Adaptive Scheduling in MCMC and Probabilistic Programming

David Tolpin  Jan Willem van de Meent  Brooks Paige  Frank Wood
Department of Engineering Science
University of Oxford
Oxford, UK
{dtolpin,jwvdm,brooks,fwood}@robots.ox.ac.uk

Abstract

We introduce an adaptive output-sensitive inference algorithm for MCMC and probabilistic programming, Adaptive Random Database. The algorithm is based on a single-site updating Metropolis-Hasting sampler, the Random Database (RDB) algorithm. Adaptive RDB (ARDB) differs from the original RDB in that the schedule of selecting variables proposed for modification is adapted based on the output of the probabilistic program, rather than being fixed and uniform. We show that ARDB still converges to the correct distribution. We compare ARDB to RDB on several test problems highlighting different aspects of the adaptation scheme.

1 Introduction

One way of improving convergence of Markov Chain Monte Carlo (MCMC) is through online adaptation of the proposal distribution [2, 11, 3], while still ensuring that the sample sequence converges to the correct equilibrium distribution. In a single-site updating Metropolis-Hastings MCMC [6], the proposal distribution can be decomposed into two components:

1. A stochastic schedule (probability distribution) for selecting the next random variable to modify.
2. The kernels from which new values for each of the variables are proposed.

In this paper we concentrate on the first component—adapting the schedule for selecting the variable.

Monte Carlo sampling is a powerful approach to implementing approximate inference for probabilistic programming [7, 8, 13]. Probabilistic programming languages facilitate development of probabilistic models using the expressive power of general programming languages. The goal of inference in such programs is to reason about the posterior distribution over execution traces of the programs conditioned on a particular evidence. Exact inference in probabilistic programs is in many cases infeasible, and approximate inference is widely used instead. One powerful approach to implementing approximate inference for probabilistic programming is based on Monte Carlo sampling techniques [7, 8, 13].

Monte Carlo samplers for probabilistic programming are based on repeated simulations of the program and are easy to implement and parallelize [12, 8]. Single-site Metropolis-Hastings sampling approach proposed in [12], referred here as Random Database (RDB), can be used to straightforwardly transform an arbitrary programming language into a probabilistic programming language. However, the case of implementation of RDB comes with a
relatively slow convergence. Advanced Monte Carlo samplers [10] achieve faster convergence but are more difficult to implement.

Here we develop an adaptive variant of RDB, which adjusts the schedule of selecting variables for modification. First, we review the general structure of a probabilistic program and discuss convergence criteria with respect to program output. Then, based on a choice of one such criterion, we suggest a scheme for tracking the influence of each variable choice to speed up convergence, specifically via adapting the probability of selecting each of the variables based on their influences. Finally, we compare original and Adaptive RDB on several test problems to show how convergence is improved due to adaptation in each of the cases. In the concluding section, we mention other areas where adaptive sampling can be beneficial for convergence in probabilistic programs.

2 Preliminaries

2.1 Adaptive Markov Chain Monte Carlo

The Metropolis-Hastings algorithm (MH) lies in the core of many MCMC methods [11]. A sample \( x \) is a vector \( \{x_1, ..., x_N\} \) in an \( N \)-dimensional space, on which distribution \( p(\cdot) \) is defined. MH draws samples from \( p(\cdot) \) by proposing a new sample \( x^{i+1} \) given the current sample \( x^i \) according to a parameterized proposal distribution \( q_\theta(x^{i+1}|x^i) \). The proposed sample is accepted with the probability

\[
\min \left( \frac{p(x^{i+1})q_\theta(x^i|x^{i+1})}{p(x^i)q_\theta(x^{i+1}|x^i)}, 1 \right)
\]

If \( x^{i+1} \) is rejected, \( x^i \) is re-used as the next sample.

