Modelling choices and their effects on inference performance in probabilistic programming languages

Razvan Ranca
Queens' College

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 5, 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: 14,235

Signed:

Date:

Abstract

This is the abstract. Write a summary of the whole thing. Make sure it fits in one page.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This is the introduction where you should introduce your work. In general the thing to aim for here is to describe a little bit of the context for your work — why did you do it (motivation), what was the hoped-for outcome (aims) — as well as trying to give a brief overview of what you actually did.

It's often useful to bring forward some "highlights" into this chapter (e.g. some particularly compelling results, or a particularly interesting finding).

It's also traditional to give an outline of the rest of the document, although without care this can appear formulaic and tedious. Your call.

1.1 Background

A more extensive coverage of what's required to understand your work. In general you should assume the reader has a good undergraduate degree in computer science, but is not necessarily an expert in the particular area you've been working on. Hence this chapter may need to summarize some "text book" material.

This is not something you'd normally require in an academic paper, and it may not be appropriate for your particular circumstances. Indeed, in some cases it's possible to cover all of the "background" material either in the introduction or at appropriate places in the rest of the dissertation.

Chapter 2

Related Work

This chapter covers relevant (and typically, recent) research which you build upon (or improve upon). There are two complementary goals for this chapter:

1. to show that you know and understand the state of the art; and
2. to put your work in context

Ideally you can tackle both together by providing a critique of related work, and describing what is insufficient (and how you do better!)

The related work chapter should usually come either near the front or near the back of the dissertation. The advantage of the former is that you get to build the argument for why your work is important before presenting your solution(s) in later chapters; the advantage of the latter is that don't have to forward reference to your solution too much. The correct choice will depend on what you're writing up, and your own personal preference.

Chapter 3

Comparing Venture and OpenBUGS

In this chapter I 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.

3.1 Motivation

A lot of current research in probabilistic programming is focused on achieving more efficient automatic inference on different types of models. 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, to employing general inference techniques on models transformed by the application of optimization techniques similar to those used in compiler architecture. These distinct approaches have lead to the development of probabilistic programming languages (and implementations of these languages) which differ in significant ways. At the moment, the relative benefits and drawbacks of these languages on different classes of models are not very well understood.

In this chapter I attempt to take a step towards better understanding the relative benefits and drawbacks these languages by looking at two PPLs which fall at different ends of the specialization/generality spectrum. I 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 reveal where future work should focus in order to alleviate these problems.

Add background info on Venture and OpenBUGS

3.2 Preliminaries

3.2.1 Webchurch's performance

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

I 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. I therefore chose to focus only on Venture and OpenBUGS.

3.2.2 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 3.1: 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.

3.3 Empirical results

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

Maybe also evaluate one model that's not from the OpenBUGS repository, since BUGS might be unreasonably optimized on its own models.

3.3.1 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 distrobution of the degrees of freedom parameter (called d). In the course discrete version d is drawn from the uniform discrete distribution between 2 and 50. In the fine discrete distribution, d is drawn uniformly from the set: {2.0, 2.1, 2.2, . . . 6.0}. Finally, in the continuous version d is drawn from the continuous uniform distribution between 2 and 100. The models' OpenBUGS implementations and results can be found in the OpenBUGS model repository at: `http://www.openbugs.net/Examples/t-df.html`

add model pseudocode

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 3.1, 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, I 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 $1000 + 1000 * 100 = 101,000$.

3.3.2 Tdf model true posteriors

Given the simplicity of the Tdf models, we can calculate the true posteriors analytically.

Missing figure

Add formula used to derive posterior, as in the log

Using this formulation we calculate the true posteriors for all 3 Tdf model and get

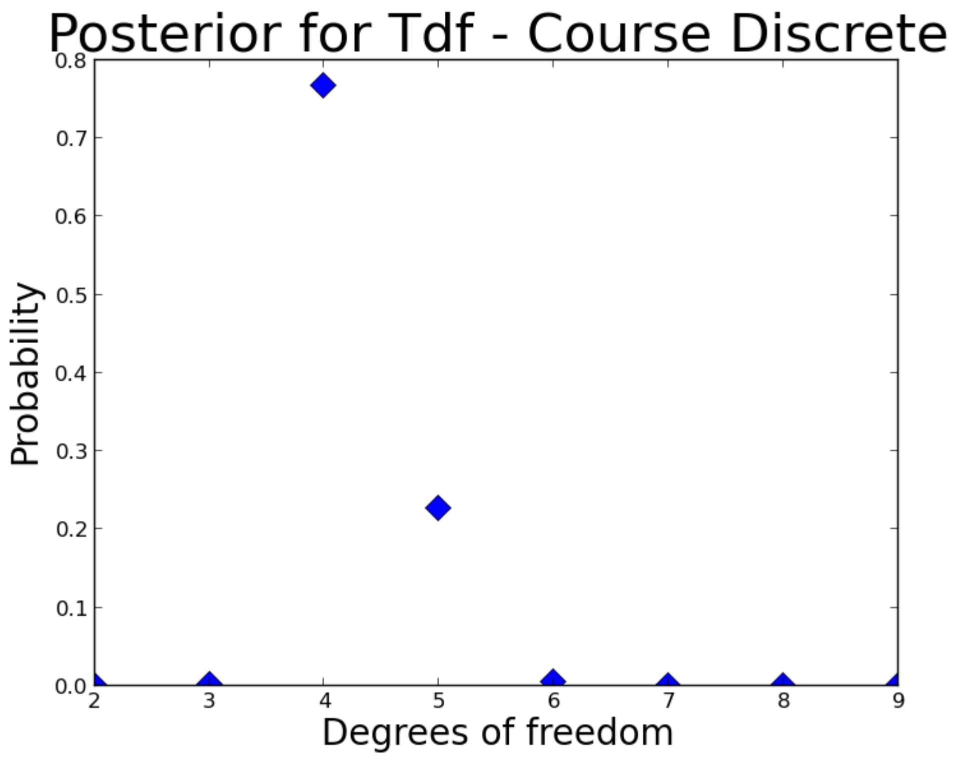


Figure 3.1: True posterior distribution for Tdf Course Discrete model.

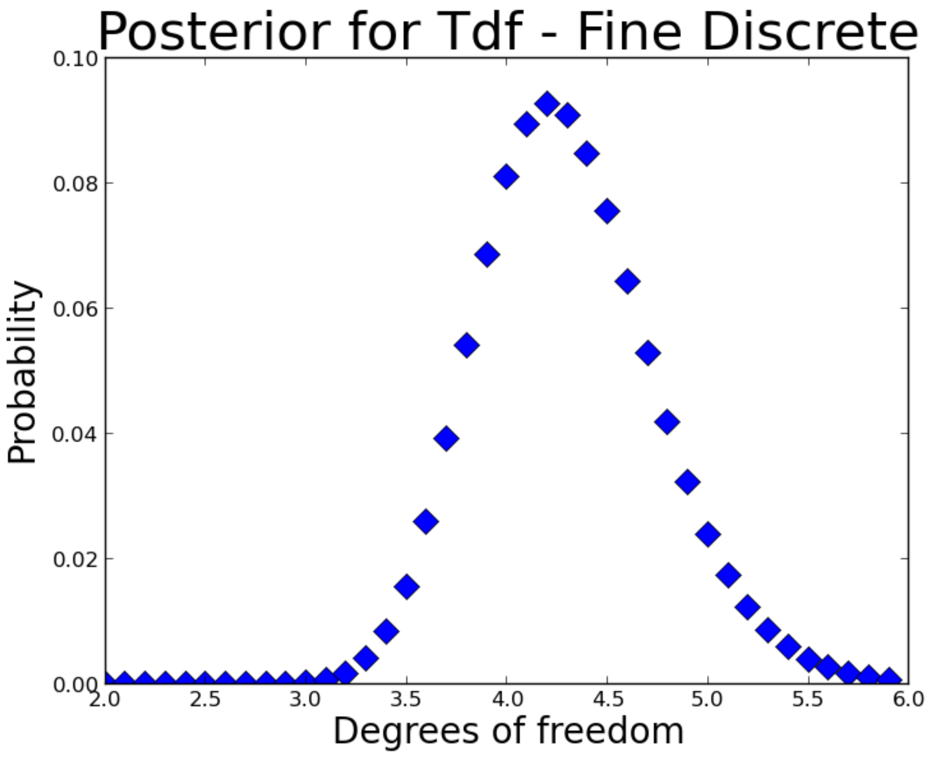


Figure 3.2: True posterior distribution for Tdf Fine Discrete model.

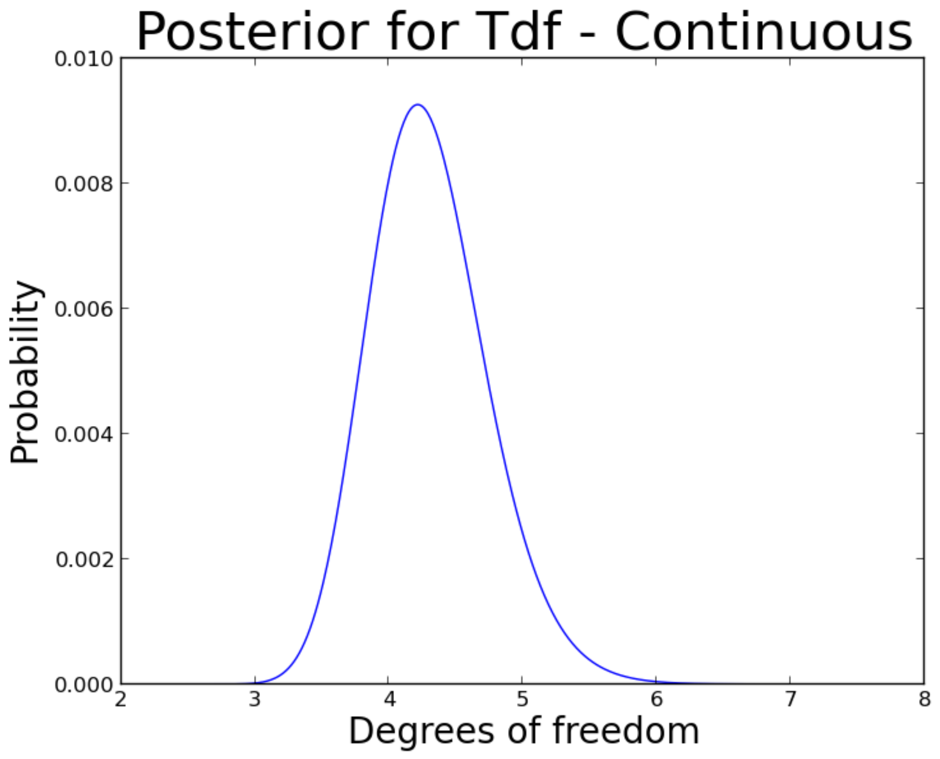


Figure 3.3: True posterior distribution for Tdf Continuous model.

