CPSC 532W Assignment 5

Ali Seyfi - 97446637

Here is the link to the repository:
https://github.com/aliseyfi75/Probabilistic-Programming/tree/master/Assignment_5

1 Code

Report successful in passing all of the deterministic tests in the HW support code.

1.1 primitives

Provide code snippets that document critical aspects of your implementation sufficient to allow us to quickly determine whether or not you individually completed the assignment.

```python
class Env(dict):
    "An environment: a dict of {'var': val} pairs, with an outer Env."
    def __init__(self, parms=(), args=(), outer=None):
        try:
            if type(args[0]) == type([]):
                self.update(pmap(zip(parms, args[0])))
            else:
                self.update(pmap(zip(parms, args)))
        except:
            self.update(pmap(zip(parms, args)))
        self.outer = outer
    def find(self, var):
        "Find the innermost Env where var appears."
        return self if (var in self) else self.outer.find(var)

class Procedure(object):
    "A user-defined Scheme procedure."
    def __init__(self, parms, body, env, eval_func):
        self.parms, self.body, self.env, self.eval = parms, body, env, eval_func
    def __call__(self, *args):
        return self.eval(self.body, Env(self.parms, args, outer=self.env))
```

Listing 1: primitives.py - Env and Procedure definitions

```python
class Dist:
    def __init__(self, name, distribution, num_par, *par):
        self.name = name
        self.distribution = distribution
        self.num_par = num_par
        self.pars = []
        for i in range(num_par):
            self.pars.append(par[i])

    def sample(self):
        return self.distribution.sample()

    def log_prob(self, x):
        return self.distribution.log_prob(x)

    def parameters(self):
        return self.distribution.Parameters()

    def make_copy_with_grads(self):
        temp_dist = self.distribution
        self.distribution = None
        dist_copy = copy.deepcopy(self)
        self.distribution = temp_dist
        dist_copy.distribution = temp_dist.make_copy_with_grads()
        return dist_copy

class normal(Dist):
    def __init__(self, pars):
        mean = pars[0]
        var = pars[1]
        super().__init__(’normal’, distributions.Normal(mean, var), 2, mean, var)

class beta(Dist):
    def __init__(self, pars):
        alpha = pars[0]
        betta = pars[1]
        super().__init__(’beta’, dist.Beta(alpha, betta), 2, alpha, betta)

class exponential(Dist):
    def __init__(self, par):
        lamda = par[0]
        super().__init__(’exponential’, dist.Exponential(lamda), 1, lamda)

class uniform(Dist):
    def __init__(self, pars):
        a = pars[0]
        b = pars[1]
        super().__init__(’uniform’, distributions.Uniform(a, b), 2, a, b)

class discrete(Dist):
    def __init__(self, pars):
        prob = pars[0]
        super().__init__(’discrete’, distributions.Categorical(prob), 0)

class bernoulli(Dist):
    def __init__(self, pars):
        p = pars[0]
        super().__init__(’bernoulli’, distributions.Bernoulli(p), 1, p)
```

Listing 2: primitives.py -distributions

```
def push_addr(alpha, value):
    return alpha + value

def vector(x):
    try:
        vector = torch.stack(x)
    except:
        vector = x
    return vector

def list(x):
    try:
        list = torch.stack(x)
    except:
        list = x
    return list

def get(x):
    if type(x[0]) == type(pmap()):
        if torch.is_tensor(x[1]):
            item = x[1].item()
        else:
            item = x[1]
        value = x[0].get(item)
    else:
        value = x[0][x[1].long()]
    return value

def put(x):
    if type(x[0]) == type(pmap()):
        if torch.is_tensor(x[1]):
            item = x[1].item()
        else:
            item = x[1]

        x[0] = x[0].set(item, x[2])
    else:
        x[0][x[1].long()] = x[2]
    return x[0]

def hash_map(x):
    keys = x[::2]
    value = x[1::2]
    new_keys = []
    for key in keys:
        if torch.is_tensor(key):
            new_keys.append(key.item())
        else:
            new_keys.append(key)
    result = pmap(zip(new_keys, value))
    return result

def append(x):
    first = x[0]
    second = x[1]

    if type(first) == type([]):
        first = torch.tensor(first)
    elif first.dim() == 0:
        first = first.unsqueeze(0)
    if type(second) == type([]):
        second = torch.tensor(second)
    if second.dim() == 0:
        second = second.unsqueeze(0)
    return torch.cat((first, second))

def cons(x):
    first = x[1]
    second = x[0]

    if type(first) == type([]):
        first = torch.tensor(first)
    elif first.dim() == 0:
        first = first.unsqueeze(0)
    if type(second) == type([]):
        second = torch.tensor(second)
    if second.dim() == 0:
        second = second.unsqueeze(0)
    return torch.cat((first, second))
```

