Hamiltonian Monte Carlo Inference for a First Order Probabilistic Programming Language

Yuan Zhou **Bradley Gram-Hansen** **Frank Wood**
Department of Engineering, University of Oxford

Abstract

In this work we describe how to implement a powerful Monte Carlo Markov Chain (MCMC) inference algorithm, that being Hamiltonian Monte Carlo (HMC), for a First Order Probabilistic Programing Language (FOPPL). We implement a compiler that transforms the program specified in a FOPPL, into Python code, to take advantage of the automatic differentiation package in the Python library Pytorch. We give examples of FOPPL programs, the compiler output and provide results of the HMC inference on those programs.

1 Introduction

Monte Carlo Markov Chain (MCMC) methods are a set of powerful inference algorithms (Berg and Billoire, 2008) that enable us to evaluate, model and analyze complicated probabilistic models. These include, but are not limited to, areas in machine learning such as Bayesian inference and learning, optimisation for finding the optimal hyper-parameters (Andrieu et al., 2003) and in natural systems, such as those found in Biology (Sorensen and Gianola, 2007) and Physics (Duane et al., 1987). However, as the dimensionality of the problem grows many MCMC methods, such as Metropolis-Hastings (Hastings, 1970), rejection and importance sampling, become ineffective at being able to generate independent samples effectively. This can be overcome in some instances by tuning particular parameters, or by choosing a better proposal distribution, but in practice this cannot always be done. One MCMC method that is able to circumvent this problem is Hamiltonian Monte Carlo (HMC) (Neal et al., 2011)(Duane et al., 1987), which takes inspiration from the physical world and uses a dynamical model to generate new proposals. This in turn enables us to explore larger spaces more effectively globally, rather than getting trapped in local regions.

This is important as choosing the right inference algorithm is critical for probabilistic programming languages (Tolpin et al., 2015) such as Anglican (Wood et al., 2014) and others, where we rely upon accurate inference and sampling procedures to evaluate our programs. Although, in practice there is no one inference or sampling algorithm to rule them all. Thus we rely on a combination of techniques to deal with discrete, finite continuous and infinite continuous parameter spaces (non-parametric models). To analyze more effectively a subset of the problems that Anglican and other higher order probabilistic programming languages (HOPPL) can, such as finite graphs, a FOPPL, a first order probabilistic programming language was formed.

In this work we make two contributions, we introduce a FOPPL compiler that transforms a FOPPL output, a finite graph, into Python code that takes advantage of the Automatic differentiation package within Pytorch (PyTorch, 2017) and an HMC implementation that correctly deals with conditional statements and finite continuous parameters. In section 2 we talk about a FOPPL, its syntax and primary use, in section 3 we introduce HMC, in section 4 we give examples of FOPPL programs and the results generated through HMC inference. In section 5 we summarize our findings and in section 6 we provide details of HMC implementation and some examples of the compiled FOPPL output.

2 FOPPL

The FOPPL(Wood Group, 2017) that we use throughout this paper has a syntax based on the `clojure` (Hickey, 2008) programming language and so one can implement conditional statements, like `if` and primitive operations such as `+,-,/,*`. However, unlike `clojure` there is no recursion and so you

Preliminary work. Under review by AISTATS 2018. Do not distribute.

cannot unroll a loop an infinite number of times. This is something that you could do, for instance, in a HOPPL. Despite this seemingly reduced functionality, a FOPPL demonstrates the wide ranging utility of probabilistic programming languages and it is flexible enough to allow one to implement complicated finite graphical models. This means that we can even write neural networks within a FOPPL (Wood Group, 2017). When we design a program in a FOPPL, a FOPPL outputs a finite directed graph that provides us with the joint distribution, from which it extracts the posterior of the model.

To construct a FOPPL program we have a set of syntactic statements at our disposal, such as `observe`, `sample`, `def`, `foppl-query`, `defn` and `let`. That allow us to `def`ine programs, and `foppl-query`ies within programs, by `let`ting you `defn`e functions and assign variables to `observe`d values and obtain probabilistic variables through `sample`ing from distribution objects. In order to compile a FOPPL program to Python, we must preserve the structure of our directed graph, like the ones presented in section 4. We map the edges, vertices and nodes accordingly so that the structure of the joint is preserved. We also take into account which values are our latent parameters and which values are observed, as within Pytorch, latent variables must be defined in a special way. This is because in order to perform HMC on continuous systems, the gradients of the latent parameters are needed and so we must predefine them as `torch.autograd.variable.Variable()` objects, with the following flag `requires_grad= True`.