3.3.3 Tdf model results

Add KL and KS difference results for these distributions

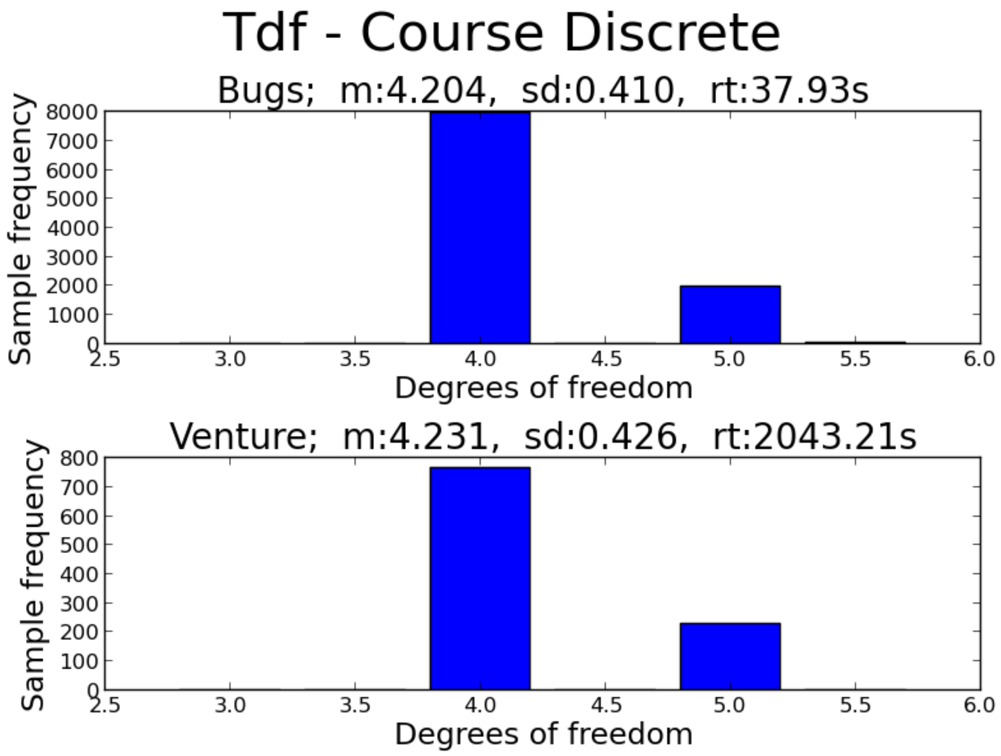


Figure 3.4: Distributions obtained by the two PPLs on the course discrete model.

On the course discrete model 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.

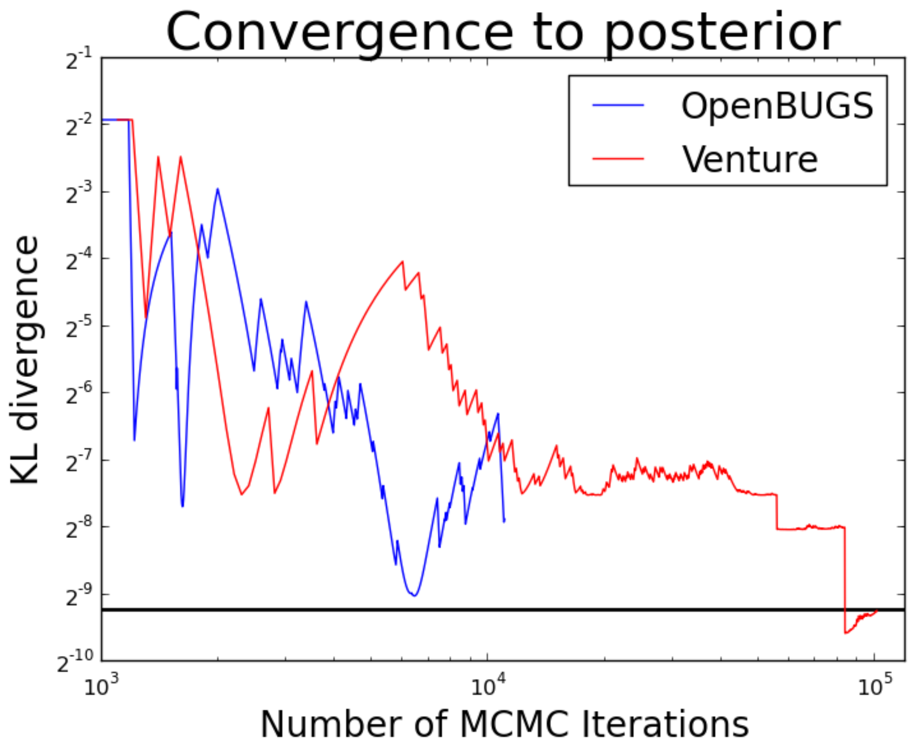


Figure 3.5: Rate of convergence of the two engines as the number of samples increases (first 1,000 samples are discarded as burn-in). The black line shows the KL divergence achieved by Venture after 101,000 inference steps.

Due to the course discrete prior, the convergence rate here is quite choppy and no significant conclusions can be drawn from this single run.

Maybe do more runs and plot quartiles

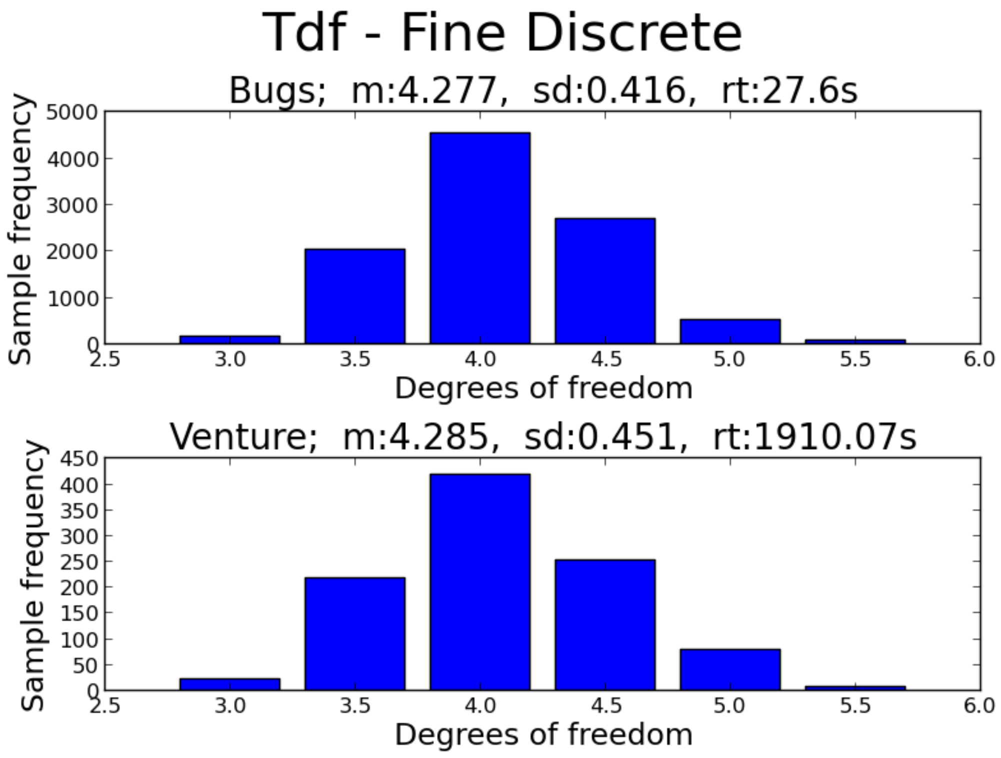


Figure 3.6: Distributions obtained by the two PPLs on the fine discrete model.

As with the course discrete case, the two engines obtain similar looking distributions, but the runtime gap actually worsens. Specifically, Venture now has a runtime ∼69 times larger than OpenBUGS.

Looking again at the convergence rate shows that Venture also does a worse job of inferring the true posterior, despite the longer runtime.

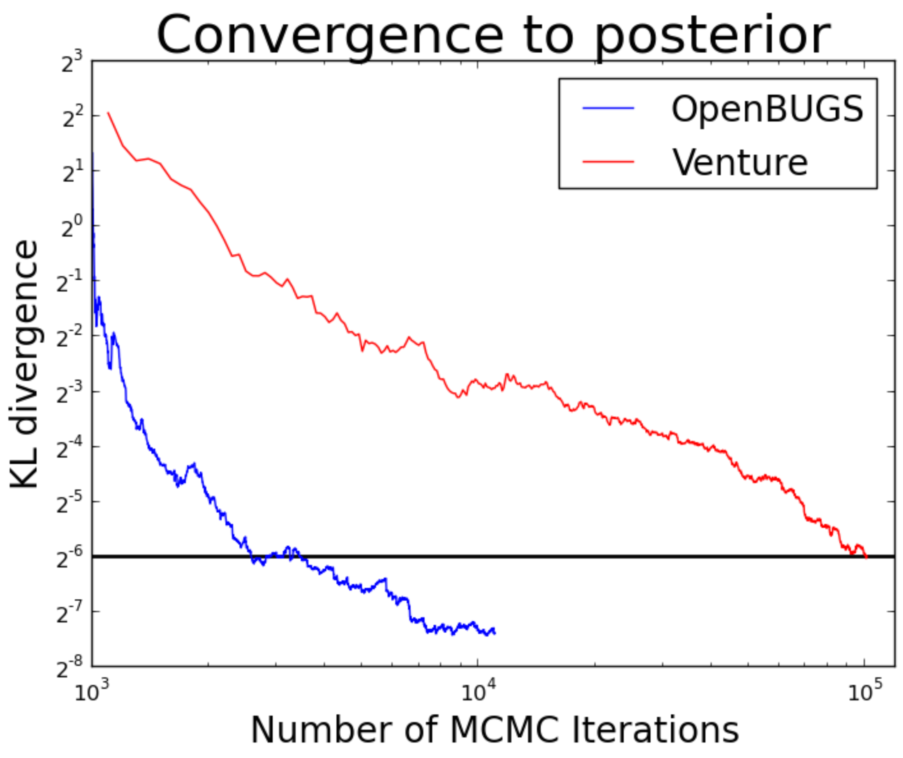


Figure 3.7: Rate of convergence of the two engines as the number of samples increases (first 1,000 samples are discarded as burn-in). The black line shows the KL divergence achieved by Venture after 101,000 inference steps.

