Probabilistic Programming in Julia

New Inference Algorithms

Kai Xu

Department of Engineering

University of Cambridge

This dissertation is submitted for the degree of

Master of Philosophy

Homerton College August 2016

Declaration

I Kai Xu of Homerton College, being a candidate for the M.Phil in Machine Learning, Speech and Language Technology, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Total word count: 13,666

Signed: Kai Xu

Date: 12/08/2016

Acknowledgements

I would like to acknowledge the valuable advice and thoughtful help from my supervisor Prof. Zoubin Ghahramani and his PhD student Hong Ge.

Abstract

In this thesis we look at the design and development of a Probabilistic Programming Language (PPL) in Julia named Turing and the challenges of implementing the Hamiltonian Monte Carlo (HMC) sampler inside the Turing framework.

This dissertation starts with a review of three important fields behind the project, which are Bayesian inference, general inference algorithms and probabilistic programming. This review provides theoretical foundations of the design of a universal PPL. Then some existing PPLs are reviewed, especially Stan and the up-to-date version of Turing. It is shown that, compared with Stan, Turing is more expressive and flexible in general.

After that, the design and implementation of the HMC sampler is given. This part starts with the design of the compiler, in which a metaprogramming technique called macro is used to support three probabilistic syntax in Turing. Then the implementation of the standard HMC algorithm is discussed. In the Turing framework, two core ingredients of HMC, the target density function and the corresponding gradient function, are not straightforward to evaluate, which are the two main challenges of this project. The first issue is solved by building connections between the probabilistic program and the corresponding energy function required by HMC using Bayes' rule, and the second one is accomplished by the use of Automatic Differentiation (AD) through probabilistic programs.

Evaluations on the implemented HMC sampler have shown that the implementation is correct and the speed of the sampler is acceptable. Further improvements in terms of the performance of the HMC sampler and the functionality of Turing are proposed in the end.

The main novel contributions of this thesis are a workable HMC sampler in Turing with acceptable performance as well as an updated version of the compiler which supports this HMC sampler and the further development of a Gibbs sampler combining HMC and PG.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With machine learning becoming increasingly ubiquitous and important, probabilistic modelling also becomes more popular. Probabilistic programming is a state-of-the-art research area, of which the mission is to produce a more flexible approach to probabilistic modelling in sense of defining and learning models, and Julia is a new programming language especially developed for efficient scientific computing. Combing both of them, the Cambridge University Engineering Department (CUED) established a probabilistic programming project named Turing, a probabilistic programming language (PPL) based on Julia. This M.Phi project is a part of this ongoing PPL project.

In this project, the objective is to implement a new inference algorithm, Hamiltonian Monte Carlo (HMC), in Turing. It involves developing a new compiler which supports the new sampler using metaprogramming and implementing the standard HMC algorithm in Turing, in the latter of which gradient information used by HMC is computed by Automatic Differentiation (AD) techniques. The final result of the project is a new compiler which is compatible with all existing samplers and a workable HMC sampler with acceptable performance in Turing.

The remaining of this dissertation starts with necessary background knowledge of this project in terms of three major topics, Bayesian inference, general inference algorithms and probabilistic programming, along with some motivations behind them in Chapter 2. Then in Chapter 3, related work including some other existing PPLs (especially Stan) as well as the existing infrastructure of Turing are introduced. In Chapter 4, the design and implementation of the HMC sampler is discussed in detail. After that, some experimental results of the HMC sampler in terms of validation of correctness and evaluations of performance and robustness are given in Chapter 5. Finally in Chapter 6, the dissertation finishes with a summary of this project with current limitations of Turing discussed, and proposes some future works to overcome these limitations.

Chapter 2

Background

This chapter gives a brief review of three core components involved in this project, which are Bayesian inference framework, general inference algorithms and probabilistic programming.

2.1 Bayesian Inference

Bayesian inference uses Bayes' rule to learn model parameters from some given data. In particular, for a given model m with some unknown parameters θ, Bayesian inference estimates the distribution of θ from observed data D using Bayes' rule (Equation 2.1)

$$\begin{aligned} p(\theta|D, m) &= \frac{p(D|\theta, m)p(\theta|m)}{p(D|m)} \\ &= \frac{p(D|\theta, m)p(\theta|m)}{\int p(D|\theta, m)p(\theta|m)d\theta}, \end{aligned} \tag{2.1}$$

where $p(D|\theta, m)$ is called the likelihood of parameters θ in model m, $p(\theta|m)$ is named the prior probability of θ in model m and $p(\theta|D, m)$ is the posterior probability of θ given data D in model m. In brief, $p(D|\theta, m)$ is just called the likelihood, $p(\theta|m)$ the prior and $p(\theta|D,m)$ the posterior.

Bayes' rule cooperates the prior knowledge $p(\theta|m)$ with the likelihood $p(D|\theta,m)$ obtained from data and updates the posterior knowledge about the parameters $p(\theta|D,m)$. This obtained posterior is actually an updated prior and can be used as the prior knowledge for further data.

With model parameters learnt, models can be used to test new data D_{new} by calculating

$$P(D_{\text{new}}|D,m) = \int P(D_{\text{new}}|\theta,D,m)P(\theta|D,m)d\theta. \tag{2.2}$$

A large prediction $P(D_{\text{new}}|D,m)$ indicates the model has a good prediction performance on the data or new data fits the model well, and verse vice. This could be used to either 1) test whether a model is satisfying by i) splitting a dataset into training set and testing set, ii) training the model on the training set and iii) testing the model on the testing set, or 2) test whether some incoming data is from the same data set or not.

In addition, simpler models are usually preferred in the sense of that they are more probable, which is called the Bayesian Ockham's razor. Simpler models can be chosen by comparing models using $p(D|m)$ in Equation 2.1 as a metric. The term $p(D|m)$ here is called the marginal likelihood or model evidence [1, 2].

A simpler model usually centres its model evidence within a small range of data while a more complex model is more expressive so it spreads its marginal probability more thinly over the data space [1]. Figure 2.1 gives an illustrations of how the model evidence distributes differently for simple and complex model. Here the x-axis represents the space of data, the range C is the corresponding range of data D in the space, model m_1 is the simpler model and model m_2 is the more complex one. It can be seen that the marginal probability of m_1 are more centred while that of m_2 spreads wider. Although model m_2 can explain more data range than m_1, for given data D, m_1 is more probable than m_2 because the corresponding integral of model evidence over C is larger. This shows a situation where a more expressive model can be less probable [1].

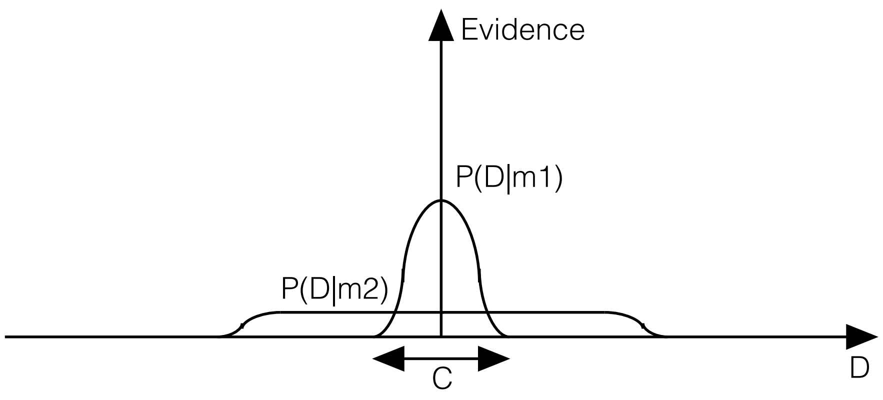


Figure 2.1: Illustration of Bayesian Ockham's Razor.

Thanks to that Bayesian inference makes it possible to learn model parameters from data by incorporating prior knowledge with likelihood and sampling model parameters from the corresponding posterior distribution, it owns three advantages over other inference methods. The first one is that Bayesian inference works well when the distribution has multiple modes, where optimisation methods that maximise the posterior distribution usually perform unsatisfactorily. Secondly, the use of prior knowledge can help prevent models from overfitting into the data as well as help utilise some useful real world knowledge in learning. For instance, it is shown that the weights of some existing object recognition neural nets can be used as priors to train a new recogniser on new dataset [3]. Thirdly, Bayesian inference gives probabilistic interpretation of parameters and predictions. With uncertainty in parameters known, the stability of the model can be told; with uncertainty in predictions known, it is possible to train a model only on data points which show high certainty.

2.2 General Inference Algorithms

Inference means estimating model parameters from data. There are generally two categories of inference methods. The first category contains exact methods including complete enumeration and exact marginalisation, and the second one consists of approximate methods including deterministic approximations and Monte Carlo methods.

Methods in the first category usually fail when distributions are intractable, i.e. it is impossible to conduct these methods mathematically. However, Monte Carlo methods can be used to do inference on any distribution, thus they are also called general inference algorithms. As probabilistic programming requires automatic inference of **any** probabilistic models defined by the user, Monte Carlo methods are necessary in this project.

In the Turing project, there are mainly three inference algorithms involved[1]. They are Sequential Monte Carlo (SMC), Particle Gibbs (PG) and Hamiltonian Monte Carlo (HMC), the third of which is the target algorithm to be implemented in this M.Phi project. This section describes the theory behind these three inference algorithms.

2.2.1 Sequential Monte Carlo and Particle Gibbs

Sequential Monte Carlo, which is also known as particle filtering (PF) or particle smoothing in the Hidden Markov Model (HMM) framework, is a set of simulation-based inference algorithms allowing us to approximate any distributions sequentially. The aim of SMC is to estimate unknown quantities from some observations, where the unobserved signal is modelled by a Markov process and the observation is assumed to be conditionally independent given the unobserved signal [4].

Specifically, the unknown quantity in the Markov process $\{X_k\}_{k\geq 1}$ is charac-

[1]The fact is true up to the date when this dissertation is written and there may be more samplers supported in Turing in the future.

terised by the initial density

$$X_1 \sim \mu(\cdot) \tag{2.3}$$

with its transition density being

$$X_k|(X_{k-1} = x_{k-1}) \sim f(\cdot|x_{k-1}). \tag{2.4}$$

By denoting $x_{i:j} = (x_i, x_{i+1}, \ldots, x_j)$ for $i \leq j$, we have the probability of the unknown sequence $x_{1:n}$

$$\begin{aligned} p(x_{1:n}) &= p(x_1) \prod_{k=2}^{n} p(x_k|x_{1:k-1}) \\ &= \mu(x_1) \prod_{k=2}^{n} f(x_k|x_{k-1}). \end{aligned} \tag{2.5}$$

Also, the conditional independence of the observation $\{Y_k\}_{k\geq 1}$ given the unobserved signal is characterised by

$$Y_k|(X_k = x_k) \sim g(\cdot|x_k). \tag{2.6}$$

Therefore, the probability of observations given hidden states is

$$p(y_{1:n}|x_{1:n}) = \prod_{k=1}^{n} g(y_k|x_k). \tag{2.7}$$

Monte Carlo Sampling Assume we are interested in estimating the probability density

$$p(x_{1:n}|y_{1:n}) = \frac{p(x_{1:n}, y_{1:n})}{p(y_{1:n})} \propto p(x_{1:n}, y_{1:n}), \tag{2.8}$$

where n is fixed.

A Monte Carlo way to approximate this density is to sample a large number of i.i.d random variable $X_{1:n}^{(i)} \sim p(x_{1:n}|y_{1:n})$ and then this density can be approximated by

$$\hat{p}(x_{1:n}|y_{1:n}) = \frac{1}{N}\sum_{i=1}^{N} \delta_{X_{1:n}^{(i)}}(x_{1:n}), \tag{2.9}$$

where $\delta_{X_{1:n}^{(i)}}(x_{1:n})$ is the delta-Dirac mass which holds

$$\int_A \delta_{a_{1:n}}(x_{1:n})dx_{1:n} = \begin{cases} 1 & \text{if } a_{1:n} \in A \subset E^n \\ 0 & \text{otherwise} \end{cases}. \tag{2.10}$$

The Monte Carlo method works well when it is possible to sample from the probability density $p(x_{1:n}|y_{1:n})$ and n is not large.

However, there are two problems of the MC method in practise.

1. For most of the problems of interest, we cannot directly sample from $p(x_{1:n}|y_{1:n})$.
2. As n increase, the sampling process of this method becomes inefficiency.

These two problems can be tackled by using a Monte Carlo sampling method called Sequential Importance Sampling (SIS), which is a sequential version of Importance Sampling (IS) [5].

Sequential Monte Carlo

A SMC sampler is essentially a sequential IS sampler with resampling process [6].

Importance Sampling Assuming that $p(x_{1:n}|y_{1:n})$ is difficult to be sampled directly, IS draws samples $X_{1:n}^{(i)}$ from a so-called importance distribution

$q(x_{1:n}|y_{1:n})$. This importance distribution is chosen to be easy to be sampled from, and is usually the prior distribution $p(x_{1:n})$. After samples are drawn from the importance distribution, each of them is then weighted by

$$w_n^{(i)} = \frac{p(x_{1:n}, y_{1:n})}{q(x_{1:n}|y_{1:n})} \propto \frac{p(x_{1:n}|y_{1:n})}{q(x_{1:n}|y_{1:n})}, \tag{2.11}$$

which is called the unnormalised importance weight.

Then, it is possible to use these weighted samples to approximate the expectation of a function $\varphi(\cdot)$ under the target distribution by

$$E_{p(x_{1:n}|y_{1:n})}(\varphi) = \frac{\sum_{i=1}^{N} w_n^{(i)} \varphi(X_{1:n}^{(i)})}{\sum_{i=1}^{N} w_n^{(i)}} \tag{2.12}$$

For example, the mean of the target distribution can be computed by setting $\varphi(x) = x$ and applying Equation 2.12, which gives

$$E(X_{1:n}^{(i)}) = \frac{\sum_{i=1}^{N} w_n^{(i)} X_{1:n}^{(i)}}{\sum_{i=1}^{N} w_n^{(i)}}. \tag{2.13}$$

Sequential Importance Sampling Importance Sampling solves the problem of the target distribution being unable to be sampled directly, to further improve the sampling efficiency, IS is adopted to be its sequential version, which is described as follow.