To ensure that those variables, other constants and objects remain unique within our compilation, we introduce a unique naming procedure to ensure that our graphical model structure is preserved. To do this we set all latent parameters equal to `'x' + str(<unique_key>)` and for each latent parameter in which some primitive operation is performed with the original variable to create a new assignment, the new assignment will have an `'x'` label and a string of numbers that is unique, but will increase by one for each operation performed. All priors, likelihoods and posterior distributions are given the labels `'p' + str(<unique_key>)`, all constants and observed values are given the label `'c' + str(<unique_key>)` and if the constant is an observed datum, then the compiler sets `'y'+str(<unique_key>)='c'+str(<unique_key>)`.

As Pytorch does not support distribution objects, we had to define our own class of distribution objects, which directly map, for example, a `normal` object in a FOPPL, to a `normal` object in Python. For these distribution objects the label `'d' + str(<unique_key>)` is assigned. See the supplementary material, section 6 for examples of the compiled output.

3 Hamiltonian Monte Carlo

In a top level view HMC is a two step process. In step one, we define a Hamiltonian function in terms of the joint probability distribution of the model that we aim to perform inference on and in step two, HMC proposes new states generated via Hamiltonian dynamics for which we apply Metropolis updates. As the proposals are being generated via a physical process, we are able to comprehensively explore our model space (Neal et al., 2011). This is in part due to certain physical properties of the Hamiltonian itself, that make HMC a very powerful, multi-purpose inference algorithm. Adopting the notation from the machine learning literature, we use $\mathbf{x} \in \mathbb{R}^{n \times d}$ to represent the parameters of interest, our latent variables, rather than the typical θ and $\mathbf{q}$ in the HMC literature. Where n represents the number of parameters of interest and d is the dimension of the system.

The Hamiltonian of a physical system is defined completely in terms of the sets of points $(\mathbf{x}, \mathbf{p})$, the position and momentum variables respectively. These points span what is called the phase space, which formally is defined as the cotangent bundle $T^*\mathcal{M}$ of the configuration space $\mathcal{M}$. Simply put, we can imagine the phase space as a manifold that shows us both how our model evolves, with respect to $\mathbf{x}$ and $\mathbf{p}$ and how it is constrained in regards to the total energy within the system. Where the total energy of the system is given by the Hamiltonian, $H(\mathbf{x}, \mathbf{p})$. The Hamiltonian is the Legendre transform of the Lagrangian and is formally defined as $H(\mathbf{x}, \mathbf{p}) = K(\mathbf{p}) + U(\mathbf{x})$ where $K(\mathbf{p})$ represents our kinetic energy and $U(\mathbf{x})$ is the potential energy. The Legendre transform is given by $H(\mathbf{x}, \mathbf{p}) = \sum_{i=1}^{d} \dot{x}_i p_i - L(\mathbf{x}, \dot{\mathbf{x}}(\mathbf{x}, \mathbf{p}))$[1]. Thus, for simplicity, if we set $d = 1$, we can derive Hamilton's equations:

$$\frac{\partial H}{\partial p} = \dot{x} + p\frac{\partial \dot{x}}{\partial p} - \frac{\partial L}{\partial \dot{x}}\frac{\partial \dot{x}}{\partial p} = \dot{x} \tag{1}$$

and

$$\frac{\partial H}{\partial x} = p\frac{\partial \dot{x}}{\partial x} - \frac{\partial L}{\partial x} - \frac{\partial L}{\partial \dot{x}}\frac{\partial \dot{x}}{\partial x} = -\frac{\partial L}{\partial x} = -\dot{p} \tag{2}$$

from which we can vectorize for higher dimensions. It should be noted that the derivatives for the Lagrangian L, come from the Euler-Lagrange equations $\frac{d}{dt}\left(\frac{\partial L}{d\dot{x}}\right) = \frac{\partial L}{dx}$ and the Lagrangian itself, is just a reformulation of Newtonian mechanics.

Within the HMC framework the positions, $\mathbf{x}$, are the variables of interest, but in order to simulate Hamil-

[1] The $\dot{x}$ represents that the variable is being differentiated with respect to time.

