INDIAN INSTITUTE OF TECHNOLOGY, MADRAS

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

UNDERGRADUATE RESEARCH IN COMPUTER SCIENCE - I

PROJECT REPORT

A PROBABILISTIC PROGRAMMING LIBRARY USING EFFECT HANDLERS IN MULTICORE OCAML

Arnhav Datar - cs18b003@smail.iitm.ac.in

under Dr. KC Sivaramakrishnan

May 1, 2021

Contents

1 Acknowledgements

This work has been done primarily because of the guidance of Dr. KC Sivaramakrishnan and Dr. Tom Kelly. I would like to thank them both for their constant support and valuable suggestions. I would also like to thank Dr. Yadu Vasudev for his coordination of the UGRC course.

2 Abstract

In this report, we present EffPPL, a shallowly embedded domain-specific probabilistic programming library in Multicore OCaml made using effect handlers. EffPPL has the capability to perform approximate Bayesian inference for models using continuous random variables. EffPPL uses the Hamiltonian Monte Carlo for performing the inference. We briefly explain the implementation of the same. In this report we also explore potential applications of EffPPL. Furthermore, we compare EffPPL's performance against Stan [1](one of the most highly optimised probabilistic programming languages) for a linear regression model, to show the practicality of EffPPL.

3 Introduction

3.1 Probabilistic Programming

Bayesian modelling lies at the heart of Machine Learning. Bayesian modelling involves assigning the parameters of the model some prior distributions, followed by giving a set of conditions, wherein, we compute the likelihood that the parameters could generate the observed data. This is then used to generate the posterior distribution.

Since exact Bayesian inference is often intractable, we work towards approximate Bayesian inference. Probabilistic programming languages(PPLs) have been used for performing approximate Bayesian inference [2] in complex generative models. PPLs allow us to create these models using simpler models and probability distributions. Probabilistic Programming [3] involves the construction of inference problems and the development of corresponding evaluators, that computationally characterize the denoted conditional distribution.

Probabilistic programming has shown applications [4] in the fields of computer vision, cryptography, quantitative finance, biology and reliability analysis. Until recently, probabilistic programming was limited in scope, and most inference algorithms had to be written manually for each task. Recently with availability of greater compute capability, larger datasets and advances in concurrent and parallel programming, the probabilistic programming paradigm is becoming ubiquitous.

3.2 Effect Handlers

Effect handlers [5] are a mechanism for modular programming with user-defined effects. Similar to defining new exception values, the user can define their own effects. These are handled with an effect handler. But unlike exceptions, effect handlers permit the control to go back to where the effect was performed, but also save the context necessary for going back in a data structure. This ability to suspend and resume computations later allows new programming paradigms, including algorithmic differentiation and probabilistic programming. Multicore OCaml incorporates effect handlers as a way of supporting concurrency primitives. The example below illustrates effect handlers:

```
let comp () =
     print_string "0 ";
     print_string (perform E);
     print_string "3 "

let main () =
     try
        comp ()
     with effect E k ->
         print_string "1 ";
         continue k "2 ";
         print_string "4 "
```

Listing 1: Demonstrating Effect Handlers in Multicore OCaml

Output : 0 1 2 3 4

Control Flow of the above program

1. Function `comp()` is tried
2. Going sequentially it prints 0.
3. Upon seeing the `(perform E)`, suspends the current computation and reifies it as a delimited continuation [6].
4. Going sequentially it prints 1.
5. Here upon seeing the delimited continuation it returns 2 back to `comp()`, wherein it prints 2.
6. Continuing `comp()` it prints 3.
7. Then, it finally continues the remaining part of the effect E. This is similar to returning from a function. It prints 4.

For further examples of effect handlers in Multicore OCaml refer to the `effects-examples` [7] repository on GitHub. Effect handlers have been used for probabilistic programming languages [8, 9]. Effect handlers are a great fit for probabilistic programming since they can be used to implement algorithmic differentiation, and the non-local control flow necessary for expressing probabilistic programming constructs.

4 Related Work

There have been many probabilistic programming efforts both as libraries and as independent languages. The development of these languages has sped up in the past few years, there being a lot of languages/libraries developed in the last 5 years. Some noteworthy mentions are Stan [1], Pyro [9], HackPPL [8], CuPPL [10], OwlPPL [11] and LF-PPL [12].

There have been a number of prior works that implement PPLs in OCaml. IBAL [13] and Hansei [14] are PPLs that have been implemented in OCaml. However, the languages are not universal languages. Universal probabilistic languages can have an unlimited number of random variables. By contrast EffPPL can handle an unlimited number of random variables. Furthermore, they are not implemented using effect handlers as opposed to the EffPPL. They also tend to use simpler inference algorithms as compared to the Hamiltonian Monte Carlo. Also these are programming languages and thus they can only serve the purpose of probabilistic programming. EffPPL being a shallowly embedded [15] DSL in OCaml can use all the other capabilities of OCaml.

Kiselyov et al. [16] present an embedded probabilistic programming library in OCaml. This library is one of the first works of an embedded PPL in OCaml. However, it used simple inference algorithms like the exact inference for discrete distributions and the importance sampling for continuous distributions. EffPPL on the other hand uses the Hamiltonian Monte Carlo. The library also does not use effect handlers as compared to EffPPL.

Recently, Anik Roy developed OwlPPL [11], a universal embedded PPL using the Owl library [17], a general-purpose numerical library for OCaml. However, similar to its predecessors, it does not use effect handlers. It implements a variety of inference algorithms ranging from simple enumeration to MCMC methods like Metropolis-Hastings. It does not implement more modern inference algorithms like the Hamiltonian Monte Carlo(HMC) [18], Stochastic gradient Hamiltonian Monte Carlo(SGHMC) [19] and the No U-Turn Sampler(NUTS) [20]. EffPPL on the other hand implements the HMC.

Effect handlers have been used for developing modern universal PPLs like HackPPL [8] from Facebook and Pyro [9] from Uber. Pyro is built on Poutine which is a library of effect handlers. Edward [21] is a modern probabilistic modelling library that supports concurrency, but it does not use effect handlers and is not universal. Given the successes of these languages, EffPPL is our attempt at taking advantage of effect handlers in Multicore OCaml to implement an universal PPL. Table 1 is a detailed table summarising the probabilistic programming languages referred to in this report.

Table 1: A collection of PPLs

PPL	Host Language	Reference
BUGS	N/A	[22]
IBAL	OCaml	[13]
JAGS	N/A	[23]
Church	LISP	[24]
HANSEI	OCaml	[14]
Infer.NET	F#	[25]
Anglican	Clojure	[26]
Pyro	Python	[9]
OwlPPL	OCaml	[11]
Edward	Python	[21]
Stan	C++,Python,R	[1]
HackPPL	Hack	[8]
CuPPL	CUDA	[10]
LF-PPL	Python	[12]

5 Implementation

5.1 Language Design

For the user to have an intuitive and easily understandable interface to create the models the language tries to mirror OCaml with the only difference being using `let*` as opposed to `let`. The applications described in section 6 show how useful this design style is. A simple example illustrating the same is given below:

```
let sum_of_normals () =
  let* x1 = normal 0. 1. in
  let* x2 = normal 0. 1. in
  x1 +. x2
```

Listing 2: Sum of standard normal variables

The language has been implemented as a shallowly embedded [15] DSL in OCaml. Being a DSL, other constructs of OCaml such as the let, if, for statements along with functions written in OCaml can be used within the library functions.

