

Improving inference performance in probabilistic programming languages

Razvan Ranca
Queens' College



UNIVERSITY OF
CAMBRIDGE

*A dissertation submitted to the University of Cambridge in partial
fulfilment of the requirements for the degree of Master of
Philosophy in Advanced Computer Science (Option B)*

University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
UNITED KINGDOM

Email: ranca.razvan@gmail.com

June 13, 2014

Declaration

I Razvan Ranca of Queens' College, being a candidate for the M.Phil in Advanced Computer Science, 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: 13,267

Signed:

Date:

This dissertation is copyright ©2014 Razvan Ranca.
All trademarks used in this dissertation are hereby acknowledged.

Abstract

In this thesis we look at the challenges involved in constructing efficient inference engines for probabilistic programming languages (PPL). Improving inference performance is one of the main obstacles that must be surpassed before PPLs can be widely adopted.

We start by empirically comparing the performance of two PPLs from different programming paradigms and with different design objectives. Based on these experiments we identify some areas of possible improvement. The first area is concerned with automatically re-writing probabilistic programs by decomposing their prior distributions, such that the inference result is left invariant but inference speed improves. The second aspect we look at is the implementation of a novel inference engine based on slice sampling.

The main novel contributions of this thesis are the prior distribution decomposition method which aims to improve inference performance under Metropolis-Hastings, and the usage of slice sampling towards the implementation of a new PPL, called “Stochastic Python”. We make Stochastic Python, which also supports Metropolis-Hastings inference, freely available online.

Contents

1	Introduction	1
1.1	Importance of Probabilistic Programming Languages	1
1.2	Need for better inference	2
1.3	Approach and Roadmap	2
2	Comparing Venture and OpenBUGS	5
2.1	Motivation	5
2.2	Preliminaries	6
2.2.1	OpenBUGS	6
2.2.2	Venture	6
2.2.3	Number of MCMC steps	7
2.3	Empirical results	8
2.3.1	Tdf model description	8
2.3.2	Tdf model true posteriors	8
2.3.3	Tdf model results	9
2.4	Analysis of Venture's performance	11
2.4.1	Webchurch's performance	12
3	Partitioned Priors	15
3.1	Preliminaries	15
3.1.1	Metropolis-Hastings	16
3.1.2	Expected number of iterations to a neighbourhood of the mode	17
3.1.3	Mixing properties around the mode	18
3.2	Sum of uniforms	18
3.2.1	Finding a good sum decomposition	19
3.2.2	Evaluating the (1,2,95) sum decomposition	21
3.3	Bit decomposition	24
3.3.1	Definition	25
3.3.2	Evaluation on Tdf and Tdf21	25
3.3.3	Getting stuck on a bad sample	27
3.3.4	Mixtures of shifted bit decompositions	29
4	Novel PPL inference techniques	35
4.1	Preliminaries	35
4.1.1	Slice sampling	35
4.1.2	Basic PPL Construction	37
4.2	Stochastic Python	37

4.3	Slice sampling inference engine	38
4.3.1	Custom Slice Sampling and Metropolis on Tdf models . .	38
4.3.2	Generic, lightweight, slice sampling inference engine . .	41
4.4	Quasi-Monte Carlo	51
5	Related Work	53
5.1	Probabilistic Programming Languages	53
5.2	Speeding up inference	54
6	Summary and Conclusions	57
6.1	Overview	57
6.2	Future Directions	58

Chapter 1

Introduction

Probabilistic programming languages (PPLs) have garnered a lot of attention recently, especially since the announcement of a DARPA¹ initiative to support their research and development [3]. However, a lot of progress has to be made before the promise of PPLs can be reached, with one of the critical areas of current research being the inference engines used to convert programs in these languages into statistical inference algorithms. This project aims to better understand the complex interactions and performance trade-offs inherent between probabilistic programming languages, model types and inference methods. By better understanding these interactions we hope to discover areas where improvements are needed and explore ways in which such improvements could be carried out.

1.1 Importance of Probabilistic Programming Languages

Machine learning has become ubiquitous, with applications ranging from self-driving cars to gene sequencing. However, building any significant machine learning application currently involves a great deal of expertise in both defining an adequate statistical model and implementing inference algorithms for this model in order to extract useful information from your data. These challenges create a significant bottleneck towards wide-range adoption of machine learning solutions.

¹Defense Advanced Research Projects Agency

Probabilistic programming attempts to alleviate this problem by letting the user describe their model and statistical queries in a high level programming language. Such a language would provide convenient methods of describing complex probability distributions, and have an inference engine capable of automatically generating the necessary statistical inference code. This higher-level approach to modelling aims to duplicate some of the benefits gained by the switch from assembly to higher-level programming languages, namely allowing a wider range of developers to work on more complex problems with less effort and at a lower cost. If successful, these techniques could not only make probabilistic modelling cheap and simple enough to become ubiquitous, but also enable the construction of applications that are inconceivable today.

1.2 Need for better inference

A lot of current research in probabilistic programming is focused on achieving more efficient automatic inference on different types of models (eg: [9, 43, 45]). This problem has been approached from many angles, ranging from the development of specialized inference methods that work well on certain, restricted, classes of models (eg: [42, 47]), to employing general inference techniques on models transformed by the application of optimization techniques similar to those used in compilers (eg: [22, 46]).

If we manage to surpass the hurdle of constructing generic inference engines which are efficient on a wide range of models and model representations, then many nascent modeling areas could accelerate their progress significantly. One early example of the types of applications these techniques could make possible is the modeling of natural language understanding by the mutually recursive simulation of a listener reasoning about a speaker who is in turn reasoning about his listener [11, 5].

1.3 Approach and Roadmap

In Chapter 2 we perform an empirical comparison of the inference performance of Venture and OpenBUGS, which are two PPLs which make very different efficiency/expresiveness compromises. By testing different models on these engines we can gain a better idea of where the current systems are in most need of improvements, and how such improvements might be carried out.

In Chapter 3 we pursue a first possible area of improvement, by considering the

feasibility of a pre-processing step which compiles a probabilistic program into an equivalent, but more efficient, form. As far as we are aware the idea for this pre-processing step is novel. The empirical results suggest that significant speed-ups are possible, but that this performance is dependant on the model being compiled.

In 4 we explore a second potential inference engine improvement, this time by implementing a slice sampling inference engine. In order to perform this investigation we create a novel probabilistic programming language, called Stochastic Python. Slice sampling has not been previously used as part of a Turing Complete PPL, and in our empirical comparison shows promising results on several models when compared to the default Metropolis-Hastings choice. Stochastic Python is made available online [35].

Chapter 5 first does a very brief review of the types of PPLs currently in existence, so that our empirical comparison of Venture and OpenBUGS can be placed in context. We then look at work related to speeding up inference in PPLs, and specifically related to rewriting probabilistic programs and to implementing engines based on inference techniques other than Metropolis-Hastings.

Finally, Chapter 6 discusses the implications of the analysis done in earlier chapters and presents some possible avenues of future work.

Chapter 2

Comparing Venture and OpenBUGS

In this chapter we perform an empirical comparison of the Venture and OpenBUGS probabilistic programming languages on several models, in order to gain a better understanding of these systems' strengths and limitations.

2.1 Motivation

As discussed in Section 5.1, there are currently a plethora of PPLs which differ in significant ways, including in expressivity, efficiency, programming paradigm and intended usage. At the moment, there are very few surveys comparing the relative benefits and drawbacks of these languages on different classes of models and so their relative performance is not very well understood.

In this chapter we attempt to take a step towards better understanding the characteristics of these languages by looking at two PPLs which fall at different ends of the efficiency/flexibility spectrum. We do this by implementing a few different models and evaluating the performance of the two different PPL's inference engines on these models. The insight thus gained will give us an idea of where the current systems most need improving and thus guide future work towards improving inference performance in these languages.

2.2 Preliminaries

2.2.1 OpenBUGS

OpenBUGS, the latest version of the BUGS project, defines a bespoke programming language that can be used to specify and perform bayesian inference on probabilistic models. The models that can be specified in OpenBUGS are finite directed acyclic graphical models, which have well-understood mathematical properties. These properties can ease the task of performing inference, and OpenBUGS aims to take advantage of them by employing an “expert system” which picks Markov Chain Monte Carlo inference methods that are efficient on the user’s particular model [31].

By restricting their language’s expressivity to that of finite graphical models, OpenBUGS sacrifices flexibility for efficiency. One essential restriction on OpenBUGS models is that they cannot use stochastic choices that will influence control flow in the program [19]. Allowing such randomness would result in trans-dimensional program traces of varying and, through recursion, potentially unbounded size, which would no longer allow OpenBUGS to compile the program into a graphical model. Other potentially useful constructs that are not allowed under the graphical model paradigm include random compound data types and any procedures that could lead to a priori unbounded executions.

However, many probabilistic models can be expressed as finite, directed, acyclic graphs, and on such models OpenBUGS’s efficiency and ease of use makes it one of the most popular PPLs.

2.2.2 Venture

Venture aims to give users a greater flexibility in defining models than OpenBUGS does, while also being sufficiently efficient and extensible for use on a wide variety of probabilistic models. Venture is an interactive virtual machine in which probabilistic programs are specified via a Turing-complete functional language build on top of Lisp. Venture also allows the user to build custom inference strategies, specified in a compositional manner from simpler inference building blocks, or even from optimized external inference code [22].

Venture’s flexibility means that it supports stochastic choices affecting control flow, as well as any type of higher-order probabilistic procedure. This means Venture can represent models outside of the scope of OpenBUGS, such as non-parametric Bayesian models [39], and cognition models involving aspects such

as reasoning about reasoning, and theory of mind (eg: [7, 6]). Additionally, Venture supports a large number of probabilistic primitives, including “likelihood-free” primitives. Finally, Venture’s higher-order nature allows it to specify the learning of Venture programs from data (including the program structure and parameters) as an inference problem in a Venture program [23].

When compared to other Turing complete, flexible, higher-order PPLs, such as Church, Venture proposes a number of changes that should make it significantly more efficient. The question of how Venture measures up against more restricted PPLs, such as OpenBUGS, is still not settled though.

2.2.3 Number of MCMC steps

Venture and OpenBUGS have a different interpretation of what an MCMC step is. This difference must be taken into account so that the empirical results of the two PPLs are comparable.

Specifically, OpenBUGS updates all currently unconditioned variables during one step, whereas Venture only updates one, randomly chosen, variable. In order to correct for this, Venture will need to perform roughly (no. of OpenBUGS steps) * (no. of unconditioned variables) steps. Ideally, if we want both the amount of work and the number of samples generated by each PPL to be comparable, then we can specify the work that must be done by each PPL as:

	OpenBUGS	Venture
Burned samples	B	$V * B$
Extracted samples	S	S
Inter-sample lag	L	$V * L$
Total MCMC steps	$B + L * S$	$V * (B + L * S)$

Table 2.1: Strategies for extracting S samples from a model with V unconditioned variables

However, in some cases it can make sense to not follow the above specification. For instance, when the performance gap between the two PPLs is very large, we may prefer not to generate very few samples with the faster PPL just so that the slower one can terminate the same amount of work in a reasonable timeframe.

2.3 Empirical results

In this section, we consider a few simple models taken from the OpenBUGS model repository, and test inference performance on them for both OpenBUGS and Venture.

2.3.1 Tdf model description

The Tdf models attempt to infer a student t distribution's degrees of freedom, by considering 1,000 samples drawn from said Student t distribution. The Tdf models come in 3 variations, which just change the prior distribution of the degrees of freedom parameter (called d). In the coarse discrete version, d is drawn from the uniform discrete distribution ranging from 2 to 50. In the fine discrete distribution, d is drawn uniformly from the set: $\{2.0, 2.1, 2.2, \dots, 6.0\}$. Finally, in the continuous version, d is drawn from the continuous uniform distribution between 2 and 100. The specification for the continuous version of the model is given below. The models' OpenBUGS implementations and inference results can be found in the OpenBUGS model repository [36].

$$\begin{aligned} \nu &\sim (\text{uniform-continuous } 2 \ 100) \\ \text{observe (student-t } \nu) &= x; \forall x \in X \quad (\text{X being the data we're conditioning on}) \end{aligned}$$

For all 3 Tdf models, OpenBUGS uses 1,000 steps for burn-in and then extracts 10,000 consecutive samples (i.e. using a lag of 1). The Tdf model has only 1 unconditioned variable and so, as explained in Table 2.1, this would correspond to $1 * (1000 + 1 * 10000) = 11,000$ MCMC steps in Venture. However, as seen below, the different inference engines employed means that Venture needs more samples than OpenBUGS to derive a reasonable posterior estimate on this model. For this reason, we used a burn-in of 1000 and then extracted an additional 1000 samples using a lag of 100. The total number of steps performed by Venture is therefore $1000 + 1000 * 100 = 101,000$.

2.3.2 Tdf model true posteriors

Given the simplicity of the Tdf models, we can calculate the true posteriors analytically. If X represents a vector of the observed datapoints (x), and ν

represents the student t distribution's degrees of freedom, then:

$$P(\nu|X) \propto P(X|\nu)P(\nu)$$

$$P(X|\nu) = \prod_{x \in X} P(x|\nu) = \prod_{x \in X} \frac{\Gamma(\frac{\nu+1}{2})}{\Gamma(\frac{\nu}{2}) \sqrt{\nu\pi}} \left(1 + \frac{x^2}{\nu}\right)^{-\frac{\nu+1}{2}}$$

Further, in all of the Tdf models, the prior on ν is uniform, therefore the posterior will be directly proportional to the likelihood. The only role of the prior is to determine the normalization constant by defining what the domain of our posterior is. Specifically, we have:

$$P(\nu|X) \propto \prod_{x \in X} \frac{\Gamma(\frac{\nu+1}{2})}{\Gamma(\frac{\nu}{2}) \sqrt{\nu\pi}} \left(1 + \frac{x^2}{\nu}\right)^{-\frac{\nu+1}{2}} \begin{cases} \nu \in [2, 100] \text{ for Tdf continuous} \\ \nu \in \{2, 3, \dots, 50\} \text{ for Tdf course discrete} \\ \nu \in \{2.0, 2.1, \dots, 6.0\} \text{ for Tdf fine discrete} \end{cases}$$

Using this formulation we calculate the true posteriors for all 3 Tdf model as shown in Figure 2.1.

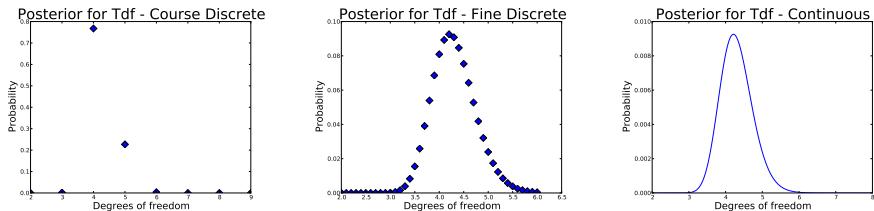


Figure 2.1: True posterior distributions for the 3 Tdf models.

2.3.3 Tdf model results

On the course discrete model (Figure 2.2a) we see that the two PPLs obtain similar results but with a large gap in runtime. Venture runs ~ 54 time more slowly than OpenBUGS. To get a better idea of the relative performance of the languages we can look at their speed of convergence to the true posterior.

As seen in Figure 2.2b, due to the course discrete prior, the convergence rate here is quite choppy and no significant conclusions can be drawn from this single run.

