Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. *http://www.jstatsoft.org/*

sparseHessianFD: An R Package for Estimating Sparse Hessian Matrices

Michael Braun
Edwin L. Cox School of Business
Southern Methodist University

Abstract

Sparse Hessian matrices occur often in statistics, and their fast and accurate estimation can improve efficiency of numerical optimization and sampling algorithms. By exploiting the known sparsity pattern of a Hessian, methods in the sparseHessianFD package require many fewer function or gradient evaluations than would be required if the Hessian were treated as dense. The package implements established graph coloring and linear substitution algorithms that were previously unavailable to R users, and is most useful when other numerical, symbolic or algorithmic methods are impractical, inefficient or unavailable.

Keywords: sparse Hessians, sparsity, computation of Hessians, graph coloring, finite differences, differentiation.

The Hessian matrix of a log likelihood function or log posterior density function plays an important role in statistics. From a frequentist point of view, the inverse of the negative Hessian is the asymptotic covariance of the sampling distribution of a maximum likelihood estimator. In Bayesian analysis, when evaluated at the posterior mode, it is the covariance of a Gaussian approximation to the posterior distribution. More broadly, many numerical optimization algorithms require repeated computation, estimation or approximation of the Hessian or its inverse; see Nocedal and Wright (2006).

The Hessian of an objective function with M variables has M^2 elements, of which $M(M+1)/2$ are unique. Thus, the storage requirements of the Hessian, and computational cost of many linear algebra operations on it, grow quadratically with the number of decision variables. For applications with hundreds of thousands of variables, computing the Hessian even once might not be practical under time, storage or processor constraints. Hierarchical models, in which each additional heterogeneous unit is associated with its own subset of variables, are particularly vulnerable to this curse of dimensionality

However, for many problems, the Hessian is *sparse*, meaning that the proportion of "structural"

zeros (Hessian matrix elements that are always zero, regardless of the value at which function is estimated) is high. Consider a log posterior density in a Bayesian hierarchical model. If the outcomes across units are conditionally independent, the cross-partial derivatives of heterogeneous variables across units are zero. As the number of units increases, the size of the Hessian still grows quadratically, but the number of *non-zero* elements grows only linearly; the Hessian becomes increasingly sparse. The row and column indices of the non-zero elements comprise the *sparsity pattern* of the Hessian, and are typically known in advance, before computing the values of those elements. R packages such as **trustOptim** (Braun 2014), **sparseMVN** (Braun 2015) and **ipoptr** (Wächter and Biegler 2006) have the capability to accept Hessians in a compressed sparse format.

The **sparseHessianFD** package is a tool for estimating sparse Hessians using finite differences of gradients. Section 1.1 will cover the specifics, but the basic idea is as follows. Consider a function $f(x)$, its gradient $\nabla f(x)$, and its Hessian $\mathsf{H}f(x)$, for $x \in \mathbb{R}^M$. Define the derivative vector as the transpose of the gradient, and a vector of partial derivatives, so $\mathsf{D}f(x) = \nabla f(x)^\top = (\mathsf{D}_1, \ldots, \mathsf{D}_M)$. (Throughout the paper, we will try to reduce notational clutter by referring to the derivative and Hessian as D and H, respectively, without the $f(x)$ symbol). Let e_m be a vector of zeros, except with a 1 in the mth element, and let δ be a sufficiently small scalar constant. A linear approximation to the mth column of the Hessian is $\mathsf{H}_m \approx (\nabla f(x + \delta e_m) - \nabla f(x))/\delta$. Estimating a dense Hessian in this way involves at least $M + 1$ calculations of the gradient: one for the gradient at x, and one after perturbing each of the M elements of x, one at a time.[1] In a sparse matrix, most of the elements are constrained to zero. Depending on the sparsity pattern of the Hessian, those constraints may let us recover the Hessian with fewer gradient evaluations by perturbing multiple elements of x together. For some sparsity patterns, estimating a Hessian in this way can be profoundly efficient. In fact, for the hierarchical models that we consider in this paper, the number of gradient evaluations does not increase with the number of additional heterogeneous units.

The package defines the *sparseHessianFD* class, whose initializer requires the user to provide functions that compute an objective function, its gradient, and the sparsity pattern of its Hessian matrix. The gradient should be computed exactly (within machine precision), as opposed to being numerically estimated (e.g., by finite differencing). The sparsity pattern (e.g., location of structural zeros) must be known in advance, and cannot vary across the domain of the objective function. The only functions and methods of the class that the end user should need to use are the initializer, methods that return the Hessian in a sparse compressed format, and perhaps some utility functions that simplify the construction of the sparsity pattern. The class also defines methods that partition the variables into groups that can be perturbed together in a finite differencing step, and recovers the elements of the Hessian via linear substitution. Those methods perform most of the work, but should be invisible to the user.

As with any computing method or algorithm, there are boundaries around the space of applications for which **sparseHessianFD** is the right tool for the job. In general, finite differencing is not a "first choice" method because the result is not exact, so **sparseHessianFD** should not be used when the application cannot tolerate any error, no matter how small. Also, we admit that some users might balk at having to provide an exact gradient, even though the Hessian will be estimated numerically. However, deriving a vector of first derivatives, and

[1] More accurate approximations require more gradient evaluations. We will consider only forward finite differences.

writing R functions to compute them, is a lot easier than doing the same for a matrix of second derivatives, and more accurate than computing second-order approximations from the objective function. Even when we have derived the Hessian symbolically, in practice it may still be faster to estimate the Hessian using **sparseHessianFD** than coding it directly. These are the situations in which **sparseHessianFD** adds the most value to the statistician's toolbox.

This article proceeds as follows. First, we present some background information about numerical differentiation, and sparse matrices in R, in Section 1. In Section 2, we explain how to use the package. Section 3 explains the underlying algorithms, and Section 4 demonstrates the scalability of those algorithms.

1. Background

Before describing how to use the package, we present two short background notes. The first note is an informal mathematical explanation of numerical estimation of the Hessian matrix, with an illustration of how the number of gradient estimates can be reduced by exploiting the sparsity pattern and symmetric structure. This note borrows heavily from, and uses the notation in, Magnus and Neudecker (2007, Chapter 6). The second note is a summary of some of the sparse matrix classes that are defined in the **Matrix** package (Bates and Maechler 2015), which are used extensively in **sparseHessianFD**.

1.1. Numerical differentiation of sparse Hessians

The partial derivative of a scalar-valued function $f(x)$ with respect to x_j (the jth component of x) is defined as

$$\mathsf{D}_j f(x) = \lim_{\delta \to 0} \frac{f(x + \delta e_j) - f(x)}{\delta} \tag{1}$$

For a sufficiently small δ, this definition allows for a linear approximation to $\mathsf{D}_j f(x)$. The derivative of $f(x)$ is the vector of all M partial derivatives.

$$\mathsf{D} f(x) = (\mathsf{D}_1 f(x), \ldots, \mathsf{D}_M f(x)) \tag{2}$$

The *gradient* is defined as $\nabla f(x) = \mathsf{D} f(x)^\top$.

