trustOptim: An R Package for Trust Region Optimization with Sparse Hessians

Michael Braun
SMU Cox School of Business
Southern Methodist University

November 5, 2013

Abstract

Trust region algorithms for nonlinear optimization are commonly believed to be more stable than their line-search counterparts, especially for functions that are non-concave, ill-conditioned, and/or exhibit regions that are close to flat. Additionally, most freely-available optimization routines do not exploit the sparsity of the Hessian when such sparsity exists, as in log posterior densities of Bayesian hierarchical models. The **trustOptim** package for the R programming language addresses both of these issues. It is intended to be both robust, scalable and efficient for a large class of nonlinear optimization problems that are often encountered in statistics, such as finding posterior modes. When used in conjunction with the **sparseHessianFD** package, the user does not need to supply the exact sparse Hessian, as long as the sparsity structure is known in advance. For models with a large number of parameters, but for which most of the cross-partial derivatives are zero (i.e., the Hessian is sparse), **trustOptim** offers dramatic performance improvements over existing options, in terms of computational time and memory footprint.

1 Introduction

The need to optimize continuous nonlinear functions occurs frequently in statistics, most notably in maximum likelihood and *maximum a posteriori* (MAP) estimation. Users of R (R Development Core Team 2012) have a choice of dozens of optimization algorithms. The most readily available algorithms are those that are accessed from the `optim` function in the base R distribution. These algorithms include conjugate gradient (`CG`), quasi-Newton using `BFGS` updates, limited-memory `BFGS` (`L-BFGS`), derivative-free heuristic search (Nelder-Mead) and simulated annealing (`SANN`). In addition, users can install contributed packages that implement algorithms that are either not available through `optim`, or improve those that are. Many are described in the CRAN Task View for Optimization and Mathematical Programming (Theussel 2013). For example, the **nloptwrap** (Borchers 2013) and **optimx** packages (Nash and Varadhan 2011) each implement many different algorithms, each with its own characteristics and limitations. Other packages, like **Rcgmin** (Nash 2013) and **trust** (Geyer 2009), are purpose-built for a single algorithm (conjugate gradient and trust region, respectively). Any particular algorithm may be more appropriate for some problems than for others, and having such a large number of alternatives lets the informed R user choose the best tool for the task at hand.

One limitation of most of these algorithms is that they can be difficult to use when there is a large number of decision variables. Search methods like Nelder-Mead are inefficient with a massive number of parameters because the search space is large, and they do not exploit information about slope and curvature to speed up the time to convergence. `CG` and `BFGS` do use gradient information, with `BFGS` tracing out the curvature of the function by using successive gradients to approximate the inverse Hessian. However, because `BFGS` stores the entire dense inverse Hessian, its use is resource-intensive when the number of parameters is large. For example, the Hessian for a 50,000 parameter model requires 20GB of RAM to store it as a standard, dense base R matrix. Conjugate gradient methods, and the limited-memory methods in **nloptwrap** (e.g., `L-BFGS`, truncated Newton and variable metric) do not store the full Hessian (or its inverse), so they can be more suited for large-scale problems. However, like `BFGS`, they are not certain to approximate the curvature of the objective function accurately at any particular iteration, especially if the function is not convex.

Conjugate gradient, `BFGS`, `L-BFGS`, truncated Newton and variable metric methods fall into the "line search" class of nonlinear optimization algorithms. In short, line search methods choose a direction along which to move from x_t to x_{t+1}, and then find the distance along that direction that yields the greatest improvement in the objective function. A simple example of a line search method is "steepest descent," which follows the direction of the gradient at x_t, and searches for the "best" point along that line. Steepest descent in known to be inefficient, which is why one might use these other methods to find a better direction in which to advance (Nocedal and Wright 2006). However, if the objective function is ill-conditioned, non-convex, or has long ridges or plateaus, the optimizer may try to search far away from x_t, only to select an x_{t+1} that is closer to x_t, but offers only small improvement in the objective function. At worst, the line search step will try to evaluate the objective function so far away from x_t that the objective function is not finite, and the algorithm will fail.

In contrast, the **trust** package (Geyer 2009), as well as the package that is presented in this paper, take a "trust region" approach. In our experience, trust region algorithms tend to be more robust and stable than line search algorithms, and may succeed for certain kinds of large-scale problems that line search methods cannot solve. Like many other nonlinear optimizers, they are iterative, and use gradient and Hessian estimates at each step to decide where to move next. Trust region methods work by choosing a maximum distance for the move from x_t to x_{t+1}, defining a "trust region" around x_t that has a radius of that maximum distance, and then letting a candidate for x_{t+1} be the minimum, within the trust region, of a quadratic approximation of the objective function. We call this constrained quadratic program the "trust region subproblem" or TRS. Because we do not consider points outside of the trust region, the algorithm never runs too far, too fast, from the current iterate. If we try to move to a point in the trust region that is worse than, or insufficiently better than, the current point, we adaptively shrink the trust region. This step excludes other points that are too far away from x_t to be reasonable candidates for x_{t+1}. We then solve the new TRS with the smaller trust region. If we accept a point close to the border of the trust region, and that point gives as a large enough improvement in the objective function, we can expand the trust region for the next iteration. By adaptively adjusting the size of the trust region, we try to prevent the algorithm from jumping over the local optimum, while allowing for steps that are large enough for the algorithm to converge quickly.