Looking at the performance results on the fine discrete prior (see Figure 2.3a) shows that, as with the course discrete case, the two engines obtain similar look-

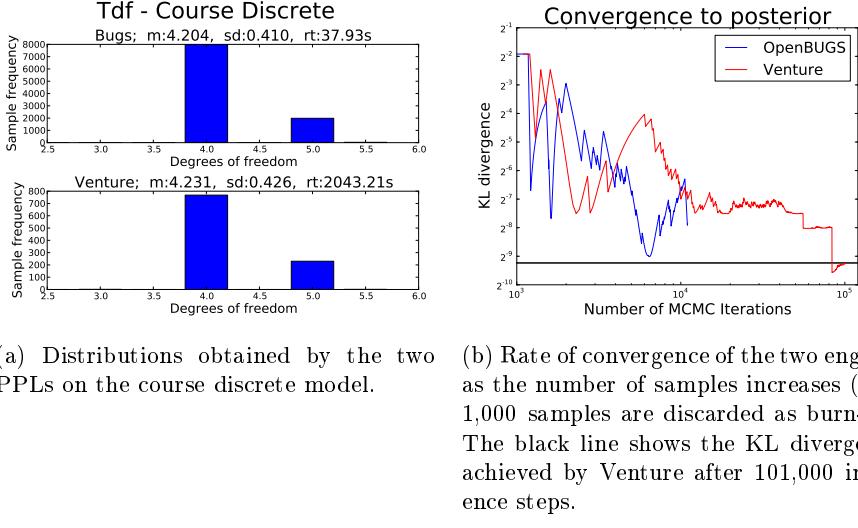


Figure 2.2: Performance of the Tdf course discrete model

ing distributions. Additionally, the runtime gap actually worsens here, Venture now having a runtime ~ 69 times larger than OpenBUGS.

The convergence rate shown in Figure 2.3b reveals that Venture also does a worse job of inferring the true posterior, despite the longer runtime. The finer prior used here results in a much smoother convergence rate and so we can see that the KL divergence reached by Venture after 101,000 samples is achieved by OpenBUGS after only 3,000 (including the burn-in). Performing 3,000 MCMC steps in OpenBUGS takes 8.7 seconds, while Venture's run took 1910. So we may say that, reported to convergence to true posterior, Venture is $1910/8.7 = \sim 220$ times slower than OpenBUGS.

On the continuous case (see Figure 2.4a), not only does the runtime gap persist (Venture is again ~ 68 times slower than OpenBUGS), but the distribution generated by Venture is also visibly noisier. One explanation for the noise could be the fact that, even though Venture is performing more MCMC iterations, due to the lag of 100 between extracted samples, it is actually generating only 1,000 samples compared to OpenBUGS' 10,000.

The convergence rates presented in Figure 2.4b show that, as with the fine discrete model, there seems to be a large qualitative gap between the performance of the two engines. Thus the KS difference achieved by Venture after 101,000 MCMC iterations is reached by OpenBUGS after only 5,000. Additionally, while Venture takes 2105 seconds to reach this performance level, OpenBUGS does it in 16.5. Venture is therefore $2105/16.5 = \sim 127$ times slower than OpenBUGS on this model.

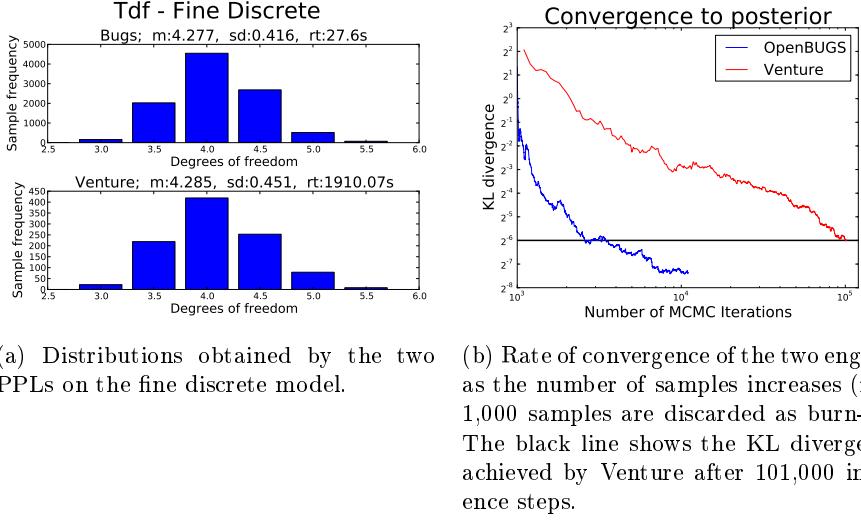


Figure 2.3: Performance of the Tdf fine discrete model

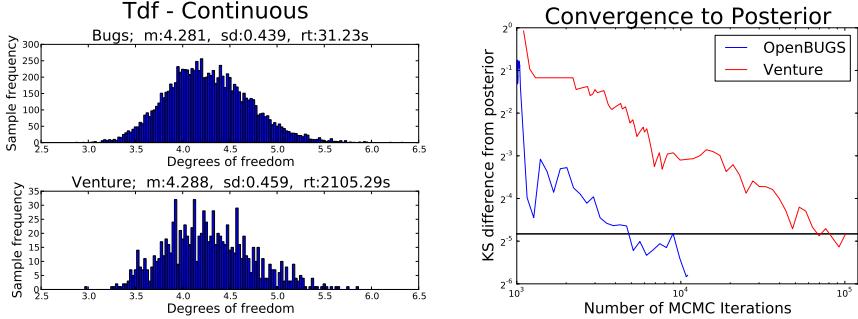
Based on these results we can say that Venture performs significantly worse than OpenBUGS on the Tdf model variants (with the last two variants exhibiting a slow-down of more than 2 orders of magnitude).

2.4 Analysis of Venture's performance

In order to understand why Venture seems to perform so badly on the Tdf models, we look at the distribution of runs of identical samples generated by the two engines.

Figure 2.5 shows that Venture has a much higher propensity for identical sample runs than OpenBUGS. Keeping in mind that Venture takes a lag of 100 samples between each 2 extracted samples, we see that Venture exhibits several runs of over 500 MCMC iterations where the observed variable does not change at all. On the other hand, OpenBUGS has no two identical consecutive samples.

This difference can be explained by the different inference strategies employed by the two engines. Venture makes use of single-site Metropolis whereas OpenBUGS uses a slice sampler on one dimensional models such as Tdf [20, 15]. We take a closer look at these different inference techniques and their performance in Chapter 4.



(a) Distributions obtained by the two PPLs on the continuous model.

(b) Rate of convergence of the two engines as the number of samples increases (first 1,000 samples are discarded as burn-in). The black line shows the KS difference achieved by Venture after 101,000 inference steps.

Figure 2.4: Performance of the Tdf continuous model

2.4.1 Webchurch’s performance

One alternative to Venture which we considered is WebChurch. WebChurch is a compiler which translated the Church PPL into JavaScript, and thus allows for in-browser execution [8].

We implemented a few very simple models (similar to the ones discussed below) in WebChurch, however the execution tended to hang even when conditioning on very few variables (less than 5) and when extracting very few samples (less than 100). Asking one of the WebChurch creators confirmed that the publicly available implementation is meant for didactic purposes and not designed to scale well. We therefore chose to focus only on Venture and OpenBUGS.

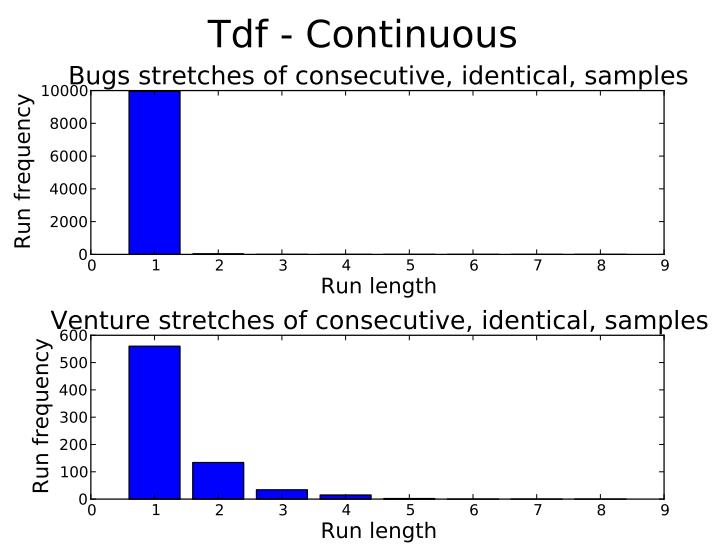


Figure 2.5: Number and size of identical sample runs generated by the two PPLs on the continuous model.

Chapter 3

Partitioned Priors

In this chapter we look at a novel approach towards improving the inference performance of basic Metropolis-Hastings by partitioning the model's prior distribution into several component parts. The idea behind this approach is that we may be able to guide the resampling process as to improve the rate of convergence to the mode and the mixing properties of local Metropolis-Hastings. If this approach proves successful, it would enable us to design a light pre-inference compiler of a probabilistic program which splits up its priors so that the subsequent inference step is performed more efficiently.

In the first instance, we focus on the splitting of uniform continuous priors. Not only are such priors commonly used in models, but sampling from any distribution ultimately relies on uniformly drawn random bits. Speeding up inference on uniform priors may therefore be a step towards speeding up inference on other prior distributions, assuming certain desirable properties, such that small changes in the uniform bits correspond to small changes in the final distributions.

For simplicity, we focus on models with a unimodal posterior distribution. The analysis for multimodal distributions would follow similar lines to the one presented here, but would have to additionally account for mode-switching.

3.1 Preliminaries

In order to explore the partitioned prior idea we need to have an understanding of the basic Metropolis-Hastings algorithm. We also need a way to evaluate the performance of a certain partition, which we propose to do by looking separately

at the time it takes for our Markov Chain to reach the true posterior's mode and at the chain's mixing properties around this mode. Finally, we can also consider the empirical distributions created by a fixed number of samples, which can be compared to a ground-truth distribution which is computer analytically. We perform this comparison by taking either the Kullback–Leibler divergence [18] (in the case of discrete distributions) or the Kolmogorov–Smirnov distance [24] (for continuous ones) between the correct posterior and the empirical distribution under consideration.

3.1.1 Metropolis-Hastings

The Metropolis-Hastings algorithm is a method of drawing samples from a probability distribution $P(x)$ when we can only compute values of the proportional function $P^*(x) = P(x) * Z$ [26, 13]. This method is useful when we want to estimate distributions (or expectations under distributions) that we cannot directly evaluate. This is a common problem in Bayesian modeling, since the probability distribution's normalization factor is often difficult to estimate.

Metropolis-Hastings is a Markov Chain Monte Carlo algorithm, which is to say that the algorithm will generate samples according to the Markov, memoryless, property where the choice of current sample depends only on the previous sample. Additionally, the markov chain implied by this sequence of samples will have $P(x)$ as its stationary distribution. This implies that, if we run Metropolis-Hastings long enough for the underlying markov chain to be sampling from it's stationary distribution, then we will be drawing samples from $P(x)$.

In order to create an underlying markov chain that has $P(x)$ as its stationary distribution it is useful to split the state transition into two components, a proposal function and an acceptance probability. Since we must respect the markov, memoryless, property, the proposal function can only take the current sample as its input and is therefore of the form $Q(x_{n+1}|x_n)$. It can then be shown [29] that in order for the markov chain to converge to the right distribution, it is sufficient to define the acceptance ratio as:

$$\alpha = \frac{P(x_{n+1})}{P(x_n)} \frac{Q(x_n|x_{n+1})}{Q(x_{n+1}|x_n)} = \frac{P^*(x_{n+1})}{P^*(x_n)} \frac{Q(x_n|x_{n+1})}{Q(x_{n+1}|x_n)}$$

For the purpose of inference in probabilistic programming languages, a simplification to the Metropolis-Hastings algorithm is usually made by assuming that the proposal kernel is symmetric: $Q(x_{n+1}|x_n) = Q(x_n|x_{n+1}) \forall x_{n+1}, x_n$, which leads to removing the 2nd term of the acceptance ratio. The full algorithm can

then be described as:

- 1 Pick a first sample x_0
- 2 Pick a proposal kernel $Q(x_{t+1}|x_t)$
- 3 Pick a desired number of samples to extract N
- 4 For $t=0; t < N; t++$
 - 4.1 Generate proposal $prop \sim Q(x_{t+1}|x_t)$
 - 4.2 Calculate acceptance ratio $\alpha = \frac{P^*(prop)}{P^*(x_t)}$
 - 4.3 Sample a uniformly random number $r \sim (\text{uniform-continuous } 0 \ 1)$
 - 4.4 Decide whether to accept the proposal $\begin{cases} x_{t+1} = prop, & \text{if } r < \alpha \\ x_{t+1} = x_t, & \text{otherwise} \end{cases}$

Intuitively we can see that the acceptance ratio checks how likely the proposal is in comparison to the current sample. If the proposal is more likely it is always accepted. Otherwise it is accepted with a probability that decreases the more unlikely the proposal is. In this way, the markov chain will have a chance to explore less likely probability regions while preferring to stay in the high-density areas.

For a more thorough discussion of Metropolis-Hastings, including proofs, please see [21, 29].

3.1.2 Expected number of iterations to a neighbourhood of the mode

The first test of the efficiency of a partition is, on average, how many iterations the algorithm will have to go through before the markov chain reaches a state close to the mode of the posterior. Since we are interested in seeing our markov chain mix around the posterior's mode, we want it to get close to the mode as soon as possible. The average number of iterations to the mode can also be viewed as a way to estimate a lower bound on the burn-in we should set for our algorithm.

When using local Metropolis-Hastings with an unpartitioned uniform prior, it is easy to analytically calculate the expected number of iterations to a mode's neighbourhood. Using the unpartitioned prior ($\text{uniform-continuous } a \ b$), we

end up sampling from the uniform distribution and accepting or rejecting those samples according to the Metropolis-Hastings acceptance ratio. In order to reach some neighbourhood of the mode $[mode - \epsilon, mode + \epsilon]$, we need to actually sample a number in that range from the prior (sample which will definitely be accepted since it will have higher log likelihood than anything outside that range). This means the number of samples it will take to get close to the mode with an unpartitioned prior will follow a geometric distribution with $p = (2\epsilon)/(b - a)$. The expected number of samples it takes to reach the neighbourhood will then be $(b - a)/(2\epsilon)$.

For partitioned priors it will usually not be possible to analytically determine the expected number of steps to $[mode - \epsilon, mode + \epsilon]$, so we shall instead perform empirical tests.

3.1.3 Mixing properties around the mode

Once the markov chain has reached the mode we wish to see how well it manages to mix around it. Here we look at different metrics that might give us an idea of the mixing properties. Visually inspecting the sample evolution will show if the inference tends to get stuck on certain values for long stretches. A numerical estimate of this can be obtained by measuring the “distance” traveled by the markov chain around the mode (i.e. the sum of absolute differences between consecutive samples). We can also inspect the sample autocorrelation, based on the idea that good mixing should imply a small autocorrelation within a sample run.

3.2 Sum of uniforms

We first consider partitioning the uniform prior into a sum of uniforms. This choice is made both for simplicity and so that we can observe some basic properties concerning local Metropolis-Hastings’s performance on partitioned priors.

