bayesGDS: an R Package for Generalized Direct Sampling

Michael Braun
SMU Cox School of Business
Southern Methodist University

December 11, 2013

Abstract

The **bayesGDS** package implements the Generalized Direct Sampling (GDS) algorithm that is presented in Braun and Damien (2013).

Generalized Direct Sampling (GDS, Braun and Damien 2013) is a method of sampling from multivariate distributions. It is particularly useful for simulating from high-dimensional, complex, bounded posterior distributions that arise from Bayesian hierarchical models. GDS is intended to replace MCMC as the preferred sampling algorithm for many, *but not all*, classes of hierarchical models, especially if the posterior is continuous and smooth. The advantages of GDS over MCMC are:

1. all samples are collected independently, so there is no need to be concerned with autocorrelation, convergence of estimations chains, and so forth;

2. there is no particular advantage to choosing model components that maintain conditional conjugacy, as is common with Gibbs sampling;

3. GDS generates samples from the target posterior entirely in parallel, which takes advantage of the most recent advances in grid computing and placing multiple CPU cores in a single computer; and

4. GDS permits fast and accurate estimation of marginal likelihoods of the data.

The derivation and justification for GDS are explained in Braun and Damien (2013) and the reader should start there for the details. The latest version of the paper is available in the doc folder in the source code of this package. The purpose of this article is to show how to sample from a posterior density in R using the **bayesGDS** package. We also highlight some practical issues in using GDS, and show an example of GDS in action.

Although the **trustOptim** (Braun 2013c), **plyr** (Wickham 2011), **sparseHessianFD** (Braun 2013a), **sparseMVN** (Braun 2013b), and **Matrix** (Bates and Maechler 2013) packages are not dependencies for **bayesGDS**, the example at the end of this note does use them. Therefore, the reader is encouraged to install those packages before proceeding.

1 The GDS Algorithm

From Braun and Damien (2013), Algorithm 1 summarizes the GDS procedure, step-by-step. Notation is consistent with that paper: $\pi(\theta|y)$ is the target posterior from which we want to sample; $\mathcal{D}(\theta, y)$ is the joint density of the data and the prior; and $g(\theta)$ is the proposal density.

2 Example: Hierarchical Binary Choice

To illustrate the use of **bayesGDS**, we will work through a simple example. Suppose there is a dataset of N households, each with T opportunities to purchase a particular product. Let y_i be the number of times household i purchases the product, out of the T purchase opportunities. Furthermore, let p_i be the probability of purchase; p_i is the same for all T opportunities, so we can treat y_i as a binomial random variable. The purchase probability p_i is heterogeneous, and depends on both k continuous

Algorithm 1 The GDS Algorithm to collect R samples from $\pi(\theta|y)$

```
1: $R \leftarrow$ number of required samples from $\pi(\theta|y)$
2: $M \leftarrow$ number of proposal draws for estimating $\widehat{q}_v(v)$.
3: $\theta^* \leftarrow$ mode of $\mathcal{D}(\theta, y)$
4: $c_1 \leftarrow \mathcal{D}(\theta^*, y)$
5: FLAG$\leftarrow$ TRUE
6: **while** FLAG **do**
7:    Choose new proposal distribution $g(\theta)$
8:    FLAG$\leftarrow$FALSE
9:    $c_2 \leftarrow g(\theta^*)$.
10:   **for** $m := 1$ **to** $M$ **do**
11:      Sample $\theta_m \sim g(\theta)$.
12:      $\log\Phi(\theta_m|y) \leftarrow \log\mathcal{D}(\theta_m, y) - \log g(\theta_m) - \log c_1 + \log c_2$.
13:      $v_m = -\log\Phi(\theta_m|y)$
14:      **if** $\log\Phi(\theta_m|y) > 0$ **then**
15:         FLAG$\leftarrow$ TRUE
16:         **break**
17:      **end if**
18:   **end for**
19: **end while**
20: Reorder elements of $v$, so $0 < v_1 < v_2 < \ldots < v_M < \infty$. Define $v_{M+1} := \infty$
21: **for** $i := 1$ **to** $M$ **do**
22:    $\widehat{q}_v(v_i) \leftarrow \sum_{j=1}^{M} \mathbb{1}\left[v_j < v_i\right]$.
23:    $\varpi_i \leftarrow \widehat{q}_v(v_i)\left[\exp(-v_i) - \exp(-v_{i+1})\right]$.
24: **end for**
25: **for** $r = 1$ **to** $R$ **do**
26:    Sample $j \sim$ Multinomial$(\varpi_1 \ldots \varpi_M)$.
27:    Sample $\eta \sim$ Uniform(0,1).
28:    $v^* \leftarrow v_j - \log\left[1 - \eta\left(1 - \exp\left(v_j - v_{j+1}\right)\right)\right]$.
29:    $p \leftarrow 0$
30:    $n_r \leftarrow 0$. {Counter for number of proposals}
31:    **while** $p > v^*$ **do**
32:       Sample $\theta_r \sim g(\theta)$.
33:       $p \leftarrow -\log\Phi(\theta_r|y)$.
34:       $n_r \leftarrow n_r + 1$.
35:    **end while**
36: **end for**
37: **return** $\theta_1 \ldots \theta_R$ (plus $n_1 \ldots n_R$ and $v_1 \ldots v_M$ if computing a marginal likelihood).
```

