CPSC532W Homework 3

Justin Reiher
Student ID: 37291151
CWL: reiher

Link to public repository for homework 3:

`https://github.com/justinreiher/probProg_Fall2021/tree/main/CS532-HW3`

In program 5, the Dirac delta function $\delta(x)$ is approximated with a Normal distribution and a $\sigma^2 = 0.1$. Approximating the Dirac function in this way allows for an easy method to score. Taking $sigma^2 \rightarrow 0$ would make for a better approximation, however then the sampling algorithms would need to run forever in order to actually capture that distribution. In program 5 the invariant that we want is for $x + y = 7$ and this can take on any values of x and y.

1 Importance Sampling (IS)

Every run of IS collects 30k samples. The Importance Sampling implements algorithm 4 from the book. The main difference is in computing the side effect of observe:

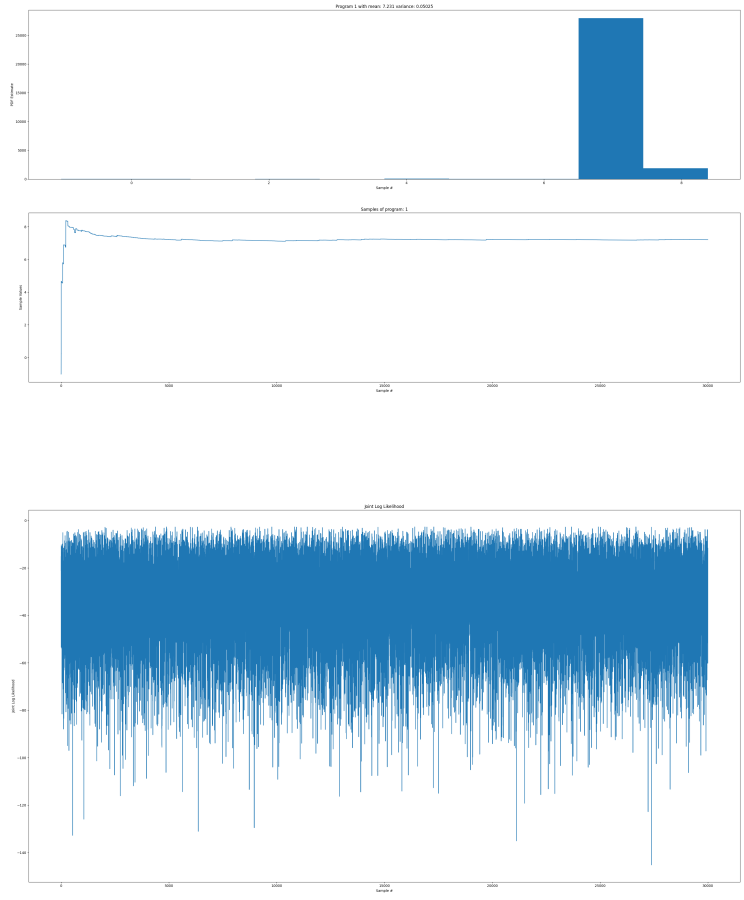
```
        elif e[0] == 'observe':
            d = e[1] # get the distribution declaration
            y = e[2] # here I don't care about y in this assignment, but put here for the future
            try: #first try to get a distribution object from the known distributions directly
                 #otherwise the distribution object may be a variable

                Dobj,sig = evaluate_program(d) #assign the distribution object
                ret,sig = evaluate_program(y)

                if Dobj == [] or y == []:
                    #variable not set, must be in a defn
                    ret = []
                else:
                    dSample = Dobj.sample() #Sample from the prior
                    sig = sig + Dobj.log_prob(ret)

            except:
```

```
                Dobj = var[d] #if retrieved from variable, then
    arguments already set
                ret = Dobj.sample()
                sig = sig + Dobj.log_prob(ret)
```

Additional operations and distributions were added to the environment. The weighted samples are return and plotted.