Like line search methods, trust region methods are guaranteed to converge to a point where the norm of the gradient is nearly zero and the Hessian is positive definite, if such a point exists. The primary advantage of trust region methods is stability. If a point along a line search path causes the objective function to be undefined or indeterminate, most implementations of line search methods will fail. It is not immediately clear how the search should proceed in that event; user intervention is usually required. In contrast, the search for x_{t+1} in a trust region algorithm is always a solution to the TRS, which should always be finite, even when the Hessian is indefinite. If the objective function, at the solution to the TRS, is not finite (or just not much better than at x_t), we reject that proposal, shrink the trust region, and try again. Furthermore, a line search requires repeated estimation of the objective function, while trust region methods evaluate the objective function only after solving the TRS. Thus, trust region methods can run a lot faster when the objective function is expensive to compute. Although there is no guarantee that trust region algorithms will always converge faster than other alternatives, they may work better for difficult optimization problems that other algorithms cannot solve.

To use the **trust** package, the user must provide a function that returns the Hessian of the objective function as a standard, dense R matrix. Because **trust** computes the eigenvalues of the Hessian to solve the TRS, it tends to work well on functions with no more than a few hundred variables. The computation time and memory utilization is too high to make it practical for larger problems. In this paper, we present the **trustOptim** package as an alternative trust region optimizer for R. Unlike **trust**, **trustOptim** is optimized for problems for which the Hessian is sparse. Sparse Hessians occur when a large number of the cross-partial derivatives of the objective function are zero. For example, suppose we want to find the mode of a log posterior density for a Bayesian hierarchical model. If we assume that individual-level parameter vectors β_i and β_j are conditionally independent, the cross-partial derivatives between all elements of β_i and β_j are zero. If the model includes a large number of heterogeneous units, and a relatively small number of population-level parameters, the proportion of non-zero entries in the Hessian will be small. Since we know up front which elements of the Hessian are non-zero, we need to compute, store, and operate on only those non-zero elements. By storing sparse Hessians in a compressed format, and using a library of numerical algorithms that are efficient for sparse matrices, we can run the optimization algorithms faster, with a smaller memory footprint, than algorithms that operate on dense Hessians.[1] In this paper, we will show that **trustOptim** can perform better than Hessian-free algorithms as well.

In the next section, we discuss the specifics of the trust region implementation in **trustOptim**. We then introduce the `trust.optim` function, describe how to use it, and demonstrate its performance in a hierarchical binary regression example. As part of this demonstration, we compare its performance to that of some other gradient-based nonlinear optimizers that are available for R.[2]

2 Algorithmic details

Consider $f(x)$, an objective function over a p-dimensional vector that we want to minimize. Let g be the gradient, and let B be the Hessian. The goal is to find a local minimum of $f(x)$, with no constraints on x, within some window of numerical precision (i.e., where $||g||/\sqrt{n} < \epsilon$ for small $\epsilon > 0$). We will assume that B is positive definite at the local optimum, but not necessarily at other values of x. Iterations are indexed by t.

[1]Both the **Matrix** package for R (Bates and Maechler 2012) and the Eigen numerical library (Guennebaud *et al.* 2012) for C++ provide classes and functions to operate on sparse matrices. We use both in **trustOptim**, although there are others that may work as well.

[2]While we recognize that many users may be loathe to provide gradients, even for differentiable objective functions, we are excluding derivative-free optimizers from the analysis.

2.1 Trust region methods for nonlinear optimization

The details of trust region methods are described in both Nocedal and Wright (2006) and Conn *et al.* (2000), and the following exposition borrows heavily from both sources. At each iteration of a trust region algorithm, we construct a quadratic approximation to the objective function at x_t, and minimize that approximation, subject to a constraint that the solution falls within a trust region with radius d. More formally, each iteration of the trust region algorithm involves solving the "trust region subproblem," or TRS.

$$\min_{s \in R^p} f^*(s) = f(x_t) + g_t' s + \frac{1}{2} s' B_t s \qquad \text{s.t. } \|s\|_M \leq d_t \tag{TRS}$$
$$s_t = \arg\min_{s \in R^p} f^*(s)$$

The norm $\|\cdot\|_M$ is a Mahalanobis norm with respect to some positive definite matrix M.

Let s_t be the solution to the TRS for iteration t, and consider the ratio

$$\rho_t = \frac{f(x_t) - f(x_t + s_t)}{f^*(x_t) - f^*(x_t + s_t)} \tag{1}$$

This ratio is the improvement in the objective function that we would get from a move from x_t to x_{t+1}, where $x_{t+1} = x_t + s_t$, relative to the improvement that is predicted by the quadratic approximation. Let η_1 be the minimum value of ρ_t for which we deem it "worthwhile" to move from x_t to x_{t+1}, and let η_2 be the maximum ρ_t that would trigger a shrinkage in the trust region. If $\rho_t < \eta_2$, or if $f(x_t + s_t)$ is not finite, we shrink the trust region by reducing d_t by some predetermined factor, and compute a new s_t by solving the TRS again. If $\rho_t > \eta_1$, we move to $x_{t+1} = x_t + s_t$. Also, if we do accept the move, and s_t is on the border of the trust region, we expand the trust region by increasing d, again by some predetermined factor. The idea is to not move to a new x if $f(x_{t+1})$ would be worse than $f(x_t)$. By expanding the trust region, we can propose larger jumps, and potentially reach the optimum more quickly. We want to propose only moves that are among those that we "trust" to give reasonable values of $f(x)$. If it turns out that a move leads to a large improvement in the objective function, and that the proposed move was constrained by the radius of the trust region, we want to expand the trust region so we can take larger steps. If the proposed move is bad, we should then reduce the size of the region we trust, and try to find another step that is closer to the current iterate. Of course, there is no reason that the trust region needs to change at after at a particular iteration, especially if the solution to the TRS is at an internal point.

