Journal of Statistical Software

June 2017, Volume 78, Code Snippet 1. *doi: 10.18637/jss.v078.c01*

Multivariate-From-Univariate MCMC Sampler: The R Package MfUSampler

Alireza S. Mahani
Sentrana Inc.

Mansour T. A. Sharabiani
Imperial College London

Abstract

The R package **MfUSampler** provides Markov chain Monte Carlo machinery for generating samples from multivariate probability distributions using univariate sampling algorithms such as the slice sampler and the adaptive rejection sampler. The multivariate wrapper performs a full cycle of univariate sampling steps, one coordinate at a time. In each step, the latest sample values obtained for other coordinates are used to form the conditional distributions. The concept is an extension of Gibbs sampling where each step involves, not an independent sample from the conditional distribution, but a Markov transition for which the conditional distribution is invariant. The software relies on proportionality of conditional distributions to the joint distribution to implement a thin wrapper for producing conditionals. Examples illustrate basic usage as well as methods for improving performance. By encapsulating the multivariate-from-univariate logic, package **MfUSampler** provides a reliable package for rapid prototyping of custom Bayesian models while allowing for incremental performance optimizations such as taking advantage of conditional independence, and high-performance implementation of function evaluations. Utility functions for MCMC diagnostics as well as sample-based construction of predictive posterior distributions are provided in **MfUSampler**.

Keywords: Markov chain Monte Carlo, slice sampler, adaptive rejection sampler, Gibbs sampling, Metropolis.

1. Introduction

Bayesian inference software such as Stan (Carpenter *et al.* 2017; Stan Development Team 2017), **OpenBUGS** (Thomas, O'Hara, Ligges, and Sturtz 2006), and **JAGS** (Plummer 2003) provide high-level, domain-specific languages (DSLs) to specify and sample from probabilistic directed acyclic graphs (DAGs). In some Bayesian projects, the convenience of using such DSLs comes at the price of reduced flexibility in model specification, and suboptimality of the underlying sampling algorithms used by the compilers for the particular distribution that

must be sampled. Furthermore, for large projects the end-goal might be to implement all or part of the sampling algorithm in a high-performance – perhaps parallel – language. In such cases, researchers may choose to start their development work by 'rolling their own' joint probability distributions from the DAG specification, followed by application of their choice of a sampling algorithm to the joint distribution.

Many Markov chain Monte Carlo (MCMC) algorithms have been proposed over the years for sampling from complex posterior distributions. Perhaps the most widely-known algorithm is Metropolis (Metropolis, Rosenbluth, Rosenbluth, Teller, and Teller 1953) and its generalization, Metropolis-Hastings (MH; Hastings 1970). These multivariate algorithms are easy to implement, but they can be slow to converge without a carefully-selected proposal distribution. A particular flavor of MH is the stochastic Newton sampler (Qi and Minka 2002), where the proposal distribution is a multivariate Gaussian based on the second-order Taylor series expansion of the log-probability. This method has been implemented in the R package **sns** (Mahani, Hasan, Jiang, and Sharabiani 2016). It can be quite effective for twice-differentiable, log-concave distributions such as those encountered in generalized linear regression (GLM) problems. Another flavor of MH is the t-walk algorithm (Christen and Fox 2010) which uses a set of scale-invariant proposal distributions to co-evolve two points in the state space. Hamiltonian Monte Carlo (HMC) algorithms (Girolami and Calderhead 2011; Neal 2011) have also gained popularity due to emerging techniques for their automated tuning (Hoffman and Gelman 2014).

Univariate samplers tend to have few tuning parameters and thus are well suited for black-box MCMC software. Two important examples are adaptive rejection sampling (Gilks and Wild 1992) (or ARS) and slice sampling (Neal 2003). ARS requires the log-density to be concave, and needs its first derivative, while the slice sampler is generic and derivative-free. To apply these univariate samplers to multivariate distributions, they must be applied one-coordinate-at-a-time according to the Gibbs sampling algorithm (Geman and Geman 1984), where at the end of each univariate step the sampled value is used to update the conditional distribution for the next coordinate. **MfUSampler** encapsulates this logic into a package function, providing a fast and reliable path towards Bayesian model estimation for researchers working on novel DAG specifications. In addition to slice sampler and ARS, the current version of **MfUSampler** (1.0.4) contains the adaptive rejection Metropolis sampler (Gilks, Best, and Tan 1995) and the univariate Metropolis sampler with Gaussian proposal. Univariate samplers have their limits: When the posterior distribution exhibits a strong correlation structure, one-coordinate-at-a-time algorithms can become inefficient as they fail to capture important geometry of the space (Girolami and Calderhead 2011). This has been a key motivation for research on black-box multivariate samplers, such as adaptations of the slice sampler (Thompson 2011) or the no-U-turn sampler (Hoffman and Gelman 2014).

The rest of this article is organized as follows. In Section 2 we provide an overview of the theory, the software components and the process flow underlying **MfUSampler**. In Section 3 we illustrate how to use the software with an example based on a real data set. Section 4 presents and illustrates several performance optimization techniques that can be used within the **MfUSampler** framework. Section 5 provides a summary and concluding remarks. Computational details are included about the session information used to run the R scripts in this paper and Appendix A contains the proof of Lemma 1. Package **MfUSampler** (Mahani and Sharabiani 2017) is available from the Comprehensive R Archive Network (CRAN) at `https://CRAN.R-project.org/package=MfUSampler`.

2. Theory and implementation of package MfUSampler

In this section, we discuss the theoretical underpinnings of the **MfUSampler** package, including extended Gibbs sampling (Section 2.1), and proportionality of conditional and joint distributions (Section 2.2). Software components of **MfUSampler**, described in Section 2.3, are best understood given this theoretical background.

2.1. Extended Gibbs sampling

Gibbs sampling (Bishop 2006) involves iterating through state space coordinates, one at a time, and drawing samples from the distribution of each coordinate, conditioned on the latest sampled values for all remaining coordinates. Gibbs sampling reduces a multivariate sampling problem into a series of univariate problems, which can be more tractable.

In what we refer to as 'extended Gibbs sampling', rather than requiring an independent sample from each coordinate's conditional distribution, we expect a Markov transition for which the conditional distribution is an invariant distribution. Among the current univariate samplers implemented in **MfUSampler**, the adaptive rejection sampler produces a standard Gibbs sampler while the remaining samplers fall in the 'extended Gibbs sampler' category. The following lemma forms the basis for proving the validity of extended Gibbs sampling as an MCMC sampler. (For a discussion of ergodicity of MCMC samplers, see Roberts and Rosenthal 1999; Jarner and Hansen 2000.)

Lemma 1. *If a coordinate-wise Markov transition leaves the conditional distribution invariant, it will also leave the joint distribution invariant.*

The proof is given in Appendix A. A full Gibbs cycle is simply a succession of coordinate-wise Markov transitions, and since each one leaves the target distribution invariant according to the above lemma, the same is true of the resulting composite Markov transition kernel.