Such a partitioning, however, should not be used in actual probabilistic program compilation techniques since it does not leave the prior invariant (that is to say, a sum of uniform variables is not a uniform variable, see Figure 3.1). The invariance is due to the uniform distribution not being infinitely divisible. The approach presented in this section could be safely used on other distributions, such as Gammas and Gaussians, which are infinitely divisible. In Section 3.3 we will present a partitioning technique which does leave the uniform prior unchanged.

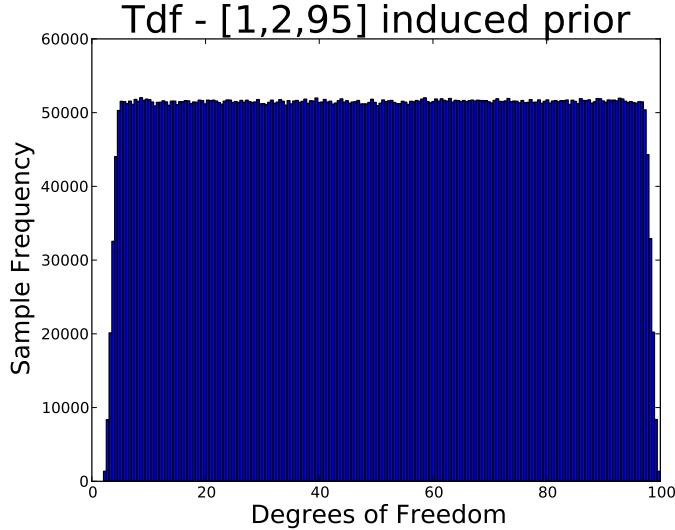


Figure 3.1: Distribution induced by partitioning the prior (uniform-continuous 2 100) into (uniform-continuous 2 95) + (uniform-continuous 0 2) (uniform-continuous 0 1).

3.2.1 Finding a good sum decomposition

First we look at the expected number of steps needed to reach a neighbourhood of the mode. As explained in Section 3.1.2, the expected number of steps to $[mode - \epsilon, mode + \epsilon]$ can be analytically computed for the unpartitioned prior (uniform-continuous a b) as $(b - a)/(2\epsilon)$.

For the partitioned priors, we instead perform empirical tests. These tests measure how many samples it takes for different partitions to reach mode neighbourhoods of different sizes. One thousand runs are done for each partition and neighbourhood sizes of 1, 0.5, 0.2 and 0.02 are considered. The partitions consist of between 2 and 5 values which were drawn with replacement from $[0, 0.5, 1, 2, 5, 7, 10, 15, 20, 25, 30, 35, 40, 45]$.

Table 3.3 contains the empirical performance for the unpartitioned prior and for the best 2 partitions on each neighbourhood size. We specify partitions in the format (x,y,z) meaning (uniform-continuous 0 x) + (uniform-continuous 0 y) + (uniform-continuous 2 z). All considered partitions respect the constraint $x + y + z = 98$, so that the sum of all the component samples will be in the $[2, 100]$ range specified by the uniform prior of the Tdf model, (uniform-continuous 2 100).

The best partition depends on the size of the neighbourhood, but we can ob-

Partition	Target neighbourhood size			
	1	0.5	0.2	0.02
Unpartitioned	98.38	199.29	494.87	4919.75
(5, 93)	89.03	122.87	172.9	670.14
(20, 78)	92.55	142.6	259.21	1834.18
(2, 96)	92.56	117.35	157.83	371.9
(1, 45, 52)	123.97	146	172.78	400.01
(0.5, 2, 95.5)	134.24	162.03	201.57	297.79
(1, 2, 95)	130.06	155.48	186.83	317.31

Table 3.1: Expected number of steps to mode neighbourhoods on the Tdf continuous model for an unpartitioned prior and some of the best sum decomposition priors.

serve partitions that consistently and significantly outperform the unpartitioned prior on all the neighbourhood sizes looked at above. In fact, for epsilon values of 0.1 and 0.01 the unpartitioned prior performs worse than any of the partitioned variants. Additionally, as epsilon gets smaller, it becomes useful to have smaller partition components. This leads to the 0.5 partition appearing in the best solution for a mode neighbourhood of size 0.02, but not in any of the top solutions for larger neighbourhoods.

The second aspect of convergence that a partition might help with is the mixing rate around the mode. In order to test the priors' mixing properties we need to consider what happens after the markov chain reaches the mode. We do this by setting the initial sample to the mode of the true posterior distribution and checking how much the chain moves over the next 1000 samples. Repeating this test 100 times and averaging the sum of absolute jump distances gives us the results in Table 3.2.

This test only measures the average distance travelled (i.e. sum of absolute differences between consecutive samples), which could be a misleading measure of mixing. However, the large difference between the unpartitioned and the partitioned variants do suggest that there is some improvement here. We perform further tests on the mixing rate in Section 3.2.2.

Based on the above results, we decide to further investigate the performance of the (1, 2, 95) partitioned prior, since this decomposition performs well both in reaching the mode and in mixing around it.

Partition	Mean distance travelled	Variance in distance travelled
Unpartitioned	7.75	13.43
(1, 1, 1, 95)	137.55	393.96
(1, 1, 1, 1, 94)	135.08	616.41
(1, 1, 2, 94)	131.19	539.03
(1, 1, 1, 2, 93)	129.91	583.16
(0.5, 1, 1, 1, 94.5)	126.76	382.09
(1, 2, 95)	125.67	408.16
(1, 1, 96)	125.44	311.76
(0.5, 1, 2, 94.5)	123.38	487.26
(1, 1, 2, 2, 92)	122.9	615.54
(2, 2, 94)	121.25	844

Table 3.2: Average distance travelled around the mode on the Tdf continuous model for an unpartitioned prior and some of the best sum decomposition priors.

3.2.2 Evaluating the (1,2,95) sum decomposition

Table 3.3 makes clear the increased speed in reaching a neighbourhood of the mode offered by the decompositions. The potential benefit conferred in mixing rate is less clear however. To test this we look at the sample evolution (Figure 3.2) and autocorrelation plots (Figure 3.3) for two runs obtained with an unpartitioned and a (1,2,95) partitioned prior. These experiments confirm our preliminary results from Section 3.2.1, and show that the partitioned prior does help with mixing around the mode and with eliminating large correlations between consecutive samples.

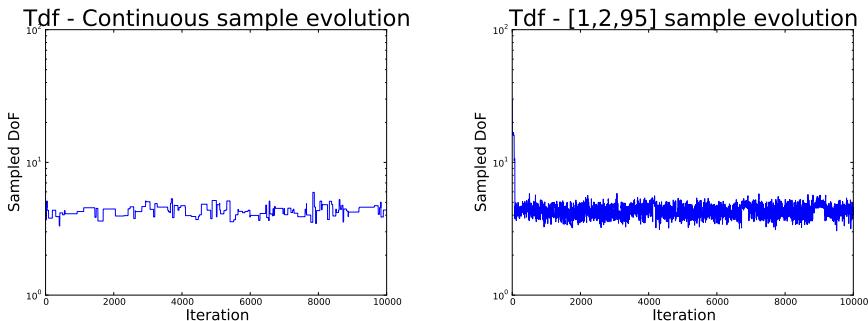


Figure 3.2: Sample evolutions for the unpartitioned and the (1, 2, 95) partitioned priors, over 10,000 samples on the Tdf continuous model.

The final test in determining the quality of the decomposition is to look at the actual sample distributions obtained under the two different prior formulations . For convenience, the true posterior for the Tdf continuous model is shown again in Figure 3.6. Looking at the sample distributions in Figure 3.4 it is quite clear

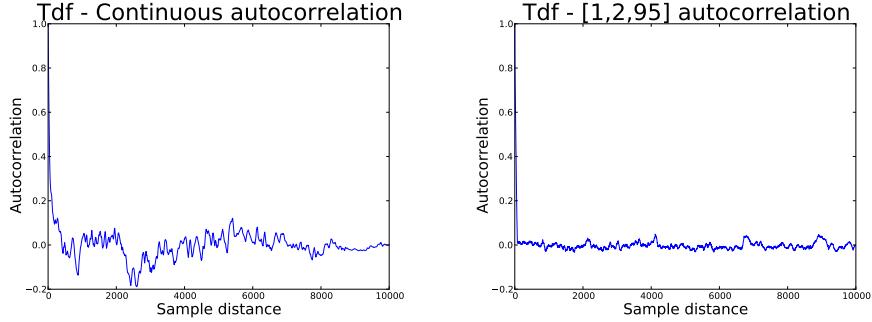


Figure 3.3: Sample autocorrelations for the unpartitioned and the (1, 2, 95) partitioned priors, over 10,000 samples on the Tdf continuous model.

the partitioned prior outperforms the original, unpartitioned, variant. However, this result may be misleading since we are evaluating it on the same distribution which we used to choose the form of the partition. To test the robustness of our partition we repeat the above tests on a new model.

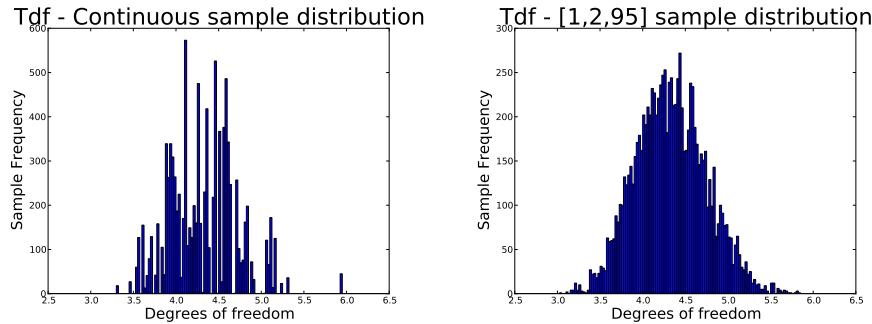


Figure 3.4: Sample distributions for the unpartitioned and the (1, 2, 95) partitioned priors, over 10,000 samples on the Tdf continuous model.

The Tdf21 model

In order to test our decomposition on a different posterior distribution, we generate 1,000 datapoints from a Student t distribution with 21 degrees of freedom and condition the Tdf Continuous model on this new dataset. The resulting posterior is shown in Figure 3.5. The mode here is actually 11.5. This is probably due to the fact that 1000 datapoints are not enough to accurately pinpoint a student-t with so many degrees of freedom (21) since, as the number of degrees of freedom increases, the differences between corresponding student-t distributions shrinks. The posterior distribution is however significantly different from the one for the previous dataset, which should be sufficient for testing the properties

of the priors.

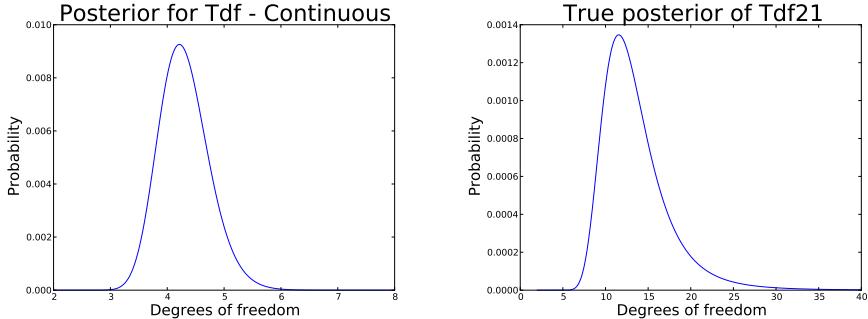


Figure 3.5: The true posteriors of the Tdf continuous and the Tdf21 continuous models.

In order to better understand how the Metropolis-Hastings algorithm will be affected by this change, Figure 3.6 also shows the log-likelihoods induced by the original Tdf Continuous model and by the Tdf21 continuous model. We can see that the Tdf21 log-likelihood is much flatter than the one for the original model. By repeating the mixing tests performed above we can test the effect of this difference.

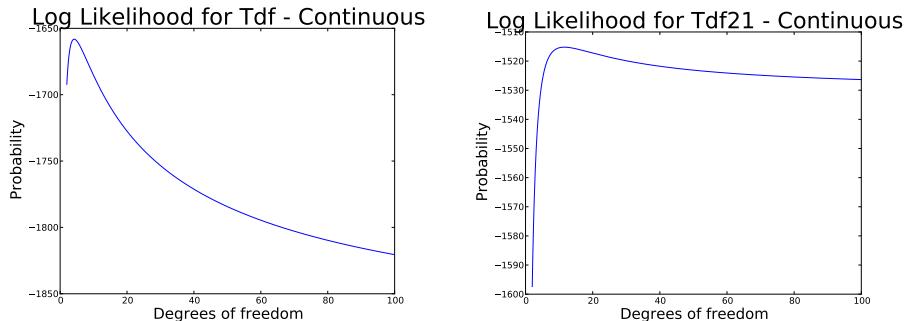


Figure 3.6: The true log-likelihoods of the Tdf continuous and the Tdf21 continuous models.

Evaluating the decomposition on the Tdf21 model

We now test the convergence of the priors on the Tdf21 model by plotting the sample evolution, the sample autocorrelation and the sample distributions. The partitioned prior is the same one we used on the previous datapoints, namely (1,2,95). From the sample evolutions (Figure 3.7) we can see that the partitioned samples tend to clump a little more since bigger changes in the samples only occur when the 95 component changes. Both versions seem to mix well though.

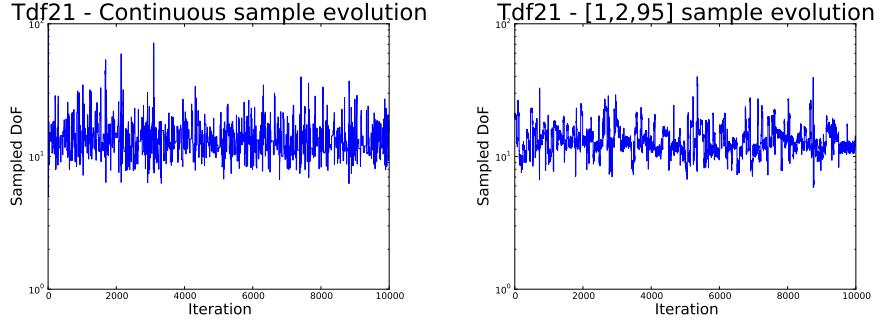


Figure 3.7: Sample evolutions for the unpartitioned and the (1, 2, 95) partitioned priors, over 10,000 samples on the Tdf21 model.

The autocorrelation plots shown in Figure 3.8 are also more similar than in the case of the original Tdf model. The unpartitioned prior does quite well here since the flat shape of the log-likelihood means there is a higher chance that a proposition drawn from the prior will be accepted by the Metropolis-Hastings algorithm.

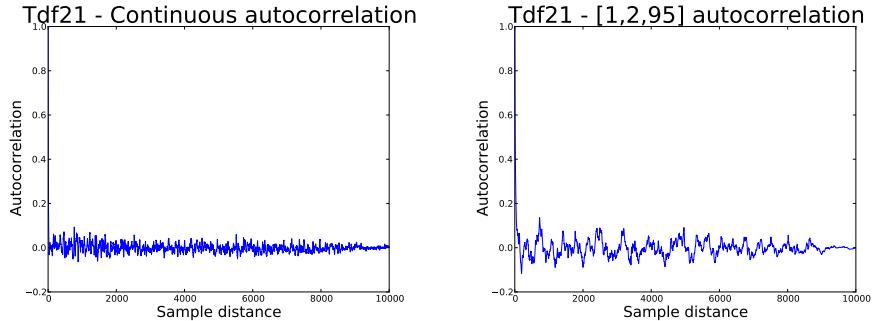


Figure 3.8: Sample autocorrelations for the unpartitioned and the (1, 2, 95) partitioned priors, over 10,000 samples on the Tdf21 model.