There are a number of different ways to solve the TRS; Conn *et al.* (2000) is authoritative and encyclopedic in this area. The **trustOptim** package uses the method described in Steihaug (1983). The Steihaug algorithm is, essentially, a conjugate gradient solver for a constrained quadratic program. If B_t is positive definite, the Steihaug solution to the TRS will be exact, up to some level of numerical precision. However, if B_t is indefinite, the algorithm could try to move in a direction of negative curvature. If the algorithm happens to stumble on such a direction, it goes back to the last direction that it moved, runs in that direction to the border of the trust region, and returns that point of intersection with the trust region border as the "solution" to the TRS. This solution is not necessarily the true minimizer of the TRS, but it still might provide sufficient improvement in the objective function such that $\rho_t > \eta_1$. If not, we shrink the trust region and try again. As an alternative to the Steihaug algorithm for solving the TRS, Conn *et al.* (2000) suggest using the Lanczos algorithm. The Lanczos approach may be more likely to find a better solution to the TRS when B_k is indefinite, but at some additional computational cost. We include only the Steihaug algorithm for now, because it still seems to work well, especially for sparse problems.

As with other conjugate gradient methods, one way to speed up the Steihaug algorithm is to rescale the trust region subproblem with a preconditioner M. Note that the constraint in TRS is expressed as an M-norm, rather than a Euclidean norm. The positive definite matrix M should be close enough to the Hessian that $M^{-1}B_t \approx I$, but still cheap enough to compute that the cost of using the preconditioner does not exceed the benefits. Of course, the ideal preconditioner would be B_t itself, but B_t is not necessarily positive definite, and we may not be able to estimate it fast enough for preconditioning to be worthwhile. In this case, one could use a modified Cholesky decomposition, as described in Nocedal and Wright (2006),

2.2 Computing Hessians

The **trustOptim** package provides three trust region "methods" that differ only in how the Hessian matrix B is computed and stored. The `Sparse` method is the main method in **trustOptim**, and is optimized for objective functions with sparse Hessians. This method requires the user to supply a function that returns the Hessian as a `dgCMatrix` matrix, as defined in the **Matrix** package (Bates and Maechler 2012). It is preferred if an analytical expression for the Hessian is readily available, or if the user can compute the Hessian using algorithmic, or automatic, differentiation (AD) software, such as the **CppAD** library for C++, (Bell 2012), or **AD Model Builder** (Fournier *et al.* 2012) with the **R2admb** package (Bolker and Skaug 2012). However, in conjunction with the **sparseHessianFD** package (Braun 2012), **trustOptim** can still be used even if the Hessian is not available, as long as the sparsity structure is known in advance. The routines in **sparseHessianFD** take as input the row and column indices of the non-zero elements of the lower triangle of the Hessian, and return functions that compute the Hessian through finite differencing of the gradient. These routines exploit the sparsity structure using the algorithms published in Coleman *et al.* (1985) and can often be faster than computing the Hessian directly.

trustOptim also includes two quasi-Newton methods, `BFGS` and `SR1` for estimating inverse Hessians when the exact Hessian is not available. These methods approximate the Hessian by tracing the curvature of the objective function through repeated estimates of the gradient, and differ only in the formula they use to for the Hessian updates. These Hessians are stored as dense matrices, so they are not appropriate for large problems. In fact, many of the algorithms in **nloptwrap** will perform better. We include `BFGS` and `SR1` in the package for convenience and completeness.

3 Using the package

To run the algorithms in **trustOptim**, the user will call the `trust.optim` function. Its signature is:

```
trust.optim(x, fn, gr, hs=NULL, method=c("SR1","BFGS","Sparse"),
              control=list(), ...)
```

The user must supply a function `fn` that returns $f(x)$, the value of the objective function to be minimized, and a function `gr` that returns the gradient. For the `Sparse` method, the function `hs` returns the Hessian as a sparse matrix of class `dgCMatrix`, which is defined in the **Matrix** package. The functions `fn, gr,` and `hs` all take a parameter vector as the first argument. Additional named arguments can be passed to `fn`, `gr` or `hs` through the ... argument. If only the sparsity structure of the Hessian is known, one can use the **sparseHessianFD** package to construct a function that can be used as the argument to `hs`. The quasi-Newton methods `SR1` and `BFGS` do not require the user to provide any Hessian information. For those methods, `hs` should be, and will default to, `NULL`.

Although it is true that the `CG` and `BFGS` methods in `optim` do not require a user-supplied gradient, those methods will otherwise estimate the gradient using finite differencing. In general, we never recommend finite-differenced gradients for any problem other than those with a very small number of variables, even when using `optim`. Finite differencing takes a long time to run when there is a large number of variables, 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 renders those Hessians next to worthless.

