CPSC532W: Probabilistic Programming, Homework 2

Namrata Deka

My code for this homework can be found here: https://github.com/namratadeka/cpsc532W/tree/main/CS532-HW2.

1. Evaluation-Based Sampling Results

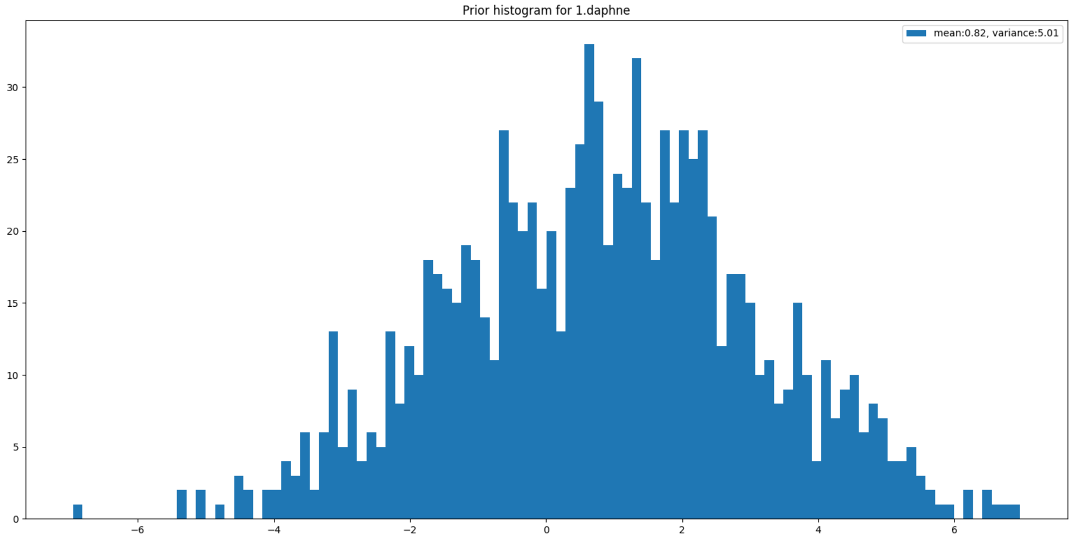


Figure 1: Samples from the prior of 1.daphne

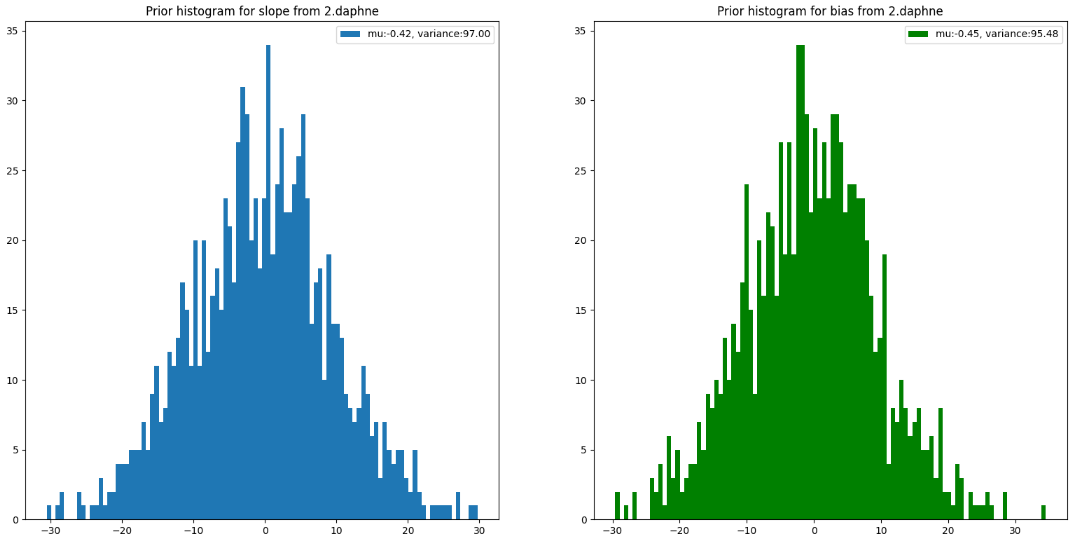


Figure 2: Samples from the prior of 2.daphne. **Right**: Prior for slope. **Left**: Prior for bias.