We define the second-order partial derivative as

$$\mathsf{D}^2_{jk} = \lim_{\delta \to 0} \frac{\mathsf{D}_j f(x + \delta e_k) - \mathsf{D}_j f(x)}{\delta} \tag{3}$$

and the Hessian as

$$\mathsf{H} f(x) = \begin{pmatrix} \mathsf{D}^2_{11} & \mathsf{D}^2_{12} & \ldots & \mathsf{D}^2_{1M} \\ \mathsf{D}^2_{21} & \mathsf{D}^2_{22} & \ldots & \mathsf{D}^2_{2M} \\ \vdots & \vdots & & \vdots \\ \mathsf{D}^2_{M1} & \mathsf{D}^2_{M2} & \ldots & \mathsf{D}^2_{MM} \end{pmatrix} \tag{4}$$

The Hessian is symmetric, so $\mathsf{D}^2_{ij} = \mathsf{D}^2_{ji}$.

To estimate the mth column of H, we again choose a sufficiently small δ, and compute

$$\mathsf{H}_m f(x) \approx \frac{\mathsf{D} f(x + \delta e_m) - \mathsf{D} f(x)}{\delta} \tag{5}$$

For $M = 2$, our estimate of a general $\mathsf{H} f(x)$ would be

$$\mathsf{H} f(x) = \begin{pmatrix} \mathsf{D}_1 f(x_1 + \delta, x_2) - \mathsf{D}_1 f(x_1, x_2) & \mathsf{D}_1 f(x_1, x_2 + \delta) - \mathsf{D}_1 f(x_1, x_2) \\ \mathsf{D}_2 f(x_1 + \delta, x_2) - \mathsf{D}_2 f(x_1, x_2) & \mathsf{D}_2 f(x_1, x_2 + \delta) - \mathsf{D}_2 f(x_1, x_2) \end{pmatrix} / \delta \tag{6}$$

This estimate requires three evaluations of the gradient to get $\mathsf{D} f(x_1, x_2)$, $\mathsf{D} f(x_1 + \delta, x_2)$, and $\mathsf{D} f(x_1, x_2 + \delta)$.

Now suppose that the Hessian is sparse, and that the off-diagonal elements are zero. That means that

$$\mathsf{D}_1 f(x_1, x_2 + \delta) - \mathsf{D}_1 f(x_1, x_2) = 0 \tag{7}$$

$$\mathsf{D}_2 f(x_1 + \delta, x_2) - \mathsf{D}_2 f(x_1, x_2) = 0 \tag{8}$$

If the identity in Equation 7 holds for x_1, it must also hold for $x_1 + \delta$, and if Equation 8 holds for x_2, it must also hold for $x_2 + \delta$. Therefore,

$$\mathsf{H} f(x) = \begin{pmatrix} \mathsf{D}_1 f(x_1 + \delta, x_2 + \delta) - \mathsf{D}_1 f(x_1, x_2) & 0 \\ 0 & \mathsf{D}_2 f(x_1 + \delta, x_2 + \delta) - \mathsf{D}_2 f(x_1, x_2) \end{pmatrix} / \delta \tag{9}$$

Only two gradients, $\mathsf{D} f(x_1, x_2)$ and $\mathsf{D} f(x_1 + \delta, x_2 + \delta)$, are needed. Being able to reduce the number of gradient evaluations from 3 to 2 depends on knowing that the cross-partial derivatives are zero.

Curtis, Powell, and Reid (1974) describe a method of estimating sparse Jacobian matrices by perturbing groups of variables together. Powell and Toint (1979) extend this idea to the general case of sparse Hessians. This method partitions the decision variables into C mutually exclusive groups so that the number of gradient evaluations is reduced. Define $\mathbf{G} \in \mathbb{R}^{M \times C}$ where $\mathbf{G}_{mc} = \delta$ if variable m belongs to group c, and zero otherwise. Define $\mathbf{G}_c \in \mathbb{R}^M$ as the cth column of $\mathbf{G}$.

Next, define $\mathbf{Y} \in \mathbb{R}^{M \times C}$ such that each column is a difference in gradients.

$$\mathbf{Y}_c = \nabla f(x + G_c) - \nabla f(x) \tag{10}$$

If $C = M$, then $\mathbf{G}$ is a diagonal matrix with δ in each diagonal element. The matrix equation $\mathsf{H}\mathbf{G} = \mathbf{Y}$ represents the linear approximation $\mathsf{H}_{im}\delta \approx y_{im}$, and we can solve for all elements of H just by computing Y. But if $C < M$, there must be at least one $\mathbf{G}_c$ with δ in at least two rows. The corresponding column $\mathbf{Y}_c$ is computed by perturbing multiple variables at once, so we cannot solve for any H_{im} without further constraints.

These constraints come from the sparsity pattern and symmetry of the Hessian. Consider an example with the following values and sparsity pattern.

$$\mathsf{H} f(x) = \begin{pmatrix} h_{11} & 0 & h_{31} & 0 & 0 \\ 0 & h_{22} & 0 & h_{42} & 0 \\ h_{31} & 0 & h_{33} & 0 & h_{53} \\ 0 & h_{42} & 0 & h_{44} & 0 \\ 0 & 0 & h_{53} & 0 & h_{55} \end{pmatrix} \tag{11}$$

Suppose $C = 2$, and define group membership of the five variables through the following $\mathbf{G}$ matrix.

$$\mathbf{G}^\top = \begin{pmatrix} \delta & \delta & 0 & 0 & \delta \\ 0 & 0 & \delta & \delta & 0 \end{pmatrix} \tag{12}$$

Variables 1, 2 and 5 are in group 1, while variables 3 and 4 are in group 2.

Next, compute the columns of $\mathbf{Y}$ using Equation 10. We now have the following system of linear equations from $\mathsf{H}\mathbf{G} = \mathbf{Y}$.

$$\begin{aligned} h_{11} &= y_{11} & h_{31} &= y_{12} \\ h_{22} &= y_{21} & h_{42} &= y_{22} \\ h_{31} + h_{53} &= y_{31} & h_{33} &= y_{32} \\ h_{42} &= y_{41} & h_{44} &= y_{42} \\ h_{55} &= y_{51} & h_{53} &= y_{52} \end{aligned} \tag{13}$$

Note that this system is overdetermined. Both $h_{31} = y_{12}$ and $h_{53} = y_{52}$ can be determined directly, but $h_{31} + h_{53} = y_{31}$ may not necessarily hold, and h_{42} could be either y_{41} or y_{22}. Powell and Toint (1979) prove that it is sufficient to solve $\mathsf{L}\mathbf{G} = \mathbf{Y}$ instead via a substitution method, where L is the lower triangular part of H. This has the effect of removing the equations $h_{42} = y_{22}$ and $h_{31} = y_{12}$ from the system, but retaining $h_{53} = y_{52}$. We can then solve for $h_{31} = y_{31} - y_{52}$. Thus, we have determined a 5×5 Hessian with only three gradient evaluations, in contrast with the six that would have been needed had H been treated as dense.

