CPSC 532 - Homework 2

Xiaoxuan Liang - 48131163

1. Evaluation-based Sampling:

   (a) Program 1): in 1000 samplings from this program, each return value is a numeric number estimating mean. Therefore, 1000 samplings return an array of length 1000 overall.

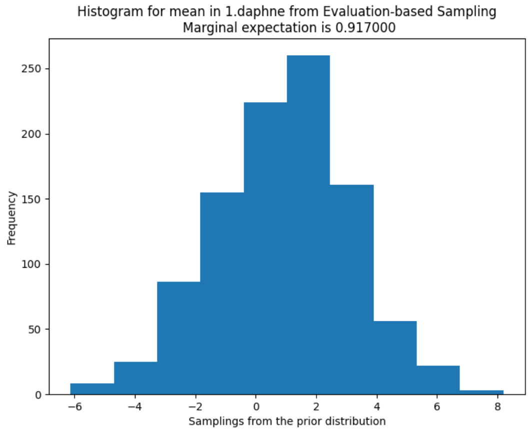
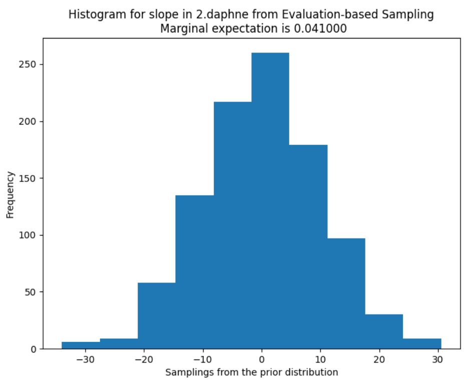
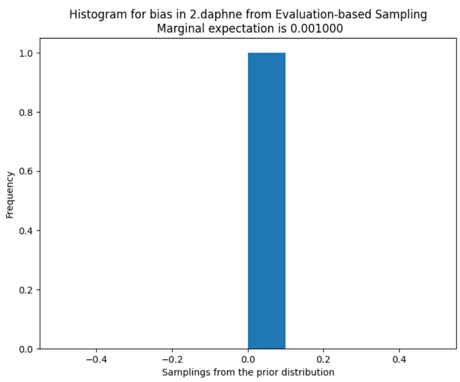


Figure 1: Histogram from the prior for mean for 1.daphne

   (b) Program 2): in 1000 samplings from this program, each return value is an array of length 2, which incorporates the estimations for both slope and bias. Therefore, 1000 samplings return a 2-D array of size 1000×2 overall.



(a) Samples from the prior for slope

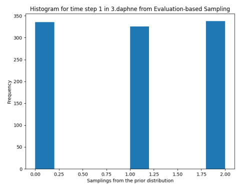


(b) Samples from the prior for bias

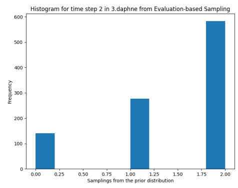
Figure 2: Histograms for 2.daphne

(c) Program 3): in 1000 samplings from this program, each return value is an array of length 17, which incorporates the estimations for 17 HMM time steps. Therefore, 1000 sampling returns a 2-D array of size 1000×17 overall.

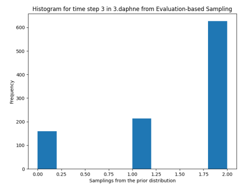
It does not provide statistical meaning for calculating the marginal expectation of the categorical variable, instead, we can approximate the stationary distribution for the HMM by constructing a 3×3 matrix to record the transition times among all states and calculating the proportion of status of states, which is $\left[1506875 \quad 0.2224375, \quad 0.626875\right]$.