Finally, Figure 3.11 shows the sample distributions. Here we can see that, despite the similar performance of the two priors on the mixing tests, the partitioned prior still results in a significantly smoother distribution.

3.3 Bit decomposition

As mentioned in Section 3.2, one problem with the sum of uniforms decomposition is that it alters the shape of a uniform prior. We would like to come up with decompositions that leave uniform priors invariant and that could therefore be applied indiscriminately to re-write probabilistic programs containing

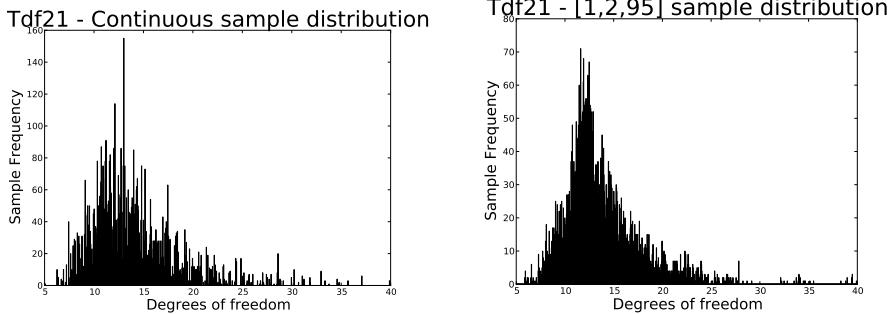


Figure 3.9: Sample distributions for the unpartitioned and the $(1, 2, 95)$ partitioned priors, over 10,000 samples on the Tdf21 model.

such distributions. A family of invariant partitions of an uniform prior can be constructed by thinking in terms of the bit representation of the uniform samples. In order to be able to represent any real number, we can consider a bit representation of the number up to a certain precision and then add a single uniform-continuous value to the bit's value.

3.3.1 Definition

In order to partition any uniform interval ($\text{uniform-continuous } a \ b$) it is sufficient to be able to partition the interval ($\text{uniform-continuous } 0 \ 1$). Once this is accomplished, the target interval can be obtained through the transformation: $(\text{uniform-continuous } a \ b) = a + (b-a) * (\text{uniform-continuous } 0 \ 1)$.

In order to partition the interval ($\text{uniform-continuous } 0 \ 1$) we first pick a bit depth, k , such that $k \in \{0, 1, \dots, \infty\}$. We then define $(\text{uniform-continuous } 0 \ 1) = \text{flip} * 2^{-1} + \text{flip} * 2^{-2} \dots + \text{flip} * 2^{-k} + (\text{uniform-continuous } 0 \ 2^{-k})$, where flip is a function which flips a coin and returns 0 or 1 with probability $1/2$ each.

3.3.2 Evaluation on Tdf and Tdf21

To get an idea of the properties of bit decomposition we perform empirical evaluations on the Tdf and Tdf21 continuous models. Looking at the sample distributions in Figures 3.10 and 3.11 we can see that a 3rd degree bit decomposition obtains similar performance to an unpartitioned prior. Intuitively this is because we are still subjecting proposals to the Metropolis-Hastings acceptance ratio, so if our program decides to flip one of the leading bits the proposal will be rejected, leading to bad mixing. As the depth of the bit decomposition increases, however, the probability that one of the leading bits is picked diminishes

and therefore we will expect the mixing rate to improve. We, however, restrict ourselves to a 3rd degree decomposition here since a higher degree variant would exhibit the problem “stuck samples”, which is discussed in Section 3.3.3.

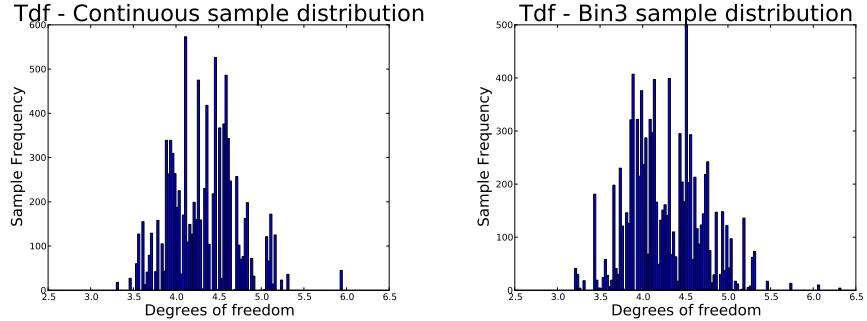


Figure 3.10: Sample distributions for an unpartitioned and a 3 bit decomposition prior, over 10,000 samples on the Tdf continuous model.

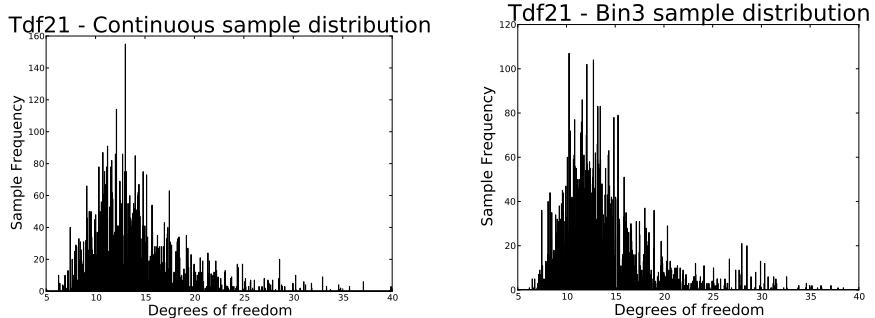


Figure 3.11: Sample distributions for an unpartitioned and a 3 bit decomposition priors, over 10,000 samples on the Tdf21 model.

The expected time to reach a mode neighbourhood is analyzed in Table 3.3. Here we can see that the 3rd degree binomial provides a significant improvement over the unpartitioned prior, though not as significant as the sum of uniforms does. On the depth 7 bit decomposition, we see good results on the small neighbourhoods but erratic ones on the larger neighbourhoods. The reason is again the fact that bit decompositions can result in “stuck samples”, which are discussed in Section 3.3.3.

Model	Partition	Target neighbourhood size			
		1	0.5	0.2	0.02
Both	Unpartitioned	98	196	490	4900
Tdf Continuous	(1, 2, 95)	130.06	155.48	186.83	317.31
	Bit 3	52	95.3	272.6	2232.4
	Bit 7	1080.78	866.64	776.26	1433.37
Tdf21 Continuous	(1, 2, 95)	93.5	128.8	173.9	714.98
	Bit 3	117.52	170.65	401.5	2969.9
	Bit 7	161.86	177.22	284.99	1739.84

Table 3.3: Expected number of steps to neighbourhoods of the mode on the Tdf and Tdf21 continuous models for an unpartitioned prior, the (1,2,95) sum decomposition and 2 bit decompositions.

3.3.3 Getting stuck on a bad sample

A problem with the bit decomposition is that it is possible to construct scenarios in which a sample will get stuck and be arbitrarily unlikely to reach a particular neighbourhood of the mode. One simple such scenario can be constructed by assuming a guiding posterior log-likelihood that is convex, symmetric around the mode and steep enough that the probability of a sample moving significantly further away from the mode is negligible.

The simplest example of getting stuck can be observed by considering the bit decomposition of depth 1: $(\text{uniform-continuous } 0 \ 1) = \text{flip} * 2^{-1} + (\text{uniform-continuous } 0 \ 2^{-1})$. In this case, if the mode is in the interval $[0.5 + \epsilon, 0.75]$, and $[\text{mode} - \epsilon, \text{mode} + \epsilon]$ is the mode neighbourhood we want to reach, then it is possible for the prior to get stuck outside of our target neighbourhood.

A concrete example would be:

$$(\text{uniform-continuous } 0 \ 1) = \text{flip} * 2^{-1} + (\text{uniform-continuous } 0 \ 2^{-1})$$

$$\text{mode} \sim 0.6$$

$$\epsilon \sim 0.05$$

$$\text{flip} \sim 0$$

$$(\text{uniform-continuous } 0 \ 2^{-1}) \sim 0.4$$

In this case, setting flip to 1 would be very likely rejected since jumping from 0.4 to $0.4 + 0.5 = 0.9$ takes us much further away from the mode at 0.6 than

we currently are. Further the uniform is likely to only accept resampled values in the $(0.4, 0.5)$ interval, which won't change the above situation. In order for us to get unstuck we would need either a very unlikely bit flip to be accepted or the uniform to accept a very unlikely resample close to 0, which could then be followed by a bit flip. Assuming we can make the log-likelihood arbitrarily steep around the mode, then we can expect to be stuck in this local optimum for an arbitrarily long number of samples.

Some possible solutions to avoid getting stuck would be:

- Having the option of changing multiple bits at a time (eg: sample a variable to determine how many bits to change). This would ensure there are no hard local maximas. However, situations would still arise where a large number of bits would need to be concurrently changed to specific values in order for a sample to be accepted, which means we might still be stuck in a certain position for a long time until just the right combination of bits is picked.
- Using multiple shifted variants of the prior, where shifting by $shift$ means mapping sample x to $(x + shift)\%1$. It can be shown that a single shift is enough to avoid getting stuck. However the effect on performance of adding shifts is complex, since performing a shift in an unstuck position can lead to us moving further away from the mode. More analysis of the effect of shifts is presented in 3.3.4.
- Using multiple variants of the prior with different bit depths. If a sample is stuck on bit b , then moving it into a prior with depth $< b$ will unstuck it. However, this will result in the bits with the highest values being resampled more often, since they will be present in the most priors, which will have a negative effect on the mixing benefits offered by the decomposition.

One idea that is tempting but incorrect is to determine, with some likelihood, when a certain markov chain has become stuck based on its sample history. This would allow us to explicitly correct for the chain getting stuck. We could toss a coin to decide if we think we're currently stuck and if we decided we were not stuck we could sample normally. If we did think we were stuck, we could determine in which bits we might be stuck and pick one of these. We could then determine the interval in which the mode should be if we were indeed stuck on the bit we picked and sample uniformly from that interval.

This approach is attractive because, for the k th bit $flip * 2^{-k}$, the target interval we would determine the mode to be in would be of size 2^{-k-1} . However, the problem with this idea is that it isn't just picking proposals from a static prior

or proposal kernel, but the proposal pattern is actually influenced by the sample history. This formulation would be conceptually similar to varying the width of our Metropolis-Hastings proposal kernel $Q(x_{new}|x_{old})$ based on the simulation history. However performing any such dynamic modification is not allowed if we wish to maintain the correct stationary distribution for our underlying Markov Chain [21, p. 380].

3.3.4 Mixtures of shifted bit decompositions

In this section we explore two variants of the mixture of shifted bits. First we look at what is necessary simply to ensure that we'll never get stuck. Second we look at what is needed to be able to move from any stuck position to a neighbourhood of the mode in one jump. This second variant ends up providing better performance.

Avoiding getting stuck

It turns out that a mixture of 2 bit decompositions, one of which is a shifted variant of the other, is sufficient to ensure that it is impossible to get stuck. However it seems we need to use a mixture of K priors in order to ensure that the binomial of depth K can move from any stuck position to the mode in one step.

To better understand how these shifts work, we consider the formulation:

$$\begin{aligned} \text{total} &= \text{flip1} * 0.5 + \text{flip2} * 0.25 + (\text{uniform-continuous } 0 \text{ } 0.25) \\ \text{flip1} &\sim 0 \\ \text{flip2} &\sim 1 \\ (\text{uniform-continuous } 0 \text{ } 0.25) &\sim \epsilon \\ \Rightarrow \text{total} &= 0.25 + \epsilon \quad (\epsilon \text{ is an arbitrarily small positive number}) \end{aligned}$$

Here we are stuck since, in order to reach the mode, we need to switch the flip2 bit to 0, but this flip will be rejected as it would mean moving significantly further away from the mode than we currently are. We want to determine what size of a shift needs to exist in order for us to be able to become unstuck.

For any shift s we choose, the shifted prior will be:

$$\text{shiftTotal} = (s + \text{total}) \% 1 = (s + 0.25 + \epsilon) \% 1$$

And flipping the second bit (*flip2*) to 0 would result in a proposal:

$$proposal = s + \epsilon$$

We can now show that, if a shift is too small, then the proposal will be rejected. Specifically:

If

$$mode = 0.25 - \epsilon$$

$$s < 0.125 - 2 * \epsilon$$

Then

$$\begin{aligned} |shiftTotal - mode| &= |s + 0.25 + \epsilon - 0.25 + \epsilon| = s + 2 * \epsilon < 0.125 \\ |proposal - mode| &= |s + \epsilon - 0.25 + \epsilon| = 0.25 - s + 2 * \epsilon > 0.125 \\ \Rightarrow distance(proposal, mode) &> distance(shiftTotal, mode) \end{aligned}$$

(which means the proposal is rejected)

The only other proposals we could make is flipping the first bit (*flip1*) to 1 or increasing the uniform, both of which are also rejected since they move away from the mode in both shifts. Therefore, for the shift to be useful on a stuck 2nd bit, we need that $s \geq 0.125$. Extrapolating, we see that in general, to get unstuck on the k th bit we require that $s \geq 2^{-k-1}$. Therefore it is sufficient to have one shift s such that $s \geq 2^{-2}$ in order to guarantee that we never get stuck.

If, in addition to not getting stuck, we also want to be able to reach the mode in one jump from a stuck position, we must consider an additional example:

$$\begin{aligned} total &= flip1 * 0.5 + flip2 * 0.25 + (\text{uniform-continuous } 0 \text{ } 0.25) \\ flip1 &\sim 0 \\ flip2 &\sim 0 \\ (\text{uniform-continuous } 0 \text{ } 0.25) &\sim 0.25 - \epsilon \\ \Rightarrow total &= 0.25 - \epsilon \quad (\epsilon \text{ is an arbitrarily small positive number}) \end{aligned}$$

As before, for any shift s we choose, the shifted prior will be:

$$shiftTotal = (s + total)\%1 = (s + 0.25 - \epsilon)\%1$$

And the only proposal that decreases the sample size is from reducing the

uniform-continuous variable. Therefore

$$proposal \geq s + \epsilon$$

We can now show that a shift that is too large fails to satisfy our criteria. Specifically:

If

$$mode = 0.25 + \epsilon$$

$$s > 0.25 + 2 * \epsilon$$

Then

$$|shiftTotal - mode| = |s + 0.25 - \epsilon - 0.25 - \epsilon| = s - 2 * \epsilon > 0.25$$

$$|proposal - mode| \geq s + \epsilon - 0.25 - \epsilon > \epsilon$$

$$\Rightarrow distance(proposal, mode) > \epsilon$$

(which means we can't reach the mode in 1 step)

The only other proposals possible in this situation are to shift one of the bits to 1, which would just take us farther from the mode. In the above scenario, it is therefore impossible to reach the mode by performing a shift and a proposal.

Note that, in this situation we are not stuck. For instance, if we accept (uniform-continuous 0 0.25) = ϵ we will have $shiftTotal > mode + \epsilon$. If we then switch back to the original 0shift we obtain in $total = mode + \epsilon - s = mode + \epsilon - s$. And since $s > 0.25 + 2 * \epsilon$ we would now have $total < mode - 0.25 - \epsilon$ and we would therefore be in a position to accept switching the 2nd bit to 1, which would unstuck us.

