

---

# Hamiltonian Monte Carlo Inference for a First Order Probabilistic Programming Language

---

Bradley Gram-Hansen

Department of Engineering, University of Oxford

Frank Wood

## Abstract

The Abstract paragraph should be indented 0.25 inch (1.5 picas) on both left and right-hand margins. Use 10 point type, with a vertical spacing of 11 points. The **Abstract** heading must be centered, bold, and in point size 12. Two line spaces precede the Abstract. The Abstract must be limited to one paragraph.

## 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 samples effectively. This can be overcome in some instances by tuning particular parameters, or generating 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.

---

Proceedings of the 20<sup>th</sup> International Conference on Artificial Intelligence and Statistics (AISTATS) 2017, Fort Lauderdale, Florida, USA. JMLR: W&CP volume 54. Copyright 2017 by the author(s).

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 parameter spaces (non-parametric models). To analyze more effectively a subset of the problems that Anglican can, such as finite graphs, FOPPL, a first order probabilistic programming language was constructed so that we could take advantage of fast inference algorithms, such as HMC.

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 that correctly deals with conditional statements and finite continuous parameters. In section 2 we talk about 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 provide details of HMC implementation and some examples of the compiled FOPPL output.

## 2 FOPPL

In this section include some stuff about FOPPL, what is it, the syntax etc.

## 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 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 notations 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  $\theta$  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 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 and the Legendre transform is given by  $H(\mathbf{x}, \mathbf{p}) = \sum_{i=1}^d \dot{x}_i p_i - L(\mathbf{x}, \dot{\mathbf{x}}(\mathbf{x}, \mathbf{p}))$ <sup>1</sup>. Thus, for simplicity, if we set  $d = 1$ , we can derive Hamiltonian equations:

$$\frac{\partial H}{\partial p} = \dot{x} + p \frac{\partial \dot{x}}{\partial p} - \frac{\partial L}{\partial \dot{x}} \frac{\partial \dot{x}}{\partial p} = \dot{x} \quad (1)$$

and

$$\frac{\partial H}{\partial x} = p \frac{\partial \dot{x}}{\partial x} - \frac{\partial L}{\partial x} - \frac{\partial L}{\partial \dot{x}} \frac{\partial \dot{x}}{\partial x} = -\frac{\partial L}{\partial x} = -\dot{p} \quad (2)$$

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  $\frac{d}{dt} \left( \frac{\partial L}{\partial \dot{x}} \right) = \frac{\partial L}{\partial x}$  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 Hamiltonian 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  $\mathbf{p} \sim \mathcal{N}(\mathbf{0}, \mathbf{1})$ , although this need not be the case. In HMC, the potential energy  $U(\mathbf{x})$  represents the negative of the 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  $\mathbf{M}$  is a symmetric, positive definite and typically diagonal matrix. Although,

<sup>1</sup>The  $\dot{x}$  represents that the variable is being differentiated with respect to time.

again we need not choose this form of kinetic energy and in some cases, such as those when we are dealing with 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  $\dot{\mathbf{p}} = \frac{d\mathbf{p}}{dt} = -\frac{\partial U}{\partial \mathbf{x}}$ .

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<sup>2</sup>,  $E$  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:

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

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,  $P(\mathbf{x}) = \exp(-U(\mathbf{x}))$ , from which can take the ln to get:

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

and so the potential is entirely dependent on the choice of the joint distribution. **CHECKPOINT** as this expression factorizes in to two independent probability distributions. We characterise the posterior distribution for the model parameters using the potential energy function:

$$U(x) = -\log[\pi(x)L(x|D)] \quad (5)$$

where  $\pi(x)$  is the prior distribution, and  $L(x|D)$  is the likelihood, not the Lagrangian, of the given data  $D$ .

### 3.1 The Integrator

**Talk about the properties of integrators, why they are important and why we are using the leapfrog integrator.** The leapfrog method enables us to discretise Hamiltons equations and as it is a valid integrator, it allows us to numerically solve Hamiltons equations ?? and ??. In doing so, we generate new proposals given some initial state We start with a state at  $t = 0$  and then evaluate at a subsequent time  $t +$

<sup>2</sup>This is actually the partition function, which to those familiar with neural nets, will know this as the *softmax* function.