The **sparseHessianFD** algorithms assign variables to groups before computing the values of the Hessian. This is why the sparsity pattern needs to be provided in advance. If a non-zero element is omitted from the sparsity pattern, the resulting estimate of the Hessian will be incorrect. The only problems with erroneously including a zero element in the sparsity pattern are a possible lack of efficiency (e.g., an increase in the number of gradient evaluations), and that the estimated value might be close to, but not exactly, zero. The algorithms for assigning decision variables to groups, and for extracting nonzero Hessian elements via substitution, are described in Section 3.

1.2. Sparse matrices and the Matrix package

The **sparseHessianFD** package uses the sparse matrix classes that are defined in the **Matrix** package (Bates and Maechler 2015). All of these classes are subclasses of *sparseMatrix*. Only the row and column indices (or pointers to them), the non-zero values, and some meta-data, are stored; unreferenced elements are assumed to be zero. Class names, summarized in Table 1, depend on the data type, matrix structure, and storage format. Values in numeric and logical matrices correspond to the R data types of the same names. Pattern matrices contain row and column information for the non-zero elements, but no values. The storage format refers to the internal ordering of the indices and values, and the layout defines a matrix as symmetric (so duplicated values are stored only once), triangular, or general. The levels of these three factors determine the prefix of letters in each class name. For example, a triangular sparse matrix of numeric (double precision) data, stored in column-compressed format, has a class *dtCMatrix*.

Storage	Layout	Data type		
		numeric	logical	pattern
Triplet	general	*dgTMatrix*	*lgTMatrix*	*ngTMatrix*
	triangular	*dtTMatrix*	*ltTMatrix*	*ntTMatrix*
	symmetric	*dsTMatrix*	*lsTMatrix*	*nsTMatrix*
Row-compressed	general	*dgRMatrix*	*lgRMatrix*	*ngRMatrix*
	triangular	*dtRMatrix*	*ltRMatrix*	*ntRMatrix*
	symmetric	*dsRMatrix*	*lsRMatrix*	*nsRMatrix*
Column-compressed	general	*dgCMatrix*	*lgCMatrix*	*ngCMatrix*
	triangular	*dtCMatrix*	*ltCMatrix*	*ntCMatrix*
	symmetric	*dsCMatrix*	*lsCMatrix*	*nsCMatrix*

Table 1: Class names for sparse matrices, as defined in the Matrix package.

Matrix also defines some other classes of sparse and dense matrices that we will not discuss here. The **Matrix** package uses the `as` function to convert sparse matrices from one format to another, and to convert a **base** R matrix to one of the **Matrix** classes.

The distinctions among sparse matrix classes is important because **sparseHessianFD**'s `hessian` method returns a *dgCMatrix*, even though the Hessian is symmetric. Depending on how the Hessian is used, it might be useful to coerce the Hessian into a *dsCMatrix* object. Also, the utility functions in Table 2 expect or return certain classes of matrices, so some degree of coercion of input and output might be necessary. Another useful **Matrix** function is `tril`, which extracts the lower triangle of a general or symmetric matrix.

2. Using the package

In this section, we demonstrate how to use the **sparseHessianFD**, using a hierarchical binary choice model as an example. Then, we discuss the sparsity pattern of the Hessian, and estimate the Hessian values.

2.1. Example model: hierarchical binary choice

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, and let p_i be the probability of purchase. The heterogeneous parameter p_i is the same for all T opportunities, so y_i is a binomial random variable.

Let $\beta_i \in \mathbb{R}^k$ be a heterogeneous coefficient vector that is specific to household i, such that $\beta_i = (\beta_{i1}, \ldots, \beta_{ik})$. Similarly, $z_i \in \mathbb{R}^k$ is a vector of household-specific covariates. Define each p_i such that the log odds of p_i is a linear function of β_i and z_i, but does not depend directly on β_j and z_j for another household $j \neq i$.

$$p_i = \frac{\exp(z_i'\beta_i)}{1 + \exp(z_i'\beta_i)}, \; i = 1...N \tag{14}$$

The coefficient vectors β_i are distributed across the population of households following a

multivariate normal distribution with mean $\mu \in \mathbb{R}^k$ and covariance $\mathbf{\Sigma} \in \mathbb{R}^{k \times k}$. Assume that we know $\mathbf{\Sigma}$, but not μ, so we place a multivariate normal prior on μ, with mean 0 and covariance $\mathbf{\Omega} \in \mathbb{R}^{k \times k}$. Thus, the parameter vector $x \in \mathbb{R}^{(N+1)k}$ consists of the Nk elements in the N β_i vectors, and the k elements in μ.

The log posterior density, ignoring any normalization constants, is

$$\log \pi(\beta_{1:N}, \mu | \mathbf{Y}, \mathbf{Z}, \mathbf{\Sigma}, \mathbf{\Omega}) = \sum_{i=1}^{N} \left(p_i^{y_i} (1 - p_i)^{T - y_i} - \frac{1}{2} (\beta_i - \mu)^\top \mathbf{\Sigma}^{-1} (\beta_i - \mu) \right) - \frac{1}{2} \mu^\top \mathbf{\Omega}^{-1} \mu \tag{15}$$

2.2. Sparsity patterns

Let x_1 and x_2 be two subsets of elements of x. Define $\mathsf{D}^2_{x_1,x_2}$ as the product set of cross-partial derivatives between all elements in x_1 and all elements in x_2. Furthermore, define an indexing function $\text{Ind}(x_i)$ that returns the indices in the parameter vector x that correspond to the elements in x_i, and H_{x_1,x_2} as the Hessian elements in rows $\text{Ind}(x_1)$ and columns $\text{Ind}(x_2)$, and in rows $\text{Ind}(x_2)$ and columns $\text{Ind}(x_1)$ (from the symmetry of the Hessian).

