Estimating Bayesian Hierarchical Models using *bayesGDS*

Michael Braun
Cox School of Business
Southern Methodist University

March 16, 2016

Braun and Damien (2015), henceforth known as BD, introduce an alternative to MCMC for sampling from posterior distributions. The main advantages of BD over MCMC are that samples can be collected in parallel, and that the algorithm is scalable (linear complexity) for hierarchical models with conditionally independent heterogeneous units. These features make BD an attractive estimation method for Bayesian hierarchical models of large datasets.

For the purposes of this vignette, we assume that the reader is familiar with the BD algorithm, and in particular, Algorithm 1. In the pages that follow, we discuss some of the practical issues involved in running the BD algorithm. The focus is on a few specific aspects of the algorithm:

1. exploiting the sparsity of the Hessian of the log posterior density;
2. finding the posterior mode through nonlinear optimization;
3. sampling from, and computing the log density of, a high-dimensional multivariate normal (MVN) distribution; and
4. computing the log marginal likelihood of the data under the model.

A common reaction to the BD algorithm, and especially to the need to find the posterior mode, is "easier said than done." We suspect that this reaction comes from people who use existing R functions and packages that are not particularly scalable. Examples include using the `optim` function in base R to find the posterior mode, or the `rmvnorm` function in the *mvtnorm* package for sampling from a multivariate normal (MVN) random variable. These are functions with which most R users are familiar, so it is understandable that the barrier to adoption of algorithms that fail when using them would be high.

Fortunately, there are alternatives to that are better suited for running the BD algorithm on hierarchical models. These packages are listed in Table 1, and are required to run the code in this vignette. The *Matrix* and *sparseHessianFD* define classes for working with sparse matrices, and the *sparseMVN*, and *trustOptim* packages respectively provide sampling and optimization routines that are designed to work with sparse matrices. Those packages are not strictly necessary to use BD, but this package (*bayesGDS*) implements only the rejection sampling phase. The *trustOptim*, *sparseHessianFD* and *sparseMVN* packages were initially written as complements to *bayesGDS*, with the BD algorithm in mind.

1 Some background

The goal is to sample a parameter vector θ from a posterior density $\pi(\theta|y)$, where $\pi(\theta)$ is the prior on θ, $f(y|\theta)$ is the data likelihood conditional on θ, and $\mathcal{L}(y)$ is the marginal likelihood of the data. Let $\mathcal{D}(\theta, y)$

Package	Specific use for the BD algorithm	Relevant lines in BD Algorithm
Matrix	Defines classes and methods for sparse matrices	3, 11, 32
sparseHessianFD	Numerical estimation of sparse Hessians when the gradient and sparsity pattern are known	Setup, 3, 11, 32
sparseMVN	Sampling from, and to computing the log density of, a multivariate normal (MVN) random variable for which either the covariance or precision matrix is sparse.	11, 12, 32, 33
trustOptim	Nonlinear optimization with a gradient-based stopping rule. Efficient for objective functions with sparse Hessians	3
bayesGDS	Rejection sampling phase of the BD algorithm, and computing marginal likelihood of the data under the model	20-37
doParallel	Multiple core, shared memory parallelization for sampling from proposal and posterior distributions	11-12, 31-35
plyr	Parallel application of a function to a row or column of a matrix	12

Table 1: R packages used in this vignette for implementing the BD algorithm

be the joint density of the data and the parameters. Therefore,

$$\pi(\theta|y) = \frac{f(y|\theta)\pi(\theta)}{\mathcal{L}(y)} = \frac{\mathcal{D}(\theta, y)}{\mathcal{L}(y)} \tag{1}$$

The model is fully specified by $\mathcal{D}(\theta, y)$, which is equivalent to the posterior density, up to a normalizing constant $1/\mathcal{L}(y)$.

If the heterogeneous units are conditionally independent, the data likelihood can be factored as

$$f(y|\theta) = \prod_{i=1}^{N} f_i\left(y_i|\beta_i, \alpha\right) \tag{2}$$

where i indexes households. Each y_i is a vector of observed data, each β_i is a vector of heterogeneous parameters, and α is a vector of homogeneous population-level parameters. The β_i are distributed across the population of households according to a mixing distribution $\pi(\beta_i|\alpha)$, which also serves as the prior on each β_i. The elements of α may influence either the household-level data likelihoods, or the mixing distribution (or both). In this example, θ includes all $\beta_1 \ldots \beta_N$ and all elements of α.

The prior itself can be factored as

$$\pi(\theta) = \prod_{i=1}^{N} \pi_i(\beta_i|\alpha) \times \pi(\alpha). \tag{3}$$

Thus, the log posterior density is written as

$$\log \pi(\beta, \alpha|y) = \sum_{i=1}^{N} \left[\log f(y_i|\beta_i) + \log \pi(\beta_i|\alpha)\right] + \log \pi(\alpha) - \log \mathcal{L}(y) \tag{4}$$

$\log \mathcal{D}(\theta, y)$ is equivalent to $\log \pi(\beta, \alpha|y)$, without the $\log \mathcal{L}(y)$ term.