tonian dynamics properly, for each $\mathbf{x}$ we must introduce an auxillary momentum variable $\mathbf{p}$. But what form should $\mathbf{p}$ take? Typically, $\mathbf{p}$ is sampled from a normal distribution $\mathbf{p} \sim \mathcal{N}(\mathbf{0}, \mathbb{1})$, although this need not be the case. In HMC, the potential energy $U(\mathbf{x})$ represents the negative log joint distribution of the model and is a key component within the algorithm. The kinetic energy $K(\mathbf{p})$, is typically taken to be the mean field approximation, which corresponds directly to the log of a centered Gaussian distribution $K(\mathbf{p}) = \frac{\mathbf{p}^T\mathbf{M}^{-1}\mathbf{p}}{2}$, where $\mathbf{M}$, the mass matrix, is a symmetric, positive definite and typically diagonal matrix. Although, again we need not choose this form of kinetic energy, when adapting HMC for discrete parameters, it is actually more beneficial to use a different kinetic function(Nishimura et al., 2017). Thus, if we are to use the standard kinetic energy, which we do for all our current models, then Hamilton's equations take the form $\dot{\mathbf{x}} = \frac{d\mathbf{x}}{dt} = [\mathbf{M}^{-1}\mathbf{p}]$ and $\dot{\mathbf{p}} = \frac{d\mathbf{p}}{dt} = -\nabla_{\mathbf{x}}U(\mathbf{x})$. To understand why the potential energy represents the joint, we take inspiration from the canonical distribution found in statistical mechanics $P(\mathbf{x},\mathbf{p}) = \frac{1}{Z}\exp\left(\frac{-E(\mathbf{x},\mathbf{p})}{\kappa_b T}\right)$, where Z is a normalization constant [2], E represents the total energy of the system, our Hamiltonian, and $\kappa_b = T = 1$ are constants that we define to be unit. Substituting the Hamiltonian H into the canonical distribution gives us the joint density of the system, not our model:

$$P(\mathbf{x},\mathbf{p}) = \frac{1}{Z}\exp(-U(\mathbf{x}))\exp(-K(\mathbf{p})) \tag{3}$$

It should be noted, that it is not always the case that Hamiltonian is separable, for example see Riemannian HMC (Girolami and Calderhead, 2011). As this expression is exponentiated and there are no implicit dependencies between the parameters, we can marginalize out the distribution of axillary momentum, leaving us with just the target distribution, the joint density, $P(\mathbf{x}) = \exp(-U(\mathbf{x}))$. In taking the ln of this, we find that:

$$U(\mathbf{x}) = -\ln P(\mathbf{x}) \tag{4}$$

and so the potential is entirely dependent on the form of the joint distribution. However, the expression $P(\mathbf{x})$ factorizes further via the product rule, into the product of a prior $p(\mathbf{x})$ for the parameters of interest and a likelihood $p(\mathbf{x}|\mathbf{y})$ given the observations $\mathbf{y}$:

$$U(\mathbf{x}) = -\log[p(\mathbf{x})p(\mathbf{x}|\mathbf{y})] \tag{5}$$

3.1 The Integrator

In order to implement HMC correctly we require an integrator that will enable us to solve the Hamilton's equations, equations (1-2). For an integrator to do this it must be both time reversible and volume preserving, as the flow of phase space is fixed. The time reversibility is due to the fact that no physical system should have a preferred direction of time, if I start at my initial conditions, I should eventually arrive back at those initial conditions. In order to ensure that our integrator is volume preserving, we require our integrator to be symplectic. This means that given a transformation $\mathbf{Q} \in Sp(2d,\mathbb{R})$ such that $\mathbf{Q}(\mathbf{p}_0,\mathbf{x}_0) \mapsto (\mathbf{x},\mathbf{p})$, which maps an initial state to some evolved state, for the transformation to preserve Hamilton's equations it must be a canonical transformation. But, this can be only true if given some matrix $J = \begin{pmatrix} \mathbf{0} & \mathbb{I} \\ -\mathbb{I} & \mathbf{0} \end{pmatrix}$ the relation $\mathbf{Q}^T J \mathbf{Q} = J$ is satisfied, which is only true if $\mathbf{Q}$ is symplectic. This can be proved as follows. If we have a transformation $R = R(\mathbf{x})$, then $\dot{R} = \mathbf{Q}^T J \mathbf{Q} \nabla_{\mathbf{Q}} H = J\nabla\mathbf{Q}H$[3], which is true if and only if $\mathbf{Q}^T J \mathbf{Q} = J$ and thus the transformation is symplectic.

A popular choice within the HMC literature is the Leapfrog integrator, equations (6-8). Not only does it satisfy the physical constraints of the model, but it has very small local $\mathcal{O}(\epsilon^2)$ and global errors $\mathcal{O}(\epsilon^3)$ with fast convergence (Neal et al., 2011). Although we shall be using the Leapfrog integrator throughout our current work, it should again be noted that this integrator can take alternative forms, for example see (Girolami and Calderhead, 2011)(Nishimura et al., 2017) and (Blanes and Iserles, 2012). The Leapfrog method enables us to generate new proposals given some initial state, that is, if we start with a state at $t = 0$ and then evaluate at a subsequent time $t+\epsilon, \ldots, t+N\epsilon$ we will generate a new state $(\mathbf{x}(t+N\epsilon), \mathbf{p}(t+N\epsilon))$ which will act as our new proposal. Where ϵ is the time step by which we increase and N is the total number of time steps.