Knowing that

$$\begin{aligned} p(x_{1:n}) &= p(x_{1:n-1}) \times f(x_n|x_{n-1}) \\ &= \mu(x_1) \prod_{k=2}^{n} f(x_k|x_{k-1}) \end{aligned}, \tag{2.14}$$

we can obtain $X_{1:n}^{(i)}$ and $w_{1:n}^{(i)}$ when we already have $X_{1:n-1}^{(i)}$ at time $n-1$

by

1. Sample $X_n^{(i)}$ from $f(\cdot|X_{n-1}^{(i)})$ (or $\mu(\cdot)$ if $n = 1$);
2. Set $w_{1:n}^{(i)} = (w_{1:n-1}^{(i)}, w_{n-1}^{(i)} \times g(y_n|X_n^{(i)}))$;
3. Set $X_{1:n}^{(i)} = (X_{1:n-1}^{(i)}, X_n^{(i)})$.

This sequential way owns the property that, no matter how large n is, there is always only one component X_n to be sampled at one time. In addition, this algorithm can be easily parallelised and distributed to multiple computers or processors, which means that the sampling speed can be further accelerated.

Resampling Another problem faced by SMC is that as n increases, all the mass of the target distribution will be concentrated on a few number of samples, i.e. some of the particles have weights with large values while others have weights close to 0.

The solution to this problem is to discard particles with low weights and multiply particles with high weights.

Knowing that at time n, IS provides the approximation of $p(x_{1:n}|y_{1:n})$

$$\hat{p}(x_{1:n}|y_{1:n}) = \sum_{i=1}^{N} W_n^{(i)} \delta_{X_{1:n}^{(i)}}(x_{1:n}), \tag{2.15}$$

where $W_n^{(i)} = \frac{w_n^{(i)}}{\sum_{i=1}^N w_n^{(i)}}$ and is named the normalised weight.

When the distribution has its mass centred in a few number of particles, it is resampled by sampling N times $X_{1:n}^{(i)} \sim \hat{p}(x_{1:n}|y_{1:n})$ to build the new approximation

$$\tilde{p}(x_{1:n}|y_{1:n}) = \frac{1}{N} \sum_{i=1}^{N} \delta_{X_{1:n}^{(i)}}(x_{1:n}) \tag{2.16}$$

The criteria to perform resampling step can be set as the Effective Sample

Size (ESS) of the samples being lower than a threshold, which is the way Turing conducts. Also, these new resampled samples are proved to be approximately distributed to $p(x_{1:n}|y_{1:n})$ but statistically dependent [5].

Particle Gibbs

The PG method is a SMC based method that runs multiple passes of the SMC algorithm, where each pass is conditional on the trajectory sampled at the last run of the SMC sampler [7]. The conditioned on trajectory here is known as a reference trajectory.

To describe the PG algorithm, let $x_{1:T} = (x_1^{b_1}, x_2^{b_2}, \ldots, x_T^{b_T})$ be the reference trajectory with ancestor indices $b_{1:T}$. For initialisation, we sample $x_0^k \sim p(x_0)$ for $k \neq b_1$ and set $\pi_0^k = 1/M$ for all k, where M is the number of particles in each particle set.

Now suppose we have a weighted sample from $p(x_t|y_{1:t})$ at iteration $t < T$, then each iteration of the PG sampling is conducted as below.

1. Sample $a_t^k \sim \text{Categorical}(\pi_t)$, $\forall k \neq b_t$;

2. Sample $x_{t+1}^k \sim p(x_{t+1}^k|x_t^{a_t^k})$, $\forall k \neq b_t$;

3. Set the weights as $w_{t+1}^k = p(y_{t+1}|x_{t+1}^k)$ and normalise the weights by $\pi_{t+1}^k = w_{t+1}^k / \sum_{i=1}^M w_{t+1}^i$ for all k sets.

Finally at iteration T, a single trajectory is selected by sampling the corresponding indices $k' \sim \text{Categorical}(\pi_T)$.

Thanks to the fact that PG uses $x_{1:T}$ as a reference trajectory in its SMC pass, it holds an invariance property that the exact target distribution is invariant, which ensures that PG is an unbiased sampling method [8].

2.2.2 Hamiltonian Monte Carlo

Hamiltonian Monte Carlo[2] is a Markov Chain Monte Carlo (MCMC) method for generating samples from a probability distribution for which direct sampling is difficult. It was originally devised by Simon Duane, A.D. Kennedy, Brian Pendleton and Duncan Roweth in 1987 [4].

MCMC is a type of general inference algorithm which constructs Markov Chain to generate samples from any distribution. Differently from the fact that the SMC method models the posterior density $p(X_k = x_k|y_{1:n})$, MCMC methods model the full posterior $p(x_{1:n}|y_{1:n})$, in which each sample in $X_{1:n}$ are dependent on the previous one according to a Markov process. (Note samples generated by SMC are independent on each other.)

The most popular MCMC algorithm is the Metropolis-Hastings (MH) method, and the HMC method is an advanced version of MH method applied in continuous state spaces, utilising the gradient of the target distribution function to accelerate the sampling process by reducing random walks [1].

Figure 2.2 gives an illustration of how HMC performs differently from MH.

As you can see in this figure, for a given number of sample number (30 here), HMC (in red) has a better exploration in the target distribution while the MH method (in green) get stuck in a corner of the distribution.

Another interpretation of the quicker convergence of HMC is that HMC reduces the correlation between consecutive samples by using a Hamiltonian evolution depending on states and momentums, thus proposing states with a higher acceptance criteria than the observed probability distribution [4, 9].

Metropolis-Hastings The MH method makes use of a proposal distribution $Q(x'; x^{(t)})$ dependent on the current state $x^{(t)}$ to draw samples for the next state. This proposal distribution can be any fixed density which is

[2]Hamiltonian Monte Carlo is historically named as hybrid Monte Carlo in origin [1].

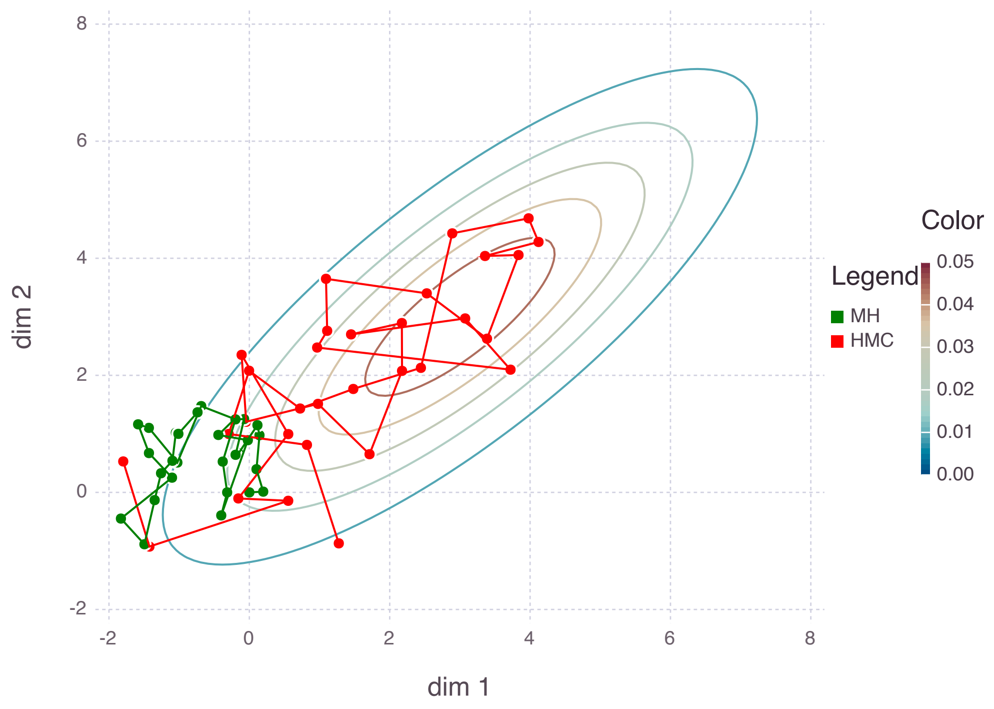


Figure 2.2: Comparison between MH and HMC methods.

feasible to draw samples from, e.g. a Gaussian distribution.

When the new state x' is proposed, depending on the probability ratio between the proposed state and the previous state

$$a = \frac{P^*(x')Q(x^{(t)};x')}{P^*(x^{(t)})Q(x';x^(t))}, \tag{2.17}$$

the following rule is used to decide whether to accept the new state or not:

- If $a \geq 1$, the new state is always accepted;
- Otherwise, the new state is accepted with a probability of a.

Note that as samples are generated using a proposal distribution depending on the previous state, samples generated by MH method are dependent on each other. However, one can proved that the convergence of MH method, which says with $t \to \infty$, the probability distribution of $x^{(t)}$ will be asymptotic to $P(x) = P^*(x)/Z$ [1].

Use of Hamiltonian Dynamics For many systems, the probability distribution $P(x)$ can be written in the form of

$$P(x) = \frac{e^{-E(x)}}{Z}, \tag{2.18}$$

where both $E(x)$ and its gradient $\frac{dE(x)}{dx}$ can be evaluated.

This $E(x)$ can be seen as the energy level of the corresponding system and is called the energy function. In HMC, the state space x is also augmented by an additional momentum variables p, and the sampling process then consists of two steps, which are described below.

1. Randomise the momentum variable p, leaving x unchanged;
   This can be done by drawing a new momentum p from the Gaussian density $\exp[-k(p)]/Z_k$, where $k(p) = p^T p/2$ is the kinetic energy. The sample in this step is always accepted.

2. Change both x and p using Hamilton dynamics defined by

   $$H(x, p) = E(x) + k(p). \tag{2.19}$$

   As the gradient of $E(x)$ determines how p changes, p in fact determines where x goes with the direction with the highest probability in the state space, or mathematically

   $$\dot{x} = p \tag{2.20}$$

   and

$$\dot{p} = -\frac{dE(x)}{dx}. \tag{2.21}$$

The sample obtained in this step is accepted depending on the difference in Hamilton dynamics

$$\Delta H = H_{\text{new}} - H_{\text{old}} \tag{2.22}$$

according to the MH rule:

- If $\Delta H < 0$, the new state is always accepted;
- Otherwise, the new state is accepted with a probability of $\exp(-\Delta H)$.

These two proposals (x and p) are then used to create asymptotically samples from the joint density below

$$P_H(x, p) = \frac{1}{Z_H} \exp[-H(x, p)] = \frac{1}{Z_H} \exp[-E(x)] \exp[-k(p)]. \tag{2.23}$$

As this joint density is separable, the desired distribution (i.e the marginal distribution of x) can be obtained by simply discarding p, which gives

$$P(x) = \frac{e^{-E(x)}}{Z}. \tag{2.24}$$

In terms of the time complexity, compared with MH, HMC lowers the time complexity from $O(N^2)$ to $O(N)$ by reducing the random walks [1].

2.2.3 Evaluating Inference Results

Samples generated from MCMC algorithms are correlated to each other, thus the ′real′ sample size is reduced by autocorrelation. Also, because MCMC methods are MC methods, there exist some estimation errors due to the limitation of Monte Carlo approximation [10]. How ′useful′ these samples are can be determined by two metrics, which are Effective Sample Size (ESS) and Monte Carlo Standard Error (MCSE).

Effective Sample Size ESS is a measure of how well a continuous chain is mixing. For a give chain $\{x_i\}_{1:n}$, ESS is defined by

$$\mathrm{ESS} = \frac{n}{1 + \sum_{k=1}^{\infty} \rho_k}, \tag{2.25}$$

where n is the total number of samples in the chain and ρ_k is the autocorrelation factor at lag k of the chain [11].

As ESS measures how many samples are effective in the chain, the larger this value is, the higher the sampling efficiency. Different MCMC samplers can be evaluated by generating the same number of samples and comparing the ESS for each sampling results.

Monte Carlo Standard Error MCSE is an estimate of the inaccuracy of MC samples. There are multiple ways to estimate MCSE, among which the batch mean method proposed in [12] is believed to be the most popular one.

The consistent batch means (CBM) is defined for $n = ab$ iterations as

$$\hat{\sigma}_g^2 = \frac{b}{a-1} \sum_{j=1}^{a} (\bar{Y}_j - \bar{g}_n)^2, \tag{2.26}$$

where $\bar{Y}_j = \frac{1}{b} \sum_{i=(j-1)b+1}^{jb} g(X_i)$ (for $j = 1, \ldots, a$), $\bar{g}_n = \frac{1}{n} \sum_{i=1}^{n} g(X_i)$ and g is any real-valued function [12].

Using the CBM, the MCSE is then estimated by

$$\mathrm{MCSE} = \frac{\hat{\sigma}_g^2}{\sqrt{n}}. \tag{2.27}$$

As MCSE measures the inaccuracy of MC samples, a smaller value of MCSE is an indicator of better sampling performance.

However, it has been argued that MCSE is generally unimportant when the goal of inference is parameters themselves rather than the expectation of parameters, in which case the ESS is would be a more important measure [13]. As some models are interested in using the expectation of parameters while some of them not, both MCSE and ESS will be used as performance metrics in this project.

2.3 Probabilistic Programming

Machine learning (ML) has become significantly important in recent year with its applications to various areas, including automatic driving, gene prediction and even playing the Go game [14]. One important approach to machine learning is probabilistic modelling, in which researchers defines a statistical model and extract information from data by doing inference on the model. It is an iterative process, where a model is proposed, fitted to some data and tweaked depending on its performance on the data. However, deriving and developing specific inference algorithms usually require expert knowledge in mathematics and computer science, which is challenging to researchers who are not experts in these areas and impedes the wide-range applications of ML in more specific fields.