The convergence rate of MH depends on parameters \( \theta \) of the proposal distribution \( q_\theta \). The parameters can be set either offline or online. Variants of MCMC in which the parameters are continuously adjusted based on the features of the sample sequence are called adaptive. Continuous adaptation of parameters of the proposal distribution is a well-known research subject [2, 11, 3]; challenges in design and analysis of Adaptive MCMC methods include optimization criteria and algorithms for the parameter adaptation, as well as conditions of convergence of Adaptive MCMC to the correct equilibrium distribution.

In a single-site updating Metropolis-Hastings algorithm [6] the components of a random sample are updated individually, in either random or systematic order. Assuming the \( j \)th component is chosen at the \( i \)th step for modification, the proposed sample \( x^{i+1} \) may differ from \( x^i \) only in that component, and \( x^{i+1}_k = x^i_k \) for all \( k \neq j \). While often assumed fixed and uniform, the probability distribution according to which the components are scheduled for modification can be varied. Parameters of this scheduling distribution may be viewed as a subset of parameters \( \theta \) of the proposal distribution \( q_\theta \), and adjusted according to optimization criteria of the sampling algorithm. This is what we propose to do.

2.2 Random Database

Following [13], we refer to the MH approach to sampling over the space of all traces—vectors of values of all random variables in the program—proposed in [12] as “random database” (RDB). A RDB sampler is a single-site updating MH sampler where a single variable drawn in the course of a particular interpretation of a probabilistic program is modified via a standard MH proposal, and this modification is accepted by comparing the value of the joint distribution of old and new program traces.

RDB maintains a mapping from vector \( X \) of all random variables associated with an execution trace \( x \) to their parameters and log probabilities of their values in the trace. A new proposal trace is initialized by picking a single variable \( X_j \) from \( X \), and resampling its value \( x'_j \) using a reversible kernel \( \kappa(X'_j|X_j) \). Starting from this initialization, the program is rerun to generate a new trace \( x' \). For each \( k \neq j \), if the type of random variable \( X_k \) remains the same, the previous value \( x_k \) is reused, and its log probability is rescored conditioned on the
Program 1: \( x_1 \) does not depend on \( x_2 \).

Program 2: \( z \) depends on both \( x_1 \) and \( x_2 \).

preceding variables where necessary. When the type of a variable has changed, or a new random variable is encountered, its value is sampled in the usual manner.

Random variables are referenced by their addresses in the execution trace, according to a certain addressing scheme \[12\]. In this paper, when we refer to random variables, or random choices, associated with a trace in the context of their selection or data related to them, we assume that the variables and accompanying data are accessed by their addresses.

2.3 A Probabilistic Programming Language

For probabilistic programs in the empirical evaluation we used a simple probabilistic programming language, similar to Church \[7\], Venture \[9\], and Anglican \[13\]. The language is derived from Scheme \[5\], and augmented by several special forms facilitating probabilistic programming. Three forms, \texttt{assume}, \texttt{observe}, and \texttt{predict}, are used in the paper.

\texttt{assume} defines a variable. The variable may take a random value and appear in \texttt{observe} and \texttt{predict} forms.

\texttt{observe} conditions the distribution of one or more variables defined in \texttt{assume} forms by constraining values of a random function of the variables.

\texttt{predict} reports the result of computing an expression, specified as the argument of the form, in each program trace. \texttt{predicts} produce the output of a probabilistic program.

We used square brackets instead of parentheses for these forms to make the forms stand out in the source code of the probabilistic program.