covariates x_i, and a heterogeneous coefficient vector β_i, such that

$$p_i = \frac{\exp(x_i'\beta_i)}{1+\exp(x_i'\beta_i)},\ i = 1\dots N \tag{1}$$

The coefficients can be thought of as sensitivities to the covariates, and they are distributed across the population of households following a multivariate normal distribution with mean μ and covariance Σ. We assume that we know Σ, but we do not know μ. Instead, we place a multivariate normal prior on μ, with mean 0 and covariance Ω, which is determined in advance. Thus, each β_i, and μ, are $k-$dimensional vectors, and the total number of unknown variables in the model is $(N+1)k$.

The log posterior density, including normalization constants, is

$$\log \pi(\beta_{1:N}, \mu | Y, X, \Sigma_0, \Omega_0) = \sum_{i=1}^{N} \log \binom{T}{y_i} + y_i \log p_i + (T - y_i) \log(1 - p_i) \tag{2}$$

$$- \frac{N}{2}\left(\log|\Sigma| + k \log(2\pi)\right) - \frac{1}{2}\sum_{i=1}^{N}(\beta_i - \mu)' \Sigma^{-1} (\beta_i - \mu) \tag{3}$$

$$- \frac{1}{2}\left(\log|\Omega_0| + k\log(2\pi) + \mu'\Omega_0^{-1}\mu\right) \tag{4}$$

where p_i is defined in Equation 1 as a function of β_i. The functions that compute the log posterior and its gradient, `demo.get.f` and `demo.get.grad` are in the `R/demo_funcs.R`. This file also includes a function, `get.hess.struct`, that returns the sparsity structure of the Hessian. These functions depend on the **Matrix** and **sparseHessianFD** packages.

In the sections that follow, I will discuss the code in the file `demo/choice_gds.R`. This example generates batches of samples in parallel. The user may need to modify the way in which the code is parallelized, depending on available resources.

2.1 Preparing for GDS

At the top of the `choice_gds.R` file, we do some initial housekeeping, and set the parameters of the GDS sampler. We also register multiple processors for running some of the steps in parallel.

```
library(Matrix)
library(trustOptim)
library(sparseHessianFD)
library(sparseMVN)
library(plyr)
library(bayesGDS)
library(doParallel)

run.par <- TRUE
if(run.par) registerDoParallel(cores=12)
 else registerDoParallel(cores=1)

set.seed(123)
```

The **sparseMVN** package includes functions for sampling from, and computing the log density of, a multivariate normal (MVN) distribution when either the covariance or precision matrix is sparse. To use these functions with **bayesGDS**, we need to place them in wrappers. The `params` argument must be a list of parameters which can then be passed to the proposal distribution functions. Of course, there is no requirement that the proposal distribution has to be MVN.

```
rmvn.sparse.wrap <- function(n.draws, params) {
  res <- rmvn.sparse(n.draws, params$mean, params$CH, prec=TRUE)
  return(res)
}

dmvn.sparse.wrap <- function(d, params) {
  res <- dmvn.sparse(d, params$mean, params$CH, prec=TRUE)
  return(res)
}
```

