HW3

Geoffrey Woollard

My code lives in the repo https://github.com/geoffwoollard/prob_prog

Acknowledgments

I acknowledge helpful discussions with Justice Sefas, Masoud Mokhatari, Dylan Green, and Jordan Lovrod, and many other classmates.

I gratefully acknowledge helpful code snippets from Masoud Mokhatari, Mohamad Amin Mohamadi, and Dylan Green, in particular during the implementation of Hamiltonian Monte Carlo.

Code snippets

In [2]:
```python
from dill.source import getsource, getsourcelines
```

Importance sampling

- I modified the evaluator from hw2 to also return σ, which gets accumulated from the `log_prob` of evaluating observes

In [3]:
```python
for line_number, function_line in enumerate(getsourcelines(evaluate)[0]):
    print(line_number, function_line,end='')
```

```
0 def evaluate(e,sigma=0,local_env={},defn_d={},do_log=False,logger_string=''):
1     # TODO: get local_env to evaluate values to tensors, not regular floats
2     # remember to return evaluate (recursive)
3     # everytime we call evaluate, we have to use local_env, otherwise it gets overwritten with the default {}
4     # if do_log: logger.info('logger_string {}'.format(logger_string))
5     if do_log: logger.info('ls {}'.format(logger_string))
6     if do_log: logger.info('e {}, local_env {}, sigma {}'.format(e, local_env, sigma))
7 
8     # get first expression out of list or list of one
9     if not isinstance(e,list) or len(e) == 1:
10         if isinstance(e,list):
11             e = e[0]
12         if isinstance(e,bool):
13             if do_log: logger.info('match case number: e {}, sigma {}'.format(e, sigma))
14             return torch.tensor(e), sigma
15         if isinstance(e, number):
16             if do_log: logger.info('match case number: e {}, sigma {}'.format(e, sigma))
17             return torch.tensor(float(e)), sigma
18         elif isinstance(e,list):
19             if do_log: logger.info('match case list: e {}, sigma {}'.format(e, sigma))
20             return e, sigma
21         elif e in list(primitives_d.keys()):
22             if do_log: logger.info('match case primitives_d: e {}, sigma {}'.format(e, sigma))
23             return e, sigma
24         elif e in list(distributions_d.keys()):
25             if do_log: logger.info('match case distributions_d: e {}, sigma {}'.format(e, sigma))
26             return e, sigma
27         elif torch.is_tensor(e):
28             if do_log: logger.info('match case is_tensor: e {}, sigma {}'.format(e, sigma))
29             return e, sigma
30         elif e in local_env.keys():
31             if do_log: logger.info('match case local_env: e {}, sigma {}'.format(e, sigma))
32             if do_log: logger.info('match case local_env: local_env[e] {}'.format(local_env[e]))
33             return local_env[e], sigma # TODO return evaluate?
34         elif e in list(defn_d.keys()):
35             if do_log: logger.info('match case defn_d: e {}, sigma {}'.format(e, sigma))
36             return e, sigma
37         elif isinstance(e,distribution_types):
38             if do_log: logger.info('match case distribution: e {}, sigma {}'.format(e,sigma))
39             return e, sigma
40         else:
41             assert False, 'case not matched'
42     elif e[0] == 'sample':
43         if do_log: logger.info('match case sample: e {}, sigma {}'.format(e,sigma))
44         distribution, sigma = evaluate(e[1],sigma,local_env,defn_d,do_log=do_log)
45         return distribution.sample(), sigma # match shape in number base case
46     elif e[0] == 'observe':
47         if do_log: logger.info('match case observe: e {}, sigma {}'.format(e,sigma))
48         e1, e2 = e[1:]
49         d1, sigma = evaluate(e1,sigma,local_env,defn_d,do_log=do_log)
50         c2, sigma = evaluate(e2,sigma,local_env,defn_d,do_log=do_log)
51         log_w =score(d1,c2)
52         if do_log: logger.info('match case observe: d1 {}, c2 {}, log_w {}, sigma {}'.format(e,d1, c2, log_
w, sigma))
53         sigma += log_w
54         return c2, sigma
55     elif e[0] == 'let':
56         if do_log: logger.info('match case let: e {}, sigma {}'.format(e, sigma))
57         # let [v1 e1] e0
58         # here
59             # e[0] : "let"
60             # e[1] : [v1, e1]
61             # e[2] : e0
62         # evaluates e1 to c1 and binds this value to e0
63         # this means we update the context with old context plus {v1:c1}
64         c1, sigma = evaluate(e[1][1],sigma,local_env,defn_d,do_log=do_log) # evaluates e1 to c1
65         v1 = e[1][0]
66         return evaluate(e[2], sigma, local_env = {**local_env,v1:c1},defn_d=defn_d,do_log=do_log)
67     elif e[0] == 'if': # if e0 e1 e2
68         if do_log: logger.info('match case if: e {}, sigma {}'.format(e, sigma))
69         e1 = e[1]
70         e2 = e[2]
71         e3 = e[3]
72         e1_prime, sigma = evaluate(e1,sigma,local_env,defn_d,do_log=do_log)
73         if e1_prime:
74             return evaluate(e2,sigma,local_env,defn_d,do_log=do_log)
75         else:
76             return evaluate(e3,sigma,local_env,defn_d,do_log=do_log)
77 
78     else:
79         cs = []
80         for ei in e:
81             if do_log: logger.info('cycling through expressions: ei {}, sigma {}'.format(ei,sigma))
82             c, sigma = evaluate(ei,sigma,local_env,defn_d,do_log=do_log)
83             cs.append(c)
84         if cs[0] in primitives_d:
85             if do_log: logger.info('do case primitives_d: cs0 {}'.format(cs[0]))
86             if do_log: logger.info('do case primitives_d: cs1 {}'.format(cs[1:]))
87             if do_log: logger.info('do case primitives_d: primitives_d[cs[0]] {}'.format(primitives_d[cs
[0]]))
88             return primitives_d[cs[0]](cs[1:]), sigma
89         elif cs[0] in distributions_d:
90             if do_log: logger.info('do case distributions_d: cs0 {}'.format(cs[0]))
91             return distributions_d[cs[0]](cs[1:]), sigma
92         elif cs[0] in defn_d:
93             if do_log: logger.info('do case defn: cs0  {}'.format(cs[0]))
94             defn_function_li = defn_d[cs[0]]
95             defn_function_args, defn_function_body = defn_function_li
96             local_env_update = {key:value for key,value in zip(defn_function_args, cs[1:])}
97             if do_log: logger.info('do case defn: update to local_env from defn_d {}'.format(local_env_updat
e))
98             return evaluate(defn_function_body,sigma,local_env = {**local_env, **local_env_update},defn_d=de
fn_d,do_log=do_log)
99         else:
100              assert False, 'not implemented'
```

I also wrote my own score function. It handles boolean cases by converting them to

In [4]:
```python
from evaluation_based_sampling import score
for line_number, function_line in enumerate(getsourcelines(score)[0]):
    print(line_number, function_line,end='')
```

```
0 def score(distribution,c):
1     """Score pytorch distributions with .log_prob, but in a robust way for the type of c
2     """
3     if isinstance(c,bool) or c.type() in ['torch.BoolTensor', 'torch.LongTensor']:
4         log_w = distribution.log_prob(c.double())
5     else:
6         log_w = distribution.log_prob(c)
7     return log_w
```

I added a few more distributions and boolean operation primitives, for the problems in this assignment

In [6]:
```python
from primitives import distributions_d
for key in distributions_d.keys() :
    print(key,':')
    for line_number, function_line in enumerate(getsourcelines(distributions_d[key])[0]):
        print(line_number, function_line,end='')
    print()
```

```
normal :
0 def normal(mean_std):
1     return two_arg_op_primitive(torch.distributions.Normal,mean_std)

beta :
0 def beta(alpha_beta):
1     return two_arg_op_primitive(torch.distributions.Beta,alpha_beta)

exponential :
0 def exponential(lam):
1     return one_arg_op_primitive(torch.distributions.Exponential,lam)

uniform :
0 def uniform(low_hi):
1     return two_arg_op_primitive(torch.distributions.Uniform,low_hi)

discrete :
0 def discrete(prob_vector):
1     return one_arg_op_primitive(torch.distributions.Categorical,prob_vector)

flip :
0 def flip(prob):
1     return one_arg_op_primitive(torch.distributions.bernoulli.Bernoulli,prob)

dirichlet :
0 def dirichlet(concentration):
1     return one_arg_op_primitive(torch.distributions.dirichlet.Dirichlet,concentration)

gamma :
0 def gamma(concentration_rate):
1     return two_arg_op_primitive(torch.distributions.gamma.Gamma,concentration_rate)
```

In [7]:
```python
from primitives import primitives_d
for key in ['and','or','>','<','>=','<=','='] :
    print(key,':')
    for line_number, function_line in enumerate(getsourcelines(primitives_d[key])[0]):
        print(line_number, function_line,end='')
    print()
```

```
and :
0 def and_primitive(arg1_arg2):
1     return two_arg_op_primitive(torch.logical_and,arg1_arg2)

or :
0 def or_primitive(arg1_arg2):
1     return two_arg_op_primitive(torch.logical_or,arg1_arg2)

> :
0 def gt_primitive(consequent_alternative):
1     return two_arg_op_primitive(torch.gt,consequent_alternative)

< :
0 def lt_primitive(consequent_alternative):
1     return two_arg_op_primitive(torch.lt,consequent_alternative)

>= :
0 def ge_primitive(consequent_alternative):
1     return two_arg_op_primitive(torch.ge,consequent_alternative)

<= :
0 def le_primitive(consequent_alternative):
1     return two_arg_op_primitive(torch.le,consequent_alternative)

= :
0 def eq_primitive(consequent_alternative):
1     return two_arg_op_primitive(torch.eq,consequent_alternative)
```

MH within Gibbs

I implented Metropolis-Hastings within Gibbs in the following manner

- parse the graph in `mh_gibbs_wrapper`
- topologically sort the graph vertices
- sample from the joint (ie prior) to initialize all values of the graph
- cycle through the graph with `gibbs_step`
- accept an update at a specific vertex with `accept`
- collect each state after a Gibbs update (all the vertices)
- return all the fully specified graphs in `gibbs` for `num_steps`
- finally, I evaluate return value (the meaning of the program) for all the sampled graphs with `evaluate_program_return_from_samples_whole_graph`

In [207…
```python
from mh_gibbs import mh_gibbs_wrapper,gibbs_step,accept,gibbs,evaluate_program_return_from_samples_whole_graph
list_of_programs = [mh_gibbs_wrapper,gibbs_step,accept,gibbs,evaluate_program_return_from_samples_whole_graph]

for program in list_of_programs:
    for line_number, function_line in enumerate(getsourcelines(program)[0]):
        print(line_number, function_line,end='')
    print()
```

```
0 def mh_gibbs_wrapper(graph,num_steps,do_log=False):
1     G = graph[1]
2     verteces = G['V']
3     A = G['A']
4     P = G['P']
5     X = set(verteces) - set(G['Y'].keys())
6     Y = G['Y']
7     Y = {key:evaluate([Y[key]], do_log=do_log)[0] for key in Y.keys()}
8 
9 
10      verteces_topsorted = sample_from_joint_precompute(graph)
11      _, local_env = sample_from_joint(graph,verteces_topsorted=verteces_topsorted)
12      local_env = {**local_env,**Y}
13      local_env_list0 = [local_env]
14 
15      local_env_list = gibbs(num_steps,local_env,P,A,X,do_log=do_log)
16 
17      local_env_list = local_env_list0 + local_env_list
18 
19      return_list, samples_whole_graph = evaluate_program_return_from_samples_whole_graph(graph,local_env_lis
t)
20 
21 
22 
23      return local_env_list

0 def gibbs_step(local_env,P,A,X_sample_vertices,do_log):
1     for vertex in X_sample_vertices:
2         link_function = P[vertex]
3         e = link_function[1]
4 
5         distribution, sigma = evaluate(e,local_env=local_env,do_log=do_log)
6         local_env_prime = local_env.copy()
7         local_env_prime[vertex] = distribution.sample()
8 
9         alpha = accept(vertex,local_env,local_env_prime,A,P,do_log=do_log)
10          u = torch.rand(1)
11          if u < alpha:
12              local_env = local_env_prime
13      return local_env

0 def accept(vertex,local_env,local_env_prime,A,P,do_log):
1     link_function = P[vertex]
2     e = link_function[1]
3     d_q, _ = evaluate(e,local_env=local_env,do_log=do_log)
4     d_q_prime, _ = evaluate(e,local_env=local_env_prime,do_log=do_log)
5     log_a = d_q_prime.log_prob(local_env[vertex]) - d_q.log_prob(local_env_prime[vertex])
6     V_x = A[vertex] + [vertex]
7     for observed_vertex in V_x:
8         d_p_prime = evaluate(P[observed_vertex][1],local_env = local_env_prime,do_log=do_log)[0]
9         if do_log: evaluate(1,do_log=do_log,logger_string='d_p_prime {}'.format(d_p_prime))
10 
11          d_p = evaluate(P[observed_vertex][1],local_env = local_env,do_log=do_log)[0]
12          if do_log: evaluate(1,do_log=do_log,logger_string='d_p {}'.format(d_p))
13          if do_log: evaluate(1,do_log=do_log,logger_string='local_env_prime {}, observed_vertex {}, local_env
_prime[observed_vertex] {}'.format(local_env_prime,observed_vertex,local_env_prime[observed_vertex]))
14          log_a += score(d_p_prime,local_env_prime[observed_vertex])
15          # log_a += d_p_prime.log_prob(local_env_prime[observed_vertex])
16          log_a -= score(d_p,local_env[observed_vertex])
17          # log_a -= d_p.log_prob(local_env[observed_vertex])
18 
19      return torch.exp(log_a)

0 def gibbs(num_steps,local_env,P,A,X,do_log):
1     local_env_list = []
2     for step in range(num_steps):
3         local_env = gibbs_step(local_env,P,A,X,do_log=do_log)
4         local_env_list.append(local_env)
5     return local_env_list

0 def evaluate_program_return_from_samples_whole_graph(graph,samples_whole_graph):
1     # evaluate samples (on whatever function, here the return of the program) as needed
2     e = graph[2]
3     # TODO suggest daphne put return as program, so return is ['sample2'] not 'sample2'
4     if isinstance(e,str):
5         e = [e]
6     return_list = []
7     for X_s in samples_whole_graph:
8         return_s, _ = evaluate(e,local_env = X_s) # TODO: handle defns
9         return_list.append(return_s)
10      return return_list
```

Hamiltonian Monte Carlo

I implemented HMC in the following way

- I parse the graph for the link functions P, and the samples X and observes Y
- I turn on autodiff on the torch.tensor(float): `turn_on_autodiff`
- I run HMC algorithm 20 from the textbook: `hmc_algo20`
  - inside I use the leapfrog algorithm 19 from the textbook: `leapfrog`
    - this relies on computing the gradient of the potential energy with respect to the values of X: `grad_U`. There are important impelentation details with pytorch autodiff, avoiding gradient accumulation
    - I also have to add Xt (a dict) and Rt (a vector) with a helper function `add_dict_to_tensor`. I use `X_vertex_names_to_idx_d` to keep track of what key in `X` corresponds to what index of `R`. This is important if the keys change order and `M` is different for different values of `X` (i.e. not proportional to the identity matrix)
  - I compute the kinetic and potential energy and the hamiltonian: `compute_K`, `compute_U` (just negative of `compute_log_joint_prob`) and `compute_H`
- After I collect samples from the whole graph, I evaluate the return function on each graph.