3 Adaptive Random Database

RDB selects a random variable to modify randomly and uniformly from all variables present in the trace. This is the most straightforward way to ensure that the resulting MH sampler is valid. However, for a faster convergence of the output of a probabilistic program it might make sense to modify different random variables with different probabilities. Consider Programs 1–3. In Program 1 the output expression \( x_1 \) does not depend on \( x_2 \), hence modifying \( x_2 \) does not affect the output sequence, and a half of RDB steps can be saved by modifying only \( x_2 \). In Program 2 the output expression \( z \) depends on both \( x_1 \) and \( x_2 \). However, a change in \( x_2 \) affects \( z \) more than a change in \( x_1 \). Intuitively, assuming that both variables are proposed from their prior distributions, \( x_2 \) should be modified more often (this is actually confirmed by the experiments): by modifying \( x_2 \) more we ensure faster exploration of the output space. Note that this example is tantamount to predicting one component of a vector distributed according to a multivariate normal distribution.\[1\]

\[1\]

\( \text{Let } h = \mathcal{N} \left( \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \right), \quad z = \begin{bmatrix} 0.1 \\ 1.0 \end{bmatrix} h. \text{ Then } E[z] = 1.1, \text{ Var}[z] = 1.01. \]
1 [assume x1 (normal 1 10)]
2 [assume x2 (normal x1 1)]
3 [observe (normal x2 1) 2]
4 [predict x1]

Program 3: x1 is affected by the observation of x2.

Program 3 contains the observe form conditioning the distribution of x2. Here, the intuition about the probabilities of modifying x1 and x2 is not obvious. Despite the fact that the output expression x1 does not directly depends on x2, we could conjecture that modifying x2 affects acceptance or rejection of future samples.

3.1 Algorithm Outline

We aim to develop an adaptive variant of RDB, which adjusts the probability distribution of selecting variables for modification. In many cases, the probability distribution can be adjusted infinitely as the sampling goes on, however certain properties of the adaptation must hold to ensure conversion to the correct equilibrium distribution [11]. Alternatively, it is possible to limit the adaptation to an initial stage of the algorithm, and then select a static non-uniform probability distribution for variable selection.

Let $X_i = \{X_{i1}, X_{i2}, \ldots, X_{iN_i}\}$ be the $N_i$ random variables of the $i$th trace of a certain probabilistic program. We define the variable selection probability distribution using a function $W_i$ from a variable to the variable weight, such that the probability $p_j^i$ of selecting the $j$th variable for modification is:

$$p_j^i = \frac{W_i(X_{ij})}{\sum_{j=1}^{N_i} W_i(X_{ij})}$$

We allow $W_i$ to evolve during an initial number of iterations $K_a$, and then fix $W_i = W_{K_a}$ for the rest of the inference. We will customarily call the first stage adaptation stage and the second exploitation stage. The case when the adaptation is performed infinitely corresponds to $K_a = \infty$. The pseudocode of Adaptive RDB is shown in Algorithm 1.

Algorithm 1 Adaptive RDB

1: Initialize $W_1(\cdot)$ to a constant.
2: Run the program.
3: for $i = 1 \ldots K_a$ do \hspace{1cm} \triangleright Adaptation stage.
4: Randomly select a variable $X_{ij}$ according to $W_i$.
5: Propose a value for $X_{ij}$.
6: Run the program, accept or reject the trace.
7: Compute $W_{i+1}$ based on the program output.
8: end for
9: for $i = K_a + 1 \ldots \infty$ do \hspace{1cm} \triangleright Exploitation stage.
10: Randomly select $X_{ij}$ according to $W_{K_a}$.
11: Propose a value for $X_{ij}$.
12: Run the program, accept or reject the trace.
13: end for

Just like RDB, Adaptive RDB runs the probabilistic program once and then selects variables for modification randomly. Adaptive RDB differs from RDB in that at each iteration the variables are selected according to function $W_i$ from variable addresses to variable weights rather than uniformly.

This high-level description of the algorithm does not detail how $W_i$ is computed for each iteration. Indeed, this is both the most essential and the most complicated part of the algorithm. There are two different aspects here — on one hand, the influence of a given
program trace on the output sequence must be quantified in terms of convergence of the sequence to the target distribution. On the other hand, the influence of the trace must be translated into re-computation of weights of random variables in the trace. Both parts of re-computation of $W_i$ are explained below.