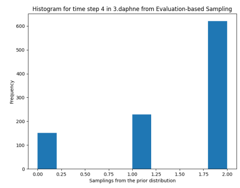
(a) Samples from the prior for HMM step 1



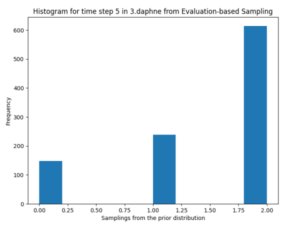
(b) Samples from the prior for HMM step 2



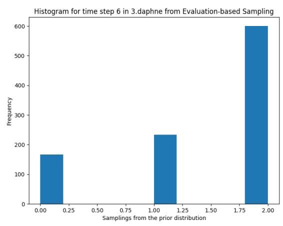
(c) Samples from the prior for HMM step 3



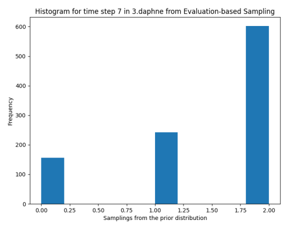
(d) Samples from the prior for HMM step 4



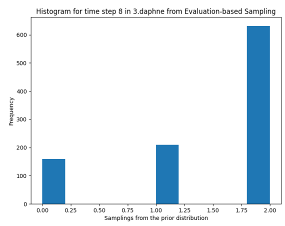
(e) Samples from the prior for HMM step 5



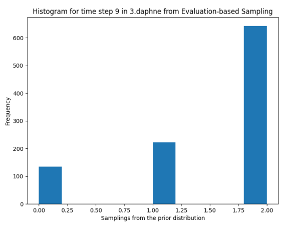
(f) Samples from the prior for HMM step 6



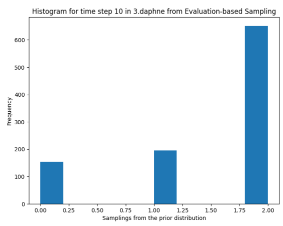
(g) Samples from the prior for HMM step 7



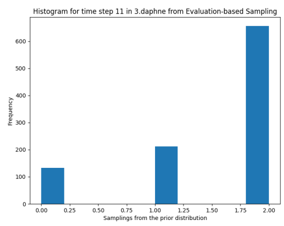
(h) Samples from the prior for HMM step 8



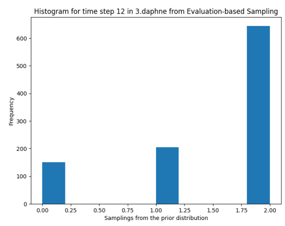
(i) Samples from the prior for HMM step 9



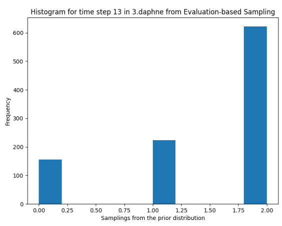
(j) Samples from the prior for HMM step 10



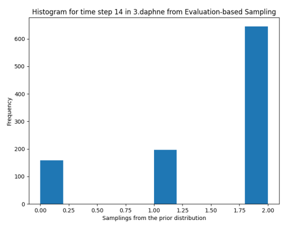
(k) Samples from the prior for HMM step 11



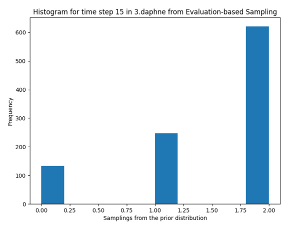
(l) Samples from the prior for HMM step 12



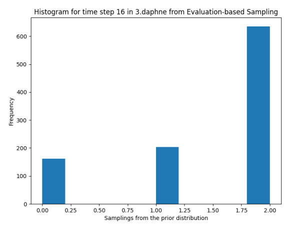
(m) Samples from the prior for HMM step 13