Listing 3: primitives.py - functions

```
env = {
        ’+’: lambda x: x[0] + x[1],
        ’-’: lambda x: x[0] - x[1],
        ’*’: lambda x: x[0] * x[1],
        ’/’: lambda x: x[0] / x[1],
        ’>’: lambda x: x[0] > x[1],
        '>=': lambda x: x[0] >= x[1],
        '<': lambda x: x[0] < x[1],
        '<=': lambda x: x[0] <= x[1],
        '==': lambda x: x[0] == x[1],
        'sqrt': lambda x: torch.sqrt(x[0]),
        'exp': lambda x: torch.exp(x[0]),
        'log': lambda x: torch.log(x[0]),
        'or': lambda x: x[0] or x[1],
        'and': lambda x: x[0] and x[1],
        'empty?': lambda x: len(x[0]) == 0,
        'vector': vector,
        'list': list,
        'get': get,
        'put': put,
        'hash-map': hash_map,
        'push-address' : push_addr,
        'first': lambda x: x[0][0],
        'last': lambda x: x[0][-1],
        'nth': lambda x: x[0][int(x[1].item())],
        'second': lambda x: x[0][1],
        'rest': lambda x: x[0][1:],
        'peek': lambda x: x[0][-1],
        'append': append,
        'cons': cons,
        'conj': append,
        'mat-add': lambda x: x[0] + x[1],
        'mat-mul': lambda x: torch.matmul(x[0], x[1]),
        'mat-transpose': lambda x: x[0].T,
        'mat-tanh': lambda x: x[0].tanh(),
        'mat-repmat': lambda x: x[0].repeat((int(x[1].item()), int(x[2].item()))),
        'normal' : normal,
        'beta' : beta,
        'exponential' : exponential,
        'uniform' : uniform,
        'discrete' : discrete,
        'bernoulli' : bernoulli,
        'uniform-continuous' : uniform,
        'flip' : bernoulli
       }
```

Listing 4: primitives.py - environment

1.2 evaluator

```
def standard_env():
    "An environment with some Scheme standard procedures."
    env = Env()
    env.update(penv)
    env.update({'alpha' : ''})
    return env
```

Listing 5: evaluator.py - standard env

```
def evaluate(exp, env=None):

    if env is None or len(env) == 0:
        env = standard_env()
    result = eval(exp[2], env=env)

    if type(result) == type(pmap()):
        result = dict(result)
    return result
```

Listing 6: evaluator.py - evaluate

```python
def eval(exp, env=None):
    "Evaluate an expression in an environment."
    if isinstance(exp, Symbol):
        if exp.startswith('"') and exp.endswith('"'):
            result = exp
        else:
            result = env.find(exp)[exp]

    elif not isinstance(exp, List):
        if isinstance(exp, int) or isinstance(exp, float):
            result = torch.tensor(float(exp))
        else:
            result = exp
    else:
        operation, *args = exp
        if operation == 'if':
            (condition, true_exp, false_exp) = args
            if eval(condition, env):
                result = eval(true_exp, env)
            else:
                result = eval(false_exp, env)

        elif operation == 'defn':
            (name, value) = args
            env[name] = eval(value, env)
            result = None

        elif operation == 'fn':
            (params, body) = args
            result = Procedure(params[1:], body, env, eval)

        elif operation == 'sample':
            alpha = eval(args[0], env)
            dist = eval(args[1], env)
            result = dist.sample()

        elif operation == 'observe':
            alpha = eval(args[0], env)
            dist = eval(args[1], env)
            observation = eval(args[2], env)
            result = observation

        elif operation == 'push-address':
            result = None

        else:
            proc = eval(operation, env)
            alpha = eval(args[0], env)
            vars = [eval(arg, env) for arg in args[1:]]
            result = proc(vars)

    return result
```

Listing 7: evaluator.py - eval

2 Program 1

Results of Tests
Here are the results of the test files:

