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

Alireza S. Mahani
Scientific Computing Group
Sentrana Inc.

Mansour T.A. Sharabiani
National Heart and Lung Institute
Imperial College London

Abstract

The R package **MfUSampler** provides Monte Carlo Markov Chain machinery for generating samples from multivariate probability distributions using univariate sampling algorithms such as slice sampler and 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, **MfUSampler** provides a reliable library for rapid prototyping of custom Bayesian models while allowing for incremental performance optimizations such as utilization of conjugacy, conditional independence, and porting function evaluations to compiled languages. Utility functions for MCMC diagnostics as well as sample-based construction of predictive posterior distributions are provided in **MfUSampler**.

Keywords: monte carlo markov chain, slice sampler, adaptive rejection sampler, gibbs sampling, Metropolis.

1. Introduction

Bayesian inference software such as Stan (Stan Development Team 2014), OpenBUGS (Thomas, O'Hara, Ligges, and Sturtz 2006), and JAGS (Plummer 2004) 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 Monte Carlo Markov Chain (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 2015). It can be quite effective for twice-differentiable, log-concave distributions such as those encountered in generlized 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 [better description]. 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 log-density to be concave, and needs its first derivative, while 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 library 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, current version of **MfUSampler** (1.0.0) contains adaptive rejection Metropolis sampler (Gilks, Best, and Tan 1995) and univariate Metropolis with Gaussian proposal. Univariate samplers have their limits: when posterior distribution exhibits 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 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 a brief overview of the extended Gibbs sampling framework used in **MfUSampler**. In Section 3 we illustrate how to use the software with an example. Section 4 shows how **MfUSampler** discussed several performance optimization techniques that can be used in conjunction with **MfUSampler**. Finally, Section 5 provides a summary and concluding remarks.

2. Theory and Implementation of 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 in 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**, adaptive rejection sampler produces a standard Gibbs sampler while the remaining samplers falls 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.*

Proof is given in Appendix B. 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, same is true of the resulting composite Markov transition density.

2.2. Proportionality of conditional and joint distributions

Using univariate samplers within Gibbs sampling framework requires access to conditional distributions, up to a multiplicative constant (in terms of coordinate being sampled). Referring to conditional distribution for the k'th 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 porportional. 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. We describe each component below. Figure 1 provides an overview of how these components fit in the overall process flow of **MfUSampler**. Below we describe each component in detail.

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`.

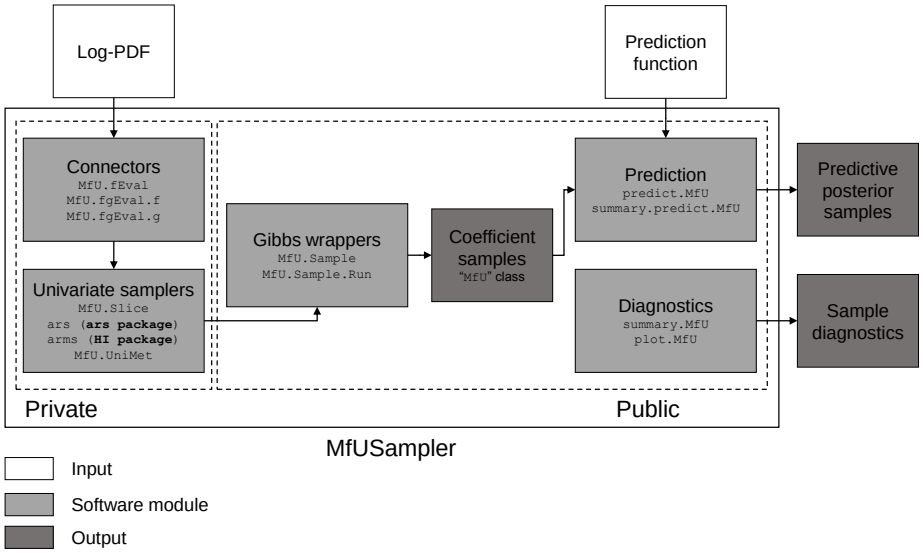

Figure 1: Software component and process flow for **MfUSampler**. Connector and sampler private modules mediate Gibbs sampling of user-supplied PDF.

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[1]. 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 (Gilks and Wild 1992), imported from R package **ars** (Perez-Rodriguez, Wild, and Gilks 2014). ARS requires log-density to be concave, and its gradient. Our experience shows that it is somewhat more sensitive to the choice of tuning parameters, compared to slice sampler (Section 3.3).

3. Adpative rejection Metropolis sampler (Gilks *et al.* 1995), imported from 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).