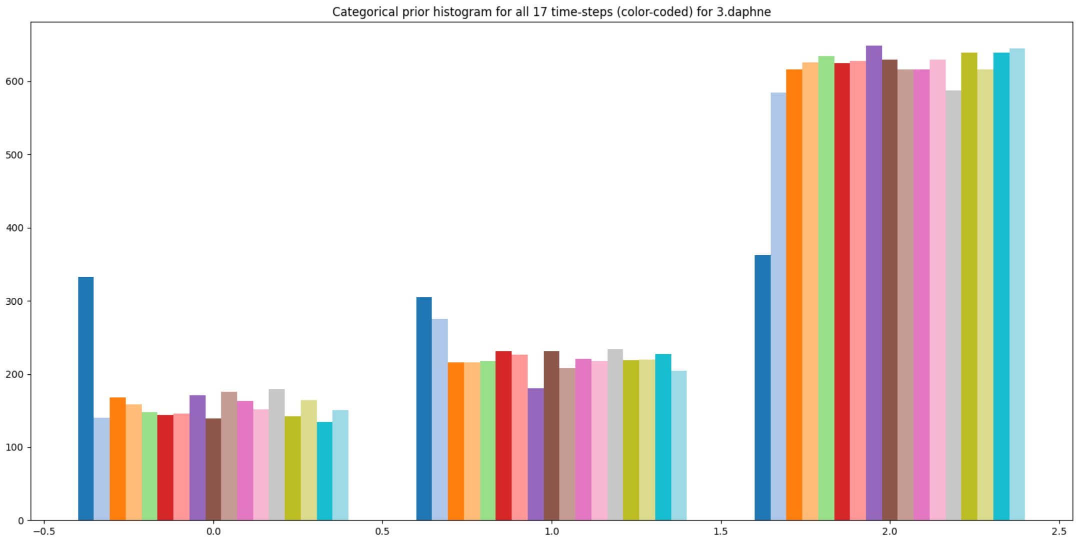


Figure 3: Samples from the prior of 3.daphne. Each color represents one of the 17 hmm steps.