2.2. Proportionality of conditional and joint distributions

Using univariate samplers within the Gibbs sampling framework requires access to conditional distributions, up to a multiplicative constant (in terms of each coordinate being sampled). Referring to the conditional distribution for the kth coordinate as $p(x_k|\mathbf{x}_{\setminus k})$, we examine the following application of Bayes' rule

$$p(x_k|\mathbf{x}_{\setminus k}) = \frac{p(x_k, \mathbf{x}_{\setminus k})}{p(\mathbf{x}_{\setminus k})} \propto p(x_k, \mathbf{x}_{\setminus k}), \tag{1}$$

to observe that, since the normalizing factor – $p(\mathbf{x}_{\setminus k})$ – is independent of x_k, the joint and conditional distributions are proportional. Therefore, the joint distribution can be supplied to univariate sampling routines in lieu of conditional distributions during each step of Gibbs sampling. **MfUSampler** takes advantage of this property, as described next.

2.3. Implementation

The **MfUSampler** software consists of 5 components: (1) connectors, (2) univariate samplers, (3) Gibbs wrappers, (4) diagnostic utilities, and (5) full Bayesian prediction. Figure 1 provides an overview of how these components fit in the overall process flow of **MfUSampler**. Below we describe each component in some detail.

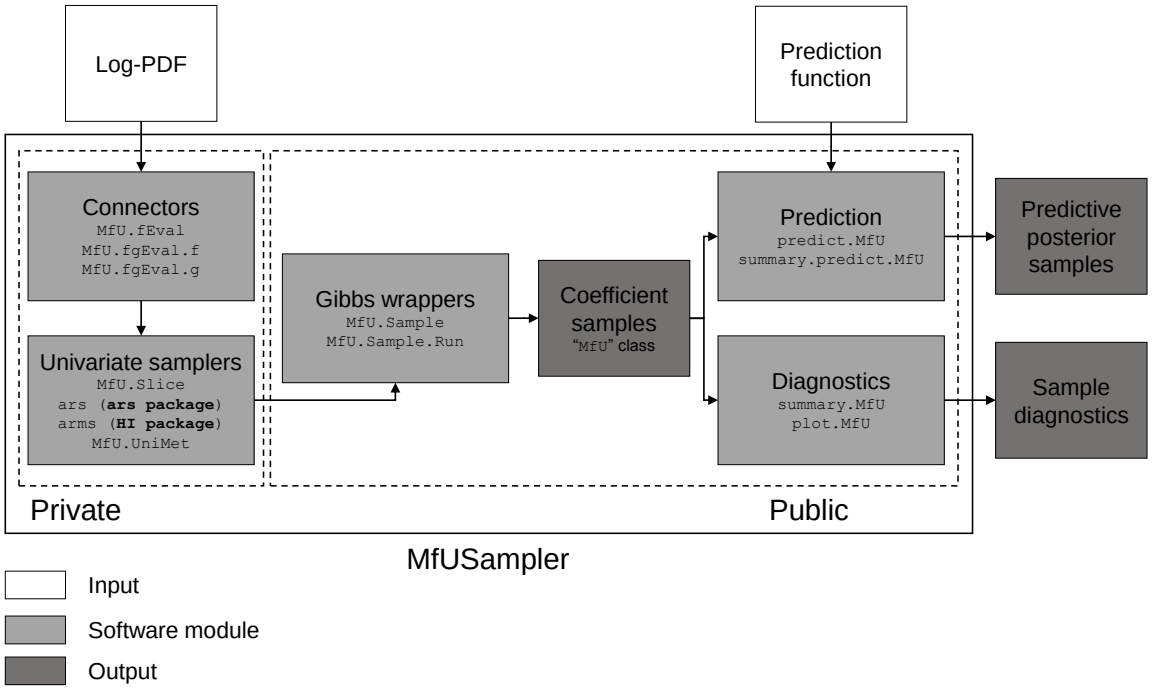

Figure 1: Software components and process flow for **MfUSampler**. Connector and sampler private modules mediate Gibbs sampling of user-supplied PDF.

Connectors: The internal functions `MfU.fEval`, `MfU.fgEval.f` and `MfU.fgEval.g` return the conditional log-density and its gradients for each coordinate, using the underlying joint log-density and its gradient vector (Section 2.2). These functions act as the bridge between the user-supplied, multivariate log-densities and the univariate samplers. The vectorized functions `MfU.fgEval.f` and `MfU.fgEval.g` are used by the `ars` function (see below). Other samplers, which are derivative-free, use `MfU.fEval`.

Univariate samplers: These functions are responsible for producing a single MCMC jump for univariate distributions resulting from applying the connector functions to the user-supplied, multivariate distribution. As of version 1.0.0, **MfUSampler** supports the following 4 samplers:

1. Univariate slice sampler with stepout and shrinkage (Neal 2003). The code, wrapped in the internal function `MfU.Slice`, is taken – with small modifications – from Radford Neal's website (`http://www.cs.toronto.edu/~radford/ftp/slice-R-prog`). The slice sampler is derivative-free and robust, i.e., its performance is rather insensitive to its tuning parameters. It is the default option in `MfU.Sample` and `MfU.Sample.Run`.
2. Adaptive rejection sampler (ARS; Gilks and Wild 1992), imported from the R package **ars** (Pérez-Rodríguez, Wild, and Gilks 2014). ARS requires the log-density to be concave, and requires access to its gradient. Our experience shows that it is somewhat more sensitive to the choice of tuning parameters, compared to the slice sampler (Section 3.3).
3. Adaptive rejection Metropolis sampler (Gilks *et al.* 1995), imported from the R

package **HI** (Petris, Tardella, and Gilks 2013). This algorithm is an adaptation of ARS with an additional Metropolis acceptance test, aimed at accommodating distributions that are not log-concave. The algorithm can also be applied directly to a multivariate distribution. However, see Gilks, Neal, Best, and Tan (1997) for a discussion of how the initial values for this algorithm may – and may not – be chosen to ensure validity of the resulting MCMC chain.

4. Univariate Metropolis with Gaussian proposal, implemented by the internal function `MfU.UniMet`. This simple sampler can be inefficient, unless the standard deviation of the Gaussian proposal is chosen carefully. It has been primarily included as a reference for other, more practical choices.

For technical details on the sampling algorithms and their tuning parameters, see the help file for `MfU.Sample`, as well as the aforementioned publications or statistical textbooks (Robert and Casella 1999).

Gibbs wrappers: The function `MfU.Sample` implements the extended Gibbs sampling concept (Section 2.1), using a `for` loop that applies the underlying univariate sampler to each coordinate of the multivariate distribution. The function `MfU.Control` allows the user to set the tuning parameters of the univariate sampler. `MfU.Sample.Run` is a light wrapper around `MfU.Sample` for drawing multiple samples.

Diagnostic utilities: Implementations of generic S3 methods `summary` and `plot` for '`MfU`' class objects – output of `MfU.Sample.Run` – are light wrappers around corresponding methods for the '`mcmc`' class in the R package **coda** (Plummer, Best, Cowles, and Vines 2006), with the addition of the sample-based covariance matrix, effective sample size, time, and number of independent samples per second returned by the `summary` method for '`MfU`' objects.

Full Bayesian prediction: The S3 `predict` method for '`MfU`' objects allows for sample-based reconstruction of the predictive posterior distribution for any user-supplied prediction function. The mechanics and advantages of full Bayesian prediction are discussed in the **sns** package vignette (Mahani *et al.* 2016). See Section 3.4 of this document for an example.