However, if we wish to jump to the mode from a position stuck on the second bit, we've shown that the shift must have the property:

$$0.125 \leq s \leq 0.25$$

And in general, to guarantee that we can jump to the mode when stuck on the k th bit, we need to have a shift s such that:

$$2^{-k-1} \leq s \leq 2^{-k}$$

This implies that we need a different sized shift for every bit.

Empirical performance

The arguments in the previous section suggest that the placement of the mode can significantly affect the likelihood of getting stuck. In order to get a better idea of the effect of the mode location we test the burn-in time for different mode placements. First we look at burn-in time averaged over all mode placements m , where $m \in [0.0005, 0.0015, 0.9995]$ and $\epsilon = 0.0005$.

In 3.12a we look at the case where we resample the shift on each iteration and we have shifts of size 2^{-k} (called maximum shifts) for each bit position k . Here the unpartitioned prior corresponds to the bit decomposition of depth 0. While the improvements in burn-in rate are not as significant when averaging over all mode placements as they were for our initial experiments on the Tdf models, a 2x speed-up can still be obtained.

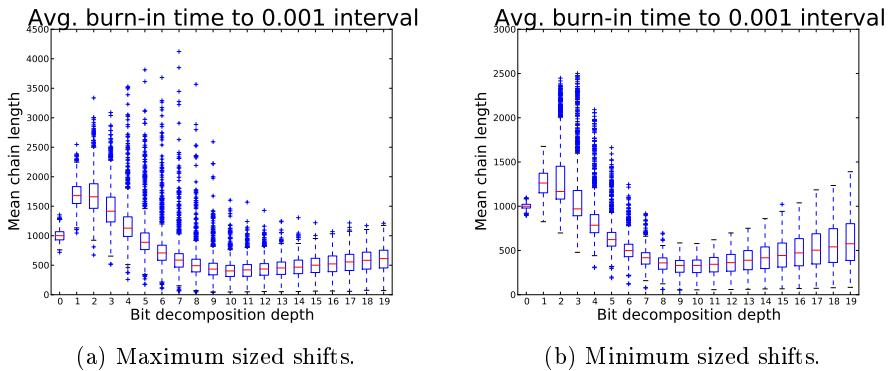


Figure 3.12: Time to 0.001 neighbourhood of mode, averaged over all mode placements, for bit decompositions of different depths using a shift for every bit.

As explained in Section 3.3.4, the shift size s for bit k needs to respect $2^{-k-1} \leq s \leq 2^{-k}$ in order for us to be able to jump to the mode in one step. In 3.12b we check whether the choice of shift size within this interval is significant, by looking at the burn-in rate for shifts of size 2^{-k-1} (called minimum shifts) for each bit position k . The results are quite similar to those obtained for the maximum shifts, which suggests that the performance isn't sensitive to the choice of shift length within $[2^{-k-1}, 2^{-k}]$.

We would also like to see what happens as the size of the target neighbourhood changes. In Figure 3.13 we repeat the max shift experiment for a target neighbourhood of size 0.01 (10x larger). While the bit decomposition still gives some advantage, as the size of the target neighbourhood increases this advantage appears to become less significant. Intuitively this happens because a larger

number of the less significant bits become irrelevant as far as ending up in the desired region is concerned.

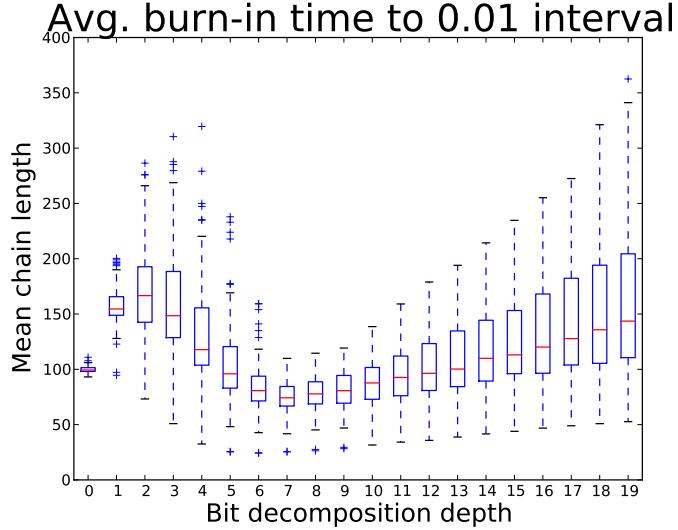


Figure 3.13: Time to 0.01 neighbourhood of mode, averaged over all mode placements, for bit decompositions of different depths using a shift for every bit.

Finally, we would also like to see what happens to the performance if we do not provide k shifts for k bits, but instead only 1 or 2 shifts. As explained in section 3.3.4, a single properly sized shift is sufficient to stop us from getting stuck. However, without k shifts for k bits, we sacrifice the ability of always moving from a stuck position to the mode in one step. As seen in Figure 3.14, this seems to have a significant negative effect on the decompositions' performance.

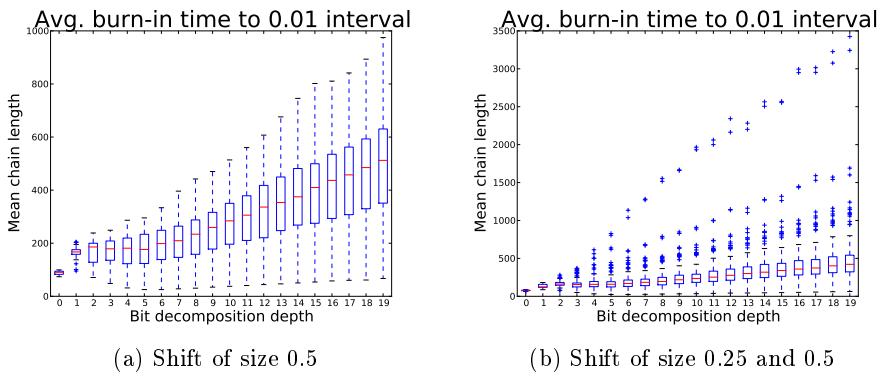


Figure 3.14: Time to 0.01 neighbourhood of mode, averaged over all mode placements, for bit decompositions of different depth using a small number of shifts.

Chapter 4

Novel PPL inference techniques

In the previous chapter we attempted to re-write probabilistic programs in such a way as to improve the performance of basic Metropolis-Hastings inference. However this approach may prove limiting, so we are also interested in exploring different inference techniques that may perform better, at least on some subset of possible models. In this chapter we explore one such technique, slice sampling, while also briefly looking at some issues of tangential interest.

4.1 Preliminaries

In order to implement a PPL based on a new inference technique we need to understand both how the inference technique in question works and how we may build a PPL in general. In this section we give a description of the main idea behind slice sampling and briefly explain one PPL construction technique.

4.1.1 Slice sampling

Like the Metropolis-Hastings method presented in Section 3.1.1, slice sampling is a Markov Chain Monte Carlo algorithm that can extract samples from a distribution $P(x)$ given that we can evaluate a proportional function $P^*(x)$ such that $P^*(x) = P(x) * Z$. The idea behind slice sampling is that, by using an auxiliary height variable u , we can sample uniformly from the region under the graph of the density function of $P(x)$ [30].

For ease of explanation, we consider slice-sampling in a one dimensional setting. Here we can view the algorithm as a method of transitioning from a point (x, u) which lies under the curve $P^*(x)$ to another point (x', u') lying under the same curve. The addition of the auxiliary variable u means that we need only sample uniformly from the area under the curve $P^*(x)$.

A basic slice sampling algorithm can be described as:

- 1 Pick an initial width w_{init}
- 2 Pick an initial sample x_0 such that $P^*(x_0) > 0$
- 3 Pick a desired number of samples to extract N
- 4 For(t=0; t<N; t++)
 - 4.1 Sample a height $u \sim (\text{uniform-continuous } 0, P^*(x_t))$
 - 4.2 Define a horizontal line across the $P^*(x)$ curve at the u height
 - 4.2.1 Initialize $x_l = x_t$
 - 4.2.2 For (iter=0; $P^*(x_l) > u$; iter++)
 - 4.2.2.1 Exponentially decrease left bound $x_l = x_l - 2^{iter} * w_{init}$
 - 4.2.3 Initialize $x_r = x_t$
 - 4.2.4 For (iter=0; $P^*(x_r) > u$; iter++)
 - 4.2.4.1 Exponentially increase right bound $x_r = x_r + 2^{iter} * w_{init}$
 - 4.3 Sample uniformly from the slice above the line between x_l and x_r
 - 4.3.1 Sample a proposition $prop \sim (\text{uniform-continuous } x_l, x_r)$
 - 4.3.2 While ($P^*(prop) \leq u$)
 - 4.3.2.1 If ($prop > x_t$) $\{x_r = prop\}$
 - 4.3.2.2 Else $\{x_l = prop\}$
 - 4.3.2.3 Resample $prop \sim (\text{uniform-continuous } x_l, x_r)$
 - 4.4 Accept the proposition $x_{t+1} = prop$

One of the main advantages of the above formulation over Metropolis-Hastings is that slice sampling self-tunes the step size. In MH, picking a wrong step size can significantly hamper progress either by unnecessarily slowing down the random walk's progress or by resulting in a large proportion of rejected samples. Slice sampling, however, adjusts an inadequate step size of size N with a cost that is only logarithmic in the size of N (because of the exponential stepping out and shrinking methods).

For a more complete discussion of slice sampling please see [30, 21].

4.1.2 Basic PPL Construction

There are many PPLs in existence and therefore many different techniques for building PPLs. Here we use the “Lightweight implementation” method proposed by Wingate [44], which can be used to modify most existing programming languages into PPLs with relative ease.

The idea behind this approach is to perform Metropolis-Hastings over probabilistic programs by evaluating the program’s traces and accumulating the likelihood of each random choice. The proposal kernels used in this variant of Metropolis-Hastings consist simply of the prior probability distributions of each variable in the program, and updates are performed one variable at a time.

Therefore, in order to create a proposal for our program we must pick a random variable and resample it. We then calculate the likelihood of the trace containing this resampled value, as well as all values we are conditioning on. In calculating the likelihood of this trace we would like to have to resample as few additional values as possible. This is both for efficiency reasons (we’d like to re-use variable samples), as to increase the likelihood of accepting the proposal (since a variable resampled from its prior could result in a bad trace likelihood) and because we would like to propose a next trace that is “close” to our current one.

An additional complication arises from the fact that many probabilistic programs are trans-dimensional, which is to say different traces may evaluate different numbers of variables. Therefore, in order for the likelihoods of different traces to be comparable, we need to keep track of the likelihoods of variables which either have just appeared in our current trace (fresh variables), or which were present in the previous trace but not in the current one (stale variables) [44]. In doing this, we are essentially implementing a reversible jump markov chain monte carlo method, which can be viewed as performing inference on the joint space of different dimensional models [12].

4.2 Stochastic Python

We implement a novel PPL, Stochastic Python, which is made available online [35] . Stochastic python has both a metropolis-hastings inference engine (implemented in the style presented in 4.1.2) and a slice sampling inference engine which we further discuss in the following sections.

In our implementation, the user can write an arbitrary python function and perform inference on this program. The only restriction on the user's function is that it must be self-contained, and so have the same output given the same input. In order to use our inference engine, the user must make use of the random primitives provided by the Stochastic Python library. These primitives are actually a thin wrapper around the equivalent `scipy.stats` primitives. In this way, Stochastic Python can keep track of things such as which variables we are conditioning on and which variables we want to observe. Having all random primitives go through the Stochastic Python library means our inference engine can keep track of all the relevant probabilities and compute the trace likelihoods needed by Metropolis-Hastings or slice sampling.

Since we created Stochastic Python in order to test slice sampling, we are currently relying on the user to provide a name for each random variable rather than pre-compiling the user's program to obtain this information. Alternatively the user can use a helper function to generate an appropriate probabilistic variable name by providing some basic information as input (like the name of the function holding his variable and the line number on which it is declared). We could relieve the user of this chore via dynamic stack examination, but that method proves to be a huge bottleneck causing 50x slow-downs in inference performance. However, it seems the method described by Wingate [44] for pre-compiling Matlab code could be readily extended to Stochastic Python as well.

4.3 Slice sampling inference engine

4.3.1 Custom Slice Sampling and Metropolis on Tdf models

As a preliminary test, we implement custom slice sampling and local metropolis-hastings algorithms for the Tdf continuous and Tdf21 continuous models (presented in Section 2.3.1). A comparison of the burn-in needed to reach a neighbourhood of the mode is presented in Figure 4.1. Slice sampling has a much shorter burn-in on these models. In fact slice sampling does better than any of the partitioned priors did (averaged over all the posterior mode placements).

However the performance of slice sampling does seem to vary with the shape of the likelihood. In Figure 4.2 we investigate this behaviour by modeling the likelihood as a Gaussian and seeing how slice performs as we vary the Gaussian's properties.

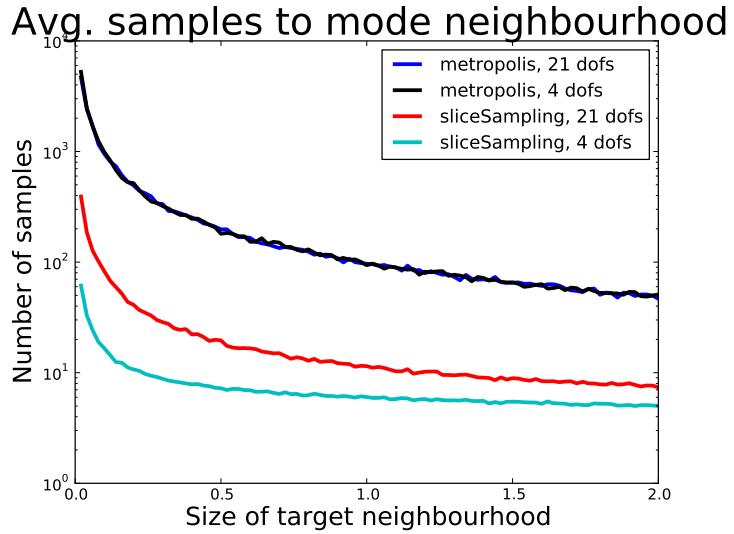


Figure 4.1: Burn-in time for local metropolis-hastings and slice sampling, on the two continuous Tdf models, as the target neighbourhood varies.

It seems that the placement of the likelihood in the interval doesn't affect performance, but the width of the distribution does. For reference, the Gaussian likelihoods with minimum and maximum standard deviations that we tested are shown in Figure 4.3.

Next, we examine the mixing properties of local metropolis-hastings and slice sampling by considering the sample evolution (Figure 4.4), the sample auto-correlation (Figure 4.5) and the empirical distributions (Figure 4.6) obtained by the two methods. For the obtained distributions to be comparable, we keep the number of log-likelihood computations performed by the two methods equal. This means that, while metropolis-hastings is allowed 10,000 samples, we only take 1962 from slice sampling (since, on this model, slice sampling averages just over 5 likelihood computations per extracted sample).