OCaml post version 4.08 allowed the defining of a custom `let*` binding. So in order to create a shallowly embedded DSL we redefine a few primitives such as the `let*` and arithmetic operators(such as `+.`, `-.`, `*.`). We also add functions(such as `normal`) to simulate primitive distributions and observe random variables.

The newly defined functions/operators invoke effect handlers upon being called. We take advantage of effect handlers ability to simulate user defined effects to overload the meaning of arithmetic operators suitably for the models. This ensures that the domain-experts of probabilistic programming need not be familiar with effect handlers to use the library to its fullest potential.

5.2 The Hamiltonian Monte Carlo

The Hamiltonian Monte Carlo(HMC) has proven a remarkable empirical success in the recent years [18]. Its variant the No-U-Turn Sampler is the industry standard and is used by a lot of recent PPLs [1, 9, 21]. The Hamiltonian Monte Carlo algorithm [20] used in EffPPL is described below:

Algorithm 1: Hamiltonian Monte Carlo

```
Given $\theta^0, L, \epsilon, \mathcal{L}, M$:
for $m = 1$ to $M$ do
    Sample $r^0 \sim \mathcal{N}(0, I)$
    $\theta^m \leftarrow \theta^{m-1}$
    $\theta' \leftarrow \theta^{m-1}$
    $r' \leftarrow r^0$
    for $i = 1$ to $L$ do
        $(\theta', r') \leftarrow Leapfrog(\theta', r', \epsilon)$
    end
    With probability $\min\{1, \frac{\exp(\mathcal{L}(\theta') - 0.5 \cdot r' \cdot r')}{\exp(\mathcal{L}(\theta^{m-1}) - 0.5 \cdot r^0 \cdot r^0)}\}$ set $\theta^m \leftarrow \theta'$ and $r^m \leftarrow -r'$
end
function $Leapfrog(\theta', r', \epsilon)$ :
    $r' \leftarrow r' + (\epsilon/2)\nabla_\theta \mathcal{L}(\theta)$
    $\theta' \leftarrow \theta' + (\epsilon)r'$
    $r' \leftarrow r' + (\epsilon/2)\nabla_\theta \mathcal{L}(\theta')$
return $(\theta', r')$
```

We briefly elaborate the various components of the HMC:

1. **Generating Transitions:**
   Starting from the current value of the parameters θ^0, a transition is made in two steps
   (a) A value for the momentum is drawn from standard normal variables in the step $r^0 \sim \mathcal{N}(0, I)$.
   (b) Now the joint system of (θ, r) is solved using the Hamiltonian equations

$$\frac{d\theta}{dt} = \frac{dH}{dr}$$
$$\frac{dr}{dt} = -\frac{dH}{d\theta}$$

   Betancourt et. al. [18] explains the mathematics behind the equations. For further details kindly refer the same.

2. **Leapfrog Integrator:**
   The last part leaves a two-state differential equation to solve. EffPPL uses the leapfrog integrator, which is a numerical integration algorithm. We use it as a function above which is repeated L times. This is the equivalent of the passing $L\epsilon$ time in a physical system.

3. **Metropolis Accept Step:**
   If the Leapfrog integrator was a perfect integrator this step would have been unnecessary. To account for numerical errors during integration a Metropolis acceptance step is applied at the end of the algorithm.

As can be seen, we need to compute $\nabla_\theta \mathcal{L}$ twice in a leapfrog iteration. We can therefore conclude that we need to calculate gradients at least $2ML$ times. This value may get larger for higher precision tasks. Since it is a heavily repeated complex part of the algorithm, it is necessary to make it efficient. To achieve the same we use effect handlers as described below.

5.3 Algorithmic Differentiation

The Hamiltonian Monte Carlo requires the algorithmic differentiation [27] to compute the updates that are need to be performed. We use the method described in Margossian et. al. [28]. We use the reverse mode differentiation [29] since its more suited when there are more parameters as compared to return values.

The traditional way of implementing reverse-mode AD is to rely on a tape or a computation graph or an explicit data structure. Wang et al. [30] shows how to use delimited continuations to implement reverse mode AD using delimited continuations to inject backward computation as a part of the normal program control flow. While Wang et al use operator overloading from Scala, we achieve the same effect in OCaml by using effect handlers. Given below we have given the cases for addition and multiplication for calculating the gradients.

```
| r ->
  r.d <- 1.0;
  ls := modif_der !ls r.v r.d;
  (r)
| effect (Add(a,b)) k ->
  let t = {v = a.v +. b.v; d = 0.; m=1.} in
  ignore (continue k t);
  a.d <- a.d +. t.d;
  b.d <- b.d +. t.d;
  ls := modif_der !ls a.v a.d;
  ls := modif_der !ls b.v b.d;
  (x)

| effect (Mult(a,b)) k ->
  let t = {v = a.v *. b.v; d = 0.;  m=1.} in
  ignore (continue k t);
  a.d <- a.d +. (b.v *. t.d);
  b.d <- b.d +. (a.v *. t.d);
  ls := modif_der !ls a.v a.d;
  ls := modif_der !ls b.v b.d;
  (x)
```

Listing 3: Algorithmic Differentiation

As we can see, we initially compute the values from the addition and multiplication of `a` and `b` in `t.v`, and then use delimited continuation to send the value for further computation. To illustrate how this works we show an example for $z = 2x + xy$ for $x = 1$ and $y = 2$. By simple calculus we can see that

$$\partial z/\partial x = 2 + y = 4 \tag{1}$$

$$\partial z/\partial y = x \quad = 1 \tag{2}$$

In EffPPL, `z` can be expressed as `Add(Mult(2,x), Mult(x,y))`. The control flow is as follows:

1. Initially `Mult(2,x)` is evaluated, here simply `x.v` is computed as 2 and sent to the continuation. Let this term be called `z1`.
2. Similarly, `Mult(x,y)` is evaluated here too `x.v` is computed as 4 and sent to the continuation. Let this term be called `z2`.
3. Then the addition is computed as 6 and sent to the continuation.
4. Finally we reach `r`, wherein we store the derivative of z with respect to itself as 1 in `ls`.

