Generalized Direct Sampling in R Using the bayesGDS Package

Michael Braun
MIT Sloan School of Management

August 23, 2012

Generalized Direct Sampling (GDS, Braun and Damien 2012) is a method of simulating from multivariate densities. 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 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 (2012, henceforth BD), and the reader should start there for the details. The purpose of this article is to show how to sample from a posterior density in R using the *bayesGDS* package. I also highlight some practical issues in using GDS, and show an example of GDS in action.

Although the *trustOptim* (Braun 2012) and *plyr* (Wickham 2011) packages are not dependencies for *bayesGDS*, the example at the end of this note does use them. Therefore, you are encouraged to install those packages before proceeding.

1 The GDS Algorithm

The goal is to sample θ from a posterior density

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

where $\mathcal{D}(\theta, y)$ is the joint density of the data and the parameters (the unnormalized posterior density). Let θ^* be the mode of $\mathcal{D}(\theta, y)$, and define $c_1 = \mathcal{D}(\theta^*, y)$. Choose some proposal distribution $g(\theta)$ that also has its mode at θ^*, and define $c_2 = g(\theta^*)$. Also, define the function

$$\Phi(\theta|y) = \frac{f(y|\theta)\pi(\theta) \cdot c_2}{g(\theta) \cdot c_1} \tag{2}$$

In summary, the steps of the GDS algorithm are as follows:

1. Find the mode of $\mathcal{D}(\theta, y)$, θ^* and compute the unnormalized log posterior density $c_1 = \mathcal{D}(\theta^*|y)$ at that mode.
2. Choose a distribution $g(\theta)$ so that its mode is also at θ^*, and let $c_2 = g(\theta^*)$.
3. Sample $\theta_1, \ldots, \theta_M$ independently from $g(\theta)$. Compute $\Phi(\theta)$ for these proposal draws. If $\Phi(\theta) > 1$ for any of these draws, repeat Step 2 and choose another proposal distribution for which $\Phi(\theta) < 1$ does hold.
4. Compute $v_i = -\log \Phi((\theta|y))$ for the M proposal draws, and place them in increasing order.
5. Evaluate, for each proposal draw,

$$q_M(v) = \sum_{i=1}^{M} \mathbb{1}\left[v_i < v\right] \tag{3}$$

   which is the empirical CDF of v_i for the M proposal draws.
6. Sample N draws of $v = v_i + \epsilon$, where a particular v_i is chosen according to the multinomial distribution with probabilities proportional to

$$w_i = q_M(v)\left[\exp(-v_i) - \exp(-v_{i+1})\right] \tag{4}$$

   and ϵ is a standard exponential random variate, truncated to $v_{i+1} - v_i$.
7. For each of the N required samples from the target posterior, sample θ from $g(\theta)$ until $-\log \Phi(\theta|y) < v$. Consider this first accepted draw to be a single draw from the target posterior $\pi(\theta|y)$.

8. Repeat steps 6 and 7 until N draws are collected.

The *bayesGDS* package does much of this work automatically. For Step 1, use your favorite nonlinear optimizer (the one in the *trustOptim* package is optimized for Bayesian hierarchical models with sparse Hessians). The `draw.MVN.proposals` and `get.log.dens.MVN` functions help with efficient execution of Step 3 when the proposal density is a multivariate normal. Steps 4 through 8 are performed by the `get.GDS.draws` function. The reason the algorithm is not fully automated from start to finish is that a check for the validity of the proposal density needs to be made at the end of Step 3.

This is all you need to do to sample from $\pi(\theta|y)$ using GDS. It's not hard, but there are some things that might cause problems with the algorithm. The rest of the paper is meant to help you avoid these obstacles.