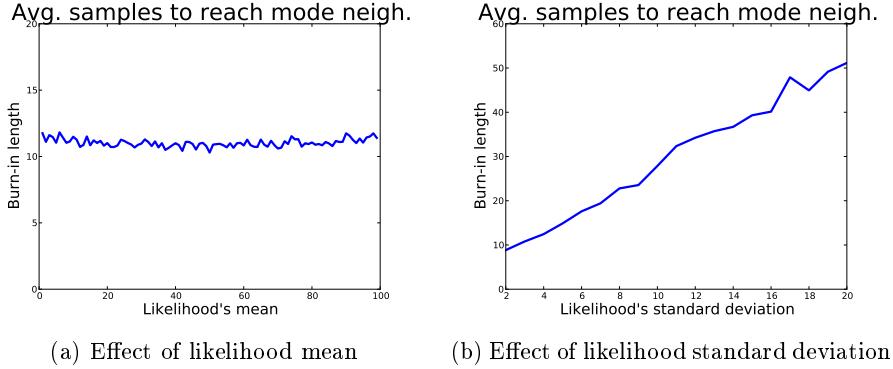


Figure 4.2: Average burn-in time for slice sampling guided by a Gaussian likelihood of varying mean and standard deviation

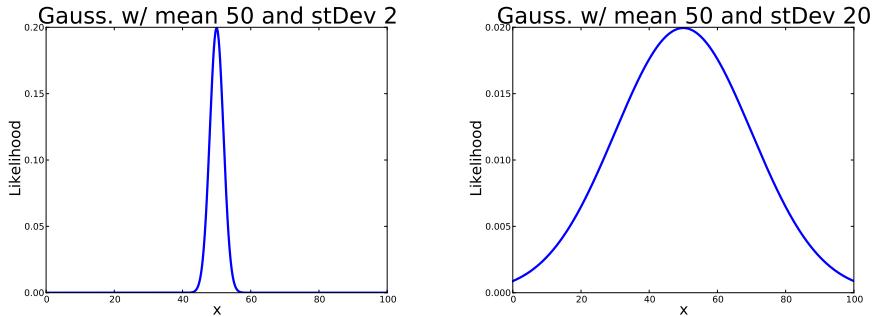


Figure 4.3: The smallest and largest Gaussian standard deviation considered in Figure 4.2

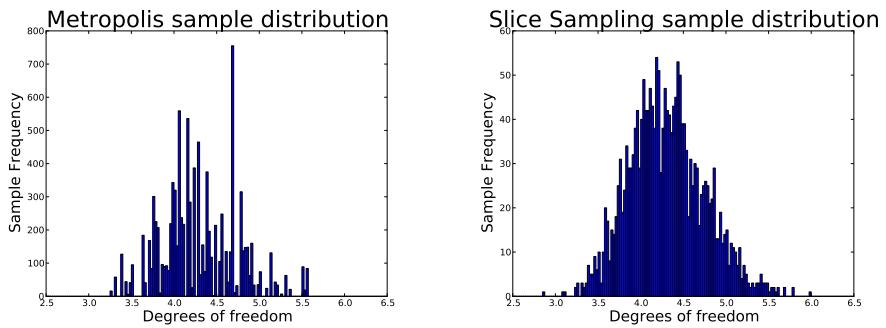


Figure 4.6: Empirical sample distribution of local metropolis and slice sampling on the Tdf continuous model

Based on these preliminary experiments slice sampling seems to confer a significant advantage over local metropolis when compared on the Tdf models.

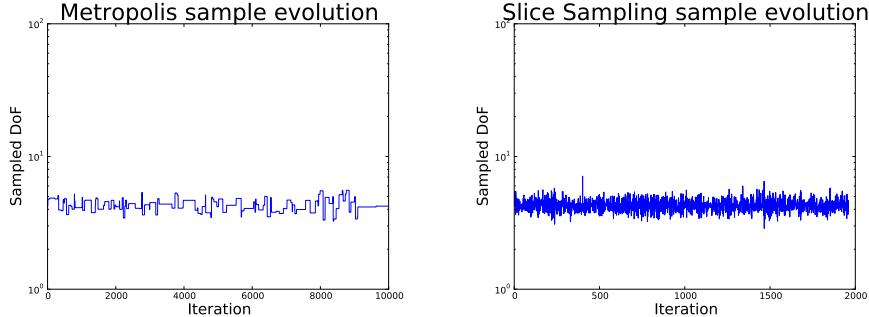


Figure 4.4: Sample evolution of local metropolis and slice sampling on the Tdf continuous model

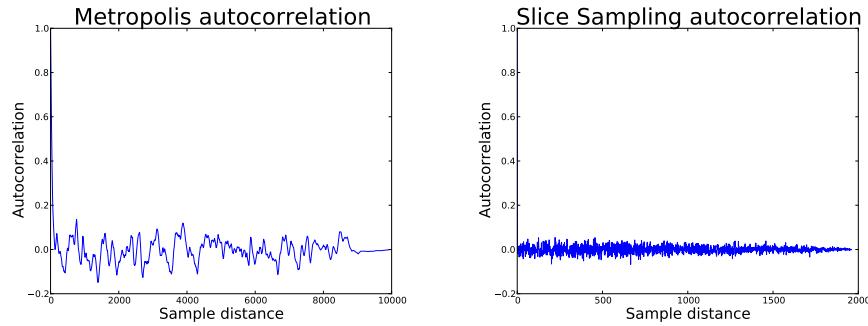


Figure 4.5: Sample autocorrelation of local metropolis and slice sampling on the Tdf continuous model

4.3.2 Generic, lightweight, slice sampling inference engine

Since the custom slice sampling tests on the Tdf models gave promising results we next look at creating a generic slice sampling implementation that can run on arbitrary probabilistic programs. Our implementation, described in Section 4.2, follows the ‘Lightweight Implementations’ style of the PPLs [44].

The slice sampling technique we use follows the basic ideas presented in Section 4.1.1. The main points are:

- sample a height u uniformly from $[0, likelihood]$. Since we are working with log likelihoods that are too small to be exponentiated, we take the sample directly in logspace by sampling from the exponential corresponding to the log of u . Specifically, if our log likelihood is ll , then $u \sim -1 * (\exp(1) + ll)$
- uniformly sample a random variable cur to modify
- find values xl and xr smaller and larger than the current value x of the random variable cur such that the log likelihood under xl and xr is smaller

than the height u . Search for these values in an exponential fashion, by doubling the last guess.

- uniformly sample a proposition for cur from $(\text{uniform } xl \ xr)$. Resample until the log likelihood under the sample is larger than the height u .

The stochastic python metropolis implementation presented in Section 4.2 runs the Tdf continuous model 2-2.5x slower than the Venture implementation. The metropolis implementation also runs about 6x faster than slice-sampling, per number of samples. As discussed above, the bottleneck is the number of trace likelihood calculations. The metropolis method calculated the log-likelihood exactly once for each sample while slice sampling will execute it a minimum of 3 times (one each for xl , xr and x). In practice, due to the stepping out and the possible resampling of a variable, we average 6 log-likelihood calculations per sample, thus explaining the 6x slow-down.

This shows that, for the tdf model, tweaking the initial width and interval search strategies as to reduce the number of log-likelihood calculations will result in a further improvement of no more than 2x.

Slice sampling on the Tdf model

We first run the 3 methods (my stochastic python metropolis and slice sampling implementations and Venture) for 10 minutes. The resulting distributions are shown in Figure 4.9.

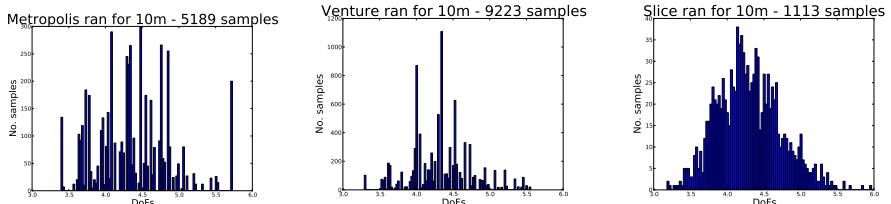


Figure 4.7: Sample distribution from running Venture and stochastic python versions of metropolis and slice sampling for 10 minutes on the Tdf continuous model.

Visually, slice sampling appears to be doing the best job despite generating much fewer samples in 10 minutes than the other methods. In order to get a quantitative evaluation of the methods we can use the Kolmogorov-Smirnov statistic and plot a graph of the decreasing differences between the true cumulative distribution and the cumulative distributions inferred by the 3 methods. This experiment (see Figure 4.8) confirms our intuition and shows slice sampling significantly outperforming the other variants.

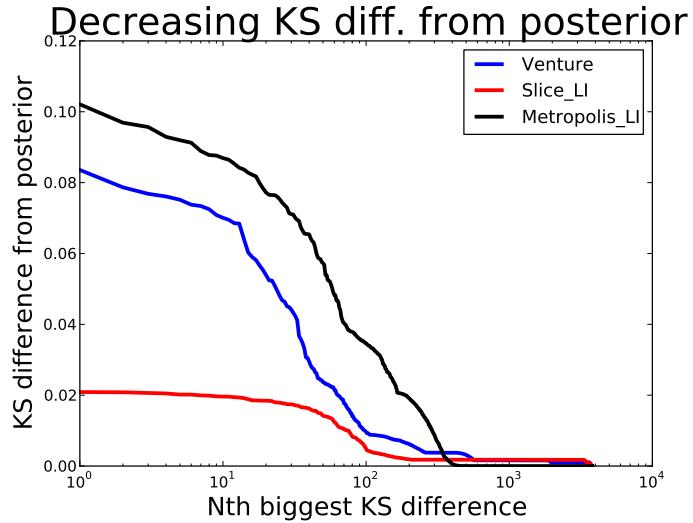


Figure 4.8: Comparison of Kolmogorov-Smirnov differences between true and inferred posteriors.

Slice sampling on gaussian mean inference models

To further test the inference performance of metropolis and slice sampling I defined 3 models based on the problem of estimating the mean of a gaussian. The 3 models are and their posteriors are:

$$\begin{array}{lll}
 \text{NormalMean1 :} & \text{NormalMean2 :} & \text{NormalMean3 :} \\
 m \sim N(0, 1) & m \sim N(0, 1) & m \sim N(0, 1) \\
 \text{observe } N(m, 1) = 5 & v \sim invGamma(3, 1) & \text{if } m < 0 \\
 \text{predict } m & \text{observe } N(m, v) = 5 & v \sim invGamma(3, 1) \\
 & \text{predict } m & \text{else} \\
 & & v = 1/3 \\
 & & \text{observe } N(m, v) = 5 \\
 & & \text{predict } m
 \end{array}$$

We now look at the performance of metropolis, slice sampling and different mixtures of slice sampling and metropolis (with different mixing proportions) over the 3 NormalMean models. The mixture methods work by flipping a biased coin before extracting each sample in order to decide which inference method to use. Since slice sampling cannot handle trans-dimensional jumps, we dissallow such jumps in the mixture algorithms. These algorithms are therefore reliant on

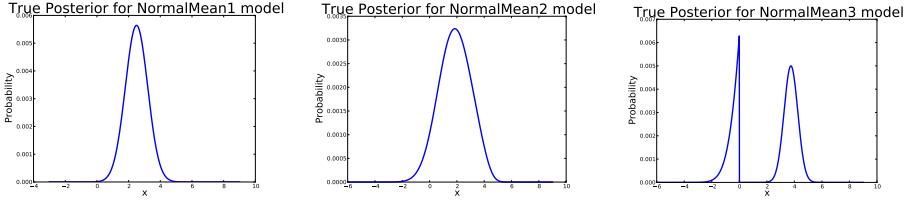


Figure 4.9: Analytically derived posteriors of the NormalMean1, NormalMean2 and NormalMean3 models.

metropolis to switch between program traces with different numbers of variables.

To compare the inference engines, we extract samples until a certain number of trace likelihood calculations are performed and then repeat this process 100 times, in order to generate independent sample runs (starting from different random seeds). Figure 4.10 shows both the runs and the quartiles of the runs on the first model.

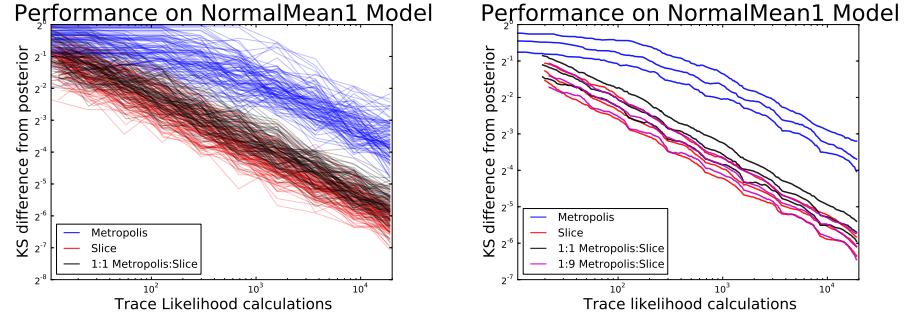


Figure 4.10: Runs and quartiles generated by slice, metropolis and mixtures of metropolis and slice on the 1 dimensional NormalMean1 model.

On the simple, 1d, model all variants of slice sampling clearly outperform metropolis. In the quartile graph we consider mixtures of metropolis and slice both with 10% metropolis and with 50% metropolis and find that the change doesn't have a significant impact on performance. This is likely because, if slice picks good samples, metropolis is likely to simply keep them unchanged (since it will tend to reject the proposal from the prior if they are worse).

On the 2d model (see Figure 4.11), slice still clearly outperforms metropolis, though the gap is not as pronounced as for the 1d model. Further, as in the 1d model, the 3 different slice variants all get quite similar performance. Additionally, on this model, the fact that the slice mixtures get more samples per LL calculation translates into a slightly better performance for them than for the pure slice sampling method.

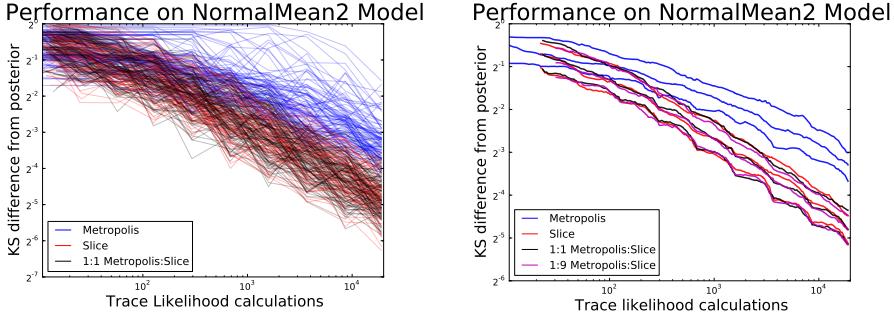


Figure 4.11: Runs and quartiles generated by slice, metropolis and mixtures of metropolis and slice on the 2 dimensional NormalMean2 model.

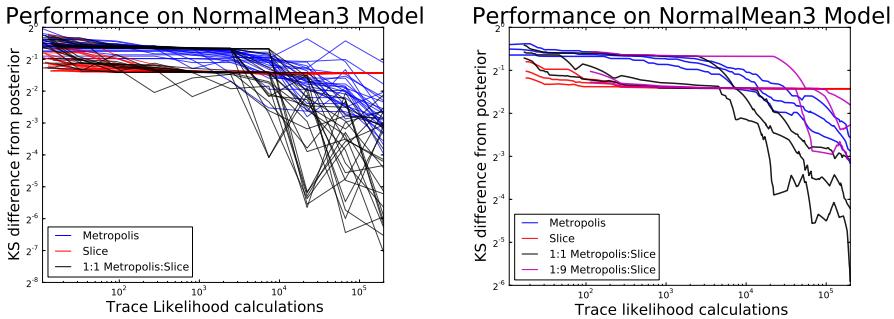


Figure 4.12: Runs and quartiles generated by slice, metropolis and mixtures of metropolis and slice on the trans-dimensional NormalMean3 model.