From the log posterior density in Equation 15, we can see that $\mathsf{D}^2_{\beta_i,\beta_i} \neq 0$ (one element of β_i could be correlated with another element of β_i), and that, for all i, $\mathsf{D}^2_{\beta_i,\mu} = 0$ (because μ is the prior mean of each β_i). However, since the β_i and β_j are independently distributed, and the y_i are conditionally independent, the cross-partial derivatives $\mathsf{D}^2_{\beta_i,\beta_j} = 0$ for all $i \neq j$. When N is much greater than k, there will be many more zero cross-partial derivatives than non-zero. Regardless of how each D^2 is mapped to H through the indexing function, the Hessian of the log posterior density will be sparse.

The sparsity pattern depends on the indexing function; that is, on how the variables are ordered in x. One such ordering is to group all of the coefficients in the β_i for each unit together.

$$\beta_{11}, \ldots, \beta_{1k}, \beta_{21}, \ldots, \beta_{2k}, \ \ldots \ , \beta_{N1}, \ldots, \beta_{Nk}, \mu_1, \ldots, \mu_k \tag{16}$$

In this case, the Hessian has a "block-arrow" structure. For example, if $N = 5$ and $k = 2$, then there are 12 total variables, and the Hessian will have the pattern in Figure 1a.

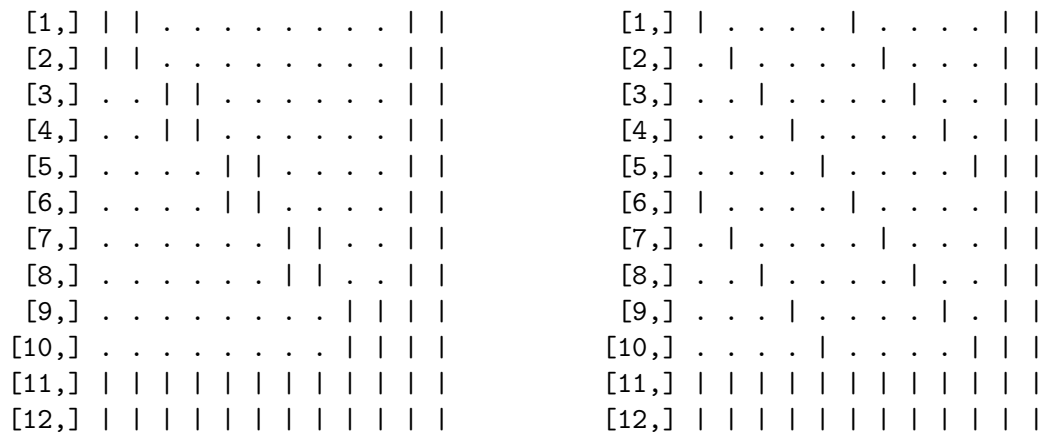
Another possibility is to group coefficients for each covariate together.

$$\beta_{11}, \ldots, \beta_{N1}, \beta_{12}, \ldots, \beta_{N2}, \ \ldots \ , \beta_{1k}, \ldots, \beta_{Nk}, \mu_1, \ldots, \mu_k \tag{17}$$

Now the Hessian has an "banded" sparsity pattern, as in Figure 1b.

In both cases, the number of non-zeros is the same. There are 144 elements in this symmetric matrix, but only 64 are non-zero, and only 38 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 $N = 1,000$ 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, then we need to work with only 7,003 values.

The sparsity pattern required by **sparseHessianFD** consists of the row and column indices of the non-zero elements in the *lower triangle* the Hessian, and it is the responsibility of the



(a) A "block-arrow" sparsity pattern. (b) A "banded" sparsity pattern.

Figure 1: Two examples of sparsity patterns for a hierarchical model.

`Matrix.to.Coord`	Returns a list of vectors containing row and column indices of the non-zero elements of a matrix.
`Matrix.to.Pointers`	Returns indices and pointers from a sparse matrix.
`Coord.to.Pointers`	Converts list of row and column indices (triplet format) to a list of indices and pointers (compressed format).

Table 2: sparseHessianFD matrix conversion functions.

user to ensure that the pattern is correct. In practice, rather than trying to keep track of the row and column indices directly, it might be easier to construct a pattern matrix first, check visually that the matrix has the right pattern, and then extract the indices.

The package defines utility functions (Table 2) to convert between sparse matrices, and the vectors of row and column indices required by the *sparseHessianFD* initializer.

The `Matrix.to.Coord` function extracts row and column indices from a sparse matrix. The following code constructs a logical block diagonal matrix, converts it to a sparse matrix, and prints the sparsity pattern of its lower triangle.

```
R> library("sparseHessianFD")
R> bd <- kronecker(diag(3), matrix(TRUE, 2, 2))
R> Mat <- as(bd, "nMatrix")
R> printSpMatrix(tril(Mat))


[1,] | . . . . .
[2,] | | . . . .
[3,] . . | . . .
[4,] . . | | . .
[5,] . . . . | .
[6,] . . . . | |

R> mc <- Matrix.to.Coord(tril(Mat))
R> mc

$rows
[1] 1 2 2 3 4 4 5 6 6

$cols
[1] 1 1 2 3 3 4 5 5 6
```

To check that a proposed sparsity pattern represents the intended matrix visually, use the **Matrix** `sparseMatrix` constructor.

```
R> pattern <- sparseMatrix(i=mc$rows, j=mc$cols)
R> printSpMatrix(pattern)


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

2.3. The *sparseHessianFD* class

The function `sparseHessianFD` is an initializer that returns a reference to a *sparseHessianFD* object. The initializer determines an appropriate permutation and partitioning of the variables, and performs some additional validation tests. The arguments to the initializer are described in Table 3.

To create a *sparseHessianFD* object, just call `sparseHessianFD`. Applying the default values for the optional arguments, the usage syntax to create a *sparseHessianFD* object is

```
obj <- sparseHessianFD(x, fn, gr, rows, cols, ...)
```

x	A numeric vector, with length M at which the object will be initialized and tested.
fn,gr	R Functions that return the value of the objective function, and its gradient. The first argument is the numeric variable vector. Other named arguments can be passed to `fn` and gr as well (see the `...` argument below).
rows, cols	Sparsity pattern: integer vectors of the row and column indices of the non-zero elements in the *lower triangle* of the Hessian.
direct	This argument is deprecated, and is included only for backwards compatibility with earlier versions.
delta	The perturbation amount for finite differencing of the gradient to compute the Hessian (the δ in Section 1.1). Defaults to `sqrt(.Machine$double.eps)`.
index1	If `TRUE` (the default), `row` and `col` use one-based indexing. If `FALSE`, zero-based indexing is used.
complex	If `TRUE`, the complex step method is used. If `FALSE` (the default), a simple finite differencing of gradients is used.
...	Additional arguments to be passed to `fn` and `gr`.

Table 3: Arguments to the sparseHessianFD initializer.

where `...` represents all other arguments that are passed to `fn` and `gr`.

The `fn`, `gr` and `hessian` methods respectively evaluate the function, gradient and Hessian at a variable vector x. The `fngr` method returns the function and gradient as a list. The `fngrhs` method includes the Hessian as well.