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

$$p_i = \frac{\exp(x_i'\beta_i)}{1+\exp(x_i'\beta_i)},\ i = 1 \dots N \tag{5}$$

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

$$\log \pi(\beta_{1:N}, \mu | Y, X, \Sigma_0, \Omega_0) = \sum_{i=1}^{N} \log \binom{T}{y_i} + y_i \log p_i + (T - y_i) \log(1 - p_i) \tag{6}$$

$$- \frac{N}{2} \left(\log |\Sigma| + k \log (2\pi) \right) - \frac{1}{2} \sum_{i=1}^{N} (\beta_i - \mu)' \Sigma^{-1} (\beta_i - \mu) \tag{7}$$

$$- \frac{1}{2} \left(\log |\Omega_0| + k \log(2\pi) + \mu' \Omega_0^{-1} \mu \right) \tag{8}$$

where p_i is defined in Equation 5 as a function of β_i. The functions that compute the log posterior and its gradient, `log.post` and `get.grad` are in the file `inst/examples/ex_funcs.R`. This file also includes a function, `get.hess.struct`, that returns the sparsity structure of the Hessian.

In the sections that follow, I will discuss the code in the file `inst/examples/ex1.R`. These sections assume that the reader has read Braun and Damien (2012), and is generally familiar with the GDS algorithm.

2.1 Preparing for GDS

At the top of the `ex1.R` file, we do some initial housekeeping, and set the parameters of the GDS sampler.

```
library(plyr)
library(Matrix)
library(mvtnorm)
library(trustOptim)
library(bayesGDS)
source("ex_funcs.R")
set.seed(123)

M <- 20000
ds.scale <- 0.92
n.draws <- 100
max.AR.tries <- 20000
```

The GDS proposal density $g(\theta)$ is a multivariate normal density, with a mean at the posterior mode, and a covariance matrix of a scaled (negative) Hessian at the mode. For a number of computational reasons (e.g., not wanting to invert the Hessian explicitly) bayesGDS works with the precision matrix instead of the covariance. The `ds.scale` corresponds to s in BD; it is the factor by which we

multiply the Hessian by, to get the precision matrix of the proposal density. So, if `ds.scale` were 1, the proposal precision would be equal to the negative Hessian. As `ds.scale` goes down, the proposal density becomes more "diffuse." Set `ds.scale` to be as high as possible, such that $\Phi(\theta|y) \leq 1$ for all of the M proposal draws. If M is large, the approximation to $p(u|y)$ will be more accurate, but it may require `ds.scale` to be smaller, and thus, the acceptance rate of the sampling algorithm may be lower.

The parameter `n.draws` is the number of samples to collect from the posterior distributon (in the rejection sampling phase of the algorithm). `max.AR.tries` is an upper bound on the number of attempts on a single posterior draw. Think of `max.AR.tries` as a "trap" to keep the algorithm from running forever, in case a particular draw of u is so high that the algorithm cannot accept a draw in a reasonable amount of time. If `max.AR.tries` is ever exceeded, try a somewhat less diffuse proposal density.

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
k <- 3
T <- 40

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))
B <- t(rmvnorm(N, mean=mu, sigma=Omega))
XB <- colSums(X * B)
log.p <- XB - log1p(exp(XB))
Y <- laply(log.p, 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 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`) and the gradient (`gr`). In this example, these functions are `log.post` and `get.grad`, respectively.

To use the SparseFD method of `trust.optim`, one also needs to supply the sparsity structure of the Hessian. This is computed by the `get.hess.struct` function (in the `ex_funcs.R` file). See the package manual and documentation for *trustOptim* for more information and the various methods, arguments and control parameters.

```
nvars <- N*k + k
start <- rnorm(nvars) ## random starting values
hess.struct <- get.hess.struct(N, k)  ## sparsity structure of Hessian

opt <- trust.optim(start, fn=log.post,
                   gr = get.grad,
                   hess.struct = hess.struct,
                   method = "SparseFD",
                   control = list(
                     report.freq=1L,
                     maxit=1000L,
                     function.scale.factor = as.numeric(-1),
                     preconditioner=1L
                     ),
                   Y=Y, X=X, inv.Omega=inv.Omega, inv.Sigma=inv.Sigma
                   )

post.mode <- opt$solution
hess <- opt$hessian
log.c1 <- opt$fval
```