Probabilistic programming languages are aimed to solve this problem by providing a considerably flexible framework to define probabilistic models and automating the model learning process using general inference algorithms. This frees researchers from writing complex models by hand and enables them to focus more on designing a suitable model by their insight.

In 2008, Goodman introduced an idea of a universal PPL that could express any generative non-parametric models, in which latent variables are unbounded [15, 16]. However, this idea was not implemented before the existing of Turing.

There are mainly two challenges when achieving a universal PPL framework in general, the first one of which is the development of efficient inference algorithms and the second one of which is how to utilise existing programming language infrastructure. In many PPLs, the second issues is tackled by embedding PPLs into some existing programming languages to use as much infrastructures as the ′master′ programming language has [16].

In the rest part of this section, how probabilistic models are represented by probabilistic programs is firstly introduced, and then three probabilistic models written in Turing and Stan (a popular PPL which also implements the HMC algorithm) are provided as illustrations of how models are written in practise. More concrete discussions on existing PPLs will be provided in Chapter 3.

2.3.1 Approaches to Probabilistic Programs

In PPLs, a user needs to write his or her mathematical model as a probabilistic program, which is essentially a computer program which describes a probabilistic distribution.

There are generally three approaches to express a probabilistic distribution as a program, which are the distributional approach, the randomised approach and the hybrid approach. Distributional and randomised approaches are inspired by work in [17] and popular in literature while the hybrid one is recently proposed in [16].

The Distributional Approach

Distributional probabilistic programs are deterministic functions to the inputs. They define concrete mappings between inputs and the target density.

One example of distributional probabilistic is the joint probability function of two inputs, e.g. θ and D in the example below

$$f : (\theta, D) \mapsto \pi_0(\theta)g_\theta(D). \tag{2.28}$$

In the Bayesian framework, this joint distribution could be the joint function of the prior $P(\theta)$ and the likelihood $P(D|\theta)$.

Many existing PPLs including Stan, Infer.NET and WinBUGS takes this approach [16].

The Randomised Approach

Randomised probabilistic programs are non-deterministic (randomised) functions to the inputs. For instance, the function below defines a randomised probabilistic program.

Listing 2.1: Exemplar Randomised Probabilistic Program

```
function f(a, b)
        pdf = p_0(a)
        z   = rand()    # draw a random value uniformly
        return pdf += g(a, b, z)
end
```

As the density returned by the function `f(a, b)` depends on the random value `z`, this function gives different returned values at each time it runs.

In the sense of that randomised probabilistic programs can generate data from some underlying generative models, randomised programs are also called data simulators [16].

The Hybrid Approach

Hybrid probabilistic programs are randomised probabilistic programs which define distributional mappings

$$\hat{f} : (\theta, D) \mapsto \pi_0(\theta)g_\theta(D|z), \tag{2.29}$$

where z is drawn from some distributions which supports sampling.

The hybrid approach actually combines the determinised approach and the randomised in the sense that it adds noise to the determinised function. This approach is able to describe any models in the Bayesian inference framework and is the approach taken by Turing.

As hybrid probabilistic programs evaluate some density functions with randomised intermediate values, they are usually called noisy evaluators [16].

2.3.2 Exemplar Probabilistic Programs

This section lists three probabilistic models written in two PPLs, Turing and Stan, as an illustration of how probabilistic models are expressed in practice. These three models will also be used for experiments in Chapter 5.

Gaussians with Conjugate Priors

The first model is a univariate Gaussian model with conjugate priors. Conjugate priors are useful to construct posteriors that own the same form after being updated by the Bayes' rule [18].

Mathematically in a univariate Gaussian model with conjugate priors, data points $x \in D$ are modelled by

$$x \sim \text{Normal}(\mu, \sqrt{\sigma}), \tag{2.30}$$

where $\sigma \sim \text{InverseGamma}(\alpha, \theta)$ and $\mu \sim \text{Normal}(0, \sqrt{\sigma})$.

Here in our example, the hyper-parameters of the inverse gamma distribution are set as $\alpha = 2$ and $\theta = 3$.

Listing 2.2 shows how this model is written in Turing and Listing 2.3 gives the corresponding code in Stan[3].

[3]The Turing version of this model is available at the home page of Turing (`https:`

Listing 2.2: The Gaussians Model in Turing

```julia
xs = [1.5, 2.0]                         # the data

@model gauss begin
  @assume s ~ InverseGamma(2, 3)        # define the variance
  @assume m ~ Normal(0, sqrt(s))        # define the mean
  for i = 1:length(xs)
    @observe xs[i] ~ Normal(m, sqrt(s))   # observe data points
  end
  @predict s m                          # predict s and m
end
```

Listing 2.3: The Gaussians Model in Stan

```julia
const gauss_data = [                    # the data
  Dict("N" => 2, "xs" => [1.5, 2.0])
]

const gauss_str = "                     # data definition
data {
  int<lower=0> N;
  real xs[N];
}
parameters {                            # parameter
    definition
  real<lower=0> s;
  real m;
}
model {                                 # model definition
  s ~ inv_gamma(2, 3);                  # define the variance
  m ~ normal(0, sqrt(s));               # define the mean
    xs ~ normal(m, sqrt(s));            # observe data points
}
"

gauss = Stanmodel(name="gauss", model=gauss_str);
```

`//github.com/yebai/Turing.jl`) and the corresponding Stan version is written by the student.

According to [19], there is an exact inference result for this model. In General, for a Gaussian model with priors $\mu \sim \text{Normal}(\mu_0, \sigma^2/k_0)$ and $\sigma^2 \sim \text{InverseGamma}(\alpha, \theta)$[4], the posterior means are

$$\bar{\mu} = \frac{k_0\mu_0 + n\bar{x}}{k_0 + n} \tag{2.31}$$

and

$$\bar{\sigma^2} = \frac{2\alpha + n}{2\alpha + n - 2}\frac{1}{2\alpha + n}(2\theta + \sum_i (x_i - \bar{x})^2 + \frac{nk_0}{k_0 + n}(\mu_0 - \bar{x})^2). \tag{2.32}$$

where n is the number of observations and $\bar{x}$ is the mean of observations.

In our example, with $n = 2$, $\bar{x} = 1.75$, $k_0 = 1$, $\alpha = 2$ and $\theta = 3$, this two means are

$$\bar{\mu} = \frac{1 \times 0 + 2 \times 1.75}{1 + 2} = \frac{7}{6} \tag{2.33}$$

and

$$\bar{\sigma^2} = \frac{2 \times 2 + 2}{2 \times 2 + 2 - 2}\frac{1}{2 \times 2 + 2}(2 \times 3 + 2 \times (\frac{1}{4})^2 + \frac{2 \times 1}{1 + 2}(\frac{7}{4})^2) = \frac{49}{24}. \tag{2.34}$$

This exact inference result will be used to validate the correctness of the HMC implementation in Section 5.1.

[4]This kind of priors is called Normal-inverse-Gamma (NIG) prior, which is a special case of Normal-inverse-chi-squared (NIX) prior. [19] gives the expressions of posterior means of NIX and the corresponding means of NIG is found by using the relation between the scaled inverse chi-squared distribution and the inverse gamma distribution, which says if $x \sim \text{ScaledInverseChiSquared}(\nu, \tau^2)$ then $x \sim \text{InverseGamma}(\frac{\nu}{2}, \frac{\nu\tau^2}{2})$.

Beta-Binomial Model

The second example is the beta-binomial model, which models data $x \in D$ as

$$x \sim \text{Bernoulli(p)}, \tag{2.35}$$

where $p \sim \text{Beta}(\alpha, \beta)$.

In our example, the parameters of the Beta distribution are set as $\alpha = 1$ and $\beta = 1$.

Listing 2.4 shows how this model is written in Turing and Listing 2.5 gives the corresponding program in Stan[5].

Listing 2.4: The Beta-Binomial Model in Turing

```julia
obs = [0, 1, 0, 1, 0, 0, 0, 0, 0, 1]    # the observations

@model betabinomial begin
  @assume p ~ Beta(1, 1)                # define the prior
  for i = 1:length(obs)
    @observe obs[i] ~ Bernoulli(p)      # observe data points
  end
  @predict p                            # predict of p
end
```

Listing 2.5: The Beta-Binomial Model in Stan (Julia Interface)

```julia
const betabinomial_data = [             # the observations
  Dict("N" => 10, "obs" => [0, 1, 0, 1, 0, 0, 0, 0, 0, 1])
]

const betabinomial_str = "
data {                                  # data definition
  int<lower=0> N;
```

[5]The Stan version of this model is adapted from the one at the home page of the Julia interface of Stan (`https://github.com/goedman/Stan.jl`) and the corresponding Stan version is written by the student.

```
  int<lower=0,upper=1> obs[N];
}
parameters {                               # parameter definition
  real<lower=0,upper=1> theta;
}
model {                                    # model definition
  theta ~ beta(1, 1);                      # define the prior
    obs ~ bernoulli(theta);                # observe data points
}
"

betabinomial = Stanmodel(name="betabinomial", model=
    betabinomial_str)
```

According to [18] there is an exact inference result for this model, which says generally for a beta-binomial model with a prior Beta(α, β), the mean of p is

$$\bar{p} = \frac{\alpha + N_1}{\alpha + \beta + N}, \tag{2.36}$$

where N_1 is the number of 1s in D and N is the total number of observations.

Here in our example, with $N_1 = 3$, $N = 10$, $\alpha = 1$ and $\beta = 1$, this expectation is

$$\bar{p} = \frac{1+3}{1+1+10} = \frac{1}{3}. \tag{2.37}$$

Again, this exact inference result will be used to validate the correctness of the HMC implementation in Section 5.1.

Logistic Regression

Logistic regression is a regression model where the dependent variable is categorical, which was firstly introduced in 1958 [20, 21]. Specifically in our example, the label of data points $x_i \in D \subset R^2$ follows a Bernoulli distribution

$$t_i \sim \text{Bernoulli}(y), \tag{2.38}$$

where $y_i = f(\beta_0 + \beta' x_i) = \frac{1}{1+\exp(-(\beta_0+\beta' x_i))}$ and t_i is the true label.

Here the function $f(x) = \frac{1}{1+exp(-x)}$ is called the logistic function and logistic regression is named because the probability is estimated by this logistic function. Also in this model, t_i and x_i has a linear relationship with weight vector $\beta = (\beta_1, \beta_2)'$ and bias β_0.

In the Bayesian framework, the the bias and the weight vector are given Gaussian priors as

$$\beta_i \sim \text{Normal}(0, \sigma^2). \tag{2.39}$$

The value of σ determines the fitting level of our model by controlling the magnitude of β_i.

Listing 2.6 shows how this model is written in Turing and Listing 2.7 gives the corresponding model in Stan.

Listing 2.6: The Logistic Regression Model in Turing

```
function f(x, beta)                     # Logistic function
  return 1 / (1 + exp(-(beta[1] + beta[2:3]' * x)[1]))
end

xs = Array[[1, 2], [2, 1], [-2, -1], [-1, -2]]  # data points
ts = [1, 1, 0, 0]                               # labels

alpha = 0.25                            # regularisation term
var = sqrt(1 / alpha)                   # variance of Gaussian prior
@model lr begin
  beta = Vector{Dual}(3)                # define a container for beta
  for i = 1:3                           # define priors
      @assume beta[i] ~ Normal(0, var)
  end
  for i = 1:4
    y = f(xs[i], beta)            # compute label
    @observe ts[i] ~ Bernoulli(y) # observe data point
  end
  @predict beta                   # output beta
end
```

Listing 2.7: The Logistic Regression Model in Stan

```
const lr_data = [              # training data
  Dict(
    "N" => 4,
    "xs_1" => [1, 2, -2, -1],
    "xs_2" => [2, 1, -1, -2],
    "ts" => [1, 1, 0, 0]
  )
]

const lr_str = "
data {                         # data definition
  int N;
  real xs_1[N];
  real xs_2[N];
  int<lower=0,upper=1> ts[N];
}
parameters {                   # parameter definition
  real beta_0;
  real beta_1;
  real beta_2;
}
transformed parameters {       # internal parameter definition
  real<lower=0,upper=1> ys[N];
  for (i in 1:N)
    ys[i] <- 1 / (1 + exp(-(beta_0 + beta_1 * xs_1[i] + beta_2 *
        xs_2[i])));
}
model {                        # model definition
  beta_0 ~ normal(0, 2);       # priors
  beta_1 ~ normal(0, 2);
  beta_2 ~ normal(0, 2);
  for (i in 1:N)
    ts[i] ~ bernoulli(ys[i]); # likelihood
}
"
```

In addition, a Bayesian neural network with a single neuron is essentially a logistic regression model. Therefore, this example can be also considered as a simple Bayesian network with only one neuron shown in Figure 2.3.

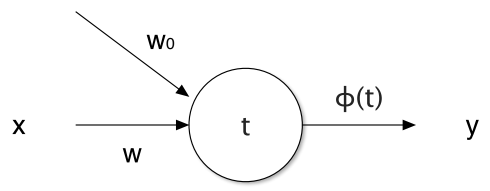


Figure 2.3: A neural network with one neuron.

As a neural net is usually trained using gradient descent (GD) methods, it would be interesting to compare the predictions from these two methods.

In general, GD updates parameters by moving them towards where the current gradient indicates. In our example, the update step is

$$\beta' = \beta + \alpha L(\beta), \tag{2.40}$$

where α is the learning rate and $L(\beta)$ is the loss function defined by

$$L(\beta) = \sum_i t_i \log(y) + (1 - t_i) \log(1 - y) + \frac{\alpha}{2}(\beta_0^2 + \beta_1^2 + \beta_2^2). \tag{2.41}$$

[1] indicates a relationship between the variance of Gaussian prior and the learning rate of GD, which is $\sigma^2 = 1/\alpha$. Also, the ′leapfrog′ step size in

HMC can also be chosen using the learning rate by $\epsilon = \sqrt{2\eta}$ [1]. These two relations can be used to set parameters to make two methods comparable and the experiment to compare Bayesian prediction against GD will be given in Section 5.4.1.

Chapter 3

Related Work

The concept of probabilistic programming is relatively newly emerging but it does experience a nearly 20 years' development history.

