Using the **sparseHessianFD** package

Michael Braun
MIT Sloan School of Management

November 15, 2012

The **sparseHessianFD** package is a tool to compute Hessians efficiently when the Hessian is sparse (that is, a large proportion of the cross-partial derivatives are zero). The user needs to supply the objective function, its gradient, and the sparsity structure of the Hessian. The non-zero elements of the Hessian are computed through finite differencing of the gradients in a way that exploits the sparsity structure. The Hessian is stored in a compressed format (specifically, an object of class `dgCMatrix`, as defined in the **Matrix** package). This allows sparse matrix algorithms to run more quickly, with a smaller memory footprint, than their dense-matrix counterparts. For example, the **trustOptim** package includes an implementation of a trust region nonlinear optimizer that is designed to take advantage of the fact that a Hessian is sparse.

For dense Hessians, a standard way of approximating the Hessian involves taking the differences between the gradient at point $x \in \mathbb{R}^p$ and the gradient with a single element of x perturbed by a small amount ϵ. If $\nabla f(x)$ is the gradient of $f(x)$, then the i^{th}column of the Hessian is equal to $(\nabla(x + \epsilon e_i) - \nabla f(x))/\epsilon$, where e_i is a vector of zeros, with a 1 in the i^{th} element. This "forward differencing" method involves computing a gradient $p + 1$ times. More accurate approximations require even more evaluations of the gradient; central differencing requires $2p$ evaluations. This method also requires the storage of p^2 elements, even if most of the elements of the Hessian are zero.

The **sparseHessianFD** package uses a graph coloring algorithm to partition the p variables into groups ("colors" in the graph theory literature), such that perturbing x_i will not affect the j^{th} element of the gradient for any j that is in the same group

as i. This will happen when the cross-partial derivative with respect to x_i and x_j is zero, or, equivalently, that element (i, j) of the Hessian is zero. This means that we can perturb all of the x's in the same group in a single computation of the gradient. When the number of groups is small, we can estimate the Hessian much more quickly. Note that for a fully dense Hessian, the number of groups is equal to p, and there is no advantage to using this algorithm. Also, the number of groups depends crucially on exactly which elements of the Hessian are non-zero; sparsity does not guarantee that this method can be used. However, for many common sparsity patterns, the computational savings is dramatic.

As an example, suppose that we have, in a hierarchical model, N units, k heterogeneous parameters per unit, and r population-level parameters. Since the cross-partial derivative between an element in β_i and an element in β_j is zero, any element of β_i and β_j can be in the same group, but since the cross partials for elements with a single β_i are not zero, these elements cannot be in the same group. Furthermore, if we assume that any β_i could be correlated with the r population-level parameters, and that the r population-level parameters may be correlated amongst themselves, we can estimate the Hessian (with forward differences) with no more than $k + r + 1$ gradient evaluations. Note that this number *does not grow with N*. Thus, computing the Hessian for a log posterior density of a hierarchical model with, say, 100 heterogeneous units, is no more expensive than for a dataset with a million heterogeneous units, and the amount of storage required for the sparse Hessian grows only linearly in N.

Curtis et al. (1974) introduce the idea of reducing the number of evaluations to estimate sparse Jacobians, and Powell and Toint (1979) describe how to partition variables into appropriate groups, and how to recover Hessian information through back-substitution. Coleman and Moré (1983) show that the task of grouping the variables amounts to a classic graph-coloring problem. Gebremedhin et al. (2005) summarize more recent advances in this area. The actual computational "engine" for **sparseHessianFD** is ACM TOMS Algorithm 636 (Coleman et al. 1985). The original Fortran code is in the file inst/include/misc/FDHS-DSSM.f. The file src/FDHS-DSSM.c is a translation of the original Fortran code into C. The copyright to both of these files is retained by the Association of Computational Machinery under terms that are included in the LICENSE file in the package source code. My contribution to the package is only the interface with R, and not

the computational algorithm itself.