3.1 Control parameters

The `control` argument takes a list of options, all of which are described in the package manual. Most of these arguments are related to the internal workings of the trust region algorithm, such as how close a step needs to be to the border of the trust region before the region expands. However, there are a few arguments that deserve some special attention.

3.1.1 Stopping criteria

The `trust.optim` function will stop when $\|g\|/\sqrt{p} < \epsilon$ for a sufficiently small ϵ, where g is the gradient, p is the number of parameters, and the norm is Euclidean. The parameter ϵ is the `prec` parameter in the control list. It defaults to $\sqrt{\texttt{.Machine\$double.eps}}$, which is the square root of the computer's floating point precision. However, sometimes the algorithm cannot get the gradient to be that flat. When that occurs, the trust region will shrink, until its radius is less than the value of the `cg.tol` parameter. The algorithm will then stop with the message "Radius of trust region is less than stop.trust.radius." This event is not necessarily a problem if the norm of the gradient is still small enough that the gradient is flat for all practical purposes. For example, suppose we set `prec` to be 10^{-7} and that, for numerical reasons, the norm of the gradient simply cannot get below 10^{-6}. If the norm of the gradient were the only stopping criterion, the algorithm would continue to run, even though it has probably hit the local optimum. With the alternative stopping criterion, the algorithm will also stop when it is clear that the algorithm can no longer take a step that leads to an improvement in the objective function.

There is, of course, a third stopping criterion. The `maxit` is the maximum number of iterations the algorithm should run before stopping. However, keep in mind that if the algorithm stops at `maxit`, it is almost certainly not at a local optimum.

Note that many other nonlinear optimizers, including `optim`, do not use the norm of the gradient as a stopping criterion. Instead, apart from the `SANN` method, `optim` stops when the absolute or relative changes in the objective function are less that `abstol` or `reltol`, respectively. This often causes `optim` to stop prematurely, when the estimates of the gradient and/or Hessian are not precise, or if there are some regions of the domain where the objective function is nearly flat. In theory, this should never happen, but in reality, it happens *all the* *time*. For an unconstrained optimization problem, there is no reason why the norm of the gradient should not be zero (within numerical precision) before the algorithm stops.

The `cg.tol` parameter specifies the desired accuracy for each solution of the trust region subproblem. If it is set too high, there is a loss of accuracy at each step, but if set too low, the algorithm may take too long at each trust region iteration. In general, each TRS solution does not need to be particularly precise. Similarly, the `trust.iter` parameter controls the maximum number of conjugate gradient iterations for each attempted solution of the trust region subproblem. To minimize the loss of accuracy that occurs when the conjugate gradient step stops prematurely, this number should be set high.

3.1.2 Preconditioners

Currently, the package offers two preconditioners: an identity preconditioner (no preconditioning), and an inexact modified Cholesky preconditioner (Nocedal and Wright 2006, Algorithm 7.3). The identity and diagonal preconditioners are available for all of the methods. For the `Sparse` method, the modified Cholesky preconditioner will use a positive definite matrix that is close to the potentially indefinite Hessian (`trust.optim` does *not* require that the objective function be positive definite). For `BFGS`, the Cholesky preconditioner is available because `BFGS` updates are always positive definite. If the user selects a Cholesky preconditioner for `SR1`, the algorithm will use the identity preconditioner instead.

There is no general rule for selecting preconditioners. There will be a tradeoff between the number of iterations needs to solve the problem and the time it takes to compute any particular preconditioner. In some cases, the identity preconditioner may even solve the problem in fewer iterations than a modified Cholesky preconditioner.

4 Example: Binary choice

In this section, we present two related examples that demonstrate that **trustOptim** performs better than many other `R` optimizers when the problem is large and the Hessian is sparse, but does not do as well for small problems with dense Hessians. We start with an example of the first case: a hierarchical binary choice model with heterogeneous coefficients. After that, we present an example of the second case, in which the coefficients are homogeneous.

4.1 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. 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

$$p_i = \frac{\exp(x_i'\beta_i)}{1+\exp(x_i'\beta_i)},\ i = 1 \ldots N \tag{2}$$

The coefficients 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

$$\log \pi(\beta_{1:N}, \mu | Y, X, \Sigma_0, \Omega_0) = \\ \sum_{i=1}^{N} \left[y_i \log p_i + (T - y_i) \log (1-p_i) \right] - \frac{1}{2} (\beta_i - \mu)' \Sigma^{-1} (\beta_i - \mu) - \frac{1}{2} \mu' \Omega_0^{-1} \mu \tag{3}$$

Since the β_i are drawn iid from a multivariate normal, $\dfrac{\partial^2 \log \pi}{\partial \beta_i \beta_j} = 0$ 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.

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, but 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 R code for this example is contained in two files. The objective function and its gradient is defined in `demo_funcs.R`, and the script to run the example is in `demo/choice_sparse.R`.

```
 [1,] | | . . . . . . . . . . | |
 [2,] | | . . . . . . . . . . | |
 [3,] . . | | . . . . . . . . | |
 [4,] . . | | . . . . . . . . | |
 [5,] . . . . | | . . . . . . | |
 [6,] . . . . | | . . . . . . | |
 [7,] . . . . . . | | . . . . | |
 [8,] . . . . . . | | . . . . | |
 [9,] . . . . . . . . | | . . | |
[10,] . . . . . . . . | | . . | |
[11,] . . . . . . . . . . | | | |
[12,] . . . . . . . . . . | | | |
[13,] | | | | | | | | | | | | | |
[14,] | | | | | | | | | | | | | |
```

Figure 1: Sparsity pattern for hierarchical binary choice example.