The third model, seen in Figure 4.12, reveals several things worth noting. First of all, pure slice sampling does very badly. This is because the simple slice sampling algorithm used in this example cannot handle trans-dimensional probabilistic models, such as NormalMean3. We will look closer at this problem in Section 4.3.2. Further, on this model, we see the first significant performance gap between the different mixtures of slice and metropolis. Since slice sampling cannot handle trans-dimensional jumps, one of the main purposes of the metropolis steps in the mixture model is to switch between program traces with different dimensionality. In the case of the 1:9 Metropolis:Slice mixture, we see that this dimensionality switch happens quite rarely, and so the markov chain is stuck on bad samples for long runs. The 1:1 mixture of slice sampling and metropolis, however, manages to switch dimensionality sufficiently often and thus outperforms pure metropolis.

Branching Model

In order to further test the slice sampling inference engine we look at the Branching model considered in [45]. This is also a trans-dimensional model, but this time operating on discrete data. The model specification we use is:

Branching3 :

```

 $pois1 \sim Poisson(4)$ 
if  $pois1 > 4$ 
 $x = 6$ 
else
 $pois2 \sim Poisson(4)$ 
 $x = fib(3 * pois1) + pois2$       ( $fib$  is the fibonacci function)
observe  $Poisson(x) = 6$ 
predict  $pois1$ 

```

In order to test the convergence rate of the inference engines we must first analytically derive the true posterior for this model. This is given, up to values of 20 in Figure 4.13.

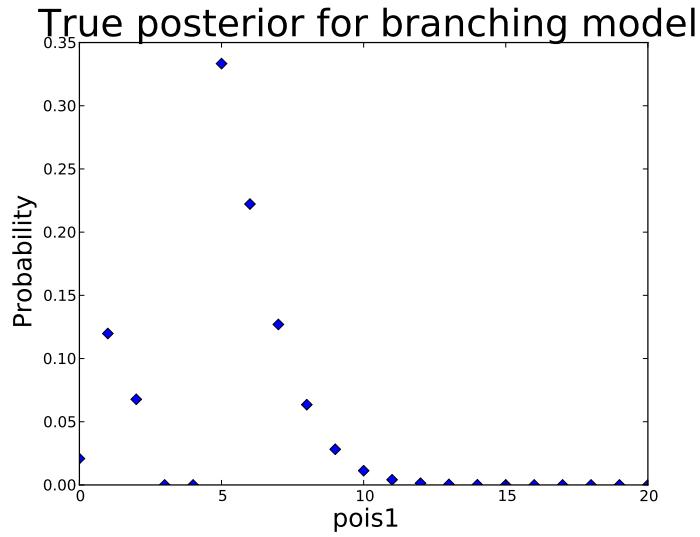


Figure 4.13: True posterior for the Branching Model

In order to evaluate the engines, we use each to generate 100 independent sample

runs, all performing an equal number of trace likelihood calculations. In Figure 4.14, we plot the evolution of the KL divergences between the empirical and the true posterior as the number of trace likelihood calculations increase.

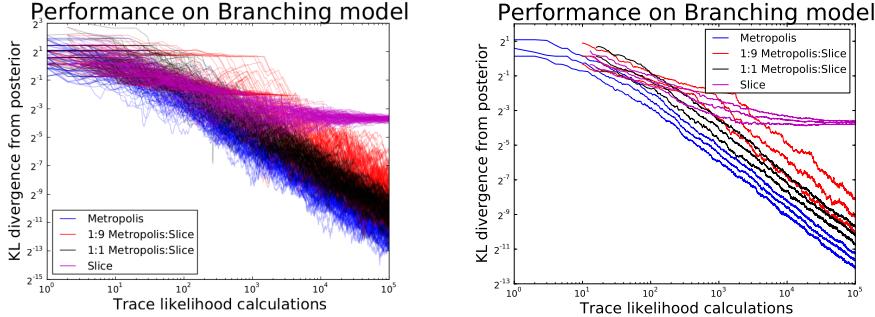


Figure 4.14: Runs and quartiles generated by slice, metropolis and mixtures of metropolis and slice on the Branching model.

As for the previous trans-dimensional model (NormalMode3), we see that the slice inference does not converge to the correct distribution since it cannot properly handle trans-dimensional jumps. The Metropolis:Slice mixtures do converge correctly but, on this model, are less efficient than the local Metropolis-Hastings.

One thing worth noting on this model, is that slice sampling proposes some values of `pois1` that are extremely unlikely (such as 60). The reason it proposes these values is that it picks a slice height based on the trace log-likelihood which in this model can be extremely low due to the distribution of the conditioned upon variable (`x`). In the Branching model, likely values of the 2 random variables (based on their priors) can result in very unlikely program traces and these traces can then result in accepting very unlikely values of our 2 random variables, since the acceptance criterion is simply that the proposed trace have likelihood higher than a number drawn uniformly from $[0, \text{oldTraceLL}]$.

However, we would expect this behaviour to only occur at the begining of a run, so the 1000 trace likelihood calculations burn-in period we are using should mitigate any influence this factor may have.

It's unclear why slice does worse on this model than on the NormalMean3 one, especially as the model distributions don't seem to affect the proposal efficiency, with both models averaging about 1 sample for 5 LL calculations.

It is informative to look at a per sample comparison of metropolis and slice sampling (see Figure 4.15), in addition to the previous per trace likelihood comparison (even though slice does "more work" to generate a sample than metropolis). In this plot we can see that the samples generated by 1:1 Metropolis:Slice are

actually slightly better than the pure metropolis ones. However the difference is not large enough to make up for the extra trace likelihood calculations that slice sampling must perform. We can also notice that the slice sampling mixtures experience a larger variance in performance than the pure metropolis method. This may be due to the fact that we are relying on only the metropolis generated samples to randomly switch between the 2 modes in the posterior.

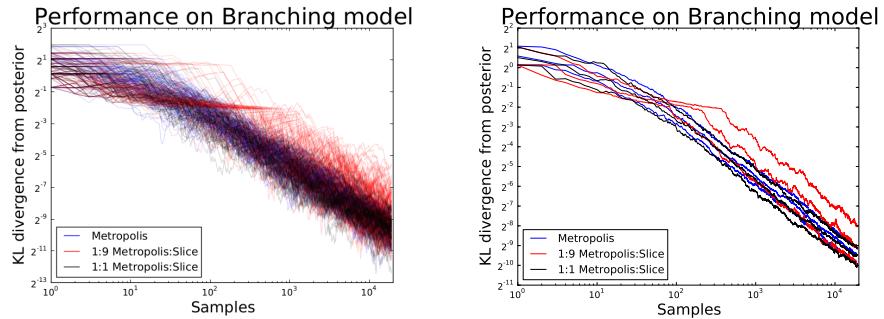


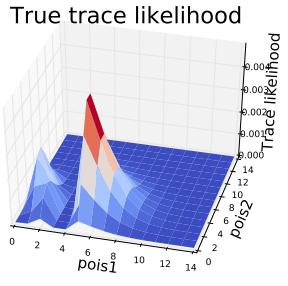
Figure 4.15: Runs and quartiles generated by slice, metropolis and mixtures of metropolis and slice on the Branching model.

Trans-dimensional slice sampling

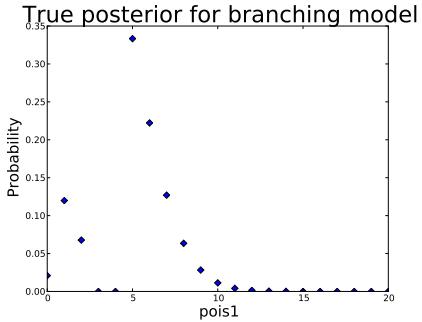
An interesting research question is whether (and how) it might be possible to modify the slice sampling algorithm so that it can correctly perform inference on probabilistic programs with varying numbers of dimensions.

As a pre-requisite to approaching this question, it is useful to investigate what is going wrong when trying to perform inference on the Branching and NormalMean3 trans-dimensional models. On the Branching model investigated in Section 4.3.2, we see that the model has 2 random variables whose values determine the distribution of a 3rd variable which we condition on. This model is trans-dimensional since on different traces either one or both of the 2 variables will be sampled.

Re-writing the model so that both variables are always sampled, even if one of them is unused, leaves the posterior invariant. Therefore one method to correctly perform inference in a trans-dimensional model is to always sample all the variables that might ever be used in any trace. This approach will however be extremely inefficient in large models and is not a viable general solution. In Figure 4.16a we use this trick to see what the space of possible trace likelihoods looks like. Integrating out the `pois2` variable from the above trace likelihood space results in the correct posterior distribution, shown in Figure 4.16b).

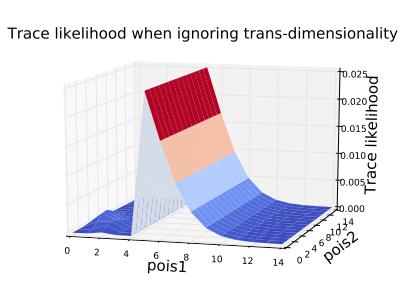


(a) Space of trace likelihoods if both variables are always sampled.

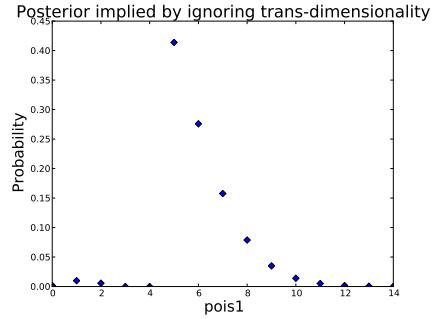


(b) True posterior of Branching model.

The issue with trans-dimensional jumps comes from the fact that the naive slice sampling algorithm will not sample the second poisson when it is not necessary, but will still think that the trace likelihoods between runs with different numbers of sampled variables are comparable. In doing so, the slice sampler will be pretending to be sampling from a 2D trace likelihood even when it really is 1D. The space of likelihoods implied by the naive slice sampling implementation is shown in Figure 4.16c. Integrating out the pois2 variable from this incorrect likelihood space results in the implied posterior shown in Figure 4.16d. This wrong posterior is the one which naive slice sampling will be attempting to infer.



(c) Space of trace likelihoods if both variables are always sampled.



(d) Space of trace likelihoods implied by naive slice sampling.

Next we'll look at one simple attempt to correct for trans-dimensional jumps, by thinking in terms of fresh and stale likelihoods, as in the metropolis acceptance ratio. We define a variable as being stale if it was used in the previous program trace but not in the current one. Conversely, a variable is considered fresh if it is being sampled in the current trace but was not used in the previous one. The correction we propose means that, when comparing a trace log-likelihood against the slice's sampled height, we won't simply consider the log likelihood

(ll), but instead $\text{ll} + \text{llStale} - \text{llFresh}$.

This correction means that when we are considering a jump to a lower dimensional space the log-likelihood of the lower dimensional space will be decreased by llStale (i.e. the likelihoods of the variables which are not part of this space). Conversely, when considering a move to a higher dimensional space, the log-likelihood of the higher-dimensional trace will be discounted by llFresh (so only the likelihood of the subset of variables that are also part of the current, lower-dimensional, space count). This simple correction seems to give correct results on the continuous NormalMean3 trans-dimensional model explored above (see Figure 4.16).

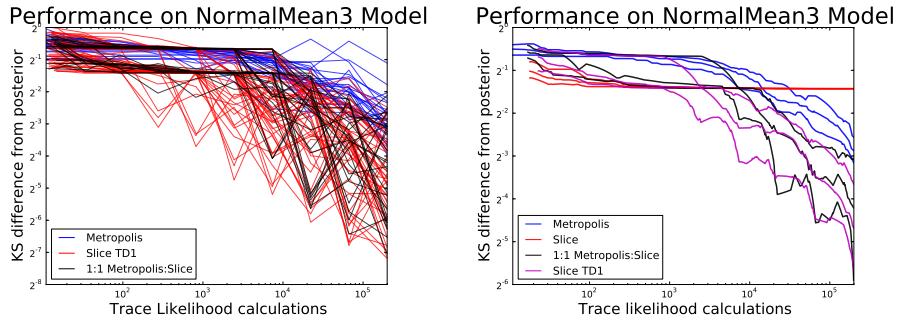


Figure 4.16: Runs and quartiles generated by metropolis, a mixture of metropolis and slice, and both the modified and the naive slice algorithms on the NormalMean3 model.

However, the specification appears to be wrong since it does not converge to the correct distribution on the Branching model (see Figure 4.17). Somewhat interestingly, it does seem to converge to a wrong value that's somewhat closer to the true posterior than the naive slice sampling does.

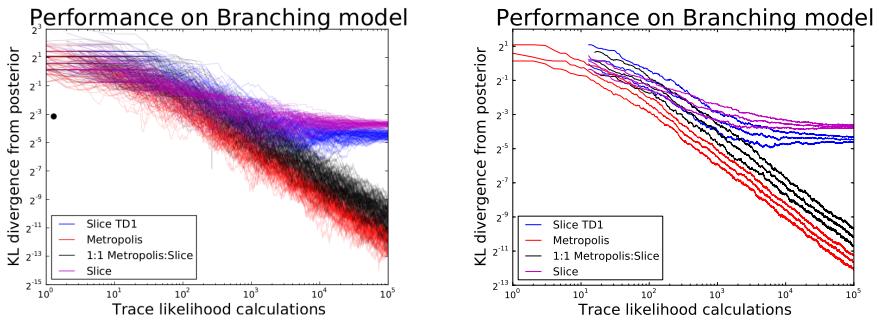


Figure 4.17: Runs and quartiles generated by metropolis, a mixture of metropolis and slice, and both the corrected slice and the naive slice algorithms on the Branching model.

4.4 Quasi-Monte Carlo

Another possible improvement on naive MC is, rather than sampling randomly from the unit interval, to instead make use of a low-discrepancy sequence that will tend to cover the interval faster [38].

A simple sequence we can use in the 1-dimensional case is the Van der Corput sequence. We test the performance of this sequence by considering time to mode of neighbourhood for a interval of size 100, neighbourhood of size 1 and modes in the range $[0.5, 1.5, \dots, 99.5]$ Naive metropolis will find this neighbourhood, on average in 100 steps. Using the Van der Corput sequence reduces this to 56 samples.

The slice sampling technique explored above, however, managed reductions to 10 and 20 steps respectively (depending on the likelihood function). Therefore I choose to focus on developing the slice sampling technique rather than further investigation Quasi-Monte Carlo methods.

Chapter 5

Related Work

5.1 Probabilistic Programming Languages

With the fast increase in both breadth and specialization of probabilistic models, the need for tools which make the modelling task easier has grown sharply. Many different probabilistic programming languages have arisen to fill this gap. At their core probabilistic programming languages are tools that formalise a model specification by providing the user with stochastic primitives with which to define their model. Given this formalisation the PPL can then attempt to automatically perform inference on the user's model. While the foundation for these stochastic languages was set a long time ago (eg: [14]), there has been a recent explosion in the popularity of PPLs. Probabilistic languages have now arisen to fill many different modelling niches by juggling tradeoffs such as efficiency, expressivity and ease of use.