$$\mathbf{p}(t+\frac{\epsilon}{2}) = \mathbf{p}(t) - \left(\frac{\epsilon}{2}\right)\nabla_{\mathbf{x}}U(\mathbf{x}(t)) \tag{6}$$

$$\mathbf{x}(t+\epsilon) = \mathbf{x}(t) + \epsilon\nabla_{\mathbf{p}}K(\mathbf{p}(t+\frac{\epsilon}{2})) \tag{7}$$

$$\mathbf{p}(t+\epsilon) = \mathbf{p}(t+\frac{\epsilon}{2}) - \left(\frac{\epsilon}{2}\right)\nabla_{\mathbf{x}}U(\mathbf{x}(t+\epsilon)) \tag{8}$$

3.2 The Algorithm

Before we provide the full HMC algorithm we shall briefly discuss how the second stage of HMC works, that is the Metropolis step. Starting with the current state $(\mathbf{x},\mathbf{p})$, Hamiltonian dynamics is simulated for L steps using the Leapfrog integrator, with a step size

[2] This is actually the partition function, which to those familiar with neural nets, will know this as the *softmax* function.

[3] We can write $\mathbf{z} = (\mathbf{x},\mathbf{p})$ and taking the vectorized form of equations (1 - 2), in terms of Laplacians, we can succinctly write Hamilton's equations as $\dot{\mathbf{z}} = J\nabla_{\mathbf{z}}H(\mathbf{z})$.

of ϵ. This generates a new proposed state and in order to decide whether we should accept or reject this proposal, (Duane et al., 1987) introduced the following acceptance (Metropolis) proposal:

$$\begin{aligned}&\min[1, \exp(-H(\mathbf{x}^*, \mathbf{p}^*) + H(\mathbf{x}, \mathbf{p})] = \\ &\min[1, \exp(-U(\mathbf{x}*) + U(\mathbf{x}) - K(\mathbf{p}^*) + K(\mathbf{p}))] \quad (9)\end{aligned}$$

where $(\mathbf{x}^*, \mathbf{p}^*)$ is the proposed state and $(\mathbf{x}, \mathbf{p})$ is the current state. A function that implements a run of the HMC algorithm is given in algorithm 1. If the proposed state is rejected, then the next state is the same as the current state and is counted again when calculating the expected value of some posterior. Theoretically speaking, in order to ensure that the proposal is symmetric, we should negate the momentum variables at the end of the trajectory, to ensure that the Metropolis proposal is symmetrical, which is needed for the acceptance probability to be valid. However, in practice we do not need to perform this negation since $K(\mathbf{p}) = K(-\mathbf{p})$ for the Gaussian momentum and after each iteration the momentum will be replaced before it is used again. Hence, we leave it out of algorithm 1. The authors issue a note of caution, as the potential is the negative of the log joint, we have equation (9). However, many implementations of HMC do not take this into account and so the proposal is in some instances defined as equation (9), but the potential is defined to be the positive of the log joint. Hence the proposal would be invalid. In our implementation, when the Hamiltonians are being computed we are dealing with the negative of the log joint, but during the leapfrog step we are dealing with the positive of the log joint, hence the sign changes in algorithm 1.

The parameters ϵ and L within algorithm 1 are parameters that need to be tuned. Likewise, if the form of the kinetic is taken to be the log of a Gaussian, $\mathbf{M}$ becomes another parameter that needs to be tuned correctly. Although, one could use Riemannian HMC (Girolami and Calderhead, 2011) to generate a mass matrix on the fly, which is based on the geometrical properties of the model that you are sampling from.