1 Using the package

Using **sparseHessianFD** involves constructing an object of class `sparseHessianObj`. The class `sparseHessianObj` contains only one slot: an external pointer to an instance of a C++ class that does all of the computation. This object stores all of the information needs to compute the objective function, gradient and Hessian for any argument vector x. The easiest way to compute this object is to use the `new.sparse.hessian.obj` function. Its signature is:

```
obj <- get.new.sparse.hessian(x, fn, gr, hs, fd.method=0,
                              eps=sqrt(.machine$double.eps), ...)
```

The function `fn` returns $f(x)$, the value of the objective function to be minimized, `gr` that returns the gradient. Both functions can take additonal named arguments which are passed through the `...` argument in `get.new.sparse.hessian`. The argument `hs` is a list that represents the sparsity structure of the Hessian. The `hs` list contains two integer vectors,`iRow` and `jCol`, that contain the row and column indices of the non-zero elements of the lower triangle of the Hessian. The length of each of these vectors is equal to the number of non-zeros in the lower triangle of the Hessian. Do *not* include any elements from the upper triangle. Entries must be in order, first by column, and then by row within each column. Indexing starts at 1. The package includes a convenience function, `Matrix.to.Coord`, that converts a matrix with the appropriate sparsity structure to a list that can be used as the `hs` argument.

Coleman et al. (1985) provides two approaches for computing a sparse Hessian: indirect (`fd.method=0`) and direct (`fd.method=1`). We refer the reader to that source for an explanation of the difference. In short, the indirect method should be somewhat faster that the direct method, with comparable accuracy. The argument `eps` is the perturbation about used in the finite differencing algorithm. Again, see Coleman et al. (1985) for more details.

The algorithms in this package work best when the gradient is computed directly (i.e., derived analytically or symbolically), or otherwise computed exactly (say,

by way of algorithmic differentiation). In general, we never recommend finite-differenced gradients. Finite differencing takes a long time to run, and is subject to numerical error, especially near the optimum when elements of the gradient are close to zero. Using **sparseHessianFD** with finite-differenced gradients means that the Hessian is "doubly differenced," and the resulting lack of numerical precision makes those Hessians nearly worthless.

Once the `sparseHessianObj` object is constructed at an initial value of x, we can then compute the function, gradient or Hessian for any other value of x. The **sparseHessianFD** includes the following methods:

```
get.fn(x, obj)
get.gr(x, obj)
get.hessian(x, obj)
get.fngr(x, obj)
```

These functions return `fn(x)`, `gr(x)`, the Hessian of `fn(x)`, and a list with both `fn(x)` and `gr(x)`, respectively. The Hessian is an object of class `dgCMatrix`. [1] These functions do not pass additional arguments to the original functions, since that information is stored in `obj`.

Alternatively, we can access the function, gradient, and Hessian functions directly from the object with:

```
obj$fn(x)
obj$gr(x)
obj$hessian(x)
obj$fngr(x)
```

2 Sparsity structure of the Hessian

In the following code, we construct a block diagonal matrix, and then use the `Matrix.to.Coord` function to generate a list of the row and column indices of the non-zero elements of the lower triangle.

[1] Even though the Hessian is symmetric, the `dgCMatrix` stores the entire matrix, and not just the lower triangle. This is because of a current limitation in the **RcppEigen** package. As **RcppEigen** functionality expands, we hope to return Hessians as `dsCMatrix` objects. This would effectively halve the storage requirements for the Hessian.

```
require(Matrix)
M <- kronecker(Diagonal(4),Matrix(1,2,2))
print(M)
8 x 8 sparse Matrix of class "dgTMatrix"
[1,] 1 1 . . . . . .
[2,] 1 1 . . . . . .
[3,] . . 1 1 . . . .
[4,] . . 1 1 . . . .
[5,] . . . . 1 1 . .
[6,] . . . . 1 1 . .
[7,] . . . . . . 1 1
[8,] . . . . . . 1 1

H <- Matrix.to.Coord(M)
print(H)
$iRow
 [1] 1 2 2 3 4 4 5 6 6 7 8 8

$jCol
 [1] 1 1 2 3 3 4 5 5 6 7 7 8
```