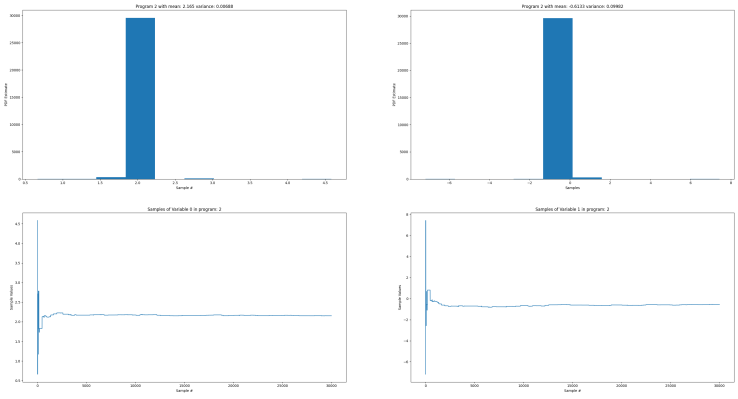
1.1 Program 1

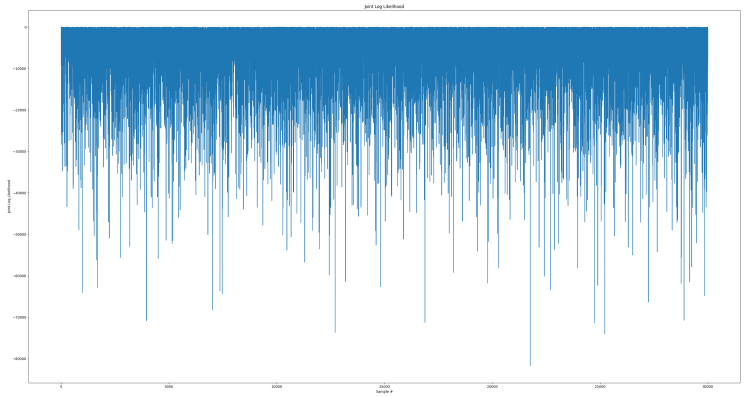
```
Collect samples denoted by program 1:
Elapsed time for program  1 .daphne is:  0:00:11.014697  seconds
tensor(7.2111)
tensor([-1.0193, -1.0217,  0.0308,  ...,  7.2112,  7.2112,  7.2112])
Mean of trace:  tensor(7.2306)
and variance of trace:  tensor(0.0502)
```



1.2 Program 2

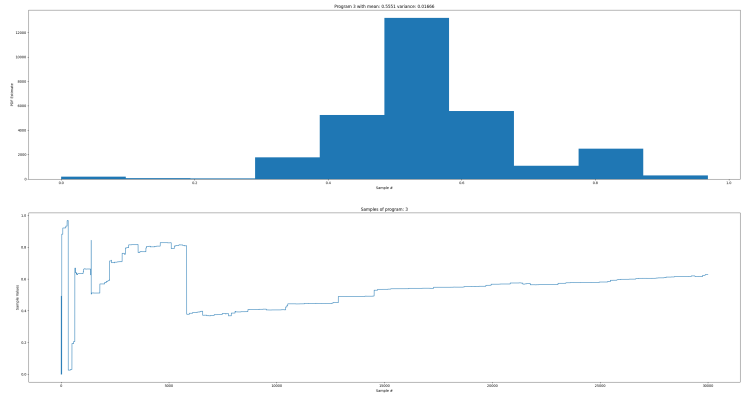
```
Collect samples denoted by program 2:
Elapsed time for program  2 .daphne is:  0:00:50.207221  seconds
tensor([ 2.1535, -0.5521])
tensor([[ 4.5837, -7.1774],
[ 4.5837, -7.1774],
[ 4.5837, -7.1774],
...,
[ 2.1535, -0.5521],
[ 2.1535, -0.5521],
[ 2.1535, -0.5521]])
Mean of trace:  tensor([ 2.1645, -0.6133])
and variance of trace:  tensor([0.0069, 0.0998])
The covariance matrix:
[[ 0.00687998 -0.02219416]
[-0.02219416  0.09982455]]
```

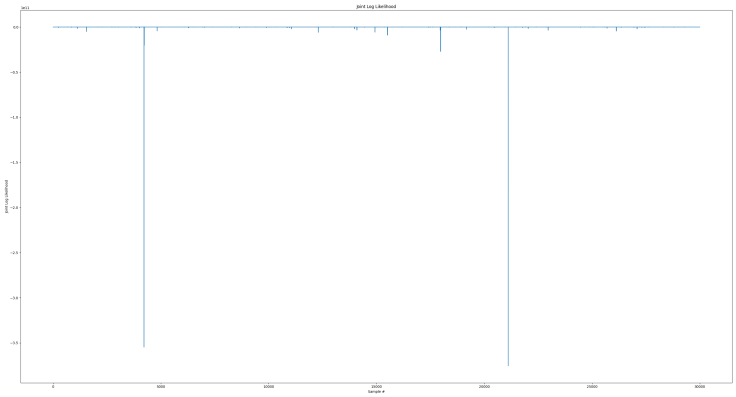




1.3 Program 3

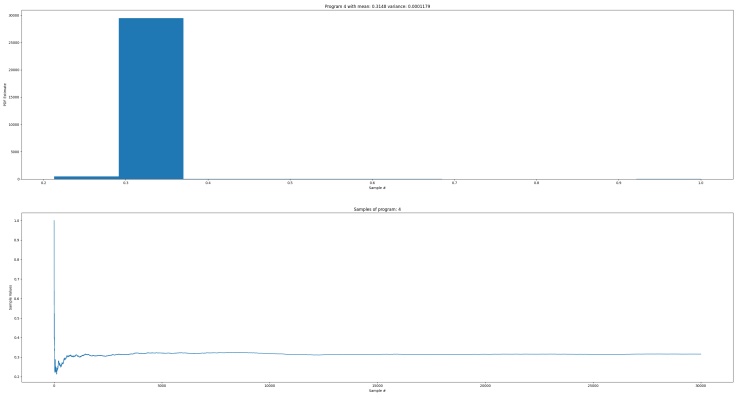
```
Collect samples denoted by program 3:
Elapsed time for program  3 .daphne is:  0:00:59.699769  seconds
tensor(0.6275)
tensor([0.0000, 0.0000, 0.4917,  ..., 0.6275, 0.6275, 0.6275])
Mean of trace:  tensor(0.5551)
and variance of trace:  tensor(0.0167)
```