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.

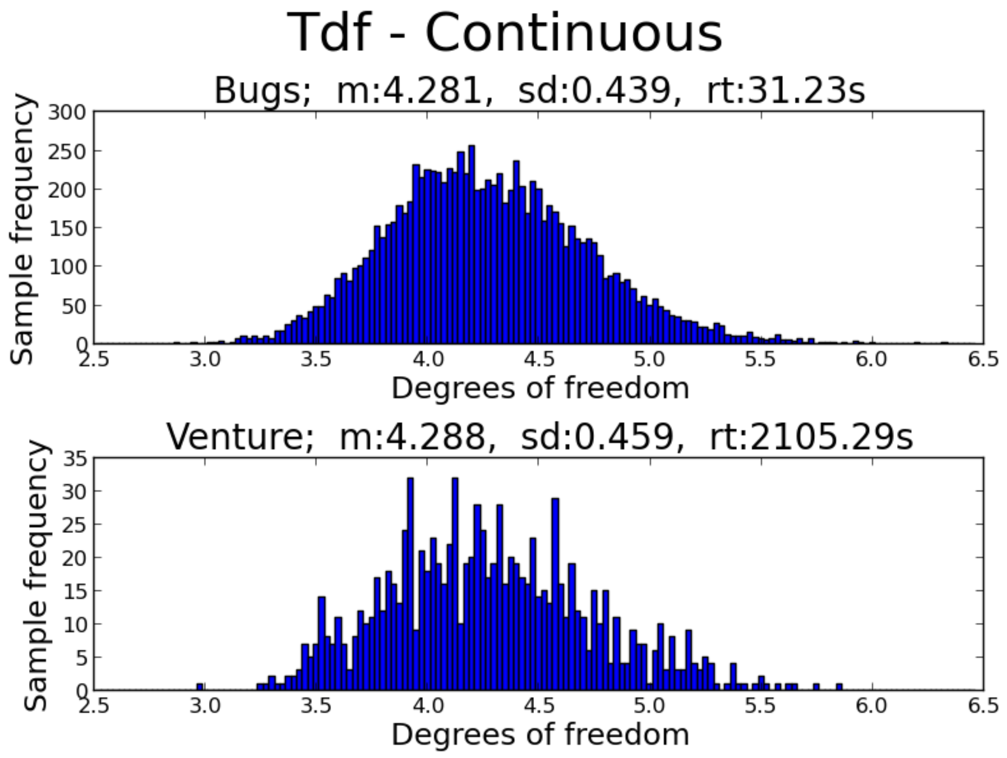


Figure 3.8: Distributions obtained by the two PPLs on the continuous model.

On the continuous case, 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.

To get a better idea of the PPL's performance we can once more look at their convergence rates.

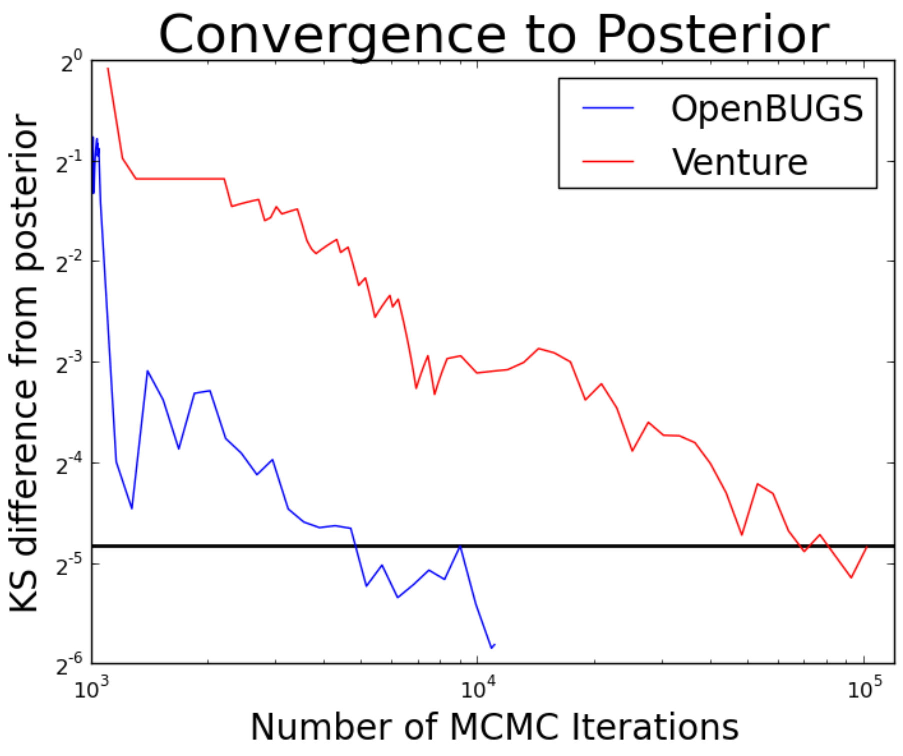


Figure 3.9: Rate of convergence of the two engines as the number of samples increases (first 1,000 samples are discarded as burn-in). The black line shows the KS difference achieved by Venture after 101,000 inference steps.

As with the fine discrete model, there seems to be a large qualitative gap between the performance of the two engines on this model. 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 =∼ 127 times slower than OpenBUGS on this model.

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

3.4 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.

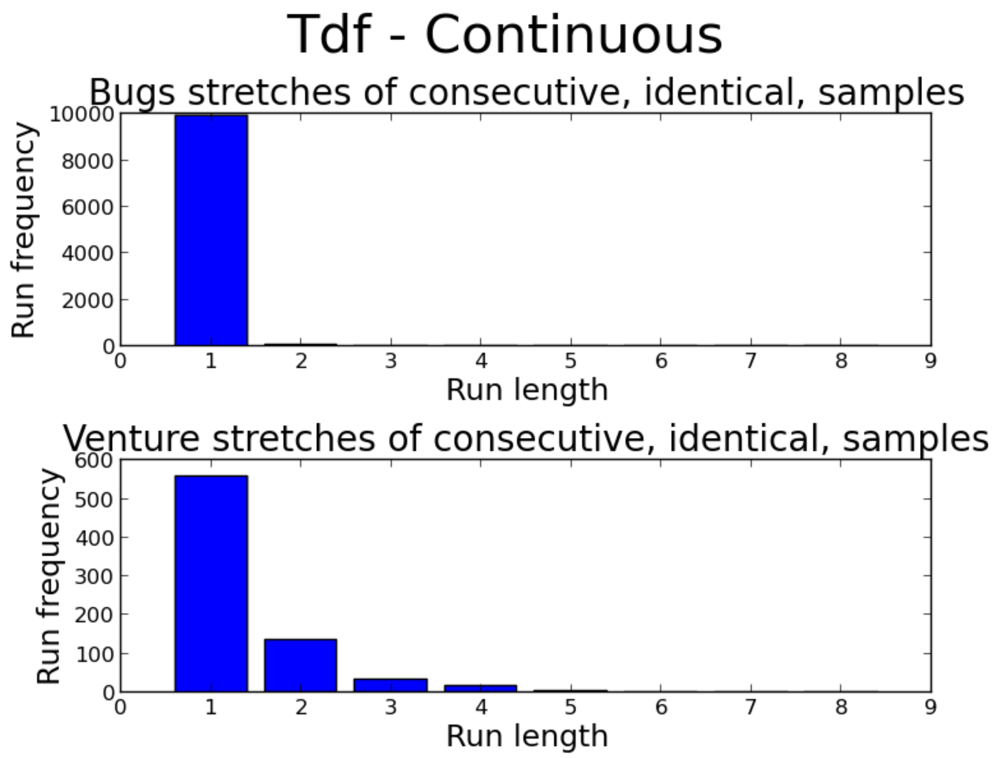


Figure 3.10: Number and size of identical sample runs generated by the two PPLs on the continuous model.

Figure 3.10 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 . We take a closer look at these different inference techniques and their performance in Chapter ??.

saw this mentioned in a paper, need to find direct source or remove

Decide is there's anything worth reporting from the performance tests done on Venture. Log dates: 2014.02.19 and 2014.02.20

Chapter 4

Partitioned Priors

One way in which we may try to improve the performance of local, single-site, Metropolis-Hastings is by paritioning the 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 convergence to the mode and the mixing properties of simple Metropolis-Hastings.

say something about proposal kernels vs. sampling from prior?

Based on this idea, it may be possible to design a light pre-inference rewriting of a probabilistic program which splits up its priors so that the subsequent inference step is performed more efficiently.

In this chapter 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 lead to speed ups on other distributions, assuming certain desirable properties, such that small changes in the uniform bits correspond to small changes in the final distribution.

not sure if this makes much sense. Need to research it more or remove

For simplicity, we also focus on models with a uni-modal posterior distribution. The analysis for multi-modal distributions would follow similar lines to the one presented here, but would also have to additionally account for mode-switching. We will look closer at this problem in Chapter ??.

4.1 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.