3. Using MfUSampler

In this section, we illustrate how **MfUSampler** can be used for building Bayesian models. We begin by introducing the data set used throughout the examples in this paper. This is followed by the illustration of how univariate samplers can be readily applied to sample from the posterior distribution of our problem. Application of diagnostic and prediction utility functions are illustrated last.

Before proceeding, we load **MfUSampler** into an R session, and select the seed value to feed to the random number generator at the beginning of each code segment (for reproducibility), and the number of MCMC samples to collect in each run:

```
R> library("MfUSampler")
R> my.seed <- 0
R> nsmp <- 1000
```

3.1. Diabetic retinopathy data set

This data set is a 2×8 contingency table, containing the number of occurrences of diabetic retinopathy for patients with 8 different durations of diabetes. The tabular form of the data set can be found in Knuiman and Speed (1988).

The mid-point of diabetes duration bands are encoded in the vector `z` below, while the number of patients with/without retinopathy are encoded in `m1` and `m2` vectors: The `prior` suffix corresponds to numbers from a previous study, while `current` reflects the results of the current study.

```
R> z <- c(1, 4, 7, 10, 13, 16, 19, 24)
R> m1.prior <- c(17, 26, 39, 27, 35, 37, 26, 23)
R> m2.prior <- c(215, 218, 137, 62, 36, 16, 13, 15)
R> m1.current <- c(46, 52, 44, 54, 38, 39, 23, 52)
R> m2.current <- c(290, 211, 134, 91, 53, 42, 23, 32)
```

Following Knuiman and Speed (1988), our model assumes that the linear predictor for this grouped logistic regression problem has three variables: unit vector (corresponding to intercept), z and z^2:

```
R> X <- cbind(1, z, z^2)
```

3.2. Slice sampling from the posterior

The following function implements the log-posterior for our problem, assuming a multivariate Gaussian prior on the coefficient vector, `beta`, with mean `beta0` and covariance matrix `W`. The default values represent a non-informative – or flat – prior.

```
R> loglike <- function(beta, X, m1, m2) {
+    beta <- as.numeric(beta)
+    Xbeta <- X %*% beta
+    return(-sum((m1 + m2) * log(1 + exp(-Xbeta)) + m2 * Xbeta))
+  }
R> logprior <- function(beta, beta0, W) {
+    return(-0.5 * t(beta - beta0) %*% solve(W, beta - beta0))
+  }
R> logpost <- function(beta, X, m1, m2, beta0 = rep(0, 0, 3),
+    W = diag(1e+6, nrow = 3)) {
+      return(logprior(beta, beta0, W) + loglike(beta, X, m1, m2))
+  }
```

(Note that, for simplicity of presentation, a straight-forward, non-optimized implementation is used.) Incorporating prior information in this problem can be done in two ways: (1) extracting `beta0` and `W` from prior data (using flat priors during estimation), and feeding these numbers as priors for estimating the model with current data, (2) simply adding prior and current numbers to arrive at the posterior contingency table. We choose the first option for this presentation which, as argued by Knuiman and Speed (1988), offers more flexibility

in the strength with which prior influences the posterior results. For presentation brevity, we use the prior parameters reported in Knuiman and Speed (1988), rather than estimating them from the data:

```
R> beta0.prior <- c(-3.17, 0.33, -0.007)
R> W.prior <- 1e-4 * matrix(c(638, -111, 3.9, -111, 24.1, -0.9, 3.9, -0.9,
+    0.04), ncol = 3)
```

We begin by drawing 1000 samples using the slice sampler, and printing a summary of samples (using a significance level of 0.01):

```
R> set.seed(my.seed)
R> beta.ini <- c(0.0, 0.0, 0.0)
R> beta.smp <- MfU.Sample.Run(beta.ini, logpost, nsmp = nsmp, X = X,
+    m1 = m1.current, m2 = m2.current, beta0 = beta0.prior, W = W.prior)
R> summ.slice <- summary(beta.smp, quantiles = c(0.005, 0.5, 0.995))
R> summ.slice

Iterations = 501:1000
Thinning interval = 1
Number of chains = 1
Sample size per chain = 500

1. Empirical mean and standard deviation for each variable,
   plus standard error of the mean:

          Mean       SD  Naive SE Time-series SE
[1,] -2.370096 0.156957 7.019e-03      0.0342498
[2,]  0.208465 0.032041 1.433e-03      0.0100992
[3,] -0.003701 0.001283 5.737e-05      0.0003809

2. Quantiles for each variable:

         2.5%       50%      97.5%
var1 -2.65060 -2.374412 -2.0369101
var2  0.14190  0.208619  0.2660260
var3 -0.00604 -0.003744 -0.0008024

time for all samples ( 1000 ): 1.061 sec
time assigned to selected samples ( 500 ): 0.5305 sec
Effective sample size / independent samples per sec:
          ess      iss
var1 21.00122 39.58760
var2 10.06534 18.97330
var3 11.34342 21.38251

R> summ.slice$covar
```