1.1 Sparse Hessians

An implication of the conditional independence assumption is that the cross-partial derivatives of the unnormalized log posterior density, $\frac{\partial^2 \log \mathcal{D}(\theta, y)}{\beta_i \beta_j}$ are zero for all $i \neq j$. As the number of households in the dataset increases, the number of elements in the Hessian matrix increases quadratically, but the number of *non-zero* elements increases only linearly. Thus, the Hessian becomes sparser as the data set gets larger.

A sparse matrix is one that has a relatively small number of nonzero elements. A sparse matrix can be represented by only the nonzero values, and the row and column indices of those values. All of the other elements are known to be zero, so they do not need to be stored explicitly. Thus, the amount of memory required to store a sparse matrix grows with the number of nonzero elements, as opposed to the product of the number of rows and columns. Also, linear algebra operations are more efficient on compressed sparse matrices, because operation on the nonzero elements can be ignored. These computational advantages come into play in nearly all of the steps of the BD algorithm. Although the sparsity of the Hessian of the log posterior density is not a requirement for the BD algorithm, it is that sparsity that makes the algorithm scalable (Braun et al. 2015, Sec. 4).

As an example, suppose we have a hierarchical model with N heterogeneous units, each with a parameter vector of length k. Also, assume that there are p population-level parameters or hyperparameters.

The *sparsity pattern* of the Hessian depends on how the variables are ordered within the vector. One such ordering is to group all of the coefficients for each unit together.

$$\beta_{11}, ..., \beta_{1k}, \beta_{21}, ..., \beta_{2k}, ... \, , \, ... \, , \, \beta_{N1} \, , \, ... \, , \, \beta_{Nk}, \mu_1, ..., \mu_p \tag{5}$$

In this case, the Hessian has a "block-arrow" structure. For example, if $N = 6$, $k = 2$, and $p = 2$, then there are 14 total variables, and the Hessian will have the sparsity pattern in Figure 1a.

Another option would be to group the coefficients by covariate.

$$\beta_{11}, ..., \beta_{1N}, \beta_{21}, ..., \beta_{2N}, ..., ..., \beta_{k1}, ..., \beta_{kN}, \mu_1, ..., \mu_p \tag{6}$$

Now the Hessian has an "off-diagonal" sparsity pattern, as in Figure 1b.

In both cases, the number of non-zeros is the same. There are 196 elements in this symmetric matrix, but only 76 are non-zero, and only 45 values are unique. Although in this example the reduction in resource consumption from using a sparse matrix structure for the Hessian may be modest, consider what would happen if $N = 1000$ instead. In that case, there are $2,002$ variables in the problem, and more than 4 million elements in the Hessian. However, only $12,004$ of those elements are non-zero. If we work with only the lower triangle of the Hessian we only need to work with only $7,003$ values.

As discussed in Section 4 of Braun and Damien (2015), the sparsity of the Hessian is what makes the method scalable, in terms of the number of heterogeneous units in the data set. Each additional unit adds k rows and columns to the Hessian, so the number of formal elements increases quadratically with N. However, in terms of *non-zero* elements, each unit adds a $k \times k$ block on the diagonal (correlation of variables within a unit), and a block in each margin (correlation between unit-level and population-level variables). Thus, the number of non-zero elements grows linearly, not quadratically, with N. Consequently, the complexity of

```
# 14 x 14 sparse Matrix of class "lgCMatrix"
#
#  [1,] | | . . . . . . . . . . | |
#  [2,] | | . . . . . . . . . . | |
#  [3,] . . | | . . . . . . . . | |
#  [4,] . . | | . . . . . . . . | |
#  [5,] . . . . | | . . . . . . | |
#  [6,] . . . . | | . . . . . . | |
#  [7,] . . . . . . | | . . . . | |
#  [8,] . . . . . . | | . . . . | |
#  [9,] . . . . . . . . | | . . | |
# [10,] . . . . . . . . | | . . | |
# [11,] . . . . . . . . . . | | | |
# [12,] . . . . . . . . . . | | | |
# [13,] | | | | | | | | | | | | | |
# [14,] | | | | | | | | | | | | | |
```

(a) A “block-arrow” sparsity pattern

```
# 14 x 14 sparse Matrix of class "lgCMatrix"
#
#  [1,] | . . . . . | . . . . . | |
#  [2,] . | . . . . . | . . . . | |
#  [3,] . . | . . . . . | . . . | |
#  [4,] . . . | . . . . . | . . | |
#  [5,] . . . . | . . . . . | . | |
#  [6,] . . . . . | . . . . . | | |
#  [7,] | . . . . . | . . . . . | |
#  [8,] . | . . . . . | . . . . | |
#  [9,] . . | . . . . . | . . . | |
# [10,] . . . | . . . . . | . . | |
# [11,] . . . . | . . . . . | . | |
# [12,] . . . . . | . . . . . | | |
# [13,] | | | | | | | | | | | | | |
# [14,] | | | | | | | | | | | | | |
```

(b) An “off-diagonal” sparsity pattern.

