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

Michael Braun

MIT Sloan School of Management

Massachusetts Institute of Technology

December 27, 2012

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 commonly 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 massive 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

Nonlinear optimization of continuous functions occurs frequently in statistics, most notably in maximum likelihood and *maximum a posteriori* (MAP) estimation. Among users of R (R Development Core Team, 2012), the `optim` function in the base R package is the most

readily available tool for nonlinear optimization. The `optim` function itself is a front-end for a variety of algorithms, such as conjugate gradient (CG), quasi-Newton using BFGS updates (BFGS and L-BFGS-B), derivative-free heuristic search (Nelder-Mead) and simulated annealing (SANN). Furthermore, there are many other contributed R packages that implement additional methods, as well as algorithms available outside of R. Having such a large number of alternatives lets the practicing statistician choose the best available tool for the task at hand.

Unfortunately, these methods can be difficult to use when there is a large number of variables over which the objective function is to be optimized. 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. Methods like CG and BFGS do use gradient information, and both BFGS and L-BFGS-B approximate the Hessian using successive gradients to trace out the curvature. However, the BFGS method stores the entire Hessian, which is resource-intensive when the number of parameters is large (the Hessian for a 50,000 parameter model requires 20GB of RAM to store it as a standard, dense base R matrix). Although L-BFGS-B is a limited-memory alternative to BFGS, neither is certain to offer an accurate approximation to the Hessian at any particular iteration, especially if the objective function is not convex (BFGS updates are always positive definite). The CG method does not store Hessian information at all, so it may the most feasible of the `optim` algorithms for large problems, although it still may not converge quickly to an optimum.

The CG, BFGS and L-BFGS-B 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 methods like CG and BFGS are used 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.

The **trustOptim** package is an alternative nonlinear optimization tool that uses a trust region approach. 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

2

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 (excluding other points that are too far away from x_t to be reasonable candidates for x_{t+1}) and solve the new TRS. 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). 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 (more on that later). 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.

The **trustOptim** package has an added benefit (not general to all trust region implementations) for being 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 use the Eigen numerical library (Guennebaud *et al.*, 2012)), we can run the optimization algorithms faster, with a smaller memory footprint, than the R `optim` algorithms.

In the next section, we discuss the specifics of the trust region implementation in **trustOptim**. We then introduce the `trust.optim` function, and describe how to use it.

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. This minimum will be a point where $||g||/\sqrt{n} < \epsilon$ where ϵ is a small precision parameter. 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 (so, for example, B_t is the Hessian at iteration t).

2.1 Trust region methods for nonlinear optimization

The details of trust region methods are described in detail 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.

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

$$s_t = \arg\min_{s \in R^k} f^*(s) \tag{2}$$

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

4

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

$$\rho_t = \frac{f(x_t) - f(x_t + s_t)}{f^*(x_t) - f^*(x_t + s_t)} \tag{3}$$

This ratio is the improvement in the objective function that we would get from a move from x_t to x_{t+1}, 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 instead. 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 use a preconditioner M to rescale the TRS. Note that the constraint in the TRS is expressed as an M-norm, rather that 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 computing the preconditioner does not exceed the benefits from using it. 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 optimized for objective functions with sparse Hessians. `Sparse` requires the user to supply a function that returns the Hessian in a sparse compressed format (namely, the `dgCMatrix` class in the **Matrix** package, Bates and Maechler 2012). The **trustOptim** package also includes two quasi-Newton methods: `BFGS` and `SR1`. The two methods do not require any information about the Hessian at all, nor do they exploit any sparsity information. They both 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 update the Hessian; BFGS updates are guaranteed to be positive definite, while SR1 updates are not (Nocedal and Wright, 2006). The quasi-Newton Hessians are stored as dense matrices, so they are not appropriate for large problems. Furthermore, from our experience it is not clear that the **trustOptim** implementations of `BFGS` and `SR1` are substantially better than the `BFGS` or `L-BFGS-B` implementations in `optim`. Nevertheless, we include them for convenience and completeness.

The `Sparse` method will be preferred if an analytical expression for the Hessian is readily available, or if the user can compute the Hessian using algorithmic differentiation (AD) software (e.g., the **CppAD** library for `C++`, Bell 2012). However, in conjunction with the **sparseHessianFD** package (Braun, 2012), the **trustOptim** `Sparse` method 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 an object that contains 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 sometimes be faster than computing the Hessian directly.

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` (this class is defined in the **Matrix** package, which is now a recommended package in R and a dependency for **trustOptim**). 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. 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`. If only the sparsity structure is known, use the **sparseHessianFD** package to construct a function that can be used as the argument to `hs`.

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 (for example, how close does a step need to get 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 and 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 just can't get the gradient to be that flat. What will then happen is that the trust region will start to 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 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. Always check the gradient to be sure.

Note that many other nonlinear optimizers, including `optim`, do not use the norm of the gradient as a stopping criterion. Instead, `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 simply 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, we do not need each TRS solution 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. Set this number high if you don't want to lose accuracy by stopping the conjugate gradient step prematurely.