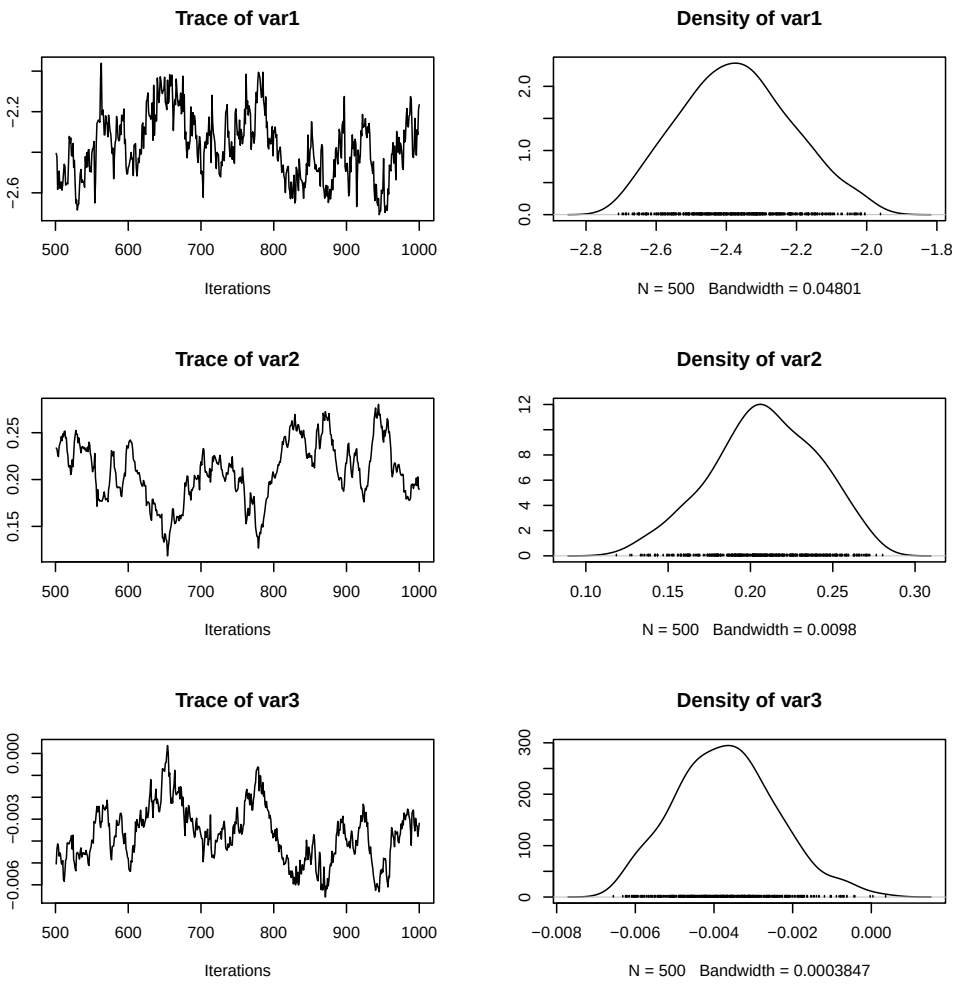

Figure 2: Output of the `plot` method for '`MfU`' objects, applied to 1000 samples drawn from the posterior distribution for the diabetic retinopathy problem, using the slice sampler. This function is a thin wrapper around the `plot` method for '`mcmc`' objects from the R package **coda**.

```
              [,1]          [,2]          [,3]
[1,]  0.0246354313 -4.457720e-03  1.574388e-04
[2,] -0.0044577203  1.026602e-03 -3.951447e-05
[3,]  0.0001574388 -3.951447e-05  1.645770e-06
```

Despite drawing 1000 nominal samples, the effective sample sizes are much smaller, due to sample auto-correlation. It would therefore be prudent to increase the number of samples, e.g., to reach 100 independent samples. Closer examination of the MCMC chains, via the command `plot(summ.slice)` can further help with diagnostics. For example, inspection of Figure 2 confirms that the chains are far more correlated than what a white-noise pattern would exhibit. The figure is generated with the following command:

```
R> plot(beta.smp)
```

3.3. Adaptive rejection sampling of the posterior

Next, we illustrate how ARS can be used for sampling from this posterior distribution. We need to implement the gradient of the log-density in order to use ARS. Furthermore, we must ensure that the distribution is log-concave, or equivalently that the Hessian of the log-density is negative-definite. It is easy to verify that this distribution satisfies this requirement. For theoretical and software support in assessing log-concavity of distributions and verifying correct implementation of their derivatives, see the vignette of the R package **sns** (Mahani *et al.* 2016).

```
R> logpost.fg <- function(beta, X, m1, m2, beta0 = rep(0.0, 3),
+    W = diag(1e+3, nrow = 3), grad = FALSE) {
+      Xbeta <- X %*% beta
+
+      if (grad) {
+        log.prior.d <- -solve(W, beta - beta0)
+        log.like.d <- t(X) %*% ((m1 + m2) / (1 + exp(Xbeta)) - m2)
+        return(log.prior.d + log.like.d)
+      }
+
+      log.prior <- -0.5 * t(beta - beta0) %*% solve(W, beta - beta0)
+      log.like <- -sum((m1 + m2) * log(1 + exp(-Xbeta)) + m2 * Xbeta)
+      log.post <- log.prior + log.like
+
+      return(log.post)
+  }
```

Note the use of the mandatory Boolean flag `grad`, indicating whether the log-density or its gradient are returned. Next we feed this log-density to `MfU.Sample.Run`. Please note that even after fixing the random seed, the results (also those not about the timings) are not exactly reproducible. We suspect that the samples drawn are getting increasingly different due to slight differences in floating-point arithmetics for different platforms which accumulate over the sampling. Note that time-dependent results (e.g. effective sampling rate or speedups) cannot be replicated exactly for obvious reasons.

```
R> set.seed(my.seed)
R> beta.ini <- c(0.0, 0.0, 0.0)
R> beta.smp <- MfU.Sample.Run(beta.ini, logpost.fg, nsmp = nsmp,
+    uni.sampler = "ars", control = MfU.Control(3, ars.x = list(
+    c(-10, 0, 10), c(-1, 0, 1), c(-0.1, 0.0, 0.1))), X = X,
+    m1 = m1.current, m2 = m2.current, beta0 = beta0.prior, W = W.prior)
R> summ.ars <- summary(beta.smp)
R> summ.ars

Iterations = 501:1000
Thinning interval = 1
Number of chains = 1
Sample size per chain = 500
```

```
1. Empirical mean and standard deviation for each variable,
   plus standard error of the mean:

          Mean       SD  Naive SE Time-series SE
[1,] -2.385601 0.126267 5.647e-03      0.0271520
[2,]  0.212728 0.024940 1.115e-03      0.0067442
[3,] -0.003859 0.001014 4.535e-05      0.0002849

2. Quantiles for each variable:

          2.5%       50%    97.5%
var1 -2.653789 -2.382039 -2.15136
var2  0.167205  0.210852  0.26317
var3 -0.005932 -0.003832 -0.00187

time for all samples ( 1000 ): 2.089 sec
time assigned to selected samples ( 500 ): 1.0445 sec
Effective sample size / independent samples per sec:
          ess      iss
var1 21.62600 20.70464
var2 13.67511 13.09249
var3 12.66920 12.12944
```

Note that we have provided custom values for the control parameter `ars.x`. For this problem, the ARS algorithm is sensitive to these initial values, and can fail to identify log-concavity of the distribution in some cases. Generally, we have found the slice sampler to require less tuning to achieve reasonable performance. On the other hand, for a suitable choice of tuning parameters, ARS can outperform the slice sampler, measured in terms of 'independent samples per second'. Interested readers can see examples of using other samplers included in **MfUSampler** – namely ARMS and univariate Metropolis – by typing `?MfU.Sample` in the R session.

As in Section 3.2, effective sample sizes are small, and suggest that increasing the nominal sample size is prudent. For example, increasing the number of samples to 10,000 will cause the coefficient means estimated using slice sampler and ARS to consistently reproduce values reported in Knuiman and Speed (1988) and Dellaportas and Smith (1993).

3.4. Full Bayesian prediction

The `predict` function in the **MfUSampler** package can be used to do sample-based reconstruction of arbitrary functions of model parameters. This includes deterministic as well as stochastic functions. For example, assume we want to know the probability distribution of the probability of retinopathy for each value of z in our training set. The prediction function has the following simple form:

```
R> predfunc.mean <- function(beta, X) {
+    return(1 / (1 + exp(-X %*% beta)))
+  }
```

We can now generate samples for this predicted quantity:

```
R> pred.mean <- predict(beta.smp, predfunc.mean, X)
R> predmean.summ <- summary(pred.mean)
R> print(predmean.summ, n = 8)

prediction sample statistics:
        (nominal sample size: 500)
        mean         sd        ess       2.5%        50%  97.5%
1 1.0227e-01 9.7785e-03 2.9238e+01 8.3478e-02 1.0193e-01 0.1215
2 1.6870e-01 9.5854e-03 1.9909e+02 1.5035e-01 1.6877e-01 0.1886
3 2.5264e-01 1.2029e-02 1.7275e+02 2.2917e-01 2.5282e-01 0.2764
4 3.4448e-01 1.6707e-02 3.7510e+01 3.1099e-01 3.4426e-01 0.3768
5 4.3245e-01 2.0333e-02 4.1497e+01 3.9308e-01 4.3232e-01 0.4711
6 5.0751e-01 2.2610e-02 1.1673e+02 4.6337e-01 5.0769e-01 0.5516
7 5.6522e-01 2.6223e-02 5.0000e+02 5.1204e-01 5.6594e-01 0.6148
8 6.2085e-01 4.2244e-02 1.5088e+02 5.3316e-01 6.2234e-01 0.7033
```