Figure 1: Examples of sparsity patterns for a hierarchical model. The pattern depends on the ordering of the coefficients. The `lgCMatrix` class is defined in the *Matrix* package.

many of the steps of the algorithm, such as multiplying matrices, generating Cholesky factors, and solving sparse linear systems, grow linearly as well.

1.2 Nonlinear optimization

Finding the posterior mode θ^* can be a hard problem when there is a high number of variables, especially when using standard optimization routines, like the `optim` function in base R, and others that are described in the CRAN Task View on Optimization. There are two common problems with these algorithms:

Premature termination Many algorithms apply a stopping rule that is based on whether the optimizer is making "sufficient progress” (note the `abstol` and `reltol` control arguments for `optim`). If the objective function does not improve by some minimum amount, the algorithm believes that it has converged to a local optimum. This can happen before the gradient of the objective function is sufficiently flat. For an unconstrained objective function, the *only* appropriate stopping rule is whether a function of the gradient (e.g., the Euclidean norm) is numerically zero. Any optimization routine that stops before that is stopping prematurely, and the resulting normal approximations to the posterior density would be invalid.

Poor scalability Search methods like Nelder-Mead (the default algorithm for `optim`) are inefficient with a massive number of parameters because the search space is large, and they do not exploit information about slope and curvature to accelerate convergence. Conjugate gradient (*CG*) and quasi-Newton (e.g., *BFGS*) do use gradient information, with *BFGS* tracing out the curvature of the function by using successive gradients to approximate the inverse Hessian. However, because *BFGS* stores the entire dense inverse Hessian, its use is resource-intensive when the number of parameters is large. *CG* and limited-memory *BFGS* methods do not store the full Hessian (or its inverse), so they can be more suited for large-scale problems. However, like *BFGS*, they are not certain to approximate the curvature of the objective function accurately at any particular iteration, especially if the function is not convex (Braun 2013).

A better choice of nonlinear optimizer would be one that uses exact calculations of the gradient and Hessian, remains scalable for large problems, stops only when the norm of the gradient is numerically zero, and is stable when passing through regions in which the surface of the objective function is flat. The `trust.optim` function in the *trustOptim* package meets those criteria. Braun (2013) describes the many advantages of the `trust.optim` function. Two advantages of note are:

1. convergence only when the norm of the gradient is sufficiently close to zero[1]; and
2. the ability to use exact Hessian information that is stored as a compressed sparse matrix.

The `trust.optim` function supports three optimization routines: two quasi-Newton methods (*SR1* and *BFGS*; see Nocedal and Wright (2006)), and a *Sparse* method. All three methods require the user to supply a function that returns the value of the objective function, *and* a function that returns the gradient. The *Sparse* method requires an additional function that returns the Hessian as a `dgCMatrix` object. The `dgCMatrix` class is defined in the *Matrix* package, and is one of several classes for the storage of, and operation on, sparse matrices.

Be sure that the Hessian matrix function returns a `dgCMatrix` matrix, and not a structured matrix like `dsCMatrix` (symmetric) or `dtCMatrix` (triangular); a row-oriented matrix like `dgRMatrix`; or even a dense matrix like `dgeMatrix` or a base R `matrix`. `trust.optim` accepts only `dgCMatrix`, even if the matrix is numerically dense.

1.3 Computing derivatives

Nearly all reasonable choices of a nonlinear optimization algorithm require the user to provide a the gradient of the objective function. The *Sparse* method for `trust.optim` requires the Hessian as well. The Hessian is also required to approximate the posterior density with a MVN distribution around the posterior mode (see Section 1.4).

For the purposes of the BD algorithm, there are two "good" ways to compute a derivative. The first is to derive it analytically, and write a function to compute it. This approach is straightforward, but it can be tedious and error-prone for complicated models.

The second is to use automatic, or algorithmic, differentiation (AD). In short, AD generates code for the derivative by applying the chain rule on the same sequence of operations that computes the objective function. There are a number of different approaches to implementing AD, and AD libraries are available for many programming languages. However, as of now, there are none for R that are well-suited for a general class of Bayesian hierarchical models. For R users, we believe that coding the objective function in C++ using the *CppAD* library, and interfacing with R using *Rcpp*, is the best option at the moment. How to do this will be the subject of a future vignette. What matters is that functions that return the gradient and Hessian of the log posterior density are available, and that they are sufficiently accurate. The advantage of both analytic and AD derivatives is that they are “exact.”