2.1 Deterministic

```
tensor(7.)
FOPPL Tests passed
tensor(1.4142)
FOPPL Tests passed
tensor(24.)
FOPPL Tests passed
tensor(0.2500)
FOPPL Tests passed
tensor(0.1802)
FOPPL Tests passed
tensor([2., 3., 4., 5.])
FOPPL Tests passed
tensor(4.)
FOPPL Tests passed
tensor([2., 3., 3., 5.])
FOPPL Tests passed
tensor(2.)
FOPPL Tests passed
tensor(5.)
FOPPL Tests passed
tensor([2.0000, 3.0000, 4.0000, 5.0000, 3.1400])
FOPPL Tests passed
tensor(5.3000)
FOPPL Tests passed
{1.0: tensor(3.2000), 6.0: tensor(2.)}
FOPPL Tests passed
```

2.2 HOPPL Deterministic

```
tensor(71.)
Test passed
tensor(89.)
Test passed
tensor(6.)
Test passed
tensor(1.)
Test passed
tensor([10., 9.])
Test passed
tensor(1.)
Test passed
tensor(4.)
Test passed
tensor([11., 2., 8.])
Test passed
tensor(4.)
Test passed
tensor(120.)
Test passed
tensor(6.)
Test passed
tensor([2., 3., 4.])
Test passed
All deterministic tests passed
```

2.3 Probabilistic

```
(’normal’, 5, 1.4142136)
p value 0.976905685570249
(’beta’, 2.0, 5.0)
p value 0.9649610549178714
(’exponential’, 0.0, 5.0)
p value 0.28116937738711634
(’normal’, 5.3, 3.2)
p value 0.21365199633322285
(’normalmix’, 0.1, -1, 0.3, 0.9, 1, 0.3)
p value 0.718135587890695
(’normal’, 0, 1.44)
p value 0.6593599766116442
All probabilistic tests passed
```

3 Program 2

I draw **20000** samples.
Mean of until success is: **98.8752**
Variance of until success is: **10014.3056**
Running time: **2 min and 9.8 seconds**