The first popular PPL is named WinBUGS in 2000, which can be used to describe graphical models, and utilises a Gibbs sampler to do inference [22]. After that many PPLs with different inference algorithms were developed. For example, Infer.NET from Microsoft makes use of message passing (2014); Stan from the Stan development team uses HMC as the inference algorithm; LibBi uses Particle MCMC (2013); and in 2014, Anglican, from Wood's group, consists various inference algorithms [16, 23, 24, 25].

This chapter gives more practical insight of how PPLs are implemented, and is structured as follow. In Section 3.1, Stan is selected to be discussed in detail as it also implements the HMC algorithm. Some features of Stan mentioned here will be linked to its performance later in Section 5.2. Then Section 3.2 will introduce the fundamental infrastructure of Turing in terms of the basic architecture, key components and the flow of how a probabilistic model is learnt.

3.1 An Existing HMC Implementation - Stan

Stan, named after the mathematician Stanislaw Ulam, who is one of the fathers of the MC method, is a C++ program to perform Bayesian inference [26]. The first version of Stan was released in 2012 and now it has a large and active development community as well as affluent documentations. There currently exist several interfaces of Stan, including command line, R, Python, Matlab, and Julia. Stan models can be defined in the host language as string and the host language can then call the Stan compiler to compile defined models into C++ programs [26].

To use Stan, a user defines a Stan program in its own syntax, which is similar to BUGS and Jags in the sense that it allows a user to write a Bayesian model in a convenient language whose code looks like statistics notation [22, 26, 27]. However, as the Stan program forces typings on variables, users also need to define the type of data and model parameters as well as intermediate parameters in the model, which makes model definition troublesome to some extent.

After a Stan model is defined, it is then compiled to a C++ program and runs along with data. Note that each time the model is amended, it requires to be compiled again. The compiling takes relatively long time compared with its fast sampling speed. As probabilistic modelling usually requires frequent amendments of models, this is argued to be a disadvantage of Stan. Some concrete compiling time of Stan programs will be given in Section 5.2.

The result output from Stan is a chain of samples of the posterior parameters in the model generated by NUTS, which is an adaptive variant of the HMC algorithm. As Stan supports only HMC, it cannot inference parameters in discrete space [26]. In addition to samples, Stan also generates and outputs useful statistics of samples such as mean, variance and ESS of the samples by default.

3.2 The Turing Infrastructure

Turing is an implementation of universal PPL ideas from [15] in Julia. In Turing, a probabilistic program can be defined using some probabilistic operations in a normal Julia program and this program can be executed by some general inference engines to learn the model parameters automatically. Generally speaking, compared with other PPLs which sacrifice expressivity for efficiency, Turing has better expressivity and meanwhile maintains good performance. This is thanks to the good performance of its master language, Julia, the design of its compiler and the efficient inference algorithms it uses [16].

Julia The programming language Turing based on is Julia, which is a high-performance dynamic programming language. Historically for numerical computing, theres was a dilemma where high level dynamic programs are usually productive to write but slow to run therefore prototypes in such languages have to be re-written to another language for speeding up [28]. However, as Julia is based on generic functions and utilises a rich type system which simultaneously enables an expressive programming model and successful type inference, it has both advantages in expressivity and speed performance for a wide range of programs [29]. Here the first advantage meets our expectation of PPLs being expressive and the second one ensures our inference engines do not suffer from unsatisfying performance due to the underlying programming language [28, 29]. Therefore, Julia is a desirable programming language to build a PPL on.

As probabilistic programs are essentially Julia programs and Julia is relatively expressive than other numerical programming languages, Turing also inherits the advantage of satisfying expressivity of probabilistic models. As you have seen in Section 2.3.2, compared with Stan, the syntax of Turing is simpler and easy-to-read. Especially Turing saves lines of codes by freeing users from defining the types of parameters in data and model as well as enables the Turing program to interact with other functions in Julia. Also, as

Julia is a dynamic programming language, Turing does not have the tedious compiling process when each time a model is changed.

3.2.1 The Framework

There are two major components in the Turing framework, which are the compiler and the samplers.

The compiler

In Turing, models are defined by a normal Julia program supported with three probabilistic operations, `@assume`, `@observe` and `@predict`, wrapped in a scope by `@model` (see Section 2.3.2 for some model examples). Section 4.1.1 will give the detail of the language design.

The defined model within `@model` will then be translated into a normal Julia program by the compiler, passed to a sampler as an expression when the sampling function `sample()` is called. The detail design of the compiler and how each operation works will be given in Section 4.1.2.

The samplers

Turing supports multiple samplers by wrapping the parameters of an algorithm into a specific type inherited from `InferenceAlgorithm` and wrapping the concrete algorithm into a type inherited from `Sampler`. The corresponding sampler will be automatically created by the sampling algorithm that user defines in the sampling process.

Up to the time when this thesis is submitted, Turing supports four inference algorithms, which are Importance Sampling (IS), Particle Gibbs (PG), Sequential Monte Carlo (SMC) and Hamiltonian Monte Carlo (HMC)[1]. These

[1]IS, PG and SMC are existing when the student joined the project. The student mainly contributes to implementation of the HMC sampler in this project. In addition, the student also re-written the IS sampling using the new data structure `ParticleContainer`,

samplers can be constructed by calling the respective algorithms with corresponding parameter(s) as follow

- IS: `IS(n_samples)`
- PG: `PG(n_particles, n_iterations)`
- SMC: `SMC(n_samples)`
- HMC: `HMC(n_samples, leapfrog_size, leapfrog_number)`

Among these four samplers, three of them, IS, PG and SMC, are particle based methods, in which samples are represented as particles and each particle needs to run a separate copy of the program independently. In order to improve sampling efficiency of these methods, coroutines, which is a similar technique to subroutines, are used to execute programs in Turing.

Coroutines have two advantages over other multi-process methods, which are

1. A coroutine can yield control multiple times during its execution and the result can be required either at that point or later;
2. There is no need to manage locks in the use of coroutine as there is no shared memory or concept of father processes [16].

This coroutine approach improves the sampling speed of IS and SMC to a great extent, but it is only helpful for the HMC algorithm to a limited extent. The reason is that in HMC each sample is dependent on the previous one, which means the algorithm has to wait for the program related to the previous sampling step to be finished. Luckily, within each iteration of HMC, it still requires to run multiple programs in parallel when computing the gradient information, where coroutines can still help speed up the sampling process.

The HMC sampler is the one this project focuses on and the corresponding design and implementation will be given in Section 4.2.

which is originally designed for the SMC sampler.

Sampling a defined model

When a model is defined, inference could be done by calling a function `sample()` with two arguments: a model and a sampling algorithm. This function will create a global sampler object, link it to Turing and execute this sampler.

For instance, if one would like to sample the Gaussian model `gauss` in Section 2.3.2 using a HMC sampler for 1000 samples with 'leapfrog' step size and 'leapfrog' step number being 0.5 and 15 respectively, the statement below could be used.

```
chain = sample(gauss, HMC(1000, 0.5, 15))
```

The return of `sample()` is the generated samples wrapped in a type called `Chain`. Each sample in a chain is a `Sample` type that stores the weight[2] and value of the sample and the `Chain` stores all samples as well as the log model evidence.

`Chain` is a general interface to fetch samples generated by all samplers in Turing. It has some intrinsic implementations to make it easy to extract inference results, e.g. samples are automatically weighted by their weights during extraction.

Using the Gaussian model as an example, one could extract all **weighted** samples of parameter `s` using a single indexing notation as below.

```
chain[:s]
```

3.3 MCMC Libraries in Julia

There are many MCMC packages available in Julia including `SimpleMCMC.jl`, `MCMC.jl` and `Mamba.jl` [3]. The first one supports doing inference on models

[2]Weights for non-IS based sampling algorithm like HMC are simply $1/n$, where n is the number of samples.

[3]`SimpleMCMC.jl` is available at `\url{https://github.com/fredo-dedup/SimpleMCMC.jl}`, `MCMC.jl` is available at `https://github.com/doobwa/MCMC.`

defined in a simple customised Julia expression using random-walk Metropolis and HMC; the second one supports applying inference algorithms (MH, HMC and slice sampling) on probability defined in the form of functions; and the third one allows users to define models in its own type and applies MCMC algorithms to them, which is similar to a PPL to some extent.

Here the package `Mamba.jl` also provides some convenient toolkits to diagnose the MCMC sampling result by computing MCSE and ESS, and thus this package is used in the experiment part of this project.

`jl` and `Mamba.jl` is available at `https://github.com/brian-j-smith/Mamba.jl`.

Chapter 4

Design and Implementation

This chapter provides information about the design and implementation of the HMC sampler in Turing[1], and is divided into two parts: the compiler and the sampler. The compiler is responsible for translating probabilistic models and the sampler is responsible for generating samples, which are discussed in Section 4.1 and Section 4.2 respectively.

4.1 Automating Bayesian Inference

Turing automates Bayesian inference by firstly defining a probabilistic model as a Julia program and then running this program using universal inference engines. This program is basically a normal Julia program extended with three probabilistic operations, namely `@assume`, `@observe` and `@predict`, wrapped in a `@model` scope (see Listing 2.2, Listing 2.4 and Listing 2.6 in Section2.3.2 for example). The design of the Turing language will be introduced in Section 4.1.1.

[1]The master repository of the Turing project is at `https://github.com/yebai/Turing.jl`. As the Turing project is still under development, implementation discussed in this thesis can be different in the future. For this reason, the code of the HMC implementation in this dissertation is stored in a separate branch, which is available at `https://github.com/xukai92/Turing.jl/tree/thesis`.

The probabilistic program is then be translated into a standard Julia program by a compiler, which involves using the metaprogramming technique, macro, in Julia. The translated program is essentially a noisy evaluators that can evaluate the density of probability distribution the model defines. The implementation of the compiler will be discussed in Section 4.1.2.

4.1.1 Language Design

Three probabilistic operations, `@assume`, `@observe` and `@predict`, are responsible for defining the prior probability, the likelihood and the priors to be outputted by the sampler respectively.

In detail, these three operations are supported with the syntax below.

- `@assume x ~ D`: declare that the (prior) variable `x` is drawn from the distribution `D`.
  Note: `x` will either be drawn from the distribution or be set using the current value stored in the sampler.

- `@observe y ~ D`: declare that the value `y` is observed to be drawn from the distribution `D`.
  Note: `y` is ought to have a concrete value in the current scope of the program.

- `@predict x`: declare that which prior(s) declared by `@assume` (e.g. `x` here) should be output from the inference engine.

Distributions here are declared in form of standard mathematical form, e.g. `Normal(0, 1)` or `Bernoulli(0.33)`[2].

Additionally, variables here can be annotated with additional arguments, `static` and `param`, passed in the distribution, e.g. `@assume mu ~ Normal (0, 1; static=true)`, to indicate their properties. In specific, these anno-

[2]The Julia package `Distributions` supports most of the common distributions. However, due to the fact the distributions in the package are not differentiable, a wrapper of common distributions are written by the student. This will be discussed in Section 4.2.2.

tations are aimed to be used as below.

- `static`: if this argument is set, it means that the existence of the corresponding variable does not rely on other variables. Therefore the variables exists in each execution of the program.
  Note: When this argument is set to `true` and the distribution is differentiable, the corresponding variable could be efficiently sampled by samplers like HMC.

- `param`: if this argument is set, it means that the corresponding variable can be treated as a model parameter.
  Note: When this argument is set to `true`, the corresponding variable could be efficiently sampled by samplers like SMC2.

This annotation feature is aimed to support a future Gibbs sampler, which combines PG and HMC by sampling discrete variables by PG and continuous variables by HMC respectively.

4.1.2 Compiler: Translating Operations using Macros

In general, a compiler translates one programming language into another. Here the compiler in Turing translates probabilistic programs into normal Julia programs by transforming the three probabilistic operations into normal Julia statements. This transformation is achieved by a metaprogramming technique in Julia called macro.

How macros work

Macros provide a way to include generated code in the final body of a program. They can map a tuple of arguments to a returned expression. Differently from functions which are executed in runtime, macros are executed in parse time [29, 30]. This allows programmers to generate and include pieces of customised codes before the full program is executed.

The macro `change_op` below illustrates how macros work and the difference between macros and functions.

```julia
macro change_op(ex)
  # Change the operation to multiplication
  ex.args[1] = :*
  # Return an expression to print the result
  return :(print($(ex)))
end
```

This macro is aimed to change the operation of the input expression to multiplication. Giving an argument `1 + 2`, it gives result as below.

```julia
@change_op 1 + 2 # call the marco
> 3              # the result of expression "print(1 * 2)"
```

Here, this macro takes an expression `1 + 2` as input and returns another expression `print(1 * 2)` to be evaluated by Julia. This transformation is impossible by using a function because, for functions, the expression `1 + 2` will be executed when being passed into the function and what the function sees is only the result value `3`. This means functions are not able to manipulate incoming expressions in their routines thus macros are necessary for a compiler.

The `@assume` macro

Using

```julia
@assume m ~ Normal(0, 1; static=true)
```

as an example, the macro `@assume` will

1. Check if there is any additional argument `static` or `param`. If so, it will do settings regarding to the argument, and discard this argument inside the distribution expression, i.e. turning `Normal(0, 1; static=true)` into `Normal(0, 1)`.

2. The `Distribution` type will be converted to a custom wrapper called `dDistribution` (differentiable distribution) by appending a letter 'd' to

the corresponding argument in the expression, i.e. turn `Normal(0, 1)` to `dNormal(0, 1)`.

3. The macro will return an expression which calls a function named `assume()` depending on the type of prior, which is explained as below.