```
R> f <- obj$fn(x)
R> df <- obj$gr(x)
R> hess <- obj$hessian(x)
R> fngr <- obj$fngr(x)
R> fngrhs <- obj$fngrhs(x)
```

2.4. An example

Now we can estimate the Hessian for the log posterior density of the model from Section 2.1. For demonstration purposes, **sparseHessianFD** includes functions that compute the value (`binary.f`), the gradient (`binary.grad`) and the Hessian (`binary.hess`) of this model. We will treat the result from `binary.hess` as a "true" value against which we will compare the numerical estimates.

To start, we load the data, set some dimension parameters, set prior values for $\mathbf{\Sigma}^{-1}$ and $\mathbf{\Omega}^{-1}$, and simulate a vector of variables at which to evaluate the function. The `binary.f` and `binary.grad` functions take the data and priors as lists. The `data(binary)` call adds the appropriate data list to the environment, but we need to construct the prior list ourselves.

```
R> set.seed(123)
```

```
R> data("binary")
R> str(binary)

List of 3
 $ Y: int [1:50] 13 1 18 18 19 6 16 6 5 8 ...
 $ X: num [1:4, 1:50] -0.07926 -0.23018 1.55871 0.00997 0.01828 ...
 $ T: num 20

R> N <- length(binary[["Y"]])
R> k <- NROW(binary[["X"]])
R> T <- binary[["T"]]
R> nvars <- as.integer(N * k + k)
R> priors <- list(inv.Sigma = rWishart(1, k + 5, diag(k))[,,1],
+   inv.Omega = diag(k))
```

This dataset represents the simulated choices for $N = 50$ customers over $T = 20$ purchase opportunties, where the probability of purchase is influenced by $k = 4$ covariates.

The next code chunk evaluates the "true" value, gradient and Hessian. The `order.row` argument tells the function whether the variables are ordered by household (`TRUE`) or by covariate (`FALSE`). If `order.row` is `TRUE`, then the Hessian will have a banded pattern. If `order.row` is `FALSE`, then the Hessian will have a block-arrow pattern.

```
R> P <- rnorm(nvars)
R> order.row <- FALSE
R> true.f <- binary.f(P, binary, priors, order.row=order.row)
R> true.grad <- binary.grad(P, binary, priors, order.row=order.row)
R> true.hess <- binary.hess(P, binary, priors, order.row=order.row)
```

The sparsity pattern of the Hessian is specified by two integer vectors: one each for the row and column indices of the non-zero elements of the lower triangule of the Hessian. For this example, we happen to have a matrix with the same sparsity pattern of the Hessian we are trying to compute, so we can use the `Matrix.to.Coord` function to extract the appropriate index vectors. In practice, it is more likely that we would need to determine the row and column indices directly, through our knowledge of the structure of the problem. For a hierarchical model, we can create a block-arrow pattern matrix using either the `Matrix::bdiag` or `kronecker` functions to create a block diagonal matrix, and concatenate dense rows and columns to the margins.

```
R> pattern <- Matrix.to.Coord(tril(true.hess))
R> str(pattern)

List of 2
 $ rows: int [1:1310] 1 2 3 4 201 202 203 204 2 3 ...
 $ cols: int [1:1310] 1 1 1 1 1 1 1 1 2 2 ...
```

Finally, we create an instance of a *sparseHessianFD* object. Evaluations of the function and gradient will generate the same results as the true values because they are computed using the same function.

```
R> obj <- sparseHessianFD(P, fn=binary.f, gr=binary.grad,
+   rows=pattern[["rows"]], cols=pattern[["cols"]],
+   data=binary, priors=priors, order.row=order.row)
R> f <- obj$fn(P)
R> identical(f, true.f)

[1] TRUE

R> gr <- obj$gr(P)
R> identical(gr, true.grad)

[1] TRUE
```

The evaluation of the Hessian is a finite differenced approximation, so it is very close to, but not identical to, the true value, in terms of mean relative difference.

```
R> hs <- obj$hessian(P)
R> mean(abs(hs-true.hess))/mean(abs(hs))

[1] 1.12e-08
```

Note that variables `f` and `true.f` were computed using the same exact calculation, as were variables `gr` and `true.grad`.In contrast, `true.hess` was computed using an exact calculation, while `hs` is the sparse finite difference approximation.

2.5. Complex step method

The **sparseHessianFD** package also allows for estimation of the Hessian using the complex step method (Squire and Trapp 1998) as an alternative to finite differencing. In this method, the definition of $\mathbf{Y}_c$ in Equation 10 is replaced with

$$\mathbf{Y}_c = \mathrm{Im}(\nabla f(x + iG_c)) \tag{18}$$

where $i = \sqrt{-1}$ and Im returns the imaginary part of a complex number. The advantage of this method is that there is no longer a subtraction operation to generate truncation error. Thus, δ can be made arbitrarily small, and the resulting estimate of the Hessian is more accurate.

To use the complex step method, include the argument `complex=TRUE` in the call to the initializer of the *sparseHessianFD* object. The functions passed as `fn` and `gr` must both accept a complex argument, and return a complex result, even though we are differentiating a real-valued function. Although **base** R supports complex arguments for basic arithmetic, trigonometric, logarithmic and exponential functions, many other common functions (e.g., `gamma`, `log1p`, `expm1`, and the probability distribution functions) do not have complex implementations. Furthermore, the complex step method is valid only if the function is holomorphic (complex differentiable). The methods in *sparseHessianFD* do *not* check that this is the case for the function at hand. We convey the following warning from the documentation of the **numDeriv** package (Gilbert and Varadhan 2012), which also implements the complex step

method: "Avoid this method if you do not know that your function is suitable. Your mistake may not be caught and the results will be spurious."

The log posterior density in Equation 15 is holomorphic, so we can estimate its Hessian using the complex step method, and compute the mean relative difference from the true Hessian.