$\epsilon, \dots, t + n\epsilon$ , where  $\epsilon$  is the step in which we increase and  $n$  is the number of time steps.

$$p_i(t + \frac{\epsilon}{2}) = p_i(t) - \left(\frac{\epsilon}{2}\right) \frac{\partial U(x(t))}{\partial x_i} \quad (6)$$

$$x_i(t + \epsilon) = x_i(t) + \epsilon \frac{\partial K(p(t + \frac{\epsilon}{2}))}{\partial p_i} \quad (7)$$

$$p_i(t + \epsilon) = p_i(t + \frac{\epsilon}{2}) - \left(\frac{\epsilon}{2}\right) \frac{\partial U(x(t + \epsilon))}{\partial x_i} \quad (8)$$

3

For the leapfrog method the local error, error after one step, is of  $\mathcal{O}(\epsilon^2)$  and a global error, error after simulating for some fixed time interval  $s$ , which requires  $\frac{s}{\epsilon}$  is  $\mathcal{O}(\epsilon^3)$  Neals implementation of the HMC can only be used to sample from continuous distributions on  $\mathbb{R}^d$  for which a density function can be evaluated.

We must be able to compute the partial derivative of the log of the density function, the joint. HMC samples from the canonical distribution for  $x$  and  $p$ .  $x$  has the distribution of interest as specified by the potential  $U(x)$ . The distribution of the  $p$ 's can be chosen by us and are independent of the  $x$ 's. The  $p$  components are specified to be independent, with component  $p_i$  having variance  $m_i$ . The kinetic energy  $K(p) = \sum_{i=1}^d \frac{p_i^2}{2m_i}(x(t + \epsilon))$

### 3.1.1 Properties of the Integrator

Reversibility

### 3.1.2 The Algorithm

1. Step 1: Changes only the momentum
2. Step 2: May change both position and momentum

Both steps leave the canonical distribution of  $(x, p)$  invariant, hence the distribution remains invariant.

In **Step 1** we first draw the  $p_i$  randomly from their Gaussian distribution independently of the current values of the position values. In **Step 2** a Metropolis update is performed, using the Hamiltonian dynamics to propose a new state. Starting with the current state  $(x, p)$ , Hamiltonian dynamics is simulated for  $L$  steps using the leapfrog method, with a stepsize of  $\epsilon$ .  $L$  and  $\epsilon$  are parameters of the model that need to be tuned. The momentum variables at the end of this  $L$ -step trajectory are then negated, giving a proposed state  $(x^*, p^*)$ . This proposed state is accepted as the next

state in the Markov Chain with probability:

$$\min[1, \exp(-H(x^*, p^*) + H(p, x))] = \min[1, \exp(-U(x^*) + U(x) - K(p^*) + K(p))] \quad (9)$$

If the proposed state is rejected, then the next state

---

### Algorithm 1 Continuous Hamiltonian Monte Carlo MCMC

---

```

1: procedure HMC( $x_0, \epsilon, L, U, M$ )
2:   for  $m = 1$  to  $M$  do
3:      $p^0 \sim \mathcal{N}(0, \mathbf{I})$ 
4:      $x^m \leftarrow x^{m-1}$ 
5:      $x' \leftarrow x^{m-1}$ 
6:      $p' \leftarrow p^0$ 
7:     for  $i = 1$  to  $L$  do
8:        $x', p \leftarrow \text{Leapfrog}(x', p', \epsilon)$ 
9:     end for
10:     $\alpha = \min \left\{ 1, \frac{\exp\{-U(x') - K(p')\}}{\exp\{U(x^{m-1}) - K(p^0)\}} \right\}$ 
11:     $u \sim \text{Uniform}(0, 1)$ 
12:    if  $u < \alpha$  then
13:      return  $x^m \leftarrow x', p^m \leftarrow -p \triangleright \text{Accept}$ 
14:    else
15:      return  $x^m \leftarrow x^{m-1}, p^m \leftarrow p^0 \triangleright \text{Reject}$ 
16:    end if
17:  end for
18:  Leapfrog( $x, p, \epsilon$ )
19:   $p' \leftarrow p - \frac{\epsilon}{2} \nabla_x U(x) \triangleright \text{Half step for momentum}$ 
20:   $x' \leftarrow x + \epsilon \nabla_p K(p') \triangleright \text{Full step for position}$ 
21:   $p' \leftarrow p - \frac{\epsilon}{2} \nabla_x U(x') \triangleright \text{Half step for momentum}$ 
22:  return  $x', p'$ 
23: end procedure