If the prior is a single variable (like `m` in the example above), the expression will become

```
1 m = Turing.assume(
2       Turing.sampler,  # the sampler defined by the user
3       dNormal(0, 1),   # the distribution
4       PriorSym(:m)     # the prior identity
5     )
```

If the prior is an indexed array or a dictionary, such as `priors[i]` in the first code block below, the expression will become the second code block below.

```
1 priors = zeros(2)      # initialise an array to store priors
2 for i = 1:2
3   @assume priors[i] ~ Normal(0, 1)
4 end
```

```
1 m = Turing.assume(
2       Turing.sampler,  # the sampler defined by the user
3       dNormal(0, 1),   # the distribution
4       PriorArr(:(priors[i], :i, 1))  # the prior identity
5     )
```

Here the value `1` in the prior identity construction (Line 4 of the second code block above) is the concrete value at a specific iteration of the loop (the loop between Line 2 to 4 in the first code block above).

The aim of using types `PriorSym` and `PriorArr` is to pass an identity of each prior so that they can be replayed by the HMC sampler specifically. Actually this way of replay is not perfect and the replay issue will be specifically discussed in Section 4.2.3.

The `@observe` macro

Exactly similarly to `@assume`, `@observe` will firstly do settings according to additional arguments, discard the annotation from distribution construction and then turn `Distribution` into `dDistribution`. The difference happens in the third step, where `observe` will pass the concrete value of the variable observed to a function called `observe()`.

For instance,

```
xs = [1.5, 2.0]

@observe xs[1] ~ Normal(0, 1)
```

will become

```
Turing.assume(
  Turing.sampler,  # the sampler defined by the user
  dNormal(0, 1),   # the distribution
  1.5              # the concrete value of observation
)
```

The `@predict` macro

Using

```
@predict m
```

as an example, if the value of `m` is `1.1` at the time when this macro is called, the statement above will become

```
Turing.predict(
  Turing.sampler,  # the sampler defined by the user
  :m,              # the symbol of variable
  1.1              # the current value of variable
)
```

The `@model` macro

The macro `@model` wraps all the original and generated statements in its scope into an expression `ex`, stores this expression in the Turing's global scope as

```
TURING[:modelex] = ex
```

and returns this expression.

4.2 Hamiltonian Monte Carlo in Turing

The general inference engine developed in this project uses the standard HMC algorithm. In this section, the implementation of the HMC sampler in Turing is given in detail, with how automatic differentiation works, both generally and inside the sampler, explained in Section 4.2.2.

4.2.1 The Algorithm in Detail

According to theory of HMC in Section 2.2.2 and referring to a Octave version of HMC from [1], the pseudocode code of HMC in Julia style in shown in Algorithm 1.

Notice here in this algorithm, initial states are required. This is done by simply drawing samples from priors and using these samples as initial states.

In general, the algorithm in Algorithm 1 can be directly applied to situation where the energy function and the corresponding gradient function are available. However, as a probabilistic program is not essentially a function but a noisy evaluator, this algorithm cannot be directly applied to the Turing framework. The way to evaluate the energy function will be discussed in this section. Nevertheless, it is also not easy to get the gradient information from the program, which is the main challenge of this project. The solution to this challenge will be discussed in Section 4.2.2.

Algorithm 1: The Hamiltonian Monte Carlo Algorithm

```
Data: sample number n, 'leapfrog' step number $\tau$, 'leapfrog' step size
      $\epsilon$, initial state x, energy function E(), gradient function gradE()
Result: n samples
for $i = 1$ to $n$ do
    p = randn(length(x));        // draw momentum from Normal(0, 1)
    oldH = p' * p / 2 + E(x);    // record old Hamiltonian
    oldx = x;                    // record old state
    // Make $\tau$ 'leapfrog' steps
    g = gradE(x);                // evaluate gradient
    for $t = 1$ to $\tau$ do
        p − = $\epsilon$ * g / 2;    // make a half step for momentum
        x + = $\epsilon$ * p;        // make a full step for state
        g = gradE(x);            // evaluate gradient
        p − = $\epsilon$ * g / 2;    // make a half step for momentum
    end
    // Decide wether to accept the proposal state or not
    H = p' * p / 2 + E(x);       // compute new Hamiltonian
    dH = H − oldH;               // compute the difference in Hamiltonian
    if $dH < 0$ then
        acc = true
    else
        if rand() < exp(-dH) then
            acc = true
        else
            acc = false
        end
    end
    if $\neg acc$ then
        x = oldx;                // rewind if rejected
    end
end
```

It is worth to be mentioned that in each iteration the number of evaluation of `E(x)` is $O(1)$ (Line 3 and Line 14) and the number of evaluation of `gradE(x)` (Line 6, and Line 10 in a τ times loop) is $O(\tau)$. So in the whole sampling process these two complexities are $O(n)$ and $O(n\tau)$ respectively.

Evaluating `E(x)`

According to Section 2.2.2, there is a relation between energy and probability as described by Equation 2.18, which can be written as

$$E(x) = -\log(P(x) \times Z), \tag{4.1}$$

where $P(x)$ is the posterior distribution $P(\theta|D)$ that we are interested in, i.e. $P(x) = P(\theta|D)$.

According to the Bayes' rule in Equation 2.1, we have

$$P(\theta|D) = P(D, \theta)/Z, \tag{4.2}$$

where $P(D, \theta)$ is the joint probability of D and θ.

By plugging Equation 4.2 into Equation 4.1 with $P(x) = P(\theta|D)$, we have

$$E(x) = -\log(P(D, \theta)). \tag{4.3}$$

That is to say, in the sense of that the only required information by the HMC algorithm are the evaluations of `E(x)` and `gradE(x)`, it actually only needs to extract the log-joint probability and the corresponding gradient from the probabilistic program.

Also, by assuming data point are conditionally independently drawn given θ ($x \perp x'|\theta$ for any $x, x' \in D$) and using the product rule, we have

$$\begin{aligned}\log(P(D,\theta)) &= \log(\prod_{x \in D} P(x|\theta)) + \log(\prod_{\theta_i \in \theta} P(\theta_i|\theta_{-i})) \\ &= \sum_{x \in D} P(x|\theta) + \sum_{\theta_i \in \theta} P(\theta_i|\theta_{-i})\end{aligned} \quad . \tag{4.4}$$

That is to say, when we run the probabilistic program, we can compute the corresponding log-joint probability by accumulating log-likelihoods and log-priors.

Therefore, the evaluation of `E(x)` in Algorithm 1 can be achieved by two steps

1. Run the model by accumulating log-probability from each statement;
2. Collect the negative log-joint accumulated by the model.

4.2.2 Computing Gradient Information using Automatic Differentiation

As discussed in Section 4.2.1, the gradient function is not directly available therefore it requires a way to find the gradient information, which is achieved by Automatic differentiation in this project.

AD is a technique for calculating derivatives of any numeric function. Compared with other techniques of finding gradient such as Numerical Differentiation and Symbolic Differentiation, AD has advantages of being both efficient and accurate [31]. Thus AD is used in this project to obtain the gradient information of any probability distribution, which is the key ingredient of HMC.

The core idea behind AD is a new type of numbers call dual numbers, which will be introduced next, followed by an example and detailed application of AD in Turing.

Dual Numbers

The mathematics behind forward mode AD are dual numbers, which are defined as formal truncated Taylor series in the form of

$$v + \dot{v}\epsilon. \tag{4.5}$$

where v is called the real part and $\dot{v}$ is called the dual part. Note that any non-dual number v can be viewed as a dual number $v + 0\epsilon$.

By defining $\epsilon^2 = 0$ on dual numbers, we have

$$(v + \dot{v}\epsilon) + (u + \dot{u}\epsilon) = (v + u) + (\dot{v} + \dot{u})\epsilon \tag{4.6}$$

and

$$(v + \dot{v}\epsilon)(u + \dot{u}\epsilon) = (vu) + (v\dot{u} + \dot{v}u)\epsilon, \tag{4.7}$$

where we can see in both of them, the coefficients of ϵ have the same results as what the symbolic differentiation rules give[3].

This means that we can use dual numbers as data structures and evaluate functions of dual numbers by

$$f(v + \dot{v}\epsilon) = f(v) + f'(v)\dot{v}\epsilon, \tag{4.8}$$

where both the real and dual part are passed.

Also the chain rule of differentiation holds, which is illustrated in Equation 4.9 below.

[3]Symbolic differentiation rules are $\frac{d}{dx}(f(x) + g(x)) = \frac{d}{dx}f(x) + \frac{d}{dx}g(x)$ and $\frac{d}{dx}(f(x)g(x)) = (\frac{d}{dx}f(x))g(x) + f(x)(\frac{d}{dx}g(x))$.

$$\begin{aligned} f(g(v + \dot{v}\epsilon)) &= f(g(v) + g'(v)\dot{v}\epsilon) \\ &= f(g(v)) + f'(g(v))g'(v)\dot{v}\epsilon \end{aligned}. \tag{4.9}$$

Here we can see the coefficient of ϵ in the second line of in Equation 4.9 has the same form as the derivative of any composition of functions.

So far we assume we know f' and g', therefore it is necessary to know the gradient functions of some elementary functions in advance. In fact, the development of AD is mainly about handling dual number operations under elementary functions. Luckily, there is a package in Julia implementing dual numbers operations called `DualNumbers`, which is used in the project.

In summary, we can use dual numbers as the basic data structure with elementary functions implemented to support passing through dual numbers according to Equation 4.6, 4.7 and 4.8. By this way, the derivative of any function $f()$ of interest is given by

$$\left.\frac{df(x)}{dx}\right|_{x=v} = \text{epsilon-coefficient}(\text{dual-version}(f)(v + 1\epsilon)). \tag{4.10}$$

Also notice, as long as there is no arithmetic operations done on dual numbers, AD can be applied to any program since the data structure is unchanged and the differentiation will continue under the dual number operations.

In practise, when using AD to find the derivative of a function w.r.t a specific variable (say a real number x), it is necessary to set this variable in dual form with the dual part being 1, i.e. $x + 1\epsilon$, and then evaluate the function under dual operations. This way of using dual numbers is called the forward mode of AD and is the current implementation of AD in Turing.

The forward mode of AD requires n times evaluation of a program or a function to get the derivatives of n variables. There is also another form of AD, which is called the reverse mode. This way of AD requires only one evaluation of target programs or functions to compute all the gradient

Table 4.1: Example of evaluating an expression with dual numbers.

Step	Dual Number	Evaluation
0	$v_0 = x$	$3 + 1\epsilon$
1	$v_1 = v_0 - 1$	$(3-1)+(1-0)\epsilon = 2+1\epsilon$
2	$v_2 = f_1(v_1)$, $f_1(x) = x^2$ and $f_1'(x) = 2 \times x$	$(2)^2 + 2 \times 2 \times 1\epsilon = 4 + 4\epsilon$
3	$v_3 = -\frac{1}{2}v_2$	$-\frac{1}{2}\times 4 + -\frac{1}{2}\times 4\epsilon = -2-2\epsilon$
4	$v_4 = f_2(v_3)$, $f_2(x) = \exp(x)$ and $f_2'(x) = \exp(x)$	$\exp(-2) + \exp(-2) \times (-2)\epsilon = 0.135 - 0.271\epsilon$

information at once, which is more complex but could potentially accelerate the sampling of HMC. Stan uses this reverse-mode in its implementation of AD [32].

Example

Assume we are interested in finding

$$\left.\frac{df(x)}{dx}\right|_{x=3}, \tag{4.11}$$

where $f(x)$ is an unnormalised Gaussian with mean 1 and variance 1:

$$f(x) = \exp(-\frac{(x-1)^2}{2}). \tag{4.12}$$

We can evaluate $f(x)$ with dual numbers using Equation 4.8 and Equation 4.9. The evaluation steps are shown in Table 4.1 with the corresponding computational graph shown in Figure 4.1.

According to Equation 4.10, the target derivative is the coefficient of ϵ of v_4 in Table 4.1, thus

$$\left.\frac{df(x)}{dx}\right|_{x=3} = -0.271. \tag{4.13}$$

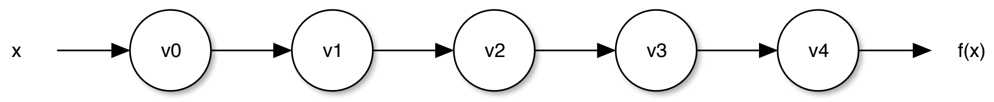


Figure 4.1: Computational graph of evaluating the function in Equation 4.12.

For verification purpose, the symbolic differentiation result is calculated as below.

$$\begin{aligned}\left.\frac{df(x)}{dx}\right|_{x=3} &= \exp(-\frac{(x-1)^2}{2})(-x+1)\Big|_{x=3} \\ &= \exp(-\frac{4}{2})(-3+1) \\ &= -2\exp(-2) \\ &= -0.271\end{aligned} \quad , \tag{4.14}$$

It can be seen that it gives exactly the same result as AD.

Evaluating `gradE(x)`

By using the forward mode of AD, the evaluation of `gradE(x)` in Algorithm 1 can be achieved by four steps

1. Set the dual part of the variable we want to get gradient w.r.t to 1;
2. Run the model by accumulating log-probability from each statement;
3. Collect the dual part of the negative log-joint accumulated by the model as the gradient;
4. Reset the dual part of the variable back to 0.

Here in Step 2, it requires passing through variables in `Dual` type to the density functions from the `Distributions` package, which is unfortunately not supported by the package. Therefore, a custom wrapper of the distribution package, `dDistritbuion`, is written in the way that the density functions are manually written by the student in normal Julia way and other functions (like `rand()`) are passed to the corresponding ones from the `Distributions` package. This is only a compromise to this issue and a more ideal way is to give a patch to the `Distributions` package to make it support `Dual` type.

4.2.3 The HMC Sampler

With all key components introduced in Section 4.2.1 and Section 4.2.2, it is possible to show the implementation of the HMC sampler now.

To begin with, as introduced in Section 3.2, samplers in Turing are wrapped by a type inherited from the `Sampler` type, including the corresponding algorithm by a type inherited from `InferenceAlgorithm`.

For the HMC sampler, a type `HMC` is inherited from `InferenceAlgorithm`, containing three attributes[4]:

- `n_samples` - number of samples (`Int64`)
- `lf_size` - leapfrog step size (`Float64`)
- `lf_num` - leapfrog step number (`Int64`)

and the sampler type `HMCSampler` is is inherited from `Sampler`, containing

- `alg` - the HMC algorithm (`HMC`)
- `model` - the model expression (`Function`)
- `samples` - an array to store concrete samples (`Array{Sample}`)

[4]What inside the brackets for each attribute listed is the type of this attribute in Julia.