4. Univariate Metropolis with Gaussian proposal, implemented by **MfUSampler** (function

[1] `http://www.cs.toronto.edu/~radford/ftp/slice-R-prog`

`MfU.Univariate.Metropolis`). This simple sampler can be inefficient, unless the standard deviation of 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 help file for `MfU.Sample`, as well as 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 - output of `MfU.Sample.Run` - are light wrappers around corresponding methods for the `mcmc` class in the R package `coda`, with the addition of sample-based covariance matrix, effective sample size, time, and number of independent samples per sec.

Full Bayesian prediction The S3 method `predict.MfU` function allows for sample-based reconstruction of predictive posterior distribution for any user-supplied prediction function. The mechanics and advantages of full Bayesian prediction are discussed in the **sns** vignette (Mahani *et al.* 2015). 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 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 <- 10
```

3.1. Diabetic retinopathy data set

This data set is a 2×8 contingency table, containing the number of occurances 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 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 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))
+  }
```

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 contigency table. While the first approach can be more flexible, as argued in Knuiman and Speed (1988), here we opt for the second approach for brevity of presentation:

```
R> m1.total <- m1.prior + m1.current
R> m2.total <- m2.prior + m2.current
```

We begin by drawing 1000 samples using the slice sampler, and printing a summary of samples:

```
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.total, m2 = m2.total)
R> summ.slice <- summary(beta.smp)
```

```
R> print(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,] -2.452413 0.134392 6.010e-03      0.0409431
[2,]  0.225965 0.025717 1.150e-03      0.0089625
[3,] -0.004234 0.001026 4.589e-05      0.0003032

2. Quantiles for each variable:

          2.5%       50%     97.5%
var1 -2.686285 -2.457768 -2.196446
var2  0.172842  0.228813  0.273421
var3 -0.006002 -0.004349 -0.002109

time for all samples ( 1000 ): 2.253 sec
time assigned to selected samples ( 500 ): 1.1265 sec
Effective sample size / independent samples per sec:
          ess      iss
var1 8.120590 7.208691
var2 4.647671 4.125762
var3 6.068079 5.386666
```

Sample mean is quite close to values reported in Knuiman and Speed (1988). Similarly, the sample covariance matrix is reasonably close to their reported values:

```
R> print(summ.slice$covar)

              [,1]          [,2]          [,3]
[1,]  0.0266989860 -4.950835e-03  1.786601e-04
[2,] -0.0049508350  1.103237e-03 -4.252586e-05
[3,]  0.0001786601 -4.252586e-05  1.739642e-06
```

3.3. Adaptive rejection sampling of posterior

Next, we illustrate how ARS can be used for this posterior distribution. We need to implement the gradient of log-density in order to use ARS. Furthermore, we must ensure that the distribution is log-concave, or equivalently that the Hessian of log-density is negative-definite. It is easy to verify that this distribution satisfies our requirement. For theoretical and software support in assessing log-concavity of distributions and verifying correct implementation of their derivatives, see the vignette for the R package **sns** (Mahani *et al.* 2015).

```
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 mandatory boolean flag `grad`, indicating whether log-density or its gradient must be returned. Next we feed this log-density to `MfU.Sample.Run`:

```
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.total, m2 = m2.total)
R> summ.ars <- summary(beta.smp)

R> print(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,] -2.396224 0.175992 7.871e-03      0.0536249
[2,]  0.213187 0.032584 1.457e-03      0.0096411
[3,] -0.003726 0.001247 5.579e-05      0.0003498

2. Quantiles for each variable:
```

```
          2.5%       50%      97.5%
var1 -2.691251 -2.412357 -2.0376872
var2  0.138264  0.216288  0.2671999
var3 -0.005851 -0.003813 -0.0007741

time for all samples ( 1000 ): 3.768 sec
time assigned to selected samples ( 500 ): 1.884 sec
Effective sample size / independent samples per sec:
          ess       iss
var1 40.69967 21.602799
var2 16.90188  8.971272
var3 17.39752  9.234350
```

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. 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.

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: 5)
       mean        sd       ess      2.5%       50%  97.5%
1 0.1233438 0.0059951 5.0000000 0.1170295 0.1222361 0.1305
2 0.1742073 0.0062654 5.0000000 0.1678182 0.1725081 0.1815
3 0.2377181 0.0057428 5.0000000 0.2319015 0.2361125 0.2443
4 0.3124788 0.0042671 5.0000000 0.3078534 0.3121914 0.3173
5 0.3950574 0.0025847 5.0000000 0.3920084 0.3954314 0.3976
6 0.4805277 0.0041450 5.0000000 0.4765281 0.4799419 0.4867
7 0.5636210 0.0078931 5.0000000 0.5546627 0.5633435 0.5746
8 0.6856569 0.0139778 5.0000000 0.6689487 0.6864122 0.7042
```