```
R> obj2 <- sparseHessianFD(P, fn=binary.f, gr=binary.grad,
+   rows=pattern[["rows"]], cols=pattern[["cols"]],
+   complex=TRUE,
+   data=binary, priors=priors, order.row=order.row)
R> hs2 <- obj2$hessian(P)
R> mean(abs(hs2-true.hess))/mean(abs(hs2))

[1] 6.75e-18
```

In short, the complex step method can be much more accurate than finite differencing, but it comes with theoretical and implementation restrictions that may limit its universality.

3. Algorithms

In this section, we explain how **sparseHessianFD** works. The algorithms are adapted from Coleman, Garbow, and Moré (1985b), who provided Fortran implementations as Coleman, Garbow, and Moré (1985a). Earlier versions of **sparseHessianFD** included licensed copies of the Coleman *et al.* (1985a) code, on which the current version no longer depends. Although newer partitioning algorithms have been proposed (e.g., Gebremedhin, Manne, and Pothen 2005; Gebremedhin, Tarafdar, Pothen, and Walther 2009), mainly in the context of automatic differentiation, we have chosen to implement established algorithms that are known to work well, and are likely optimal for the hierarchical models that many statisticians will encounter.

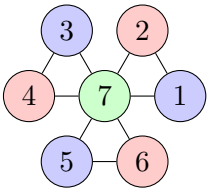
3.1. Partitioning the variables

Finding consistent, efficient partitions can be characterized as a vertex coloring problem from graph theory (Coleman and Moré 1984). In this sense, each variable is a vertex in an undirected graph, and an edge connects two vertices i and j if and only if $\mathsf{H}_{ij} f(x) \neq 0$. The sparsity pattern of the Hessian is the adjacency matrix of the graph. By "color," we mean nothing more than group assignment; if a variable is in a group, then its vertex has the color associated with that group. A "proper" coloring of a graph is one in which two vertices with a common edge do not have the same color. Coleman and Moré (1984) define a "triangular coloring" as a proper coloring with the additional condition that common neighbors of a vertex do not have the same color. A triangular coloring is a special case of an "cyclic coloring," in which any cycle in the graph uses at least three colors (Gebremedhin, Tarafdar, Manne, and Pothen 2007).

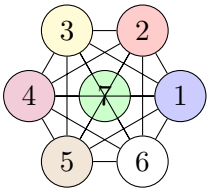
An "intersection set" contains characteristics that are common to two vertices, and an "intersection graph" connects vertices whose intersection set is not empty. In our context, the set in question is the row indices of the non-zero elements in each column of L. In the intersection graph, two vertices are connected if the corresponding columns in L have at least one non-zero element in a common row.

	1	2	3	4	5	6	7
1	1						
2	1	1					
3	0	0	1				
4	0	0	1	1			
5	0	0	0	0	1		
6	1	0	0	0	1	1	
7	1	1	1	1	1	1	1

(a) Adjacency matrix.



(b) Sparsity graph.



(c) Intersection graph.

Figure 2: Unpermuted matrix.

Powell and Toint (1979) write that a partitioning is consistent with a substitution method if and only if no columns of the of lower triangle of the Hessian that are in the same group have a non-zero element in the same row. An equivalent statement is that no two adjacent vertices in the intersection graph can have the same color. Thus, we can partition the variables by creating a proper coloring of the intersection graph of L.

This intersection graph, and the number of colors needed to color it, are not invariant to permutation of the rows and columns of H. Let π represent such a permutation, and let L_π be the lower triangle of $\pi\mathsf{H}\pi^\top$. Coleman and Moré (1984, Theorem 6.1) show that a coloring is triangular if and only if it is also a proper coloring of the intersection graph of L_π. Furthermore, Coleman and Cai (1986) prove that a partitioning is consistent with a substitution method if and only if it is an acyclic coloring of the graph of the sparsity pattern of the Hessian. Therefore, finding an optimal partitioning of the variables involves finding an optimal combination of a permutation π, and coloring algorithm for the intersection graph of L_π.

These ideas are illustrated in Figures 2 and 3. Figure 2a shows the sparsity pattern of the lower triangle of a Hessian as an adjacency matrix, and Figure 2b is the associated graph with a proper vertex coloring. Every column (and thus, every pair of columns) in Figure 2a has a non-zero element in row 7, so there are no non-empty intersection sets across the columns. All vertices are connected to each other in the intersection graph (Figure 2c), which requires seven colors for a proper coloring. Estimating a sparse Hessian with this partitioning scheme would be no more efficient than treating the Hessian as if it were dense.

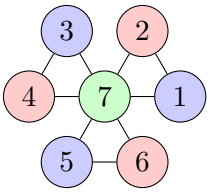
Now suppose we were to rearrange H so the last row and and column were moved to the front. In Figure 3a, all columns share at least one non-zero row with the column for variable 7, but variable groups $\{2, 4, 6\}$ and $\{1, 3, 5\}$ have empty intersection sets. The intersection graph in Figure 3c has fewer edges than Figure 2c, and can be colored with only three colors.

The practical implication of all of this is that by permuting the rows and columns of the Hessian, we may be able to reduce the number of colors needed for a cyclic coloring of the graph of the sparsity pattern. Fewer colors means fewer partitions of the variables, and that means fewer gradient evaluations to estimate the Hessian.

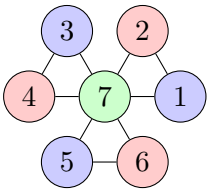
The *sparseHessianFD* class finds a permutation, and partitions the variables, when it is initialized. The problem of finding a cyclic coloring of the graph of the sparsity pattern is NP-complete (Coleman and Cai 1986), so the partitioning may not be truly optimal. Fortunately, we just need the partitioning to be reasonably good, to make the effort worth our

	7	1	2	3	4	5	6
7	1						
1	1	1					
2	1	1	1				
3	1	0	0	1			
4	1	0	0	1	1		
5	1	0	0	0	0	1	
6	1	0	0	0	0	1	1

(a) Adjacency matrix.



(b) Sparsity graph.



(c) Intersection graph.

Figure 3: Permuted matrix.

while. A plethora of vertex coloring heuristics have been proposed, and we make no claims that any of the algorithms in **sparseHessianFD** are even "best available" for all situations.

The first step is to permute the rows and columns of the Hessian. A reasonable choice is the "smallest-last" ordering that sorts the rows and columns in decreasing order of the number of elements (Coleman and Moré 1984, Theorem 6.2). To justify this permutation, suppose non-zeros within a row are randomly distributed across columns. If the row is near the top of the matrix, there is a higher probability that any non-zero element is in the upper triangle, not in the lower. By putting sparser rows near the bottom, we do not change the number of non-zeros in the lower triangle, but we should come close to minimizing the number of non-zeros in each row. Thus, we would expect the number of columns with non-zero elements in common rows to be smaller, and the intersection graph to be sparser (Gebremedhin *et al.* 2007).

The adjacency matrix of the intersection graph of the permuted matrix is the Boolean crossproduct, $\mathsf{L}_\pi^\top \mathsf{L}_\pi$. Algorithm 1 is a "greedy" vertex coloring algorithm, in which vertices are colored sequentially. The result is a cyclic coloring on the sparsity graph, which in turn is a consistent partitioning of the variables.

3.2. Computing the Hessian by substitution

The cycling coloring of the sparsity graph defines the $\mathbf{G}$ matrix from Section 1.1. We then estimate $\mathbf{Y}$ using Equation 10. Let C_m be the assigned color to variable m. The substitution method is defined in Coleman and Moré (1984, Equation 6.1).

$$\mathsf{H}_{ij} f(x) = \mathbf{Y}_{i,C_j}/\delta - \sum_{l>i, l\in C_j} \mathsf{H}_{li} f(x) \tag{19}$$

We implement the substitution method using Algorithm 2. This algorithm completes the bottom row of the lower triangle, copies values to the corresponding column in the upper triangle, and advances upwards.

3.3. Software libraries

The coloring and substitution algorithms use the **Eigen** numerical library (Guennebaud, Jacob *et al.* 2010), and the **Rcpp** (Eddelbuettel and François 2011) and **RcppEigen** (Bates and

Algorithm 1 Consistent partitioning of variables for a triangular substitution method.

Require: $P[i], i = 1, \ldots, M$: sets of column indices of non-zero elements in row i.
Require: $F[i], i = 1, \ldots, M$: sets of "forbidden" colors for vertex i (initially empty)
Require: U: set of used colors (initially empty)
Require: $C[i], i = 1, \ldots, M$: vector to store output of assigned colors (initially all zero).