```

---

is the same as the current state and is counted again when calculating the expected value of some function. The negation of the momentum variables at the end of the trajectory makes the Metropolis proposal symmetrical, as needed for the acceptance probability above to be valid. This negation need not be done in practice, since  $K(p) = K(-p)$ , and the momentum will be replaced before it is used again, in the first step of the next iteration. Where  $U = -\log(\pi(x))$ ,  $M$  is the number of samples we wish to take,  $(x', p')$  is the generated proposal and we propose setting  $x^m = x'$  and  $p^m = -p'$  and then accept or reject this proposal according to the Metropolis update step. A function that implements a single iteration of the HMC algorithm is given in algorithm 2. There are three additional functions within this iteration:  $U$ , which returns the potential energy given a value for  $x$ ,  $\nabla U$ , which returns the vector of partial derivatives of  $U$  given  $x$  and  $\nabla K$ , which returns the vector of partial derivatives of  $K$  given  $p$ . Other arguments are the stepsize,  $\epsilon$ , for leapfrog steps, the number of leapfrog steps in the trajectory,  $L$ , and the current position,  $x_{\text{current}}$ , that the

---

<sup>3</sup>For the usual choice of kinetic energy, we have  $\frac{\partial K(p + \frac{\epsilon}{2})}{\partial p_i} = \frac{p_i(t + \frac{\epsilon}{2})}{m_i}$

trajectory starts from. Momentum variables are sampled within this function, and discarded at the end, with only the next position being returned.

## 4 Example Programs and Experiments

### 4.1 Programs

We now present a few simple FOPPL programs and discuss both their statistical and syntactic structures.

### 4.2 Conjugate Gaussian

The conjugate Gaussian, is a model in which we can analytically calculate the true posterior  $p(x|y) \propto p(x, y)$ , 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  $p(y|x) = \mathcal{N}(y|x, 1)$ . In FOPPL we have the following output:

```
;;; conjugate_gaussian
(def src0
  (foppl-query
    (let [x (sample (normal 0.0 1.0))]
      (observe (normal x 1.0) 7.0)
      x)))
```



Figure 1: conditional gauss.

Thus, the joint  $p(x, y) \propto p(x|y) = p(x)p(y|x) = \mathcal{N}(\bar{\mu}, \bar{\sigma}^2)$ , 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  $\bar{\sigma}^2 = \left( \frac{1}{\sigma_0^2} + \frac{n}{\sigma^2} \right)^{-1}$ . Where in this model  $n$  is the total number of observed datum  $y$ ,  $\mu_0 = 0$  and  $\sigma_0^2 = 1$  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  $\bar{\mu} = 5$ .

### 4.3 Conditional If

```
;;; conditional if
(def if-src
  (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)))
```