(n) Samples from the prior for HMM step 14

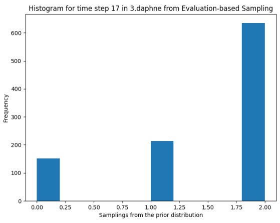


(o) Samples from the prior for HMM step 15



(p) Samples from the prior for HMM step 16

Figure 3: Partial Histograms for 3.daphne

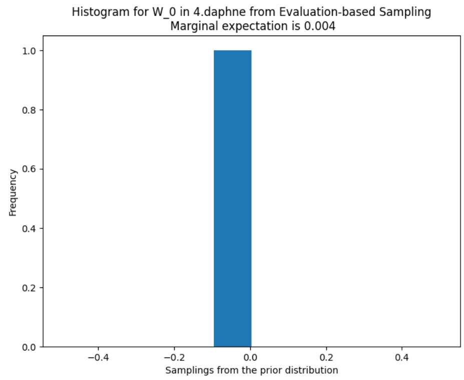


(a) Samples from the prior for HMM step 17

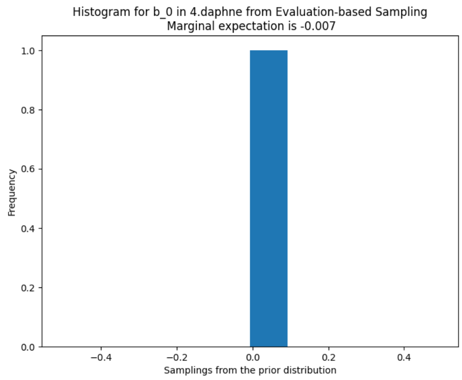
Figure 4: Histograms for 3.daphne

(d) Program 4): in 1000 samplings from this program, each return values is a 3-D array incorporating the estimations for W_0, b_0, W_1, b_1. W_0 is a 2-D array of size 10×1, b_0 is a 2-D array of size 10×1, W_1 is a 2-D array of size 10×10, and b_1 is a 2-D array of size 10×1.

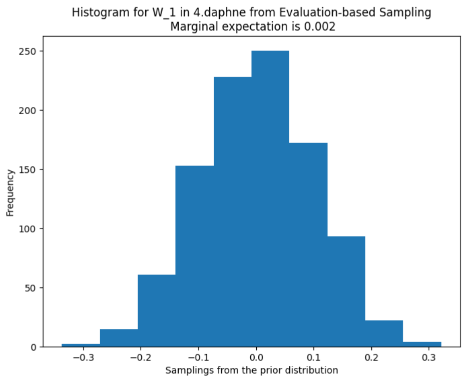
Since all W_0, b_0, W_1, b_1 are sampled from standard normal distribution, I took the mean across the sampled array from each sampling regarded as the resulted sampling, instead of plotting too many histograms.



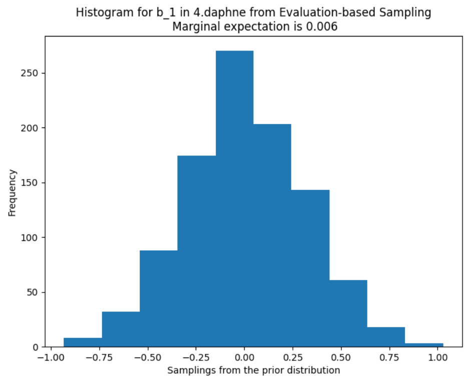
(a) Samples from the prior for W_0



(b) Samples from the prior for b_0



(c) Samples from the prior for W_1



(d) Samples from the prior for b_1

Figure 5: Histograms for 4.daphne

2. Graph-based Sampling:

   (a) Program 1): in 1000 samplings from this program, each return value is a numeric number estimating mean. Therefore, 1000 samplings return an array of length 1000 overall.

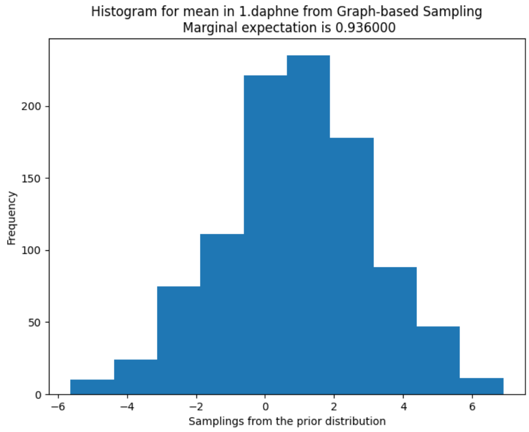
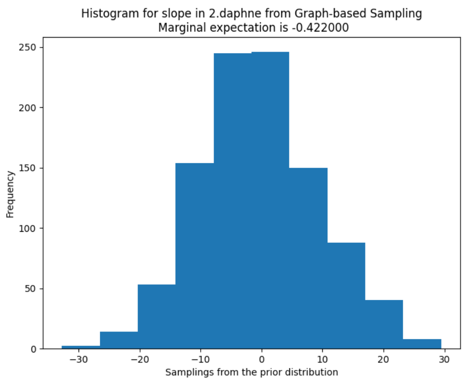
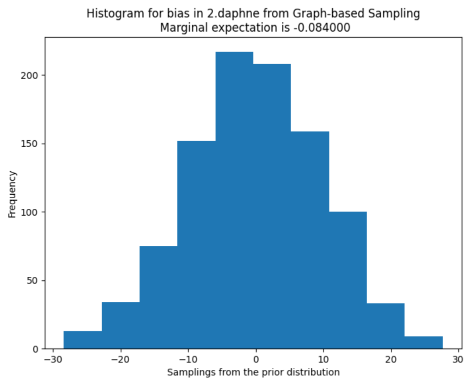