In [208…
```python
from hmc import hmc_wrapper,turn_on_autodiff,hmc_algo20,leapfrog,grad_U,add_dict_to_tensor,compute_K,compute_lo

list_of_programs = [hmc_wrapper,turn_on_autodiff,hmc_algo20,leapfrog,grad_U,add_dict_to_tensor,compute_K,comput

for program in list_of_programs:
    for line_number, function_line in enumerate(getsourcelines(program)[0]):
        print(line_number, function_line,end='')
    print()
```

```
0 def hmc_wrapper(graph,num_samples,T=10,epsilon=0.1,M=tensor(1.)):
1     #set up X, Y list of verteces
2     G = graph[1]
3     verteces = ['V']
4     Y = G['Y']
5     P = G['P']
6 
7     # evaluate to constants
8     Y = {key:evaluate([value])[0] for key,value in Y.items()}
9 
10      #X = set(vertices) - set(Y.keys())
11      #X = sample_from_joint(graph)
12      _, X0 = sample_from_joint(graph) # does not include observes
13 
14      # initialize in dict
15      X_vertex_names_to_idx_d = {key:idx for idx,key in enumerate(X0.keys())}
16 
17      # set up autograd on tensors
18      turn_on_autodiff(X0)
19      turn_on_autodiff(Y) # TODO: why do we need this?
20 
21 
22      # run HMC algorithm 20 from book
23          # inside use leapfrog algorithm 19 from book
24          # include kinetic and potential energy functions
25          # MC acceptance criteria
26      samples_whole_graph = hmc_algo20(X0,num_samples,T,epsilon,M,Y,P,X_vertex_names_to_idx_d)
27 
28      # evaluate samples (on whatever function, here the return of the program) as needed
29      e = graph[2]
30      # TODO suggest daphne put return as program, so return is ['sample2'] not 'sample2'
31      if isinstance(e,str):
32          e = [e]
33      return_list = []
34      for X_s in samples_whole_graph:
35          return_s, _ = evaluate(e,local_env = X_s) # TODO: handle defns
36          return_list.append(return_s)
37 
38      return return_list, samples_whole_graph

0 def turn_on_autodiff(dictionary_of_tensors):
1     """
2     cant be integers, ie long tensors, but floats or complex
3     """
4     for x in dictionary_of_tensors.values():
5         if torch.is_tensor(x):
6             x.requires_grad = True

0 def hmc_algo20(X0,num_samples,T,epsilon,M,Y,P,X_vertex_names_to_idx_d):
1     X_s = X0
2     samples = []
3     size = len(X0.keys())
4     normal_R_reuse = torch.distributions.Normal(torch.zeros(size),M)
5 
6     for s in range(num_samples):
7         R_s = normal_R_reuse.sample()
8         R_p, X_p = leapfrog(copy.deepcopy(X_s),copy.deepcopy(R_s),T,epsilon,Y,P,X_vertex_names_to_idx_d)
9         # X_p, R_p = leapfrog(X_s,R_s,T,epsilon,X_vertex_names_to_idx_d)
10              # copy X_s?
11          u = torch.rand(1)
12          delta_H = compute_H(X_p,R_p,M,Y,P) - compute_H(X_p,R_p,M,Y,P)
13          boltzmann_ratio = torch.exp(-delta_H)
14          if u < boltzmann_ratio:
15              X_s = X_p
16          #no need to update X_s because should stay the same for next round.
17          #X_s turns into X_s_minus from algo 20 by indexing
18          samples.append(X_s)
19      return samples

0 def leapfrog(X0,R0,T,epsilon,Y,P,X_vertex_names_to_idx_d):
1     """
2     leapfrog as in algo 19 of book
3     Y and P needed for grad calc
4     """
5 
6     epsilon_2 = epsilon/2
7     R_t = R0 - epsilon_2 * grad_U(X0,Y,P,X_vertex_names_to_idx_d)
8 
9     X_t = X0
10      for t in range(T-1):
11          # TODO: save all in loop instead of overwriting to visualize
12          X_t = add_dict_to_tensor(X_t,epsilon*R_t,X_vertex_names_to_idx_d)
13          R_t = R_t - epsilon*grad_U(X_t,Y,P,X_vertex_names_to_idx_d)
14      X_T = add_dict_to_tensor(X_t,epsilon*R_t,X_vertex_names_to_idx_d)
15      R_T = R_t - epsilon_2*grad_U(X_T,Y,P,X_vertex_names_to_idx_d)
16      return R_T, X_T

0 def grad_U(X,Y,P,X_vertex_names_to_idx_d):
1     """
2     call autodiff backward pass, with constant indexing given by X_vertex_names_to_idx_d
3     return vector of gradients
4     """
5     energy_U = compute_U(X,Y,P)
6 
7     # Zero the gradients.
8     # without this running grad_U back to back accumulates the grad (not what we want!)
9     for key in X.keys():
10          if X[key].grad is not None:
11              X[key].grad.zero_()
12 
13      energy_U.backward()
14      grads = torch.zeros(len(X.keys()))
15      for key in X.keys():
16 #         print('key',key)
17          idx = X_vertex_names_to_idx_d[key]
18 #         print('idx',idx)
19 #         print('X[key]',X[key])
20 #         print('X[key].grad',X[key].grad)
21          grads[idx] = X[key].grad
22              # Need to have been referenced in linking function.
23              # So key connected to evaluation of energy.
24              # Otherwise key not a part of the computational graph of energy_U,
25                  # and grad remains none when run energy_U.backwards()
26      return grads

0 def add_dict_to_tensor(X,R,X_vertex_names_to_idx_d):
1     """
2     X+R using mapping from X_vertex_names_to_idx_d
3     TODO: avoid detach?
4     R must be dimension 1, not 0 D
5     similar to using with torch.no_grad() as in https://github.com/MasoudMo/cpsc532w_hw/blob/master/HW3/graph
_based_sampling.py#L275
6     """
7     assert R.dim() >= 1
8     X_new = {}
9     for vertex in X.keys():
10          idx = X_vertex_names_to_idx_d[vertex]
11          # overwriting the value, and only want to autograd accumulated gradient to depend on the final value
12          # TODO: would this problem go away if we stored a vector over all leapfrog time steps?
13          X_new[vertex] = X[vertex].detach() + R[idx]
14          X_new[vertex].requires_grad = True
15      return X_new

0 def compute_K(R,M):
1     R_over_2M = R/(2*M) # TODO: generalize for non scalar M, e.g. diagonal M
2     if R.dim() == 0:
3         energy_K = R*R_over_2M
4     elif R.dim() >= 1:
5         energy_K = torch.matmul(R,R_over_2M)
6     else:
7         assert False
8     return energy_K

0 def compute_log_joint_prob(X,Y,P):
1     """
2     call link functions under context and score.
3     TODO: remove Y dependence
4     TODO: add in user defns https://github.com/MasoudMo/cpsc532w_hw/blob/master/HW3/graph_based_sampling.py#L
208
5     need to parse link functions like
6         # 'sample2': ['sample*', ['normal', 1, ['sqrt', 5]]]
7         # 'observe3': ['observe*', ['normal', 'sample2', ['sqrt', 2]], 8],
8     """
9     log_prob = tensor(0.0)
10      for X_vertex in X.keys():
11          e = P[X_vertex][1]
12          distribution = evaluate(e,local_env=X)[0]
13          log_prob += score(distribution,X[X_vertex])
14      for Y_vertex in Y.keys():
15          e = P[Y_vertex][1]
16          distribution = evaluate(e,local_env=X)[0]
17          log_prob += score(distribution,Y[Y_vertex])
18 
19      return log_prob

0 def compute_U(X,Y,P):
1     energy_U = -compute_log_joint_prob(X,Y,P)
2     return energy_U

0 def compute_H(X,R,M,Y,P):
1     """Compute Hamiltonian.
2     """
3     energy_U = compute_U(X,Y,P)
4     energy_K = compute_K(R,M)
5     energy_H = energy_U + energy_K
6     return energy_H
```

In [3]:
```
import torch
import numpy as np
import os, json
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import importlib

from evaluation_based_sampling import evaluate, evaluate_program
from daphne import daphne

from graph_based_sampling import sample_from_joint
```

Problem 1

In [2]:
```
from load_helper import ast_helper, graph_helper
```

Importance sampling

10k in 3.34s implies 1.7e6 in 10 min

In [3]:
```
import parse
import importance_sampling
import importlib
importlib.reload(parse)
```

Out[3]:
```
<module 'parse' from '/Users/gw/repos/prob_prog/hw/hw3/parse.py'>
```

In [35]:
```
fname = '1.daphne'
ast = ast_helper(fname)
ast
```

Out[35]:
```
[['let',
  ['mu', ['sample', ['normal', 1, ['sqrt', 5]]]],
  ['let',
   ['sigma', ['sqrt', 2]],
   ['let',
    ['lik', ['normal', 'mu', 'sigma']],
    ['let',
     ['dontcare0', ['observe', 'lik', 8]],
     ['let', ['dontcare1', ['observe', 'lik', 9]], 'mu']]]]]]
```

In [53]:
```
%%time
num_samples=1790000
samples, sigmas = parse.take_samples(num_samples,ast)
```

```
CPU times: user 10min 27s, sys: 3.59 s, total: 10min 31s
Wall time: 10min 32s
```

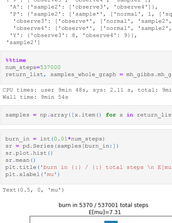
In [54]:
```
samples = np.array([sample.item() for sample in samples])
```

In [55]:
```
posterior_mean, probs = importance_sampling.weighted_average(samples,sigmas)
posterior_mean
```

Out[55]:
```
7.254986058314825
```

In [60]:
```
# pd.Series(samples).plot.hist()
_ = plt.hist(samples, weights=probs, bins=100)
plt.title('Problem 1 \n Importance sampling \n importance sampling weighted counts from proposal'.format(fname))
plt.xlabel('mu')
plt.ylabel('(weighted) Counts')
```

Out[60]:
```
Text(0, 0.5, '(weighted) Counts')
```



In [57]:
```
expectation_samples_2, probs = importance_sampling.weighted_average(samples**2,sigmas)
posterior_variance = expectation_samples_2 - posterior_mean**2
```

In [58]:
```
"Importance sampling: posterior mean of mu {:1.3f} | variance {:1.3e}".format(posterior_mean,posterior_variance)
```

Out[58]:
```
'Importance sampling: posterior mean of mu 7.255 | variance 8.292e-01'
```

MH within Gibbs

5k in 5.58s implies 537k samples in 10 min

In [111]:
```
import mh_gibbs
from hmc import compute_log_joint_prob
importlib.reload(mh_gibbs)
```

Out[111]:
```
<module 'mh_gibbs' from '/Users/gw/repos/prob_prog/hw/hw3/mh_gibbs.py'>
```

In [112]:
```
fname = '1.daphne'
graph = graph_helper(fname)
graph
```

Out[112]:
```
[(),
 {'V': ['observe3', 'observe4', 'sample1'],
  'A': {'sample2': ['observe3', 'observe4']},
  'P': {'sample1': ['sample', ['normal', 1, ['sqrt', 5]]],
   'observe3': ['observe', ['normal', 'sample2', ['sqrt', 2]], 8],
   'observe4': ['observe', ['normal', 'sample2', ['sqrt', 2]], 9]},
  'T': {'observe3': 0, 'observe4': 0},
  'sample2'}]
```

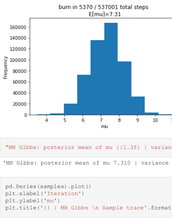
In [113]:
```
%%time
num_steps=537000
return_list, samples_whole_graph = mh_gibbs.mh_gibbs_wrapper(graph,num_steps=num_steps)
```

```
CPU times: user 9min 48s, sys: 2.11 s, total: 9min 51s
Wall time: 9min 54s
```

In [114]:
```
samples = np.array([x.item() for x in return_list])
```

In [115]:
```
burn_in = int(0.01*num_steps)
sr = pd.Series(samples[burn_in:])
sr.plot.hist()
sr.mean()
plt.title('burn in {:} / {:} total steps \n E[mu]={:1.2f}'.format(burn_in, len(samples),sr.mean()))
plt.xlabel('mu')
```

Out[115]:
```
Text(0.5, 0, 'mu')
```

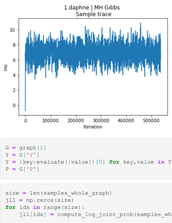


In [116]:
```
"MH Gibbs: posterior mean of mu {:1.3f} | variance {:1.3e}".format(sr.mean(),sr.var())
```