We can also ask a different question: What is the distribution of the occurrence of retinopathy (as a binary variable) in each given band of diabetes duration. The prediction function is a slight modification of the previous one:

```
R> predfunc.binary <- function(beta, X) {
+    return(as.numeric(runif(nrow(X)) < 1/(1 + exp(-X %*% beta))))
+  }
R> pred.binary <- predict(beta.smp, predfunc.binary, X)
R> predbinary.summ <- summary(pred.binary)
R> print(predbinary.summ, n = 8)

prediction sample statistics:
        (nominal sample size: 500)
       mean        sd       ess      2.5%       50% 97.5%
1   0.12200   0.32761 500.00000   0.00000   0.00000     1
2   0.15800   0.36511 500.00000   0.00000   0.00000     1
3   0.24800   0.43228 500.00000   0.00000   0.00000     1
4   0.32800   0.46996 400.26263   0.00000   0.00000     1
5   0.41400   0.49304 500.00000   0.00000   0.00000     1
6   0.49000   0.50040 423.67823   0.00000   0.00000     1
7   0.58600   0.49304 550.23479   0.00000   1.00000     1
8   0.64200   0.47989 500.00000   0.00000   1.00000     1
```

We see that the mean values from the two predictions are close, and in the limit of infinite samples they will converge towards the same values. However, the SD numbers are much larger for the binary prediction as it combines the uncertainty of estimating the coefficients, with the uncertainty of the stochastic process that generates the (binary) outcome. The value of full Bayesian prediction, particularly in business and decision-making settings, is that it

combines these two sources of uncertainty to provide the user with a full representation of uncertainty in estimating actual outcomes, and not just mean/expected values.

4. Performance improvement

Applying `MfU.Sample.Run` to the full joint PDF of a Bayesian model, implemented in R, is often a good starting point. For small data sets, this may well be sufficient. For example, in the diabetic retinopathy data set described in Section 3, and using an average laptop, we should be able to draw 10,000 samples from the posterior distribution in a few minutes. However, for large data sets we must look for opportunities to improve the performance. In this section, we describe two general strategies for speeding up sampling of posterior distributions within the framework of **MfUSampler**: (1) utilizing the structure of the underlying model graph, and (2) high-performance evaluation of the posterior function. We describe these two strategies using extensions of the diabetic retinopathy data set.

4.1. Diabetic retinopathy: Hierarchical Bayesian with continuous z

To illustrate the performance optimization strategies discussed in this section, we extend the diabetic retinopathy data set – by simulations – to define a hierarchical Bayes (HB) problem with continuous z. In other words, we turn the grouped logistic regression into a standard logistic regression, where the outcome is not frequency of occurrence of retinopathy within a diabetic duration band (with mid-point z), but a binary indicator for each continuous value of z. We generate coefficients for 5 observation groups (`ngrp = 5`) from the multivariate Gaussian prior discussed in the last section (numbers from Knuiman and Speed 1988 are used), and simulate binary outcome in each group based on its coefficients. The number of observations per group can be adjusted via the parameter `nrep`. z values are sampled – with replacement – from real data, with the addition of a small random jitter. The data generation code is as follows:

```
R> library("mvtnorm")
R> set.seed(my.seed)
R> nrep <- 50
R> m.current <- m1.current + m2.current
R> nz.exp <- nrep * sum(m.current)
R> jitter <- 1.0
R> z.exp <- sample(z, size = nz.exp, replace = TRUE, prob = m.current) +
+    (2 * runif(nz.exp) - 1) * jitter
R> X.exp <- cbind(1, z.exp, z.exp^2)
R> ngrp <- 5
R> beta.mat <- t(rmvnorm(ngrp, mean = beta0.prior, sigma = W.prior))
R> y.mat.exp <- matrix(as.numeric(runif(ngrp * nz.exp) <
+    1 / (1 + exp(-X.exp %*% beta.mat))), ncol = ngrp)
```

In real problems, `ngrp` must typically be larger than 5 to justify the pooling of information across groups. But here we limit ourselves to this small number to keep code execution times relatively small.

4.2. Utilizing graph structure

In directed acyclic graphs, the joint distribution can be factorized into the product of conditional distributions for all nodes, conditioned on parent nodes of each node (Bishop 2006). For undirected graphs, factorization can be done over maximal cliques of the graph. When sampling from conditional distribution of a variable – conditioned on all remaining variables – as is done during Gibbs sampling, not all such multiplicative factors depend on the variable being sampled, and can be safely ignored during evaluation of the conditional distribution and its derivatives. In some cases, the resulting time savings can be quite significant, with a prime example being the hierarchical Bayesian models (Gelman and Hill 2006). In HB models, the conditional distribution of the low-level coefficient vector for each group during Gibbs sampling contains multiplicative contributions from other groups. This is reflected in the following log-posterior functions for the coefficients of all groups that contain one additive term per group:

```
R> hb.logprior <- function(beta.flat, beta0, W) {
+    beta.mat <- matrix(beta.flat, nrow = 3)
+    return(sum(apply(beta.mat, 2, logprior, beta0, W)))
+  }
R> hb.loglike <- function(beta.flat, X, y) {
+    beta.mat <- matrix(beta.flat, nrow = 3)
+    ngrp <- ncol(beta.mat)
+    return(sum(sapply(1:ngrp, function(n) {
+      xbeta <- X %*% beta.mat[, n]
+      return(-sum((1 - y[, n]) * xbeta + log(1 + exp(-xbeta))))
+    })))
+  }
R> hb.logpost <- function(beta.flat, X, y, beta0, W) {
+    return(hb.logprior(beta.flat, beta0, W) +
+      hb.loglike(beta.flat, X, y))
+  }
```

A naive implementation of the full PDF is thus pointlessly duplicating computations by `ngrp` times:

```
R> nsmp <- 10
R> set.seed(my.seed)
R> beta.flat.ini <- rep(0.0, 3 * ngrp)
R> beta.flat.smp <- MfU.Sample.Run(beta.flat.ini, hb.logpost, X = X.exp,
+    y = y.mat.exp, beta0 = beta0.prior, W = W.prior, nsmp = nsmp)
R> t.naive <- attr(beta.flat.smp, "t")
R> cat("hb sampling time - naive method:", t.naive, "sec\n")

hb sampling time - naive method: 43.635 sec
```

Note that, in the above, we have made the simplifying assumption that we know the true values of the parameters of the multivariate Gaussian prior, i.e., `beta0.prior` and `W.prior`. In reality, of course, prior parameters must also be estimated from the data, and thus the

Gibbs cycle includes not just the lower-level coefficients but also the prior parameters. (Interested readers can consult the literature on 'eliciting prior distributions'. See, e.g., O'Hagan 1998.) However, all the strategies discussed in this section can be conceptually illustrated while focusing only on sampling `beta`.