Algorithm 1 **Continuous Hamiltonian Monte Carlo MCMC**

```
procedure HMC($x_0$, $\epsilon$, $L$,$U$, $M$)
    for $m = 1$ to $M$ do
        $\mathbf{p}^0 \sim \mathcal{N}(0, \mathbb{1})$
        $(\mathbf{x}_0, \mathbf{p}_0) \leftarrow (\mathbf{x}^{(t)}, \mathbf{p}^{(t)})$
        $\mathbf{p}_0 \leftarrow \mathbf{p}_0 + \frac{\epsilon}{2}\nabla_{\mathbf{x}} U(\mathbf{x}_0)$
        for $i = 1$ to $L$ do
            $(\hat{\mathbf{x}}, \hat{\mathbf{p}}) \leftarrow \text{Leapfrog}(\mathbf{x}_0, \mathbf{p}_0, \epsilon)$
        end for
        $\mathbf{p}_L \leftarrow \mathbf{p}_L - \frac{\epsilon}{2}\nabla_{\mathbf{x}} U(\mathbf{x}_L)$
        $\alpha = \min\left\{1, \exp\left\{H(\mathbf{x}^{(t)}, \mathbf{p}^{(t)}) - H(\hat{\mathbf{x}}, \hat{\mathbf{p}})\right\}\right\}$
        $u \sim Uniform(0, 1)$
        if $u < \alpha$ then
            return $\mathbf{x}^{(t+1)} \leftarrow \hat{\mathbf{x}}$                ▷ Accept
        else
            return $\mathbf{x}^{(t+1)} \leftarrow \mathbf{x}^t$                ▷ Reject
        end if
    end for
    Leapfrog($\mathbf{x}$, $\mathbf{p}$, $\epsilon$)
    $\mathbf{x}_i \leftarrow \mathbf{x}_i + \epsilon\nabla_{\mathbf{p}} K(\mathbf{p}_i)$
    $\mathbf{p}_i \leftarrow \mathbf{p}_{i-1} + \frac{\epsilon}{2}\nabla_{\mathbf{x}} U(\mathbf{x}_{i-1})$
    return $\hat{\mathbf{x}}$ , $\hat{\mathbf{p}}$
end procedure
```

4 Example Programs and Experiments

4.1 Programs

We now present a few simple FOPPL programs and discuss both their statistical forms and the results generated via HMC inference. In all the graphical model plots, shaded circles represent observed parameters, un-shaded circles represent the latent variables, our parameters of interest, and diamonds represent a deterministic statement.

4.1.1 Conjugate Gaussian

The conjugate Gaussian, see figure (1), is a model in which we can analytically calculate the true posterior $p(x|y)$, as the product of two Gaussians is also a Gaussian. For our particular model we `sample` $x \sim \mathcal{N}(0, 1)$ and we `observe` $y = 7$, with likelihood $p(y|x) = \mathcal{N}(y|x, 1)$ and we aim to infer the mean of the posterior. The FOPPL program for figure (1) is as follows:

```
;;; conjugate_gaussian
(def conjgauss
  (foppl-query
    (let [x (sample (normal 0.0 1.0))]
      (observe (normal x 1.0) 7.0)
      x)))
```

In calculating the joint of the model, we arrive as the following $p(x|y) \propto p(x, y) = p(x)p(y|x) = \mathcal{N}(\bar{\mu}, \bar{\sigma}^2)$, where $\bar{\mu} = \left(\frac{\mu_0\sigma^2 + \sigma_0^2\sum_{i=1}^n y_i}{\sigma^2\sigma_0^2}\right)\bar{\sigma}^2$ and $\bar{\sigma}^2 = \left(\frac{1}{\sigma_0^2} + \frac{n}{\sigma^2}\right)^{-1}$, n is the total number of observed datum y, $\mu_0 = 0$, $\sigma_0^2 = 1$ are the prior mean and variance and $\sigma^2 = 1.0$ is the likelihood variance. Thus, the true mean that our model should infer is $\bar{\mu} = 5$. This is

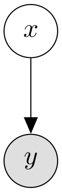


Figure 1: The graphical model for the conjugate Gaussian

indeed the value that we get as can be seen via figures (4a-4c).

4.1.2 Conditional If

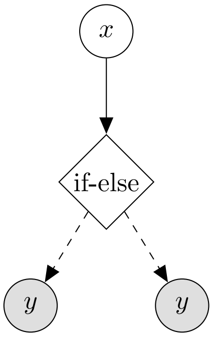


Figure 2: The graphical model for condition if

The conditional `if` statement, see figure (2), is typically challenging to implement for gradient based methods, due to the branching effects that occur at the condition. For trace based samplers, such as importance sampling, this is not a problem as multiple traces are recorded for each path taken. However, in higher dimensions these types of samplers are heavily impaired, which is why we need additional MCMC methods to be able to perform efficiently in higher dimensions. For the purpose of this paper we provide a low dimensional model, so that we can easily visualize the output of our HMC implementation of the conditional program. In a FOPPL we would write a simple conditional `if` program as follows:

```
;;; conditional if
(def condif
  (foppl-query
    (let [x (sample (normal 0.0 1.0))]
      (if (> x 0)
          (observe (normal 1.0 1.0) 1.0)
          (observe (normal -1 1.0) 1.0))
      x)))
```

In this model, we have a joint distribution of the form $p(x|y) \propto p(x,y) = \mathcal{N}(0,1)\mathcal{N}(y = 1|1,1)^{\mathbb{I}(x>0)}\mathcal{N}(y = 1|-1,1)^{\mathbb{I}(x<0)}$, where $\mathbb{I}(\cdot)$ represents the indicator function. The empirical mean that we generate for this model is 0.58 and if you plot the unnormalized joint, the shape is the same, but the density values are of course different, see figures (4d-4f). This implies that we can be confident in the results generated by the HMC. For additional verification the same program was run in Anglican Wood et al. (2014) to ensure that it generated the same results.