Out[116]:
```
'MH Gibbs: posterior mean of mu 7.310 | variance 8.421e-01'
```

In [117]:
```
pd.Series(samples).plot()
plt.xlabel('Iteration')
plt.ylabel('mu')
plt.title('{} | MH Gibbs \n Sample trace'.format(fname))
```

Out[117]:
```
Text(0.5, 1.0, '1.daphne | MH Gibbs \n Sample trace')
```

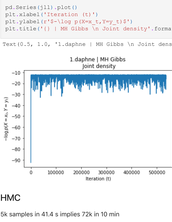


In [118]:
```
G = graph[1]
V = G['V']
T = {key:evaluate([value])[0] for key,value in T.items()}
P = G['P']
```

In [119]:
```
size = len(samples_whole_graph)
jll = np.zeros(size)
for idx in range(size):
    jll[idx] = compute_log_joint_prob(samples_whole_graph[idx],T,P)
```

In [120]:
```
pd.Series(jll).plot()
plt.xlabel('Iteration (t)')
plt.ylabel(r'$-\log p(x_t,s_t,\tau,\rho)$')
plt.title('{} | MH Gibbs \n Joint density'.format(fname))
```

Out[120]:
```
Text(0.5, 1.0, '1.daphne | MH Gibbs \n Joint density')
```



HMC

5k samples in 41.4 s implies 72k in 10 min

In [61]:
```
fname = '1.daphne'
graph = graph_helper(fname)
```

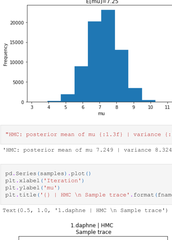
In [62]:
```
import hmc
importlib.reload(hmc)
from hmc import hmc_wrapper
```

In [ ]:
```
num_samples=72000
return_list, samples_whole_graph = hmc_wrapper(graph,num_samples,T=20,epsilon=0.1)
```

In [70]:
```
samples = np.array([x.item() for x in return_list])
```

In [71]:
```
burn_in = int(0.01*num_samples) # ~500 from inspecting joint density plot, with given hyper params T, epsilon,
sr = pd.Series(samples[burn_in:])
sr.plot.hist()
sr.mean()
plt.title('{} | HMC \n {:} total steps \n E[mu]={:1.2f}'.format(fname,len(samples),sr.mean()))
plt.xlabel('mu')
```

Out[71]:
```
Text(0.5, 0, 'mu')
```

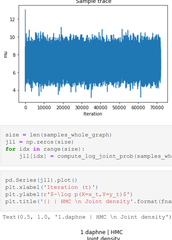


In [72]:
```
"HMC: posterior mean of mu {:1.3f} | variance {:1.3e}".format(sr.mean(),sr.var())
```

Out[72]:
```
'HMC: posterior mean of mu 7.249 | variance 8.324e-01'
```

In [73]:
```
pd.Series(samples).plot()
plt.xlabel('Iteration')
plt.ylabel('mu')
plt.title('{} | HMC \n Sample trace'.format(fname))
```

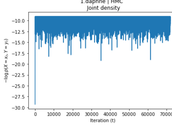
Out[73]:
```
Text(0.5, 1.0, '1.daphne | HMC \n Sample trace')
```



In [74]:
```
size = len(samples_whole_graph)
jll = np.zeros(size)
for idx in range(size):
    jll[idx] = compute_log_joint_prob(samples_whole_graph[idx],T,P)
```

In [75]:
```
pd.Series(jll).plot()
plt.xlabel('Iteration (t)')
plt.ylabel(r'$-\log p(x_t,s_t,\tau,\rho)$')
plt.title('{} | HMC \n Joint density'.format(fname))
```

Out[75]:
```
Text(0.5, 1.0, '1.daphne | HMC \n Joint density')
```



In [1]:
```python
import torch
import numpy as np
import os, json
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import importlib

from evaluation_based_sampling import evaluate, evaluate_program
from daphne import daphne

from graph_based_sampling import sample_from_joint
```

Problem 2

10k samples in 1.59s implies 384k samples

In [2]:
```python
from load_helper import ast_helper, graph_helper
```

Importance sampling

In [3]:
```python
import parse
import importance_sampling
import importlib
importlib.reload(parse)
```

Out[3]:
```
<module 'parse' from '/Users/gw/repos/prob_prog/hw/hw3/parse.py'>
```

In [52]:
```python
fname = '2.daphne'
ast = ast_helper(fname)
```

In [53]:
```python
%%time
num_samples=384000
samples, sigmas = parse.take_samples(num_samples,ast=ast)
```

```
CPU times: user 10min 28s, sys: 918 ms, total: 10min 29s
Wall time: 10min 29s
```

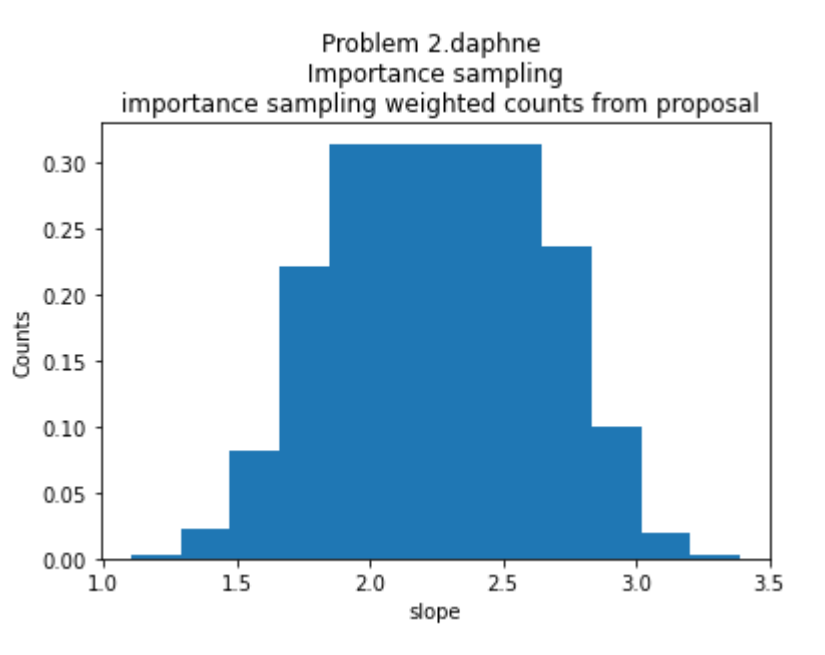
In [54]:
```python
samples = np.array([sample.tolist() for sample in samples])
```

In [55]:
```python
posterior_mean, probs = importance_sampling.weighted_average(samples,sigmas,reshape_probs=(-1,1),axis=0)
posterior_mean
```

Out[55]:
```
array([ 2.15895652, -0.53834023])
```

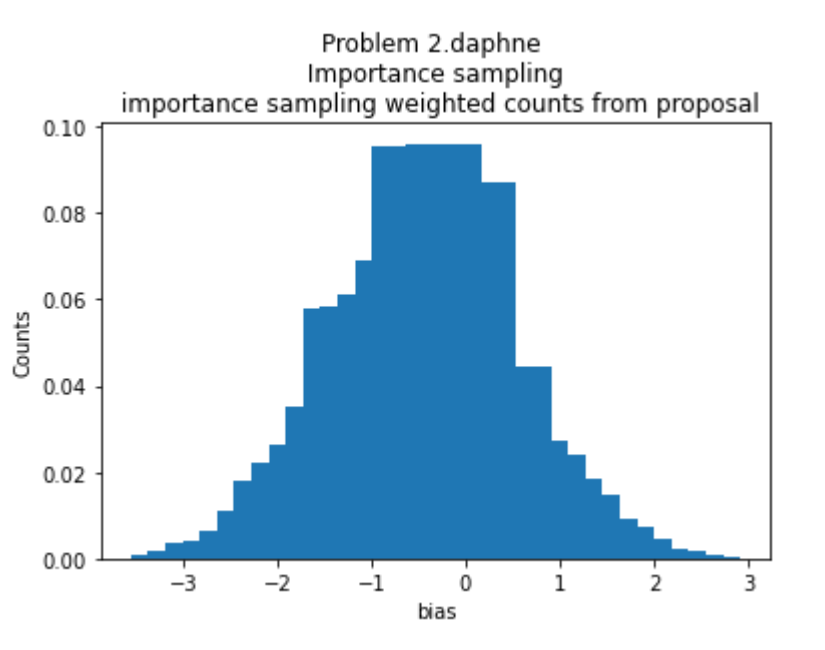
In [106…
```python
counts_bins = np.histogram(samples[:,0], weights=probs, bins=500)
counts, bins = counts_bins[0], counts_bins[1]
idx = (counts > counts.max()*0.005)
plt.bar(bins[1:][idx],counts[idx])
plt.title('Problem {} \n Importance sampling \n importance sampling weighted counts from proposal'.format(fname
plt.ylabel('Counts')
plt.xlabel('slope')
```

Out[106…
```
Text(0.5, 0, 'slope')
```



In [104…
```python
counts_bins = np.histogram(samples[:,1], weights=probs, bins=500)
counts, bins = counts_bins[0], counts_bins[1]
idx = (counts > counts.max()*0.005)
plt.bar(bins[1:][idx],counts[idx])
plt.title('Problem {} \n Importance sampling \n importance sampling weighted counts from proposal'.format(fname
plt.ylabel('Counts')
plt.xlabel('bias')
```

Out[104…
```
Text(0.5, 0, 'bias')
```



In [58]:
```python
expectation_samples_2, probs = importance_sampling.weighted_average(samples**2,sigmas,reshape_probs=(-1,1),axis
posterior_variance = expectation_samples_2 - posterior_mean**2
posterior_variance
```

Out[58]:
```
array([0.05376331, 0.79199928])
```

In [59]:
```python
expectation_samplex_sampley, probs = importance_sampling.weighted_average(samples[:,0]*samples[:,1],sigmas)
covariance = expectation_samplex_sampley - posterior_mean[0]*posterior_mean[1]
covariance
```

Out[59]:
```
-0.18463132614140898
```

In [60]:
```python
for result in [
    "{} Importance sampling: posterior mean slope {:1.3f} | variance slope {:1.3e}".format(fname,posterior_mear
    "{} Importance sampling: posterior mean bias {:1.3f} | variance bias {:1.3e}".format(fname,posterior_mean[1
    "{} Importance sampling: posterior covariance of slope and bias variance bias {:1.3e}".format(fname,covaria

]:
    print(result)
```

```
2.daphne Importance sampling: posterior mean slope 2.159 | variance slope 5.376e-02
2.daphne Importance sampling: posterior mean bias -0.538 | variance bias 7.920e-01
2.daphne Importance sampling: posterior covariance of slope and bias variance bias -1.846e-01
```

Numpy contains it's own method for computing this, and we can check it agrees with our results (where things are spelt out a bit more for learning purposes).

In [61]:
```python
np.cov(samples.T,aweights=probs,ddof=0)
```

Out[61]:
```
array([[ 0.05376331, -0.18463133],
       [-0.18463133,  0.79199928]])
```

MH Gibbs

5k samples in 21.3s implies 140k samples in 10 min

In [12]:
```python
import mh_gibbs
from hmc import hmc_wrapper, compute_log_joint_prob
importlib.reload(mh_gibbs)
```

Out[12]:
```
<module 'mh_gibbs' from '/Users/gw/repos/prob_prog/hw/hw3/mh_gibbs.py'>
```

In [13]:
```python
fname = '2.daphne'
graph = graph_helper(fname)
graph
```

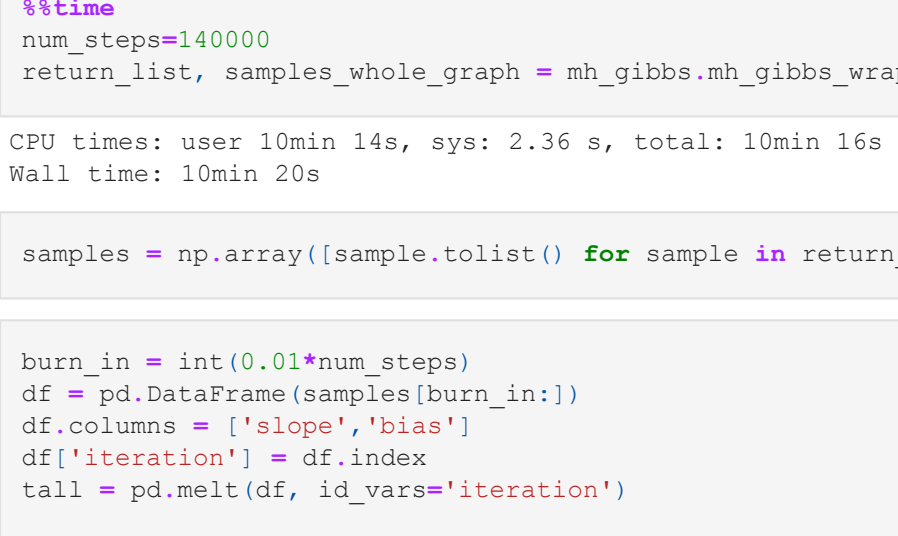
Out[13]:
```
[{'observe-data': ['fn',
   ['_', 'data', 'slope', 'bias'],
   ['let',
    ['xn', ['first', 'data']],
    ['let',
     ['yn', ['second', 'data']],
     ['let',
      ['zn', ['+', ['*', 'slope', 'xn'], 'bias']],
      ['let',
       ['dontcare9', ['observe', ['normal', 'zn', 1.0], 'yn']],
       ['rest', ['rest', 'data']]]]]]]},
 {'V': ['observe3',
   'observe6',
   'observe4',
   'observe7',
   'sample2',
   'sample1',
   'observe8',
   'observe5'],
  'A': {'sample2': ['observe3',
    'observe6',
    'observe4',
    'observe7',
    'observe8',
    'observe5'],
   'sample1': ['observe3',
    'observe6',
    'observe4',
    'observe7',
    'observe8',
    'observe5']},
  'P': {'sample1': ['sample*', ['normal', 0.0, 10.0]],
   'sample2': ['sample*', ['normal', 0.0, 10.0]],
   'observe3': ['observe*',
    ['normal', ['+', ['*', 'sample1', 1.0], 'sample2'], 1.0],
    2.1],
   'observe4': ['observe*',
    ['normal', ['+', ['*', 'sample1', 2.0], 'sample2'], 1.0],
    3.9],
   'observe5': ['observe*',
    ['normal', ['+', ['*', 'sample1', 3.0], 'sample2'], 1.0],
    5.3],
   'observe6': ['observe*',
    ['normal', ['+', ['*', 'sample1', 4.0], 'sample2'], 1.0],
    7.7],
   'observe7': ['observe*',
    ['normal', ['+', ['*', 'sample1', 5.0], 'sample2'], 1.0],
    10.2],
   'observe8': ['observe*',
    ['normal', ['+', ['*', 'sample1', 6.0], 'sample2'], 1.0],
    12.9]},
  'Y': {'observe3': 2.1,
   'observe4': 3.9,
   'observe5': 5.3,
   'observe6': 7.7,
   'observe7': 10.2,
   'observe8': 12.9}},
 ['vector', 'sample1', 'sample2']]
```