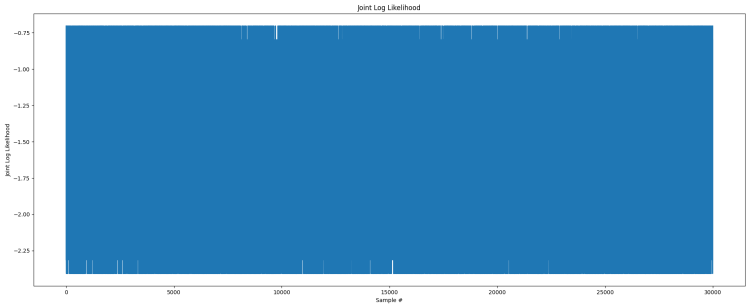




1.4 Program 4

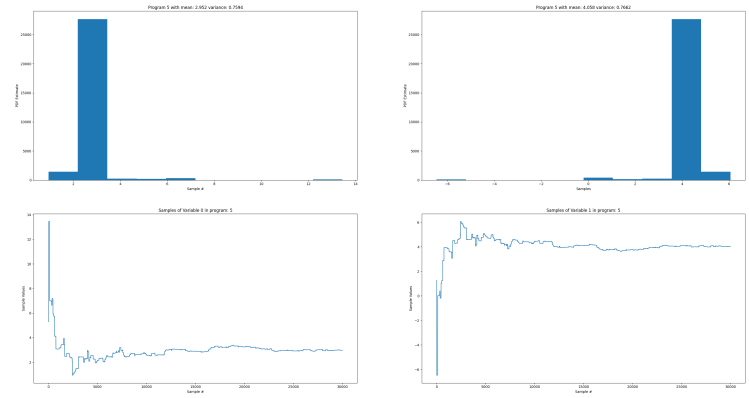
```
Collect samples denoted by program 4:
Elapsed time for program  4 .daphne is:  0:00:18.411620  seconds
tensor(0.3158)
tensor([1.0000, 1.0000, 0.5690,  ..., 0.3158, 0.3158, 0.3158])
Mean of trace:  tensor(0.3148)
and variance of trace:  tensor(0.0001)
```

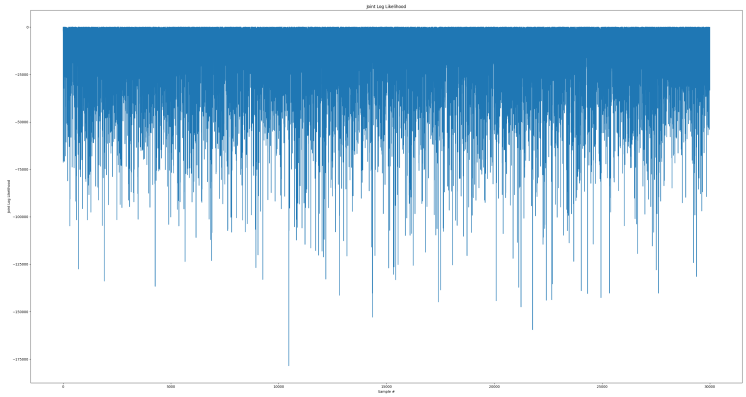




1.5 Program 5

```
Collect samples denoted by program 5:
Elapsed time for program  5 .daphne is:  0:00:11.613676  seconds
tensor([2.9748, 4.0338])
tensor([[5.3001, 1.2459],
[5.3001, 1.2459],
[5.3001, 1.2459],
...,
[2.9748, 4.0338],
[2.9748, 4.0338],
[2.9748, 4.0338]])
Mean of trace:  tensor([2.9516, 4.0578])
and variance of trace:  tensor([0.7594, 0.7662])
```





2 Metropolis-Hastings within Gibbs (MH-Gibbs)

This sampler implements algorithm 1 of the textbook using the graph based evaluator. The first thing that happens like in HW2 is a topological sort of the vertices. From the set of vertices, a mapping to the unobserved variables and their link function is created:

```
    #initialize the child-parent mapping with empty lists:
    for i in vertices:
        P[i] = []
        #get the map Q of unobserved variables
        if i.find('sample') == 0:
          X[i] = i
```

An optimization to speed up computation is to create a mapping of parent nodes from children nodes, i.e. being able to look up in a map what the parents of the current node is:

```
def childParentMapping(vertices,edges,P):

  for i in vertices:
    for j in edges:
      if(i in edges[j]):
        listOfP = P[i]
        #want this to be a set, so no duplication
        if not(j in listOfP):
          listOfP.append(j)
  return P
```

Then an initial initialization for the unobserved variables X_0 is assigned sampling from their prior:

```
    #set X_0, by sampling every node
    for i in X:
        linkFuncsEval = bindingVars(X,linkFuncs[i])
```