5. Now we return to `Add($z2, z2$)`. Here since `t.d` has been updated to 1 both the derivatives of `z` with respect to `z1` and `z2` get the value 1.0

6. The program then goes to `Mult(x,y)`. Here `t.d` is 1 and `a.v = 1` while `b.v`=2, this makes `a.d`(x) as 2 and `b.d`(y) as 1. Hence, at this point the derivatives stored in ls are:

   (a) $\partial z/\partial x = 2.0$

   (b) $\partial z/\partial y = 1.0$

   (c) $\partial z/\partial z = 1.0$

7. Finally the program sees `Mult(2,x)`. Again `t.d` is 1 therefore `a.d`(2) gets updated to 1 and `b.d`(x) gets updated to $2 + 2 = 4$. But the `modif_der` recognises 2 to be a constant and does not consider it. So finally we get the derivatives as :

   (a) $\partial z/\partial x = 4.0$

   (b) $\partial z/\partial y = 1.0$

   (c) $\partial z/\partial z = 1.0$

Therefore, we finally get the derivative of z with respect to x as 4, while we get the derivative of z with respect to y as 1 as was computed in equations (1) and (2).

5.4 Testing

5.4.1 Algorithmic Differentiation

To test the algorithmic differentiation we used two methods

1. **Hand calculation of gradients:** this was done for a few models by writing a script and checking if the values actually give the expected gradients

2. **Plots:** Figures 1 and 2 represent the plots for the gradients of sum of normal variables and the product of normal variables. As can be seen the sum of normal variables closely mirrors a normal slope, while the product of normal variables is a normal variable multiplied with the gradient of a normal variable.

We tried these two tests because at times the the gradient for simple models was easily calculable so we checked manually by calculating gradients. But such tests did not guarantee that the sampling was being done randomly neither did they ensure the same for all values. Plots help us visualise the same.

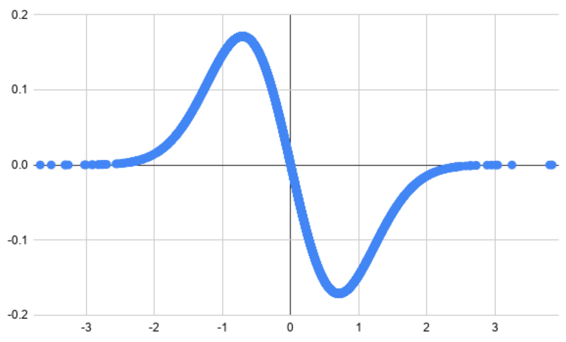


Figure 1: Sum of normals

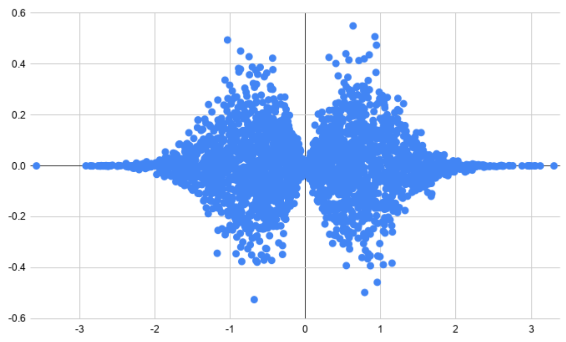


Figure 2: Product of normals

5.4.2 Samplings

To check the final HMC was functioning correctly we sampled from a variety of distributions and checked if their means, medians and standard deviations match with the expected values. This was done for all standard distributions along with some distributions involving combinations of standard distributions. One exception is Cauchy distribution whose mean and variance is undefined. We also plotted the same and checked if the plots match the expected distributions as can be seen in section 6.4.2.

We also perform the Kolmogorov-Smirnov test [31] (K-S test) on these samples, to find low values implying that the sampling is accurate. The Kolmogorov-Smirnov test is a non-parametric goodness-of-fit test, and is used to determine whether an underlying probability distribution differs from a hypothesized distribution. Figure 3 illustrates how the statistic of the K-S test is calculated.

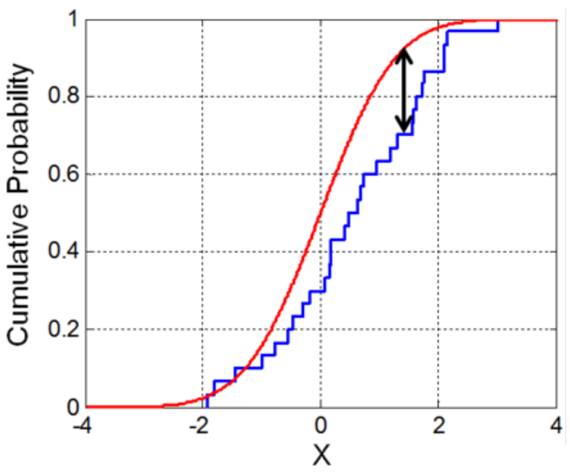


Figure 3: The red line is a model CDF, the blue line is an empirical CDF, and the black arrow is the K–S statistic.

sl As can be seen from the figure above when the K-S statistic is low(close to 0), it implies that the sampling is good representation of the proposed distribution. The detailed table (Table 2) showing the values of all K-S tests along with the number of samples drawn for each can be found below.

Table 2: K-S Test results

Distribution	Samples	K-S test statistic
Sum of normals	10000	0.02704
Chi-squared	2000	0.05836
exponential	10000	0.08071
$Beta(1,1)$	10000	0.00972
$Beta(1,2)$	5000	0.01866
$Beta(2,1)$	5000	0.02334
Continous	5000	0.00645

6 Applications

In this section, we describe a few application that have been implemented using EffPPL. As can be seen with the accompanying code, the models are quite naturally expressed in EffPPL. We have provided a detailed list of functions and primitives used in EffPPL in Table 4 in the Appendix.

6.1 Machine Learning models

Probabilistic programming libraries have shown many applications in the domains of machine learning and deep learning. We present two simple and fundamentals applications here.

6.1.1 Linear Regression

The 1-D linear regression model is the following, with a single predictor and a slope and intercept coefficient, and normally distributed noise. This model can be written using standard regression notation as

$$Y_n \sim \mathcal{N}(\alpha + \beta X_n, \sigma)$$

The code for performing inference here is given below:

```
let lin obs_points ax ay () =
  let* m = normal 2. 3. in
  let* c = normal 0. 10. in
  let* s = exp 5. in

  for i = 0 to (obs_points-1) do
    observe (mk ay.(i) -. m*.mk ax.(i) -. c)
    (logpdf Primitive.(normal 0. (get s)))
  done ;
  m
```

Listing 4: Linear Regression in EffPPL

A walkthrough through the same is as follows:

1. We initially pass `obs_points, ax, ay` as parameters to `lin`.
2. Here we give some prior that we may have about `m,c,s`. As can be seen the priors given by us are not particularly accurate for the expected values of $m = 3, c = 2$ and $s = 1$. If the user is very unsure, he can simply choose the prior as a normal distribution with a large standard deviation.
3. We then use a for loop to iterate over all the observation points. In the statement `observe (mk ay.(i)-. m*.mk ax.(i)-. c)(logpdf Primitive.(normal 0. (get s)))` we try to say that $(y(i) - m * x(i) - c) \sim \mathcal{N}(0, s)$. The first term in the code statement denotes the L.H.S. while the next statement denotes a function that takes in an input float and return the logpdf of a normal distribution.