In [14]:
```python
%%time
num_steps=140000
return_list, samples_whole_graph = mh_gibbs.mh_gibbs_wrapper(graph,num_steps)
```

```
CPU times: user 10min 14s, sys: 2.36 s, total: 10min 16s
Wall time: 10min 20s
```

In [15]:
```python
samples = np.array([sample.tolist() for sample in return_list])
```

In [16]:
```python
burn_in = int(0.01*num_steps)
df = pd.DataFrame(samples[burn_in:])
df.columns = ['slope','bias']
df['iteration'] = df.index
tall = pd.melt(df, id_vars='iteration')

g = sns.FacetGrid(tall, col="variable")
g.map(sns.histplot, "value")

plt.suptitle('MH Gibbs | {}'.format(fname))
plt.subplots_adjust(top=0.8)
```



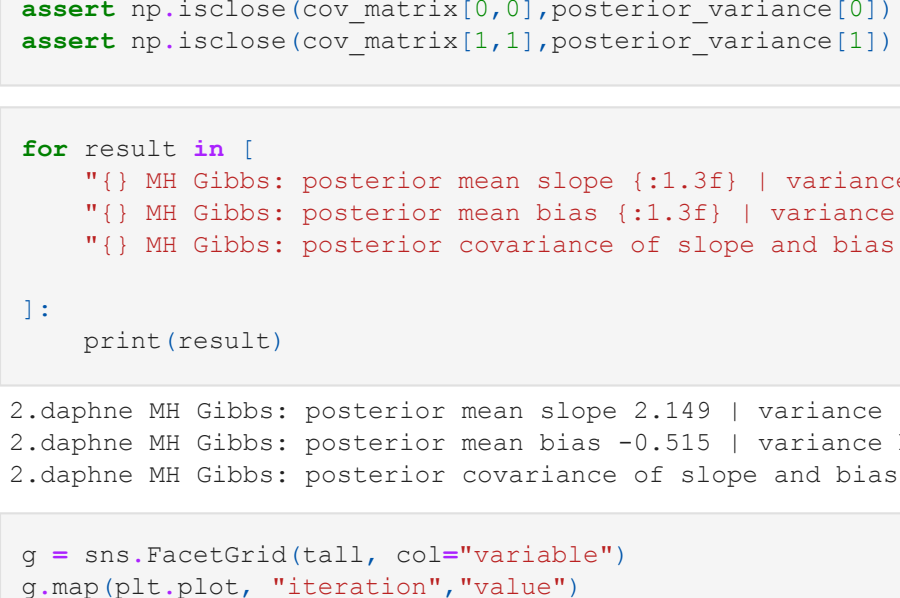
In [17]:
```python
posterior_mean = samples[burn_in:].mean(0)
cov_matrix = np.cov(samples[burn_in:].T,ddof=0)
posterior_variance = samples[burn_in:].var(0)
covariance = cov_matrix[0,1]
assert np.isclose(cov_matrix[0,0],posterior_variance[0])
assert np.isclose(cov_matrix[1,1],posterior_variance[1])
```

In [18]:
```python
for result in [
    "{} MH Gibbs: posterior mean slope {:1.3f} | variance slope {:1.3e}".format(fname,posterior_mean[0],posteri
    "{} MH Gibbs: posterior mean bias {:1.3f} | variance bias {:1.3e}".format(fname,posterior_mean[1],posterior
    "{} MH Gibbs: posterior covariance of slope and bias variance bias {:1.3e}".format(fname,covariance),

]:
    print(result)
```

```
2.daphne MH Gibbs: posterior mean slope 2.149 | variance slope 5.918e-02
2.daphne MH Gibbs: posterior mean bias -0.515 | variance bias 8.965e-01
2.daphne MH Gibbs: posterior covariance of slope and bias variance bias -2.070e-01
```

In [19]:
```python
g = sns.FacetGrid(tall, col="variable")
g.map(plt.plot, "iteration","value")

plt.suptitle('MH Gibbs | {}'.format(fname))
plt.subplots_adjust(top=0.75)
plt.suptitle('{} | MH Gibbs \n Sample trace'.format(fname))
```

Out[19]:
```
Text(0.5, 0.98, '2.daphne | MH Gibbs \n Sample trace')
```

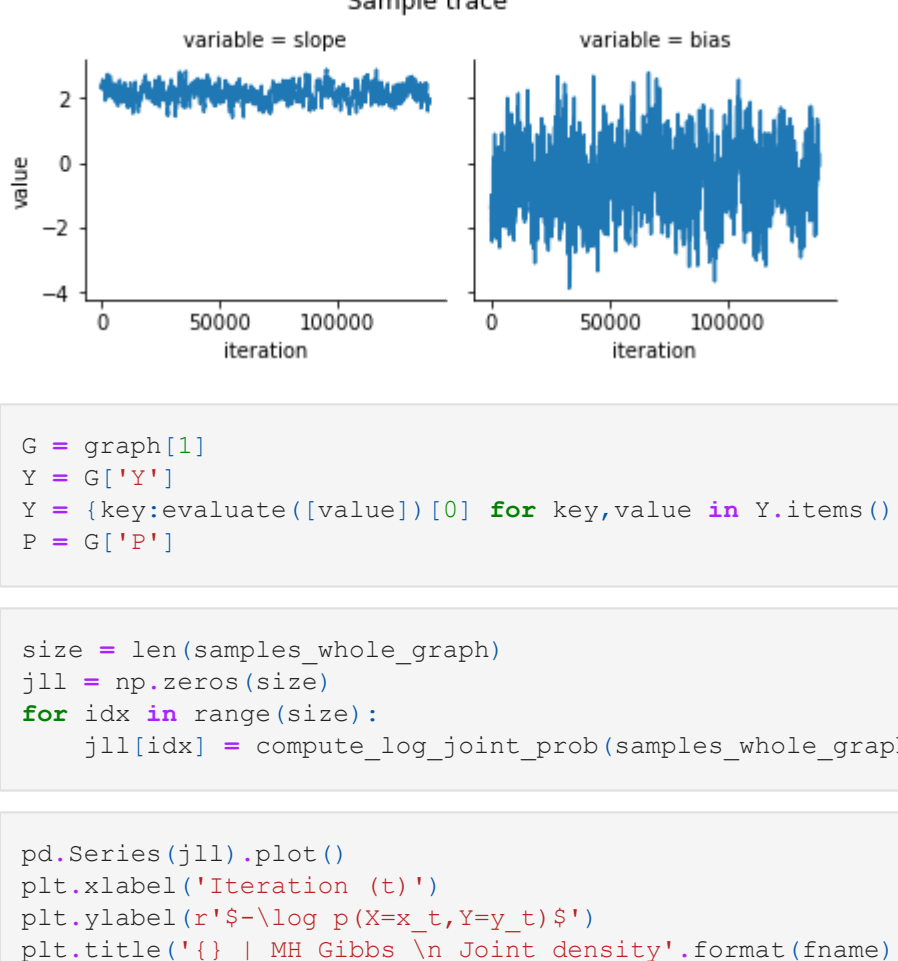


In [20]:
```python
G = graph[1]
Y = G['Y']
Y = {key:evaluate([value])[0] for key,value in Y.items()}
P = G['P']
```

In [21]:
```python
size = len(samples_whole_graph)
jll = np.zeros(size)
for idx in range(size):
    jll[idx] = compute_log_joint_prob(samples_whole_graph[idx],Y,P)
```

In [22]:
```python
pd.Series(jll).plot()
plt.xlabel('Iteration (t)')
plt.ylabel(r'$-\log p(X=x_t,Y=y_t)$')
plt.title('{} | MH Gibbs \n Joint density'.format(fname))
```

Out[22]:
```
Text(0.5, 1.0, '2.daphne | MH Gibbs \n Joint density')
```



HMC

4.2s/ 200 samples implies 28.5k samples in 10 min 1k samples in 21.3 s implies 28k samples

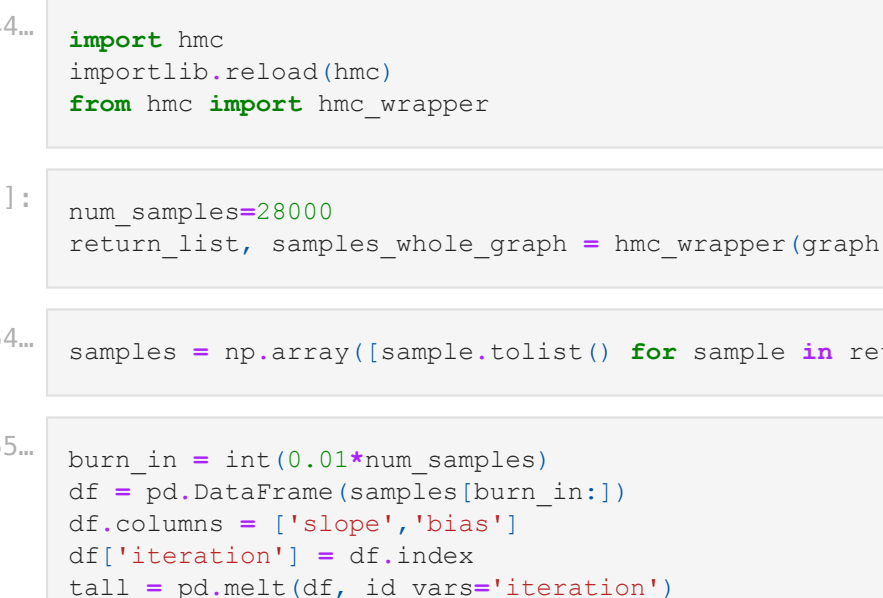
In [143…
```python
fname = '2.daphne'
graph = graph_helper(fname)
```

In [144…
```python
import hmc
importlib.reload(hmc)
from hmc import hmc_wrapper
```

In [ ]:
```python
num_samples=28000
return_list, samples_whole_graph = hmc_wrapper(graph,num_samples,T=20,epsilon=0.01)
```

In [154…
```python
samples = np.array([sample.tolist() for sample in return_list])
```

In [155…
```python
burn_in = int(0.01*num_samples)
df = pd.DataFrame(samples[burn_in:])
df.columns = ['slope','bias']
df['iteration'] = df.index
tall = pd.melt(df, id_vars='iteration')

g = sns.FacetGrid(tall, col="variable")
g.map(sns.histplot, "value")

plt.suptitle('HMC | {}'.format(fname))
plt.subplots_adjust(top=0.8)
```



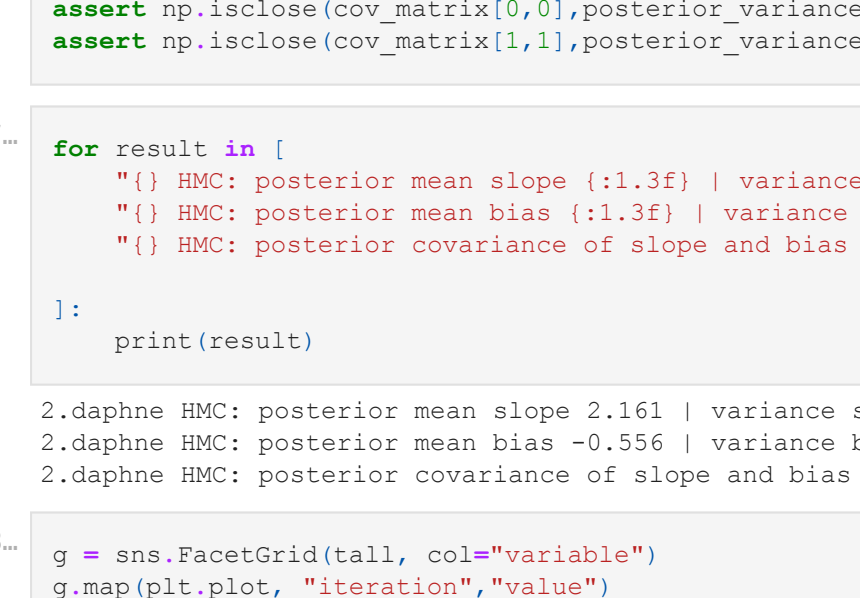
In [156…
```python
posterior_mean = samples[burn_in:].mean(0)
cov_matrix = np.cov(samples[burn_in:].T,ddof=0)
posterior_variance = samples[burn_in:].var(0)
covariance = cov_matrix[0,1]
assert np.isclose(cov_matrix[0,0],posterior_variance[0])
assert np.isclose(cov_matrix[1,1],posterior_variance[1])
```

In [157…
```python
for result in [
    "{} HMC: posterior mean slope {:1.3f} | variance slope {:1.3e}".format(fname,posterior_mean[0],posterior_va
    "{} HMC: posterior mean bias {:1.3f} | variance bias {:1.3e}".format(fname,posterior_mean[1],posterior_vari
    "{} HMC: posterior covariance of slope and bias variance bias {:1.3e}".format(fname,covariance),

]:
    print(result)
```

```
2.daphne HMC: posterior mean slope 2.161 | variance slope 5.652e-02
2.daphne HMC: posterior mean bias -0.556 | variance bias 8.609e-01
2.daphne HMC: posterior covariance of slope and bias variance bias -1.984e-01
```