Then for the specified number of samples, the following loop is run:

```
    for s in range(num_samples):
        jll.append(0)
        Xret = gibbs_step(X)

        if type(E) == str:
          #there is an assumption that if the return expression E
    is singular
        #then it belongs to some node
          retExp = Xret[E]
        else:
          # find and replace the variables with the results from
    evaluating the link functions
          retExp = bindingVars(Xret,E)
```

at every Gibbs step, the variables are bound to their expressions in the associated link function, the link function is evaluated to generate a proposal assignment, the acceptance ratio for a one step MH is computed and the proposed sample is either accepted or rejected.

```
def gibbs_step(X):
    for x in X:
      linkEval = bindingVars(X,Q[x])
      Dobj = deterministic_eval(linkEval)
      Xp = X.copy()
      Xp[x]  = Dobj.sample()
      alpha = accept(x,Xp,X)
      u  = dist.Uniform(0,1).sample()
      if u < alpha:
        X = Xp.copy()
    return X
```

In order to compute the acceptance ratio, we first need to compute the Markov blanket of the variable in question from the graph:

```
        for x in vertices:
            linkFuncsEval = bindingVars(X,linkFuncs[x])
            if(linkFuncsEval[0] == 'sample*'):
                linkFuncsEval[0] == 'observe*'
                linkFuncsEval.append(X[x].clone().detach())
            jll[s] = jll[s] + deterministic_eval(linkFuncsEval)

    return Xs,jll

def getMarkovBlanket(x):
  nodes = [x]
  parentsX = P[x]
  for i in parentsX:
    if not(i in nodes) and not(i == x):
      nodes.append(i)
  childrenX = edges[x]
  for i in childrenX:
    if not(i in nodes) and not(i == x):
      nodes.append(i)
  for i in childrenX:
    parentChild = P[i]
```

```
    for j in parentChild:
      if not(j in nodes) and not(j == x):
        nodes.append(j)
```

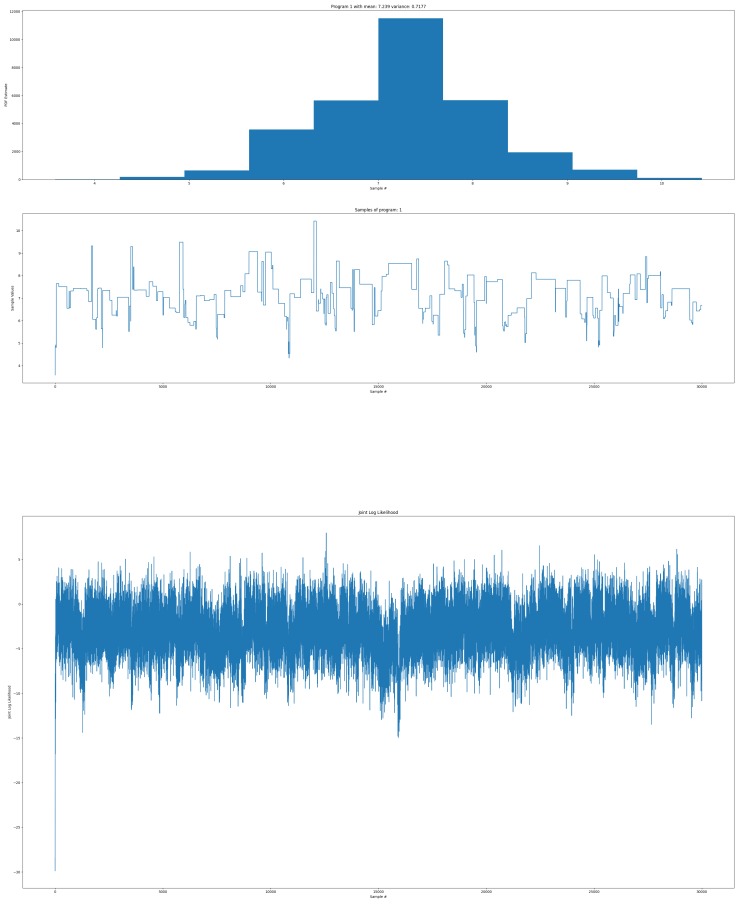
The variables returned from the Markov blanket are scored and the acceptance ratio is returned:

```
def accept(x,Xp,X):
    linkEvalX = bindingVars(X,Q[x])
    DobjX = deterministic_eval(linkEvalX)
    linkEvalXp = bindingVars(Xp,Q[x])
    DobjXp = deterministic_eval(linkEvalXp)

    logAlpha = DobjXp.log_prob(X[x]) - DobjX.log_prob(Xp[x])
    markovBlanket = getMarkovBlanket(x)

    for v in markovBlanket:
      linkEvalX = bindingVars(Xp,linkFuncs[v])
      linkEvalXp = bindingVars(X,linkFuncs[v])

      if linkEvalX[0] == 'sample*':
        linkEvalX[0] = 'observe*'
        linkEvalX.append(Xp[v])
      if linkEvalXp[0] == 'sample*':
        linkEvalXp[0] = 'observe*'
        linkEvalXp.append(X[v])

      logAlpha = logAlpha + deterministic_eval(linkEvalX)
      logAlpha = logAlpha - deterministic_eval(linkEvalXp)

    return torch.exp(logAlpha)
```

Each program is run with 30k samples, except for program 3 which runs much much slower and thus only has 8k samples.