To check that the indices do, in fact, represent the sparsity pattern of the lower triangular Hessian, you can convert the list back to a pattern `Matrix` using the `Coord.to.Matrix` function.

```
M2 <- Coord.to.Pattern.Matrix(H, 8,8)
print(M2)
8 x 8 sparse Matrix of class "ngCMatrix"

[1,] | . . . . . . .
[2,] | | . . . . . .
[3,] . . | . . . . .
[4,] . . | | . . . .
[5,] . . . . | . . .
[6,] . . . . | | . .
[7,] . . . . . . | .
[8,] . . . . . . | |
```

Notice that M2 is only lower-triangular. Even though M was symmetric, H contains only the indices of the non-zero elements in the lower triangle. To recover the pattern of the *symmetric* matrix, use the `Coord.to.Sym.Pattern.Matrix` function.

```
M3 <- Coord.to.Sym.Pattern.Matrix(H,8)
print(M3)
8 x 8 sparse Matrix of class "nsTMatrix"

[1,] | | . . . . . .
[2,] | | . . . . . .
[3,] . . | | . . . .
[4,] . . | | . . . .
[5,] . . . . | | . .
[6,] . . . . | | . .
[7,] . . . . . | |
[8,] . . . . . | |
```

3 An example

As an example,let's compute the Hessian of the log posterior density of a hierarchical model. 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

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

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_0, 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, ignoring any normalization constants, is

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

Since the β_i are drawn iid from a multivariate normal, $\frac{\partial^2 \log \pi}{\partial \beta_i \beta_j} = 0$ for all $i \neq j$. We also know that all of the β_i are correlated with μ. Therefore, the Hessian will be sparse with a "block-arrow" structure. For example, if $N = 6$ and $k = 2$, then $p = 14$ and the Hessian will have the pattern as illustrated in Figure 1.

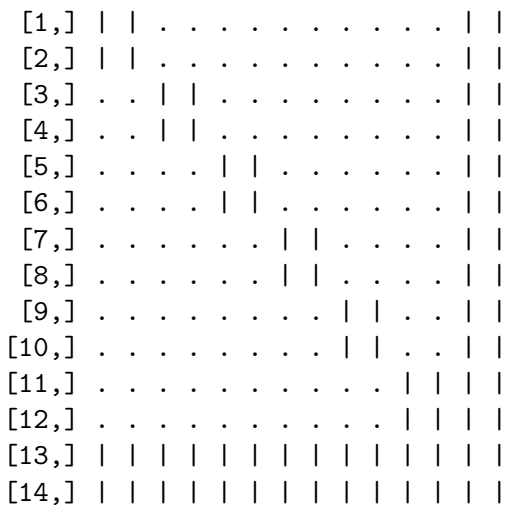

Figure 1: Sparsity pattern for hierarchical binary choice example.

There are 196 elements in this symmetric matrix, but only 169 are non-zero, and only 76 values are unique. Although the reduction in RAM 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 (e.g., through a Cholesky decomposition), we only need to work with only 7,003 values.

The file `inst/examples/example.R` demonstrates how to estimate the Hessian for this model. The function, gradient, and "true" Hessian are computed using func-

tions in the file `inst/examples/ex_funcs.R`). In `example.R`, we first simulate some data. The `hess.struct` function returns the list than can be used for the `hs` argument in the `get.new.sparse.hessian` function.

We then create `obj` using the defaults for `fd.method` and `eps`. Finally, we compute the function, gradient and Hessian using the two different methods on `obj`.

The `get.hess` function (defined in `ex_funcs.R`) returns the exact Hessian, derived analytically. You can see that the Hessian is the same as the one that is computed by way of `get.hessian`.

4 Multivariate normal distribution with sparse precision matrices

The `rmvnorm` and `dmvnorm` functions in the **mvtnorm** package (Genz et al. 2012) are functions that sample from, and compute the density of, a multivariate normal distribution with mean μ and covariance Σ. Σ must be a "base" matrix; that is, dense. If the covariance matrix is sparse, these functions are inefficient for several reasons.