Figure 6: Histogram from the prior for mean for 1.daphne

   (b) Program 2): in 1000 samplings from this program, each return value is an array of length 2, which incorporates the estimations for both slope and bias. Therefore, 1000 samplings return a 2-D array of size 1000×2 overall.



(a) Samples from the prior for slope

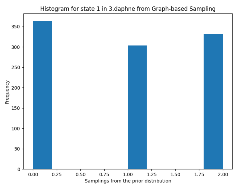


(b) Samples from the prior for bias

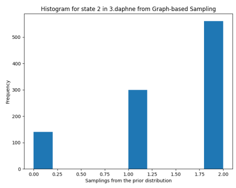
Figure 7: Histograms for 2.daphne

(c) Program 3): in 1000 samplings from this program, each return value is an array of length 17, which incorporates the estimations for 17 HMM time steps. Therefore, 1000 sampling returns a 2-D array of size 1000×17 overall.

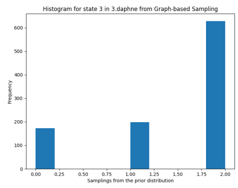
It does not provide statistical meaning for calculating the marginal expectation of the categorical variable, instead, we can approximate the stationary distribution for the HMM by constructing a 3×3 matrix to record the transition times among all states and calculating the proportion of status of states, which is $\begin{bmatrix} 0.1550625 & 0.219625 & 0.6253125 \end{bmatrix}$.



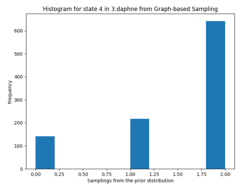
(a) Samples from the prior for HMM step 1



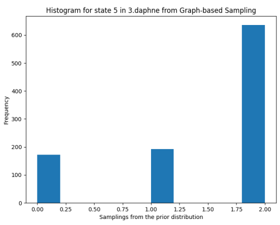
(b) Samples from the prior for HMM step 2



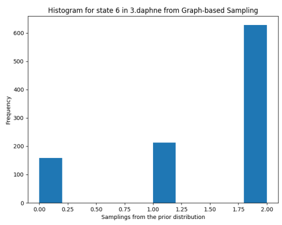
(c) Samples from the prior for HMM step 3



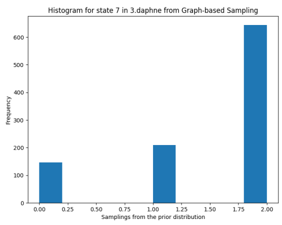
(d) Samples from the prior for HMM step 4



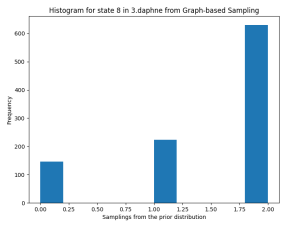
(e) Samples from the prior for HMM step 5



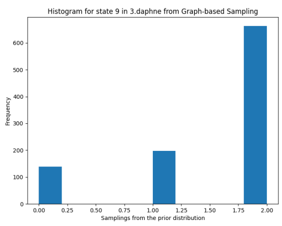
(f) Samples from the prior for HMM step 6



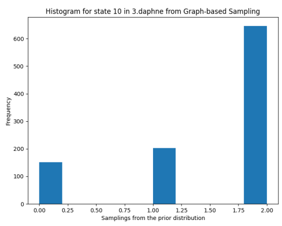
(g) Samples from the prior for HMM step 7



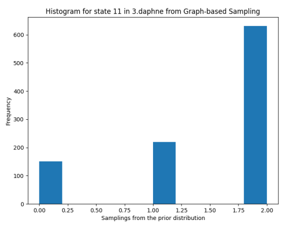
(h) Samples from the prior for HMM step 8