In [158…
```python
g = sns.FacetGrid(tall, col="variable")
g.map(plt.plot, "iteration","value")

plt.suptitle('HMC | {}'.format(fname))
plt.subplots_adjust(top=0.75)
plt.suptitle('{} | HMC \n Sample trace'.format(fname))
```

Out[158…
```
Text(0.5, 0.98, '2.daphne | HMC \n Sample trace')
```

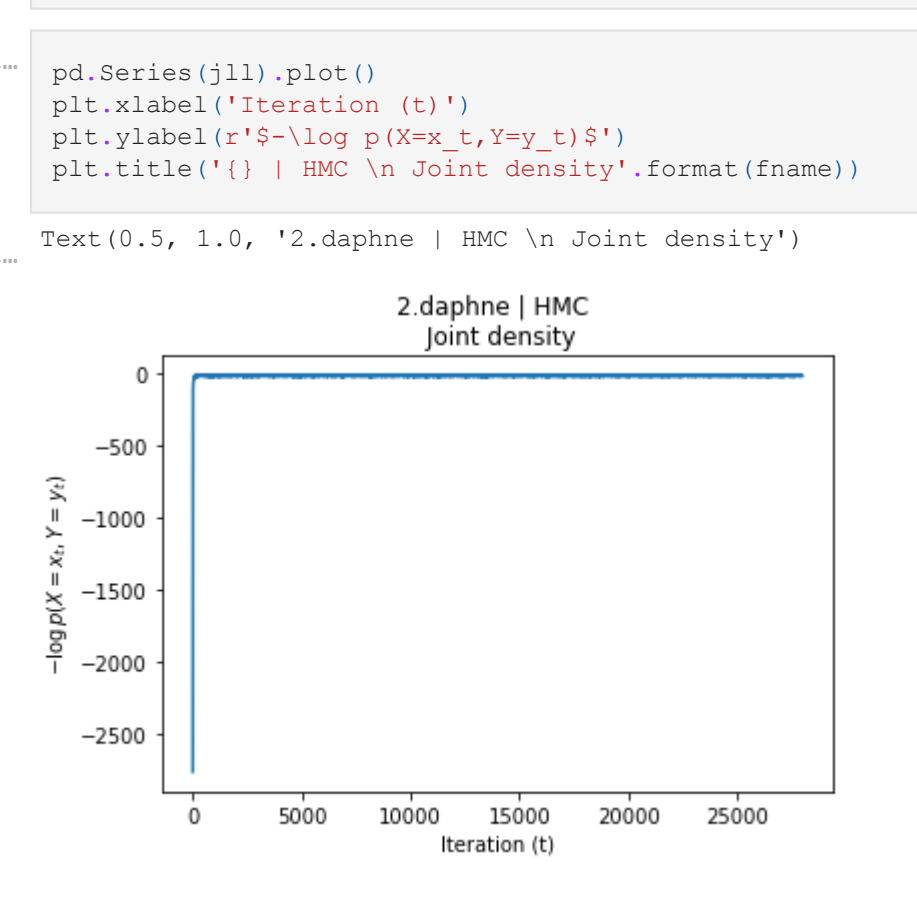


In [159…
```python
G = graph[1]
Y = G['Y']
Y = {key:evaluate([value])[0] for key,value in Y.items()}
P = G['P']
```

In [160…
```python
size = len(samples_whole_graph)
jll = np.zeros(size)
for idx in range(size):
    jll[idx] = compute_log_joint_prob(samples_whole_graph[idx],Y,P)
```

In [161…
```python
pd.Series(jll).plot()
plt.xlabel('Iteration (t)')
plt.ylabel(r'$-\log p(X=x_t,Y=y_t)$')
plt.title('{} | HMC \n Joint density'.format(fname))
```

Out[161…
```
Text(0.5, 1.0, '2.daphne | HMC \n Joint density')
```



In [1]:
```python
import torch
import numpy as np
import os, json
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from evaluation_based_sampling import evaluate, evaluate_program
from daphne import daphne

from graph_based_sampling import sample_from_joint
```

Problem 3

In [2]:
```python
from load_helper import ast_helper, graph_helper
```

Importance sampling

- 10k samples in 1.59s implies 384k samples in 10 min
- 1k samples in 5.29s implies 113k samples in 10 min

In [3]:
```python
import parse
import importance_sampling
```

In [4]:
```python
fname = '3.daphne'
ast = ast_helper(fname)
ast
```

Out[4]:
```
[['let',
  ['data', ['vector', 1.1, 2.1, 2.0, 1.9, 0.0, -0.1, -0.05]],
  ['let',
   ['likes',
    ['vector',
     ['let',
      ['mu', ['sample', ['normal', 0.0, 10.0]]],
      ['let',
       ['sigma', ['sample', ['gamma', 1.0, 1.0]]],
       ['normal', 'mu', 'sigma']]],
     ['let',
      ['mu', ['sample', ['normal', 0.0, 10.0]]],
      ['let',
       ['sigma', ['sample', ['gamma', 1.0, 1.0]]],
       ['normal', 'mu', 'sigma']]],
     ['let',
      ['mu', ['sample', ['normal', 0.0, 10.0]]],
      ['let',
       ['sigma', ['sample', ['gamma', 1.0, 1.0]]],
       ['normal', 'mu', 'sigma']]]]],
   ['let',
    ['pi', ['sample', ['dirichlet', ['vector', 1.0, 1.0, 1.0]]]],
    ['let',
     ['z-prior', ['discrete', 'pi']],
     ['let',
      ['z',
       ['vector',
        ['let',
         ['y', ['get', 'data', 0]],
         ['let',
          ['z', ['sample', 'z-prior']],
          ['let', ['_', ['observe', ['get', 'likes', 'z'], 'y']], 'z']]],
        ['let',
         ['y', ['get', 'data', 1]],
         ['let',
          ['z', ['sample', 'z-prior']],
          ['let', ['_', ['observe', ['get', 'likes', 'z'], 'y']], 'z']]],
        ['let',
         ['y', ['get', 'data', 2]],
         ['let',
          ['z', ['sample', 'z-prior']],
          ['let', ['_', ['observe', ['get', 'likes', 'z'], 'y']], 'z']]],
        ['let',
         ['y', ['get', 'data', 3]],
         ['let',
          ['z', ['sample', 'z-prior']],
          ['let', ['_', ['observe', ['get', 'likes', 'z'], 'y']], 'z']]],
        ['let',
         ['y', ['get', 'data', 4]],
         ['let',
          ['z', ['sample', 'z-prior']],
          ['let', ['_', ['observe', ['get', 'likes', 'z'], 'y']], 'z']]],
        ['let',
         ['y', ['get', 'data', 5]],
         ['let',
          ['z', ['sample', 'z-prior']],
          ['let', ['_', ['observe', ['get', 'likes', 'z'], 'y']], 'z']]],
        ['let',
         ['y', ['get', 'data', 6]],
         ['let',
          ['z', ['sample', 'z-prior']],
          ['let', ['_', ['observe', ['get', 'likes', 'z'], 'y']], 'z']]]]],
      ['=', ['first', 'z'], ['second', 'z']]]]]]]]
```

In [36]:
```python
%%time
num_samples=113000
samples, sigmas = parse.take_samples(num_samples,ast=ast)
```

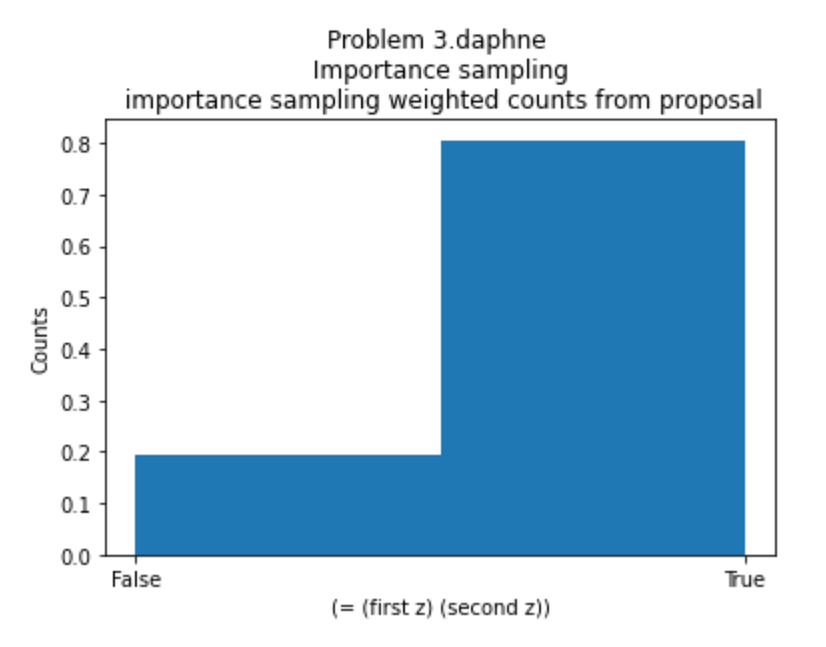
```
CPU times: user 10min 7s, sys: 1.91 s, total: 10min 9s
Wall time: 10min 11s
```

In [37]:
```python
samples = np.array([sample.item() for sample in samples])
```

In [38]:
```python
posterior_mean, probs = importance_sampling.weighted_average(samples,sigmas)
```

In [39]:
```python
_ = plt.hist(samples.astype(int), weights=probs, bins=2)
plt.xlabel('(= (first z) (second z))')
plt.title('Problem {} \n Importance sampling \n importance sampling weighted counts from proposal'.format(fname
plt.xticks([0,1],["False","True"])
plt.ylabel('Counts')
```

Out[39]:
```
Text(0, 0.5, 'Counts')
```



In [40]:
```python
method = "Importance sampling"

""" {}: The posterior probability that the first and second datapoint are in the same cluster,\
i.e. the posterior probability that z[1] == z[2]: {:0.3f}\
""".format(method,posterior_mean)
```

Out[40]:
```
' Importance sampling: The posterior probability that the first and second datapoint are in the same cluster,i.
e. the posterior probability that z[1] == z[2]: 0.806'
```

MH Gibbs

0.27 s / sample implies ~2.2k samples in 10 min 0.1k in 30.3s implies 1.98k in 10 min

My code is slow, perhaps because of some internal logging and so the burn in needs to be quite large compared with the total run. But the relatively long burn in is justified by the joint increasing.

In [10]:
```python
import graph_based_sampling
import mh_gibbs
from hmc import compute_log_joint_prob
```

In [11]:
```python
fname = '3.daphne'
graph = graph_helper(fname)
graph
```

Out[11]:
```
[{},
 {'V': ['sample5',
   'sample0',
   'observe18',
   'observe12',
   'sample17',
   'sample7',
   'sample9',
   'sample15',
   'sample19',
   'sample6',
   'observe14',
   'observe10',
   'sample13',
   'sample4',
   'sample2',
   'sample1',
   'observe8',
   'sample3',
   'sample11',
   'observe16',
   'observe20'],
  'A': {'sample5': ['observe18',
    'observe12',
    'observe14',
    'observe10',
    'observe8',
    'observe16',
    'observe20'],
   'sample0': ['observe18',
    'observe12',
    'observe14',
    'observe10',
    'observe8',
    'observe16',
    'observe20'],
   'sample17': ['observe18'],
   'sample7': ['observe8'],
   'sample9': ['observe10'],
   'sample15': ['observe16'],
   'sample19': ['observe20'],
   'sample6': ['sample17',
    'sample7',
    'sample9',
    'sample15',
    'sample19',
    'sample13',
    'sample11'],
   'sample13': ['observe14'],
   'sample4': ['observe18',
    'observe12',
    'observe14',
    'observe10',
    'observe8',
    'observe16',
    'observe20'],
   'sample2': ['observe18',
    'observe12',
    'observe14',
    'observe10',
    'observe8',
    'observe16',
    'observe20'],
   'sample1': ['observe18',
    'observe12',
    'observe14',
    'observe10',
    'observe8',
    'observe16',
    'observe20'],
   'sample3': ['observe18',
    'observe12',
    'observe14',
    'observe10',
    'observe8',
    'observe16',
    'observe20'],
   'sample11': ['observe12']},
  'P': {'sample5': ['sample*', ['gamma', 1.0, 1.0]],
   'sample0': ['sample*', ['normal', 0.0, 10.0]],
   'observe18': ['observe*',
    ['get',
     ['vector',
      ['normal', 'sample0', 'sample1'],
      ['normal', 'sample2', 'sample3'],
      ['normal', 'sample4', 'sample5']],
     'sample17'],
    -0.1],
   'observe12': ['observe*',
    ['get',
     ['vector',
      ['normal', 'sample0', 'sample1'],
      ['normal', 'sample2', 'sample3'],
      ['normal', 'sample4', 'sample5']],
     'sample11'],
    2.0],
   'sample17': ['sample*', ['discrete', 'sample6']],
   'sample7': ['sample*', ['discrete', 'sample6']],
   'sample9': ['sample*', ['discrete', 'sample6']],
   'sample15': ['sample*', ['discrete', 'sample6']],
   'sample19': ['sample*', ['discrete', 'sample6']],
   'sample6': ['sample*', ['dirichlet', ['vector', 1.0, 1.0, 1.0]]],
   'observe14': ['observe*',
    ['get',
     ['vector',
      ['normal', 'sample0', 'sample1'],
      ['normal', 'sample2', 'sample3'],
      ['normal', 'sample4', 'sample5']],
     'sample13'],
    1.9],
   'observe10': ['observe*',
    ['get',
     ['vector',
      ['normal', 'sample0', 'sample1'],
      ['normal', 'sample2', 'sample3'],
      ['normal', 'sample4', 'sample5']],
     'sample9'],
    2.1],
   'sample13': ['sample*', ['discrete', 'sample6']],
   'sample4': ['sample*', ['normal', 0.0, 10.0]],
   'sample2': ['sample*', ['normal', 0.0, 10.0]],
   'sample1': ['sample*', ['gamma', 1.0, 1.0]],
   'observe8': ['observe*',
    ['get',
     ['vector',
      ['normal', 'sample0', 'sample1'],
      ['normal', 'sample2', 'sample3'],
      ['normal', 'sample4', 'sample5']],
     'sample7'],
    1.1],
   'sample3': ['sample*', ['gamma', 1.0, 1.0]],
   'sample11': ['sample*', ['discrete', 'sample6']],
   'observe16': ['observe*',
    ['get',
     ['vector',
      ['normal', 'sample0', 'sample1'],
      ['normal', 'sample2', 'sample3'],
      ['normal', 'sample4', 'sample5']],
     'sample15'],
    0.0],
   'observe20': ['observe*',
    ['get',
     ['vector',
      ['normal', 'sample0', 'sample1'],
      ['normal', 'sample2', 'sample3'],
      ['normal', 'sample4', 'sample5']],
     'sample19'],
    -0.05]},
  'Y': {'observe8': 1.1,
   'observe10': 2.1,
   'observe12': 2.0,
   'observe14': 1.9,
   'observe16': 0.0,
   'observe18': -0.1,
   'observe20': -0.05}},
 ['=', 'sample7', 'sample9']]
```