We do not recommend estimating the gradient by numerical approximation via finite differencing (FD). FD involves computing $\partial f/\partial x_j \approx \left[f\left(x_j+h\right)-f\left(x_j\right)\right]/h$, or some variation thereof, for each of the $j=1...J$ variables, using an arbitrarily small h as a “perturbation factor.” As $h\rightarrow 0$, this estimated difference approaches the gradient. Not only are FD methods are highly vulnerable to numerical precision problems, but the complexity of the method grows with the number of variables Thus, FD is not a reasonable option for estimating the gradient when the number of variables is large. The time to compute the gradient using AD,

[1] There is an exception. If the trust region gets too small for any further progress to be made, `trust.optim` will stop. This may happen if the norm of the gradient is larger than the specified tolerance, but still pretty small. In that case, there is no problem. If the trust region radius is nearly zero, but the gradient is nowhere close to zero, then there is some kind of problem that requires further investigation.

on the other hand, is only a small fixed multiple of the time to compute the objective function, regardless of the number of variables (Griewank and Walther 2008).

The computational cost of computing a dense Hessian using FD is quadratic in the number of variables, and the numerical precision problems are even more pronounced than for a gradient. Nevertheless, one can use FD to estimate the Hessian if the Hessian is sparse, *and* the sparsity pattern is known in advance, *and* the gradient is exact (either derived analytically or computed using AD). The *sparseHessianFD* package defines an R class for doing this (the algorithms are based on Coleman et al. (1985b) and Coleman et al. (1985a)). An object of class `sparseHessianFD` is constructed from functions that return the value of the log density and its gradient; any additional arguments that are passed to these functions; and the row and column indices of the non-zero elements of the lower triangle of the Hessian. The `sparseHessianFD` object contains member functions not only for the log density and the gradient, but also for the sparse Hessian, as a `dgCMatrix` object. See the *sparseHessianFD* documentation and vignette for more details.

The `sparseHessianFD` class is useful for hierarchical models because the sparsity pattern is predictable. Even if the Hessian were derived analytically, it still may be faster to use `sparseHessianFD` for repeated estimation, because of the way the package exploits sparsity. Given the typical sparsity pattern of a hierarchical model, the time to estimate a Hessian using *sparseHessianFD* does not grow with the number of heterogeneous units, making it a scalable way of estimating sparse Hessians for large datasets.

The trade-off from using the *sparseHessianFD* package is that the Hessian is still a numerical approximation. We cannot guarantee that this approximation is "good enough" for all cases. However, the performance of an optimization algorithm is likely to be more sensitive to numerical imprecision in the gradient than in the Hessian. Thus, we are comfortable working with finite-differenced sparse Hessians, even though we will not use finite-differenced gradients. This approach will almost certainly fail if the gradient itself is estimated using FD. In that case, the estimate of the Hessian would be a finite difference of finite differences, with too much numerical imprecision to be of much value.

1.4 High-dimensional MVN distribution

Another potential source of poor scalability is in sampling from, and computing the log density of, a multivariate normal (MVN) distribution. The `rmvnorm` and `dmvnorm` functions in the *mvtnorm* package (Genz et al. 2012) are inefficient in the context of the BD algorithm for three reasons.

1. They require the covariance matrix as one of the arguments, meaning that the negative Hessian must be inverted explicitly.

2. They do not exploit the sparsity of the Hessian for computational gain; and

3. The `rmvnorm` and `dmvnorm` functions factor the covariance matrix every time the function is called, even if the matrix has not changed.

The *sparseMVN* package addresses these issues. The `rmvn.sparse` and `dmvn.sparse` functions take, as one of the arguments, the Cholesky decomposition of either a sparse covariance or sparse precision matrix. Thus, the user can use either the covariance or precision matrix, depending on which is more convenient. The Cholesky decomposition must be a `Cholesky` object, computed using the `Cholesky` function, both of which are defined in the *Matrix* package. The size of the sparse Hessian grows only linearly with the size of the dataset, the *sparseMVN* package is a scalable alternative for working with an MVN. More details are available in the *sparseMVN* documentation and vignette, as well as in the example in 3.

2 Example: hierarchical binary choice

Before going into the details of how to put all of the pieces of the BD algorithm together, let's consider the following example of a log posterior density function with a sparse Hessian. Suppose we have 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 covariates x_i, and a heterogeneous coefficient vector β_i, such that

$$p_i = \frac{\exp(x_i'\beta_i)}{1+\exp(x_i'\beta_i)},\ i = 1...N \tag{7}$$

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_0. Thus, each β_i, and μ are $k-$dimensional vectors, and the total number of unknown variables in the model is $(N+1)k$.

In this model, we will make an assumption of *conditional independence* across households. A household's purchase count y_i depends on that household's β_i, but not the parameters of any other household, β_j, conditional on other population level parameters. Since μ and Σ depend on β_i for *all* households, we cannot say that y_i and y_j are truly independent. A change in β_i affects μ and Σ, which in turn affect β_j for some other household j. However, if we condition on μ and Σ, then y_i and y_j are independent, so we describe the data likelihoods as conditionally independent.