2.1 Program 1

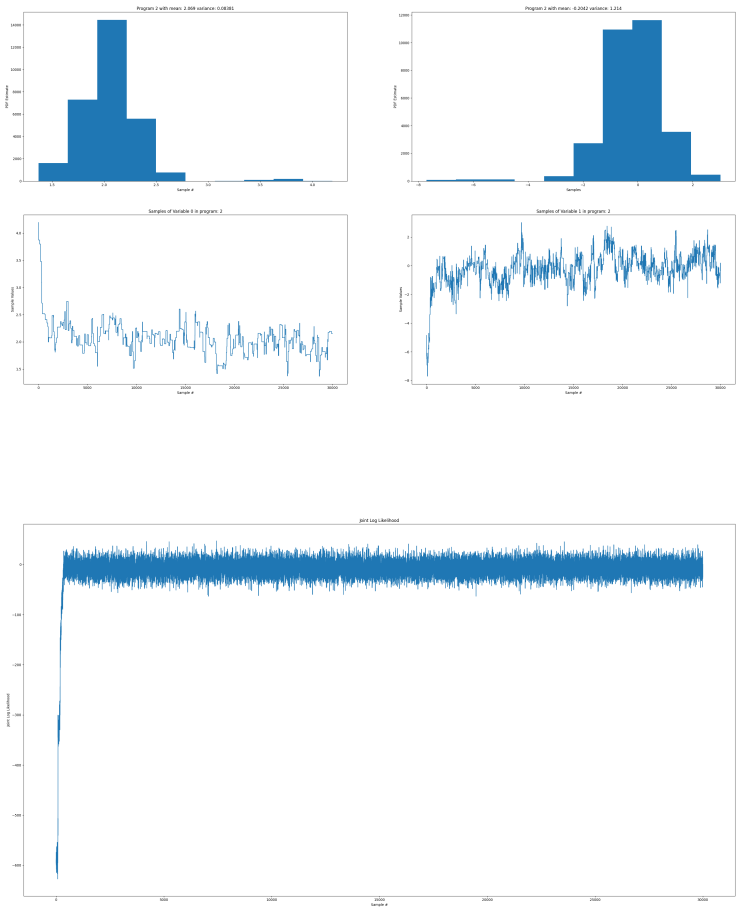
```
Collect samples denoted by program 1:
Elapsed time for program  1 .daphne is:  0:00:35.695762  seconds
tensor([3.5825, 3.5825, 3.5825,  ..., 6.6816, 6.6816, 6.6816])
Mean of trace:  tensor(7.2390)
and variance of trace:  tensor(0.7177)
```



2.2 Program 2

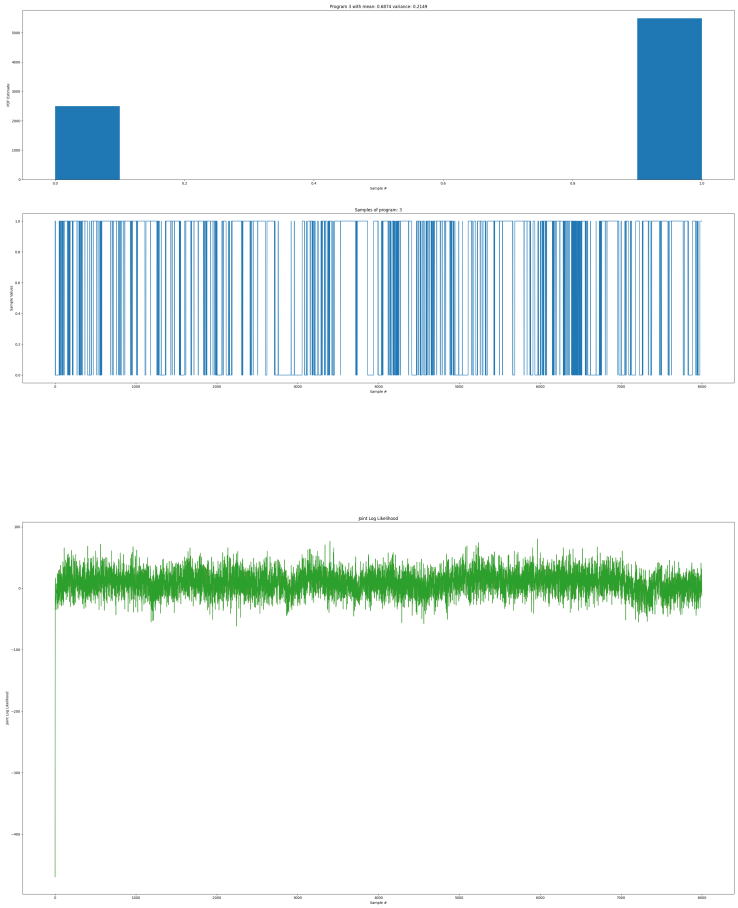
```
Collect samples denoted by program 2:
Elapsed time for program  2 .daphne is:  0:02:54.739333  seconds
tensor([[ 4.1943, -4.8585],
[ 4.1943, -4.8585],
[ 4.1943, -4.8585],
...,
[ 2.1528, -0.3358],
[ 2.1528, -0.3358],
[ 2.1528, -0.3358]])
Mean of trace:  tensor([ 2.0691, -0.2042])
and variance of trace:  tensor([0.0838, 1.2141])
The covariance matrix:
```

```
[[ 0.08381463 -0.29515154]
[-0.29515154  1.21406202]]
```