4.1.3 Linear Regression

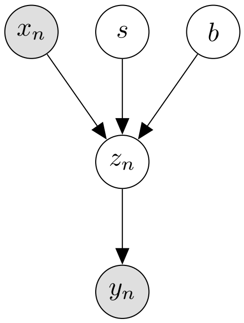


Figure 3: The graphical model for Linear regression

In this problem we have a set of points $\{(x_n, y_n)\}$ and we wish to infer the equation of a line in a Bayesian manner, such that the line goes through all of those points. See figure (3) for a graphical model of the model. To do this, we must infer the slope and the bias of the line, which is done by placing priors on our parameters of interest. The FOPPL program corresponding to figure (3) can be written as follows:

```
;;; linear regression
(def linreg
  (foppl-query
    (defn observe-data [_ data slope bias]
      (let [xn (first data)
            yn (second data)
            zn (+ (* slope xn) bias)]
        (observe (normal zn 1.0) yn)
        (rest (rest data))))

    (let [slope (sample (normal 0.0 10.0))
          bias  (sample (normal 0.0 10.0))
          data (vector
                 1.0 2.1 2.0 3.9 3.0 5.3)]
      (loop 3 data observe-data slope bias)
       (vector slope bias))))
```

In this model for both the slope and bias, we `sample` from $\mathcal{N}(0, 10.0)$ and we state that the likelihood of the model is of the form of a conditional Gaussian $\mathcal{N}(y_n|z_n, 1.0)$ with a mean conditioned on our sampled lines, given the observed points. Therefore, we can construct the joint for this model as $p(s, b, z_n|x_n, y_n) \propto p(x_n, s, b, z_n, y_n) = p(s)p(b)p(y_n|z_n, 1) = \mathcal{N}(0, 5^2)\mathcal{N}(y_n|z_n, 1.0)$. We can analytically calculate the equation of a straight line by using the formula $\left(\frac{y-y_*}{x-x_*}\right) = s$, which leads to $y = sx + b$. From which we find that the true slope $s = 1.6$ and the bias $b = 0.5$. We again see from figures (4g-4i), that the HMC correctly calculates the true values of the parameters.

4.2 Experimental Results

Here we present the results of running HMC inference on our FOPPL programs via the compiled Python output. All plots were generated by running HMC until 1000 samples had been collected, we use no burn in period and so all results presented are of all samples generated. See figure (4).

5 Discussion

In this paper we have presented the HMC algorithm, we have introduced the reader to a FOPPL and have explained how we merge together FOPPL programs and HMC inference. We are currently expanding the HMC algorithm for use with both discrete and continuous parameters. This does require the design of new integrator, which has been proposed by (Nishimura et al., 2017), however, it has never been implemented for a probabilistic programming language, to the authors knowledge.

6 Supplementary Material

In the supplementary material we show some examples of the compiler output for the conjugate Gaussian and linear regression models. We also briefly discuss the implementation of the HMC algorithm in Python. All code used to generate results can be found on github[4]

6.1 Compiler Output

Here we present some examples of the FOPPL compiler output for Python. The following output is for the conjugate Gaussian model:

```
c24039= torch.Tensor([1.0])
c24040= torch.Tensor([2.0])
# prior
d24041 = Normal(c24039, c24040)
# sample
x22542 = Variable(d24041.sample().data,\
  requires_grad = True)
# log prior
p24042 = x24041.logpdf( x22542)
c24043= torch.Tensor([3.0])
# likelihood
d24044 = Normal(x22542, c24043)
c24045= torch.Tensor([7.0])
# obs
y22543 = c24045
# log likelihood
p24046 = d24044.logpdf( y22543)
# log joint
p24047 = Variable.add(p24042,p24046)

return p24047, x22542
```