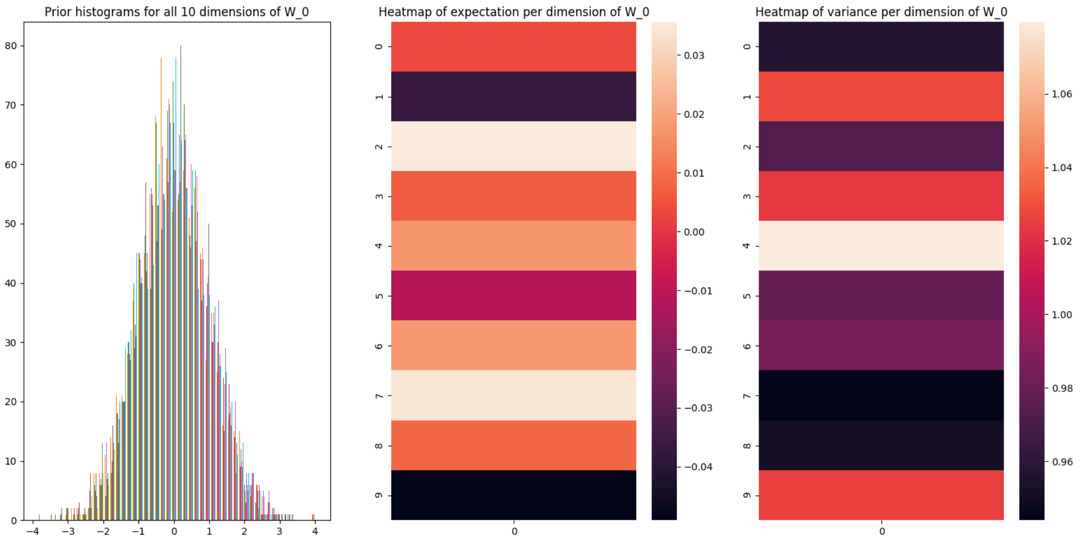


Figure 4: Histogram and heatmaps of mean and variance for W_0 from 4.daphne

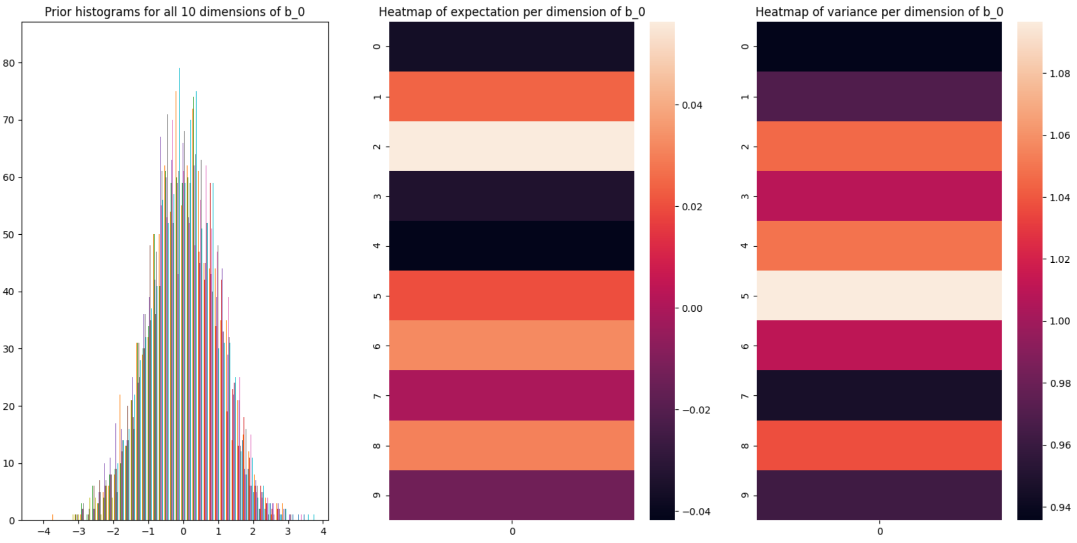


Figure 5: Histogram and heatmaps of mean and variance for b_0 from 4.daphne

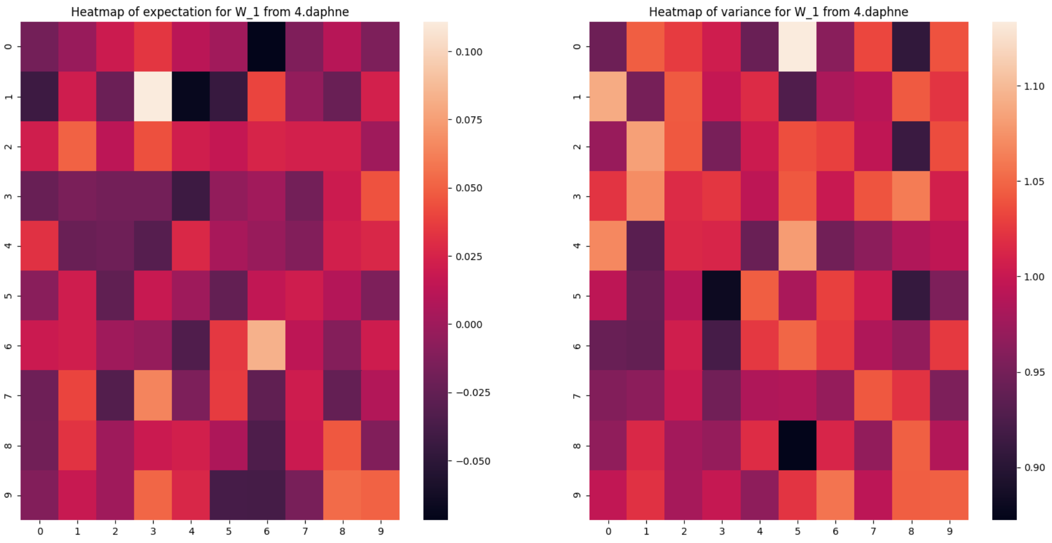


Figure 6: Heatmaps of mean and variance for W_1 from 4.daphne