- `logjoint` - log-joint probability of data (`Dual{Float64}`)
- `predicts` - a dictionary to store outputs (`Dict{Symbol, Any}`)
- `priors` - a dictionary to store values of all priors (`Dict{Any, Any}`)
- `first` - a flag to tell if the program runs for the first time (`Bool`)

The HMC sampler has corresponding `assume()`, `observe()` and `predict()` functions that are called in the translated probabilistic program, as well as its main loop inside the `run()` function.

`assume()`

The first step of the `assume()` function is to produce priors, which performers differently depending on the `first` flag. If the program runs for the first time, priors will be drawn from the prior distributions, converted into a `Dual` type and store in the dictionary `priors`; if it is not the first time, priors will be fetched from `priors`, which is called the replay of priors. After that in the second step, the value of prior will be used to compute the log probability density, which is then accumulated to `logjoint`.

Here the replay of priors is done by using the corresponding symbols as keys to store and fetch values to and from a dictionary. To be more specific, if the prior is stored in a single variable (say `s`), the key will be `:s`; if the prior is stored in some position within an array (say `p[i]`), the key will be `p[1]`, where `i` is converted to its concrete value when the prior is called (`1` here). The passing and conversion of prior variables is done by a costumed type `Prior`, which determines the type of priors and constructs corresponding sub-type of `Prior` (`PriorSym` for single variables and `PriorArr` for arraies or dicionaries) in compiling time and convert the `Prior` to the corresponding symbol in the runtime.

In fact this way of replay is not necessary in most of the scenarios because for a normal probabilistic model the orders of priors being generated are always the same in each time the program runs. In such senerios, a more simpler implementation to replay priors is to simply store them in an array in the order of being called and fetch them in order as well. However, since Turing allows users to use branches and loops when defining probabilistic programs, the order of priors being called may differ if there are branches existing. In such cases, this new implementation of replaying priors is useful.

In fact, this implementation still has problems when the container of priors being more complex, e.g nested arrays or customised types. A more universal approach to solve the replay issue is to combine these two methods above. To be more specific, Julia can generate an ID for each macro it runs. The IDs of macros can be used as keys to store the corresponding priors for each statement in a dictionary. Additionally, the data structure inside each key is designed to be a linked list. This allows priors defined in a loop to be stored in the dictionary with the same key but different indices in a list. This way of prior replay can support priors stored in any Julia container but has not been implemented due to the limitation of time.

`observe()`

The `observe()` function simply computes the log density of the observed data point and adds it to `logjoint`.

`predict()`

The `predict()` function stores the **real part** of priors to be predicted in `predicts`. This makes sure that priors are only in `Dual` type within the sampling process of HMC and all input/output (I/O) interfaces only interact with real numbers.

`run()`

The main body of the sampler follows the algorithm description in Algorithm 1 with initial states drawn from priors and `E(x)` and `gradE(x)` evaluated by the ways introduced previously in this section. One thing to mention here is that the `logjoint` of the sampler needs to be re-set to 0 manually at some point after each time the program runs. This is because the value needs to be used after the evaluation of the program but also needs to be 0 before each evaluation.

Additionally, as the HMC algorithm implemented in this project is an unbounded version, the program would fail to run if the ′leapfrog′ step makes a variable out of the domain of definition of the corresponding distribution. For instance, the variance of a normal distribution should be greater than 0 otherwise Julia will give domain error when computing the density. This issue is currently solved by a naive method, in which the HMC step is abandoned and restarts when such error occurs. However, if the parameters of HMC is set with some extreme values (e.g. very large step size), the program will fail nearly every time. Therefore, a threshold on the number of re-runs is set and the HMC sampler will stop and give corresponding error prompt if there are too many re-runs.

When the sampling is completed, the re-run number as well as the accept ratio in the HMC algorithm itself, will be prompted to users to help users evaluate the sampling result as well as tune the HMC settings.

Chapter 5

Evaluation

This chapter is aimed to give evaluations of the HMC sampler implemented in this project, in terms of correctness, performance and robustness, which are shown in Section 5.1, Section 5.2 and Section 5.3 respectively. Also, some more interesting experiments were conducted with findings given in Section 5.4.

5.1 Validation of Correctness

In order to validate the correctness of the HMC implementation, the result of three models in Section 2.3.2 obtained from the HMC sampler are compared with that of the same models from other samplers in Turing and Stan. These results are also compared with the exact inference result when available.

As the inference results are non-deterministic, in order to reduce the experimental error, 100 chains were generated by each sampler and the corresponding expectations of each variable were calculated and recorded.

Table 5.1 shows the inference results using different samplers (and exact inference when available), and Table 5.2 gives the corresponding settings samplers used.

Table 5.1: Inference results of different samplers.

Sampler	Gaussian		Beta-binomial	Logistic regression		
	$\bar{s}$	$\bar{m}$	$\bar{p}$	$\bar{w}_0$	$\bar{w}_1$	$\bar{w}_2$
HMC	2.00	1.16	0.330	-0.0112	1.70	1.72
PG	2.02	1.20	0.313	-0.164	1.78	1.72
SMC	1.98	1.15	0.302	0.806	1.24	1.23
HMC (Stan)	1.98	1.15	0.330	-0.00283	1.71	1.72
Exact	$2.041\dot{6}$	$1.1\dot{6}$	$0.33\dot{3}$	-	-	-

Table 5.2: Parameter settings of samplers in Table 5.1.

Sampler	Parameter	Gaussian	Beta-binomial	Logistic regression
HMC	N	1000	1000	5000
	ϵ	0.55	0.1	0.1
	τ	4	2	5
PG	N	500	500	1000
	τ	100	200	200
SMC	N	1000	4000	5000
HMC (Stan)	N	1000	1000	1000
	$\bar{\epsilon}$	0.55	1.04	0.38
	$\bar{\tau}$	3.9	2.5	7.3

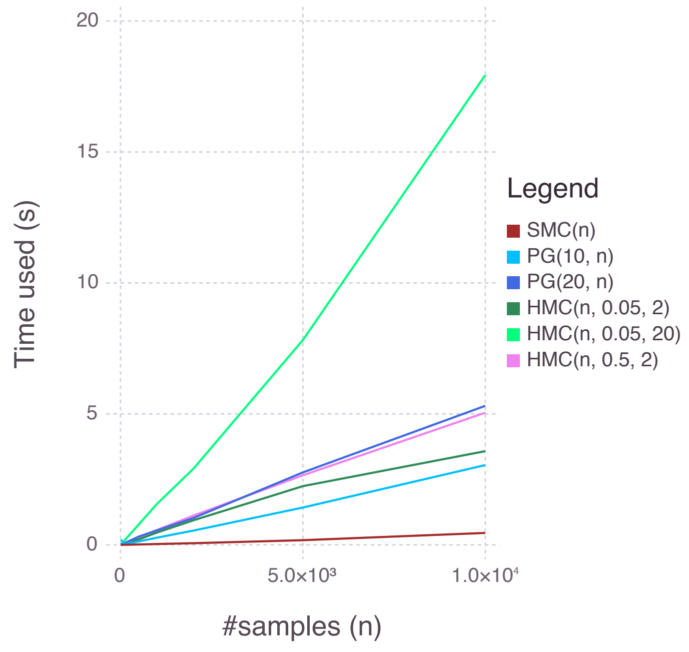
It can be seen for all models tested, the inference results of the new HMC sampler are consistent with those of others, which means the implementation is valid.

5.2 Performance

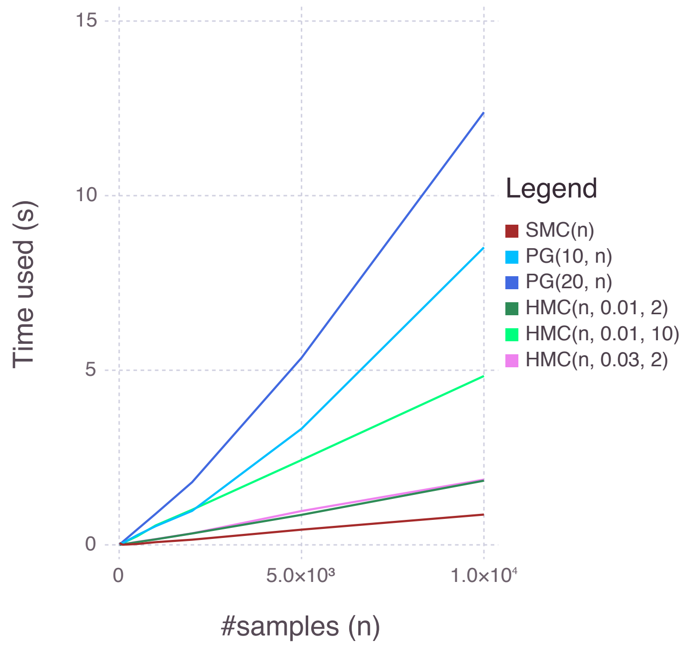
In this section, the HMC sampler is firstly compared against other samplers within Turing, and then against Stan.

5.2.1 Comparison within Turing

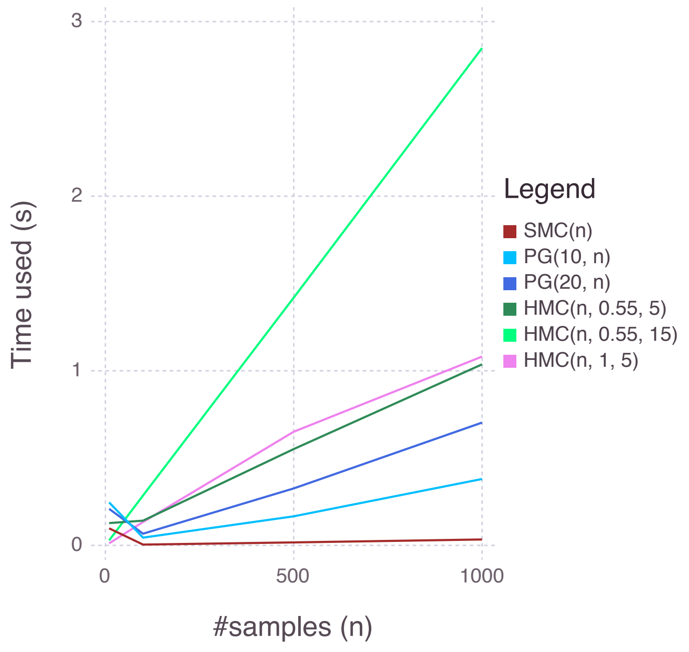
Samplers within Turing were given different parameter settings to perform inference and the time used by samplers to generate samples with number of samples varying for the three models in Section 2.3.2 are shown in Figure 5.1.



(a) The Gaussian model.



(b) The Beta-Binomial model.



(c) The logistic regression model.

Figure 5.1: Time used by different sampling algorithms with number of samples varying for three models.

For the result of Gaussian model in Figure 5.1a, as you can see, SMC uses the least time among all of the samplers while, generally speaking, PG and HMC show similar speed. In fact, the time used by PG and HMC is dependent on the parameters used by the algorithm: more particles and iterations in PG slow the speed a lot and more 'leapfrog' steps also lead to significant increase in sampling time. Also, as PG is based on SMC, there is actually a relation between the time used by PG and SMC: if the time used by `SMC(n)` is t, the time used by `PG(k, n)` is kt.

In terms of the Beta-Binomial model in Figure 5.1b, similarly to the Gaussian model, SMC is the fastest among all of the samplers. However in this example, HMC shows faster speed than PG with the same settings.

Regarding the results of the logistic regression model in Figure 5.1c, one setting of PG (in light blue) performs the best while all HMC settings performs extremely badly. This is argued to be the drawback of the forward mode of AD as the logistic regression model has more variables than the first two models.

To emphasise, because the performance of both PG and HMC is highly dependent on its parameter settings, it is difficult to draw precise conclusions on the performance of HMC here. However, it can still be concluded from these three figures that, generally speaking, the performance of the HMC sampler is acceptable in the Turing framework.

5.2.2 Comparison against Stan

As the inference engine in Stan is also based on HMC, the inference result from Stan can be used to compared with the HMC sampler in Turing. Note that the HMC algorithm implemented in Stan is not the standard HMC but a optimised version called NUTS.

As noticed in the previous experiments, the performance of HMC is highly sensitive to the 'leapfrog' step size ϵ and 'leapfrog' step number τ, therefore it would take a long time to tune these two parameters to achieve accept-

Table 5.3: Comparison of HMC samplers in Turing and Stan (the Gaussian model).

PPL	s		m		$\bar{\epsilon}$	$\bar{\tau}$	Time
	MCSE	ESS	MCSE	ESS			
Turing	0.195	181	0.027	825	0.55	4	0.594s
Stan	0.106	356	0.046	378	0.55	3.9	0.180s

able sampling performance. Thankfully NUTS solves this problem by using a recursive algorithm to build a set of likely candidate points that explores the target distribution well, automatically stopping sampling process when it starts to stuck and retrace its steps [33]. And empirically, NUTS can achieve almost the same level of efficiency as well-tuned HMC algorithms without any turning work for most of the models [33]. In addition, for some particular probabilistic programs with branches, as different HMC parameters are needed for different branches, NUTS would outperform standard HMC algorithm to a great extent.

As NUTS automatically sets ′leapfrog′ step size and ′leapfrog′ step number, in order to get relatively reasonable results, in our experiments, models were firstly learnt in Stan and the means of ′leapfrog′ step size and ′leapfrog′ step number provided by the result from Stan were used as the parameters for the HMC sampler in Turing when feasible[1].

To compare the HMC samplers in Turing against that in Stan, the models in Section 2.3.2 were used and inference were done with **1000** samples for 100 times for each model in each language. Generated samples were then built into chains to compute MCSE and ESS by the `Mamba.jl` package.

