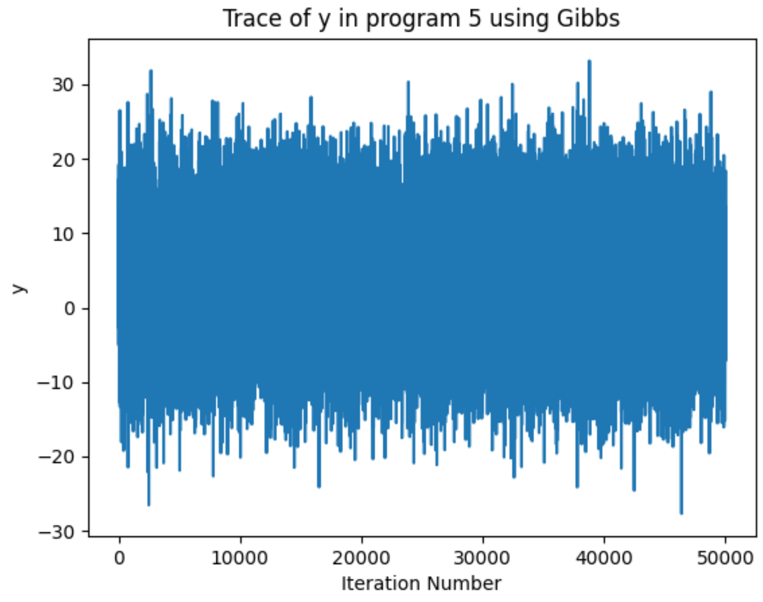

Trace of y in program 5 using HMC
y
Iteration Number
20
10
0
−10
−20
0
10000
20000
30000
40000
50000


Trace of x in program 5 using HMC
x
Iteration Number
20
10
0
−10
−20
0
10000
20000
30000
40000
50000

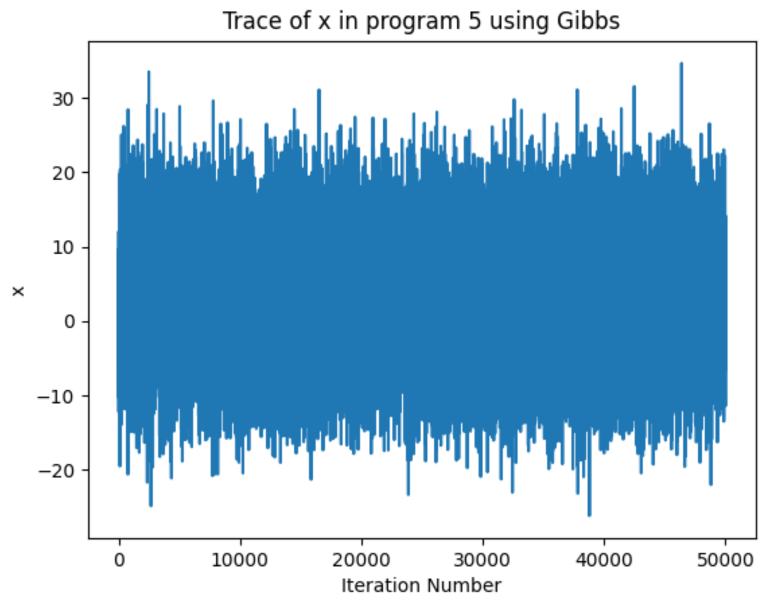
Gibbs



Log joint of program 5 using HMC
log joint
−4
−6
−8
−10
−12
0
10000
20000
30000
40000
50000
Iteration Number



Trace of y in program 5 using Gibbs
y
30
20
10
0
−10
−20
−30
0
10000
20000
30000
40000
50000
Iteration Number



Program 5

/home/jsefas/probprog/cpsc532w-hw2/trace-x-5-Importance.png n order to solve this problem exactly, we recognize that we must sample from the line $y = 7 - x$. Because X and Y are independent, their joint distribution is a spherical bivariate normal centered at the origin with diagonal variance and no covariance. Therefore any slice of this distribution is also Normal$(0, 10)$, so we can simply sample from this distribution. Once we have sample $z \sim$ N$(0, 10)$, we view it as a sample from a vertical slice $P(Y = z|X = c)$ distance c to the right of the y-axis where c is the length from the origin to the point $(3.5, 3.5)$ on $y = 7 - x$. The sample we get from z defines a point in the line $x = c$ at $[c\ z]$. We then rotate this vector $\pi/4$ back onto the line $y = 7 - x$ in order to recover our samples for x and y. In particular, our equations are $x = \frac{7}{2} -$

Code

Evaluation Based Importance Sampling

```python
# observe expression
if isinstance(ast, list) and 'observe' in ast:
    if 'observe' == ast[0]:
        d, sigma = eval(ast[1], sigma, local_v)
        c, sigma = eval(ast[2], sigma, local_v)
        sigma['logW'] += d.log_prob(c)
        return c, sigma
```