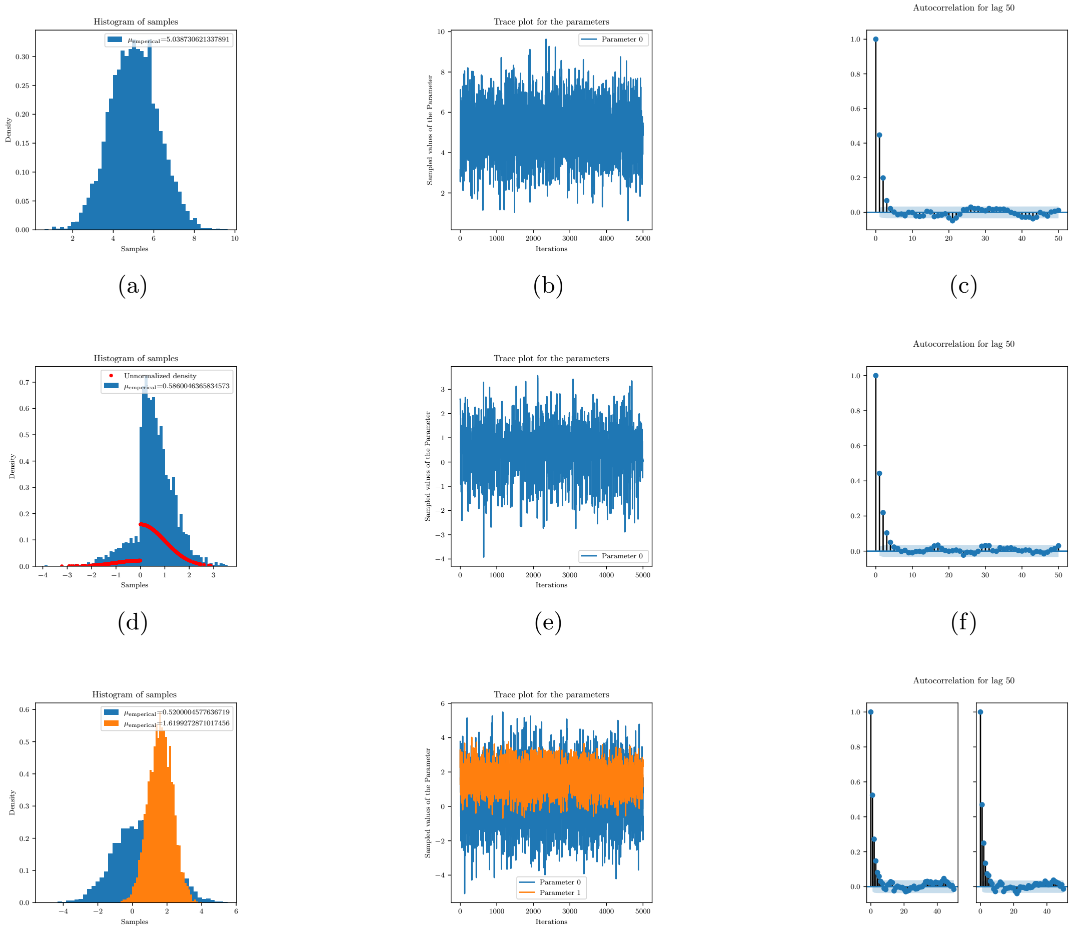
This is the output for the linear regression model:

```
# FOPPL compiler output
c23582= torch.Tensor([0.0])
c23583= torch.Tensor([10.0])
d23584 = Normal(c23582, c23583)
#sample
x23474 =  d23584.sample()
#prior
p23585 = d23584.logpdf( x23474)
c23586= torch.Tensor([0.0])
c23587= torch.Tensor([10.0])
d23588 = Normal(c23586, c23587)
#sample
x23471 = d23588.sample()
#prior
p23589 = d23588.logpdf( x23471)
c23590= torch.Tensor([1.0])
x23591 = torch.mul(x23471.data, c23590)
x23592 = torch.add(x23591,x23474.data)
c23593= torch.Tensor([1.0])
d23594 = Normal(x23592,  c23593)
#obs, log likelihood
c23595= torch.Tensor([2.1])
y23481 = c23595
p23596 = d23594.logpdf( y23481)

c23597= torch.Tensor([2.0])
x23598 = torch.mul(x23471, c23597)
x23599 = torch.add(x23598,x23474)
c23600= torch.Tensor([1.0])
d23601 = Normal(x23599,  c23600)
#obs, log likelihood
c23602= torch.Tensor([3.9])
y23502 = c23602
p23603 = d23601.logpdf( y23502)

c23604= torch.Tensor([3.0])
```

[4] https://github.com/Bjgh/Project_notes



(a) (b) (c)

(d) (e) (f)

(g) (h) (i) Left: bias . Right: slope.