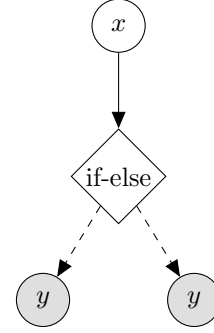


Figure 2: Conditional if graphical model

The `if` statement 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. 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. If we are to analytically calculate the mean, we have to look at the case where  $x < 0$  and the case where  $x > 0$ . Using the sum rule the evidence can be found by  $p(y) = \int p(x, y) dx$ , which we analytically calculate to be,  $p_{x<0}(y) = -\mathcal{N}(y = 1|-1, 1)\sqrt{2\pi}$  and for  $x > 0$  we have  $p_{x>0}(y) = \mathcal{N}(y = 1|1, 1)\sqrt{2\pi}$ . Thus, by Bayes rule, the full posterior is given as  $p(x|y) = \frac{p(y|x)p(x)}{p(y)} = \frac{\mathcal{N}(0, 1) \mathcal{N}(y=1|1, 1)^{\mathbb{I}(x>0)} \mathcal{N}(y=1|-1, 1)^{\mathbb{I}(x<0)}}{\sqrt{(2\pi)(\mathcal{N}(y=1|-1, 1)^{\mathbb{I}(x<0)} + \mathcal{N}(y=1|1, 1)^{\mathbb{I}(x>0)})}}$  and so for  $x < 0$ ,  $\mathbb{E}[x < 0|y] = \frac{-1}{\sqrt{2\pi}}$  and for  $x > 0$  we have  $\mathbb{E}[x > 0|y] = \frac{1}{\sqrt{2\pi}}$ .

### 4.4 Linear Regression

```
;;; linear regression
(def lr-src
  (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)))
```

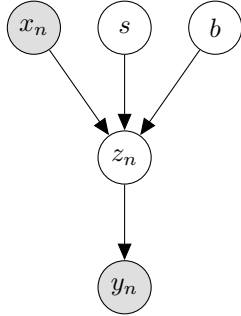


Figure 3: Linear regression model as a graph.

In this problem we have a set of points  $(x, y)$  and we wish to infer equation of a line in a Bayesian manner, that goes through all of those points. 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 and observations. We can analytically calculate the equation of a straight line by using the formula  $\left(\frac{y-y_*}{x-x_*}\right) = m$ , which leads to  $y = mx + b$ . From this we find that the true slope  $m = 1.6$  and the bias  $b = 0.5$ .

#### 4.5 Experimental Results

### 5 Supplementary Material

In the supplementary material we show sum examples of the compiler output for the various models shown and we also discuss the implementation of the HMC algorithm itself in pytorch.

```
# FOPPL compiler output
c23582= torch.Tensor([0.0])
c23583= torch.Tensor([10.0])
x23584 = Normal(c23582, c23583)
#sample
x23474 = x23584.sample()
#prior
p23585 = x23584.logpdf( x23474)
c23586= torch.Tensor([0.0])
c23587= torch.Tensor([10.0])
x23588 = Normal(c23586, c23587)
#sample
x23471 = x23588.sample()
#prior
p23589 = x23588.logpdf( x23471)
c23590= torch.Tensor([1.0])
x23591 = torch.mul(x23471.data, c23590)
x23592 = torch.add(x23591,x23474.data)
```

```
c23593= torch.Tensor([1.0])
x23594 = Normal(x23592, c23593)
#obs, log likelihood
c23595= torch.Tensor([2.1])
y23481 = c23595
p23596 = x23594.logpdf( y23481)

c23597= torch.Tensor([2.0])
x23598 = torch.mul(x23471, c23597)
x23599 = torch.add(x23598,x23474)
c23600= torch.Tensor([1.0])
x23601 = Normal(x23599, c23600)
#obs, log likelihood
c23602= torch.Tensor([3.9])
y23502 = c23602
p23603 = x23601.logpdf( y23502)

c23604= torch.Tensor([3.0])
x23605 = torch.mul(x23471, c23604)
x23606 = torch.add(x23605,x23474)
c23607= torch.Tensor([1.0])
x23608 = Normal(x23606, c23607)
#obs, log likelihood
c23609= torch.Tensor([5.3])
y23527 = c23609
p23610 = x23608.log_pdf( y23527)
p23611 = torch.add([p23585,p23589,/
                    p23596,p23603,p23610])
# return E from the model
x23612 = [x23471,x23474]
```

```
import torch
import numpy as np
import importlib
from torch.autograd import Variable
from Utils.core import VariableCast
from Utils.kinetic import Kinetic
from Utils.integrator import Integrator
from Utils.metropolis_step import Metropolis

class HMCsampler():

    def __init__(self, burn_in= 100, \
                 n_samples= 1000, model = 'conjgauss',\
                 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
        self.model = model
        # External dependencies
        program = \
            getattr(importlib.\
                    import_module('Utils.program'), model)
```