The average statistics of corresponding inference results and parameters were recorded for the three models, and are shown in Table 5.3, Table 5.4 and Table 5.5 respectively.

The sampling efficiency, measured by the MCSE and ESS for 1000 samples,

[1]Some of the parameters used by Stan is not suitable for the HMC sampler in Turing because Turing implements unbounded HMC. In some situations, large step size will cause too many rejections from variables out of range.

Table 5.4: Comparison of HMC samplers in Turing and Stan (the beta-binomial model).

PPL	p		$\bar{\epsilon}$	$\bar{\tau}$	Time
	MCSE	ESS			
Turing	0.00106	208	0.1	2	0.235s
Stan	0.00653	459	1.0	2.5	0.172s

Table 5.5: Comparison of HMC samplers in Turing and Stan (the logistic regression model).

PPL	β_0		β_1		β_2		$\bar{\epsilon}$	$\bar{\tau}$	Time
	MCSE	ESS	MCSE	ESS	MCSE	ESS			
Turing	0.0488	903	0.0465	871	0.0449	895	0.55	5	1.25s
Stan	0.0728	558	0.0722	503	0.0709	515	0.55	5.4	0.215s

varies from model to model and even parameter to parameter. For the Gaussian, Turing has better sampling efficiency for the parameter m but worse for s. This is interesting in the sense that neither Stan nor Turing wins in both and also indicates that some variables may be more sensible to the parameter settings in Turing, which will be further experimented in Section 5.4.2. For the beta-binomial model, Stan outperforms Turing with an obvious advantage. For the logistic regression model, Turing shows better efficiency in all three parameters. This loss of efficiency in Stan may be due to its implementation of NUTS, which automates the tuning process but sacrifices the sampling efficiency to some extent. However notice that Turing still takes longer sampling time and requires tuning in practise.

In addition to the difference caused by algorithms, Stan's optimisation also makes it faster. To be more specific, Stan's implementation of HMC has two significant advantages over the implementation in Turing. One is that Stan uses the reverse mode of AD, which requires only one evaluation of the program to evaluate the gradient of all variables. Another is that all numerical computations in Stan are vectorised, which improve the fundamental running speed.

Table 5.6: Comparison between Turing and Stan (ESS per second)

Sampler	Parameter	Turing	Stan	Ratio (Stan/Turing)
Gaussian	s	305	1977	6.48
	m	1389	2100	1.51
SMC	p	885	2269	2.56
Logistic Regression	β_0	722	2595	3.59
	β_1	697	2340	3.35
	β_2	716	2395	3.34

In terms of the concrete time, Stan shows an obvious advantage over Turing, with about 6 timers faster than Turing for the logistic regression model as highest. However, as Stan actually needs to compile its model to C++ each time the model changes, it will take a relative long time to run a new model (or a newly-amended model). For instance, it takes 5.06s to compile the Gaussian model in Stan (via Julia interface) to C++ program, 5.66s for the beta-binomial model and 4.87 for the logistic regression model. Luckily Turing does not have such a drawback, which means if the user iterates his or her model very frequently, Turing would show its advantage.

Taking the time into account, it is also worth the measure the sampling efficiency by the number of ESS per second, which is shown in Table 5.6.

According to this table, Stan shows an obvious advantage in sampling efficiency. Despite of the compiling time for models in Stan, Stan is concluded to has a much faster HMC implementation than Turing. Some possible approaches to improve the sampling efficiency of the current HMC implementation in Turing will be discussed in Section 6.2.

5.3 Robustness

This section is to discuss how the HMC sampler performs differently with the number of samples and variables increasing.

Increasing the number of samples In order to evaluate how the sampler performs with the number of samples increasing, three HMC results of different models are taken from Figure 5.1 and put together in Figure 5.2. (Recall that the HMC settings here are $\epsilon = 0.05$ and $\tau = 2$ for `gauss`, $\epsilon = 0.01$ and $\tau = 10$ for `beta`, and $\epsilon = 1$ and $\tau = 5$ for `lr`.)

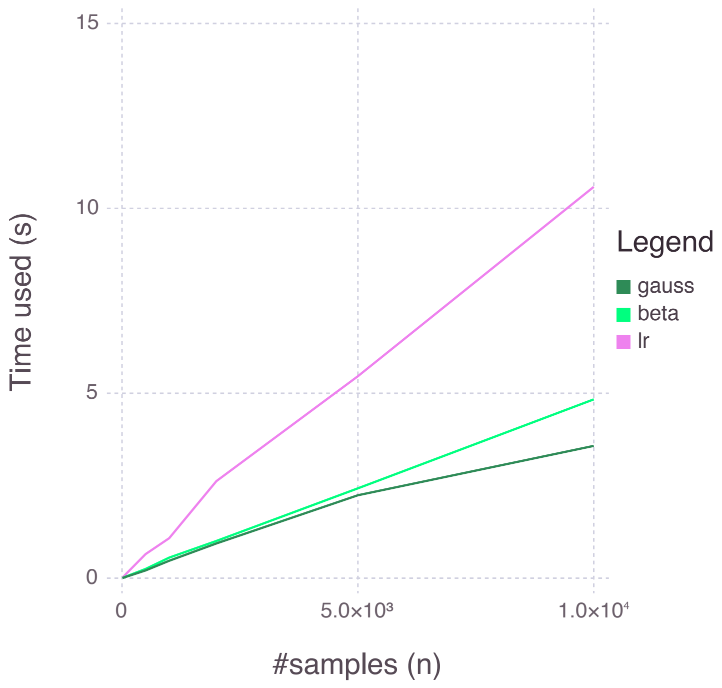


Figure 5.2: Time used the HMC sampler with number of samples varying.

As you can see, the sampling time has a (roughly) linear relation with the number of samples. This is satisfying because it shows that the performance of the HMC sampler is stable when the number of samples increases. In fact, this is the nature of the HMC algorithm as samples are consecutively generated depending on their predecessors.

Note that the nature of SMC and PG does not have such property. Turing improves their sampling time to be also proportional to the sample numbers by the use of coroutines, which allows multiple particles to be simulated in the same time.

Increasing the number of variables In order to evaluate how the sampler performs with the number of variables increasing, a set of toy models as below with the number of priors `M` varying from 1 to 9 are used. Note that the number of variables here is actually the number of priors and also the dimensionality.

Listing 5.1: Toy Model for the Experiment of Varying Number of Variables

```julia
xs = [1.5, 2.0]       # the observations
M = 1                 # number of means, varying from 1 to 9
@model gauss_var begin
  ms = Vector{Dual}(M) # initialise an array to store means
  for i = 1:M
    @assume ms[i] ~ Normal(0, sqrt(2))  # define the mean
  end
  for i = 1:length(xs)
    m = mean(ms)
    @observe xs[i] ~ Normal(m, sqrt(2)) # observe data points
  end
  @predict ms         # ask predictions of s and m
end
```

The experiment was conducted by learning these models by the HMC sampler for 10 times with $n = 250$, $\epsilon = 0.45$ and $\tau = 5$, and the average running times were recorded. A plot showing how the sampling time changes with the number of variables in given in Figure 5.3.

As you can see from the figure, the sampling performance of the HMC degenerates with the number of variables increasing, i.e. dimensionality increasing. This is not a nature of HMC but the limitation of the implementation because MH based methods do not suffer from high dimensionality [1]. This degeneration is believed to be caused by the use of forward mode of AD, which requires n times running of the probabilistic program for a model with n priors. This is not satisfying and possible approach to solve this problem will be proposed in Section 6.2.

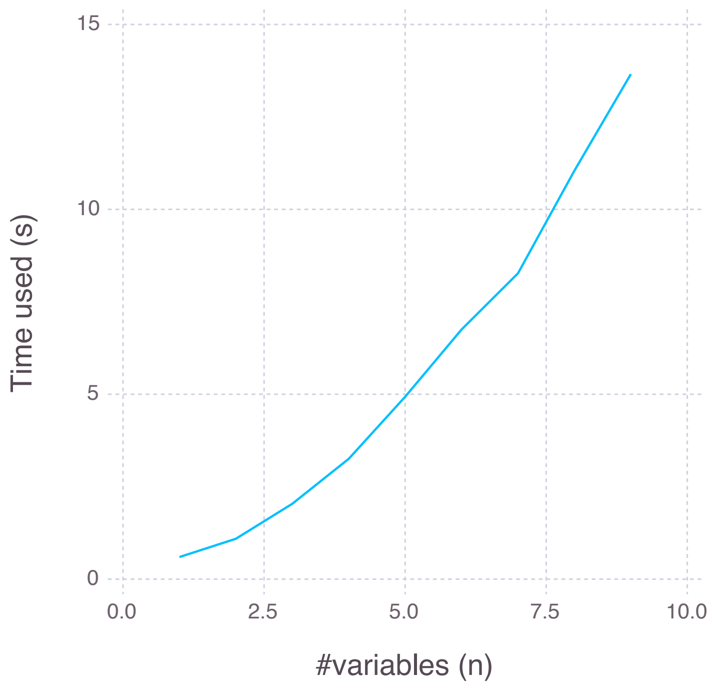


Figure 5.3: Time used the HMC sampler with number of variables varying.

5.4 More Experiments

In previous sections, some interesting experiments are proposed to be conducted. In specific, Section 2.3 mentions the comparison between a neural net trained by GD and Bayesian approach, and Section 5.2 proposes a further investigation into the difference influences of HMC parameters to different priors. The results of these two experiments are given in this section.

5.4.1 Bayesian Inference versus Gradient Descend

It is interesting to compare the predictions from a Bayesian approach and an optimisation approach. Figure 5.4 shows the predictions of the logistic regression model (i.e. a neural net with a single neuron) from Turing and a normal GD method.

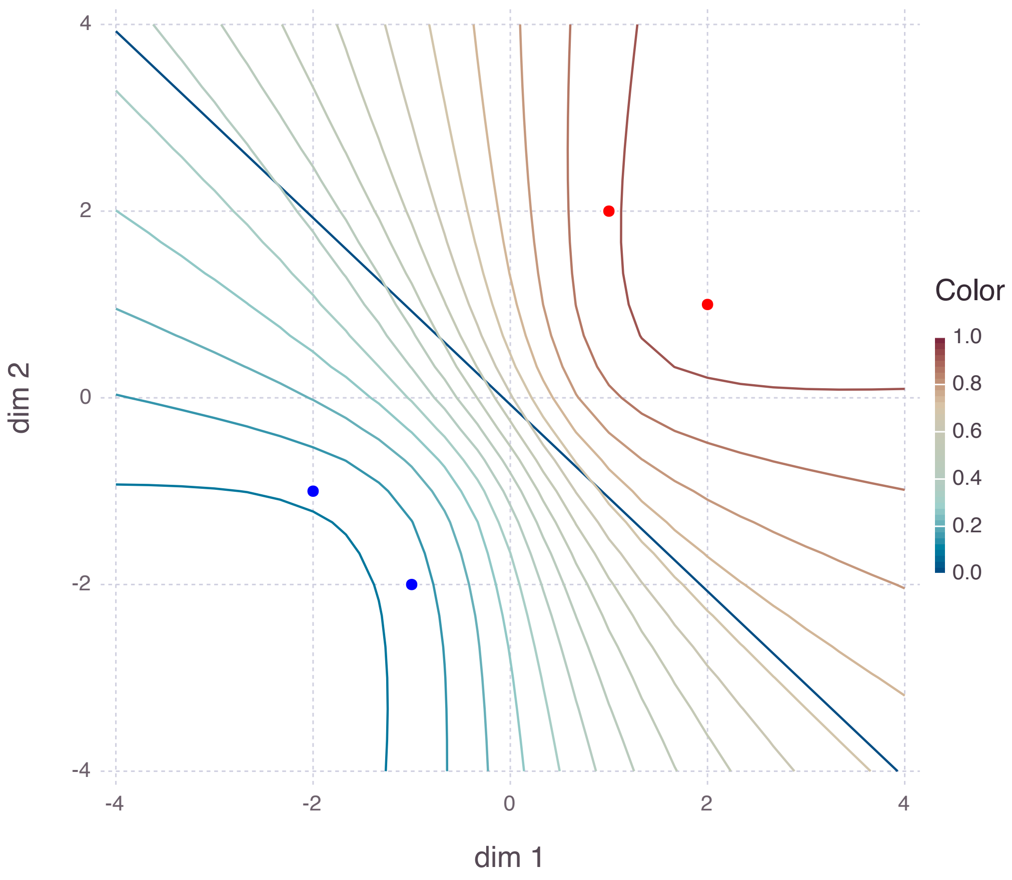


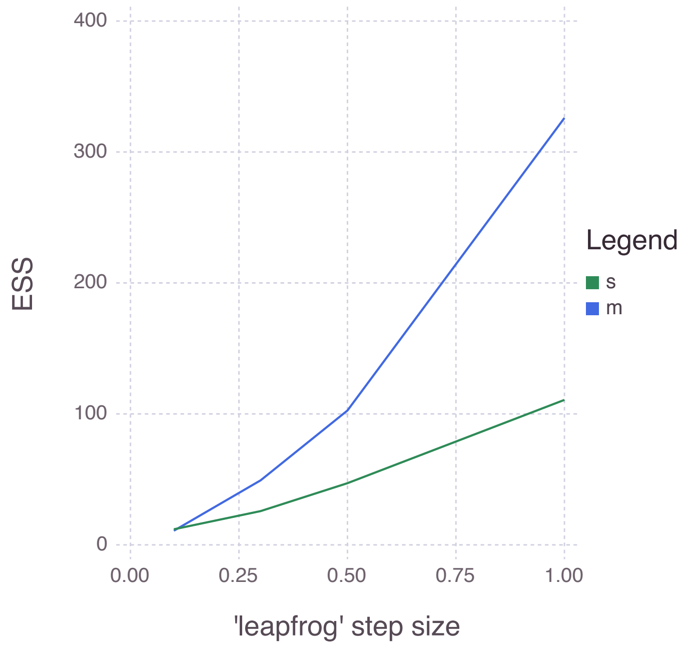
Figure 5.4: Predictions of the Single Neuron BNN from GD (decision boundary in blue) and Bayesian approach (coloured contour for probability).