We use the logpdf, because we need to compute the product of all pdfs in the HMC implementation. Most hardwares won't be able to deal with floating point numbers this small, hence we convert it to log and and add it up in the HMC implementation. To test the correctness we gave it data which was sampled from $\mathcal{N}(3x + 2, 1)$. When we did an HMC sampling we got the mean slope as 3.02 and the constant as 1.84. A similar value was obtained via Stan. More details on the same can be found in section 7.

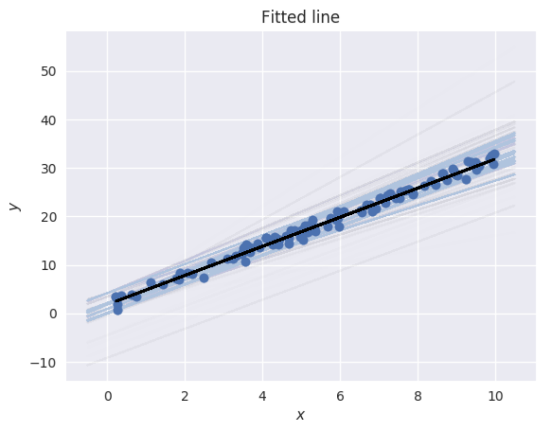


Figure 4: Linear Regression Plot

Figure 4 is a plot showing sampled lines and the mean line. The black line indicates the line with the mean slope and constant. While the other faint blue lines indicate samples that PPL drew.

6.1.2 Binary Classification

The task of classification is one of the most famous ones in machine learning. The fundamental task revolves around classifying the elements of a set into two groups on the basis of previous data. Many algorithms(such as support vector machines, random forests etc.) have been used to tackle the problem. In a PPL setting we propose an elementary algorithm for classification. The idea behind it is that for an ideal linear a correctly classified point should be much more likelier than an incorrectly classified point. This can be seen in the code below as we give a logpdf of +1 when we correctly classify and -1 when we incorrectly classify. The function which does so is:

`(fun x -> if((Float.mul x ay.(i))> 0.)then 1.0 else -1.0)`.

The function basically checks if the predicted and real y have the same sign, if yes it returns 1 else -1 as the logpdf.

```
let class obs_points ax1 ax2 ay () =
  let* m1 = normal 0. 2. in
  let* m2 = normal 0. 2. in
  for i = 0 to (obs_points-1) do
    observe (m1*.mk ax1.(i) +. m2*.mk ax2.(i) +. mk 1.0)
  (fun x -> if( (Float.mul x ay.(i)) > 0.) then 1.0 else -1.0)
  done;
  m1
```

Listing 5: Classification in EffPPL

Given below in figure 5 are some of the classifications we have sampled, with the central black line representing the mean of the parameters.

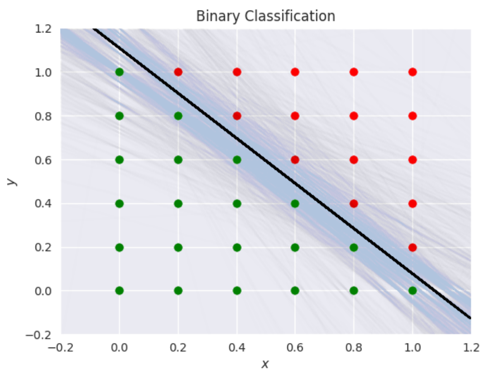


Figure 5: Classification Plot

6.2 Time Series Models

In mathematics, a time series is a series of data points indexed (or listed or graphed) in time order. Time series analysis comprises methods for analyzing time series data in order to extract meaningful parameters of the data. In this section we list one common example of time series analysis

6.2.1 Auto-Regressive Model

The auto-regressive model specifies that the output variable depends linearly on its own previous values and on a stochastic term(usually a normal variable). Here we consider the simplest case(AR(1)) wherein the output is only dependant on its last value. Figure 6 is a plot for an AR(1) model for $\alpha = 5$ and $\beta = 0.4$:

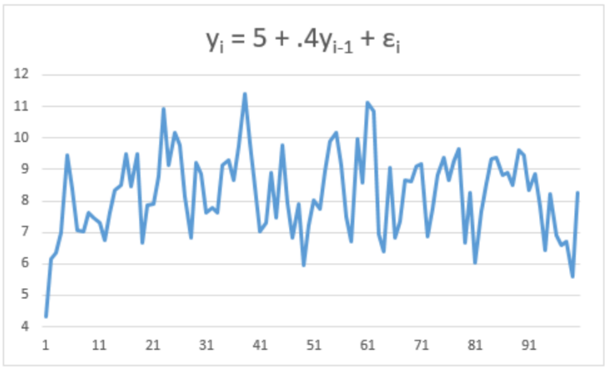


Figure 6: AR(1) model

The auto-regressive model can therefore be defined as follows:

$$Y_n \sim \mathcal{N}(\alpha + \beta Y_{n-1}, \sigma)$$

where $\{Y_i\}$ is a sequence of random variables taken from a time-series. The equivalent EffPPL code to infer the parameters α and β can be found below.

```
let autoreg obs_points ay () =
  let* alp = normal 1. 1. in
  let* bet = normal 1. 1. in
  let* s = exp 5. in

  for i = 0 to (obs_points-2) do
    observe ((mk ay.(i+1)) -. alp -. bet*.(mk ay.(i)))
    (logpdf Primitive.(normal 0. s))
  done
```

Listing 6: Auto-Regressive model in EffPPL

Here we can see that the code is similar to linear regression. But instead of observing a line through y and x, this time we observe it between consecutive values of y in the time series.

To test correctness, it was fed data generated with $\alpha = 0.5$ and $\beta = 1.03$. Figures 7 and 8 show us the plots for Alpha and Beta respectively. EffPPL has been able to predict β with a sufficiently high precision, while α is a bit less accurate for the chosen L and ϵ.

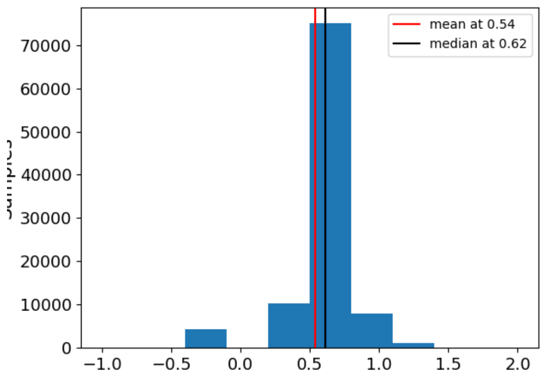


Figure 7: Alpha

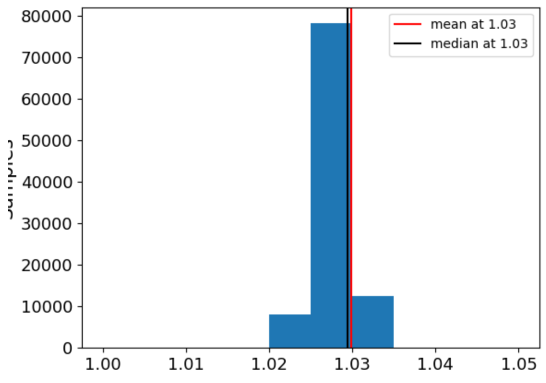


Figure 8: Beta

6.3 Data Analysis

At times we have some model for the data but we don't know parameters associated with the same. EffPPL is able to get these parameters.