We can also ask a different question: what is the distribution of percentage of population with retinopathy 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 (1*(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: 5)
      mean       sd      ess     2.5%      50% 97.5%
1  0.00000  0.00000  0.00000  0.00000  0.00000   0.0
2  0.20000  0.44721  5.00000  0.00000  0.00000   0.9
3  0.40000  0.54772  5.00000  0.00000  0.00000   1.0
4  0.40000  0.54772  5.00000  0.00000  0.00000   1.0
5  0.40000  0.54772 33.75000  0.00000  0.00000   1.0
6  0.40000  0.54772  5.00000  0.00000  0.00000   1.0
7  0.60000  0.54772 33.75000  0.00000  1.00000   1.0
8  0.60000  0.54772  5.00000  0.00000  1.00000   1.0
```

We see that 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 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 reprsentation 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, we are able to draw 10,000 samples from the posterior distribution in less than xx seconds. 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-peformance evaluation of posterior function. We describe these two strategies using extensions of the diabetic retinopathy data set. At the end of this section, we provide an overview of other performance optimization approaches, as well as pointers for further reading.

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 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 occurance of retinopathy within a diabetic duration band (z), but a binary indicator for each continuous value of z. We generate coefficients for 50 observation groups from the multivariate Gaussian prior discussed in the last section (number from Knuiman and Speed (1988) are used), and simulate binary outcome in each group based on its coefficients. 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. 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 = T, prob = m.current) +
+    (2*runif(nz.exp) - 1) * jitter
R> X.exp <- cbind(1, z.exp, z.exp^2)
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)
R> ngrp <- 50
R> beta.mat <- t(rmvnorm(ngrp, mean = beta0.prior, sigma = W.prior))
R> y.mat.exp <- 1* matrix(runif(ngrp * nz.exp) <
+    1 / (1 + exp(-X.exp %*% beta.mat)), ncol = ngrp)
```

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 involve 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 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: 53.545 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. However, all the strategies discussed in this section can be conceptually illustrated while focusing only on `beta`'s.

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`. As expected, this revised approach produces a speedup factor that approaches `ngrp`:

```
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: 9.885 sec
```

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, while conditioning all distributions on values of remaining variables (Mahani and Sharabiani 2015). We can use the `doParallel` package for multi-core parallelization.

```
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: 7.759 sec
```

In the above, we have continued to use the defualt 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).

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 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++) {
+      // (1-y[, n]) * xbeta + log(1 + exp(-xbeta))
+      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 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")