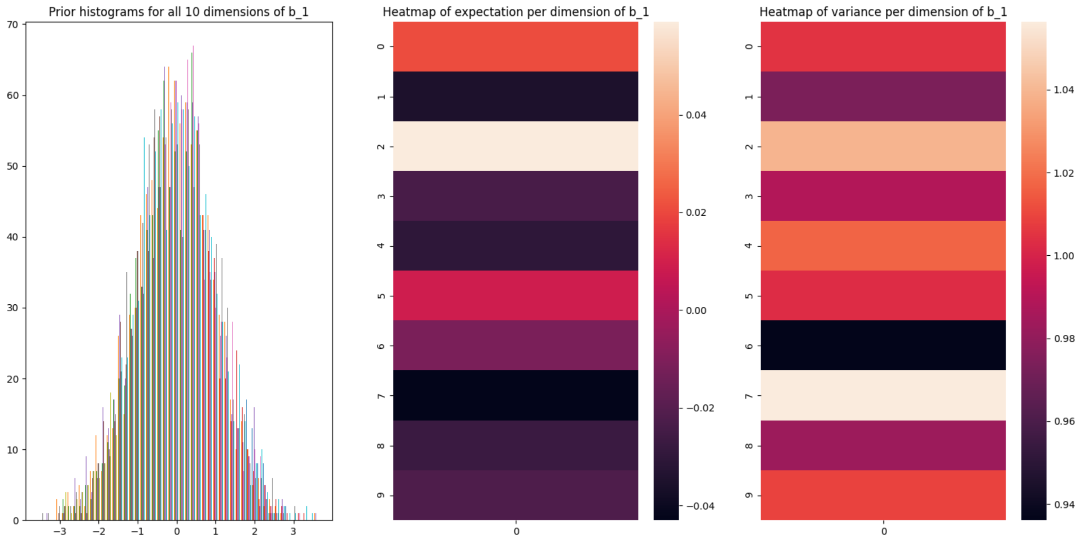


Figure 7: Histogram and heatmaps of mean and variance for b_1 from 4.daphne

2. Graph-Based Sampling Results

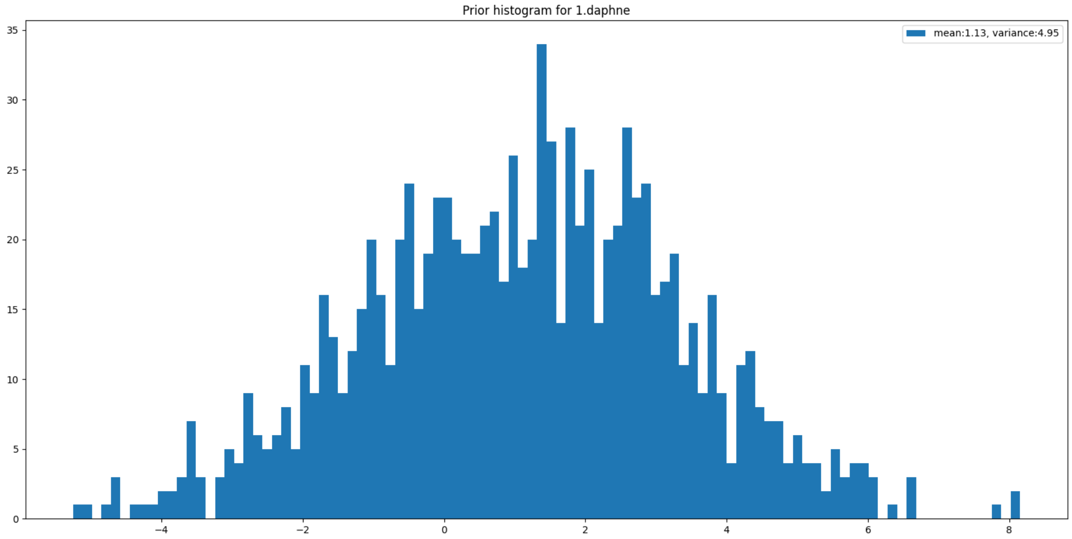


Figure 8: Samples from the prior of 1.daphne

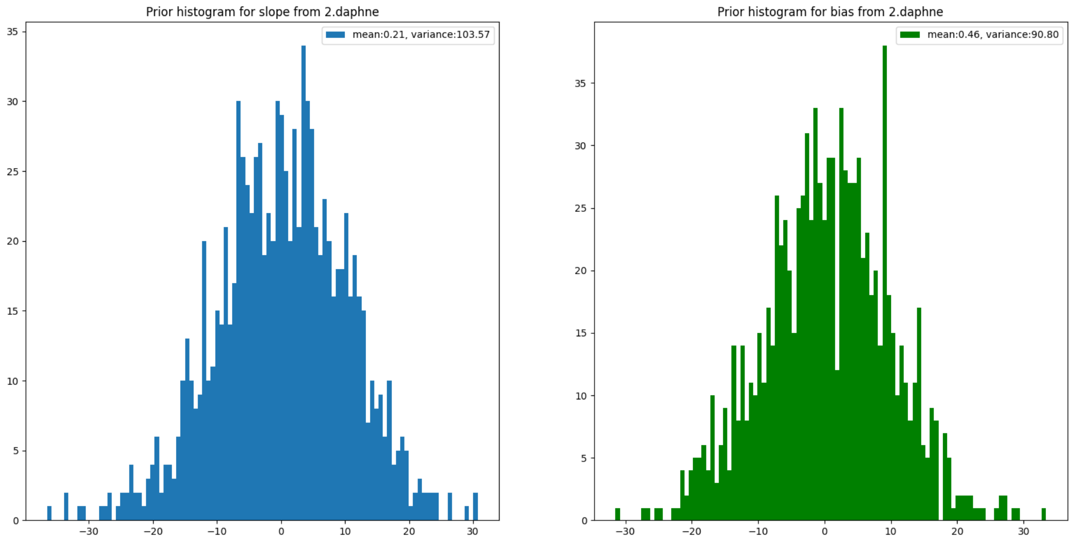


Figure 9: Samples from the prior of 2.daphne. **Right**: Prior for slope. **Left**: Prior for bias.