6.3.1 Truncated Data

When we are given a data from a distribution and it has been truncated but we do not know precisely where it has been truncated we can use this model to find out the possible values at which the data has been truncated.

```
let truncated_data obs_points ay () =
  let* lb = normal (-2.5) 1. in
  let* ub = normal (2.5) 1. in
  for i = 0 to (obs_points-1) do
    observe (mk ay.(i))
    (fun x ->
      if (x < (get lb) || x > (get ub) ) then
        -1000.0
      else
        (logpdf Primitive.(normal 0. 1.)) x
    )
  done ;
  lb
```

Listing 7: Truncated data parameter retrieval in EffPPL

Here we give priors to `lb` and `ub`. If a value does not lie within the bounds we give a very large negative number as logpdf(equivalent to 0 probability). Otherwise we use the logpdf of the standard normal distribution(from where we know the truncated data comes).

We generated data from the standard normal distribution and bounds lying two standard deviations from the mean. Given below are the plots for the estimates of *lb* and *ub*. As can be seen from figures 9 and 10, they have the highest probability near 2 (or -2) and then it tapers down towards infinity.

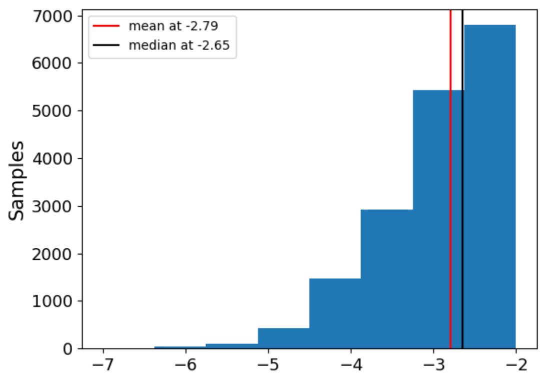


Figure 9: Lower bound Inference

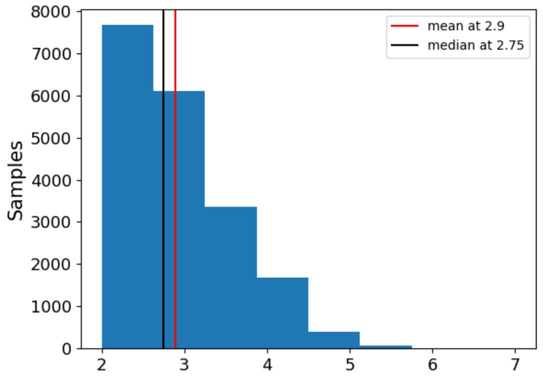


Figure 10: Upper bound inference

6.3.2 Missing Data

Similarly if we do not have the parameters of a particular distribution we can estimate them. Suppose we have a normal distribution whose mean and variance is unknown to us. We can find out these parameters as can be seen below.

```
let missing_data obs_points ay () =
  let* mu = normal ay.(0) 5. in
  let* si = exp 5.0  in

  for i = 0 to (obs_points-1) do
    observe (mk ay.(i))
    (fun x ->
      (logpdf Primitive.(normal (get mu) (get si))) x )
  done ;
  mu
```

Listing 8: Missing parameters

Here we simply observe the data(`ay`) as logpdf of the normal distribution, for parameters `mu` and `si`. When given a few points from the standard normal distribution($mu = 0$ and $si = 1$) it was correctly able to predict the `mu` and `si`. Given below are histograms for the same. As can be seen, `mu` has correctly centered around 0 and `si` has correctly centered around 1.

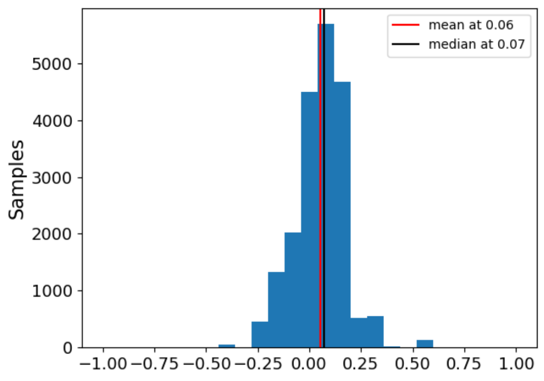


Figure 11: Mean

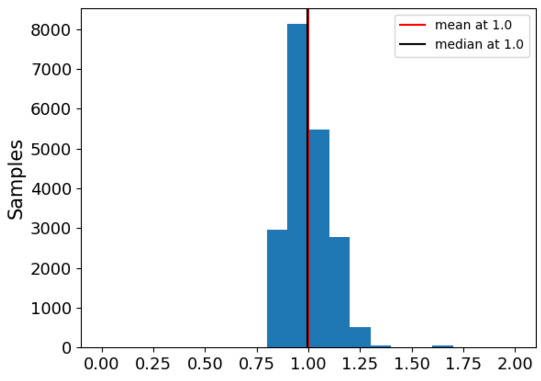


Figure 12: Std. Dev.

The knowledge of these missing parameters can then be used for a various applications such as sampling more data from the same distribution for analysis or for testing against unknown data.

6.4 Sampling

6.4.1 Sum of distributions

Figure 13 shows an example of sampling from the sum of two normal random variables $x1 \sim \mathcal{N}(1, 3)$ and $x2 \sim \mathcal{N}(1, 4)$. Given below is the code for the same.

```
let f1 () =
  let* x1 = normal 1. 3. in
  let* x2 = normal 1. 4. in
  x1 +. x2
```

Listing 9: Sum of normals sampling

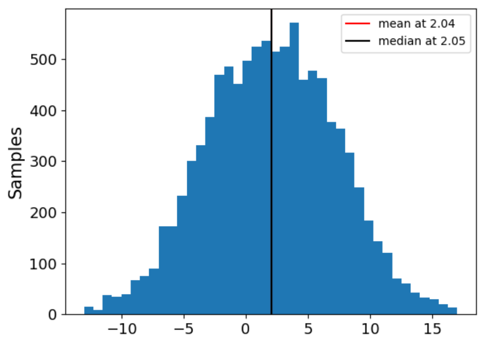


Figure 13: Sum of Normals

The output samples have a mean close to 2.0 and a standard deviation close to 5.0, given below is the plot for the same. This is because if $X \sim \mathcal{N}(\mu_1, \sigma_1)$ and $Y \sim \mathcal{N}(\mu_2, \sigma_2)$, then

$$X + Y = Z \sim \mathcal{N}(\mu_1 + \mu_2, \sqrt{\sigma_1^2 + \sigma_2^2})$$

6.4.2 Variety of Distributions

The EffPPL handles everything via effects, to keep things simple we decided to restrict ourselves to the following distributions

1. Normal Distribution
2. Gamma Distribution
3. Beta Distribution
4. Cauchy Distribution

However, quite a few distributions such as the Exponential distribution, the Chi-squared distribution and the continuous distribution can be obtained from these primitive distributions.