3.2 Quantifying the Influence

Extensive research literature is available on criteria for tuning parameters of Adaptive MCMC [2] [11] [3]. The case of inference in probabilistic programs is different though: the user of a probabilistic program is interested in fast convergence of the program output rather than of the vector of the program’s random variables. Assuming for simplicity that there is a single, scalar, and numeric predict form in the program, the variable selection probability distribution of Adaptive RDB must be tuned in such a way that the sequence of numbers produced by this predict converges to the correct equilibrium distribution as fast as possible. Note that or multiple predicts, any multi-criteria optimization technique can be used; a basic approach that appears to work quite well in practice is to maximize the average convergence measure over all predicts. Numeric predicts corresponding to the non-numeric ones can be added to facilitate adaptation.

In adaptive MCMC variants the acceptance rate can be efficiently used as the optimization objective [11]. However, for convergence of the output sequence an accepted sample that outputs the same value is indistinguishable from a rejected sample. Additionally, while optimal values of the acceptance rate [2] [11] can be used to tune parameters in adaptive MCMC, in Adaptive RDB we do not change the parameters of proposal distributions of individual variables, and assume that they are fixed; by changing variable selection probabilities we attempt to maximize the change in the output sequence so that it converges faster anyway.

Based on these considerations, we quantify the influence of sampling on the output sequence by counting ‘surprising values’. Formally, let $z = \{z_1, \ldots, z_{i-1}, z_i, \ldots\}$ be the output sequence of a probabilistic program. Then the influence of a sample that produced $z_i$ is measured by the sample’s reward $r_i$:

$$r_i = \begin{cases} 1, & \text{if } p(z_i | z_{1:i-1}) < p(z_{i-1} | z_{1:i-1}), \\ 0, & \text{otherwise}. \end{cases}$$

(2)

In other words, an output value is assigned a reward of 1 if it is more surprising than the previous value, and 0 otherwise. Here $p(z_i | z_{1:i-1})$ is the conditional probability, for discrete output, or probability density, for continuous output, of value $z_k$ given the sequence of samples $\{z_1, \ldots, z_{i-1}\}$, according to some external estimate. Since only the ordering of probabilities, rather than their exact values, affects the reward, either multinomial or normal estimate is usually sufficient.

During the adaptation stage, the reward is used to adjust the variable selection probability distribution for the subsequent steps of Adaptive RDB by computing $W_j$ for $j > i$. It may seem sufficient to update, for example, by averaging the rewards, the weight of the variable modified in the same trace in which the sample was produced. This approach will not work for Adaptive RDB though. Consider Program 3: modifying $x2$ may result in an accepted trace, but the value of $x1$, predicted by the program, will remain the same as in the previous trace. Only when $x1$ is also modified and a new trace with the updated values for both $x1$ and $x2$ is accepted, the earlier change in $x2$ is reflected in the output of the program.

In the next section, we discuss propagation of rewards to variable selection probabilities in detail.

3.3 Selecting Variables Based on the Influence

Both RDB and Adaptive RDB modify a single variable per trace, and either re-use or re-compute the probabilities of values of all other variables (except those absent from the previous trace or having an incompatible distribution, for which new values are also sampled). Due to this single-site updating, the influence of modifying a variable on the output can be delayed by several iterations. We propose the following propagation scheme: the history of
Original source code:

```lisp
1 (assume z (lambda (x)
2     (if (> x 1) (flip (/ 1 x))
3     (z (normal 1 10)))))
4 [predict (z 0)]
```

Source code augmented by Adaptive RDB:

```lisp
1 (assume z (lambda (x)
2     (if (> x 1) (RC 1 (flip (/ 1 x)))
3     (z (RC 2 (normal 1 10)))))
4 [predict (z 0)]
```

Program 4: Augmenting the program code for the mapping of variables to rewards.

Variables selected for modification is maintained, and the reward is distributed between all of the variables in the history, such that more recent variables receive a greater reward.