It is beyond the scope of this work to go into details about all the various PPLs (see [1] for a short summary of some of the most well known PPLs), but we shall briefly list a few different paradigms, just to give an idea of the diversity of PPLs. The logic programming paradigm has attracted quite a few probabilistic extensions ([16, 4, 40, 34]) which are used for tasks such as symbolic-statistical modeling and statistical relational learning. Other PPLs adopt a functional programming paradigm ([10, 22, 45]), by extending LISP [2] which, being a higher-order language, gives the user a large degree of flexibility over his probabilistic program specification. Some methods for creating embedded, domain specific, PPLs have also emerged [17]. Finally, BUGS is an example of a PPL that does not extend a previous programming language, but instead defines a simple, bespoke, language that emphasizes usability, since it enables the user to

define their model in a declarative fashion [19].

Switching to another criteria, some PPLs focus on efficiency at the cost of expressivity, generally by restricting themselves to programs that can be compiled into finite graphical models [19, 37, 25, 41, 28]. Other PPLs emphasize flexibility and expressiveness ([27, 32, 33, 10]), by allowing the user to specify programs that cannot be expressed as graphical models and that therefore can't make use of the standard graphical model inference methods. Church and Venture are the two languages that perhaps focus most on flexibility from the above, which is what makes them well suited for complex, potentially recursive, reasoning tasks like the cognition models briefly mentioned in Section 1.2.

It is based on these criteria that we decided to compare two popular PPLs. Namely the efficient, declarative, OpenBUGS and the flexible, functional Venture.

5.2 Speeding up inference

As we saw in Section 1.2, one of the crucial problems that must be solved before PPLs can reach their promise is that of efficient, general inference. Many attempts have been made towards this goal. One possibility is to perform exact inference by applying dynamic programming (DP) techniques to manage the exponential number of possible execution paths [42]. There is potential for future work in this area by analysing the performance of different DP and approximate-DP techniques. More generally, it is clear that DP won't work on all models, but understanding what the subclass of models is on which exact inference might be tractable would be very useful to known, from a PPL designer's point of view.

When exact inference is intractable, we have to settle for approximate solutions, usually obtained via Markov chain Monte Carlo (MCMC). Figuring out how to best take advantage of the underlying structure of the distributions, as to obtain better mixing rates and therefore faster inference, is currently an area of active research. Two broad approaches here are to compile the probabilistic program into a form more amenable to efficient inference, or to implement novel inference techniques which can achieve better performance on some class of models.

In the program compilation category, one attempt uses nonstandard interpretations to create monad-like side computations which can extract structural information, such as gradients [43]. This information can then enable the use of sophisticated MCMC techniques, such as Hamiltonian MC, which can lead

to big boosts in performance over Metropolis-Hastings and other naive MCMC methods. There seems to be much potential in further improving inference through the application of compiler design and program analysis techniques. For instance, speed-ups of over an order of magnitude were shown to be possible by applying techniques such as JIT compilation, dead code elimination, allocation removal and incremental optimization [46]. Our partitioned prior idea also falls in the program compilation category and is, as far as we can tell, a novel program transformation.

Regarding the implementation of new inference methods, one recent attempt [45] looked at the application of particle MCMC towards inference on probabilistic programs. The authors implement particle MCMC in a Turing Complete functional language and show that their implementation outperforms Metropolis-Hastings on certain models while also being very easy to parallelize. Other inference methods which specialized to a small niche have also been proposed. For instance, in [47] the authors propose a method called “locally annealed reversible jump MCMC” which they show to be very efficient at the computer graphics problem of synthesizing open world layouts. Our implementation of slice sampling as a inference technique in PPLs has also shown to outperform the default Metropolis-Hastings algorithm on some models and, assuming the trans-dimensionality issue is solvable, should prove a valuable addition to the inference toolbelt available to PPL designers.

Chapter 6

Summary and Conclusions

6.1 Overview

We've performed an empirical evaluation of OpenBUGS and Venture on a few simple models and shown that OpenBUGS significantly outperforms Venture. This discrepancy in performance is partly attributable to the fact that OpenBUGS sacrifices some expressivity in exchange for efficiency. However there is no good reason to believe that Venture's flexibility has to come at such a high computational cost. In particular, OpenBUGS employs a large array of inference methods and has an expert system to decide which method to apply to which model. While Venture also employs different types of inference techniques, it seems that there is still progress to be made in getting high-level probabilistic programming languages to correctly apply less general but potentially more efficient algorithms in order to speed up inference on certain classes of models.

We've explored a pre-processing technique that aims to compile probabilistic programs into equivalent but more efficient versions of themselves. We show, that on some models, large performance gains can be made by this technique, but that the general case is more difficult to solve. We also present some negative results, detailing why some a priori plausible solutions either do not leave our programs invariant or do not have good empirical performance.

Towards improving inference performance in Turing Complete PPLs, we implemented such a language in python and made it available online. We then implemented a slice sampling inference engine in our PPL, thus exploring an inference technique which hadn't previously been used in the context of probabilistic programs. We show that slice sampling can lead to large speed-ups on

certain models and discuss the problem of making slice sampling work correctly on trans-dimensional probabilistic spaces. We also provide a negative result here, showing that one reasonable solution to the trans-dimensional slicing fails on some models (though it succeeds on others).

6.2 Future Directions

Both OpenBUGS and Venture are used on a large variety of models with varying qualities of model specification and varying user expectations. As such it would be interesting to conduct a more comprehensive comparison of the two languages, which would cover a much wider variety of models and inference requirements than we have done. Additionally, Venture offers us the tools to tweak its inference engine or even provide an external, bespoke, inference technique for parts of a model. As such it would be interesting to see what level of customization would need to be done to Venture’s inference engine before the performance it obtained on certain models was on par with OpenBUGS.

Our analysis of partitioned priors has clearly shown that this technique can significantly speed up inference on some models, but that it can also hinder performance on others. It would be useful to perform a thorough, theoretical or empirical, analysis trying to define the class of models in which partitioned priors would help. It would also be interesting to test the effect of the sum partition method on infinitely divisible prior distributions. Additional techniques for preventing stuck bits should also be investigated, such as the mixture over bit decompositions of different depths.

For slice sampling, the main problem is devising a trans-dimensional approach which is usable on arbitrary probabilistic programs, or proving that this is impossible. Assuming a trans-dimensional slicer exists, then it would be very interesting to determine the class of models on which slice sampling outperform metropolis-hastings. Our preliminary results seem to indicate that slice sampling is well suited for continuous, low-dimensional model. Versions of slice sampling which are designed to handle high-dimensional cases better should also be investigated. Finally, in order for stochastic python to be generally usable as a PPL, the interface should be made more user-friendly by automatically inferring the name that must be given to random variables, rather than having the user provide it.

Bibliography

- [1] Probabilistic programming wiki. <http://probabilistic-programming.org/>, 2014. (Cited on page 53.)
- [2] Harold Abelson, RK Dybvig, CT Haynes, GJ Rozas, NI Adams IV, DP Friedman, E Kohlbecker, GL Steele Jr, DH Bartley, R Halstead, et al. Revised report on the algorithmic language scheme. *ACM SIGPLAN Lisp Pointers*, 4(3):1–55, 1991. (Cited on page 53.)
- [3] DARPA. Probabilistic programming for advancing machine learning. http://www.darpa.mil/Our_Work/I20/Programs/Probabilistic_Programming_for_Advanced_Machine_Learning_%28PPAML%29.aspx, 2013. (Cited on page 1.)
- [4] Luc De Raedt and Kristian Kersting. *Probabilistic inductive logic programming*. Springer, 2008. (Cited on page 53.)
- [5] Michael C Frank and Noah D Goodman. Predicting pragmatic reasoning in language games. *Science*, 336(6084):998–998, 2012. (Cited on page 2.)
- [6] Tobias Gerstenberg and Noah D Goodman. Ping pong in church: Productive use of concepts in human probabilistic inference. In *Proceedings of the 34th annual conference of the cognitive science society*, 2012. (Cited on page 7.)
- [7] ND Goodman and JB Tenenbaum. *Probabilistic Models of Cognition*. (electronic), 2013. (Cited on page 7.)
- [8] Noah Goodman. Webchurch. <https://github.com/probmods/webchurch>. (Cited on page 12.)
- [9] Noah D Goodman. The principles and practice of probabilistic programming. In *ACM SIGPLAN Notices*, volume 48, pages 399–402. ACM, 2013. (Cited on page 2.)
- [10] Noah D Goodman, Vikash K Mansinghka, Daniel M Roy, Keith Bonawitz, and Joshua B Tenenbaum. Church: A language for generative models. In *In Uncertainty in Artificial Intelligence*, 2008. (Cited on pages 53 and 54.)
- [11] Noah D Goodman and Andreas Stuhlmüller. Knowledge and implicature: Modeling language understanding as social cognition. *Topics in cognitive science*, 5(1):173–184, 2013. (Cited on page 2.)

- [12] Peter J Green and David I Hastie. Reversible jump mcmc. *Genetics*, 155(3):1391–1403, 2009. (Cited on page 37.)
- [13] W Keith Hastings. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57(1):97–109, 1970. (Cited on page 16.)
- [14] Claire Jones and Gordon D Plotkin. A probabilistic powerdomain of evaluations. In *Logic in Computer Science, 1989. LICS'89, Proceedings., Fourth Annual Symposium on*, pages 186–195. IEEE, 1989. (Cited on page 53.)
- [15] Jonathan Keith, George Sofronov, and Dirk Kroese. The generalized gibbs sampler and the neighborhood sampler. In *Monte Carlo and Quasi-Monte Carlo Methods 2006*, pages 537–547. Springer, 2008. (Cited on page 11.)
- [16] Angelika Kimmig, Bart Demoen, Luc De Raedt, Vitor Santos Costa, and Ricardo Rocha. On the implementation of the probabilistic logic programming language problog. *Theory and Practice of Logic Programming*, 11(2-3):235–262, 2011. (Cited on page 53.)
- [17] Oleg Kiselyov and Chung-chieh Shan. Embedded probabilistic programming. In *Domain-Specific Languages*, pages 360–384. Springer, 2009. (Cited on page 53.)
- [18] Solomon Kullback and Richard A Leibler. On information and sufficiency. *The Annals of Mathematical Statistics*, pages 79–86, 1951. (Cited on page 16.)
- [19] David Lunn, David Spiegelhalter, Andrew Thomas, and Nicky Best. The bugs project: Evolution, critique and future directions. *Statistics in medicine*, 28(25):3049–3067, 2009. (Cited on pages 6 and 54.)
- [20] 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. (Cited on page 11.)
- [21] David JC MacKay. *Information theory, inference and learning algorithms*. Cambridge university press, 2003. (Cited on pages 17, 29, and 37.)
- [22] Vikash Mansinghka, Daniel Selsam, and Yura Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv preprint arXiv:1404.0099*, 2014. (Cited on pages 2, 6, and 53.)
- [23] Vikash Kumar Mansinghka. *Natively probabilistic computation*. PhD thesis, Massachusetts Institute of Technology, 2009. (Cited on page 7.)
- [24] Frank J Massey Jr. The kolmogorov-smirnov test for goodness of fit. *Journal of the American statistical Association*, 46(253):68–78, 1951. (Cited on page 16.)
- [25] Andrew McCallum, Karl Schultz, and Sameer Singh. Factorie: Probabilistic programming via imperatively defined factor graphs. In *NIPS*, pages 1249–1257, 2009. (Cited on page 54.)
- [26] Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087–1092, 1953. (Cited on page 16.)

- [27] Brian Milch, Bhaskara Marthi, Stuart Russell, David Sontag, Daniel L Ong, and Andrey Kolobov. Blog: Probabilistic models with unknown objects. *Statistical relational learning*, page 373, 2007. (Cited on page 54.)
- [28] Tom Minka, John Winn, John Guiver, and David Knowles. Infer .net 2.4, 2010. microsoft research cambridge. <http://research.microsoft.com/infernet/>. (Cited on page 54.)
- [29] Radford M Neal. Probabilistic inference using markov chain monte carlo methods. Technical report, Department of Computer Science, University of Toronto Toronto, Ontario, Canada, 1993. (Cited on pages 16 and 17.)
- [30] Radford M Neal. Slice sampling. *Annals of statistics*, pages 705–741, 2003. (Cited on pages 35 and 37.)
- [31] Thomas Neal. Openbugs overview. <http://www.openbugs.net/w/Overview>, 2010. (Cited on page 6.)
- [32] Avi Pfeffer. Ibal: A probabilistic rational programming language. In *IJCAI*, pages 733–740. Citeseer, 2001. (Cited on page 54.)
- [33] Avi Pfeffer. Figaro: An object-oriented probabilistic programming language. *Charles River Analytics Technical Report*, page 137, 2009. (Cited on page 54.)
- [34] David Poole. The independent choice logic and beyond. In *Probabilistic inductive logic programming*, pages 222–243. Springer, 2008. (Cited on page 53.)
- [35] Razvan Ranca. Stochastic python. <https://github.com/RazvanRanca/StocPy>, 2014. (Cited on pages 3 and 37.)
- [36] OpenBUGS Model Repository. Tdf models. <http://www.openbugs.net/Examples/t-df.html>. (Cited on page 8.)
- [37] Matthew Richardson and Pedro Domingos. Markov logic networks. *Machine learning*, 62(1-2):107–136, 2006. (Cited on page 54.)
- [38] Christian P Robert and George Casella. *Monte Carlo statistical methods*, volume 319. Citeseer, 2004. (Cited on page 51.)
- [39] DM Roy, VK Mansinghka, ND Goodman, and JB Tenenbaum. A stochastic programming perspective on nonparametric bayes. In *Nonparametric Bayesian Workshop, Int. Conf. on Machine Learning*, volume 22, page 26, 2008. (Cited on page 6.)
- [40] Taisuke Sato and Yoshitaka Kameya. Prism: a language for symbolic-statistical modeling. In *IJCAI*, volume 97, pages 1330–1339. Citeseer, 1997. (Cited on page 53.)
- [41] Stan Development Team. Stan: A c++ library for probability and sampling, version 2.2. <http://mc-stan.org/>, 2014. (Cited on page 54.)
- [42] Andreas Stuhlmüller and Noah D Goodman. A dynamic programming algorithm for inference in recursive probabilistic programs. *Statistical Relational AI workshop at UAI. 2012.*, 2012. (Cited on pages 2 and 54.)

- [43] David Wingate, Noah D Goodman, Andreas Stuhlmüller, and Jeffrey Mark Siskind. Nonstandard interpretations of probabilistic programs for efficient inference. In *NIPS*, pages 1152–1160, 2011. (Cited on pages 2 and 54.)
- [44] David Wingate, Andreas Stuhlmüller, and Noah D Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *International Conference on Artificial Intelligence and Statistics*, pages 770–778, 2011. (Cited on pages 37, 38, and 41.)
- [45] 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*, 2014. (Cited on pages 2, 46, 53, and 55.)
- [46] Lingfeng Yang, P Hanrahan, and N Goodman. Incrementalizing mcmc on probabilistic programs through tracing and slicing. *Under review*, 2013. (Cited on pages 2 and 55.)
- [47] Yi-Ting Yeh, Lingfeng Yang, Matthew Watson, Noah D Goodman, and Pat Hanrahan. Synthesizing open worlds with constraints using locally annealed reversible jump mcmc. *ACM Transactions on Graphics (TOG)*, 31(4):56, 2012. (Cited on pages 2 and 55.)