This conditional independence assumption is what allows us to write the joint likelihood of the data as a product of individual-level probability models. Therefore, the log posterior density, ignoring any normalization constants, is

$$\log\pi(\beta_{1:N},\mu|Y,X,\Sigma_0,\Omega_0) = \sum_{i=1}^{N} p_i^{y_i}(1-p_i)^{T-y_i} - \frac{1}{2}\left(\beta_i-\mu\right)'\Sigma^{-1}\left(\beta_i-\mu\right) - \frac{1}{2}\mu'\Omega_0^{-1}\mu \tag{8}$$

2.1 Functions, sample data, and priors for the example

The *bayesGDS* package includes a simulated dataset for N households and k covariates per household. There are q population-level parameters, which for this example happens to equal k. A household makes Y purchases out of T opportunities. The purchase probability for each household depends on a covariate matrix X. So let's start by loading the data, and setting hyperprior values for Σ^{-1} and Ω^{-1}.

```
data(binary_small)
binary <- binary_small #rename for brevity
str(binary)

# List of 3
#  $ Y: int [1:4] 135 127 114 90
#  $ X: num [1:2, 1:4] -0.06369 0.33905 0.00157 0.23102 -0.20344 ...
#  $ T: num 200

N <- length(binary[["Y"]])
k <- NROW(binary[["X"]])
q <- k
```

```
nvars <- as.integer(N*k + q)
priors <- list(inv.Sigma = diag(k), ##rWishart(1,k+5,diag(k))[,,1],
               inv.Omega = diag(k))
data.frame(parameter = c("N","k","q"),
           value =  c(N, k, q))
```

```
#   parameter value
# 1         N     4
# 2         k     2
# 3         q     2
```

The function `binary.f` returns the unnormalized log posterior density of this model, $\log \mathcal{D}(\theta, y)$, evaluated at the vector passed as the first argument. This function takes two additional named arguments, `data` and `priors`. The function `binary.grad` returns the gradient, and `binary.hess` returns a Hessian, in a sparse compressed format.

```
start <- rnorm(nvars) ## random starting values
f <- binary.f(start, data=binary, priors=priors)
f
```

```
# [1] -563.53
```

```
df <- binary.grad(start, data=binary, priors=priors)
str(df)
```

```
#  num [1:10] -0.861 8.19 1.523 6.715 1.095 ...
```

```
d2f <- binary.hess(start, data=binary, priors=priors)
print(d2f[1:6,1:6], digits=3)
```

```
# 6 x 6 sparse Matrix of class "dgCMatrix"
#
# [1,]  -1.20   1.08 .        .        .       .
# [2,]   1.08  -6.73 .        .        .       .
# [3,] .      .       -1.0001  -0.0181 .       .
# [4,] .      .       -0.0181  -3.6621 .       .
# [5,] .      .      .        .         -3.059  -0.273
# [6,] .      .      .        .         -0.273  -1.036
```

2.2 Using sparseHessianFD when the Hessian is unknown

Hessians can be difficult to derive analytically, but Hessians for hierarchical models have predictable sparsity patterns. The *sparseHessianFD* package simplifies numerical approximation of a sparse Hessian, and should be used if exact methods are unavailable (see Section 1.3. The `sparseHessianFD` function initializes an object of the `sparseHessianFD` class. This initializer requires the sparsity pattern of the lower triangle of the Hessian of $\log \mathcal{D}(\theta, y)$. The sparsity pattern is defined by integer vectors of the row and column indices of the nonzero elements.

A straightforward way to generate the sparsity pattern is to build a sparse integer or logical matrix, and then use the `Matrix.to.Coord` function from *sparseHessianFD*. The `Matrix.to.Coord` function returns a list of two integer vectors that can be passed to `sparseHessianFD`.

```
require(sparseHessianFD, quietly=TRUE)
hs <- Matrix(0, nvars, nvars)
for (i in 1:(N + 1)) {
    ## range of row / col indices of block diagonal
    rng <- ((i-1)*k+1):(k*i)
    hs[rng, rng] <- tril(Matrix(1,k,k)) ## lower triangle
}
hs[N*k + 1:q, 1:(N*k)] <- 1 ## bottom margin
hsNZ <- Matrix.to.Coord(hs)
str(hsNZ)

# List of 2
#  $ rows: int [1:31] 1 2 9 10 2 9 10 3 4 9 ...
#  $ cols: int [1:31] 1 1 1 1 2 2 2 3 3 3 ...
```

Now we can construct the `sparseHessianFD` object.

```
FD <- sparseHessianFD(start, binary.f, binary.grad,
                         rows=hsNZ[["rows"]], cols=hsNZ[["cols"]],
                         data=binary, priors=priors)
```