The first optimization strategy is to take advantage of the conditional independence property by evaluating only the relevant term during sampling of coefficients in each group:

```
R> hb.loglike.grp <- function(beta, X, y) {
+    beta <- as.numeric(beta)
+    xbeta <- X %*% beta
+    return(-sum((1 - y) * xbeta + log(1 + exp(-xbeta))))
+  }
R> hb.logprior.grp <- logprior
R> hb.logpost.grp <- function(beta, X, y, beta0 = rep(0.0, 3),
+    W = diag(1e+6, nrow = 3)) {
+      return(hb.logprior.grp(beta, beta0, W) +
+        hb.loglike.grp(beta, X, y))
+  }
```

The price to pay is that we must implement a custom `for` loop to replace `MfU.Sample.Run`:

```
R> set.seed(my.seed)
R> beta.mat.buff <- matrix(rep(0.0, 3 * ngrp), nrow = 3)
R> beta.mat.smp <- array(NA, dim = c(nsmp, 3, ngrp))
R> t.revised <- proc.time()[3]
R> for (i in 1:nsmp) {
+    for (n in 1:ngrp) {
+      beta.mat.buff[, n] <- MfU.Sample(beta.mat.buff[, n], hb.logpost.grp,
+        uni.sampler = "slice", X = X.exp, y = y.mat.exp[, n],
+        beta0 = beta0.prior, W = W.prior)
+    }
+    beta.mat.smp[i, , ] <- beta.mat.buff
+  }
R> t.revised <- proc.time()[3] - t.revised
R> cat("hb sampling time - revised method:", t.revised, "sec\n")

hb sampling time - revised method: 8.064 sec

R> cat("incremental speedup:", t.naive / t.revised, "\n")

incremental speedup: 5.411086
```

As expected, this revised approach produces a speedup factor that is close to `ngrp`, i.e., ∼5. (The fact that actual speedup is slightly higher than 5 is interesting and suggests other factors such as cache utilization might be at play. Studying these computational factors are beyond the scope of this paper. Interested readers can see Mahani and Sharabiani (2015) for a thorough analysis of performance optimization techniques for Bayesian MCMC.

Another implication of conditional independence for HB models is that the conditional distribution for coefficients of each group does not include the coefficients of other groups. This can be verified by examining `hb.logpost.grp`. As such, it is mathematically valid to sample coefficients of all groups concurrently, while conditioning all distributions on values of the remaining variables (Mahani and Sharabiani 2015). We can use the **doParallel** package for multi-core parallelization (Revolution Analytics and Weston 2015):

```
R> library("doParallel")
R> ncores <- 2
R> registerDoParallel(ncores)
R> set.seed(my.seed)
R> beta.mat.buff <- matrix(rep(0.0, 3 * ngrp), nrow = 3)
R> beta.mat.smp <- array(NA, dim = c(nsmp, 3, ngrp))
R> t.parallel <- proc.time()[3]
R> for (i in 1:nsmp) {
+    beta.mat.buff <- foreach(n = 1:ngrp, .combine = cbind,
+      .options.multicore = list(preschedule = TRUE)) %dopar% {
+      MfU.Sample(beta.mat.buff[, n], hb.logpost.grp,
+        uni.sampler = "slice", X = X.exp, y = y.mat.exp[, n],
+        beta0 = beta0.prior, W = W.prior)
+    }
+    beta.mat.smp[i, , ] <- beta.mat.buff
+  }
R> t.parallel <- proc.time()[3] - t.parallel
R> cat("hb sampling time - revised & parallel method:", t.parallel, "sec\n")

hb sampling time - revised & parallel method: 5.42 sec

R> cat("incremental speedup:", t.revised / t.parallel, "\n")

incremental speedup: 1.487823
```

In the above, we have continued to use the default R random number generator for code brevity. In practice, in order to generate uncorrelated random numbers across multiple execution threads, one should use parallel RNG streams such as those provided in the R package **rstream** (Leydold 2015). Also, note that the parallelization speedup does not equal the number of cores used, i.e., 2. We expect the speedup to improve as we increase `ngrp`. It is also reasonable to expect that multi-threading in R has significantly larger overhead, compared to doing so in a high-performance language such as C.

4.3. High-performance PDF evaluation

For most MCMC algorithms, the majority of sampling time is spent on evaluating the log-density (and its derivatives if needed). Efficient implementation of functions responsible for log-density evaluation is therefore a rewarding optimization strategy which can be combined with the strategies discussed in Section 4.2. The **Rcpp** (Eddelbuettel and François 2011)

framework offers a convenient way to port R functions to C++. Here we use the **RcppArmadillo** (Eddelbuettel and Sanderson 2014) package for its convenient matrix algebra operations to transform the log-likelihood component of the log-posterior (as it takes the majority of time for large data, compared to the log-prior):

```
R> library("RcppArmadillo")
R> library("inline")
R> code <- "
+    arma::vec beta_cpp = Rcpp::as<arma::vec>(beta);
+    arma::mat X_cpp = Rcpp::as<arma::mat>(X);
+    arma::vec y_cpp = Rcpp::as<arma::vec>(y);
+    arma::vec xbeta = X_cpp * beta_cpp;
+    int n = X_cpp.n_rows;
+    double logp = 0.0;
+    for (int i = 0; i < n; i++) {
+      logp -= (1.0 - y_cpp[i]) * xbeta[i] + log(1.0 + exp(-xbeta[i]));
+    }
+    return Rcpp::wrap(logp);
+  "
R> hb.loglike.grp.rcpp <- cxxfunction(
+    signature(beta = "numeric", X = "numeric", y = "numeric"),
+    code, plugin = "RcppArmadillo")
R> hb.logpost.grp.rcpp <- function(beta, X, y, beta0 = rep(0.0, 3),
+    W = diag(1e+6, nrow = 3)) {
+    return(hb.logprior.grp(beta, beta0, W) +
+      hb.loglike.grp.rcpp(beta, X, y))
+  }
```

We simply replace `hb.logpost.grp` with `hb.logpost.grp.rcpp` in the parallel sampling approach from the previous section:

```
R> set.seed(my.seed)
R> beta.mat.buff <- matrix(rep(0.0, 3 * ngrp), nrow = 3)
R> beta.mat.smp <- array(NA, dim = c(nsmp, 3, ngrp))
R> t.rcpp <- proc.time()[3]
R> for (i in 1:nsmp) {
+    beta.mat.buff <- foreach(n = 1:ngrp, .combine = cbind,
+      .options.multicore = list(preschedule = TRUE)) %dopar% {
+      MfU.Sample(beta.mat.buff[, n], hb.logpost.grp.rcpp,
+        uni.sampler = "slice", X = X.exp, y = y.mat.exp[, n],
+        beta0 = beta0.prior, W = W.prior)
+    }
+    beta.mat.smp[i, , ] <- beta.mat.buff
+  }
R> t.rcpp <- proc.time()[3] - t.rcpp
R> cat("hb sampling time - revised & parallel & rcpp method:", t.rcpp,
+    "sec\n")
```