In [ ]:
```python
num_steps=1980
return_list, samples_whole_graph = mh_gibbs.mh_gibbs_wrapper(graph,num_steps)
```

In [44]:
```python
samples = np.array([sample.item() for sample in return_list])
```

In [54]:
```python
burn_in = 1500#int(0.01*num_steps)
pd.Series(samples[burn_in:]).astype(float).plot.hist()
plt.xlabel('(= (first z) (second z))')
plt.title('MH Gibbs | {}'.format(fname))
```

Out[54]:
```
Text(0.5, 1.0, 'MH Gibbs | 3.daphne')
```

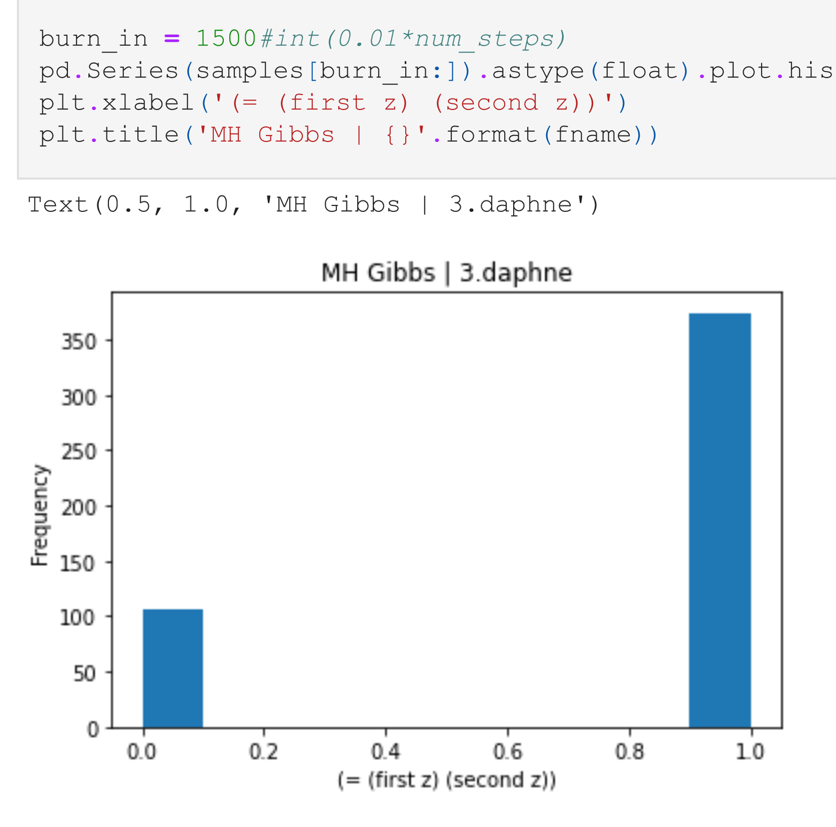
In [55]:
```python
posterior_mean = samples[burn_in:].mean(0)
```

In [56]:
```python
method = "MH Gibbs"

""" {}: The posterior probability that the first and second datapoint are in the same cluster,\
i.e. the posterior probability that z[1] == z[2]: {:0.3f}\
""".format(method,posterior_mean)
```

Out[56]:
```
' MH Gibbs: The posterior probability that the first and second datapoint are in the same cluster,i.e. the post
erior probability that z[1] == z[2]: 0.778'
```

In [17]:
```python
pd.Series(samples).astype(int).plot()
plt.xlabel('Iteration')
plt.ylabel('z[1] == z[2]')
plt.title('{} | MH Gibbs \n Sample trace'.format(fname))
```

Out[17]:
```
Text(0.5, 1.0, '3.daphne | MH Gibbs \n Sample trace')
```

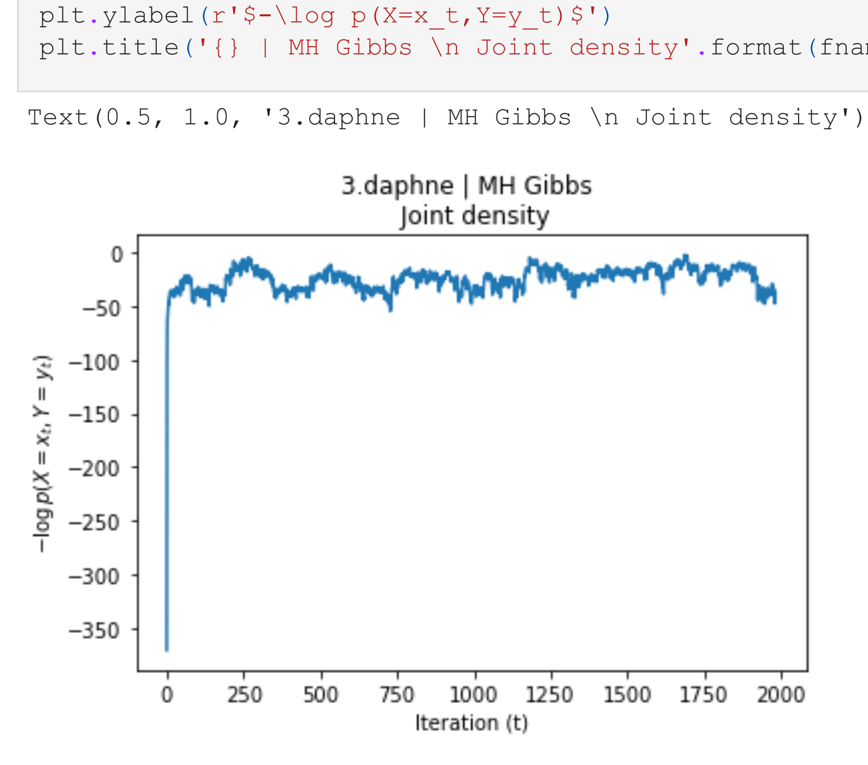


In [18]:
```python
G = graph[1]
Y = G['Y']
Y = {key:evaluate([value])[0] for key,value in Y.items()}
P = G['P']
```

In [19]:
```python
size = len(samples_whole_graph)
jll = np.zeros(size)
for idx in range(size):
    jll[idx] = compute_log_joint_prob(samples_whole_graph[idx],Y,P)
```

In [20]:
```python
pd.Series(jll).plot()
plt.xlabel('Iteration (t)')
plt.ylabel(r'$-\log p(X=x_t,Y=y_t)$')
plt.title('{} | MH Gibbs \n Joint density'.format(fname))
```

Out[20]:
```
Text(0.5, 1.0, '3.daphne | MH Gibbs \n Joint density')
```



In [1]:
```
import torch
import numpy as np
import os, json
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from evaluation_based_sampling import evaluate, evaluate_program
from daphne import daphne
```

Problem 4

In [2]:
```
from load_helper import ast_helper, graph_helper
```

Importance sampling

- 10k samples in 6.82s implies 879k samples in 10 min

In [3]:
```
import parse
import importance_sampling
```

In [4]:
```
fname = '4.daphne'
ast = ast_helper(fname)
ast
```

Out[4]:
```
[['let',
  ['sprinkler', True],
  ['let',
   ['wet-grass', True],
   ['let',
    ['is-cloudy', ['sample', ['flip', 0.5]]],
    ['let',
     ['is-raining',
      ['if',
       ['=', 'is-cloudy', True],
       ['sample', ['flip', 0.8]],
       ['sample', ['flip', 0.2]]]],
     ['let',
      ['sprinkler-dist',
       ['if', ['=', 'is-cloudy', True], ['flip', 0.1], ['flip', 0.5]]],
      ['let',
       ['wet-grass-dist',
        ['if',
         ['and', ['=', 'sprinkler', True], ['=', 'is-raining', True]],
         ['flip', 0.99],
         ['if',
          ['and', ['=', 'sprinkler', False], ['=', 'is-raining', False]],
          ['flip', 0.0],
          ['if',
           ['or', ['=', 'sprinkler', True], ['=', 'is-raining', True]],
           ['flip', 0.9],
           None]]]],
       ['let',
        ['dontcare0', ['observe', 'sprinkler-dist', 'sprinkler']],
        ['let',
         ['dontcare1', ['observe', 'wet-grass-dist', 'wet-grass']],
         'is-raining']]]]]]]]
```

In [26]:
```
%%time
num_samples=879000
samples,sigmas = parse.take_samples(num_samples,ast=ast)
```

```
CPU times: user 10min 22s, sys: 2.84 s, total: 10min 25s
Wall time: 10min 27s
```

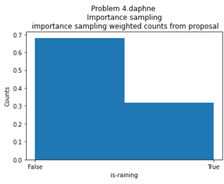
In [27]:
```
samples = np.array([sample.item() for sample in samples])
```

In [28]:
```
posterior_mean, probs = importance_sampling.weighted_average(samples,sigmas)
posterior_mean
```

Out[28]:
```
0.31957416455608195
```

In [29]:
```
_ = plt.hist(samples.astype(int), weights=probs, bins=2)
plt.xlabel('is-raining')
plt.title('Problem {} \n Importance sampling \n importance sampling weighted counts from proposal'.format(fname
plt.xticks([0,1],["False","True"])
plt.ylabel('Counts')
```

Out[29]:
```
Text(0, 0.5, 'Counts')
```



In [30]:
```
"""The posterior probability that it is raining, i.e. of "is-raining.": {:0.3f}""".format(posterior_mean)
```

Out[30]:
```
'The posterior probability that it is raining, i.e. of "is-raining.": 0.320'
```

MH Gibbs

2k samples in 8.55s implies 140k samples in 10 min

In [10]:
```
import graph_based_sampling
import mh_gibbs
from hmc import compute_log_joint_prob
```

In [11]:
```
fname = '4.daphne'
graph = graph_helper(fname)
graph
```

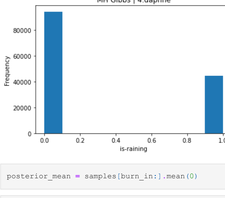
Out[11]:
```
[{},
 {'V': ['observe6', 'sample4', 'sample2', 'observe5', 'sample3'],
  'A': {'sample2': ['observe6', 'observe5'],
   'sample4': ['observe6'],
   'sample3': ['observe6']},
  'P': {'sample2': ['sample*', ['flip', 0.5]],
   'sample3': ['sample*', ['flip', 0.8]],
   'sample4': ['sample*', ['flip', 0.2]],
   'observe5': ['observe*',
    ['if', ['=', 'sample2', True], ['flip', 0.1], ['flip', 0.5]],
    True],
   'observe6': ['observe*',
    ['if',
     ['and',
      True,
      ['=', ['if', ['=', 'sample2', True], 'sample3', 'sample4'], True]],
     ['flip', 0.99],
     ['if',
      ['and',
       False,
       ['=', ['if', ['=', 'sample2', True], 'sample3', 'sample4'], False]],
      ['flip', 0.0],
      ['if',
       ['or',
        True,
        ['=', ['if', ['=', 'sample2', True], 'sample3', 'sample4'], True]],
       ['flip', 0.9],
       None]]],
    True]},
  'Y': {'observe5': True, 'observe6': True}},
 ['if', ['=', 'sample2', True], 'sample3', 'sample4']]
```

In [ ]:
```
num_steps=140000
return_list, samples_whole_graph = mh_gibbs.mh_gibbs_wrapper(graph,num_steps)
```

In [13]:
```
samples = np.array([sample.item() for sample in return_list])
```

In [14]:
```
burn_in = int(0.01*num_steps)
pd.Series(samples[burn_in:]).astype(float).plot.hist()
plt.xlabel('is-raining')
plt.title('MH Gibbs | {}'.format(fname))
```

Out[14]:
```
Text(0.5, 1.0, 'MH Gibbs | 4.daphne')
```



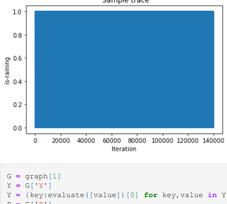
In [15]:
```
posterior_mean = samples[burn_in:].mean(0)
```

In [16]:
```
"""The posterior probability that it is raining, i.e. of "is-raining.": {:0.3f}""".format(posterior_mean)
```

Out[16]:
```
'The posterior probability that it is raining, i.e. of "is-raining.": 0.321'
```

In [17]:
```
pd.Series(samples).astype(int).plot()
plt.xlabel('Iteration')
plt.ylabel('is-raining')
plt.title('{} | MH Gibbs \n Sample trace'.format(fname))
```

Out[17]:
```
Text(0.5, 1.0, '4.daphne | MH Gibbs \n Sample trace')
```

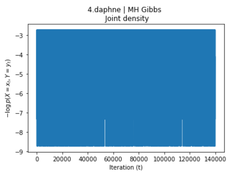


In [18]:
```
G = graph[1]
Y = G['Y']
Y = {key:evaluate([value])[0] for key,value in Y.items()}
P = G['P']
```

In [19]:
```
size = len(samples_whole_graph)
jll = np.zeros(size)
for idx in range(size):
    jll[idx] = compute_log_joint_prob(samples_whole_graph[idx],Y,P)
```

In [34]:
```
pd.Series(jll).plot()
plt.xlabel('Iteration (t)')
plt.ylabel(r'$-\log p(X=x_t,Y=y_t)$')
plt.title('{} | MH Gibbs \n Joint density'.format(fname))
```

Out[34]:
```
Text(0.5, 1.0, '4.daphne | MH Gibbs \n Joint density')
```



In [1]:
```
import torch
import numpy as np
import os, json
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import importlib

from evaluation_based_sampling import evaluate, evaluate_program
from daphne import daphne

from graph_based_sampling import sample_from_joint
```

Problem 5

I approach this problem with approximate Baysean computation. Relaxing the dirac into a normal. This is inspired by the definition of the dirac as a normal pdf in the limit of the variance going to zero.