We show some samplings from these distributions below:

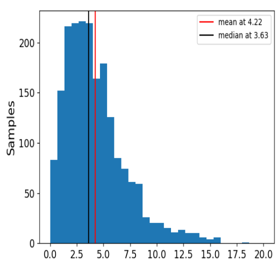


Figure 14: $\chi^2(4)$

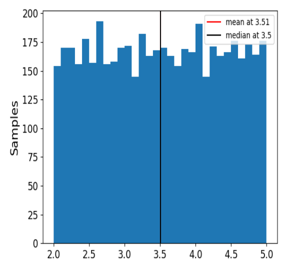


Figure 15: $Cont(2, 5)$

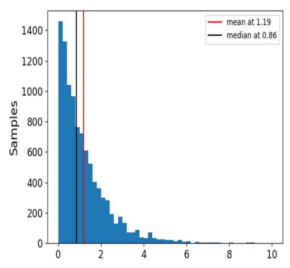


Figure 16: $Exp(1)$

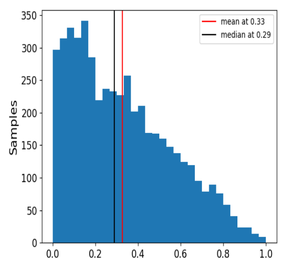


Figure 17: $\beta(1, 2)$

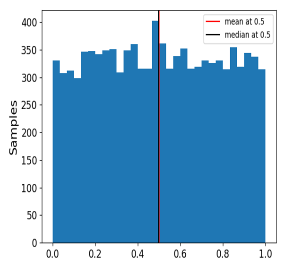


Figure 18: $\beta(1, 1)$

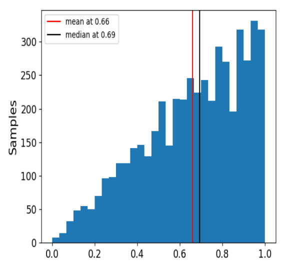


Figure 19: $\beta(2, 1)$

6.5 Discrete Models

While the HMC requires a continuous distribution, as it needs to calculate the gradient of the Probability density function (PDF), we can simulate discreteness by using conditional statements as can be seen in the example below.

```
let f1 () =
  let∗ p = uniform 0. 1. in

  let∗ flip1 = uniform 0. 1. in
  let∗ flip2 = uniform 0. 1. in
  let∗ flip3 = uniform 0. 1. in

  let flip1' = if((get x) > (get flip1)) then 1.0 else 0.0 in
  let flip2' = if((get x) > (get flip2)) then 1.0 else 0.0 in
  let flip3' = if((get x) > (get flip3)) then 1.0 else 0.0 in

  let∗ total = (mk flip1') +. (mk flip2') +. (mk flip3')  in
  observe total (fun x-> if x=2.0 then 0.0 else -1000.0);

  total
```

Listing 10: Coin Flips

The above model is used to simulate the following, if we flip 3 coins and observe 2 heads, and we think the prior of the coin weight is uniformly distributed, then what is the posterior? As can be seen, the above example simulates a Bernoulli random variable(`flip`' variables) with probability p, where p has a prior from the uniform continuous $[0, 1]$ distribution. The total is then observed with a function that returns a large negative number to denote its impossible if the total is not 2. For the number of heads equal to 2 and 3 the histogram plots are shown below in figures 20 and 21.

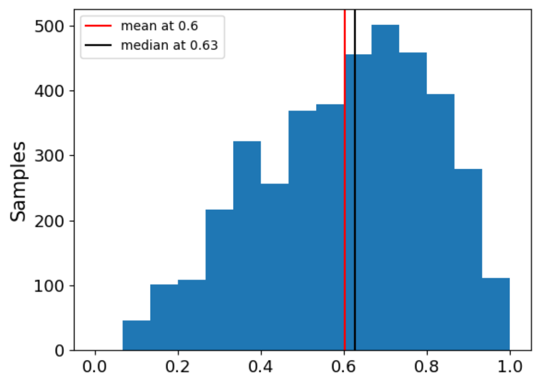


Figure 20: 2 heads of 3 flips

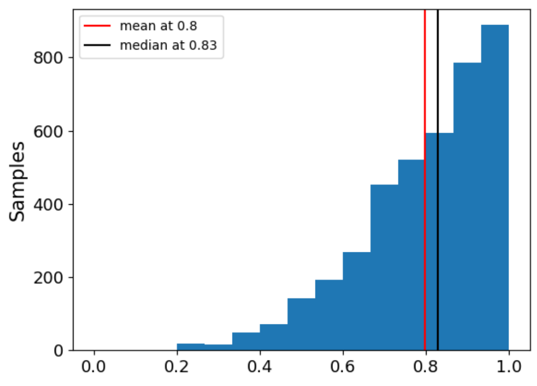


Figure 21: 3 heads of 3 flips

7 Evaluation

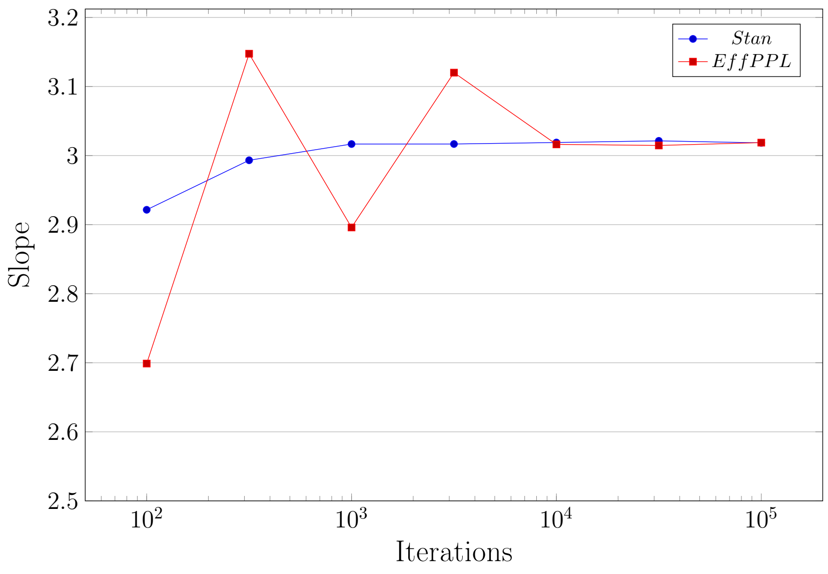
We compare our EffPPL with Stan in this section. Stan has been written in C++ and has witnessed a lot of years of development under many developers. It is considered an industrial strength PPL. Furthermore, it uses the No-U-Turn Sampler, which is widely considered to perform better than the HMC. We are pleasantly surprised that EffPPL compares well with respect to Stan for this example.