2.4 Preparing the GDS sampler

With the posterior mode, and the Hessian at the mode, in hand, we can now start the GDS sampler itself. As mentioned above, the proposal density is a multivariate normal. Instead of using the functions `rmvnorm` and `dmvnorm` from the *mvtnorm* package (Genz et al. 2012), we use two functions that are included in

bayesGDS: `draw.MVN.proposals` and `get.log.dens.MVN`. The reason is that the mvtnorm functions take a full, dense covariance matrix as an argument. Thus, these functions can be slow, and consume a lot of memory, for large problems. The *bayesGDS* functions take the Cholesky decomposition of the precision matrix, either as a dense matrix or in a compressed sparse format (from the *Matrix* package). Given the sparsity of the Hessian for hierarchical models, these functions will be more efficient than `rmvnorm` and `dmvnorm`.

Thus, before doing any GDS sampling, we need to prepare the list of parameters for the MVN sampling and density functions. This list has two elements: `mu` is the mean, and `chol.prec` is the lower triangle Cholesky decomposition of the precision matrix. Since the proposal precision is the scaled Hessian, we multiply the Cholesky decomposition by the square root of the `ds.scale` parameter.

```
chol.hess <- t(chol(-hess))
prop.params <- list(mu = post.mode,
                    chol.prec = sqrt(ds.scale)*chol.hess
                    )
log.c2 <- get.log.dens.MVN(post.mode, prop.params)
log.const <- log.c1 - log.c2
```

Note that the `chol.prec` element in the `prop.params` list must be the *lower* triangle of the Cholesky decomposition. In R, the `chol` function defaults to an upper triangle. Hence, we transpose.

2.5 Computing $q_M(v|y)$

Next, we construct $q_M(v|y)$ which is an 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 $\theta_1 \dots \theta_M$.

```
draws.m <- as(draw.MVN.proposals(M,prop.params),"matrix")
log.post.m <- aaply(draws.m, 2,log.post,
                    Y=Y, X=X, inv.Omega=inv.Omega, inv.Sigma=inv.Sigma,
                    .parallel=FALSE, .progress="text")
log.prop.m <- get.log.dens.MVN(draws.m,params=prop.params)
log.phi <- log.post.m - log.prop.m +log.c2 - log.c1


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

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 `ds.scale`. However, if `ds.scale` is to low, the acceptance rate in the rejection sampling phase may be unacceptably low. The trick is to get `ds.scale` as high as you can, such that $\log \Phi(\theta|y) \leq 0$ for all of the proposals, but not so high that $\log \Phi(\theta|y) > 0$ for too many posterior samples in the rejection sampling phase. There is still no guarantee that $\log \phi(\theta|y) < 0$ for all of proposals in the rejection sampling phase. Don't be too "fine" in choosing `ds.scale`. The decimal places should be plenty.

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.