4.1.1 Local Metropolis-Hastings

add basic explanation

4.1.2 Expected number of iterations to a neighbourhood of the mode

The first test of the efficiency of a partition which we propose 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 neifghbourhood. Using the unpartitioned prior (uniform-continuous a b), we end up sampling from the uniform distribution and accepting or rejecting those samples according to the Metroplis-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 make use of empirical tests.

4.1.3 Mixing properties around the mode

Once the markov chain has reached the mode we will wish to see how well it manages to mix around it. Here we will 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 longs stretch. A numerical estimate of this can be obtained by measuring the "distance" traveled by the merkov chain around the mode (i.e. the sum of absolute differences between consecutive samples). We can also inspect the sample autocorrelation, with the idea that good mixing properties should imply a small autocorrelation within a sample run.

4.2 Sum of uniforms

We first consider partitioning the uniform prior into a sum of uniforms. This choice is made for simplicity and in order to observe some basic properties concerning local Metropolis-Hasting'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 4.1). The invariance is due to the uniform distribution not being infinitely divisible. The approach presented here could be safely used on other distributions, such as Gammas and Gaussians, which are infinitely divisible. In Section **??**

we will present a partitioning technique which does leave the uniform prior unchanged.

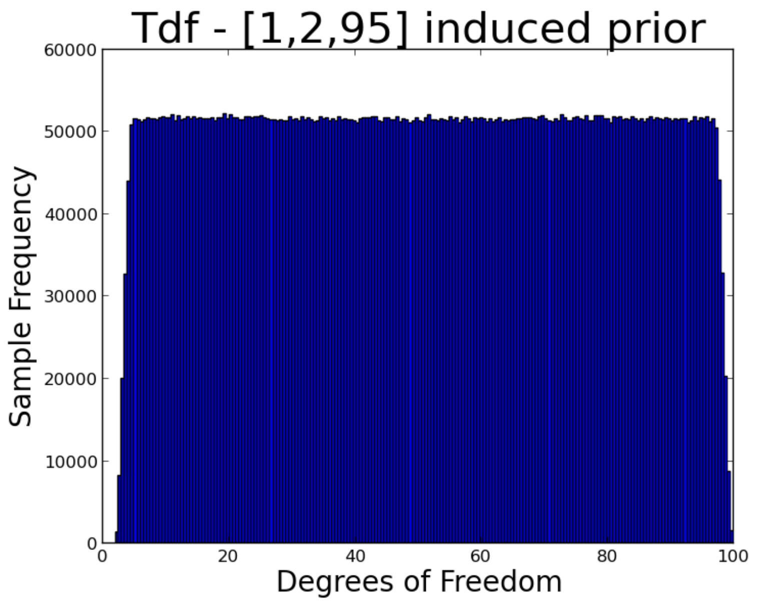


Figure 4.1: Prior distribution induced by partitioning the prior (uniform-continuous 2 100) into (uniform-continuous 2 95) + (uniform-continuous 0 2) (uniform-continuous 0 1).

4.2.1 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 4.1.2, the expected number of steps to $[mode - \epsilon, mode + \epsilon]$ of the mode can be analytically computed for the unpartitioned prior (uniform-continuous a b) as being $(b - a)/(2\epsilon)$.

For the partitioned priors, we instead perform empirical tests. These tests measure how many samples it takes for different partitions to reach neighbourhoods of the mode of different sizes. One thousand runs are done for each partition and neighbourhoods of 0.5, 0.25, 0.1 and 0.01 are considered. The partitions consited 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]. A final component value is added such that the sum of all components is the uniform prior specified in the Tdf model, namely (uniform-continuous 2 100). 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 $x + y + z = 98$.

Table 4.3 contains the empirical performance for the unpartitioned prior and for the best 2 partitions on each neighbourhood size.

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

The best partition seems to depend somewhat on the size of the neighbourhood, but there also seem to be 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.

It also seems that, as epsilon gets smaller, it's useful to have smaller partition components, such that we get the 0.5 partition in the best solution for a mode neighbourhood of size 0.02, while this partition size makes no appearance in the top partition lists for other neighbourhood sizes.

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 can consider what happens after the markov chain reaches the mode. Specifically, we set the initial sample to the mode of the true posterior distribution and check how much the chain moves over the next 1000 samples. Repeating this test 100 times and averaging the sum of jumps gives us the results in Table **??**.

The format is (partitions, average distance travelled, variance)

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

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. The large difference between the unpartitioned and the partitioned variants do suggest that there is some improvement here though. The mixing reate is investigates more thoroughly in Section **??**

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.

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

Table 4.3 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 and autocorrelation plots for two runs obtained with an unpartitioned and a (1,2,95) partitioned prior.

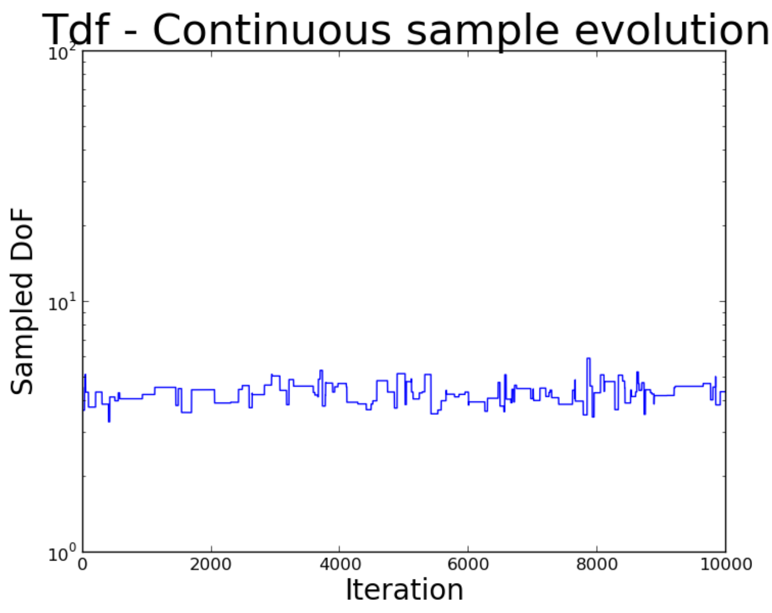


Figure 4.2: The sample evolution, over 10,000 samples, for the unpartitioned prior.

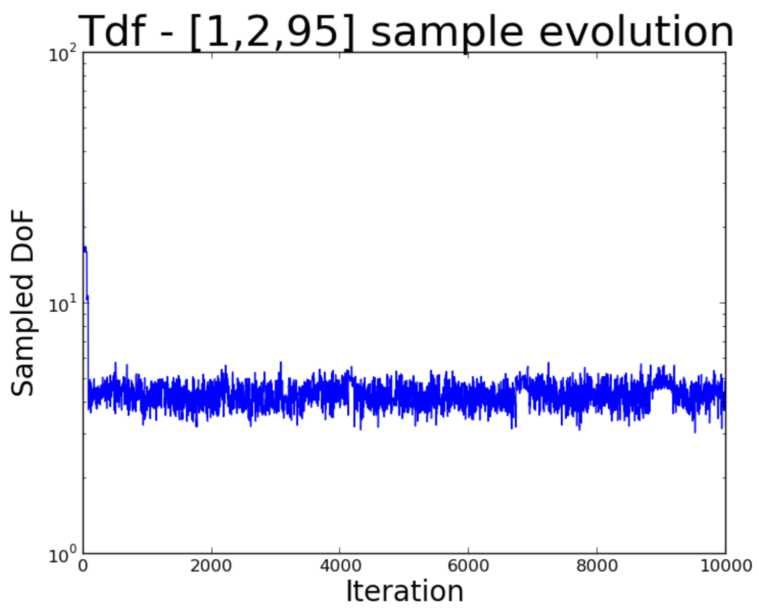


Figure 4.3: The sample evolution, over 10,000 samples, for the (1, 2, 95) partitioned prior.

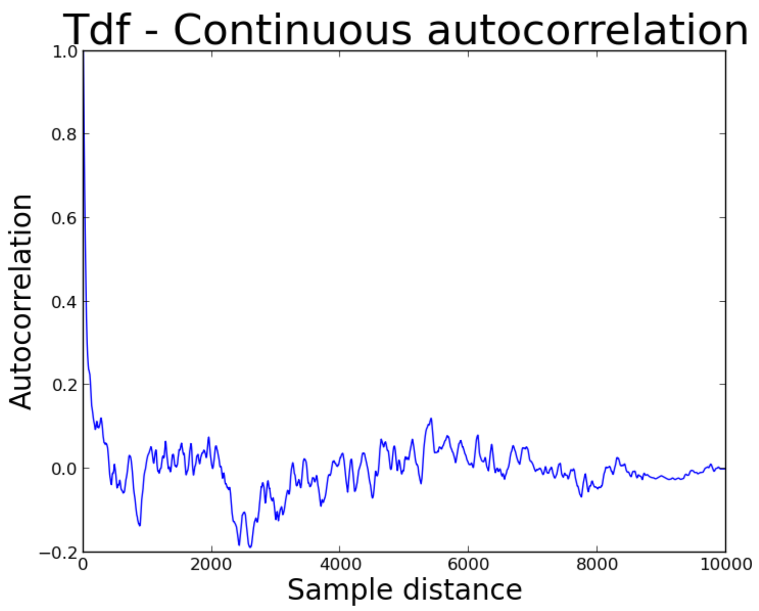


Figure 4.4: The autocorrelation, over 10,000 samples, for the unpartitioned prior.