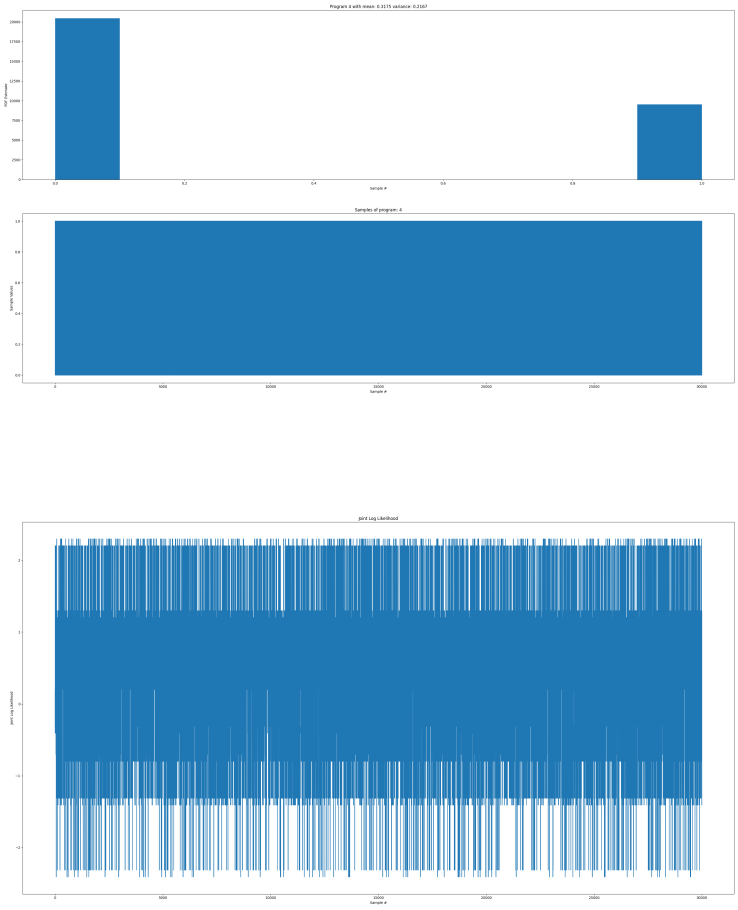
2.3 Program 3

```
Collect samples denoted by program 3:
Elapsed time for program  3 .daphne is:  0:11:39.550165  seconds
tensor([1., 0., 1.,  ..., 1., 1., 1.])
Mean of trace:  tensor(0.6874)
and variance of trace:  tensor(0.2149)
```



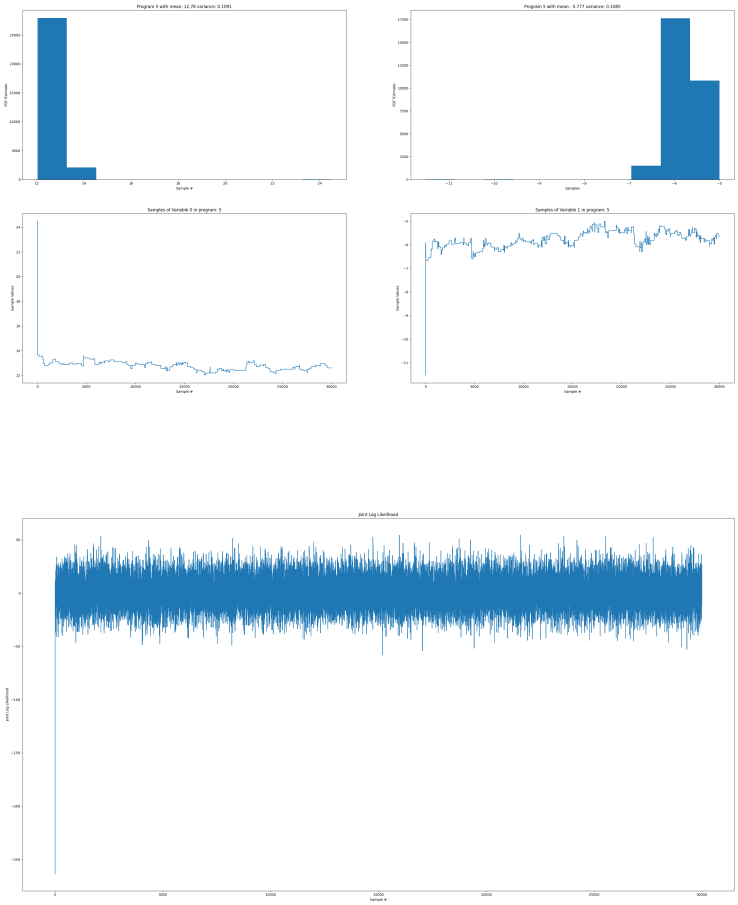
2.4 Program 4

```
Collect samples denoted by program 4:
Elapsed time for program  4 .daphne is:  0:04:13.502343  seconds
tensor([0., 0., 0.,  ..., 1., 1., 0.])
Mean of trace:  tensor(0.3175)
and variance of trace:  tensor(0.2167)
```