This scheme can be realized under the following simplifying assumptions:

- A single entry in the history is ‘responsible’ for the reward of the most recent sample.
- The conditional probability \( p_r \) that an entry is responsible for the reward, given none of the more recent entries is responsible, is constant and known in advance.
- The reward is distributed among the variables according to their responsibility probability.

Let \( h = \{h_1, \ldots, h_i\} \) be the history of variables selected for modification, numbered backwards, such that at the \( i \)th iteration the most recent selection is \( h_1 \) and the first selection is \( h_i \). Then the \( j \)th entry receives the reward \( \varphi_{ij} r_i \), where

\[
\varphi_{ij} = \begin{cases} 
(1 - p_r)^{j-1} p_r & \text{if } j < i, \\
(1 - p_r)^{i-1} & \text{if } j = i. 
\end{cases}
\]  

(3)

It is easy to see that \( \sum_{j=1}^{i} \varphi_{ij} = 1 \).

Average rewards \( R = \{R_1, R_2, \ldots, R_N\} \) of the random variables can be used to compute their weights \( W_i(X_j) \). However, implemented straightforwardly, this approach faces two problems.

1. Variables are referenced by their addresses in the trace [12], and in the presence of recursion the set of variable addresses is potentially infinite.
2. Sample average rewards may deviate significantly from the reward means, resulting in some variables being selected too rarely or not selected at all. As a result, the output sequence may not converge to the correct equilibrium distribution.

3.3.1 Address Mapping

A finite sufficiently small set of maintained average rewards is crucial for variable selection: if the set is too large, little information is used to update every single member of the set. Adaptive RDB achieves this by mapping variable addresses to a finite set of reward addresses, for example, by automatically augmenting (through source-to-source transformation) every location of a random choice in the source code by a unique identifier. Let \( M \) be a mapping from a variable address to the reward address. Then, for history entry \( h \) the reward for address \( M(h) \) is updated, and the probability to select variable \( X \) is based on the reward for address \( M(X) \).

Program [4] is an example of automatic code augmentation. There are two source code locations with random choices, marked by RC 1 and RC 2 in the augmented code. Due to recursion, infinitely many variable addresses can correspond to either of the choices in the course of a program run, and multiple variables can map to RC 2 in any single trace. Adaptive RDB would accumulate rewards for RC 1 and RC 2 based on influence of all of
the variables mapped to each of the source code addresses, and select any of the variables corresponding to each choice with equal probability.

### 3.3.2 Upper Confidence Bounds on Rewards

To avoid degeneracy due to underestimation of average rewards, we use upper confidence bounds on rewards to compute the variable selection probabilities, an idea which we borrowed from the UCB family of algorithms for multi-armed bandits \cite{auer2002}. For each reward address we maintain both the average reward $R$ and the normalized count $C$, which we compute as the sum of reward fractions (Equation 3) used to compute $R$. Following UCB1 \cite{auer2002}, we compute the upper confidence bound $\hat{R}$ as the sum of the average reward and the exploration term:

$$\hat{R} = R + \sqrt{\gamma \frac{\log K}{C}} \tag{4}$$

where $K$ is the sum of reward fractions for all rewards, equal to the number of iterations of Adaptive RDB performed so far, and $\gamma$ is the exploration factor. The default value for $\gamma$ is 2 in UCB1; in practice, a lower value of $\gamma$ is often preferable, and the best value depends on the acceptance rate of the Metropolis-Hastings chain. Note that variable selection in Adaptive RDB is different from arm selection in multi-armed bandits: unlike in bandits, where we want to sample the best arm at an increasing rate, in Adaptive RDB we expect $W_i(\cdot)$ to converge to an equilibrium, where zero weights (if any) may correspond only to variables which never affect the output sequence.