```
k ← 0 {Largest color index used}
Insert 0 in U
for i = 1 to M do
  if F[i] is empty (no forbidden colors) then
    C[i] ← 0
  else
    V ← U − F[i]{Used colors that are not forbidden}
    if V is empty then
      k ← k + 1
      Insert k into U
      C[i] ← k
    else
      C[i] ← min(V) {Assign smallest existing non-forbidden color to i}
    end if
  end if
end for
for j in P[i] do
  Insert C[i] into F[j] {Make i's color forbidden to all of its uncolored neighbors}
end for
return C
```

Algorithm 2 Triangular substitution method.

Require: $P[i], i = 1, \ldots, M$: sets of column indices of non-zero elements in row i.
Require: $C[i], i = 1, \ldots, M$: vector of assigned colors
Require: $\mathbf{H}$, an $M \times M$ Hessian (initialized to zero)
Require: $\mathbf{B}$, a $\max(C) \times M$ matrix (initialized to zero)
Require: $\mathbf{Y}$, a matrix of finite differences
Require: δ, the small constant used to estimate $\mathbf{Y}$

```
for i = M to 1 do
  for All j in P_i do
    z ← Y[i, C[j]]/δ − B[C[j], i]
    B[C[i], j] ← B[C[i], j] + z
    H[i, j] ← z
    H[j, i] ← H[i, j]
  end for
end for
```

			Hessian time				Hessian/Gradient ratio			
			numDeriv		**sparseHessianFD**		**numDeriv**		**sparseHessianFD**	
N	k	M	mean	sd	mean	sd	mean	sd	mean	sd
9	2	20	363	6.4	2.8	2.2	871	63	6.5	4.2
12	2	26	629	8.9	2.9	2.1	1469	124	6.5	3.3
9	3	30	819	27.7	3.6	2.2	1974	161	8.4	3.7
15	2	32	973	25.8	3.0	2.2	2191	209	6.5	3.4
12	3	39	1407	39.3	3.5	2.0	3291	285	8.0	3.5
9	4	40	1445	28.8	4.3	2.1	3453	292	9.9	3.1
15	3	48	2138	92.7	3.5	2.0	4861	341	7.8	3.4
12	4	52	2400	98.6	4.3	2.1	5679	503	10.1	3.7
15	4	64	3643	209.0	4.2	2.2	8945	828	10.0	2.8

Table 4: Computation times (milliseconds) for computing Hessians using the **numDeriv** and **sparseHessianFD** packages, across 200 replications. Rows are ordered by the number of variables.

Eddelbuettel 2013) R packages. The **testthat** (Wickham 2011), **scales** (Wickham 2016) and **knitr** (Xie 2016) packages were used for testing, and to prepare this article.

4. Speed and scalability

As far as we know, **numDeriv** (Gilbert and Varadhan 2012) is the only other R package that computes numerical approximations to derivatives. It differs from **sparseHessianFD** in some important ways.

1. It treats all Hessians as dense;
2. It computes each element of the Hessian using a second-order finite differencing approximation that does not require the user to supply the gradient; and
3. It implements iterative algorithms to improve accuracy, at the expense of speed.

Nevertheless, **numDeriv** is an easy-to-use tool for numerical differentiation, so it is worthwhile to compare its performance to that of **sparseHessianFD**. To prepare Table 4, we estimated Hessians of the log posterior density in Equation 15 with different numbers of heterogeneous units (N) and within-unit parameters (k). As in the previous section, the total number of variables is $M = (N + 1)k$. Table 4 shows the mean and standard deviations (across 20 replications) for the time (in milliseconds) to compute a Hessian using each package. The difference in run times is dramatic, especially because the computation time for **numDeriv** grows quadratically in the number of variables. The setup time for **sparseHessianFD** was about 7 milliseconds for all cases. Times were collected using the **microbenchmark** package (Mersmann 2014).

Because **numDeriv** does not scale, we cannot use it to benchmark **sparseHessianFD** for larger sparse Hessians. To help us understand just how scalable **sparseHessianFD** is, we ran another set of simulations, for the same hierarchical model, for different values of N and k. We

Measure	Description
Function	estimating the objective function
Gradient	estimating the gradient
Hessian	computing the Hessian (not including initialization or partitioning time)
Partitioning	finding a consistent partitioning of the variables (the vertex coloring problem)
Initialization	total setup time (including the partitioning time)
Hessian/Gradient	ratio of the Hessian time to the gradient time

Table 5: Summary of timing tests (see Figure 4).

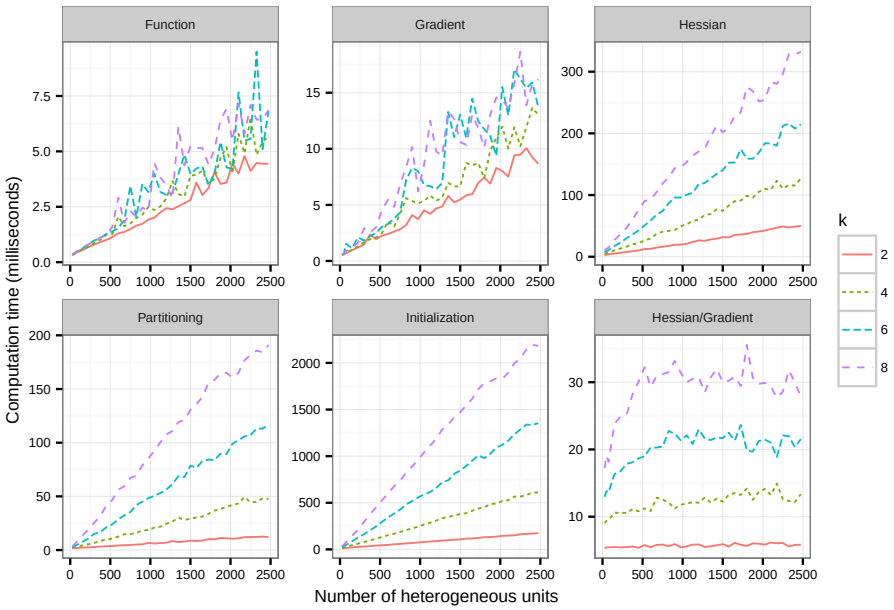


Figure 4: Run times for sparse Hessian computation.

then recorded the run times for different steps in the sparse Hessian estimation, across 200 replications. The steps are summarized in Table 5.

In the plots in Figure 4, the number of heterogeneous units (N) is on the x-axis, and mean run time, in milliseconds, is on the y-axis. Each panel shows the relationship between N and run time for a different measure of time, and each curve in a panel represents a different number of within-unit parameters (k).