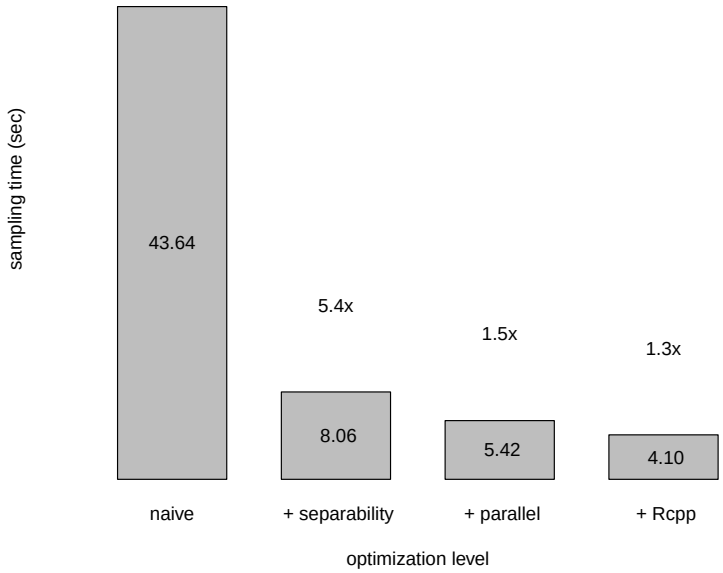

Figure 3: Time needed to draw 1000 samples for the HB logistic regression problem, based on the diabetic retinopathy data set introduced in Section 3.1, at various stages of optimization. Each step represents the cumulative effect of strategies, starting with the left-most bar corresponding to the naive implementation. Numbers above bars show speedup due to each optimization.

```
hb sampling time - revised & parallel & rcpp method: 4.105 sec
```

```
R> cat("incremental speedup:", t.parallel / t.rcpp, "\n")
```

```
incremental speedup: 1.320341
```

While the result is a decent speedup given the relatively small effort put in, yet the impact is not as significant as the previous two strategies. It must be noted that matrix algebra operations in R are handled by **BLAS** and **LAPACK** libraries, written in C and Fortran. Therefore, the major benefit of porting the log-likelihood function to C++ in the above example is likely to be the consolidation of data and control transfer between the interpretation layer and the computational back-end. For large problems, even parallel hardware such as Graphic Processing Units (GPUs) can be utilized by writing log-density functions in languages such as CUDA (Nickolls, Buck, Garland, and Skadron 2008), while continuing to take advantage of **MfUSampler** for sampler control logic. Minimizing data movement between processor and co-processor is a key performance factor in such cases.

An even easier method for improving the performance of log-density evaluation function is compiling it to byte code, using the **compiler** package (`cmpfun` function). However, testing this approach on our logistic regression problem resulted in very modest speedup (less than 10%).

Figure 3 summarizes the impact of the three optimization strategies discussed in this section. Combining all three optimization strategies (while using only 2 cores for parallelization) has provided a significant speedup over the naive approach:

```
R> cat("combined speedup:", t.naive / t.rcpp, "\n")

combined speedup: 10.62972
```

In addition to the above-mentioned strategies, there are several other options available for improving performance of MCMC sampling techniques for Bayesian models. Examples include differential update, single-instruction multiple-data (SIMD) parallelization of log-likelihood calculation, and batch random number generation. For a detailed discussion of these topics, see Mahani and Sharabiani (2015).

5. Summary

The R package **MfUSampler** enables MCMC sampling of multivariate distributions using univariate algorithms. It relies on an extension of Gibbs sampling from univariate independent sampling to univariate Markov transitions, and proportionality of conditional and joint distributions. By encapsulating these two concepts in a package, **MfUSampler** reduces the possibility of subtle mistakes by researchers while re-implementing the Gibbs sampler and thus allows them to focus on other, more innovative aspects of their Bayesian modeling. Brute-force application of **MfUSampler** allows researchers to get their project off the ground, maintain full control over model specification, and utilize robust univariate samplers. This can be followed by an incremental optimization approach by taking advantage of DAG properties such as conditional independence and by porting log-density functions to high-performance languages and hardware. As of this writing, **MfUSampler** has been utilized in the R package **BSGW** (Mahani and Sharabiani 2016) for Gibbs sampling of posteriors in dynamic Bayesian survival models, and in **HBglm** (Hasan and Mahani 2015) for hierarchical Bayesian generalized linear regression models.

Computational details

All R code shown in this paper was executed on an Intel Xeon W3680, with a CPU clock rate of 3.33GHz and 24GB of installed RAM. Below is the corresponding R session information, obtained using the command `sessionInfo()`:

```
R version 3.3.3 (2017-03-06)
Platform: x86_64-redhat-linux-gnu (64-bit)
Running under: Amazon Linux AMI 2017.03

locale:
 [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
 [3] LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
 [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
 [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
 [9] LC_ADDRESS=C               LC_TELEPHONE=C
[11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C

attached base packages:
```

```
[1] methods   parallel  stats     graphics  grDevices utils     datasets
[8] base

other attached packages:
[1] inline_0.3.14             RcppArmadillo_0.7.900.2.0
[3] doParallel_1.0.10         iterators_1.0.8
[5] foreach_1.4.3             mvtnorm_1.0-6
[7] MfUSampler_1.0.4

loaded via a namespace (and not attached):
[1] compiler_3.3.3   ars_0.5          HI_0.4           coda_0.19-1
[5] Rcpp_0.12.11     codetools_0.2-15 grid_3.3.3       lattice_0.20-34
```

Acknowledgments

The authors wish to thank the JSS editors and reviewers for their thorough and constructive feedback, resulting in significant improvements in the quality of our software and manuscript.

References

Bishop CM (2006). *Pattern Recognition and Machine Learning*, volume 1. Springer-Verlag.

Carpenter B, Gelman A, Hoffman M, Lee D, Goodrich B, Betancourt M, Brubaker M, Guo J, Li P, Riddell A (2017). "Stan: A Probabilistic Programming Language." *Journal of Statistical Software*, **76**(1), 1–32. doi:10.18637/jss.v076.i01.

Christen JA, Fox C (2010). "A General-Purpose Sampling Algorithm for Continuous Distributions (the t-Walk)." *Bayesian Analysis*, **5**(2), 263–281. doi:10.1214/10-ba60.

Dellaportas P, Smith AFM (1993). "Bayesian Inference for Generalized Linear and Proportional Hazards Models via Gibbs Sampling." *Journal of the Royal Statistical Society C*, **42**(3), 443–459. doi:10.2307/2986324.

Eddelbuettel D, François R (2011). "**Rcpp**: Seamless R and C++ Integration." *Journal of Statistical Software*, **40**(8), 1–18. doi:10.18637/jss.v040.i08.

Eddelbuettel D, Sanderson C (2014). "**RcppArmadillo**: Accelerating R with High-Performance C++ Linear Algebra." *Computational Statistics & Data Analysis*, **71**, 1054–1063. doi:10.1016/j.csda.2013.02.005.

Gelman A, Hill J (2006). *Data Analysis Using Regression and Multilevel/Hierarchical Models*. Cambridge University Press.

Geman S, Geman D (1984). "Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **PAMI-6**(6), 721–741. doi:10.1109/tpami.1984.4767596.

Gilks WR, Best NG, Tan KKC (1995). "Adaptive Rejection Metropolis Sampling within Gibbs Sampling." *Journal of the Royal Statistical Society C*, **44**(4), 455–472. doi:10.2307/2986138.

Gilks WR, Neal RM, Best NG, Tan KKC (1997). "Corrigendum: Adaptive Rejection Metropolis Sampling." *Journal of the Royal Statistical Society C*, **46**(4), 541–542. doi:10.1111/1467-9876.00091.

Gilks WR, Wild P (1992). "Adaptive Rejection Sampling for Gibbs Sampling." *Journal of the Royal Statistical Society C*, **41**(2), 337–348. doi:10.2307/2347565.