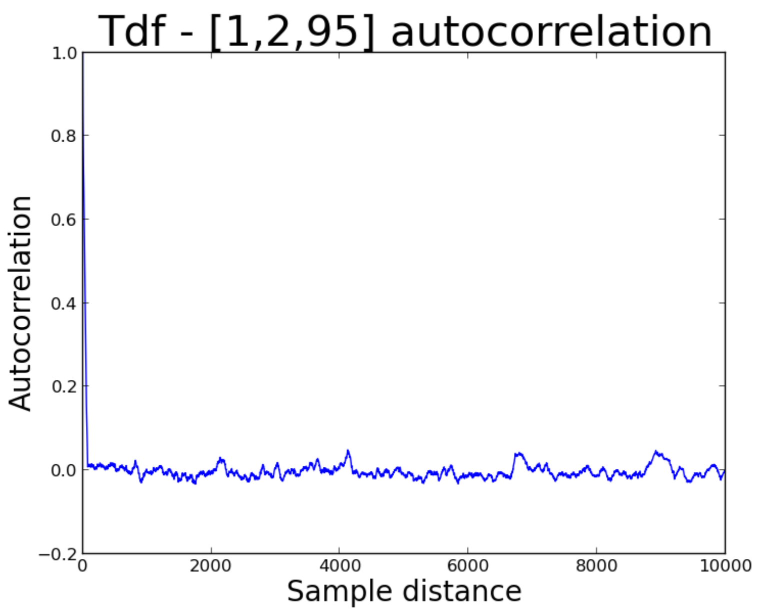


Figure 4.5: The autocorrelation, over 10,000 samples, for the (1, 2, 95) partitioned prior.

These graphs confirm our preliminary results from Section **??**, and show that the partitioned prior does help with mixing around the mode and with eliminating large correlations between consecutive samples.

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.

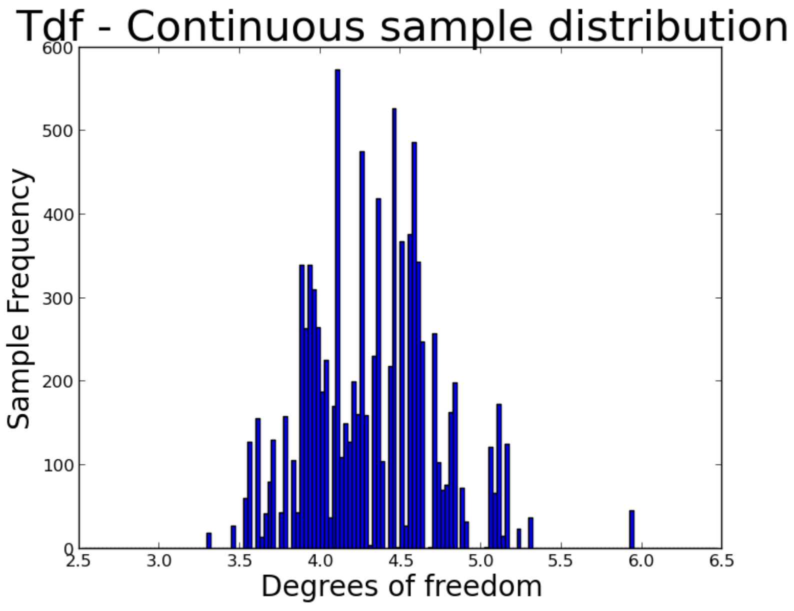


Figure 4.6: The sample distribution obtained from 10,000 samples using an unpartitioned prior.

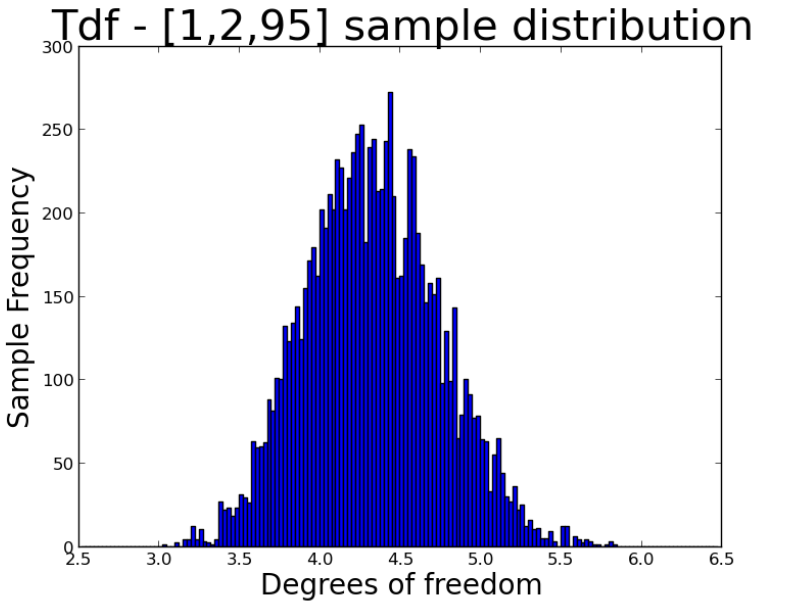


Figure 4.7: The sample distribution obtained from 10,000 samples using an (1, 2, 95) partitioned prior.

The true posterior for the Tdf continuous model was given in Section **??**. Looking at these it is quite clear the partitioned prior outperforms the origina, 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.

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 4.8.

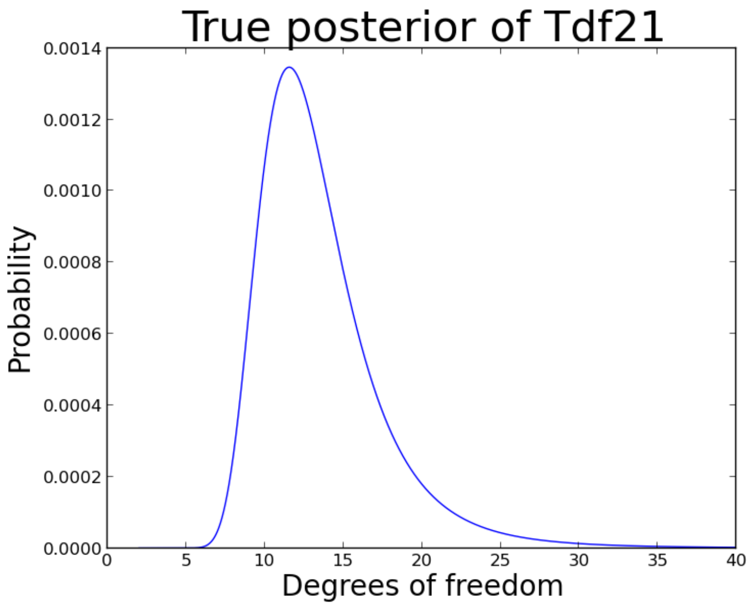


Figure 4.8: The true posterior of the Tdf21 model

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.

In order to better understand how the Metropolis-Hastings algorithm will be affected by this change we can also look at the log-likelihoods induced by the original Tdf Continuous model and by the Tdf21 model.

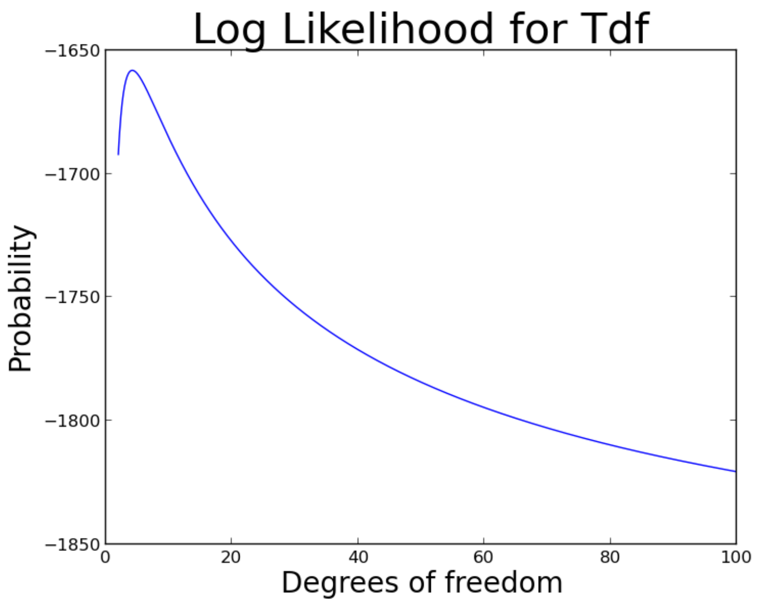


Figure 4.9: The log-likelihood of the Tdf Continuous model.

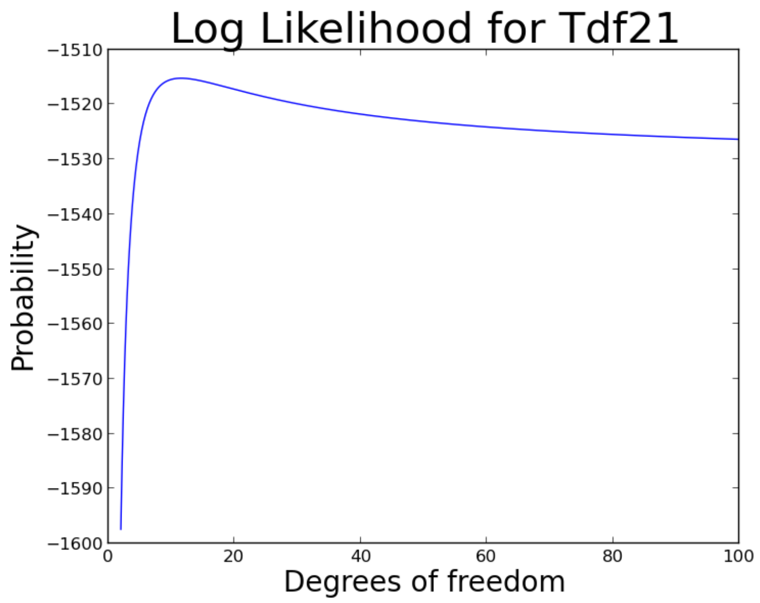


Figure 4.10: The log-likelihood of the Tdf21 model.

As can be seen, 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.