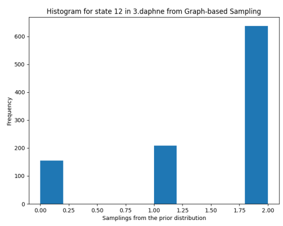
(i) Samples from the prior for HMM step 9



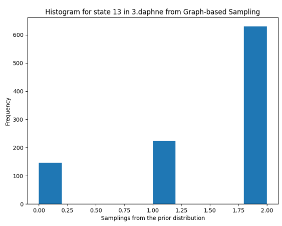
(j) Samples from the prior for HMM step 10



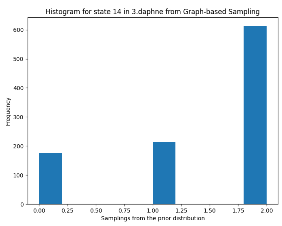
(k) Samples from the prior for HMM step 11



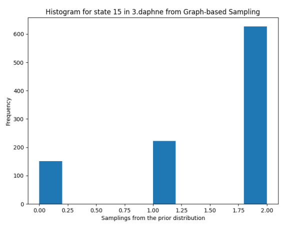
(l) Samples from the prior for HMM step 12



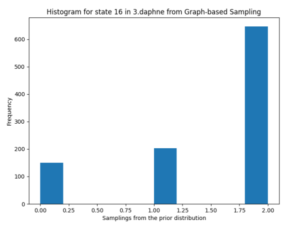
(m) Samples from the prior for HMM step 13



(n) Samples from the prior for HMM step 14

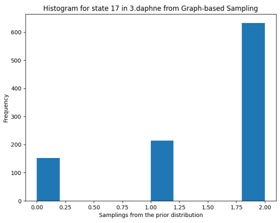


(o) Samples from the prior for HMM step 15



(p) Samples from the prior for HMM step 16

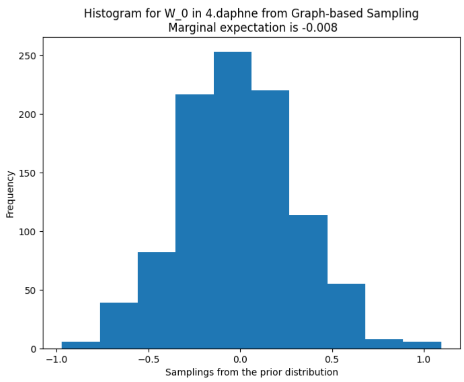
Figure 8: Partial Histograms for 3.daphne



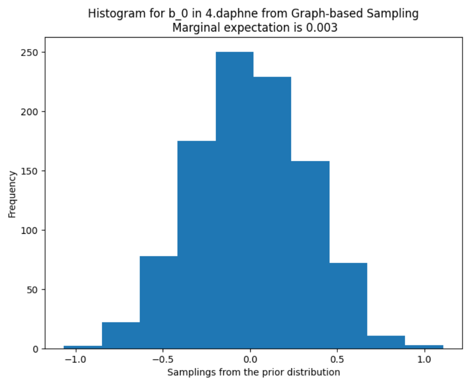
(a) Samples from the prior for HMM step 17

Figure 9: Histograms for 3.daphne

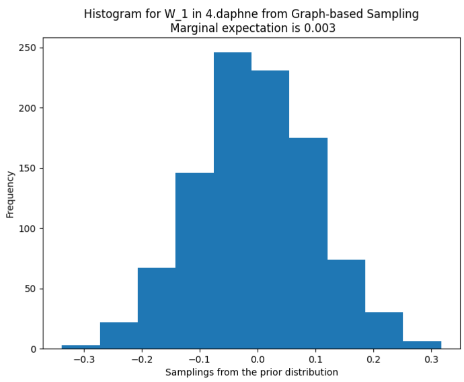
(d) Program 4): in 1000 samplings from this program, each return values is a 3-D array incorporating the estimations for W_0, b_0, W_1, b_1. W_0 is a 2-D array of size 10×1, b_0 is a 2-D array of size 10×1, W_1 is a 2-D array of size 10×10, and b_1 is a 2-D array of size 10×1.