One advantage to using `sparseHessianFD` is that any additional arguments that need to be passed to the objective function are stored within the object. This means that we do not need to repeatedly pass the data and prior arguments when computing the log posterior density and its derivatives.

```
f <- FD$fn(start)
df <- FD$gr(start)
hess <- FD$hessian(start)
```

As a check, we see that the upper-left corner of the Hessian is sparse (the entire Hessian is too big to print), and `FD$hessian` returns the same matrix as `binary.hess`.

```
print(hess[1:6,1:6], digits=3)

# 6 x 6 sparse Matrix of class "dgCMatrix"
#
# [1,]  -1.20   1.08 .        .        .       .
# [2,]   1.08  -6.73 .        .        .       .
# [3,] .      .       -1.0001  -0.0181 .       .
# [4,] .      .       -0.0181  -3.6621 .       .
# [5,] .      .      .        .         -3.059  -0.273
# [6,] .      .      .        .         -0.273  -1.036

all.equal(hess, d2f, tolerance = 1e-7)

# [1] TRUE
```

3 Running the algorithm

Now it's time to estimate the posterior distributions of the model parameters.

3.1 Finding the posterior mode

For reasons discussed in Section 1.2, we use the *trustOptim* package to find the optimum of $\log\mathcal{D}(\theta, y)$. For the `fn`, `gr` and `hs` arguments, we provide the member functions of the FD object. Details on the control list for `trust.optim` are available in the *trustOptim* documentation. The most important option to mention here is that `function.scale.factor` must be positive for minimization, and negative for maximization.

```
require(trustOptim, quietly=TRUE)
opt <- trust.optim(start, fn=FD$fn, gr = FD$gr, hs = FD$hessian,
                   method = "Sparse",
                   control = list(
                       start.trust.radius=5, stop.trust.radius = 1e-7,
                       prec=1e-7, report.precision=1,
                       maxit=500, preconditioner=1,
                       function.scale.factor=-1
                       )
                   )

# Beginning optimization
#
# iter     f   nrm_gr                     status
#   1   534.5    0.4                Continuing
#   2   534.5    0.0                Continuing
#   3   534.5    0.0                Continuing
#
# Iteration has terminated
#   3   534.5    0.0                   Success

theta.star <- opt[["solution"]]
hess <- opt[["hessian"]]
```

3.2 Defining proposal functions

Next, we define functions to sample from a proposal density, and to compute the log density of a proposal draw. As discussed in Section 1.4, `rmvn.sparse` and `dmvn.sparse` require a mean vector, and a Cholesky decomposition of either the covariance or precision matrix, as separate arguments. The `sample.GDS` function in *bayesGDS* requires those parameters to be in a single list. That means we need some wrapper functions.

```
require(sparseMVN, quietly=TRUE)
rmvn.sparse.wrap <- function(n.draws, params) {
    rmvn.sparse(n.draws, params[["mean"]], params[["CH"]], prec=TRUE)
}
dmvn.sparse.wrap <- function(d, params) {
    dmvn.sparse(d, params[["mean"]], params[["CH"]], prec=TRUE)
}
```

The proposal mean is the posterior mean. The proposal precision is the negative Hessian, times a scaling factor. The `Cholesky` function is defined in the *Matrix* package. Do not use the base R function `chol`, which is not designed for sparse matrices, and cannot be used by the *sparseMVN* functions.

```
scale <- .96
chol.hess <- Cholesky(-scale*hess)
prop.params <- list(mean = theta.star, CH = chol.hess)
```

3.3 Side note: running the algorithm in parallel

An advantage of BD over MCMC is that both proposal and posterior samples can be generated in parallel. There are a number of different mechanisms available in R for running jobs in parallel. One that we think is easy to use is based on the *doParallel* package. In the next code chunk, I allocate 10 cores for parallel processing, and set a random seed for simulating random variables (more on that later). If you do not want to run the code in parallel, change the flag to `run.par <- FALSE`. This will change the number of allocated cores to 1.

```
library(doParallel, quietly=TRUE)
run.par <- TRUE
if(run.par) registerDoParallel(cores=2) else registerDoParallel(cores=1)
seed.id <- 123
set.seed(seed.id)
```