Evaluating the decomposition on the Tdf21 model

On this new model posterior we can now test the convergence of the priors by once again 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)

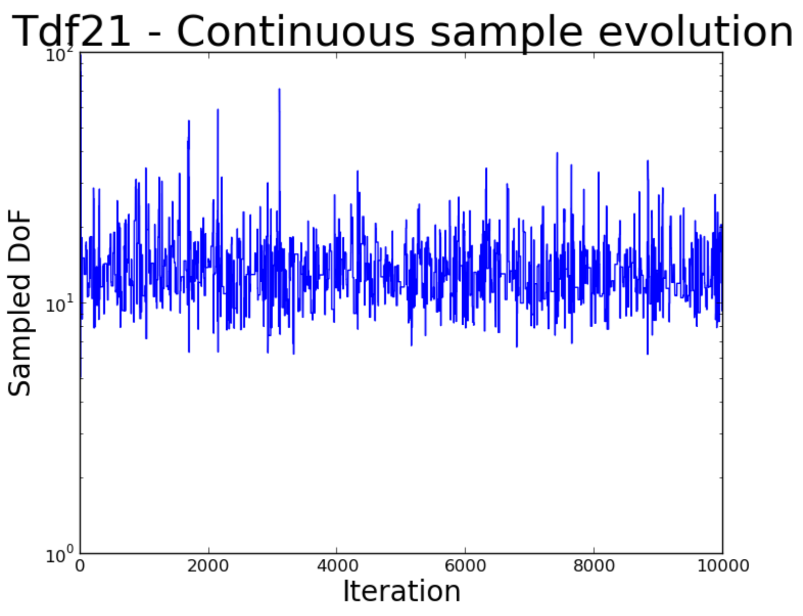


Figure 4.11: The sample evolution, on the Tdf21 model for, the unpartioned prior.

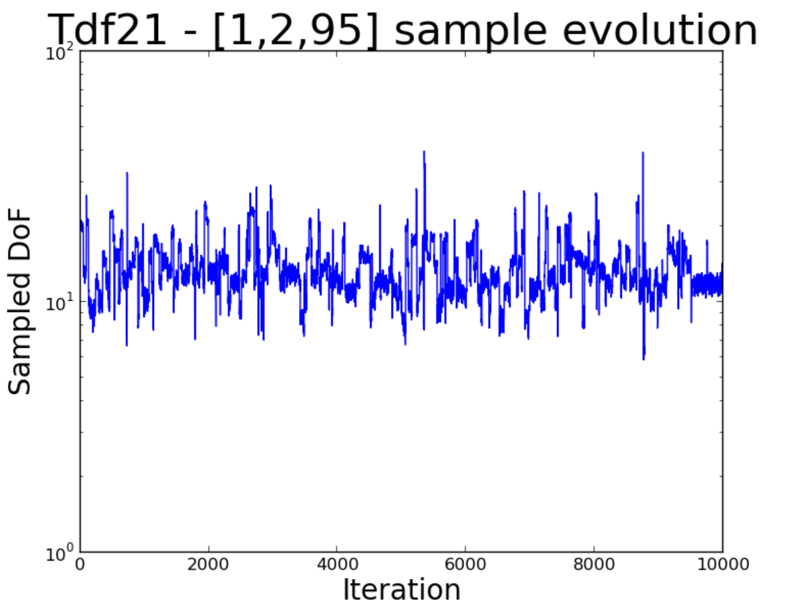


Figure 4.12: The sample evolution, on the Tdf21 model for, for the (1, 2, 95) partitioned prior.

From the sample evolutions 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.

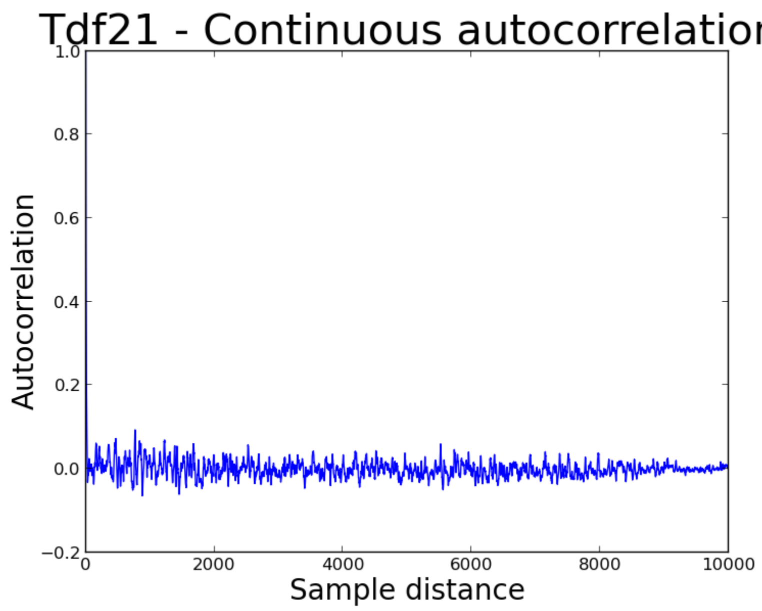


Figure 4.13: The autocorrelation, on the Tdf21 model for, for the unpartitioned prior.

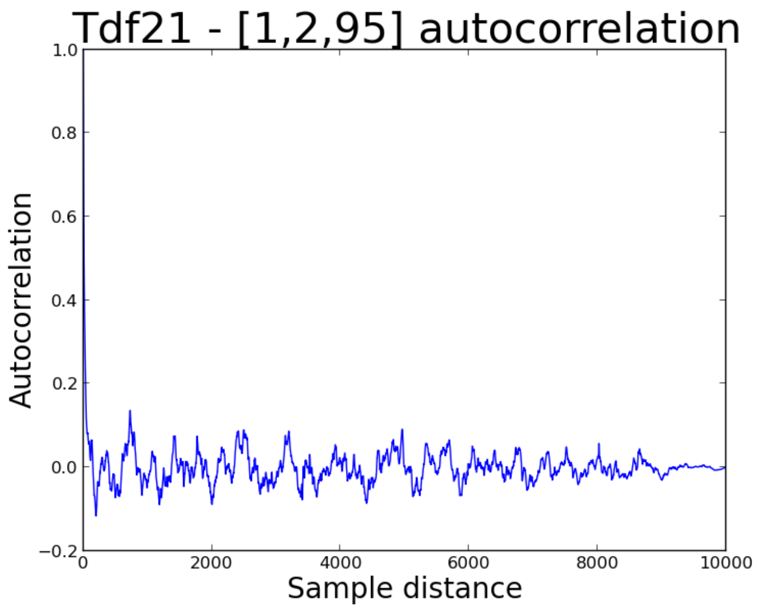


Figure 4.14: The autocorrelation, on the Tdf21 model for, for the (1, 2, 95) partitioned prior.

The autocorrelation plots 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.

The final test has to again be the actual sample distributions.

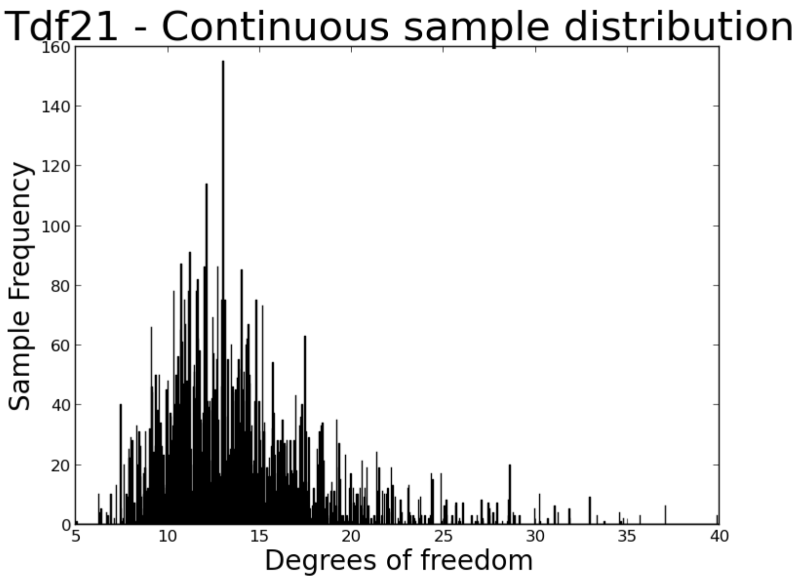


Figure 4.15: The sample distribution obtained from 10,000 samples using an unpartitioned prior on the Tdf21 model.

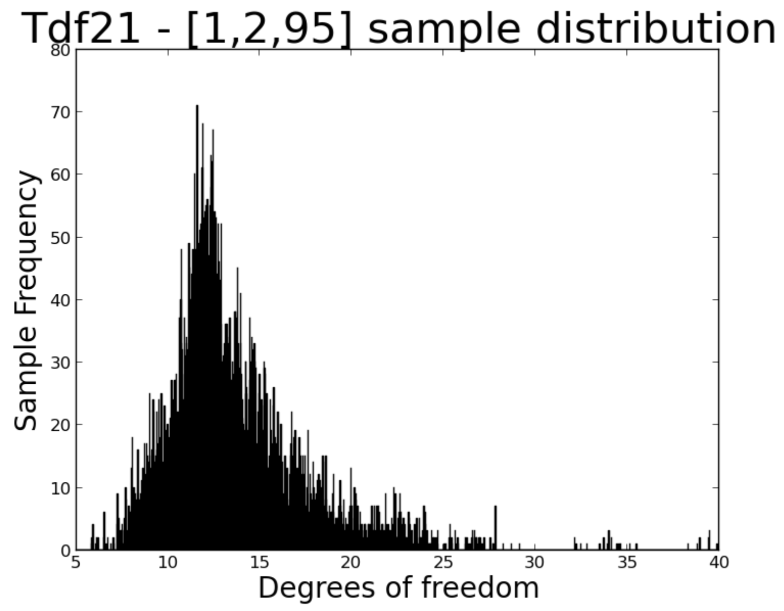


Figure 4.16: The sample distribution obtained from 10,000 samples using an (1, 2, 95) partitioned prior on the Tdf21 model.

Here we can see that the partitioned prior still results in a smoother distribution that does a better job of representing the true posterior.

4.3 Bit decomposition

As mentioned in Section **??**, one problem with the sum of uniforms decomposition is that it alters the shape of the prior. We would like to come up with decompositions that leave the prior invariant so that, since such decompositions could be applied indescriminately to re-write any probabilistic program. A family of invariant partitions of an uniform prior can be constructed by considering the bit representation of the uniform samples up to a certain depth and then adding a single uniform-continuous value of the correct size.