2.5 Program 5

```
Collect samples denoted by program 5:
Elapsed time for program  5 .daphne is:  0:01:24.723336  seconds
tensor([[ 24.5000, -11.5213],
[ 13.6642,  -9.7524],
[ 13.6642,  -9.7524],
...,
[ 12.6109,  -5.6384],
[ 12.6109,  -5.6384],
[ 12.6109,  -5.6384]])
Mean of trace:  tensor([12.7786, -5.7767])
and variance of trace:  tensor([0.1091, 0.1085])
```



3 Hamoltanian Monte Carlo - HMC

The HMC algorithm implements algorithms 18 and 19 where Pytorch autodiff is used to compute ∇U. Similarly to the MH within Gibbs, the unobserved variables (latent) are initialized sampled from their respective priors and the Leapfrog integration technique numerically integrates the Hamiltonian formulation which is used to computed the acceptance ratio for a proposed new assignment for X:

```
    for s in range(num_samples):
      jll.append(0)
      for x in X:
        R[x] = dist.Normal(0,M).sample()

      [Xp,Rp] = leapfrog(X,R,T,eps)
```

```
      u = dist.Uniform(0,1).sample()

      if u < torch.exp(-H(Xp,Rp,M) + H(X,R,M)):
```

The Leapfrog integration algorithm implemented with dictionaries (would like to turn into matrix-vector operations!) follows algorithm 19:

```
def leapfrog(X,R,T,eps):
  Rh = {}
  gradU = grad(X)
  for x in X:
    Rh[x] = R[x] - 0.5*eps*gradU[x]
  Xt = X.copy()
  for t in range(1,T-1):
    for x in X:
      Xt[x] = Xt[x].detach() + eps*Rh[x]
      Xt[x].requires_grad_(True) #re-init gradient for next
    iteration
    gradU = grad(Xt)
    for x in X:
      Rh[x] = Rh[x] - eps*gradU[x]
  XT ={}
  RT = {}
  for x in X:
    XT[x] = Xt[x].detach() + eps*Rh[x]
    XT[x].requires_grad_(True)
  gradU = grad(XT)
  for x in X:
    RT[x] = Rh[x] - 0.5*eps*gradU[x]
  return XT,RT
```

Where ∇U is computed:

```
def grad(X):
  gradU = {}
  u = 0
  for x in vertices:
    evalX = bindingVars(X,linkFuncs[x])
    if(evalX[0] == 'sample*'):
      evalX[0] = 'observe*'
      evalX.append(X[x].clone().detach())

    u -= deterministic_eval(evalX)

  u.backward()
  for x in X:
    gradU[x] = X[x].grad
  return gradU
```

It is important to detach the gradient information at the next iteration because we want the gradient at the current point, not the gradient of the current point and all previous points. Lastly the Hamiltonian is computed as:

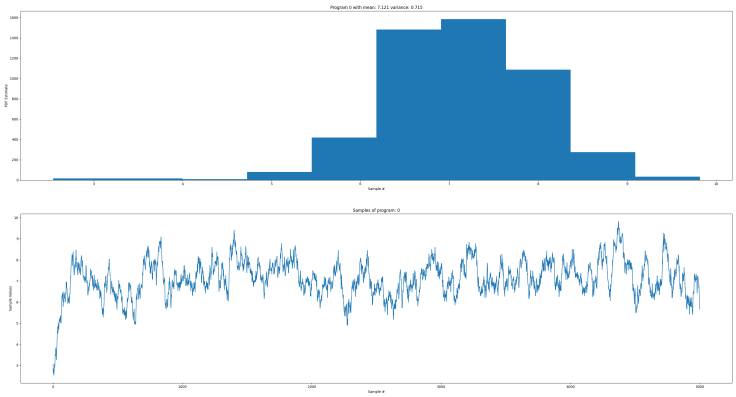
```
          linkFuncsEval.append(Xjll[x].clone().detach())
        jll[s] = jll[s] + deterministic_eval(linkFuncsEval)

      Xs.append(deterministic_eval(retExp).detach())