hb sampling time - revised & parallel & rcpp method: 4.847 sec
```

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 majpr 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.

Figur 2 summarizes the impact of the three optimization stragies discussed in this section. We see that, for this particular problem and set of parameters, the cumulative impact of the 3 optimization strategies is a xx times speedup.

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 library, **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

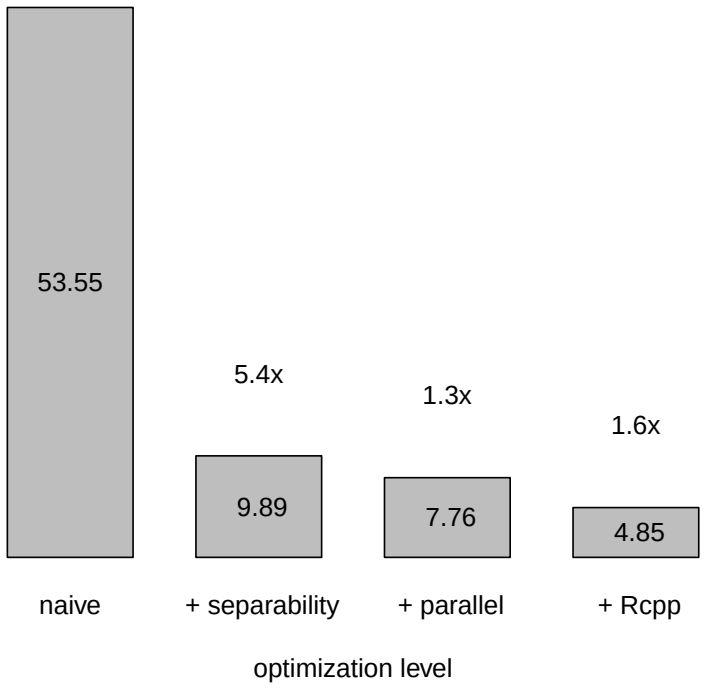

Figure 2: Time needed to draw 1000 samples for the HB logistic regression problem, based on the diabetic retinopathy data set introduced in Section 3.1, at various stages of optimization. Each step represents the cumulative effect of strategies, starting with the left-most bar corresponding to the naive implementation. Numbers above bars show speedup due to each optimization.

by an incremental optimization approach by taking advantage of DAG properties such as conditional independence and by porting log-density functions to high-peformance languages and hardware.

References

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

Christen JA, Fox C (2010). "A general purpose sampling algorithm for continuous distributions (the t-walk)." *Bayesian Analysis*, **5**(2), 263–281.

Eddelbuettel D, François R (2011). "Rcpp: Seamless R and C++ Integration." *Journal of Statistical Software*, **40**(8), 1–18. URL http://www.jstatsoft.org/v40/i08/.

Eddelbuettel D, Sanderson C (2014). "RcppArmadillo: Accelerating R with high-performance C++ linear algebra." *Computational Statistics and Data Analysis*, **71**, 1054–1063. URL http://dx.doi.org/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." *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, (6), 721–741.

Gilks WR, Best N, Tan K (1995). "Adaptive rejection Metropolis sampling within Gibbs sampling." *Applied Statistics*, pp. 455–472.

Gilks WR, Neal R, Best N, Tan K (1997). "Corrigendum: adaptive rejection metropolis sampling." *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, **46**(4), 541–542.

Gilks WR, Wild P (1992). "Adaptive Rejection Sampling for Gibbs Sampling." *Applied Statistics*, pp. 337–348.

Girolami M, Calderhead B (2011). "Riemann Manifold Langevin and Hamiltonian Monte Carlo Methods." *Journal of the Royal Statistical Society B (Statistical Methodology)*, **73**(2), 123–214.

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

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.

Knuiman M, Speed T (1988). "Incorporating prior information into the analysis of contingency tables." *Biometrics*, pp. 1061–1071.

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

Mahani AS, Hasan A, Jiang M, Sharabiani MT (2015). *sns: Stochastic Newton Sampler (SNS)*. R package version 1.1.0, URL http://CRAN.R-project.org/package=sns.

Mahani AS, Sharabiani MT (2015). "SIMD parallel MCMC sampling with applications for big-data Bayesian analytics." *Computational Statistics & Data Analysis*, **88**, 75–99.

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.

Neal R (2011). "MCMC Using Hamiltonian Dynamics." *Handbook of Markov Chain Monte Carlo*, **2**.

Neal RM (2003). "Slice sampling." *Annals of Statistics*, pp. 705–741.

Nickolls J, Buck I, Garland M, Skadron K (2008). "Scalable Parallel Programming with CUDA." *Queue*, **6**(2), 40–53. ISSN 1542-7730. doi:10.1145/1365490.1365500. URL http://doi.acm.org/10.1145/1365490.1365500.

Perez-Rodriguez P, Wild P, Gilks W (2014). *ars: Adaptive Rejection Sampling*. R package version 0.5, URL http://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 http://CRAN.R-project.org/package=HI.

Plummer M (2004). "JAGS: Just another Gibbs sampler."

Qi Y, Minka TP (2002). "Hessian-based Markov Chain Monte-Carlo Algorithms."

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 (Statistical Methodology)*, **61**(3), 643–660.

Stan Development Team (2014). "Stan: A C++ Library for Probability and Sampling, Version 2.5.0." URL http://mc-stan.org/.

Thomas A, O'Hara B, Ligges U, Sturtz S (2006). "Making BUGS Open." *R news*, **6**(1), 12–17.

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

A. Setup

All R code shown in this paper were 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.

```
R> sessionInfo()

R version 3.3.0 (2016-05-03)
Platform: x86_64-apple-darwin13.4.0 (64-bit)
Running under: OS X 10.11.6 (El Capitan)

locale:
[1] C/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods
[7] base

other attached packages:
[1] inline_0.3.14             RcppArmadillo_0.6.700.6.0
[3] mvtnorm_1.0-5             MfUSampler_1.0.3

loaded via a namespace (and not attached):
[1] ars_0.5         tools_3.3.0     HI_0.4          coda_0.18-1
[5] Rcpp_0.12.5     grid_3.3.0      lattice_0.20-33
```

B. 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 satifies the premise.

Affiliation:

Alireza S. Mahani
Scientific Computing Group

Sentrana Inc.
1725 I St NW
Washington, DC 20006
E-mail: alireza.mahani@sentrana.com