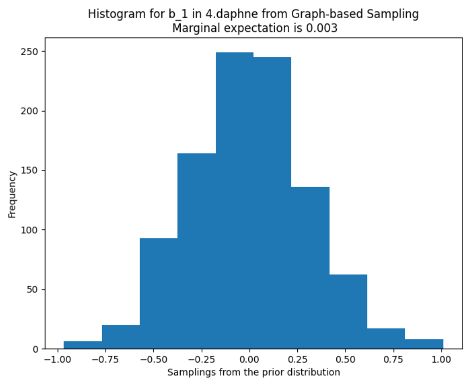
(a) Samples from the prior for W_0



(b) Samples from the prior for b_0



(c) Samples from the prior for W_1



(d) Samples from the prior for b_1

Figure 10: Histograms for 4.daphne

3. Tests in Evaluation-based sampling.py are passed:

```
/usr/local/bin/python3.8 "/Users/xiaoxuanliang/Desktop/CPSC 532W/HW/a2/evaluation_based_sampling.py"
Test passed
Test passed
Test passed
Test passed
Test passed
Test passed
Test passed
Test passed
Test passed
Test passed
Test passed
Test passed
Test passed
All deterministic tests passed
('normal', 5, 1.4142136)
p value 0.3508378292611998
Test passed
('beta', 2.0, 5.0)
p value 0.9153404165306447
Test passed
('exponential', 0.0, 5.0)
p value 0.12008347356762727
Test passed
('normal', 5.3, 3.2)
p value 0.29109421528517276
Test passed
('normalmix', 0.1, -1, 0.3, 0.9, 1, 0.3)
p value 0.3975385092697641
Test passed
('normal', 0, 1.44)
p value 0.9819723300694428
Test passed
All probabilistic tests passed
```

Tests in Graph-based sampling.py are passed:

```
/usr/local/bin/python3.8 "/Users/xiaoxuanliang/Desktop/CPSC 532W/HW/a2/graph_based_sampling.py"
Test passed
Test passed
Test passed
Test passed
Test passed
/Users/xiaoxuanliang/Desktop/CPSC 532W/HW/a2/primitives.py:101: UserWarning: To copy construct from a
  ast[i] = torch.tensor(ast[i])
Test passed
Test passed
/Users/xiaoxuanliang/Desktop/CPSC 532W/HW/a2/primitives.py:101: UserWarning: To copy construct from a
  ast[i] = torch.tensor(ast[i])
Test passed
Test passed
Test passed
/Users/xiaoxuanliang/Desktop/CPSC 532W/HW/a2/primitives.py:101: UserWarning: To copy construct from a
  ast[i] = torch.tensor(ast[i])
Test passed
Test passed
All deterministic tests passed
('normal', 5, 1.4142136)
p value 0.269506904271271
('beta', 2.0, 5.0)
p value 0.03478389512321067
('exponential', 0.0, 5.0)
p value 0.2535032616339814
('normal', 5.3, 3.2)
p value 0.6074611209109635
('normalmix', 0.1, -1, 0.3, 0.9, 1, 0.3)
p value 0.2127559950978155
('normal', 0, 1.44)
p value 0.3962568246398177
All probabilistic tests passed
```

4. Code snippets: Evaluation-based sampling.py:

```
def evaluate_program(ast):
    variables_dict = {}
    functions_dict = {}

    if type(ast[0]) is list and ast[0][0] == 'defn':
        function_expression = [ast[0][1], ast[0][2], ast[0][3]]
        functions_dict[ast[0][1]] = function_expression
        ast = ast[1]
    else:
        ast = ast[0]
    return evaluate_variable(ast, variables_dict, functions_dict)

primitives_operations = ['+', '-', '*', '/', 'sqrt', 'vector', 'hash-map', 'get', 'put', 'first', 'second', 'rest',
                         'last', 'append', '<', '<=', '>', '>=', '==', 'mat-transpose', 'mat-tanh', 'mat-mul', 'mat-add',
                         'mat-repmat']
distribution_types = ['normal', 'beta', 'exponential', 'uniform', 'discrete']
condition_types = ['sample', 'let', 'if', 'defn', 'observe']
```