```

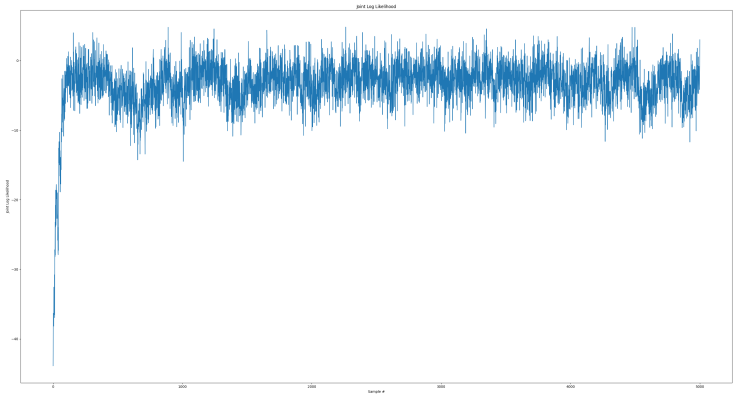
```
6
7       return Xs,jll
8
9   def H(X,R,M):
10    U = 0
11    Rv = []
12    for x in vertices:
13      evalX = bindingVars(X,linkFuncs[x])
14      if(evalX[0] == 'sample*'):
15        evalX[0] = 'observe*'
16        evalX.append(X[x].detach())
```

Picking the number of integration steps and the step size to take from the gradient required some trial and error. A step size that is too large results in poor acceptance and no convergence, however a step size too small results in requiring many more samples to converge or explore the space. Here 5k samples are collected with each integration taking $T = 20$ steps, and a step size $\epsilon = 0.01$ works reasonably well.

3.1 Program 1

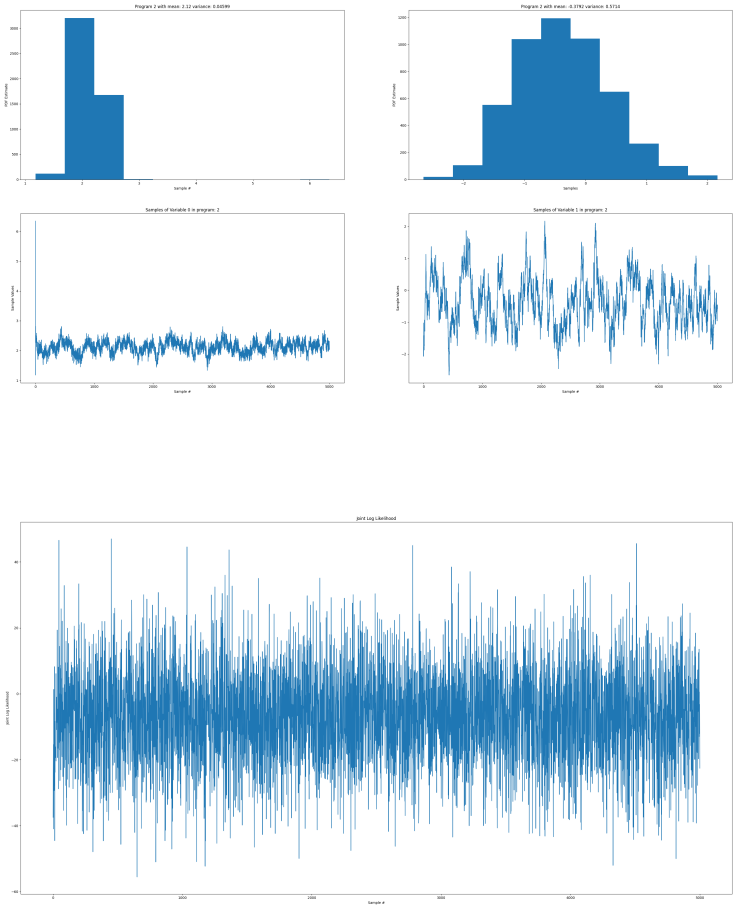
```
Collect samples denoted by program 1:
Elapsed time for program  1 .daphne is:  0:01:41.703409  seconds
tensor(7.1211)
tensor([3.0656, 2.6720, 2.6331,  ..., 5.7209, 5.7209, 5.6698])
Mean of trace:  tensor(7.1211)
and variance of trace:  tensor(0.7150)
```





3.2 Program 2

```
Collect samples denoted by program 2:
Elapsed time for program  2 .daphne is:  0:04:12.794452  seconds
tensor([ 2.1201, -0.3792])
tensor([[ 6.3440, -1.0281],
[ 1.1802, -2.0720],
[ 2.8143, -1.7802],
...,
[ 2.2089, -0.4581],
[ 2.1194, -0.5892],
[ 2.0376, -0.5493]])
Mean of trace:  tensor([ 2.1201, -0.3792])
and variance of trace:  tensor([0.0460, 0.5714])
The covariance matrix:
[[ 0.04598997 -0.1326305 ]
[-0.1326305   0.57140268]]
```



3.3 Program 5

For program 5, I had to change the step size parameters ϵ from 0.1 to 0.01 otherwise HMC fails. This allows the system to be kicked harder. However notice that when it snaps into the Dirac distributed area it stays there.

```
Collect samples denoted by program 5:
Elapsed time for program  5 .daphne is:  0:01:34.285442  seconds
tensor([-2.4993,  9.3750])
tensor([[-1.2606,  8.4221],
[ 0.1093,  6.8895],
[ 0.1093,  6.8895],
...,
[-2.5023,  9.3779],
```

```
[-2.5023,  9.3779],
[-2.5023,  9.3779]])
Mean of trace:  tensor([-2.4993,  9.3750])
and variance of trace:  tensor([0.0080, 0.0075])
```