7.1 Linear Regression

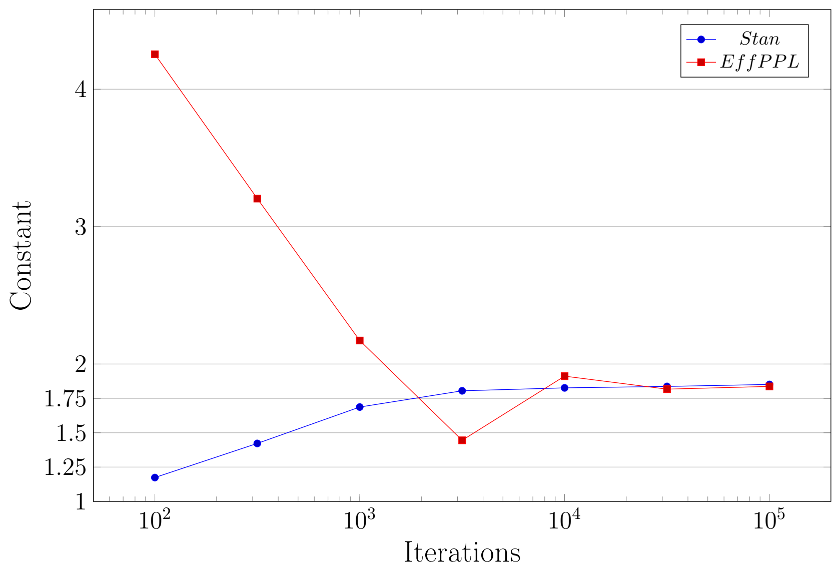
As a sample example we analyse the performance of two linear regression codes written in Stan and EffPPL against each other inferring the same data. We use the linear regression example mentioned in section 6. We give both ppls the same data, and see how well they estimate the parameters and in how much time. For Stan, while performing the inference we gave the number of chains as 4 and number of warmup iterations as 7. EffPPL doesn't use chains or warmup iterations as of now so we thought of keeping the value a bit smaller. For larger values of chains/warmup, we would have higher time taken along with better convergence to the mean.

7.1.1 Parameter Estimation

Given below is a plot between the mean slope versus the number of samples drawn for both EffPPL(red) and Stan(blue). We see that while there is a significant difference in the mean slopes in the initial iterations(100 and 316) between Stan and EffPPL, they converge to a value near 3 in the later iterations.



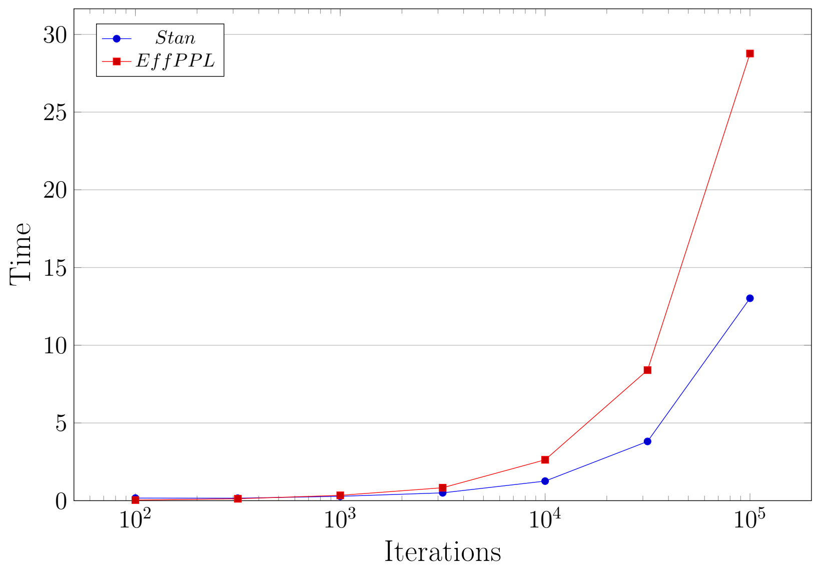
Similarly given below is the similar comparative plot between the mean constants obtained by the two libraries. As can be see they both converge to a value close to 1.8 although Stan converges a bit faster.



The above graphs show that while Stan is a bit faster at converging. For large enough iterations, we can choose either of the libraries, as both converge to the same values.

7.1.2 Time Taken

Below we also compare the times taken by both with respect to iterations. As can be seen while in the earlier iterations EffPPL takes a lead, Stan is roughly 1.5-2 times faster for large number of iterations in the range of 10^4-10^5.



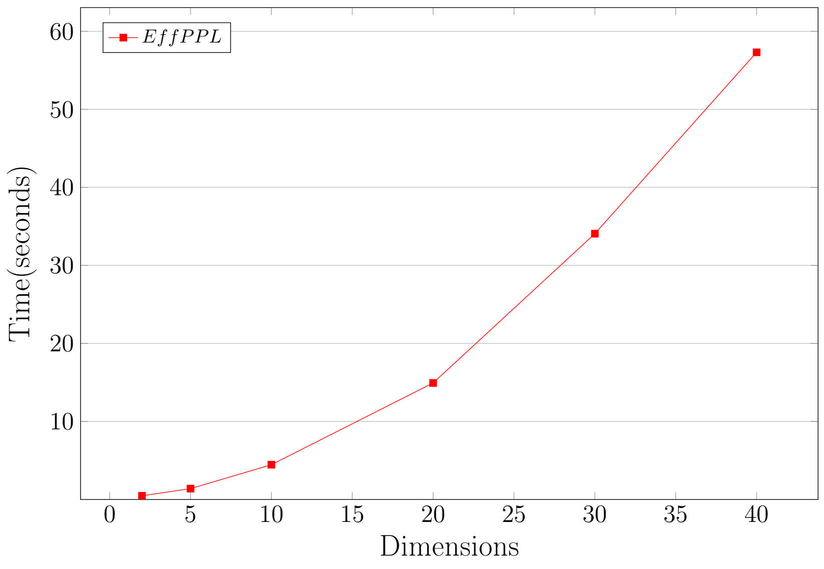
While plot seem to be rising exponentially, they are linearly rising. This is because the x-axis is plotted on a logarithmic scale and hence linear plots look exponential. This can be seen from the table 3 denoting the times taken by both libraries below:

epochs	PyStan (in seconds)	Effppl(in seconds)
100	0.165	0.039
316	0.160	0.123
1000	0.280	0.343
3162	0.503	0.832
10000	1.259	2.629
31623	3.810	8.400
100000	**13.023**	**28.769**

Table 3: Times taken

7.1.3 Dimension

We also compare how the linear regression in EffPPL performs for larger dimensions. Since we have used a simple list as a data structure we expect it to be $O(p^2)$, where p are the number of parameters. In an optimal setting it will be $O(p)$.



As can be seen the plot has a greater than linear component associated with it. However, even for a 40-dimensional linear regression it was able to go through a 1000 iterations with 100 points within a minute.

8 Further Work

While the inference algorithm is performing well, there are some places where the library can be improved. These are:

1. Using a No-U-Turn Sampler instead of the Hamiltonian Monte Carlo. This would save users from choosing appropriate step sizes and iterations.
2. Adding an inference algorithm that can also deal with discrete models. Currently we have to use conditionals on the continuous distributions to get discrete models.