2.2 Simulating data and setting priors

The code in the next section simulates some data from the model, and defines the prior parameters.

```
N <- 100  ## number of heterogeneous units
k <- 3  ## number of covariates
T <- 40  ## number of "purchase opportunities per unit
nvars <- N*k + k ## total number of parameters

## Simulate data and set priors

x.mean <- rep(0,k)
x.cov <- diag(k)
mu <- rnorm(k,0,10)
Omega <- diag(k)
inv.Sigma <- rWishart(1,k+5,diag(k))[,,1]
inv.Omega <- solve(Omega)
X <- t(rmvnorm(N, mean=x.mean, sigma=x.cov)) ## k x N
B <- t(rmvnorm(N, mean=mu, sigma=Omega)) ## k x N
XB <- colSums(X * B)
log.p <- XB - log1p(exp(XB))
Y <- apply(as.matrix(log.p), 1,function(q) return(rbinom(1,T,exp(q))))
```

This section should be self-explanatory, as the variables correspond to the model specification above. The arguments `X`, `Y`, `inv.Sigma` and `inv.Omega` will be passed to the functions that compute the log posterior and gradient.

2.3 Estimating the Hessian

Functions for computing the log posterior and its gradient are in the file `R/demo_funcs.R`. To estimate the Hessian, we use the **sparseHessianFD** package. Since we assume that household purchases are conditionally independent, the cross-partial derivatives for β_i across households are all zero. This means that the Hessian is sparse, with a known sparsity pattern. The `demo.get.hess.struct` function returns this pattern.

The `new.sparse.hessian.obj` constructs an object with functions that return the log posterior, gradient, and the full, sparse Hessian. The Hessian is returned in `dsCMatrix` format, which can be used by both **sparseMVN** and **trustOptim**. By exploiting the sparsity of the Hessian, we can find the posterior mode, and construct MVN precision matrices, faster than if we treated the Hessian as dense. See the documentation for **sparseHessianFD** for more details on how the Hessian is estimated. One warning is that **sparseHessianFD** will not work well if the gradient is not "exact". Specifically, the gradient should not be estimated numerically using finite differencing. The gradient should be either derived analytically (as we do in this example), or use some kind of automatic differentiation.

Before setting up the sparse Hessian object, we sample a hypothetical starting value from a vector of standard normal random variates.

```
start <- rnorm(nvars) ## random starting values
hess.struct <- bayesGDS::demo.get.hess.struct(N, k)

## Setting up function to compute Hessian using sparseHessianFD package.
obj <- new.sparse.hessian.obj(start, fn=bayesGDS::demo.get.f,
                              gr=bayesGDS::demo.get.grad,
                              hs=hess.struct, Y=Y, X=X,
                              inv.Omega=inv.Omega,
                              inv.Sigma=inv.Sigma, T=T)

get.f.wrap <- function(x) return(obj$fn(x))
get.df.wrap <- function(x) return(obj$gr(x))
get.hessian.wrap <- function(x) return(obj$hessian(x))
```

2.4 Finding the posterior mode

Although one could use any number of methods to find the posterior mode, the `trust.optim` function from the **trustOptim** package is a good choice for hierarchical models. This is because the Hessian of the log posterior is sparse. The `trust.optim` function requires the user to supply a function that computes the objective function (`fn`) , the gradient (`gr`), and the Hessian (`hs`). In this example, these functions are `get.f.wrap`, `get.df.wrap`, and `get.hessian.wrap`, respectively. See the package manual and documentation for **trustOptim** for more information and the various methods, arguments and control parameters.

```
 opt <- trust.optim(start, fn=get.f.wrap,
                    gr = get.df.wrap,
                    hs = get.hessian.wrap,
                    method = "Sparse",
                    control = list(
                      start.trust.radius=5,
                      stop.trust.radius = 1e-7,
                      prec=1e-7,
                      function.scale.factor=-1,
                      report.freq=1L,
                      report.level=4L,
                      report.precision=1L,
                      maxit=500L,
                      preconditioner=0L
                      )
                    )

post.mode <- opt$solution
hess <- opt$hessian
```