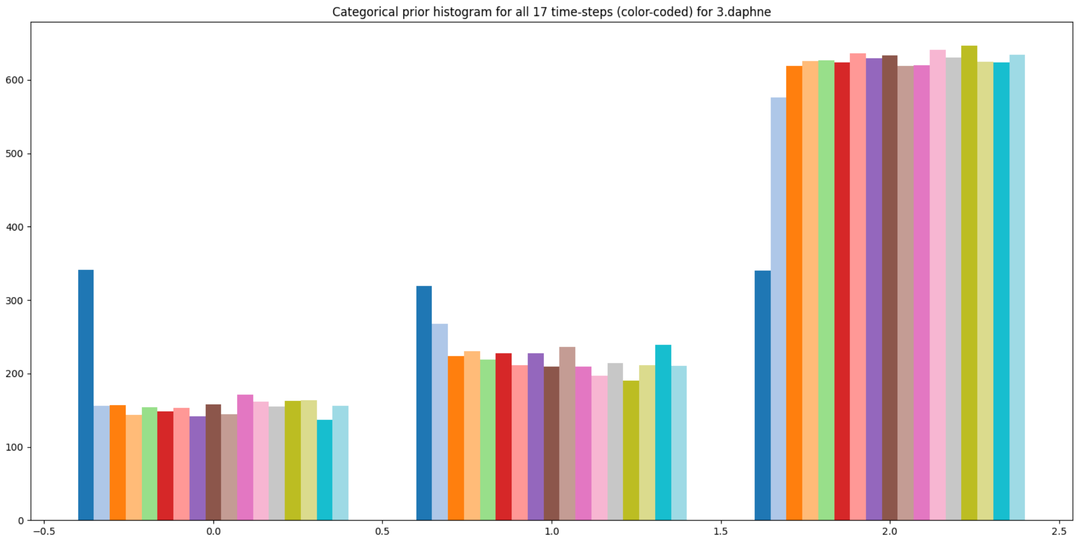


Figure 10: Samples from the prior of 3.daphne. Each color represents one of the 17 hmm steps.