1. We need to store the entire covariance matrix densely, even if most of the elements are non-zero;
2. There are Cholesky decomposition algorithms that are optimized for sparse matrices that `rmvnorm` and `dmvnorm` cannot exploit;
3. These functions will perform a new decomposition every time it is called; and
4. We often have the precision matrix readily available (e.g., the Hessian at a posterior mode), which we would need to invert explicitly if we wanted to use `rmvnorm` or `dmvnorm`.

The **sparseHessianFD** package includes two functions that are more efficient for cases in which we have a sparse covariance or precision matrix: `rmvn.sparse` and `dmvn.sparse`. In the **Matrix** package, `chol` calls Cholesky decomposition algorithm that is optimized for sparse matrices, so we can use that to decompose

the covariance or precision matrix before calling `rmvn.sparse` or `dmvn.sparse`. However, the return value of `chol` is an upper triangular matrix, so you will need to remember to transpose and coerce it to a lower triangular matrix.

The details for using these functions are in the package documentation. There is also an example in the file `inst/examples/mvn.R`. This example simulates a block diagonal covariance matrix and uses that to draw from a multivariate normal. We also compare the results and run times with calls to `rmvnorm` and `dmvnorm`. Even for modestly-sized distributions, the sparse functions in **sparseHessianFD** are dramatically faster than their dense counterparts from **mvtnorm**.

5 Discussion points

For many functions, like log posterior densities, deriving and coding a gradient analytically is straightforward (either by hand, or using a symbolic computation tool like Mathematica). Analytic Hessians can be very messy to derive and code, and even then, storing and working with a $p \times p$ matrix is expensive when p is large. The **sparseHessianFD** package is useful when the Hessian is sparse and the sparsity structure is known in advance, even when p is massively large. The speed at which **sparseHessianFD** computes the Hessian depends crucially on the sparsity structure. For block diagonal Hessians, as in the example above, computation time will grow with the size of each heterogeneous parameter, and the number of population-level parameters, but not with the number of heterogeneous units. As N grows, the number of non-zero elements in the Hessian grows linearly, and the number of gradient differences that need to be computed is constant.

We should note that finite differencing is not the current "state of the art" for estimating sparse Hessians. Algorithmic differentiation (AD) packages can be faster and more exact (and of course they can compute the gradient as well). A critical requirement of an AD package when we need to differentiate scalar-valued functions with large p is that it support "reverse-mode" differentiation. For C++, **CppAD** and **Adol-C** are popular choices, and others may be available for Matlab and Python. However, to my knowledge there is not yet AD library available for R, or at least one that supports reverse mode.

What this means is that even though finite differencing introduces numerical error into calculations of the Hessian, it is still the best alternative when programming in R. The **sparseHessianFD** package, as an interface to ACM TOMS Algorithm 636 (Coleman et al. 1985) takes away much of the pain when the Hessian is sparse and the sparsity structure is known.

References

Thomas F Coleman and Jorge J Moré. Estimation of Sparse Jacobian Matrices and Graph Coloring Problems. *SIAM Journal on Numerical Analysis*, 20(1):187–209, February 1983.

Thomas F Coleman, Burton S Garbow, and Jorge J Moré. Software for Estimating Sparse Hessian Matrices. *ACM Transaction on Mathematical Software*, 11(4):363–377, December 1985.

A R Curtis, M J D Powell, and J K Reid. On the Estimation of Sparse Jacobian Matrices. *Journal of the Institute of Mathematics and its Applications*, 13:117–119, 1974.

Assefaw Hadish Gebremedhin, Fredrik Manne, and Alex Pothen. What Color is your Jacobian? Graph Coloring for Computing Derivatives. *SIAM Review*, 47(4): 629–705, 2005.

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.

M J D Powell and Ph. L. Toint. On the Estimation of Sparse Hessian Matrices. *SIAM Journal on Numerical Analysis*, 16(6):1060–1074, December 1979.