Girolami M, Calderhead B (2011). "Riemann Manifold Langevin and Hamiltonian Monte Carlo Methods." *Journal of the Royal Statistical Society B*, **73**(2), 123–214. doi:10.1111/j.1467-9868.2010.00765.x.

Hasan A, Mahani AS (2015). ***HBglm**: Hierarchical Bayesian Regression for GLMs*. R package version 0.1, URL https://CRAN.R-project.org/package=HBglm.

Hastings WK (1970). "Monte Carlo Sampling Methods Using Markov Chains and Their Applications." *Biometrika*, **57**(1), 97–109. doi:10.2307/2334940.

Hoffman MD, Gelman A (2014). "The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo." *Journal of Machine Learning Research*, **15**, 1593–1623.

Jarner SF, Hansen E (2000). "Geometric Ergodicity of Metropolis Algorithms." *Stochastic Processes and Their Applications*, **85**(2), 341–361. doi:10.1016/s0304-4149(99)00082-4.

Knuiman M, Speed T (1988). "Incorporating Prior Information into the Analysis of Contingency Tables." *Biometrics*, **44**(4), 1061–1071. doi:10.2307/2531735.

Leydold J (2015). ***rstream**: Streams of Random Numbers*. R package version 1.3.3, URL https://CRAN.R-project.org/package=rstream.

Mahani AS, Hasan A, Jiang M, Sharabiani MTA (2016). *Stochastic Newton Sampler: The R Package **sns***. doi:10.18637/jss.v074.c02.

Mahani AS, Sharabiani MTA (2015). "SIMD Parallel MCMC Sampling with Applications for Big-Data Bayesian Analytics." *Computational Statistics & Data Analysis*, **88**, 75–99. doi:10.1016/j.csda.2015.02.010.

Mahani AS, Sharabiani MTA (2016). ***BSGW**: Bayesian Survival Model with Lasso Shrinkage Using Generalized Weibull Regression*. R package version 0.9.2, URL https://CRAN.R-project.org/package=BSGW.

Mahani AS, Sharabiani MTA (2017). ***MfUSampler**: Multivariate-from-Univariate (MfU) MCMC Sampler*. R package version 1.0.4, URL https://CRAN.R-project.org/package=MfUSampler.

Metropolis N, Rosenbluth AW, Rosenbluth MN, Teller AH, Teller E (1953). "Equation of State Calculations by Fast Computing Machines." *The Journal of Chemical Physics*, **21**(6), 1087–1092. doi:10.1063/1.1699114.

Neal R (2011). "MCMC Using Hamiltonian Dynamics." *Handbook of Markov Chain Monte Carlo*, **2**, 113–162. doi:10.1201/b10905-6.

Neal RM (2003). "Slice Sampling." *The Annals of Statistics*, **31**(3), 705–741. doi:10.1214/aos/1056562461.

Nickolls J, Buck I, Garland M, Skadron K (2008). "Scalable Parallel Programming with CUDA." *Queue*, **6**(2), 40–53. doi:10.1145/1365490.1365500.

O'Hagan A (1998). "Eliciting Expert Beliefs in Substantial Practical Applications." *Journal of the Royal Statistical Society D*, **47**(1), 21–35. doi:10.1111/1467-9884.00114.

Pérez-Rodríguez P, Wild P, Gilks W (2014). ***ars**: Adaptive Rejection Sampling*. R package version 0.5, URL https://CRAN.R-project.org/package=ars.

Petris G, Tardella L, Gilks WR (2013). ***HI**: Simulation from Distributions Supported by Nested Hyperplanes*. R package version 0.4, URL https://CRAN.R-project.org/package=HI.

Plummer M (2003). "**JAGS**: A Program for Analysis of Bayesian Graphical Models Using Gibbs Sampling." In K Hornik, F Leisch, A Zeileis (eds.), *Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003)*. Vienna. URL https://www.R-project.org/conferences/DSC-2003/Proceedings/Plummer.pdf.

Plummer M, Best N, Cowles K, Vines K (2006). "**coda**: Convergence Diagnosis and Output Analysis for MCMC." *R News*, **6**(1), 7–11. URL http://CRAN.R-project.org/doc/Rnews.

Qi Y, Minka TP (2002). "Hessian-Based Markov Chain Monte-Carlo Algorithms." Unpublished Manuscript, URL https://www.cs.purdue.edu/homes/alanqi/papers/qi-minka-HMH-AMIT-02.ps.

Revolution Analytics, Weston S (2015). ***doParallel**: Foreach Parallel Adaptor for the **parallel** Package*. R package version 1.0.10, URL https://CRAN.R-project.org/package=doParallel.

Robert CP, Casella G (1999). *Monte Carlo Statistical Methods*. Springer-Verlag.

Roberts GO, Rosenthal JS (1999). "Convergence of Slice Sampler Markov Chains." *Journal of the Royal Statistical Society B*, **61**(3), 643–660. doi:10.1111/1467-9868.00198.

Stan Development Team (2017). "Stan: A C++ Library for Probability and Sampling, Version 2.14.0." URL http://mc-stan.org/.

Thomas A, O'Hara B, Ligges U, Sturtz S (2006). "Making BUGS Open." *R News*, **6**(1), 12–17. URL http://CRAN.R-project.org/doc/Rnews.

Thompson MB (2011). *Slice Sampling with Multivariate Steps*. Ph.D. thesis, University of Toronto.

A. Proof of extended Gibbs sampling lemma

The premise can be mathematically expressed as

$$p(x'_k|\mathbf{x}_{\setminus k}) = \int_{x_k} T(x'_k, x_k|\mathbf{x}_{\setminus k}) p(x_k|\mathbf{x}_{\setminus k}) \, \mathrm{d}x_k, \tag{2}$$

while the conclusion can be expressed as

$$p(x'_k, \mathbf{x}_{\setminus k}) = \int_{x_k} T(x'_k, x_k|\mathbf{x}_{\setminus k}) p(x_k, \mathbf{x}_{\setminus k}) \, \mathrm{d}x_k. \tag{3}$$

In the above $\mathbf{x}_{\setminus k}$ denotes all coordinates except for x_k and $T(x'_k, x_k|\mathbf{x}_{\setminus k})$ denotes the coordinate-wise Markov transition density from x'_k to x_k. Employing the product rule of probability, we have $p(x_k, \mathbf{x}_{\setminus k}) = p(x_k|\mathbf{x}_{\setminus k}) \times p(\mathbf{x}_{\setminus k})$. Since the coordinate-wise Markov transition does not change $\mathbf{x}_{\setminus k}$, we can factor $p(\mathbf{x}_{\setminus k})$ out of the integral, thereby easily reducing Equation 3 to Equation 2.

Note that standard Gibbs sampling is a special case of the above lemma where $T(x'_k, x_k|\mathbf{x}_{\setminus k}) = p(x'_k|\mathbf{x}_{\setminus k})$. The reader can easily verify that this special transition density satisfies the premise.

Affiliation:

Alireza S. Mahani
Scientific Computing Group
Sentrana Inc.
1725 I St NW
Washington, DC 20006, United States of America
E-mail: alireza.mahani@sentrana.com

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
June 2017, Volume 78, Code Snippet 1 *Submitted:* 2014-12-28
doi:10.18637/jss.v078.c01 *Accepted:* 2016-05-22