3. Using alternative data-structures to improve the efficiency of the PPL for higher dimensions

4. Adding a Multicore effect scheduler to make the library parallelised.

9 Appendix

Given below is a table of keywords that are used in the EffPPL library

Table 4: EffPPL keywords and primitives

Keyword	Function type	Use
`let*`	`t -> (t -> t ) -> t`	A primitive for declaring variables
`normal`	`float -> float -> t`	Given mean and std. dev. samples numbers from the normal/gaussian distribution
`cauchy`	`float -> float -> t`	Given loc and scale samples numbers from the cauchy distribution
`beta`	`float -> float -> t`	Given a and b samples numbers from the beta distribution
`gamma`	`float -> float -> t`	Given shape and scale samples numbers from the gamma distribution
`exp`	`float -> t`	Given rate samples numbers from the exponential distribution
`chi2`	`float -> t`	Given degrees of freedom samples numbers from the exponential distribution
`uniform`	`float -> float -> t`	Given start and end samples numbers from the uniform distribution
`hmc`	`( unit -> t) -> int -> float -> int -> float list list`	Given function, leapfrog iterations, step size and number of epochs returns a list of samples having length number of epochs
`observe`	`t -> (float -> float) -> unit`	Given a variable to observe and function that returns the logpdf it takes this observation into account while doing inference

References

[1] Bob Carpenter, Andrew Gelman, Matthew Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan : A Probabilistic Programming Language. *Journal of Statistical Software*, 76, 01 2017.

[2] Atilim Güneş Baydin, Lei Shao, Wahid Bhimji, Lukas Heinrich, Lawrence Meadows, Jialin Liu, Andreas Munk, Saeid Naderiparizi, Bradley Gram-Hansen, Gilles Louppe, Mingfei Ma, Xiaohui Zhao, Philip Torr, Victor Lee, Kyle Cranmer, Prabhat, and Frank Wood. Etalumis: Bringing probabilistic programming to scientific simulators at scale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[3] Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. An Introduction to Probabilistic Programming, 2018.

[4] Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. Probabilistic Programming. In *Future of Software Engineering Proceedings*, FOSE 2014, page 167–181, New York, NY, USA, 2014. Association for Computing Machinery.

[5] Matija Pretnar. An Introduction to Algebraic Effects and Handlers. Invited tutorial paper. *Electronic Notes in Theoretical Computer Science*, 319:19–35, 12 2015.

[6] Marek Materzok and Dariusz Biernacki. Subtyping Delimited Continuations. *SIGPLAN Not.*, 46(9):81–93, September 2011.

[7] Multicore OCaml. Effects Examples .

[8] Jessica Ai, Nimar S. Arora, Ning Dong, Beliz Gokkaya, Thomas Jiang, Anitha Kubendran, Arun Kumar, Michael Tingley, and Narjes Torabi. HackPPL: A Universal Probabilistic Programming Language. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2019, page 20–28, New York, NY, USA, 2019. Association for Computing Machinery.

[9] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. Pyro: Deep Universal Probabilistic Programming, 2018.

[10] Alexander Collins and Vinod Grover. Probabilistic Programming with CuPPL, 2020.

[11] Anik Roy. A probabilistic programming language in OCaml. 2020.

[12] Yuan Zhou, Bradley J. Gram-Hansen, Tobias Kohn, Tom Rainforth, Hongseok Yang, and Frank Wood. LF-PPL: A Low-Level First Order Probabilistic Programming Language for Non-Differentiable Models, 2019.

[13] Avi Pfeffer. The Design and Implementation of IBAL: A General-Purpose Probabilistic Language. *Applied Sciences*, 01 2000.

[14] Oleg Kiselyov and Chung-Chieh Shan. Embedded Probabilistic Programming. In *Proceedings of the IFIP TC 2 Working Conference on Domain-Specific Languages*, DSL '09, page 360–384, Berlin, Heidelberg, 2009. Springer-Verlag.

[15] Jeremy Gibbons and Nicolas Wu. Folding Domain-Specific Languages: Deep and Shallow Embeddings (Functional Pearl). *SIGPLAN Not.*, 49(9):339–347, August 2014.

[16] Oleg Kiselyov and Chung-chieh Shan. "Embedded Probabilistic Programming". In Walid Mohamed Taha, editor, *Domain-Specific Languages*, pages 360–384, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[17] Liang Wang. Owl: A General-Purpose Numerical Library in OCaml, 2018.

[18] Michael Betancourt. A Conceptual Introduction to Hamiltonian Monte Carlo, 2018.

[19] Tianqi Chen, Emily B. Fox, and Carlos Guestrin. Stochastic Gradient Hamiltonian Monte Carlo. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, page II–1683–II–1691. JMLR.org, 2014.

[20] Matthew D. Hoffman and Andrew Gelman. The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo, 2011.

[21] Dustin Tran, Matthew D. Hoffman, Rif A. Saurous, Eugene Brevdo, Kevin Murphy, and David M. Blei. Deep probabilistic programming. In *International Conference on Learning Representations*, 2017.

[22] Wally R Gilks, Andrew Thomas, and David J Spiegelhalter. A language and program for complex Bayesian modelling. *Journal of the Royal Statistical Society: Series D (The Statistician)*, 43(1):169–177, 1994.

[23] Martyn Plummer. JAGS: A Program for Analysis of Bayesian Graphical Models using Gibbs Sampling. *3rd International Workshop on Distributed Statistical Computing (DSC 2003); Vienna, Austria*, 124, 04 2003.

[24] Noah Goodman, Vikash Mansinghka, Daniel M Roy, Keith Bonawitz, and Joshua B Tenenbaum. Church: a language for generative models. *arXiv preprint arXiv:1206.3255*, 2012.

[25] Shen SJ Wang and Matt P Wand. Using infer. net for statistical analyses. *The American Statistician*, 65(2):115–126, 2011.

[26] David Tolpin, Jan-Willem van de Meent, Hongseok Yang, and Frank Wood. Design and implementation of probabilistic programming language anglican. In *Proceedings of the 28th Symposium on the Implementation and Application of Functional programming Languages*, pages 1–12, 2016.

[27] Atılım Günes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: A survey. *J. Mach. Learn. Res.*, 18(1):5595–5637, January 2017.

[28] Charles C. Margossian. A review of automatic differentiation and its efficient implementation. *WIREs Data Mining and Knowledge Discovery*, 9(4), Mar 2019.

[29] Barak A. Pearlmutter and Jeffrey Mark Siskind. Reverse-mode ad in a functional framework: Lambda the ultimate backpropagator. *ACM Trans. Program. Lang. Syst.*, 30(2), March 2008.

[30] Fei Wang, Daniel Zheng, James Decker, Xilun Wu, Grégory M. Essertel, and Tiark Rompf. Demystifying Differentiable Programming: Shift/Reset the Penultimate Backpropagator, 2019.

[31] *"Kolmogorov–Smirnov Test"*, pages 283–287. Springer New York, New York, NY, 2008.