Using upper confidence bounds with a sufficiently large exploration factor $\gamma$ ensures that the convergence conditions for Adaptive MCMC \cite{abbeel2004} hold for Adaptive RDB. Indeed, in the case of the reward based on the ratio of surprising values in the output sequence (Equation 2), the condition of diminishing adaptation holds, because the change in the average reward is bounded (the reward of 1 is distributed between random choices) and the upper bound on the change decreases (the change in the total reward is at most $1/i$ at the $i$th iteration of the algorithm). In addition, bounded convergence is guaranteed due to selection of random choices based on upper confidence bounds on their average rewards, such that all choices that can potentially affect the output sequence are sampled infinitely.

A formal proof of convergence of Adaptive RDB with infinite adaptation phase (Algorithm 1) is beyond the scope of this paper, and is a subject for future work. However, empirical evaluation presented in the next section demonstrates practical convergence of Adaptive RDB on a range of inference examples.

### 4 Empirical Evaluation

We compare Adaptive RDB to RDB on a set of probabilistic programs. Implementations of both algorithms were verified on a set of tests to converge to the correct distribution. Whenever convergence rates of Adaptive RDB and RDB are compared, the rates are presented with respect to the number of tests, or simulations, of the probabilistic programs. The additional computation involved into adaptation takes negligible time, and the computational effort per sample is approximately the same for both algorithms.

The first comparison is illustrative, and its purpose is to visualize the anticipated influence of adaptation on sampling. The algorithms are compared on a probabilistic program without any observe and with two predicts (Program 5). There are 17 independent normal random variables, $x_1 \ldots x_{17}$, and the predicted expressions $z_1$ and $z_2$ are linear combinations of some of the variables. Just like in Program 3, this is equivalent to predicting only two components of a multivariate normal distribution. However, the sampling algorithm treats a probabilistic program as a black box, and must explore the whole space in order to output the predictions.

The scatter plots in Figure 1 display 500 predicted samples ($z_1, z_2$) after 5000 samples of burn-in. The cyan background is formed by 20000 samples produced by each of the algorithms and approximates the shape of the output sample distribution. Adaptive RDB explores the space better than RDB, and the effective sample size of Adaptive RDB is
Program 5: Predicting components of a multivariate normal distribution.

\begin{verbatim}
1 [assume x1 (normal 1 1)]
2 ...
3 [assume x17 (normal 1 1)]
4 [assume z1 (+ (* 0.4 x1) (* 0.5 x2) (* 0.1 x3))]
5 [assume z2 (+ (* 0.6 x1) (* 0.3 x2)
6 (* 0.1 x4) (* 0.2 x5))]
7 [predict z1]
8 [predict z2]
\end{verbatim}

Figure 1: Predicting components of a multivariate normal distribution: RDB vs. Adaptive RDB, 500 samples after 5000 samples of burn-in.

noticeably greater; the Kolmogorov-Smirnov statistics for \( z_1 \), \( z_2 \) are (0.2073, 0.1003) for RDB and (0.0803, 0.0495) for Adaptive RDB. The histogram shows relative rewards and modification counts for each of the random variables. According to the histogram, \( x_1 \) and \( x_2 \) were modified most frequently, followed by \( x_3 \), \( x_4 \), and \( x_5 \). Other random variables were modified at a much lower rate. Indeed, their modification does not affect \( (z_1, z_2) \).

