8 Appendix

8.1 Analysis of $B(p_1)$

The right-hand side of (15) is

$$B(p_1) = \frac{1 + \frac{p_1 \log p_1}{1-p_1}}{\frac{1}{p_1} + \frac{p_1 \log p_1}{1-p_1}} \tag{17}$$

We will show that the function is monotonous and defined everywhere in $p_1 \in [0, 1]$. The function is immediately defined for $p_1 \in (0, 1)$, because both the numerator and the denominator are defined, and the denominator is not equal to zero. To determine the values of $B(p_1)$ for $p_1 = 0$ and $p_1 = 1$, we shall compute the limits of $B(p_1)$ at these values (for example, using Maxima [12]).

$$\begin{aligned} \lim_{p_1 \to 0} B(p_1) &= 0 \\ \lim_{p_1 \to 1} B(p_1) &= \frac{1}{3} \end{aligned} \tag{18}$$

It now remains to show that $B(p_1)$ is monotonic for $p_1 \in (0, 1)$. One can use Maxima again (or perform the derivations manually) to obtain $\frac{dB(p_1)}{dp_1}$ symbolically and verify that it is positively defined on the open interval.

8.2 Additional discussion on ergodicity

Even in cases where the required conditions for an adaptive MCMC scheme to be ergodic are not met, if we assume that any particular choice of proposal we may adapt to would itself leave the stationary distribution invariant, then a valid sampling scheme always can be established by adapting the proposal distribution for only some finite number of iterations, then using the tuned proposal for the remainder of inference.

Before directly addressing the ergodicity of the adaptive LMH algorithm, we first look at the adaptive componentwise Metropolis-Hastings approach outlined in Algorithm 1. Conditions under which such an adaptive random scan Metropolis-within-Gibbs algorithm is ergodic have been established explicitly [8, Theorems 4.10 and 5.5]. Given suitable assumptions on the target density of interest, we additional require the probability vector $|\alpha^t - \alpha^{t-1}| \to 0$, and that for any particular component k, the we have probability $\alpha_k^t > \epsilon > 0$. Both of these are satisfied by our approach: from Corollary 1, we ensure that rewards across each x_i converge to positive values.

The class of models representable by probabilistic programs is very broad; for a large class of models, the adaptive LMH algorithm is exactly a an adaptive random scan Metropolis-within-Gibbs algorithm. To discuss the conditions under which this is the case, we introduce the concept of *structural* versus *structure-preserving* random choices, following [18]. For our purposes we define a *structure-preserving* random choice x_k to be all those which do not affect the existence of other x_m in the trace. Structural random choices are defined as all x_i which are not structure-preserving.

Suppose we were to restrict the expressiveness of our language to admit only programs with no structural random choices: in such a language, the LMH algorithm in Algorithm 2 reduces to the adaptive componentwise MH algorithm, and will be ergodic for a wide class of models, due the results of [8]. To see this, note that in a program with only structure-preserving random choices, the full set of latent random variables $\boldsymbol{x}$ is finite, and fixed across all execution traces, so the number of random variables N_t is fixed for all t. Furthermore, re-scoring is always possible; that is, after selecting any single variable x_k to update, for any sampled value x'_k, and any $m > k$, the existing value x_m is still in the support of x'_m. Value only fall out of support when there is a change in random procedure type, which can only occur in the event of a structural change.

While the theoretical result here requires restricting ourselves to models containing only structure-preserving random choices, we note that nearly all well-studied statistical models consist only of such choices, including three of the examples we investigate empirically below. We conjecture that the result does indeed extend to the more general case. For example, in all programs which we expect to halt with probability one, recursion depth cannot truly be infinite, allowing us to at least bound the dimension of $\boldsymbol{x}$. Static code analysis could be used to identify which additional random variables are always resampled as the result of modifying a given structural random choice, and treated conceptually as a block Gibbs update. We leave a precise theoretical analysis of the space of probabilistic programs in which adaptive MCMC schemes may be ergodic to future work.

8.3 A Probabilistic Programming Language

For probabilistic programs in the empirical evaluation we use a simple probabilistic programming language, similar to Church [6], Venture [11], and Anglican [17]. The language is derived from Scheme, and augmented by several special forms facilitating probabilistic programming. Three forms, `assume`, `observe`, and `predict`, are used in the paper.

`[assume v e]` binds a variable `v` to the value of the expression `e`. The bound variable may take a random value and appear in `observe` and `predict` forms.
`[observe d e]` conditions distribution `d` on the specified value `e`.
`[predict e]` adds the value of `e` to the program output.

We use square brackets instead of parentheses for these forms to distinguish them from other expressions in the probabilistic program.

8.4 Source code of Probabilistic Programs

Program 1 HMM

```
[assume initial-state-dist (list 1/3 1/3 1/3)]
[assume get-t
  (lambda (s)
    (cond ((= s 0) (list 0.1 0.5 0.4))
          ((= s 1) (list 0.2 0.2 0.6))
          ((= s 2) (list 0.15 0.15 0.7))))]
[assume transition (lambda (prev-state) (sample (discrete (get-t prev-state))))]
[assume get-state
  (mem (lambda (index)
          (if (<= index 0)
            (sample (discrete initial-state-dist))
            (transition (get-state (- index 1))))))]
[assume get-obs-mean
  (lambda (s) (cond ((= s 0) -1)
                    ((= s 1) 1)
                    ((= s 2) 0)))]
[observe (normal (get-obs-mean (get-state 1)) 1) .9]
...
[observe (normal (get-obs-mean (get-state 16)) 1) -1]
[predict (get-state 0)]
[predict (get-state 17)]
```

Program 2 Gaussian process hyperparameter estimation

```
[assume data '((0.0  0.5 ) (1.0  0.4 ) (2.0  0.2 ) (3.0 -0.05) (4.0 -0.2 ) (5.0  0.1))]
[assume belief (normal 0 1)]
[assume positive-belief (gamma 1 1)]
[assume a (sample belief)]
[assume b (sample belief)]
[assume c (sample belief)]
[assume d (sample positive-belief)]
[assume g (sample positive-belief)]
[assume m (lambda (x) (+ c (* x (+ b (* x a)))))]
[assume k (lambda (x y) (let ((dx (- x y))) (* d (exp (- (/ (* dx dx) 2. g))))))]
[assume gp (GP m k)]
(reduce (lambda (_ pnt) [observe gp pnt]) () data)
[predict a]
[predict b]
[predict c]
```

Program 3 Logistic regression on Iris dataset

```
;; test-setosa and test-not-setosa are randomly
;; chosen and fixed for each run of the inference.
[assume features (lambda (record) (cons 1 (butlast record)))]

;; remove test data from the training set
[assume iris-data (filter
                    (lambda (record)
                   (not (or (= record test-setosa)
                    (= record test-not-setosa))))
                    iris-data)]

[assume sigma (sqrt (sample (gamma 1 1)))]
[assume b (repeatedly 5 (lambda () (sample (normal 0. sigma))))]
[assume z (lambda (x) (/ 1. (+ 1. (exp (* -1. (dot b x))))))]

;; train
(reduce (lambda (_ record)
          [observe (flip (z (features record))) (= (last record) iris-setosa)])
        () iris-data)
[assume is-setosa (lambda (x) (sample (flip (z x))))]

;; classify
[predict (is-setosa (features test-setosa))]
[predict (is-setosa (features test-not-setosa))]
```