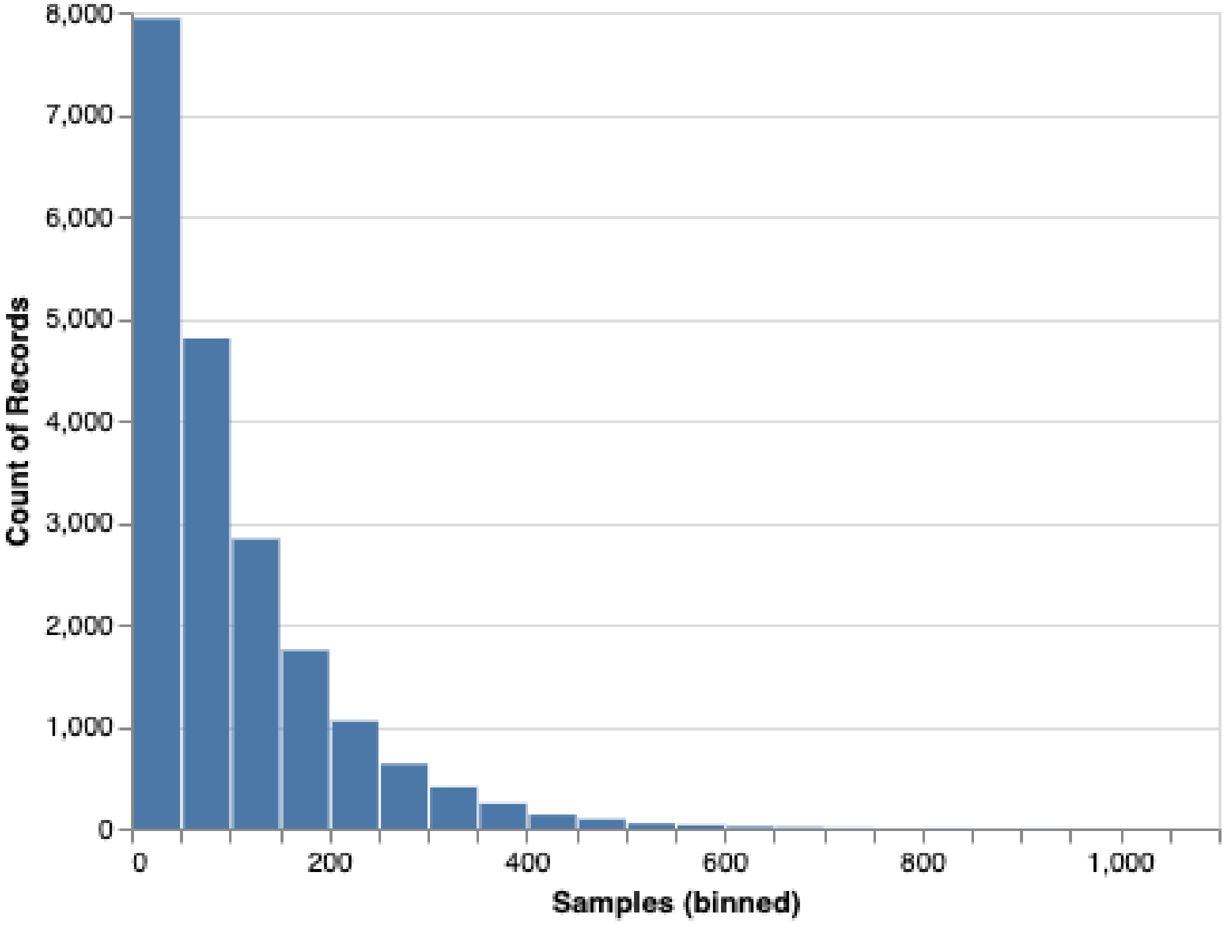

Figure 1: Histogram of untill success

4 Program 3

I draw **100000** samples.
Mean of μ is: **1.0016**.
Variance of μ is: **4.9907**
Running time: **38.8 seconds**

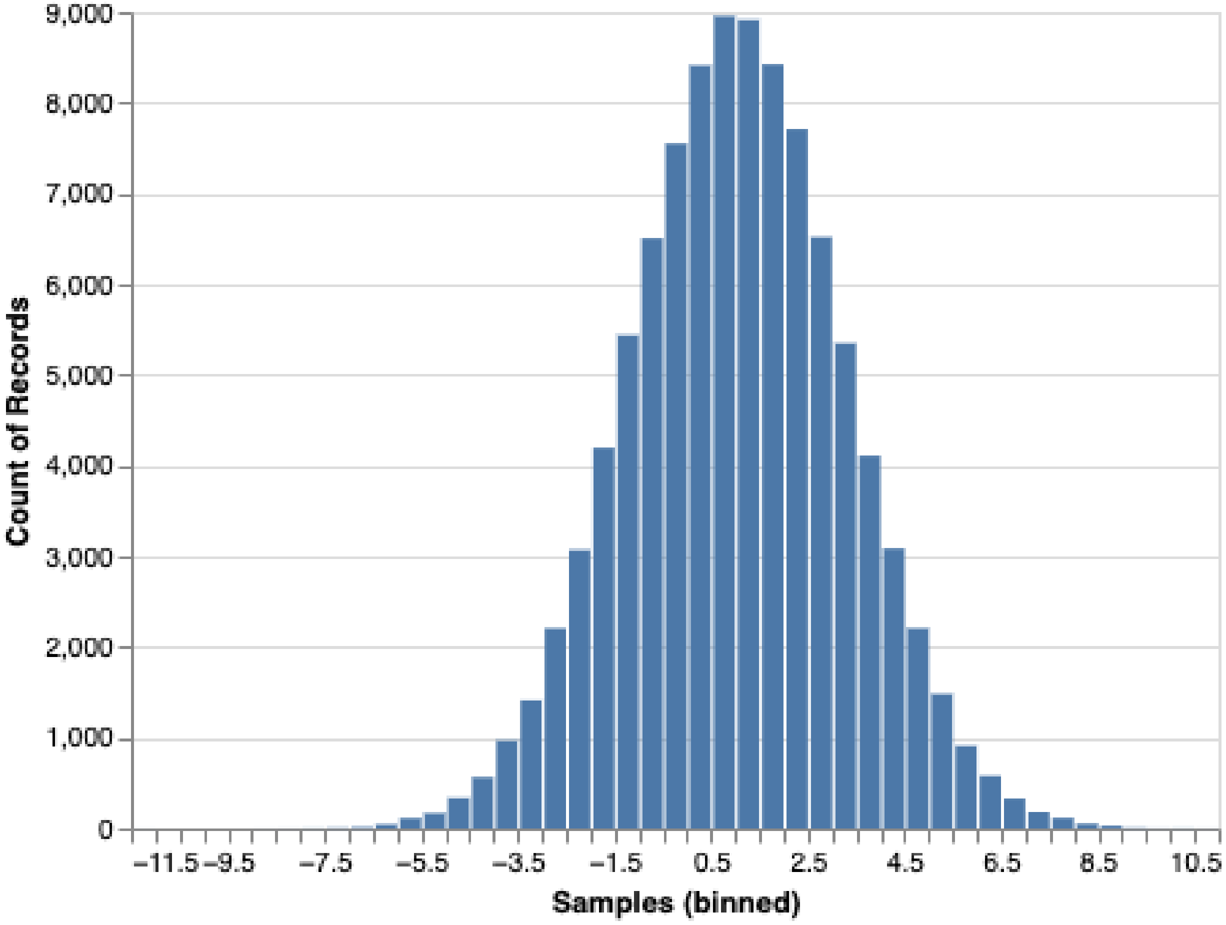

Figure 2: Histogram of μ

5 Program 4

I draw **100000** samples.
Running time: **3 minutes and 39.4 seconds**

The distribution over states in each step is:

$$\begin{bmatrix}
0.33351 & 0.33351 & 0.33141 \\
0.15164 & 0.28333 & 0.56503 \\
0.15587 & 0.21700 & 0.62713 \\
0.15293 & 0.21487 & 0.63220 \\
0.15289 & 0.21514 & 0.63197 \\
0.15137 & 0.21454 & 0.63409 \\
0.15278 & 0.21604 & 0.63118 \\
0.15353 & 0.21461 & 0.63186 \\
0.15249 & 0.21352 & 0.63399 \\
0.15324 & 0.21412 & 0.63264 \\
0.15311 & 0.21334 & 0.63355 \\
0.15281 & 0.21573 & 0.63146 \\
0.15356 & 0.21176 & 0.63468 \\
0.15296 & 0.21367 & 0.63337 \\
0.15222 & 0.21193 & 0.63585 \\
0.15261 & 0.21384 & 0.63355 \\
0.15254 & 0.21318 & 0.63428
\end{bmatrix}$$

The mean of the state value in each step is:

$$\begin{bmatrix}
0.99790 \\
1.41339 \\
1.47126 \\
1.47927 \\
1.47908 \\
1.48272 \\
1.47840 \\
1.47833 \\
1.48150 \\
1.47940 \\
1.48044 \\
1.47865 \\
1.48112 \\
1.48041 \\
1.48363 \\
1.48094 \\
1.48174
\end{bmatrix}$$

The variance of the state value in each step is:

$$\begin{bmatrix} 0.664922 \\ 0.545784 \\ 0.560920 \\ 0.555436 \\ 0.555348 \\ 0.552447 \\ 0.555099 \\ 0.556596 \\ 0.554643 \\ 0.556061 \\ 0.555843 \\ 0.555170 \\ 0.556769 \\ 0.555542 \\ 0.554178 \\ 0.554862 \\ 0.554752 \end{bmatrix}$$

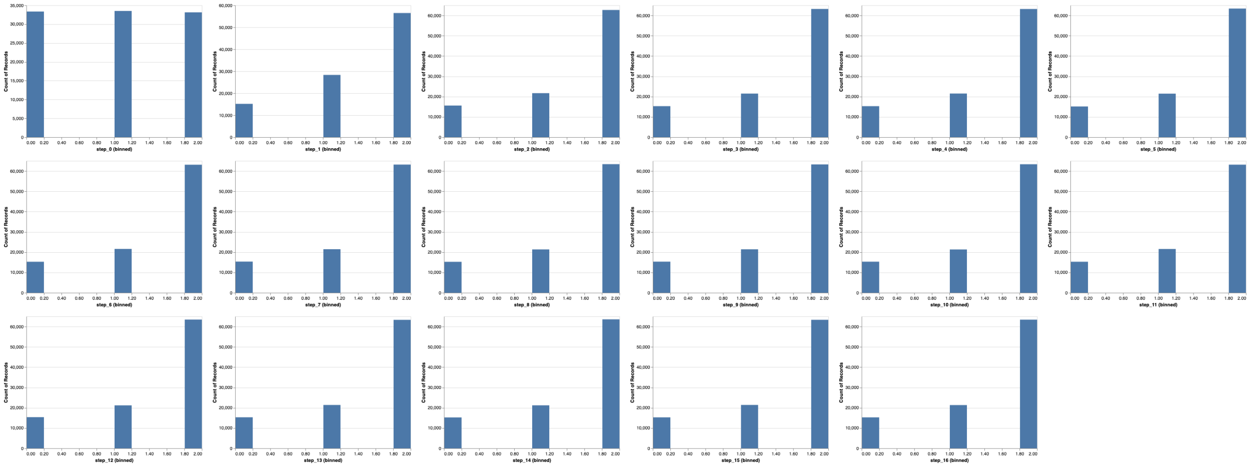

Figure 3: Histogram of states in each step