Computation times for the function and gradient, as well as the setup and partitioning times for the *sparseHessianFD* object, grow linearly with the number of heterogenous units. The time for the Hessian grows linearly as well, and that might be partially surprising. We saw in Section 3.1 that adding additional heterogeneous units in a hierarchical model does not increase the number of required gradient evaluations. So we might think that the time to compute a Hessian should not increase with N *at all*. The reason it does is that each gradient evaluation takes longer. The plot of the ratio of Hessian-to-gradient run times is relatively flat

as the size of the dataset increases. We can conclude that the **sparseHessianFD** algorithms are quite efficient and scalable for hierarchical models.

References

Bates D, Eddelbuettel D (2013). "Fast and Elegant Numerical Linear Algebra Using the RcppEigen Package." *Journal of Statistical Software*, **52**(5), 1–24. URL https://www.jstatsoft.org/v52/i05.

Bates D, Maechler M (2015). ***Matrix**: Sparse and Dense Matrix Classes and Methods*. R package version 12-4, URL http://CRAN.R-project.org/package=Matrix.

Braun M (2014). "trustOptim: An R Package for Trust Region Optimization with Sparse Hessians." *Journal of Statistical Software*, **60**(4), 1–16. URL http://www.jstatsoft.org/v60/i04/.

Braun M (2015). *sparseMVN: An R Package for MVN Sampling with Sparse Covariance and Precision Matrices*. R package version 0.2.0, URL http://cran.r-project.org/package=sparseMVN.

Coleman TF, Cai JY (1986). "The Cyclic Coloring Problem and Estimation of Sparse Hessian Matrices." *SIAM Journal on Algebraic Discrete Methods*, **7**(2), 221–235. doi:10.1137/0607026.

Coleman TF, Garbow BS, Moré JJ (1985a). "Algorithm 636: Fortran Subroutines for Estimating Sparse Hessian Matrices." *ACM Transactions on Mathematical Software*, **11**(4), 378. doi:10.1145/6187.6193.

Coleman TF, Garbow BS, Moré JJ (1985b). "Software for Estimating Sparse Hessian Matrices." *ACM Transactions on Mathematical Software*, **11**(4), 363–377. doi:10.1145/6187.6190.

Coleman TF, Moré JJ (1984). "Estimation of Sparse Hessian Matrices and Graph Coloring Problems." *Mathematical Programming*, **28**(3), 243–270. doi:10.1007/BF02612334.

Curtis AR, Powell MJD, Reid JK (1974). "On the Estimation of Sparse Jacobian Matrices." *Journal of the Institute of Mathematics and its Applications*, **13**, 117–119. doi:10.1093/imamat/13.1.117.

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

Gebremedhin AH, Manne F, Pothen A (2005). "What Color is your Jacobian? Graph Coloring for Computing Derivatives." *SIAM Review*, **47**(4), 629–705. doi:10.1137/S0036144504444711.

Gebremedhin AH, Tarafdar A, Manne F, Pothen A (2007). "New Acyclic and Star Coloring Algorithms with Application to Computing Hessians." *SIAM Journal of Scientific Computation*, **29**(3), 1042–1072. doi:10.1137/050639879.

Gebremedhin AH, Tarafdar A, Pothen A, Walther A (2009). "Efficient Computation of Sparse Hessians Using Coloring and Automatic Differentiation." *INFORMS Journal on Computing*, **21**(2), 209–223. doi:10.1287/ijoc.1080.0286.

Gilbert P, Varadhan R (2012). ***numDeriv**: Accurate Numerical Derivatives*. R package version 2014.2-1, URL http://cran.r-project.org/package=numDeriv.

Guennebaud G, Jacob B, *et al.* (2010). ***Eigen***. Version 3, URL http://eigen.tuxfamily.org.

Magnus JR, Neudecker H (2007). *Matrix Differential Calculus with Applications in Statistics and Econometrics*. URL http://www.janmagnus.nl/misc/mdc2007-3rdedition.

Mersmann O (2014). ***microbenchmark**: Accurate Timing Functions*. R package version 1.4-2, URL http://cran.R-project.org/package=microbenchmark.

Nocedal J, Wright SJ (2006). *Numerical Optimization*. 2nd edition. Springer-Verlag.

Powell MJD, Toint PL (1979). "On the Estimation of Sparse Hessian Matrices." *SIAM Journal on Numerical Analysis*, **16**(6), 1060–1074. doi:10.1137/0716078.

Squire W, Trapp G (1998). "Using Complex Variables to Estimate Derivatives of Real Functions." **40**(1), 110–112. doi:10.1137/S003614459631241X.

Wächter A, Biegler LT (2006). "On the Implementation of a Primaldual Interior Point Filter Line Search Algorithm for Large-scale Nonlinear Programming." *Mathematical Programming*, **106**(1), 25–57.

Wickham H (2011). "**testthat**: Get Started with Testing." *The R Journal*, **3**, 5–10. URL http://journal.r-project.org/archive/2011-1/RJournal_2011-1_Wickham.pdf.

Wickham H (2016). ***scales**: Scale Functions for Visualization*. R package version 0.4.0, URL https://CRAN.R-project.org/package=scales.

Xie Y (2016). ***knitr**: A General-Purpose Package for Dynamic Report Generation in R*. R package version 1.12.3, URL https://CRAN.R-project.org/package=knitr.

Affiliation:

Michael Braun
Edwin L. Cox School of Business
Southern Methodist University
6212 Bishop Blvd.
Dallas, TX 75275
E-mail: braunm@smu.edu
URL: http://www.smu.edu/Cox/Departments/FacultyDirectory/BraunMichael

Journal of Statistical Software http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/

Volume VV, Issue II *Submitted:* yyyy-mm-dd
MMMMMM YYYY *Accepted:* yyyy-mm-dd