These results are encouraging. Nonetheless, the harder challenges for adaptive sampling are mutually dependent random variables, such that one variable is a parameter of the distribution of another variable, and observed values of functions of random variables; besides that, the influence of adaptation on convergence must be quantitatively assessed. In the following tests (Programs 6 and 7, Figures 2–4) the convergence rates of RDB and Adaptive RDB are compared. Kulback-Liebler (KL) divergences of program outputs with respect to the target distributions, obtained using independent exact methods, are reported. In addition, the relative modification counts and relative average rewards for each of the variables are shown. In each test, 50000 samples are produced, KL divergences for the range...
1 [assume initial-state-distribution
2 (normalize (list 1.0 1.0 1.0))]
3 [assume get-transition-vector
4 (lambda (s)
5  (cond ((= s 0) (list 0.1 0.5 0.4))
6  ((= s 1) (list 0.2 0.2 0.6))
7  ((= s 2) (list 0.15 0.15 0.7))))]
8 [assume transition
9  (lambda (prev-state)
10       (discrete
11       (get-transition-vector prev-state)))]
12 [assume get-state
13    (mem (lambda (index)
14        (if (<= index 0)
15          (discrete
16          initial-state-distribution)
17          (transition
18          (get-state (- index 1))))))]
19 [assume get-state-observation-mean
20   (lambda (s)
21       (cond ((= s 0) -1)
22         ((= s 1) 1)
23         ((= s 2) 0)))]
24 [observe (normal (get-state-observation-mean
25         (get-state 1)) 1) .9]
26 ...
27 [observe (normal (get-state-observation-mean
28         (get-state 16)) 1) -1]
29 [predict (get-state 0)]
30 ...
31 [predict (get-state 17)]

Program 6: HMM

of 10,000–50,000 samples, and rewards and counts at the end of the inference, are displayed.
In the KL divergence plots the solid lines corresponds to the median, and the dashed lines
to 25% and 75% percentiles, taken over 25 runs of the corresponding inference algorithm
with different random number seeds. In all invocations of Adaptive RDB the probability
\( p_r \) (Equation 3) was set to 0.5, and the exploration factor \( \gamma \) (Equation 4) to 0.1. Either normal
or multinomial probability estimate was used for computing sample rewards (Equation 2).

Program 6 is a latent state inference problem in an HMM with three states, one-dimensional
normal observations (0.9, 0.8, 0.7, 0, -0.025, 5, 2, 0.1, 0, 0.13, 0.45, 6, 0.2, 0.3, -1, -1) with
variance 1.0, a known transition matrix, and known initial state distribution. There are 17
distinct random variable addresses in all traces of the program.

When only the 0th state is predicted (Figure 2), resampling of the transition between the
0th and the 1st state obtains the highest average reward. By choosing for modification the
random variable corresponding to this transition, Adaptive RDB achieved a much faster
convergence — the median of KL divergence for RDB is above the 75% percentile of the
KL divergence for Adaptive RDB. The total number of modifications of the variable corre-
sponding to the transition is approximately 4 times greater than of any other variable.

When only the 17th state is predicted (Figure 3), many variables obtain similar average
rewards. Still, the 7th, 8th, and 13th random variables have lower rewards and are chosen
less frequently than the rest of the variables, apparently due to observed values for the
 corresponding states. As a result, Adaptive RDB converges faster. Although the difference
between KL divergences is not as prominent as in Figure 2 KL divergence for Adaptive
RDB is still consistently lower than for RDB.

Program 6 was chosen as a test because of branching—depending on the value of \( r \), there are
either one or two random variables in any given trace. The convergence rate of Adaptive
RDB is again higher than of RDB (Figure 4). The adaptation of variable modification
probabilities is counter-intuitive at the first glance: according to the histogram, the second random variable, which corresponds to the sampling from the Poisson distribution in the expression for 1, and appears only in some of the traces, should be selected for modification, if present in the trace, much less frequently than the other random variable, which is always in the trace.

This adaptation, however, makes perfect sense in the context of the RDB caching scheme. Values for random variables, which are not present in the current trace, are dropped from the cache [12]. Hence, in an accepted trace with two random variables the second variable was just re-sampled, a ‘successful’ value for the variable was obtained, and should not be modified again immediately. This test demonstrates the ability of Adaptive RDB to compensate for peculiarities of the underlying sampling scheme in addition to adaptation to the structure of the probabilistic program.