2.5 Preparing the GDS sampler

With the posterior mode, and the Hessian at the mode, in hand, we can now start the GDS sampler itself. In our example, we will collect 20 samples from the target posterior, and we will estimate the marginal auxiliary parameter using 10,000 samples from the MVN proposal distribution. The proposal will have a mean at the posterior mode, and a precision matrix of $-.94$, times the Hessian at the mode.

```
n.draws <- 20
M <- 50000
ds.scale <- 0.94
fn.dens.prop <- dmvn.sparse.wrap
fn.draw.prop<- rmvn.sparse.wrap

chol.hess <- Cholesky(-ds.scale*hess)
```

```
prop.params <- list(mean = post.mode,
                    CH = chol.hess
                    )


log.c1 <- opt$fval
log.c2 <- dmvn.sparse.wrap(post.mode, prop.params)
```

Next, we construct the empirical approximation to the density of the threshold draws that we will use in the rejection sampling phase of the algorithm. After taking M draws from the proposal distribution, we evaluate the log posterior and MVN densities at each of those proposals, and compute $\log \Phi(\theta_i|y)$ for all proposals $\theta_1 \ldots \theta_M$.

```
draws.m <- as(fn.draw.prop(M,prop.params),"matrix")
log.post.m <- aaply(draws.m, 1,get.f.wrap, .parallel=run.par)
log.prop.m <- fn.dens.prop(draws.m,params=prop.params)
log.phi <- log.post.m - log.prop.m +log.c2 - log.c1
```

If $\log \Phi(\theta|y) > 0$ for any of these proposals, the proposal density is not valid, and we need to try again. Typically, this adaptation will just mean a change in the scale of the precision matrix of the proposal distribution.

```
invalid.scale <- any(log.phi>0)
cat("Are any log.phi > 0?  ",invalid.scale,"\n")
```

2.6 Collecting posterior draws

Finally, we come to the rejection sampling phase of the GDS algorithm. Details of the arguments are in the package documentation, and most of them have already been discussed in this note.

The biggest advantage to using GDS is that samples can be collected in parallel. One way to do is is to launch multiple instances of the `sample.GDS` with the routines in the **foreach** package. For example, if we need 100 samples, and we have 10 processing cores available, we can run 10 instances of `sample.GDS`, with each instance collecting

10 samples. In the function below, each of the `n.batch` instances of `sample.GDS` collects `batch.size` samples per batch. Since `foreach` combines the results of each instance into a single list, we use `Map` and `Reduce` to combine those results into a single list.

```
n.batch <- floor(n.draws / batch.size)
 draws.list <- foreach(i=1:n.batch, .inorder=FALSE) %dopar% sample.GDS(
                                  n.draws=batch.size,
                                  log.phi=log.phi,
                                  post.mode=post.mode,
                                  fn.dens.post = get.f.wrap,
                                  fn.dens.prop = dmvn.sparse.wrap,
                                  fn.draw.prop = rmvn.sparse.wrap,
                                  prop.params = prop.params,
                                  max.tries=max.tries,
                                  report.freq=50,
                                  announce=TRUE,
                                  thread.id = i,
                                  seed=as.integer(seed.id*i))

draws <- Reduce(function(x,y) Map(rbind,x,y), draws.list)
```

The list `draws` contains the following elements.

1. `draws` - the samples from the target posterior distribution. each column is a draw.

2. `counts` - the number of proposals that it took to get an acceptance. If a draw is accepted on the first proposal, the count is 1.

3. `gt.1` - an indicator that the log.phi for that particular draw happened to be greater than 0. In theory, this should not happen if the posterior is sufficiently dominated by the proposal density.

4. `log.post.dens`, log.prop.dens - the log posterior and proposal densities for the draws.

5. `log.thresholds` - the threshold that determines if the -log.phi from a proposal is low enough to be accepted.