```
draws <- get.GDS.draws(n.draws = n.draws,
                       log.phi=log.phi,
                       log.const = log.const,
                       log.post.func = log.post,
                       draw.prop.func = draw.MVN.proposals,
                       prop.log.dens.func = get.log.dens.MVN,
                       prop.params = prop.params,
                       max.tries=max.AR.tries,
                       max.batch = 20,
                       est.acc.rate=.05,
                       debug=FALSE,
                       report.freq=10,
                       Y=Y, X=X, inv.Omega=inv.Omega, inv.Sigma=inv.Sigma
                       )
```

The `max.batch` and `est.acc.rate` are tuning parameters to help with memory allocation. It is faster to collect proposal MVN draws in batches, rather than one at a time. The expected number of proposal draws that we need is the number of posterior draws that we need (`n.draws`), divided by the expected acceptance rate (`est.acc.rate`). So it would make sense to draw this number of proposals in a single batch. If it turns out that we need more proposals, we can sample a new batch, using the acceptance rate of earlier draws as a guide for the size of the new batch.

The problem arises when the dimension of the posterior distribution is high, and the acceptance rate is low. For example, if we want to collect 100 independent draws, and we think the acceptance rate is 0.01, the expected batch size 10,000

proposal draws. If the posterior density has 50,000 parameters, and each value uses 8 bytes of RAM, the storage requirement is 4 GB. Depending on the circumstances, batches of this size might be too large, especially if multiple instances of `get.GDS.draws` are run in parallel. The `max.batch` argument controls the number of proposal draws in each batch.

2.7 Output

The `get.GDS.draws` function returns a list with 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. If there are a lot of draws for which gt.1 is 1, you might want to try again with a lower `ds.scale` factor. If there are only a few of these, you could just keep the draw. This would be a case where the posterior density is high, relative to the proposal density, so it would not be totally unreasonable to accept the draw.
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 of the accepted draw

The `draws`, `counts` and `log.post.dens` values are the ones that really matter; the others are primarily diagnostic. The acceptance rate of the algorithm is 1/mean(counts).

If any of the posterior draws has an NA value, that means that the count exceeded `max.AR.tries`. Either increase this value, or adjust the proposal density.

2.8 Log marginal likelihoods

GDS algorithm offers a straightforward way to estimate the log marginal likelihood.