In this figure, the training data are the two red and two blue points, which are labelled with 1s and 0s respectively. This prediction results illustrates the advantages of the Bayesian approach mentioned in Section 2.1, which is that the GD can only give hard prediction with its decision boundary while the Bayesian prediction is not only soft but also provides the certainty of its prediction.

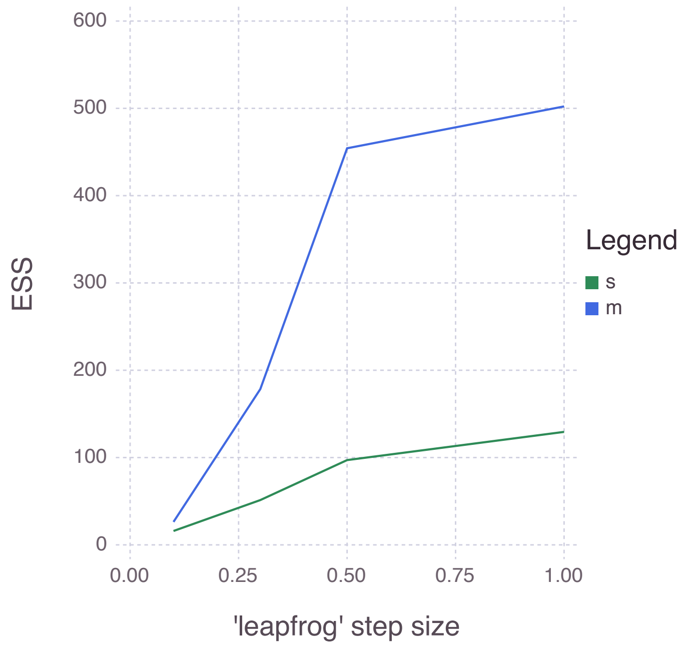
5.4.2 Sensitivity of Different Variables

In order to evaluate the sensitivity of different variables to the parameters of the HMC sampler, a set experiments were done on the Gaussian model using different settings of the HMC sampler. The corresponding result is shown in Figure 5.5.

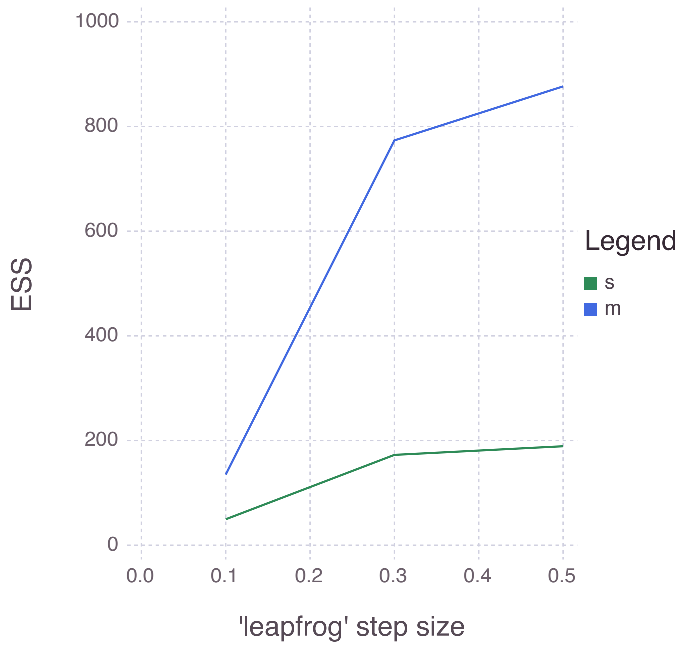
As you can see from this figure, the prior m is more sensible to the change of parameters than s. This may be due to the nature of the prior distributions they are drawn from. Also, the increase of s tends to stop early than that of m in Figure 5.5c, which indicates that there would be different optimal settings for different variables in the same model. Also, compared with HMC that has obvious gaps in ESS between s and m, the results of NUTS in Table 5.3 shows similar ESS for both s and m. Both of these two findings indicate that the NUTS algorithm has an advantage of automatically tuning the parameters in the sampling process to balance the HMC move in different dimensions.



(a) $\tau = 1$



(b) $\tau = 2$



(c) $\tau = 5$

Figure 5.5: Sensitivity of s and m in the Gaussian model for different parameter settings of the HMC sampler.

Chapter 6

Summary and Conclusions

In conclusion, this M.Phi project successfully contributes a workable HMC sampler to the Turing framework, with all the related design and implementation details given in this dissertation. Also, some evaluations were done on the sampler to prove its correctness and measure its performance and robustness.

The two main challenges of this project, computing the target density function and the corresponding gradient function for the HMC sampler, are successfully solved. The first one is done by building connections between the probabilistic program and the energy function required by HMC using Bayes' rule, and the second one is accomplished by applying the forward mode of AD through probabilistic programs respectively.

The final HMC sampler is workable and could be used in practise but there are till some limitations, which will be discussed in Section 6.1. Also some potential approaches to overcome these limitations will be given in Section 6.2.

6.1 Limitations

There are several limitations of the current HMC sampler, which can be divided in two categories. The first category is about the sampling efficiency. According to the evaluations given, this implementation of HMC has an acceptable sampling speed for most of the circumstances, which is similar to the PG sampler and the SMC sampler in Turing. However, this implementation has some potential flaws. One is that the performance of the HMC sampler degenerates with the number of parameters increasing because it uses the forward mode of AD. Another one is that, compared with the NUTS implementation in Stan, the HMC sampler shows worse sampling efficiency in cases where different settings are needed for different parameters. In addition, the sampling time used by the HMC sampler in Turing is still much higher than that of Stan in general.

The second category is about the functionality of the current HMC sampler. First of all, as the HMC algorithm can be only applied to continuous spaces, inference of models with both continuous and discrete parameters is not supported by the HMC sampler. Secondly, the parameters in the sampler are unbounded. This will cause a situation where out-of-bound errors occur, which is currently solved by rerunning the failed step. However, this solution is inefficient and would fail in some circumstances. Thirdly, the HMC sampler needs to be tuned for good performance, which is tedious and time-consuming.

6.2 Future Work

There are several future tasks to be done to improve the performance of the HMC sampler and the functionality of Turing, which are listed below.

Improving sampling speed

- Optimise the fundamental implementation by getting rid of some duplicated calculations and object copying.

  There are some duplicates of variables inside the HMC sampler. For example, priors need to be copied from the container to local in order to calculate the log density and the container of priors needs to be copied using `deepcopy()` in order to prevent some exceptions. However, these kinds of work could be potentially accomplished in a better way with care.

- Implement the reverse mode of AD.

  The reverse mode of AD could compute the derivative of all the variables in one through, which could improve the performance, especially for large models with many parameters. Also, there is a variant of forward mode of AD available, in which a dual number is extended to contain multiple dual parts. This implementation is expected to have better performance than the original forward mode but worse performance than the reverse mode, and is also worth to be implemented as a milestone.

- Vectorise the computation inside Turing.

  As mentioned in Section 5.2, Stan improves its fundamental running speed by vectorisation, which could also be done in Turing. Currently in Turing, especially for the HMC sampler, vectors and single-valued variables are processed separately. This improvement would involve designing a consistent interface to support vectorisation within Turing and with other involved packages. Also, the forward mode AD also needs be replaced for this reason as it does not support vectors directly.

Enhancing functionality

- Implement the bounded HMC sampler.

  As the current implementation of HMC deals with out-of-bound problems by simply re-run the corresponding HMC step, there are huge time wasted when there are too many re-runs. This could be completely avoided by a bounded HMC version. The implementation of this would also involve re-designing the compiler to support definition of the bound of variable in the program, or automatically extracting and setting domain restrictions from the `Distribution` package somehow.

- Implement the NUTS algorithm.

  As mentioned before, NUTS can free users to tune the parameters of the HMC sampler without obvious loss of efficiency (and even has better performance in some models with branches). This variant of HMC is expected to be ultimately supported by Turing in the future.

- Implement a Gibbs sampler combing HMC and PG.

  As HMC can only sample continuous variables (but with better performance compared with PG), it is possible to build a sampler with better sampling efficiency than pure PG sampler by combining HMC and PG into a Gibbs sampler. In such Gibbs sampler, discrete variables are ought to be sampled by PG and continuous variables are ought to be sampled by HMC. This was already mentioned in Section 4.1.2 where supporting additional arguments annotated on variables is the preparatory work for this Gibbs sampler.

- Make the `Distribution` package compatible with `Dual` type.

  Currently there are only a limited number (10) of differential distributions (`dDistribution` type) in Turing. Two approaches could be done to solve this issues. One is to continuously manually implement more distributions in Turing and the ideal coverage is about 80% of the `Distributions` package. However, this approach is not only tedious but less valuable in the sense that other developer cannot use our `dDistribution` easily. Therefore another approach may be considered, which is to apply a patch to the `Distributions` package to make it support variables in `Dual` type. This may face some difficulties but is worth to be done if there is more time because it can solve the problem radically. Besides, it is also feasible to currently separate the distribution references used by HMC and other samplers to make other samplers works on the `Distributions` package independently.

- Implement the replay of priors in more universal way.

  In Section 4.2.3, a universal way to replay priors is proposed, which is worth to be implemented to make Turing support priors defined by any possible Julia data structure or type.

Bibliography

[1] David JC MacKay. *Information theory, inference and learning algorithms.* Cambridge university press, 2003.

[2] Zoubin Ghahramani. Probabilistic machine learning and artificial intelligence. 521(7553):452–459, 2015.

[3] Nadir Murru and Rosaria Rossini. A bayesian approach for initialization of weights in backpropagation neural net with application to character recognition. *Neurocomputing*, 193:92–105, 2016.

[4] Arnaud Doucet, Nando De Freitas, and NJ Gordon. An introduction to sequential monte carlo method. *SMC in Practice*, 2001.

[5] Arnaud Doucet. Introduction to sequential monte carlo methods. 2008.

[6] Christophe Andrieu, Arnaud Doucet, and Roman Holenstein. Particle markov chain monte carlo methods. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 72(3):269–342, 2010.

[7] Darren Wilkinson. Introduction to the particle gibbs sampler. `https://darrenjw.wordpress.com/2014/01/25/introduction-to-the-particle-gibbs-sampler/`, 2014. [Online; accessed 10-July-2016].

[8] Fredrik Lindsten, Michael I Jordan, and Thomas B Schön. Particle gibbs with ancestor sampling. *Journal of Machine Learning Research*, 15(1):2145–2184, 2014.

[9] Radford M Neal et al. Mcmc using hamiltonian dynamics. *Handbook of Markov Chain Monte Carlo*, 2:113–162, 2011.

[10] Robert E Kass, Bradley P Carlin, Andrew Gelman, and Radford M Neal. Markov chain monte carlo in practice: a roundtable discussion. *The American Statistician*, 52(2):93–100, 1998.

[11] Dr. Orlaith Burke. *Statistical Methods Autocorrelation: MCMC Output Analysis.* Department of Statistics, University of Oxford, 2012.

[12] James M Flegal, Murali Haran, and Galin L Jones. Markov chain monte carlo: Can we trust the third significant figure? *Statistical Science*, pages 250–260, 2008.

[13] Andrew Gelman, John B Carlin, Hal S Stern, and Donald B Rubin. Bayesian data analysis. texts in statistical science series, 2004.

[14] Razvan Ranca. Improving inference performance in probabilistic programming languages. 2014.

[15] 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.

[16] Ge, Hong, Adam Scibior, and Zoubin Ghahramani. Turing: rejuvenating probabilistic programming in julia. 2016.

[17] Dexter Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22(3):328 – 350, 1981.

[18] Kevin P Murphy. *Machine learning: a probabilistic perspective.* MIT press, 2012.

[19] Kevin P. Murphy. Conjugate bayesian analysis of the gaussian distribution. 2007.

[20] David A Freedman. *Statistical models: theory and practice.* cambridge university press, 2009.

[21] David R Cox. The regression analysis of binary sequences. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 215–242, 1958.

[22] David J Lunn, Andrew Thomas, Nicky Best, and David Spiegelhalter. Winbugs-a bayesian modelling framework: concepts, structure, and extensibility. *Statistics and computing*, 10(4):325–337, 2000.

[23] Tom Minka, John Winn, John Guiver, and David Knowles. Infer .net 2.4, 2010. microsoft research cambridge.

[24] Bob Carpenter, Andrew Gelman, Matt Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Michael A Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: A probabilistic programming language. *J Stat Softw*, 2016.

[25] Frank Wood, Jan Willem van de Meent, and Vikash Mansinghka. A new approach to probabilistic programming inference. In *Proceedings of the 17th International conference on Artificial Intelligence and Statistics*, pages 1024–1032, 2014.

[26] Andrew Gelman, Daniel Lee, and Jiqiang Guo. Stan a probabilistic programming language for bayesian inference and optimization. *Journal of Educational and Behavioral Statistics*, page 1076998615606113, 2015.

[27] Martyn Plummer et al. Jags: A program for analysis of bayesian graphical models using gibbs sampling. In *Proceedings of the 3rd international workshop on distributed statistical computing*, volume 124, page 125. Vienna, 2003.

[28] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *arXiv preprint arXiv:1411.1607*, 2014.

[29] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. *CoRR*, abs/1209.5145, 2012.

[30] Julia Community. Julia documentation. 2016.

[31] Atilim Baydin, Barak A Pearlmutter, and Alexey Radul. Automatic differentiation in machine learning: a survey. 2015.

[32] Bob Carpenter, Matthew D Hoffman, Marcus Brubaker, Daniel Lee, Peter Li, and Michael Betancourt. The stan math library: Reverse-mode automatic differentiation in c++. *arXiv preprint arXiv:1509.07164*, 2015.

[33] Matthew D Hoffman and Andrew Gelman. The no-u-turn sampler: adaptively setting path lengths in hamiltonian monte carlo. *Journal of Machine Learning Research*, 15(1):1593–1623, 2014.