As an example, we set $T = 20$, $N = 500$, and $k = 8$; there are 4,008 parameters over which we are optimizing the objective function. Once the package is installed, the user can run this example as `demo(choice_sparse)`. In what follows, we work through the demo step by step.

First, we load libraries that are necessary to run the demo, set the parameters of the simulation study, simulate the data, and set the priors and starting values. We use rough GLM estimates to center the distribution of starting values.

```
library("Matrix")
library("mvtnorm")
library("plyr")
library("trustOptim")
library("sparseHessianFD")
set.seed(123)
N <- 500
k <- 8
T <- 20
x.mean <- rep(0,k-1)
x.var <- rep(0.1,k-1)
x.cov <- diag(x.var)
x.cov[1,k-2] <- 0.8*sqrt(x.var[1]*x.var[k-2])
x.cov[k-2,1] <- x.cov[1,k-2]

mu <- rnorm(k,0,4)
```

```
Omega <- diag(k)
inv.Sigma <- rWishart(1,k+5,diag(k))[,,1]
inv.Omega <- solve(Omega)

X <- rbind(1,t(rmvnorm(N, mean=x.mean, sigma=x.cov))) ## k x N
B <- t(rmvnorm(N, mean=mu, sigma=diag(k))) ## k x N
XB <- colSums(X * B)
log.p <- XB - log1p(exp(XB))
Y <- laply(log.p, function(q) return(rbinom(1,T,exp(q))))

## get reasonable starting values
reg <- glm((Y/T)~t(X)-1,family=binomial)
start.mean <- coefficients(reg)
start.cov <- summary(reg)$cov.unscaled
start <- as.vector(t(rmvnorm(N+1,mean=start.mean,sigma=start.cov)))
```

Next, we use the **sparseHessianFD** package to set up a function that will return the sparse Hessian. The `get.hess.struct` function returns a list of the row and column indices of the non-zero elements of the lower triangle of the Hessian (this function is defined in the `demo_funcs.R` file). The function `new.sparse.hessian.obj` is defined in **sparseHessianFD**, and its return value contains functions that return the objective function, the gradient, and the Hessian.

```
hess.struct <- get.hess.struct(N, k)
obj <- new.sparse.hessian.obj(start, fn=get.f, gr=get.grad,
                              hs=hess.struct, Y=Y, X=X,
                              inv.Omega=inv.Omega,
                              inv.Sigma=inv.Sigma, T=T)
```

An additional advantage of using `new.sparse.hessian.obj` is that when we pass additional arguments to the objective function here, they are stored in `obj`, and we do not need to include them again in the call to the optimizer.

Next, we run the optimizer. Definitions of the control parameters are described in detail in the package documentation. The control parameters to which a user might want to pay the most attention are those related to convergence of the main algorithm (`stop.trust.radius`, `prec` and `maxit`), verbosity of the reporting of the status of the algoritm (`report.freq`, `report.level` and `report.freq`), and the selection of the preconditioner (0 for no preconditioner, and 1 for a modified Cholesky preconditioner).

```
td <- system.time(opt <- trust.optim(start, fn=obj$fn,
                    gr = obj$gr,
                    hs = obj$hessian,
                    method = "Sparse",
                    control = list(
                      start.trust.radius=5,
                      stop.trust.radius = 1e-7,
                      prec=1e-7,
                      report.freq=1L,
                      report.level=4L,
                      report.precision=1L,
                      maxit=500L,
                      preconditioner=1L
                      )
                    )
                  )
```

```
> source("choice_sparse.R")
Using Sparse method in trust.optim
Beginning optimization

iter    f    rm_gr                 status    rad  CG iter       CG result
 1 11306.1   653.1  Continuing - TR expand   15.0     9  Intersect TR bound
 2  4345.9  1498.9  Continuing - TR expand   45.0     9  Intersect TR bound
 3  2221.3     2.4              Continuing   45.0   168  Reached tolerance
 4  2220.9     0.0              Continuing   45.0   177  Reached tolerance
 5  2220.9     0.0              Continuing   45.0   175  Reached tolerance
 6  2220.9     0.0              Continuing   45.0   180  Reached tolerance

Iteration has terminated
 6  2220.9     0.0                 Success
```

The output of the algorithm supplies the iteration number, the value of the objective function and norm of the gradient, whether the trust region is expanding, contracting, or staying the same size, and the current radius of the trust region. It will also report the number of iterations it took for the Steihaug algorithm to solve the trust region subproblem, and the reason the Steihaug algorithm stopped. In this example, for the first two iterations, the solution to the TRS was reached after only eight conjugate gradient steps, and this solution was at the boundary of the trust region. Since the improvement in the objective function was substantial, we expand the trust region and try again. By the third iteration, the trust region is sufficiently large that the TRS solution is found in the interior through subsequent conjugate gradient steps. Once the interior solution of the TRS is found, the trust region algorithm moves to the TRS solution, recomputes the gradient and Hessian of the objective function, and repeats until the first-order conditions of the objective function are met.

This problem has 4,008 parameters, and converged in less than four seconds.

The return value of the `trust.optim` function returns all of the important values, such as the solution to the problem, the value, gradient and Hessian of the objective function, the number of iterations, the final trust radius, the number of non-zeros in the Hessian, and the method used.