Figure 4: Each row corresponds to the inference output of each model and provides a histogram of the samples, the trace of the samples for each parameter of interest in the model and the autocorrelation between the samples for a lag $l = 50$. *Top row* represents the conjugate Gaussian model. *Middle row* represents the conditional if model. *Bottom row* represents the linear regression model with parameter 0, in blue, representing the bias and parameter 1, in orange, representing the slope of the line.

```
x23605 = torch.mul(x23471, c23604)
x23606 = torch.add(x23605,x23474)
c23607= torch.Tensor([1.0])
d23608 = Normal(x23606,  c23607)
#obs, log likelihood
c23609= torch.Tensor([5.3])
y23527 = c23609
p23610 = d23608.log_pdf( y23527)
# log joint
p23611 = torch.add([p23585,p23589,/
                p23596,p23603,p23610])
# return E from the model
x23612 = [x23471,x23474]

return p23611, x23612
```

6.2 HMC Implementation

Below we present a top level view of the HMC sampler, without going into explicit details. Although as stated previously, all code can be found on our github page. The HMC can take a variety of inputs, depending on the users specification. The most important input is the `program()` object, which is a class of the compiled output. The `program()` itself inherits directly from a base class which performs the differentiation of the log joint distribution, with respect to the parameters of interest. Within the `program()` class we have a method `program.generate()` which simulates the chosen model once and intialises the latent variables within the model. This returns all initial values such as the log joint, the initial state, the gradient of the log joint and the number of parameters of interest. It then calls another method within class program.eval(), which no longer performs the sampling steps and instead evaluates the model at various values generated via the Leapfrog function. The rest of the program follows from algorithm 1.

```
import torch
import numpy as np
from Utils.kinetic import Kinetic
from Utils.integrator import Integrator
from Utils.metropolis_step import Metropolis

class HMCsampler():

  def __init__(self, program, burn_in= 100, \
          n_samples= 1000, M= None,\
    min_step= None, max_step= None,\
    min_traj= None, max_traj= None):
    self.burn_in    = burn_in
    self.n_samples  = n_samples
    self.M          = M
    # External dependencies
    self.potential  = program()
    self.integrator= Integrator(\
    self.potential,min_step,\
             max_step, min_traj, max_traj)


  def run_sampler(self):
    print(' The sampler is now running')
    logjoint_init, values_init, grad_init,\
     dim = self.potential.generate()
    metropolis = Metropolis(\
     self.potential,self.integrator, self.M)
    temp,count = metropolis.acceptance(\
     values_init,logjoint_init, grad_init)
    samples =\
     Variable(torch.zeros(self.n_samples,dim))
    samples[0]= temp.data.t()


    # Then run for loop from 2:n_samples
    for i in range(self.n_samples-1):
      logjoint_init, grad_init = \
      self.potential.eval(temp,grad_loop= True)
      temp, count = metropolis.acceptance(temp,\
       logjoint_init,grad_init)
      samples[i + 1, :] = temp.data.t()

    # Basic summary statistics
    target_acceptance =  count / (self.n_samples)
    samples_reduced   = samples[self.burn_in:, :]
    mean = torch.mean(samples_reduced,dim=0,\
     keepdim= True)

    return samples, samples_reduced,  mean
```

Acknowledgements

The authors acknowledge Yuan for her help with designing the compiler.

References

Andrieu, C., De Freitas, N., Doucet, A., and Jordan, M. I. (2003). An introduction to mcmc for machine learning. *Machine learning*.

Berg, B. A. and Billoire, A. (2008). *Markov chain monte carlo simulations*. Wiley Online Library.

Blanes, S. and Iserles, A. (2012). Explicit adaptive symplectic integrators for solving hamiltonian systems. *Celestial Mechanics and Dynamical Astronomy*.

Duane, S., Kennedy, A. D., Pendleton, B. J., and Roweth, D. (1987). Hybrid monte carlo. *Physics letters B*.

Girolami, M. and Calderhead, B. (2011). Riemann manifold langevin and hamiltonian monte carlo methods. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*.

Hastings, W. K. (1970). Monte carlo sampling methods using markov chains and their applications. *Biometrika*.

Hickey, R. (2008). The clojure programming language. *Proceedings of the 2008 Symposium on Dynamic Languages*.

Neal, R. M. et al. (2011). Mcmc using hamiltonian dynamics. *Handbook of Markov Chain Monte Carlo*.

Nishimura, A., Dunson, D., and Lu, J. (2017). Discontinuous hamiltonian monte carlo for sampling discrete parameters. *arXiv preprint arXiv:1705.08510*.

PyTorch (2017). Pytorch, automatic differentiation package. https://github.com/pytorch/pytorch.

Sorensen, D. and Gianola, D. (2007). *Likelihood, Bayesian, and MCMC methods in quantitative genetics*. Springer Science & Business Media.

Tolpin, D., van de Meent, J.-W., and Wood, F. (2015). Probabilistic programming in anglican. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer.

Wood, F., Meent, J. W., and Mansinghka, V. (2014). A new approach to probabilistic programming inference. In *Artificial Intelligence and Statistics*.

Wood Group, O. (2017). *In Prep* a tutorial on probabilistic programming. *Foundations and Trends in Machine Learning*.