3.4 Estimating the density of threshold values

Braun et al. (2015) define the following:

$$\Phi(\theta|y) = \frac{\mathcal{D}(\theta, y) \cdot c_2}{g(\theta) \cdot c_1} \tag{9}$$

$$c_1 = \mathcal{D}(y, \theta^*) \tag{10}$$

$$c_2 = g(\theta^*) \tag{11}$$

where $g(\theta)$ is chosen such that $0 < \Phi(\theta|y) \leq 1$. The next step in the BD algorithm is to simulate an empirical approximation to the cumulative distribution of $-\log \Phi(\theta|y)$. We sample M times from the proposal distribution, and compute $\log \Phi(\theta|y)$ for each proposal sample. The next code chunk also uses the `aaply` function from the *plyr* package to compute the log posterior density for each proposal draw. This step will run in parallel if `run.par == TRUE`.

```
M <- 10000  ## proposal draws
log.c1 <- FD$fn(theta.star)
log.c2 <- dmvn.sparse.wrap(theta.star, prop.params)
draws.m <- as(rmvn.sparse.wrap(M,prop.params),"matrix")
log.post.m <- plyr::aaply(draws.m, 1, FD$fn, .parallel=run.par)
log.prop.m <- dmvn.sparse.wrap(draws.m, params=prop.params)
log.phi <- log.post.m - log.prop.m + log.c2 - log.c1
valid.scale <- all(log.phi <= 0)
stopifnot(valid.scale)
```

The last two lines in the previous code chunk are checks that $\Phi(\theta|y) \leq 1$ for all of the proposal draws. If the check fails, then the proposal distribution is invalid. If this happens, the proposal distribution $g(\theta)$ may need to be more diffuse by reducing the scaling factor on the negative Hessian.

If M is too low, there may not be enough proposal draws to confirm that the proposal distribution is sufficiently diffuse. This can cause problems later. Also, making the proposal distribution too "tight" may make it too hard to accept posterior samples. We think that time invested in “optimizing” the proposal distribution is not a well-used resource.

3.5 Posterior draws via rejection sampling

Finally we can start sampling from the posterior density, using functions in the *bayesGDS* package. We should proceed to this step only if `valid.scale==TRUE`.

The `sample.GDS` function is the "workhorse" function of *bayesGDS*. Each call to `sample.GDS` collects posterior samples serially. The argument `n.draws` is the number of draws for that particular call to `sample.GDS`. The other required arguments to `sample.GDS` are the M-length vector of $\log\Phi(\theta|y)$, the posterior mode θ^*; the function that returns the log posterior density; the functions that sample from, and compute the log density of, the proposal distribution; parameters of the proposal distribution. See the *bayesGDS* package documentation for descriptions of the function arguments.

The following code chunk runs `sample.GDS` on a simple processor.

```
n.draws <- 5  ## total number of draws needed
max.tries <- 100000  ## to keep sample.GDS from running forever
if (!run.par) {
    draws <- sample.GDS(n.draws = n.draws,
                        log.phi = log.phi,
                        post.mode = theta.star,
                        fn.dens.post = FD$fn,
                        fn.dens.prop = dmvn.sparse.wrap,
                        fn.draw.prop = rmvn.sparse.wrap,
                        prop.params = prop.params,
                        report.freq = 1, announce=TRUE)
}
```

The `foreach` function runs `sample.GDS` in parallel on multiple processors. Each instance of `sample.GDS` is responsible for a batch of posterior draws. If we need `n.draws` samples from the posterior, and want to run `n.batch` batches in parallel (say, one batch for each core), then each instance will collect `batch.size` samples (rounding up to the nearest integer). The return object of `foreach` is a list, with each element being a return object of `sample.GDS`. The order in which the `sample.GDS` objects are returned does not matter. Samples can be identified by batch with the `thread.id` argument. It is important to start each instance with a different random seed, which we specify in the `seed` argument.

When all processing cores have finished collecting their samples, we combine the list elements using a Map-Reduce construct.

```
if (run.par) {
    n.batch <- 10
    batch.size <- ceiling(n.draws / n.batch)
    draws.list <- foreach(i=1:n.batch, .inorder=FALSE) %dopar% sample.GDS(
        n.draws = n.draws,
        log.phi = log.phi,
        post.mode = theta.star,
        fn.dens.post = FD$fn,
        fn.dens.prop = dmvn.sparse.wrap,
        fn.draw.prop = rmvn.sparse.wrap,
        prop.params = prop.params,
        report.freq = 1,
        thread.id = i,
        announce=TRUE,
        seed=as.integer(seed.id*i))
    ## combine results from each batch
    draws <- Reduce(function(x,y) Map(rbind,x,y), draws.list)
}
```

There are almost certainly ways to run sample.GDS on distributed memory platforms (e.g., MPI, Amazon, etc), but we have not tried them yet. If you are able to do that successfully, please share your experiences.

One problem with this batch-oriented parallel sampling scheme is that the algorithm does not end until all of the samples are collected from all of the batches. This means that some batches will finish before others. Suppose we are running sample.GDS on two processing cores. It may happen that one core finishes, but the other core still has more than one sample to go. There is no way for the first core to help the second one. It's probably possible, but it is not yet implemented in this package.