Figure 11: Function evaluate˙program

```
def evaluate_variable(ast, variables_dict, functions_dict):
    if type(ast) is not list:
        if ast in primitives_operations:
            return ast
        elif type(ast) is torch.Tensor:
            return ast
        elif type(ast) is int:
            return torch.tensor(ast)
        elif type(ast) is float:
            return torch.tensor(ast)
        elif ast in distribution_types:
            return ast
        elif ast in variables_dict:
            return variables_dict[ast]
        elif ast in functions_dict:
            return functions_dict[ast]
        elif ast is None:
            return None

    elif type(ast) is list:
        if ast[0] in condition_types:
            return conditions_evaluation(ast, variables_dict, functions_dict)
        else:
            sub_ast = []
            for elem in ast:
                elem = evaluate_variable(elem, variables_dict, functions_dict)
                sub_ast.append(elem)
            if sub_ast[0] in primitives_operations:
                return primitives_evaluation(sub_ast)
            elif sub_ast[0] in distribution_types:
                return distributions_evaluation(sub_ast)
            elif type(sub_ast[0]) is list and sub_ast[0][0] in functions_dict:
                variables = sub_ast[0][1]
                values = sub_ast[1:]
                for i in range(len(variables)):
                    variables_dict[variables[i]] = values[i]
                return evaluate_variable(sub_ast[0][2], variables_dict, functions_dict)
```

Figure 12: Helper function for evaluation˙program

```python
def conditions_evaluation(ast, variables_dict, functions_dict):
    if ast[0] == 'sample':
        object = evaluate_variable(ast[1], variables_dict, functions_dict)
        sample = object.sample()
        return sample
    elif ast[0] == 'let':
        variable_value = evaluate_variable(ast[1][1], variables_dict, functions_dict)
        variables_dict[ast[1][0]] = variable_value
        return evaluate_variable(ast[2], variables_dict, functions_dict)

    elif ast[0] == 'if':
        boolean = evaluate_variable(ast[1], variables_dict, functions_dict)
        if boolean:
            variable_type = evaluate_variable(ast[2], variables_dict, functions_dict)
            return variable_type
        else:
            variable_type = evaluate_variable(ast[3], variables_dict, functions_dict)
            return variable_type

    elif ast[0] == 'observe':
        return evaluate_variable(None, variables_dict, functions_dict)
```

Figure 13: Helper function for evaluation˙program

Graph-based sampling.py:

```python
env = {'normal': dist.Normal,
       'beta': dist.Beta,
       'exponential': dist.Exponential,
       'uniform': dist.Uniform,
       'discrete': dist.Categorical,
       '+': torch.sum,
       '*': torch.multiply,
       '-': primitives.minus,
       '/': primitives.divide,
       'sqrt': torch.sqrt,
       'vector': primitives.vector,
       'hash-map': primitives.hashmap,
       'get': primitives.get,
       'put': primitives.put,
       'first': primitives.first,
       'second': primitives.second,
       'rest': primitives.rest,
       'last': primitives.last,
       'append': primitives.append,
       '<': primitives.smaller,
       '>': primitives.larger,
       'mat-transpose': primitives.mat_transpose,
       'mat-tanh': primitives.mat_tanh,
       'mat-mul': primitives.mat_mul,
       'mad-add': primitives.mat_add,
       'if': primitives.iff
       }


def deterministic_eval(exp):
    "Evaluation function for the deterministic target language of the graph based represen
    if type(exp) is list:
        op = exp[0]
        args = exp[1:]
        return env[op](*map(deterministic_eval, args))

    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 14: function maps and function deterministic˙eval

```python
def sample_from_joint(graph):

    "This function does ancestral sampling starting from the prior."

    vertices = graph[1]['V']
    edges = graph[1]['A']
    links = graph[1]['P']
    flows = graph[1]['Y']
    returnings = graph[2]
    variables_dict = {}

    unique_vertices = []
    degrees = {}
    for vertex in vertices:
        if vertex not in degrees:
            degrees[vertex] = 0
            unique_vertices.append(vertex)

    for vertex in unique_vertices:
        if vertex in edges:
            leaves = edges[vertex]
            for leave in leaves:
                degrees[leave] += 1

    ordering = []
    while len(ordering) != len(unique_vertices):
        for vertex in unique_vertices:
            degrees[vertex] -= 1
            if degrees[vertex] == -1:
                ordering.append(vertex)
```

Figure 15: Function sample˙from˙joint Part I