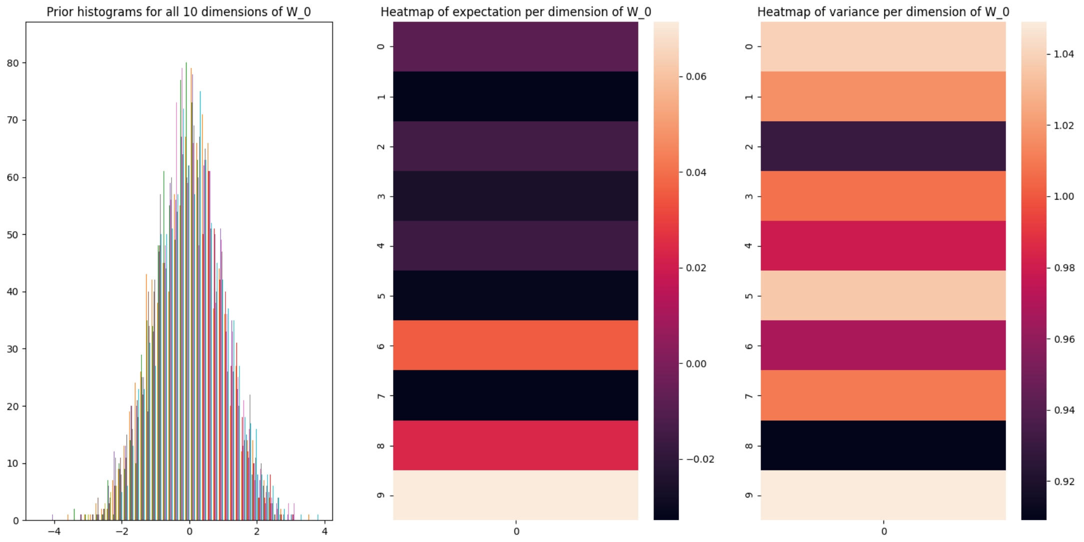


Figure 11: Heatmaps of mean and variance for W_0 from 4.daphne

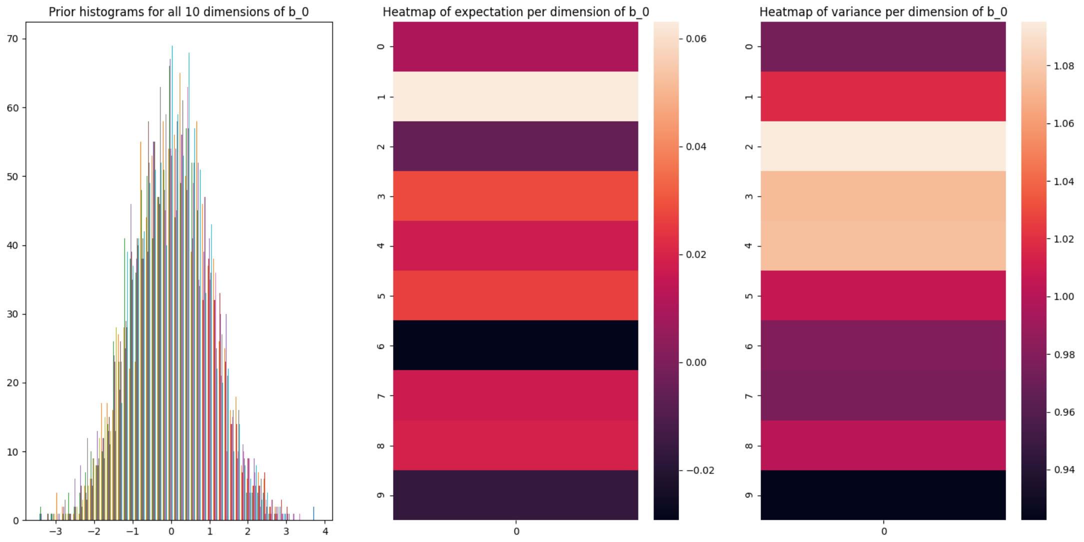


Figure 12: Heatmaps of mean and variance for b_0 from 4.daphne

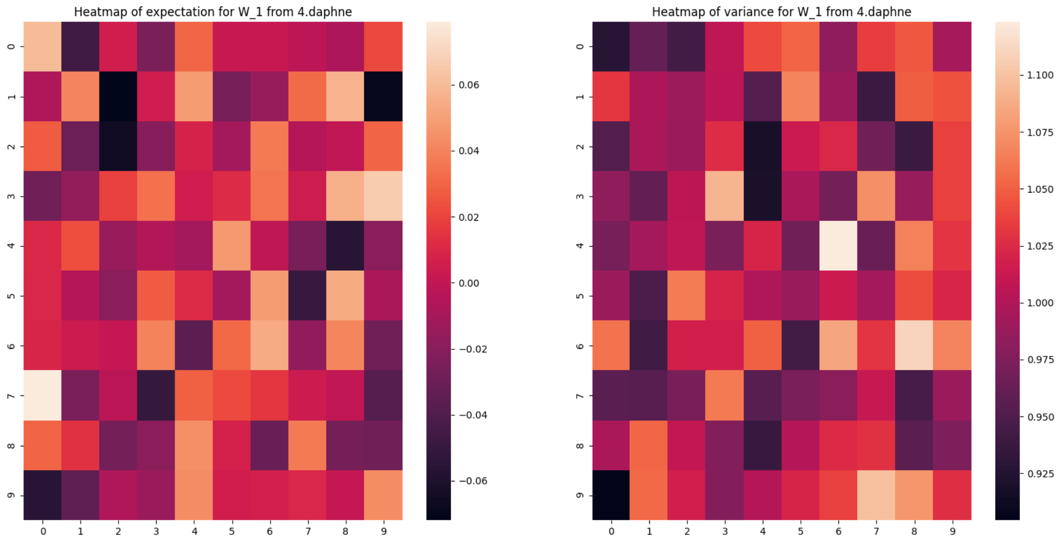


Figure 13: Heatmaps of mean and variance for W_1 from 4.daphne

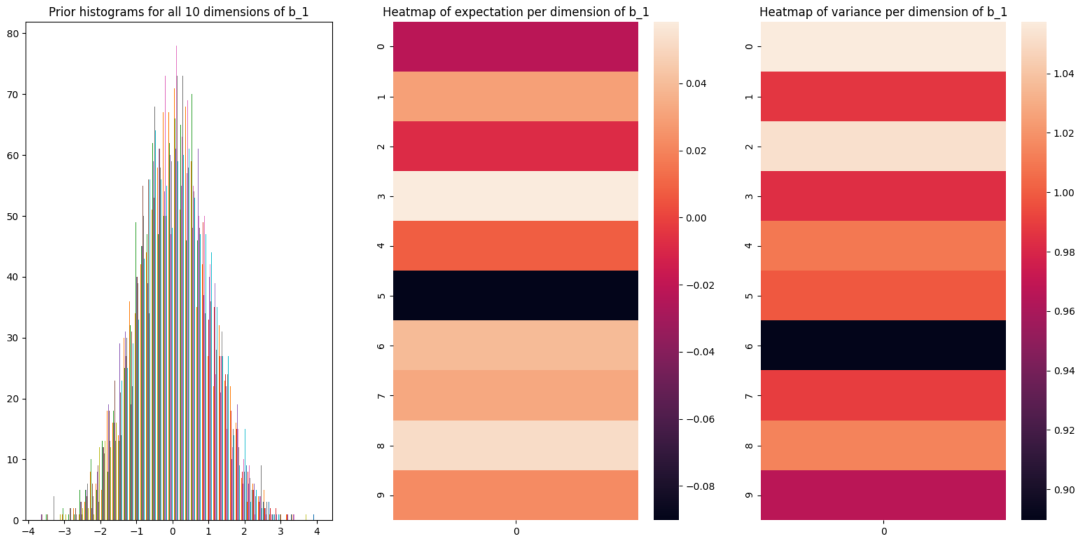
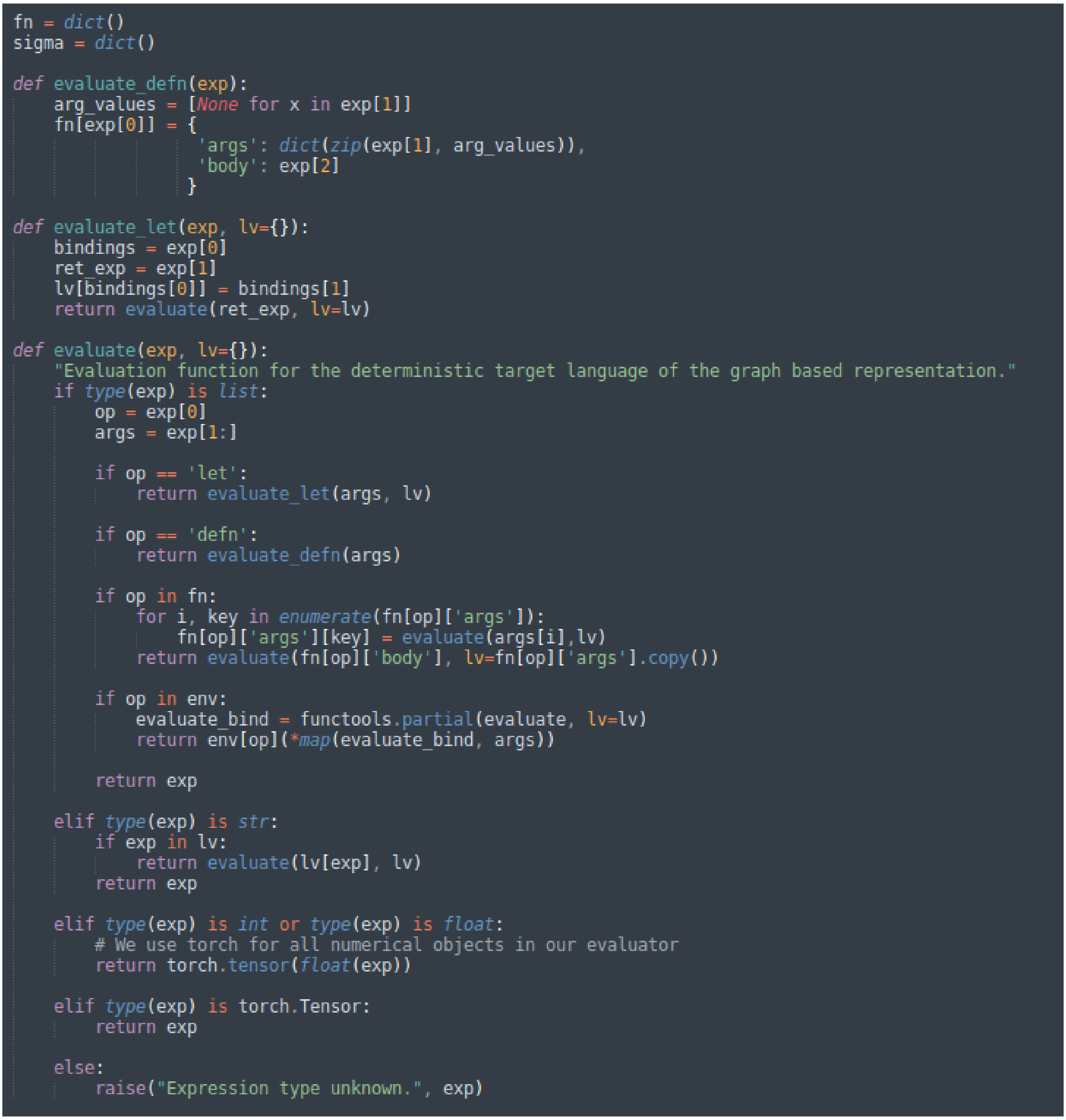