- Khuri, A. I. (2004). Applications of Dirac's delta function in statistics. International Journal of Mathematical Education in Science and Technology, 35(2), 185–195. http://doi.org/10.1080/00207390310001638313

I set the variance of the normal to 0.1^2 more detailed study could be done showing the behaviour as a function of the variance. At $\sigma^2 <= 0.03^2$ I encoutered errors in HMC, likely caused by things going to infinity (log_prob scoring was out of distribution, and it is normal... ?)

Comments on ABC for this problem

There are really only one degree of freedom for this problem, since when one value is fixed, the other must be seven minus it, to be compatible with the observe statement `(observe (dirac (+ x y)) 7)`. So there is only one variance, and no covariance. In the ABC approximation however, we only strictly enforce that $x+y=7$ and thus we get a covariance of x and y.

Any $x+y=7$ is compatible with the observe, and this "manifold" is the line. The ABC makes this line have non-zero measure.

Another way to do this problem, would be to incorporate the observe into the program, and replace y with 7-x somehow to enforce the constraint.

Comparison of IS, MH Gibbs & HMC

- IS. has high variance, but gets reasonable means of near 3.5
- Gibbs is still and so can't update very well. It is exploring along a very narrow ridge, instead of along the line. x and y should be updated together as a block. The problem is very stiff (the dicar makes it initifely stiff).
- HMC has variance near IS. We can see from the joint that it is converging with iterations gradually.

Error trace

for normal that is too narrow (intentionally put below)

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/var/folders/bg/cb0cr7ls61352lhy50167r0c0000gn/T/ipykernel_58709/4110678344.py in <module>
      1 num_samples=28000//10
----> 2 return_list, samples_whole_graph = hmc_wrapper(graph,num_samples,T=40,epsilon=0.1)

~/repos/prob_prog/hw/hw3/hmc.py in hmc_wrapper(graph, num_samples, T, epsilon, M)
     33         # include kinetic and potential energy functions
     34         # MC acceptance criteria
---> 35     samples_whole_graph = hmc_algo20(X0,num_samples,T,epsilon,M,Y,P,X_vertex_names_to_idx_d)
     36 
     37     # evaluate samples (on whatever function, here the return of the program) as needed

~/repos/prob_prog/hw/hw3/hmc.py in hmc_algo20(X0, num_samples, T, epsilon, M, Y, P,
X_vertex_names_to_idx_d)
     56     for s in range(num_samples):
     57         R_s = normal_R_reuse.sample()
---> 58         R_p, X_p =
leapfrog(copy.deepcopy(X_s),copy.deepcopy(R_s),T,epsilon,Y,P,X_vertex_names_to_idx_d)
     59         # X_p, R_p = leapfrog(X_s,R_s,T,epsilon,X_vertex_names_to_idx_d)
     60             # copy X_s?

~/repos/prob_prog/hw/hw3/hmc.py in leapfrog(X0, R0, T, epsilon, Y, P, X_vertex_names_to_idx_d)
     83         # TODO: save all in loop instead of overwriting to visualize
     84         X_t = add_dict_to_tensor(X_t,epsilon*R_t,X_vertex_names_to_idx_d)
---> 85         R_t = R_t - epsilon*grad_U(X_t,Y,P,X_vertex_names_to_idx_d)
     86     X_T = add_dict_to_tensor(X_t,epsilon*R_t,X_vertex_names_to_idx_d)
     87     R_T = R_t - epsilon_2*grad_U(X_T,Y,P,X_vertex_names_to_idx_d)

~/repos/prob_prog/hw/hw3/hmc.py in grad_U(X, Y, P, X_vertex_names_to_idx_d)
    112     return vector of gradients
    113     """
--> 114     energy_U = compute_U(X,Y,P)
    115 
    116     # Zero the gradients.

~/repos/prob_prog/hw/hw3/hmc.py in compute_U(X, Y, P)
    177 
    178 def compute_U(X,Y,P):
--> 179     energy_U = -compute_log_joint_prob(X,Y,P)
    180     return energy_U
    181 

~/repos/prob_prog/hw/hw3/hmc.py in compute_log_joint_prob(X, Y, P)
    167         e = P[X_vertex][1]
    168         distribution = evaluate(e,local_env=X)[0]
--> 169         log_prob += score(distribution,X[X_vertex])
    170     for Y_vertex in Y.keys():
    171         e = P[Y_vertex][1]

~/repos/prob_prog/hw/hw3/evaluation_based_sampling.py in score(distribution, c)
     55         log_w = distribution.log_prob(c.double())
     56     else:
---> 57         log_w = distribution.log_prob(c)
     58     return log_w
     59 

~/miniconda2/envs/prob_prog/lib/python3.9/site-packages/torch/distributions/normal.py in
log_prob(self, value)
     71     def log_prob(self, value):
     72         if self._validate_args:
---> 73             self._validate_sample(value)
     74         # compute the variance
     75         var = (self.scale ** 2)

~/miniconda2/envs/prob_prog/lib/python3.9/site-packages/torch/distributions/distribution.py in
_validate_sample(self, value)
    275         assert support is not None
    276         if not support.check(value).all():
--> 277             raise ValueError('The value argument must be within the support')
    278 
    279     def _get_checked_instance(self, cls, _instance=None):

ValueError: The value argument must be within the support
```

In [2]:
```
from load_helper import ast_helper, graph_helper
```

Importance sampling

30k in 10s implies 1.698 million

In [3]:
```
import parse
import importance_sampling
import importlib
importlib.reload(parse)
```

Out[3]:
```
<module 'parse' from '/Users/gw/repos/prob_prog/hw/hw3/parse.py'>
```

In [4]:
```
fname = '5_abc.daphne'
ast = ast_helper(fname)
ast
```

Out[4]:
```
[['let',
  ['x', ['sample', ['normal', 0, 10]]],
  ['let',
   ['y', ['sample', ['normal', 0, 10]]],
   ['let',
    ['dontcare0', ['observe', ['normal', ['+', 'x', 'y'], 0.09], 7]],
    ['vector', 'x', 'y']]]]]
```

In [5]:
```
%%time
num_samples=int(1.698e6)
samples, sigmas = parse.take_samples(num_samples,ast=ast)
samples = np.array([sample.tolist() for sample in samples])
```

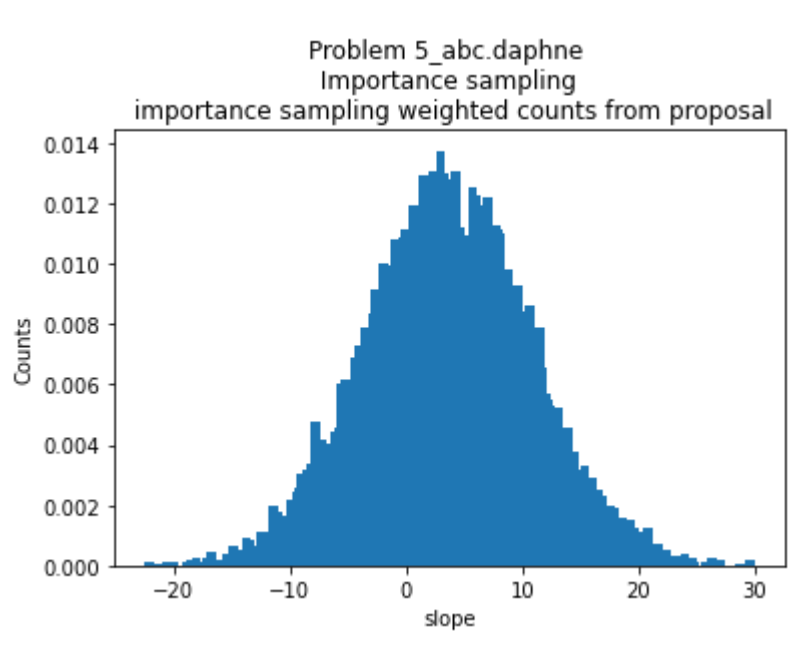
```
CPU times: user 9min 37s, sys: 2.48 s, total: 9min 40s
Wall time: 9min 40s
```

In [6]:
```
posterior_mean, probs = importance_sampling.weighted_average(samples,sigmas,reshape_probs=(-1,1),axis=0)
posterior_mean
```

Out[6]:
```
array([3.53386895, 3.46580443])
```

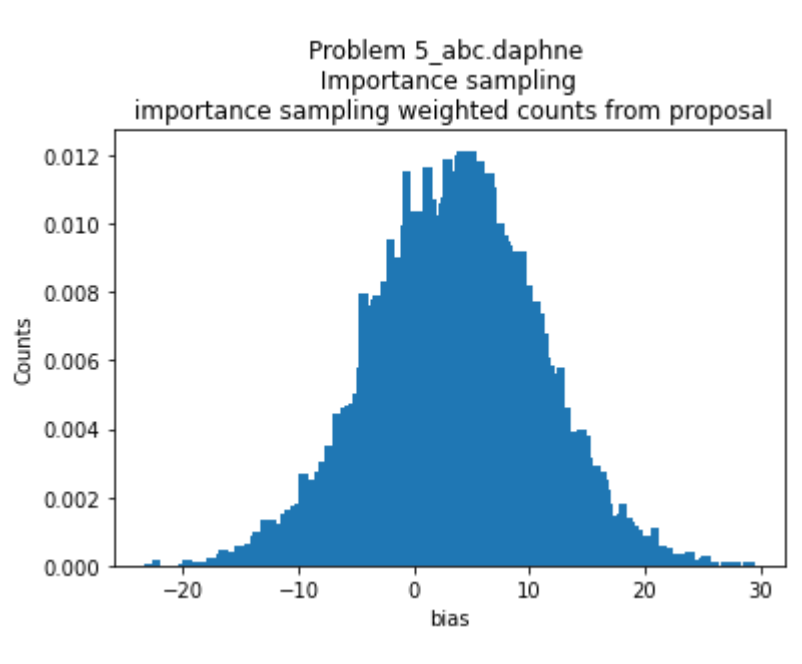
In [7]:
```
counts_bins = np.histogram(samples[:,0], weights=probs[:,0], bins=500)
counts, bins = counts_bins[0], counts_bins[1]
idx = (counts > counts.max()*0.005)
plt.bar(bins[1:][idx],counts[idx])
plt.title('Problem {} \n Importance sampling \n importance sampling weighted counts from proposal'.format(fname
plt.ylabel('Counts')
plt.xlabel('slope')
```

Out[7]:
```
Text(0.5, 0, 'slope')
```



In [8]:
```
counts_bins = np.histogram(samples[:,1], weights=probs[:,0], bins=500)
counts, bins = counts_bins[0], counts_bins[1]
idx = (counts > counts.max()*0.005)
plt.bar(bins[1:][idx],counts[idx])
plt.title('Problem {} \n Importance sampling \n importance sampling weighted counts from proposal'.format(fname
plt.ylabel('Counts')
plt.xlabel('bias')
```

Out[8]:
```
Text(0.5, 0, 'bias')
```



In [9]:
```
expectation_samples_2, probs = importance_sampling.weighted_average(samples**2,sigmas,reshape_probs=(-1,1),axis
posterior_variance = expectation_samples_2 - posterior_mean**2
posterior_variance
```

Out[9]:
```
array([49.98343282, 49.98004001])
```

In [10]:
```
expectation_samplex_sampley, probs = importance_sampling.weighted_average(samples[:,0]*samples[:,1],sigmas)
covariance = expectation_samplex_sampley - posterior_mean[0]*posterior_mean[1]
covariance
```

Out[10]:
```
-49.9777814630405
```

In [11]:
```
for result in [
    "{} Importance sampling: posterior mean slope {:1.3f} | variance slope {:1.3e}".format(fname,posterior_mear
    "{} Importance sampling: posterior mean bias {:1.3f} | variance bias {:1.3e}".format(fname,posterior_mean[1
    "{} Importance sampling: posterior covariance of slope and bias variance bias {:1.3e}".format(fname,covaria