```python
    for vertex in ordering:
        link = links[vertex]
        if link[0] == 'sample*':
            record = evaluate(link[1], variables_dict)
            try:
                dist = deterministic_eval(record)
                value = dist.sample()
                variables_dict[vertex] = value
            except:
                ordering.append(vertex)
                pass

        elif link[0] == 'observe*':
            continue

    record = evaluate(returnings, variables_dict)
    return deterministic_eval(record)
```

Figure 16: Function sample˙from˙joint Part II

```python
def evaluate(exp, variables_dict):

    if type(exp) is not list:
        if exp in variables_dict:
            return variables_dict[exp]
        else:
            return exp
    else:
        record = []
        for sub_exp in exp:
            value = evaluate(sub_exp, variables_dict)
            record.append(value)
        return record
```

Figure 17: Helper function for sample˙from˙joint

Primitives.py:

```
def distributions_evaluation(ast):
    if ast[0] == 'normal':
        dist = distributions.normal.Normal(float(ast[1]), float(ast[2]))
        return dist
    elif ast[0] == 'beta':
        dist = distributions.beta.Beta(float(ast[1]), float(ast[2]))
        return dist
    elif ast[0] == 'exponential':
        dist = distributions.exponential.Exponential(float(ast[1]))
        return dist
    elif ast[0] == 'uniform':
        dist = distributions.uniform.Uniform(float(ast[1]), float(ast[2]))
        return dist
    elif ast[0] == 'discrete':
        for i in range(len(ast[1])):
            ast[1][i] = float(ast[1][i])
        dist = distributions.categorical.Categorical(ast[1])
        return dist
    else:
        print("need define distribution for: %s" % ast[0])
```

Figure 18: helper function

Primitives.py:

```python
def primitives_evaluation(ast):

    if ast[0] == '+':
        return torch.sum(torch.tensor(ast[1:]))
    elif ast[0] == '-':
        return ast[1] - (torch.sum(torch.tensor(ast[2:])))
    elif ast[0] == '*':
        return torch.prod(torch.tensor(ast[1:]))
    elif ast[0] == '/':
        return ast[1] / torch.prod(torch.tensor(ast[2:]))
    elif ast[0] == 'vector':
        try:
            return torch.stack(ast[1:])
        except:
            return ast[1:]
    elif ast[0] == 'sqrt':
        return torch.sqrt(torch.tensor([ast[1]]))
    elif ast[0] == 'hash-map':
        ast = np.reshape(np.array(ast[1:]), (-1, 2))
        ast = dict((ast[i][0], torch.tensor(ast[i][1])) for i in range(ast.shape[0]))
        return ast
    elif ast[0] == 'get':
        try:
            return ast[1][ast[2].item()]
        except:
            return ast[1][int(ast[2])]
    elif ast[0] == 'put':
        (ast[1])[int(ast[2])] = ast[3]
        return ast[1]
```

Figure 19: helper function

```python
elif ast[0] == 'first':
    return (ast[1])[0]
elif ast[0] == 'second':
    return (ast[1])[1]
elif ast[0] == 'rest':
    return (ast[1])[1:]
elif ast[0] == 'last':
    return (ast[1])[len(ast[1]) - 1]
elif ast[0] == 'append':
    return torch.cat((ast[1], torch.tensor([ast[2]])), dim=0)
elif ast[0] == '<':
    return ast[1] < ast[2]
elif ast[0] == '>':
    return ast[1] > ast[2]
elif ast[0] == 'mat-transpose':
    return ast[1].T
elif ast[0] == 'mat-tanh':
    return torch.tanh(ast[1])
elif ast[0] == 'mat-mul':
    ast[1] = ast[1].float()
    ast[2] = ast[2].float()
    return torch.matmul(ast[1], ast[2])
elif ast[0] == 'mat-add':
    return ast[1] + ast[2]
elif ast[0] == 'mat-repmat':
    return torch.tensor(ast[1]).repeat(int(ast[2]), int(ast[3]))
```

Figure 20: helper function