6. `log.phi` - the $\log \Phi(\theta|y)$ of the accepted draw

A note on `gt.1`: All of these values really should be zero. If there is a very small proportion of ones, it's probably not a big deal. It just means that the proposal density is under-scaled, and does not fully dominate the target posterior over the domain for reasonable values. This could be because of deviations from normality in regions of high posterior mass, or because the tails of the proposal fall too quickly. These are samples that you probably want to accept anyway. However, we have not yet developed any theory to assess just how much error is introduced by an underscaled proposal.

Also, we have found in practice that if the proposal density is not sufficiently diffuse, the thresholds for acceptance can be quite high, leading to very low acceptance rates. Therefore, we continue to recommend running the algorithm again with a different proposal density if any elements of `gt.1` are not zero.

If any of the posterior draws is `NA`, then none of the first `max.tries` proposals was accepted as a sample from the target posterior. If this happens, either increase this value, or adjust the proposal density.

2.7 Log marginal likelihoods

GDS algorithm also offers a straightforward way to estimate the log marginal likelihood. See Braun and Damien (2013).

```
  if (any(is.na(dd$counts))) {
        LML <- NA
  } else {
    LML <- get.LML(draws$counts, log.phi, post.mode,
               fn.dens.post=get.f.wrap,
               fn.dens.prop=dmvn.sparse.wrap,
               prop.params=prop.params)
}
```

3 Practical advice

3.1 Exploiting sparsity

The advantages of GDS over MCMC are most evident when the Hessian of the log posterior is sparse. This will occur when heterogeneous units are conditionally independent, such that the cross-partial derivatives across unit-specific parameters are zero. As discussed in depth in Braun and Damien (2013), GDS scales linearly in the number of heterogeneous units under the conditional independence assumption.

There are a number of resources available in R for exploiting this sparsity. The **Matrix** package provides classes to store sparse matrices in a compressed format. The **trustOptim** package includes a trust region nonlinear optimizer that accepts the Hessian is sparse format, and **sparseHessianFD** can estimate a sparse Hessian efficiently when the sparsity pattern is known in advance (which it typically is). The **sparseMVN** package samples from, and computes the log density of, a multivariate normal distribution when either the covariance or precision matrix is sparse.

3.2 Choosing the mode-finding algorithm wisely

Since finding the posterior mode is a critical element in the GDS procedure, some thought should go into the best way to find it. If the gradient is computed either analytically or through automatic differentiation, and if the Hessian is sparse, the **trustOptim** and **sparseHessianFD** packages are effective tools. Even for problems with dense Hessians, the advantage of **trustOptim** is that it uses the norm of the gradient to determine when the algorithm has converged. Most (if not all) other optimization routines in other packages like **optim**, **optimx**, **Rcgmin**, and **nloptwrap** stop when the algorithm fails to make sufficient progress in terms of the objective function or the parameters. In our experience, those algorithms stop prematurely (i.e., the gradient is not flat at the solution). One approach we have taken when the Hessian of the log posterior is dense has been to run one of these other optimization algorithms until convergences, and then continue using **trustOptim**.

References

Bates D, Maechler M (2013). *Matrix: Sparse and Dense Matrix Classes and Methods.* R package version 1.1-0.

Braun M (2013a). *sparseHessianFD: an R package for estimating sparse Hessians.* R package version 0.1.1, URL `cran.r-project.org/web/packages/sparseHessianFD`.

Braun M (2013b). *sparseMVN: an R package for MVN sampling with sparse covariance and precision matrices.* R package version 0.1.0, URL `cran.r-project.org/web/packages/sparseMVN`.

Braun M (2013c). *trustOptim: an R package from optimization using trust regions.* R package version 0.8.2, URL `cran.r-project.org/web/packages/trustOptim`.

Braun M, Damien P (2013). "Generalized Direct Sampling for Hierarchical Bayesian Models." www.cox.smu.edu/web/michaelbraun.

Wickham H (2011). "The Split-Apply-Combine Strategy for Data Analysis." *Journal of Statistical Software*, **40**(1), 1–29.