```
LML <- get.LML(draws$counts, log.phi, log.const)
```

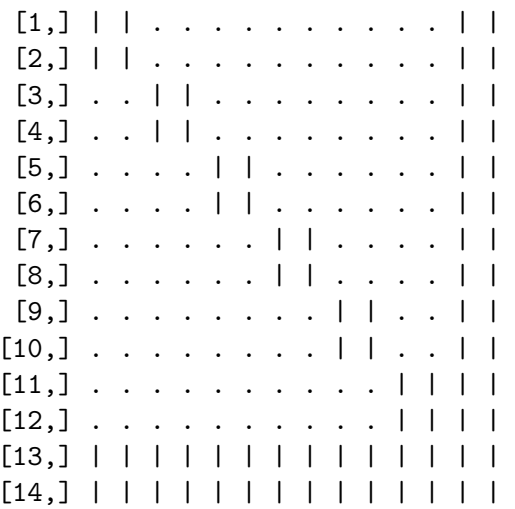
Estimating the marginal likelihood is the only reason we included the normalizing constants in the objective function.

3 Practical advice

Here are some suggestions on how to implement GDS effectively.

3.1 Exploiting sparsity

With Bayesian hierarchical models, under assumptions of conditional independence, the Hessian of the log posterior density has a "block arrow" structure. For example, the example in this note has a Hessian with a form similar to:



In this case, $N = 6$ and $k = 2$. There are 196 elements in this symmetric matrix, but only 169 are non-zero, and only 76 values are unique. If $N = 1000$ instead, 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 (in addition to index pointers that identify the row and column for each element). If you are interested in sampling from a posterior density of a Bayesian hierarchical model with thousands of parameters, chances are the Hessian is sparse.

A standard matrix object in R would store each zero explicity, as a distinct floating point value. So, the memory footprint of storing 4 million elements, at 8 bytes per element, is about 32MB. If $N = 50000$, the matrix uses more than 20GB of RAM. In addition, any mathematical operation on the matrix will operate on all of the

zeros. For something like a matrix-vector multiplication, that's a lot of zeros being multiplied by zero. So when we here statements like "R doesn't work well with large datasets," or "you can't find a posterior mode quickly if you have a lot of parameters," one possible (and common) reason could be that the person is storing data in dense structures.

Fortunately, the idea of exploiting sparsity in numerical computation is not a new idea, and R users have access to a number of tools that are optimized for sparse matrices. The first is the *Matrix* package, which is now a "recommended" package in base R. The *Matrix* package contains classes that define different storage schemes for matrices. The classes vary according to whether the matrix is dense or sparse, numeric, integer, or logical, symmetric or not, triangular or not, and so forth. For example, the "dgCMatrix" class defines a matrix that has double precision, general structure (i.e., not triangular or symmetric), and stored in CSC (compressed sparse column) format. *Matrix* can convert among different classes, and even identify if a matrix is best stored as sparse, symmetric, and so forth. *Matrix* also has linear algebra functions that are optimized for sparse and/or structured matrices, as well as an interface to the CHOLMOD library for sparse Cholesky decompositions (Chen et al. 2008).

Why do we care so much about sparsity? Suppose you want to use a multivariate normal as the GDS proposal density. The most common way to generate MVN draws in a matrix Θ is to transform a matrix of standard normal draws.

$$\Theta = \mu\iota' + LZ \tag{9}$$

where μ is the mean, L is the Cholesky decomposition of the covariance matrix $-H^{-1}$, Z is a matrix of standard normal draws, and ι is a vector of ones. It is easy to show that $\text{cov}(\Theta) = LL' = -H^{-1}$, which is the covariance we want.

To use the `rmvnorm` function in the *mvtnorm* package, you will need to supply a full, dense *covariance* matrix. That means you need to invert the Hessian explicitly, which is not necessarily sparse (even if the Hessian is). And since R does not know that the Hessian is sparse, the inversion and the Cholesky decomposition can each take a long time.

Instead, what if we stored the Hessian H in a sparse compressed format, and took a Cholesky decomposition of that? Let $H = \Lambda\Lambda'$ be that decomposition. Then, consider the the transformation

$$\Theta = \mu\iota' + \Lambda^{'-1}Z \tag{10}$$

Again, the covariance of Θ is H^{-1}. But more importantly, since Λ^{-1} is triangular, we can compute Θ efficiently by solving for $\Theta - \mu\iota'$ using a solver that is optimized for sparse, triangular linear systems. This is exactly what the `draw.MVN.proposal`

function does, and why it takes a sparse representation of Λ (the Cholesky decomposition) as an argument. The `get.log.dens.MVN` function computes the log MVN density in a similar way.

Without sparse Hessian structures, GDS can take a long time to run, not because of potentially low acceptance rates, but because of the time it takes to allocate and manipulate huge, dense arrays. Use sparse matrices, and life is good.

3.2 Posterior modes

When first hearing about GDS, a common reaction is that finding the mode (or modes) of the posterior density is easier said than done. It is true that optimizing a function with $50,000$ parameters using the `optim` function in R is a hopeless endeavor. Algorithms that do not use Hessian information (e.g., Nelder-Mean, conjugate gradient) can converge slowly, especially if the objective function is ill-conditioned or poorly scaled. Algorithms that approximate Hessians, like BFGS, may still iterate slowly if they have to store the approximation as a dense matrix. Also, the BFGS approximations (and those its limited-memory variant L-BFGS-B) may not be very accurate, especially if the objective function is not convex (all BFGS updates are positive definite). In addition, line-search methods (which include conjugate gradient and BFGS) can be unstable when the objective function has ridges or plateaus, or if the approximation to the Hessian is poor. And even worse, the stopping criteria in `optim` are the relative changes in the objective function, which may cause the algorithm to terminate long before the elements of the gradient are all sufficiently close to zero.