]:
    print(result)
```

```
5_abc.daphne Importance sampling: posterior mean slope 3.534 | variance slope 4.998e+01
5_abc.daphne Importance sampling: posterior mean bias 3.466 | variance bias 4.998e+01
5_abc.daphne Importance sampling: posterior covariance of slope and bias variance bias -4.998e+01
```

Numpy contains it's own method for computing this, and we can check it agrees with our results (where things are spelt out a bit more for learning purposes).

In [12]:
```
np.cov(samples.T,aweights=probs,ddof=0)
```

Out[12]:
```
array([[ 49.98343282, -49.97778146],
       [-49.97778146,  49.98004001]])
```

MH Gibbs

14k in 22.8s implies 368k in 10 min

In [13]:
```
import mh_gibbs
from hmc import hmc_wrapper, compute_log_joint_prob
importlib.reload(mh_gibbs)
```

Out[13]:
```
<module 'mh_gibbs' from '/Users/gw/repos/prob_prog/hw/hw3/mh_gibbs.py'>
```

In [14]:
```
fname = '5_abc.daphne'
graph = graph_helper(fname)
graph
```

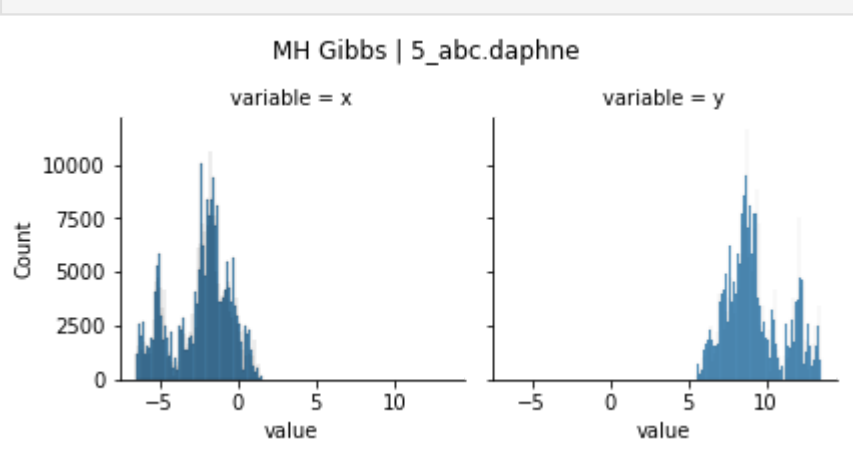
Out[14]:
```
[{},
 {'V': ['observe3', 'sample2', 'sample1'],
  'A': {'sample2': ['observe3'], 'sample1': ['observe3']},
  'P': {'sample1': ['sample*', ['normal', 0, 10]],
   'sample2': ['sample*', ['normal', 0, 10]],
   'observe3': ['observe*', ['normal', ['+', 'sample1', 'sample2'], 0.09], 7]},
  'Y': {'observe3': 7}},
 ['vector', 'sample1', 'sample2']]
```

In [15]:
```
%%time
num_steps=368000
return_list, samples_whole_graph = mh_gibbs.mh_gibbs_wrapper(graph,num_steps)
samples = np.array([sample.tolist() for sample in return_list])
```

```
CPU times: user 9min 48s, sys: 1.51 s, total: 9min 49s
Wall time: 9min 50s
```

In [16]:
```
samples = np.array([sample.tolist() for sample in return_list])
```

In [17]:
```
burn_in = int(0.01*num_steps)
df = pd.DataFrame(samples[burn_in:])
df.columns = ['x','y']
df['iteration'] = df.index
tall = pd.melt(df, id_vars='iteration')

g = sns.FacetGrid(tall, col="variable")
g.map(sns.histplot, "value")

plt.suptitle('MH Gibbs | {}'.format(fname))
plt.subplots_adjust(top=0.8)
```



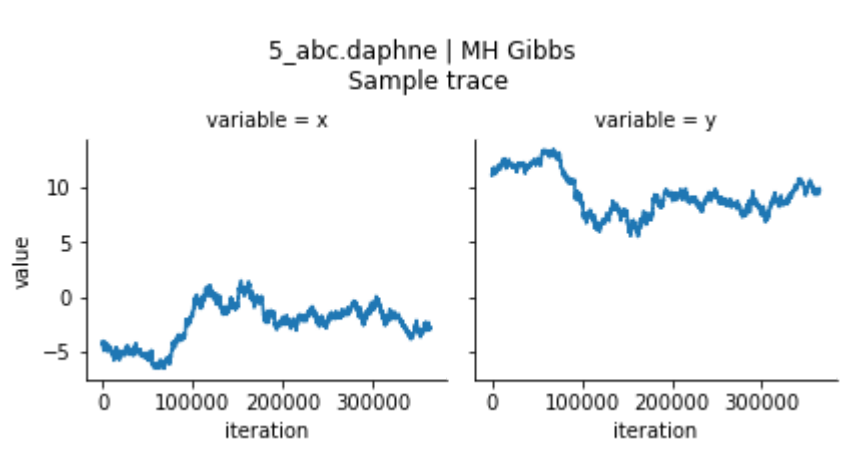
In [18]:
```
posterior_mean = samples[burn_in:].mean(0)
cov_matrix = np.cov(samples[burn_in:].T,ddof=0)
posterior_variance = samples[burn_in:].var(0)
covariance = cov_matrix[0,1]
assert np.isclose(cov_matrix[0,0],posterior_variance[0])
assert np.isclose(cov_matrix[1,1],posterior_variance[1])
```

In [19]:
```
for result in [
    "{} MH Gibbs: posterior mean slope {:1.3f} | variance slope {:1.3e}".format(fname,posterior_mean[0],posteri
    "{} MH Gibbs: posterior mean bias {:1.3f} | variance bias {:1.3e}".format(fname,posterior_mean[1],posterior
    "{} MH Gibbs: posterior covariance of slope and bias variance bias {:1.3e}".format(fname,covariance),

]:
    print(result)
```

```
5_abc.daphne MH Gibbs: posterior mean slope -2.343 | variance slope 3.465e+00
5_abc.daphne MH Gibbs: posterior mean bias 9.342 | variance bias 3.465e+00
5_abc.daphne MH Gibbs: posterior covariance of slope and bias variance bias -3.461e+00
```

In [20]:
```
g = sns.FacetGrid(tall, col="variable")
g.map(plt.plot, "iteration","value")

plt.suptitle('MH Gibbs | {}'.format(fname))
plt.subplots_adjust(top=0.75)
plt.suptitle('{} | MH Gibbs \n Sample trace'.format(fname))
```

Out[20]:
```
Text(0.5, 0.98, '5_abc.daphne | MH Gibbs \n Sample trace')
```

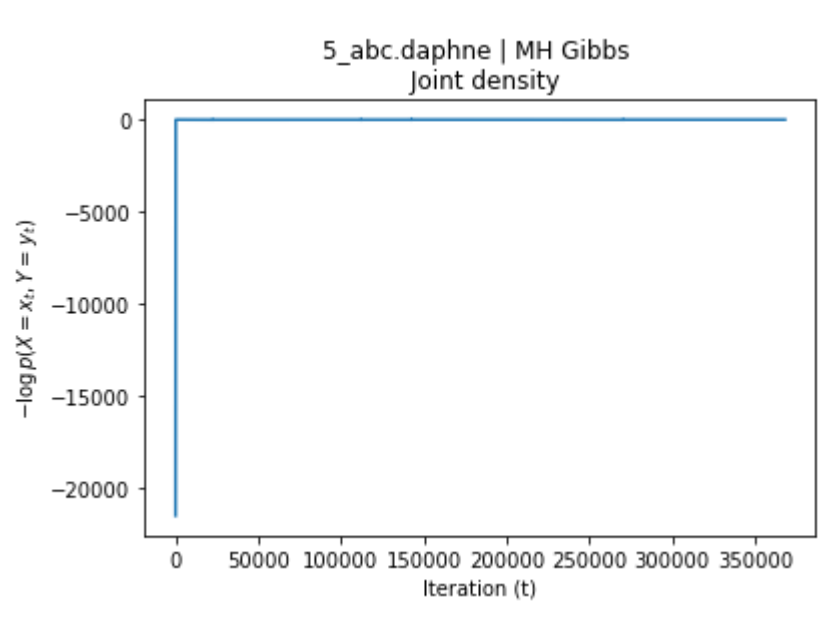


In [21]:
```
G = graph[1]
Y = G['Y']
Y = {key:evaluate([value])[0] for key,value in Y.items()}
P = G['P']
```

In [22]:
```
size = len(samples_whole_graph)
jll = np.zeros(size)
for idx in range(size):
    jll[idx] = compute_log_joint_prob(samples_whole_graph[idx],Y,P)
```

In [23]:
```
pd.Series(jll).plot()
plt.xlabel('Iteration (t)')
plt.ylabel(r'$-\log p(X=x_t,Y=y_t)$')
plt.title('{} | MH Gibbs \n Joint density'.format(fname))
```

Out[23]:
```
Text(0.5, 1.0, '5_abc.daphne | MH Gibbs \n Joint density')
```



HMC

- 0.56k in 16.2s implies 20.7k in 10 min

In [24]:
```
fname = '5_abc.daphne'
graph = graph_helper(fname)
graph
```

Out[24]:
```
[{},
 {'V': ['observe3', 'sample2', 'sample1'],
  'A': {'sample2': ['observe3'], 'sample1': ['observe3']},
  'P': {'sample1': ['sample*', ['normal', 0, 10]],
   'sample2': ['sample*', ['normal', 0, 10]],
   'observe3': ['observe*', ['normal', ['+', 'sample1', 'sample2'], 0.09], 7]},
  'Y': {'observe3': 7}},
 ['vector', 'sample1', 'sample2']]
```

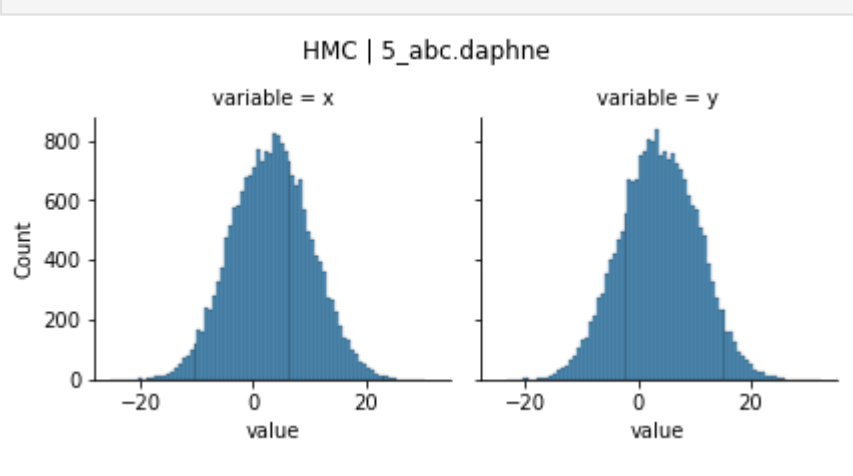
In [25]:
```
import hmc
importlib.reload(hmc)
from hmc import hmc_wrapper
```

In [26]:
```
%%time
num_samples=20700
return_list, samples_whole_graph = hmc_wrapper(graph,num_samples,T=40,epsilon=0.1)
```

```
CPU times: user 9min 40s, sys: 1.46 s, total: 9min 42s
Wall time: 9min 43s
```

In [27]:
```
samples = np.array([sample.tolist() for sample in return_list])
```

In [28]:
```
burn_in = int(0.01*num_samples)
df = pd.DataFrame(samples[burn_in:])
df.columns = ['x','y']
df['iteration'] = df.index
tall = pd.melt(df, id_vars='iteration')

g = sns.FacetGrid(tall, col="variable")
g.map(sns.histplot, "value")

plt.suptitle('HMC | {}'.format(fname))
plt.subplots_adjust(top=0.8)
```



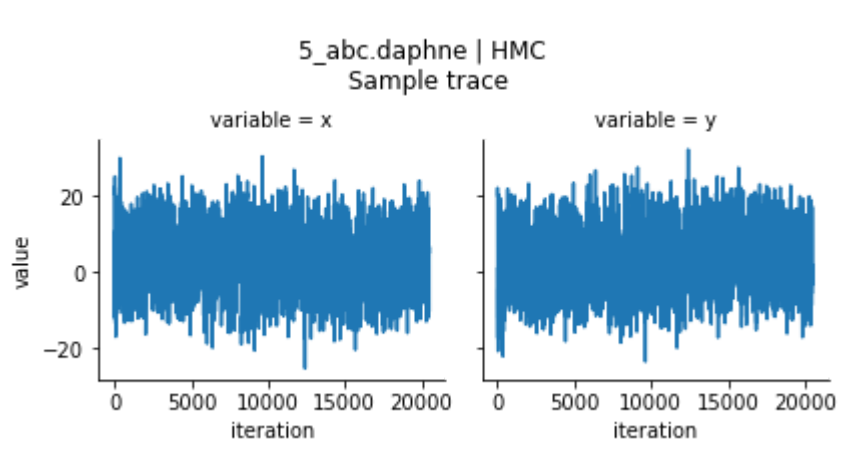
In [29]:
```
posterior_mean = samples[burn_in:].mean(0)
cov_matrix = np.cov(samples[burn_in:].T,ddof=0)
posterior_variance = samples[burn_in:].var(0)
covariance = cov_matrix[0,1]
assert np.isclose(cov_matrix[0,0],posterior_variance[0])
assert np.isclose(cov_matrix[1,1],posterior_variance[1])
```

In [30]:
```
for result in [
    "{} HMC: posterior mean slope {:1.3f} | variance slope {:1.3e}".format(fname,posterior_mean[0],posterior_va
    "{} HMC: posterior mean bias {:1.3f} | variance bias {:1.3e}".format(fname,posterior_mean[1],posterior_vari
    "{} HMC: posterior covariance of slope and bias variance bias {:1.3e}".format(fname,covariance),

]:
    print(result)
```

```
5_abc.daphne HMC: posterior mean slope 3.279 | variance slope 4.787e+01
5_abc.daphne HMC: posterior mean bias 3.721 | variance bias 4.788e+01
5_abc.daphne HMC: posterior covariance of slope and bias variance bias -4.739e+01
```

In [31]:
```
g = sns.FacetGrid(tall, col="variable")
g.map(plt.plot, "iteration","value")

plt.suptitle('HMC | {}'.format(fname))
plt.subplots_adjust(top=0.75)
plt.suptitle('{} | HMC \n Sample trace'.format(fname))
```

Out[31]:
```
Text(0.5, 0.98, '5_abc.daphne | HMC \n Sample trace')
```



In [32]:
```
G = graph[1]
Y = G['Y']
Y = {key:evaluate([value])[0] for key,value in Y.items()}
P = G['P']
```

In [33]:
```
size = len(samples_whole_graph)
jll = np.zeros(size)
for idx in range(size):
    jll[idx] = compute_log_joint_prob(samples_whole_graph[idx],Y,P)
```

In [34]:
```
pd.Series(jll).plot()
plt.xlabel('Iteration (t)')
plt.ylabel(r'$-\log p(X=x_t,Y=y_t)$')
plt.title('{} | HMC \n Joint density'.format(fname))
```

Out[34]:
```
Text(0.5, 1.0, '5_abc.daphne | HMC \n Joint density')
```