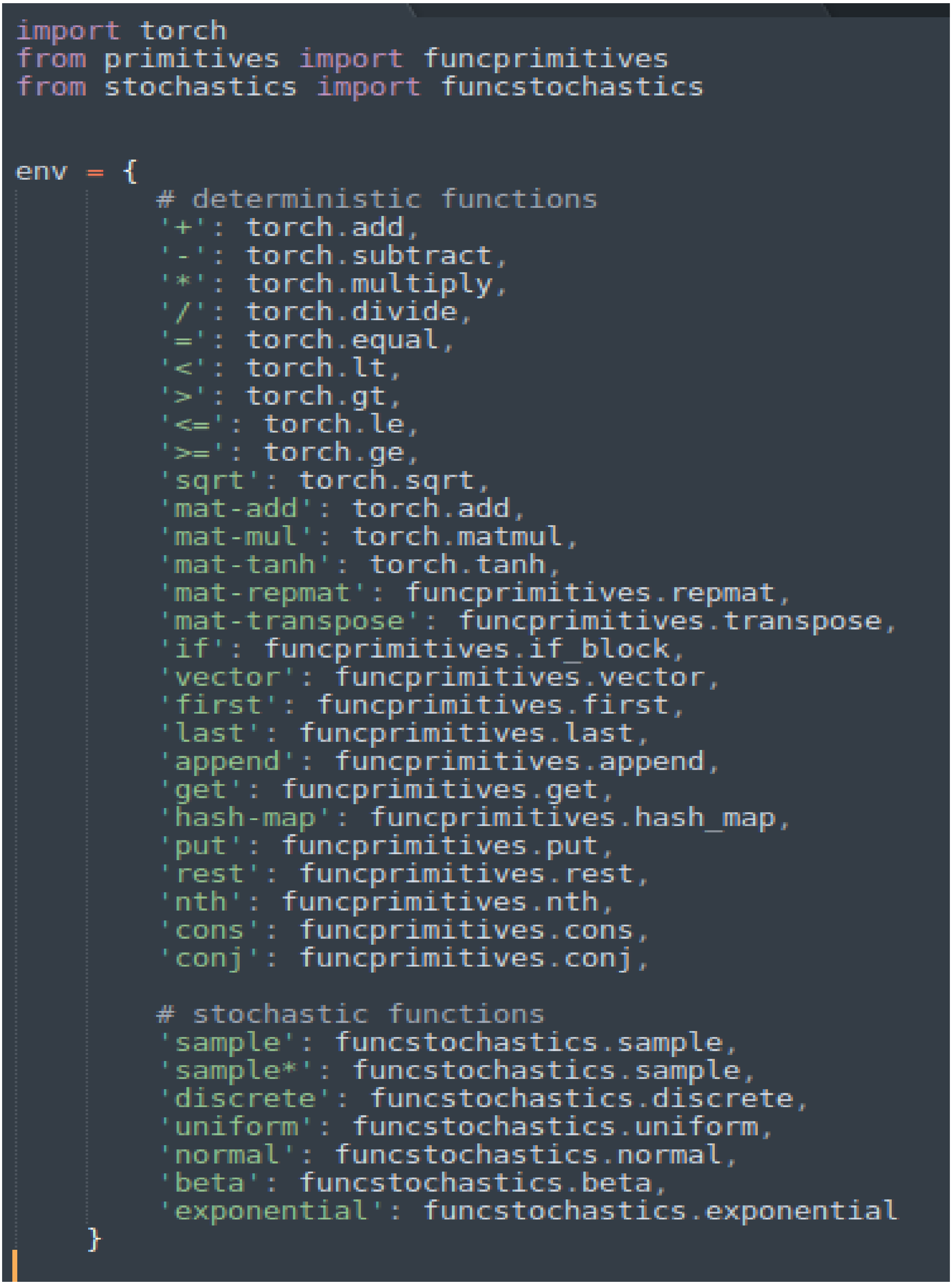
Figure 14: Heatmaps of mean and variance for b_1 from 4.daphne