Next, we compare the performance of `trust.optim` to some alternative nonlinear optimizers in R. The methods are summarized in Table 1. The three methods from the **nloptwrap** package are "limited memory," in the sense that they do not compute or store a complete, exact Hessian (or inverse of it). The conjugate gradient method in the **Rcgmin** falls into this category as well. The only method that is called from the base R package is `BFGS`, which is identified as `bfgs-optim` in the subsequent text. I exclude the others because the conjugate gradient and `L-BFGS` methods are largely superseded by those in **Rcgmin** and **nloptwrap**, and because Nelder-Mead and SANN are of a completely different class of optimizer than the one considered in this paper. As described in the introduction, **trust** (Geyer 2009) is another stable and robust implementation of a trust region optimizer, and we found that it works well for modestly-sized problems of no more than a few hundred parameters. Unlike **trustOptim**, it requires the user to provide a complete Hessian as a dense matrix, so it cannot exploit sparsity when that sparsity exists. It also uses eigenvalue decompositions to solve the TRS, as opposed to the Steihaug conjugate gradient approach. Finally, stopping criterion in for the algorithm in **trust** is based on the change in the value of the objective function, and not the norm of the gradient.

Naturally, there are many other optimization tools available for R users. These are described in the R Task View on Optimization and Mathematical Programming.

We compare the algorithms by simulating datasets from the hierarchical binary choice model, and using the optimization algorithms to find the mode of the log posterior density. There are six test conditions, determined by crossing the number of heterogeneous units ($N \in (50, 100, 500, 2500, 5000)$) and number of parameters per unit ($k \in (3, 8)$). Within each condition, we simulated five datasets, ran the optimizers, and averaged the performance statistics of interest: total clock time, the number of iterations of the algorithm, and both the Euclidean and maximum norms for the gradient at the local optimum. These results

Package	method	Type	User-supplied gradient	User-supplied Hessian	Stores dense Hessian
trustOptim	`Sparse`	trust region	Yes	Yes	No
trustOptim	`Sparse-precond`	trust region	Yes	Yes	No
nloptwrap	`L-BFGS`	line search	optional	No	No
nloptwrap	`varmetric`	line search	optional	No	No
nloptwrap	`tnewton`	line search	optional	No	No
R base	`bfgs-optim`	line search	optional	No	Yes
Rcgmin	`Rcgmin`	line search	optional	No	No
trust	`trust`	trust region	Yes	Yes	Yes

Table 1: Summary of optimization algorithms included in comparison. Methods for which the gradient is optional will estimate the gradient numerically.

are in Table 2. We used **sparseHessianFD** to compute the Hessian for both `Sparse` and `trust` (converting to a dense matrix in the case of `trust`). We called `Sparse` both with and without applying the modified Cholesky preconditioner. We ran the algorithms on a 2010-vintage Mac Pro with 12 cores running at 2.93 GHz, and 32 GB of RAM.

With respect to run time, we see that for small datasets (e.g., $N = 50$), there is no clear reason to prefer **trustOptim** over some of the other packages. However, when the datasets get large, `Sparse` is clearly the fastest. The $N = 5000$, $k = 8$ case has more than 40,000 parameters, yet the `Sparse` method converges in less than 20 seconds with the preconditioner, and 15 seconds without it. One reason that `BFGS` and `Rcgmin` appear to run as fast as they do, even for small problems, is that they are prone to stopping before the norm of the gradient is even close to zero. In fact, one may question whether these methods have found a local optimum at all.