4.3.1 Definition

In order to partition any uniform interval (uniform-continuous a b) it is sufficient to be able to partition the interval (uniform-continuous 0 1). Once this is accomplished, the target interval can be obtained as (uniform-continuous a b) = a + (b-a) * (partitioned(uniform-continuous 0 1)).

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

4.3.2 Evaluation on Tdf and Tdf21

To get an idea of the properties of this decomposition we perform an empirical evaluation on the Tdf continuous model and on the Tdf21 model.

Looking at mixing rates ?? shows that a 3rd degree bit decomposition obtains similar performance to the 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 smaller the probability that one of the leading bits are picked and therefore the better we will expect the mixing to be. We, however, restrict ourselves to a 3rd degree decomposition here since a higher degree variant would exhibit other problems, which are discussed in Section ??.

Should be possible to give some more formal results here.

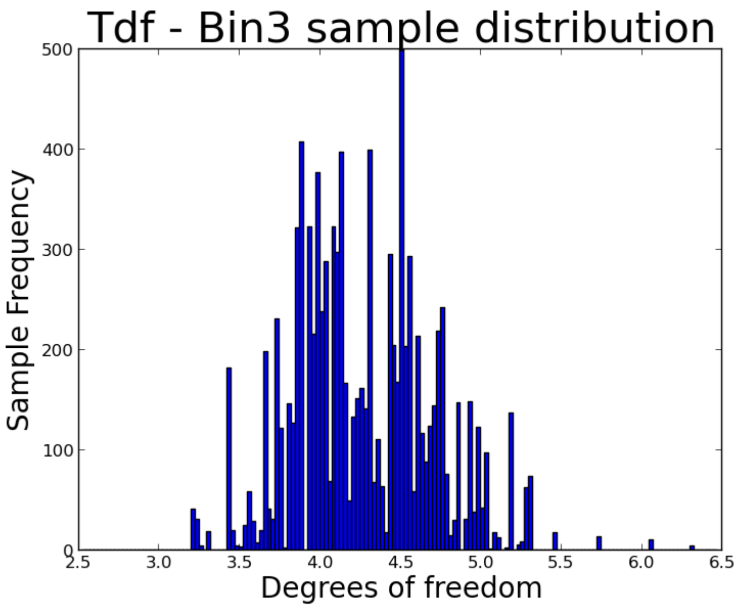


Figure 4.17: The sample distribution obtained from 10,000 samples using a 3rd degree bit decomposition on the Tdf model.

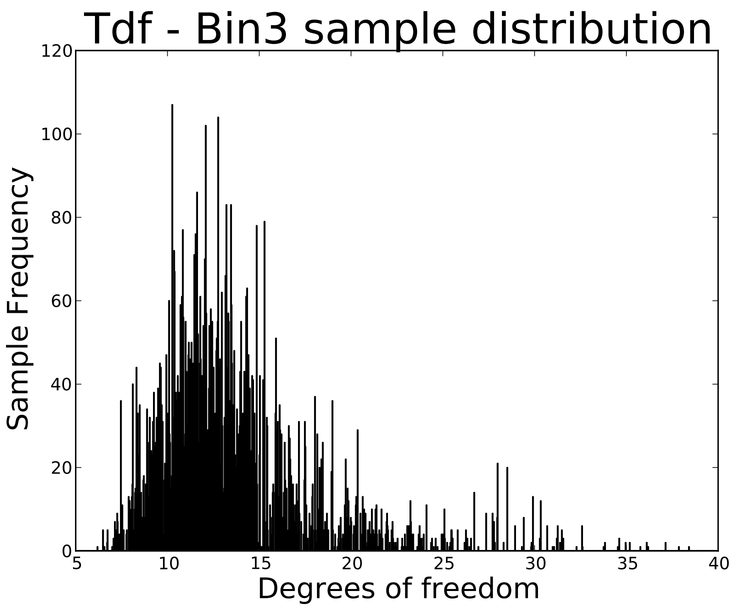


Figure 4.18: The sample distribution obtained from 10,000 samples using a 3rd degree bit decomposition on the Tdf21 model.

talk about how binomials of different depth perform here

Turning to the expected time to reach (see Table **??**) the mode reveals that the 3rd degree binomial provides a significant improvement on 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 decomposition can get stuck, as discussed in Section **??**

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 4.3: 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.

4.3.3 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 never 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 such the probability a sample will move significantly further away from the mode is neglijible. In such a scenario it is possible to get stuck in local optima outside of our desired mode neighbourhood.

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

A concrete example would be: $mode = 0.6; \epsilon = 0.05; flip = 0$; (uniform-continuous 0 0.5) = 0.4

In this case, setting flip to true 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 then we currently are. Further the uniform can only successfully resamples values between (0.4, 0.5), which won't change the situation. So 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 and then for the bit shift to be accepted. We will therefore be likely stuck in this interval for a long time before reaching a neighbourhood of the mode.

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 untill just the right combination of bits are pickes.

is a more thorough analysis feasible here?

- Using multiple shifted variants of the prior (shifted by x means mapping sample s to (x + s)%1). It seems that by choosing the shifted priors carefully we could ensure it is impossible for a sample to get stuck in all priors. It can be shown (see Section **??**) 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 **??**.

- 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 unstick 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 seems to not work but is tempting to consider is to attempt 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. If we don't think we're stuck we sample normally. If we do think we're stuck, we can determine in which bits we might be stuck and pick one of these. We can then determine the interval in which the mode should be if we're indeed stuck on this bit (for $flip * 2^{-k}$ the interval will have size 2^{-k-1}) and pick a potential sample uniformly from that interval. If this sample had better log likelihood we would accept.

The problem with this idea is that it isn't just picking proposals from a static prior or proposal kernel anymore, but the proposal pattern will actually be influenced by the log likelihood distribution.

explain why this is wrong

4.3.4 Mixtures of shifted bit decompositions

Here we explore two variants of the mixture of shifted bits solution. 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 a better performance.

Avoiding getting stuck

It turns out that a combination of bit decomposition, 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 see this consider the formulation: total = flip1*0.5 + flip2*0.25 + (uniform-continuous 0 0.25) mode = 0.25 - ϵ flip1 = 0 flip2 = 1 (uniform-continuous

0 0.25) = ϵ => total = 0.25 + ϵ where ϵ can be an arbitrarily small positive number

Here we are stuck since changing the flip to False will be rejected as having lower LL. 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: shiftTotal = (0.25 + s + ϵ) % 1 And flipping the second bit to False would result in a proposal: proposal = s + ϵ

If our shift s is such that s < 0.125 - 2ϵ Then the proposal will be rejected since: |shiftTotal - mode| = |0.25 + s + ϵ - 0.25 + ϵ| = s + 2ϵ < 0.125 |proposal - mode| = | s + ϵ - 0.25 + ϵ | = 0.25 - s + 2ϵ > 0.125 The only other proposal we might make are flipping the first bit 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 usefull on a stuck 2nd bit, we need that s > 0.125. In general, to get unstuck on the kth bit we require that 2^{-k-1} <= s. So any shift larger than 0.25 will guarantee that we never get stuck.

If we also want to be able to reach the mode in one jump from a stuck position, we must also consider: total = flip1*0.5 + flip2*0.25 + (uniform-continuous 0 0.25) mode = 0.375 - ϵ flip1 = 0 flip2 = 0 (uniform-continuous 0 0.25) = 0.25 - ϵ => total = 0.25 - ϵ

For any shift s we choose, the shifted prior will be: shiftTotal = (0.375 + s - ϵ) % 1

If our shift s is such that s > 0.25 + 2*ϵ Then the distance to the mode is: |shiftTotal - mode| = |0.375 + s - ϵ - 0.375 - ϵ| = s - 2*ϵ > 0.25

Therefore, setting the uniform to 0 would still leave the shiftTotal' > mode + ϵ and thus leave us unable to reach the mode in one step.

Note that, in this situation we are not stuck, since setting the uniform to 0 leaves us with shiftTotal' > mode + ϵ that means the shift will accept any reduction to the uniform. Specifically if we accept (uniform-continuous 0 0.25) = ϵ Then switching back to the original 0 shift results in total = mode

+ ϵ - s - ϵ = mode - s And since s > 0.25 + 2*ϵ We are now in a position to accept switching the 2nd bit to 1, which would unstick us.

However, if we wish to jump to the mode from a postiion stuck on the second bit, we've shown that the shift must have the property: 0.125 <= s <= 0.25 And in general, to guarantee that we can jump to the mode when stuck on the kth bit, we need to have a shift s such that: $2^{-k-1} <= s <= 2^{-k}$ Which means that each bit would need a different sized shift.

Empirical performance

Based on the arguments in the previous section, it seems that the placement of the mode can significantly affects the likelihood of getting stuck for binomials of certain depths. 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 performance averaged over all mode placements where $m \in [0.0005, 0.0015, , 0.9995]$ and $\epsilon = 0.0005$

We first 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.

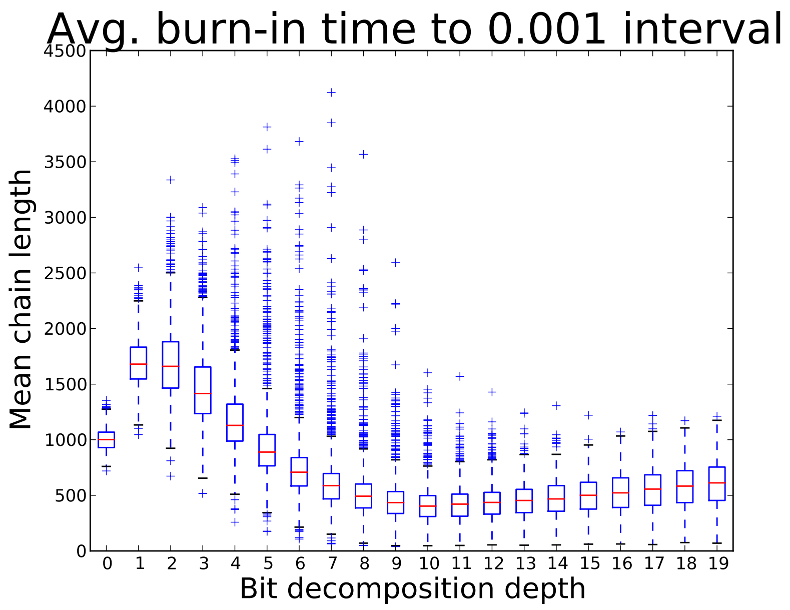


Figure 4.19: Time to 0.001 neighbourhood of mode, averaged over all mode placements, for bit decompositions of different depths using a shift for every bit.