Code Snippets



```python
fn = dict()
sigma = dict()

def evaluate_defn(exp):
    arg_values = [None for x in exp[1]]
    fn[exp[0]] = {
                'args': dict(zip(exp[1], arg_values)),
                'body': exp[2]
               }

def evaluate_let(exp, lv={}):
    bindings = exp[0]
    ret_exp = exp[1]
    lv[bindings[0]] = bindings[1]
    return evaluate(ret_exp, lv=lv)

def evaluate(exp, lv={}):
    "Evaluation function for the deterministic target language of the graph based representation."
    if type(exp) is list:
        op = exp[0]
        args = exp[1:]

        if op == 'let':
            return evaluate_let(args, lv)

        if op == 'defn':
            return evaluate_defn(args)

        if op in fn:
            for i, key in enumerate(fn[op]['args']):
                fn[op]['args'][key] = evaluate(args[i],lv)
            return evaluate(fn[op]['body'], lv=fn[op]['args'].copy())

        if op in env:
            evaluate_bind = functools.partial(evaluate, lv=lv)
            return env[op](*map(evaluate_bind, args))

        return exp

    elif type(exp) is str:
        if exp in lv:
            return evaluate(lv[exp], lv)
        return exp

    elif type(exp) is int or type(exp) is float:
        # We use torch for all numerical objects in our evaluator
        return torch.tensor(float(exp))

    elif type(exp) is torch.Tensor:
        return exp

    else:
        raise("Expression type unknown.", exp)
```

Figure 15: Main recursive evaluation function.



```
import torch
from primitives import funcprimitives
from stochastics import funcstochastics


env = {
        # deterministic functions
        '+': torch.add,
        '-': torch.subtract,
        '*': torch.multiply,
        '/': torch.divide,
        '=': torch.equal,
        '<': torch.lt,
        '>': torch.gt,
        '<=': torch.le,
        '>=': torch.ge,
        'sqrt': torch.sqrt,
        'mat-add': torch.add,
        'mat-mul': torch.matmul,
        'mat-tanh': torch.tanh,
        'mat-repmat': funcprimitives.repmat,
        'mat-transpose': funcprimitives.transpose,
        'if': funcprimitives.if_block,
        'vector': funcprimitives.vector,
        'first': funcprimitives.first,
        'last': funcprimitives.last,
        'append': funcprimitives.append,
        'get': funcprimitives.get,
        'hash-map': funcprimitives.hash_map,
        'put': funcprimitives.put,
        'rest': funcprimitives.rest,
        'nth': funcprimitives.nth,
        'cons': funcprimitives.cons,
        'conj': funcprimitives.conj,

        # stochastic functions
        'sample': funcstochastics.sample,
        'sample*': funcstochastics.sample,
        'discrete': funcstochastics.discrete,
        'uniform': funcstochastics.uniform,
        'normal': funcstochastics.normal,
        'beta': funcstochastics.beta,
        'exponential': funcstochastics.exponential
    }
```

Figure 16: Supported operations