Graph Based Gibbs Sampling and HMC

```python
def sample_initial(graph):
    samples, local_v = sample_from_joint(graph)
    return local_v

def computeU_old(X: torch.tensor, var_names: List[str], Y: dict, P: dict, sigma: dict)
    U = torch.tensor([0.0])
    local_map = {**{k:v for k,v in zip(var_names, X)}, **Y}
    for name, value in {k:v for k,v in zip(var_names, X)}.items():
        U -= eval(P[name][1], sigma, local_map)[0].log_prob(value)
    for name, value in Y.items():
        U -= eval(P[name][1], sigma, local_map)[0].log_prob(value)
    return U

def diffU_old(X: torch.tensor, var_names: List[str], Y: dict, P: dict, sigma: dict):
    U = computeU_old(X, var_names, Y, P, sigma)
    U.backward()

def updateR(R, eps, Xt):
    diffU(X, Y, P, sigma)
    for key in R.keys():
        R[key] = R[key] - (1/2)*eps*Xt[key].grad
        Xt[key].grad.data.zero_()
    return R

def leapfrog_old(X: torch.tensor, var_names: List[str], Y: dict, P: dict, R: torch.tens
    Xt = X
    diffU_old(Xt, var_names, Y, P, sigma)
    R_half = R - (1/2)*eps*Xt.grad
    Xt.grad.data.zero_()

    for t in range(1, T):
        Xt.data = Xt.data + eps*R_half

        diffU_old(Xt, var_names, Y, P, sigma)
        R_half -= eps*Xt.grad
        Xt.grad.data.zero_()

    Xt.data = Xt.data + eps*R_half

    diffU_old(Xt, var_names, Y, P, sigma)
    Rt = R_half - (1/2)*eps*Xt.grad
    Xt.grad.data.zero_()

    return Xt, Rt

def H(X, R, M, var_names, Y, P, sigma):
    return computeU_old(X, var_names, Y, P, sigma) + (1/(2*M))*torch.square(R).sum()

def hmc_sample(graph, S):
    "This function does HMC sampling"
    G = graph[1]
    P = G['P']
    Y = G['Y']
    A = G['A']
    V = G['V']
    sigma = {'logW': 0}

    local_v = sample_initial(graph)

    observeds = Y.keys()
    var_names = [v for v in V if v not in observeds]

    Y = {key: torch.tensor([value], requires_grad=False) for key, value in Y.items()}
    X = torch.tensor([value for key, value in local_v.items() if key in var_names], rec

    return hmc(X, var_names, Y, P, sigma = {'logW':0}, S=S)
```

```python
def hmc(X: torch.tensor, var_names: List, Y: dict, P: dict, sigma: dict,
        T: int = 10, eps: float = 0.1, M: float = 1.0, S: int=10000):
    local_vars = []
    Xs = X
    for s in range(S):
        Rs = dist.MultivariateNormal(torch.zeros(len(Xs)), M*torch.eye(len(Xs))).sample
        Xprime, Rprime = leapfrog_old(Xs, var_names, Y, P, Rs, sigma, T, eps)
        if torch.rand(1) < torch.exp(-H(Xprime, Rprime, M, var_names, Y, P, sigma) + H
            Xs = Xprime
        local_vars.append({var_name: value for var_name, value in zip(var_names, X)})
    return local_vars

def accept(x: str, new_map: dict, old_map: dict, P: dict, A: dict, sigma: dict):
    """ Computes acceptance probability for MH
    arg x: name of newly proposed variable
    arg new_map: map from variable names to sample values with the new proposal value
    arg old_map: map from variable names to sample values with the old proposal value
    return: MH acceptance probability
    """
    # prior distribution
    d, sigma = eval(P[x][1], sigma, old_map)
    # prior distribution (I don't see how this can be different from d)
    d_prime, sigma = eval(P[x][1], sigma, new_map)

    # compute proposal ratio
    # (1) *given* the *new* value of x (from d_prime) calculate the probability of the
    # (2) *given* the *old* value of x (from d) calculate the probability of the *new
    # loga = (1) - (2)
    loga = d_prime.log_prob(old_map[x]) - d.log_prob(new_map[x])

    # get nodes where x is a parent
    vx = A[x] + [x]

    # compute posterior probability
    for v in vx:
        d1, sigma = eval(P[v][1], sigma, new_map)
        log_update_pos = d1.log_prob(new_map[v])

        d2, _ = eval(P[v][1], sigma, old_map)
        log_update_neg = d2.log_prob(old_map[v])

        loga = loga + log_update_pos - log_update_neg
    return np.exp(loga)


def gibbs_step(old_map: dict, unobserveds: List[str], P: dict, A: dict, sigma: dict):
    for x in unobserveds:
        d, sigma = eval(P[x][1], sigma, old_map)
        new_map = old_map.copy()
        new_map[x] = d.sample()
        alpha = accept(x, new_map, old_map, P, A, sigma)
        if torch.rand(1) < alpha:
            old_map = new_map.copy()
    return old_map


def gibbs_sample(graph, S = 100000):
    "This function does MH for each step of Gibbs sampling."
    G = graph[1]
    P = G['P']
    Y = G['Y']
    A = G['A']
    V = G['V'].copy()
    sigma = {'logW': 0}

    local_v = sample_initial(graph)

    observeds = Y.keys()
    unobserveds = [v for v in V if v not in observeds]

    samples: List[dict] = [local_v]
    for s in range(S):
        local_v = gibbs_step(local_v, unobserveds, P, A, sigma)
        samples.append(local_v)

    return samples
```