To summarize, Adaptive RDB consistently attained faster convergence than RDB, measured by KL divergence between the ongoing output distribution of the random program and the target independently obtained distribution. The variable selection probabilities computed
Figure 3: HMM, predicting the 17th state.

Program 7: Simple branching.

by Adaptive RDB are justified by the structure of the probabilistic programs, as well as by the properties of the sampling algorithm.
5 Contribution and Future Work

In this paper we introduced a new algorithm, Adaptive RDB, for approximate inference in probabilistic programs. This algorithm adjusts sampling parameters based on the output of the probabilistic program in which the inference is performed. Contributions of the paper include

- A scheme of rewarding random samples based on program output.
- An approach to propagation of sample rewards to MH proposal scheduling parameters.
- An application of this approach to RDB, where the probabilities of selecting each variable for modification are adjusted.

Adaptive RDB was compared to RDB, its non-adaptive counterpart, and was found to consistently outperform RDB on several probabilistic programs, while still being almost
as easy to implement. The time cost of additional computation due to adaptation was negligible.

Although presented in the context of a particular sampling algorithm, the adaptation approach can be extended to other sampling methods. We believe that various sampling algorithms for probabilistic programming can benefit from output-sensitive adaptation. An additional potential for improvement lies in reward computation and propagation. Preliminary experiments showed that a reward scheme that takes into account the amount of difference between the samples improves convergence. Acquisition of dependencies between predicted expressions and random variables is another promising research direction.

Overall, output-sensitive approximate inference appears to bring clear advantages and should be further explored in the context of probabilistic programming models and algorithms.

6 Acknowledgments

This material is based on research sponsored by DARPA through the U.S. Air Force Research Laboratory under Cooperative Agreement number FA8750-14-2-0004.

References

[1] Christophe Andrieu, Nando de Freitas, Arnaud Doucet, and Michael I. Jordan. An introduction to MCMC for machine learning. Machine Learning, 50(1-2):5–43, 2003.

[2] Christophe Andrieu and Johannes Thoms. A tutorial on adaptive MCMC. Statistics and Computing, 18(4):343–373, 2008.

[3] Yves Atchadé, Gersende Fort, Eric Moulines, and Pierre Priouret. Adaptive Markov chain Monte Carlo: theory and methods. In David Barber, A. Taylan Cemgil, and Silvia Chiappa, editors, Bayesian Time Series Models, pages 32–51. Cambridge University Press, 2011. Cambridge Books Online.

[4] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the Multiarmed Bandit problem. Machine Learning, 47(2-3):235–256, 2002.

[5] R. Kent Dybvig. The Scheme Programming Language. MIT Press, fourth edition, 2009.

[6] Dani Gamerman and Hedibert F. Lopes. Markov Chain Monte Carlo: Stochastic Simulation for Bayesian Inference. Chapman and Hall/CRC, 2006.

[7] Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: a language for generative models. In Proc. of Uncertainty in Artificial Intelligence, 2008.

[8] Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. Probabilistic programming. In International Conference on Software Engineering (ICSE, FOSE track), 2014.

[9] Vikash K. Mansinghka, Daniel Selsam, and Yura N. Perov. Venture: a higher-order probabilistic programming platform with programmable inference. CoRR, abs/1404.0099, 2014.

[10] B. Paige, F. Wood, A. Doucet, and Y.W. Teh. Asynchronous anytime sequential Monte Carlo. In Advances in Neural Information Processing Systems, 2014, to appear.

[11] Gareth O. Roberts and Jeffrey S. Rosenthal. Examples of adaptive MCMC. Journal of Computational and Graphical Statistics, 18(2):349–367, 2009.

[12] David Wingate, Andreas Stuhlmüller, and Noah D. Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In Proc. of the 14th Artificial Intelligence and Statistics, 2011.

[13] Frank Wood, Jan Willem van de Meent, and Vikash Mansinghka. A new approach to probabilistic programming inference. In Artificial Intelligence and Statistics, 2014.