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.

Next we see if the choice of shift size within the interval $2^{-k-1} <= s <= 2^{-k}$ is significant by looking at the burn-in rate for shifts of size 2^{-k-1} (called minimum shifts) for each bit position k.

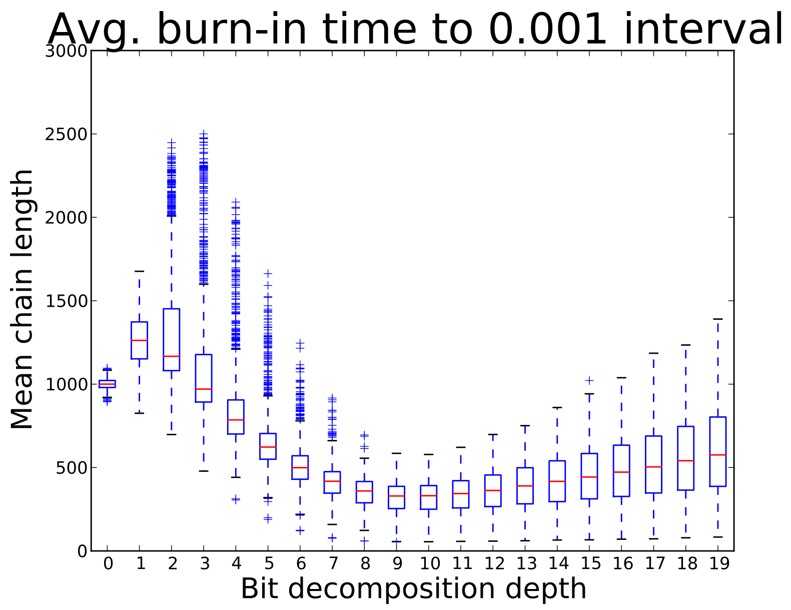


Figure 4.20: Time to 0.001 neighbourhood of mode, averaged over all mode placements, for bit decompositions of different depths using a shift for every bit.

The choice of shift length withing the $[2^{-k-1}, 2^{-k}]$ doesn't appear to be significant.

We would also like to see what happens as the size of the target neighbourhood changes. We repeat the above experiment for a target neighbourhood of size 0.01 (10x larger).

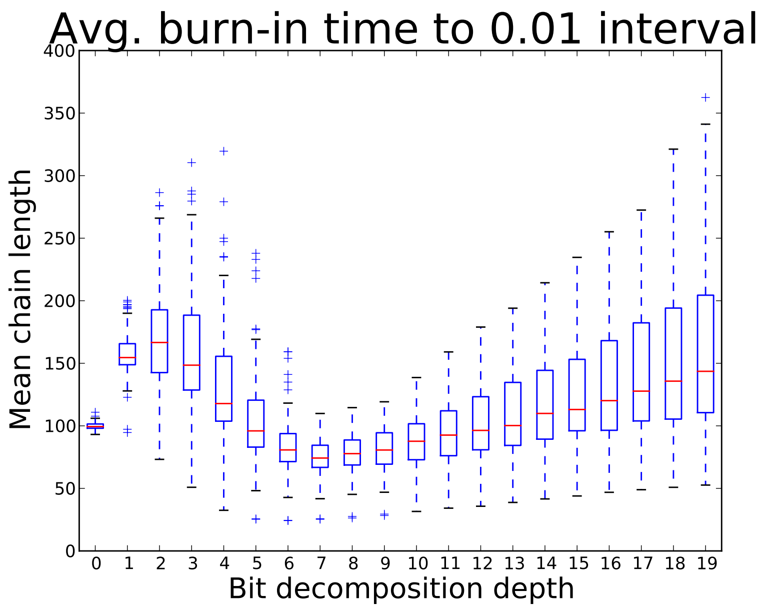


Figure 4.21: Time to 0.01 neighbourhood of mode, averaged over all mode placements, for bit decompositions of different depths using a shift for every bit.

While the bit decomposition still gives some advantage, as the size of the target neighbourhood increases this advantage appears to become less significant, since a larger number of the small bits become irrelevant to our ending up in the desired region.

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 that are sufficiently large to stop bits from getting stuck.

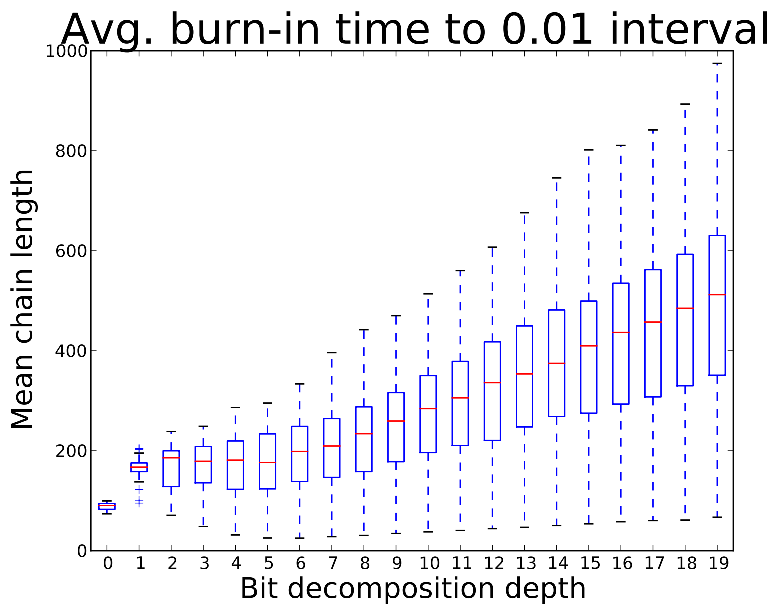
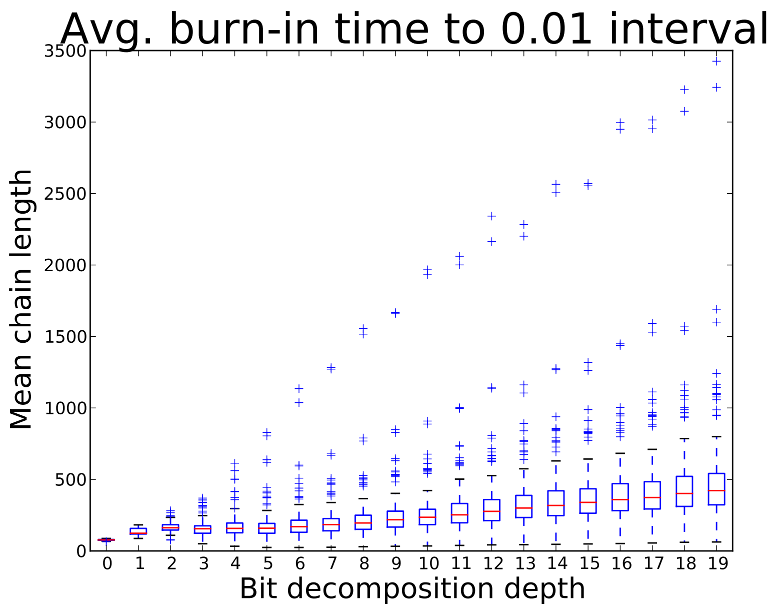


Figure 4.22: Time to 0.01 neighbourhood of mode, averaged over all mode placements, for bit decompositions of different depth using only a shift of size 0.5



talk about distributions over shifts, and more about benefits/drawback of shifts and/or changing depths

Figure 4.23: Time to 0.01 neighbourhood of mode, averaged over all mode placements, for bit decompositions of different depth using only a shift of size 0.25 and 0.5

The performance with a small number of shifts seems significantly worse. It seems using 1 or 2 shifts still allows the algorithm to avoid getting stuck, but the extra number of steps it has to perform to reach the mode from a stuck position degrades the performance.

Determining optimal shifts and shift transitions

It seems we have a wide variety of options regarding what shifts to use and how to transition between them. It therefore seems worthwhile to consider the question of whether we can determine any optimal solutions analytically.

In order to do so, we make some observations:

- The location of the mode neighbourhood determines which regions are traps (i.e. in which regions, once you enter, you can no longer reach the mode without shifting)

- The current sample determines what portion of the traps (if any) are still accessible. Here we make the assumption that the LL of consecutive samples is non-decreasing
- I don't think the prob. of getting to the mode or falling in a trap can be modelled with (mixtures of) geometric distributions since a sample's probability distribution depends on the previous samples (and is therefore not uniform, unless we integrate out its history)
- For a set mode and LL function we can represent the transition between samples via a bit-level markov chain which ignores the trailing uniform. We can then create absorbing states representing the mode neighbourhood and the traps.
- If, for instance, the neighbourhood is only 0.1 of the length of the smallest bit we would implicitly represent the uniform only in the corresponding bit state by giving a 0.1 transition from this bit state to the mode-neighbourhood absorbing state. (this however ignores the movement of the uniform during normal sampling which may end up biasing the results since, for instance, if the mode is in the leftmost bit the uniform will likely already have a very small value by the time the correct bit state is reached. There seem to be heuristic ways to address this but I'll wait to see if it's actually a big issue)
- Using this formulation we should be able to analytically derive the probability of getting stuck and the expected number of steps to the mode. Representing shifted priors should also be possible. The simplest way would be to have 2x the nodes for a combination of 2 shifted priors.

try to get some results using markov chain implementation. Otherwise there's not much point in describing it ...

Chapter 5

Summary and Conclusions

As you might imagine: summarizes the dissertation, and draws any conclusions. Depending on the length of your work, and how well you write, you may not need a summary here.

You will generally want to draw some conclusions, and point to potential future work.