I contend that the frustration behind optimizing large-scale problems in R comes from using the wrong tool for the job. Fortunately, there are alternatives for large, sparse, ill-conditioned objective functions. The *trustOptim* package (Braun 2012) finds the minimum of a function using a trust region algorithm. The primary concern is stability, so *trustOptim* should succeed when line search optimizers fail. All else being equal, trust region methods may or may not be faster than line search methods. However, *trustOptim* is optimized for functions with sparse Hessians. Although the user can supply a function that returns the Hessian in a sparse compressed format, all that is really needed is the *structure* of the Hessian. This structure is a list of the row and column indices of the nonzero elements of the lower triangle. In our experience, *trustOptim* has been faster, and more reliable, than any of the other nonlinear optimizers in R that we have tried so far. Please see the documentation and vignette for *trustOptim* for more information and details.

One may also wonder how GDS would handle posterior densities with multiple modes. As with any sampling method (even MCMC), the user has the responsibil-

ity for finding these modes, or at least as many as he can. A simple approach is to run the `trust.optim` function from multiple random starting points, to see if they converge to different local optima, but there are more sophisticated approaches as well (e.g., simulated annealing, genetic algorithms, grid search). One can then estimate the Hessian at each mode, and use a mixture of MVNs as the proposal density. Although there is no guarantee that we can find all of the modes of any function, extant sampling methods like MCMC offer no such guarantees either (at least during a finite period of time).

3.3 Computing gradients and Hessians

The `optim` function in R gives users the option, but not the requirement, to supply a function that returns the gradient of the objective function. Without the explicit gradient, `optim` estimates it using finite differencing. Not only does the time spent estimating the gradient this way grow linearly with the number of parameters, but the accuracy is subject to numerical precision issues, especially near the local optimum where the gradient is close to zero. For derivative-free methods like Nelder-Mead, a gradient is not required, but convergence can be very slow for high-dimensional problems. Therefore, it is usually preferable to write a function that computes the gradient directly.

An alternative is to use what is called "automatic" or "algorithmic" differentiation', abbreviated AD. In short, AD works by having the user write the objective function using a library of specialized numerical types. When an operation between two variables takes place (e.g., multiplication), the types store not only the result of the operation, but additional information that allows for efficient computation of derivatives. The sequence of operations is stored in an object (called a "tape") that can return not only the value of the objective function, but also derivatives of multiple order. In other words, the user needs to code only the objective function, and the routines in the AD library will return gradients, Hessians, and even higher-order objects.

The good news is that there are established AD libraries that are available for C++, Fortran, Matlab, Python, and others (see www.autodiff.org for a list). The bad news is that, at this time, there is no suitable AD library for R. That means that to use AD in R, you need to write the objective function in another language. Fortunately, the *Rcpp* and *RcppEigen* packages offer interfaces with C++ in general, and classes and methods in the *Eigen* numerical library (Guennebaud et al. 2012). I have found Eigen plays nicely with the *CppAD* library of AD routines (Bell 2012).

I believe that it is worth the investment in time to learn how to write objective functions in C++, and how to use the *Eigen*, *RcppEigen* and *CppAD* libraries. The next best option is to derive gradients analytically, and to note the structure of the

Hessian. In that case, both *trustOptim* and *bayesGDS* together should be highly efficient.

References

B.M. Bell. CppAD: a package for C++ algorithmic differentiation. *Computational Infrastructure for Operations Research*, 2012. URL `http://www.coin-or.org/CppAD`.

Michael Braun. trustOptim: a trust-region nonlinear optimizer for R. 2012. URL `http://CRAN.R-project.org/package=trustOptim`.

Michael Braun and Paul Damien. Generalized Direct Sampling for Hierarchical Bayesian Models. August 2012. URL `http://arxiv.org/abs/1108.2245v3`.

Yanqing Chen, Timothy A Davis, William W Hager, and Sivasankaran Rajamanickam. Algorithm 887: CHOLMOD, Supernodal Sparse Cholesky Factorization and Update/Downdate. *ACM Transactions on Mathematical Software*, 35(3):1–14, October 2008. doi: 10.1145/1391989.1391995. URL `http://doi.acm.org/10.1145/1391989.1391995`.

Alan Genz, Frank Bretz, Tetsuhisa Miwa, Xuefei Mi, Friedrich Leisch, Fabian Scheipl, and Torsten Hothorn. mvtnorm: Multivariate normal and t distributions. 2012. URL `http://CRAN.R-project.org/package=mvtnorm`. R package version 0.9-9992.

Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2012.

Hadley Wickham. The split-apply-combine strategy for data analysis. *Journal of Statistical Software*, 40(1):1–29, 2011. URL `http://www.jstatsoft.org/v40/i01/`.