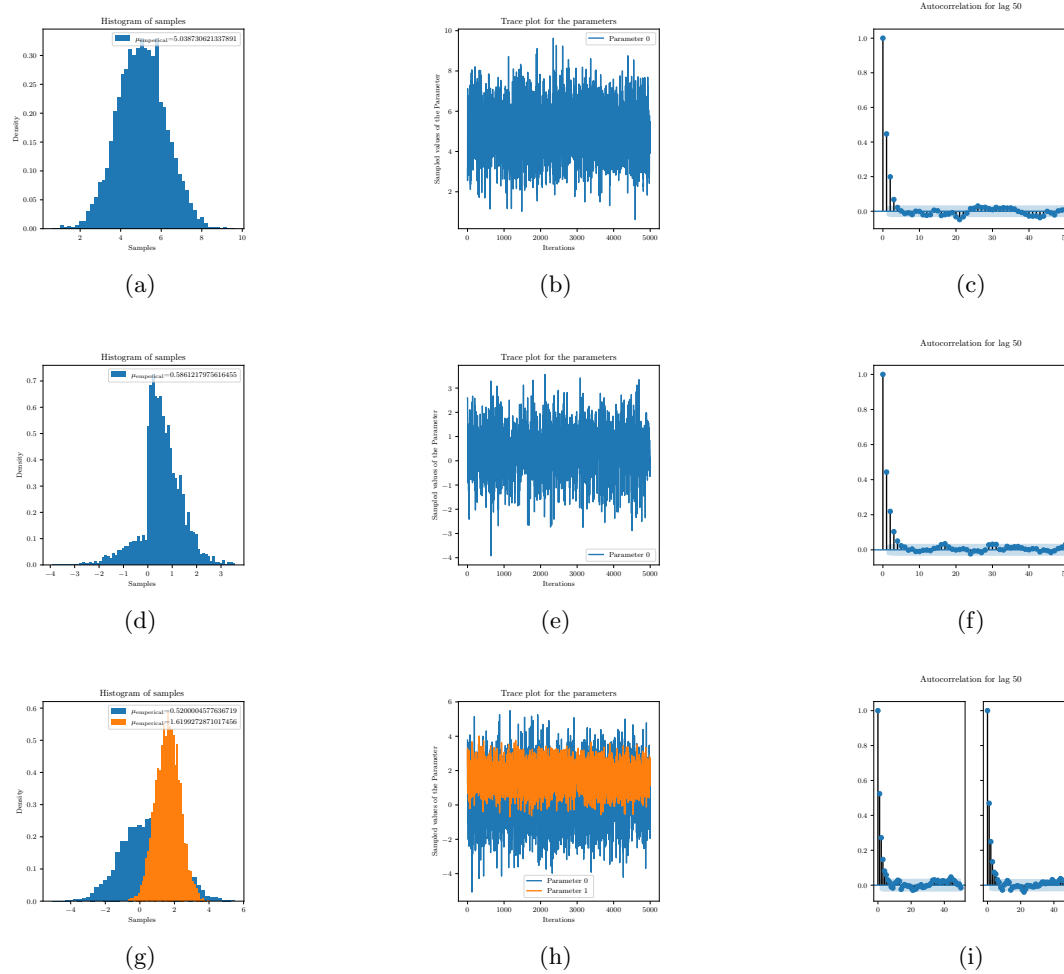


Figure 4: Each row corresponds to the inference output of each model and provides a histogram of the samples, the trace of the samples for each parameter of interest in the model and the autocorrelation between the samples for a lag  $l = 50$ . The top row represents the conjugate Gaussian model, the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

```

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_reduced, samples, mean
    
```

## Acknowledgements

Use unnumbered third level headings for the acknowledgements. All acknowledgements go at the end of the paper.

## 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.
- 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*.
- 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*.