		k=3				k=8			
N	method	secs	$\|g\|_2$	$\|g\|_\infty$	iters	secs	$\|g\|_2$	$\|g\|_\infty$	iters
50	Sparse	.1	2.5e-5	1.2e-5	5	.3	1.5e-5	6.8e-6	7
50	Sparse-precond	.1	1.9e-5	6.5e-6	5	.3	9.5e-6	4.6e-6	7
50	lbfgs	.1	3e-6	1.3e-6	57	.2	4.3e-6	1.2e-6	161
50	varmetric	.3	1.7e-4	6.2e-5	178	1.5	1.5e-4	5e-5	672
50	bfgs-optim	.1	.011	.0062	46	.4	.031	.0096	104
50	trust	.9	3.2e-8	1.8e-8	6	4.1	2.1e-8	1.4e-8	8
50	Rcgmin	.5	4.5e-5	1.9e-5	156	1.7	1.2e-4	5.5e-5	911
50	tnewton	.2	8.1e-9	2.7e-9	89	.9	3.2e-9	7.5e-10	447
100	Sparse	.2	4.5e-5	1.8e-5	5	.4	2e-5	9.2e-6	6
100	Sparse-precond	.1	2e-5	1.5e-5	5	.4	6.1e-5	3.6e-5	6
100	lbfgs	.1	6e-6	2.9e-6	62	.4	1e-5	4.1e-6	172
100	varmetric	.5	1.6e-4	7.7e-5	241	3.0	3.6e-4	1.2e-4	766
100	bfgs-optim	.2	.027	.015	46	1.3	.056	.019	101
100	trust	2.1	3.2e-8	7.8e-9	6	12.2	3.2e-8	2.3e-8	8
100	Rcgmin	.7	1.1e-4	5.6e-5	181	2.8	3.1e-4	1.3e-4	1057
100	tnewton	.2	2.7e-9	1e-9	107	1.4	1.4e-9	4.5e-10	526
500	Sparse	.3	4.1e-5	2.5e-5	4	1.5	6.8e-5	4.1e-5	5
500	Sparse-precond	.3	1.6e-5	1.3e-5	4	2.5	4e-5	2.1e-5	5
500	lbfgs	.4	1e-5	5.4e-6	70	3.7	1.4e-4	7.5e-5	261
500	varmetric	1.6	6.4e-4	2.7e-4	295	62.4	.0033	.0022	1774
500	bfgs-optim	2.1	.13	.11	39	57.5	.2	.084	103
500	trust	25.5	2.7e-7	1.6e-7	6	349.1	1.9e-7	1e-7	7
500	Rcgmin	1.1	.0012	5.9e-4	127	9.0	.003	.0012	1008
500	tnewton	.7	1.8e-9	6.3e-10	100	11.4	5.4e-9	1.5e-9	710
2500	Sparse	2.3	1.2e-4	6.9e-5	4	26.5	3.3e-5	1.7e-5	5
2500	Sparse-precond	1.5	1.5e-4	9.7e-5	4	8.1	6e-5	3.6e-5	5
2500	lbfgs	4.3	3.0e-5	1.6e-5	88	112.2	.0021	.0014	1326
2500	varmetric	33.4	.0053	.0027	588	217.6	.008	.0045	2702
2500	bfgs-optim	6.7	.38	.23	34	1345.2	.92	.4	72
2500	Rcgmin	7.7	.012	.0056	114	29.9	.037	.015	380
2500	tnewton	4.8	3.0e-9	1.2e-9	113	34.3	4.6e-9	1.3e-9	634
5000	Sparse	4.7	2.5e-4	1.4e-4	4	107.0	8.3e-5	4.8e-5	5
5000	Sparse-precond	2.4	2.0e-4	9.7e-5	4	16.7	2.9e-5	1.5e-5	5
5000	lbfgs	8.6	6.7e-4	4.8e-4	138	267.6	.0041	.0027	3063
5000	varmetric	64.2	.01	.0062	1064	471.2	.038	.022	5842
5000	bfgs-optim	196.3	.77	.66	35	3935.8	1.4	.74	83
5000	Rcgmin	11.3	.035	.014	126	44.8	.085	.029	434
5000	tnewton	6.1	1.2e-8	6.3e-9	116	55.3	4.6e-9	1.9e-9	788

Table 2: Convergence times and gradient norms for hierarchical binary choice example. See Table 1 for descriptions of the methods. Because of time and memory constraints, we did not run the `trust` method for the $N = 2500$ and $N = 5000$ cases.

4.2 Homogeneous model with dense Hessian

The **trustOptim** package is optimized for problems for which the Hessian is sparse. As an example of how other optimizers might perform better than **trustOptim**, we consider a binary regression model for which the response coefficients are common for all individuals. Suppose the prior on β is multivariate normal with a prior mean at the origin, and a prior covariance of $\Omega_0 = 100 I_k$. In this case, the log posterior density is

$$\log \pi(\beta|Y, X, \Omega_0) = \sum_{i=1}^{N} \left[y_i \log p_i + (T - y_i) \log (1 - p_i) \right] - \frac{1}{2} \beta' \Omega_0^{-1} \beta_i \tag{4}$$

where

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

For this model, the k elements of β are the only parameters. For the timing comparison, we consider conditions of $k \in (2, 25, 250)$ for $N = 1000$. The results are in Table 3. We see that as the number of parameters increases, the limited memory methods in **nloptwrap** run faster that those in **trustOptim** and **trust**. This is because, when the Hessian is dense, storing it in a sparse matrix structure actually results in greater memory consumption, because the indices are stored in addition to the data. Also, as before, we see that the `optim` implementation of `BFGS` appears to run quickly, but stops before the norm of the gradient is close enough to zero. This example should highlight the importance of selecting the best tool for the job at hand.

5 Implementation details

The **trustOptim** package was written primarily in C++, using the Eigen Numerical Library (Guennebaud *et al.* 2012). The **trustOptim** package uses the Eigen headers from the **RcppEigen** package (Bates and Eddelbuettel 2013), so the user does not need to install Eigen separately. The user will call the `trust.optim` function from R (defined in the `call-Trust.R` file), which will in turn pass the arguments to the compiled code using functions in the **Rcpp** package (Eddelbuettel and François 2011). The `trust.optim` function then gathers results and returns them to the user in R.

The `src/Rinterface.cpp` file defines the C++ functions that collect data from R, passes

k	method	time	nrm.gr	max.abs.gr	iters
2	`Sparse`	.02	6.6e-9	4.7e-9	6
2	`Sparse-precond`	.02	6.6e-9	4.8e-9	6
2	`trust`	.04	6.6e-9	4.8e-9	6
2	`L-BFGS`	.02	1.4e-7	1.4e-7	20
2	`varmetric`	.02	9.6e-6	7.4e-6	21
2	`bfgs-optim`	.01	5.4e-5	4.0e-5	10
2	`tnewton`	.02	2.5e-13	2.2e-13	24
25	`Sparse`	.75	1.2e-9	6.9e-10	9
25	`Sparse-precond`	.66	1.4e-5	6.1e-6	10
25	`trust`	.78	2.7e-12	1.1e-12	9
25	`L-BFGS`	.10	3.2e-6	1.7e-6	32
25	`varmetric`	.30	2.4e-5	1.0e-5	32
25	`bfgs-optim`	.16	.053	.023	43
25	`tnewton`	.14	9.4e-12	5.7e-12	53
250	`Sparse`	134.3	3.6e-5	8.6e-6	27
250	`Sparse-precond`	106.1	3.2e-5	5.3e-6	19
250	`trust`	128.9	1.1e-8	1.9e-9	21
250	`L-BFGS`	13.2	5.2e-5	1e-5	511
250	`varmetric`	19.5	1.1e-4	2.2e-5	772
250	`bfgs-optim`	8.6	.023	.0044	288
250	`tnewton`	29.9	.073	.016	1196