3.6 The output

Let's take a look at the sample.GDS output.

```
str(draws)

# List of 7
#  $ draws          : num [1:50, 1:10] -0.5527 -2.2338 -0.3063 -0.0173 -0.1215 ...
#  $ counts         : num [1:10, 1:5] 1 1 1 1 1 1 1 1 1 1 ...
#  $ gt.1           : int [1:10, 1:5] 0 0 0 0 0 0 0 0 0 0 ...
#  $ log.post.dens  : num [1:10, 1:5] -538 -538 -537 -537 -541 ...
#  $ log.prop.dens  : num [1:10, 1:5] -10.17 -9.71 -9.07 -8.36 -12.53 ...
#  $ log.thresholds: num [1:10, 1:5] 1.062 0.663 1.484 0.631 0.802 ...
#  $ log.phi        : num [1:10, 1:5] -0.1486 -0.1415 -0.1121 -0.0869 -0.2519 ...
```

The draws element has a posterior sample in each row. Each column is a variable. The counts vector contains the number of proposal draws that were necessary to accept that particular posterior draw. This vector is needed to compute the log marginal likelihood (LML, see below), and to assess the efficiency of the algorithm. The gt.1 vector flags whether a threshold value was greater than 1. This is an important check, because if any elements of that vector are 1, that means that the proposal density was still just a bit too tight. If this is the case for only a couple of draws, it's probably not a big deal. If it is true for a large number of draws, not only is the proposal density invalid, but something probably went wrong earlier in the algorithm.

You can get summaries and quantiles for your parameters of interest by applying summary or quantile (or another such function) to each column. In this case, let's just summarize the population-level parameters.

```
quants <-  plyr::aaply(draws[["draws"]][,(N*k+1):nvars], 2,
                       quantile, probs=c(.025, .5, .975),
                       .parallel = run.par)
quants

#
# X1       2.5%      50%    97.5%
#   1 -2.45623 -0.87219 0.62268
#   2  0.34247  1.11739 1.99833
```

If any elements in counts is NA, it means that the proposal count reached the value in max.tries without an acceptance. This would suggest an inefficient sampler that requires further investigation. This could be that the proposal function is too diffuse, or, surprisingly, that it is too tight.

4 Estimating the log marginal likelihood

The log marginal likelihood (LML) is the likelihood of the data under the model, $\mathcal{L}(y)$. For better or worse, the LML is often used for model comparison (e.g., computing Bayes factors). As discussed in Braun et al. (2015), there is no generally accepted method for computing the LML from MCMC output. However, estimating the LML using the `LML` function in this package is straightforward. All of the arguments to `LML` were used in earlier function calls, or returned by `sample.GDS`.

```
if (any(is.na(draws[["counts"]]))) {
    LML <- NA
} else {
    LML <- get.LML(counts=draws$counts,
                   log.phi=log.phi,
                   post.mode=theta.star,
                   fn.dens.post= FD$fn,
                   fn.dens.prop=dmvn.sparse.wrap,
                   prop.params=prop.params)
}
draws[["LML"]] <- LML
draws[["acc.rate"]] <- 1/mean(draws$counts)
cat("Acceptance rate: ",draws[["acc.rate"]],
    "\nLog marginal likelihood: ",draws[["LML"]],"\n")

# Acceptance rate:  0.80645
# Log marginal likelihood:  -528.15
```

5 Future development

The *bayesGDS* package remains incomplete. Here are some improvements and enhancements that are under consideration for future releases.

1. A wrapper function for the entire BD algorithm;
2. Functions to summarize and display the output;
3. Options to retain a subset of variables and discard others (e.g., keep only population-level parameters);
4. A simpler interface for parallel sampling;
5. Examples of parallel execution on distributed platforms;
6. Better load balancing during parallel execution;
7. More examples of different kinds of applications; and
8. More details in the vignette about how `sample.GDS` works.

References

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

Braun, Michael and Paul Damien (2015). Scalable Rejection Sampling for Bayesian Hierarchical Models. *Marketing Science* Articles in Advance, 1–18. DOI: 10.1287/mksc.2014.0901.

Coleman, Thomas F, Burton S Garbow, and Jorge J Moré (1985a). Algorithm 636: Fortran Subroutines for Estimating Sparse Hessian Matrices. *ACM Transactions on Mathematical Software* **11**(4), 378. DOI: 10.1145/6187.6193.

Coleman, Thomas F, Burton S Garbow, and Jorge J Moré (1985b). Software for Estimating Sparse Hessian Matrices. *ACM Transactions on Mathematical Software* **11**(4), 363–377. DOI: 10.1145/6187.6190.

Genz, Alan, Frank Bretz, Tetsuhisa Miwa, Xuefei Mi, Friedrich Leisch, Fabian Scheipl, and Torsten Hothorn (2012). *mvtnorm: Multivariate Normal and t Distributions*. R package. Version 0.9-9994.

Griewank, Andreas and Andrea Walther (2008). *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. 2nd ed. Philadelphia: Society for Industrial and Applied Mathematics.

Nocedal, Jorge and Stephen J Wright (2006). *Numerical Optimization*. Second edition. Springer-Verlag.