Table 3: Convergence times and gradient norms for binary choice example with homogeneous coefficients. See Table 1 for descriptions of the methods. The `Rcgmin` method is excluded because it would not converge reliably after multiple attempts.

them to the optimization routines, and return the results. There is one function for `Sparse` and another for the quasi-Newton methods `SR1` and `BFGS`. Each function constructs an optimizer object of the class that is appropriate for that method. The class `Trust_CG_Optimizer`, for the quasi-Newton methods is defined in the file `inst/include/CG-quasi.h`, and the class `Trust_CG_Sparse`, is defined in the file`inst/CG-sparse.h`. Both of these classes inherit from the `Trust_CG_Base` class, which is defined in `inst/CG-base.h`. All of the optimization is done by member functions in `Trust_CG_Base`; `Trust_CG_Optimizer` and `Trust_CG_Sparse` differ only in how they handle the Hessian and the preconditioners.

The `Rfunc` and `RfuncHess` classes are defined in the files `inst/Rfunc.h` and `inst/RfuncHess.h`, respectively. These classes contain functions that return the value of the objective function, the gradient, and the Hessian. `Rfunc` is used for the quasi-Newton methods, `RfuncHess` is used for `Sparse`. Both classes contain references to `Rcpp::Function` objects that, in turn, are references to the R functions that compute the objective function and gradient. Thus, a call to the `get_f()` function will return the result of a call to the corresponding R function. The `RfuncHess` class returns the Hessian, as an Eigen sparse matrix, in a similar way.

6 Discussion

The motivation behind **trustOptim** was the practical difficulty in finding modes of posterior densities of hierarchical models. Existing optimization tools in the both the base R distribution and other contributed packages were either too cumbersome to use when there are a large number of parameters, too imprecise when encountering ridges, plateaus or saddle points in the objective function, or too lenient in determining when the optimization algorithm should stop. The product of the effort behind addressing these problems is a package that can be more robust, efficient and precise than existing options. This is not to say that **trustOptim** will outperform other nonlinear optimizers in all cases. But at least for hierarchical models, or other models with sparse Hessians, **trustOptim** is a useful tool in the statisticians toolbox.

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 `http://www.jstatsoft.org/v52/i05/`.

Bates D, Maechler M (2012). *Matrix: Sparse and Dense Matrix Classes and Methods.* R package version 1.0-6, URL `http://CRAN.R-project.org/package=Matrix`.

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

Bolker B, Skaug H (2012). *R2admb: ADMB to R interface functions.* R package version 0.7.5.3, URL `http://CRAN.R-project.org/package=R2admb`.

Borchers HW (2013). *nloptwrap: Wrapper for Package nloptr.* R package version 0.5-1, URL `http://CRAN.R-project.org/package=nloptwrap`.

Braun M (2012). *sparseHessianFD: an R package for estimating sparse Hessians.* URL `http://CRAN.R-project.org/package=sparseHessianFD`.

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

Conn AR, Gould NIM, Toint PL (2000). *Trust-Region Methods.* SIAM-MPS, Philadelphia.

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/`.

Fournier DA, Skaug HJ, Ancheta J, Ianelli J, Magnusson A, Maunder MN, Nielsen A, Sibert J (2012). "AD Model Builder: Using Automatic Differentiation for Statistical Inference of Highly Parameterized Complex Nonlinear Models." *Optimization Methods and Software*, **27**(2), 233–249.

Geyer CJ (2009). *trust: Trust Region Optimization.* R package version 0.1-2, URL `http://www.stat.umn.edu/geyer/trust/`.

Guennebaud G, Jacob B, *et al.* (2012). "Eigen v3." http://eigen.tuxfamily.org.

Nash JC (2013). *Rcgmin: Conjugate gradient minimization of nonlinear functions with box constraints.* R package version 2013-02.20, URL `http://CRAN.R-project.org/package=Rcgmin`.

Nash JC, Varadhan R (2011). "Unifying Optimization Algorithms to Aid Software System Users: optimx for R." *Journal of Statistical Software*, **43**(9), 1–14. URL `http://www.jstatsoft.org/v43/i09/`.

Nocedal J, Wright SJ (2006). *Numerical Optimization.* Second edition edition. Springer.

R Development Core Team (2012). *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL `http://www.R-project.org/`.

Steihaug T (1983). "The Conjugate Gradient Method and Trust Regions in Large Scale Optimization." *SIAM Journal on Numerical Analysis*, **20**(3), 626–637.

Theussel S (2013). "CRAN Task View: Optimization and Mathematical Programming." URL `http://cran.r-project.org/web.views/Optimization.html`.