3.1.2 Preconditioners

Currently, the package offers two preconditioners: an identity preconditioner (no pre-conditioning), and a modified Cholesky preconditioner. The identity and diagonal pre-conditioners are available for all of the methods. For the `Sparse` method, the modified Cholesky preconditioner will use a positive definite matrix that is closest 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 you select a Cholesky preconditioner for the SR1 method, 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: 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

$$p_i = \frac{\exp(x_i'\beta_i)}{1 + \exp(x_i'\beta_i)}, \ i = 1 \ldots N \tag{4}$$

The coefficients can be thought of as sensitivities to the covariates, and they are distributed across the population of households following a multivariate normal distribution with mean μ and covariance Σ. We assume that we know Σ, but we do not know μ. Instead, we place a multivariate normal prior on μ, with mean 0 and covariance Ω_0, which is determined in advance. Thus, each β_i, and μ are $k-$dimensional vectors, and the total number of unknown variables in the model is $(N+1)k$.

The log posterior density, ignoring any normalization constants, is

$$\log \pi(\beta_{1:N}, \mu | Y, X, \Sigma_0, \Omega_0) = \sum_{i=1}^{N} p_i^{y_i} (1 - p_i)^{T - y_i} (\beta_i - \mu)' \Sigma^{-1} (\beta_i - \mu) + \mu' \Omega_0^{-1} \mu \qquad (5)$$

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

```
 [1,] | | . . . . . . . . . . | |
 [2,] | | . . . . . . . . . . | |
 [3,] . . | | . . . . . . . . | |
 [4,] . . | | . . . . . . . . | |
 [5,] . . . . | | . . . . . . | |
 [6,] . . . . | | . . . . . . | |
 [7,] . . . . . . | | . . . . | |
 [8,] . . . . . . | | . . . . | |
 [9,] . . . . . . . . | | . . | |
[10,] . . . . . . . . | | . . | |
[11,] . . . . . . . . . . | | | |
[12,] . . . . . . . . . . | | | |
[13,] | | | | | | | | | | | | | |
[14,] | | | | | | | | | | | | | |
```
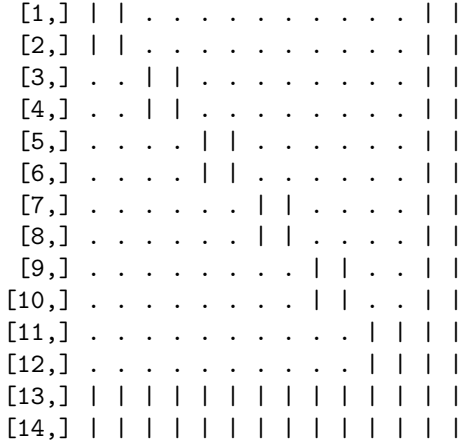
Figure 1: Sparsity pattern for hierarchical binary choice example.

There are 196 elements in this symmetric matrix, but only 169 are non-zero, and only 76 values are unique. Although the reduction in RAM from using a sparse matrix structure for the Hessian may be modest, consider what would happen if $N = 1000$ instead. In that case, there are 2,002 variables in the problem, and more than 4 million elements in the Hessian. However, only 12,004 of those elements are non-zero. If we work with only the lower triangle of the Hessian (e.g., through a Cholesky decomposition), we only need to work with only 7,003 values.

The R code for this example is contained in two files: `examples/ex1.R` and `examples/ex_funcs.R`. What follows is a discussion of the `ex1.R` file. As an example, we set $T = 20$, $N = 250$, and $k = 8$; there are 2,005 parameters over which we are optimizing the objective function.

First, we load libraries that are necessary to simulate the data and run the algorithm, and set the parameters of the simulation study.

```
library("plyr")
```

```
library("Matrix")
library("mvtnorm")
library("trustOptim")
library("sparseHessianFD")
source("ex_funcs.R")  ## fn, gr and hs functions
set.seed(123)
N <- 250
k <- 8
T <- 20
```

Next, we choose the trust.optim method we want to test, and initialize the control parameters. Definitions of these 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), the function scale factor (which must be positive if minimizing and negative if maximizing), and the selection of the preconditioner (0 for no preconditioner, and 1 for a modified Cholesky preconditioner).

```
method <- "Sparse"
control.list <- list(start.trust.radius=5,
                     stop.trust.radius = 1e-5,
                     prec=1e-7,
                     report.freq=1L,
                     report.level=4L,
                     report.precision=1L,
                     maxit=500L,
                     function.scale.factor = as.numeric(-1),
                     preconditioner=1L
                     )
```

In the next section, we simulate data, set priors and choose a starting value for the optimizer. The laply function is part of the **plyr** package.

```
x.mean <- rep(0,k)
x.cov <- diag(k)
mu <- rnorm(k,0,10)
Omega <- diag(k)
inv.Sigma <- rWishart(1,k+5,diag(k))[,,1]
inv.Omega <- solve(Omega)
```

11

```
X <- t(rmvnorm(N, mean=x.mean, sigma=x.cov))
B <- t(rmvnorm(N, mean=mu, sigma=Omega))
XB <- colSums(X * B)
log.p <- XB - log1p(exp(XB))
Y <- laply(log.p, function(q) return(rbinom(1,T,exp(q))))
nvars <- N*k + k
start <- rnorm(nvars) ## random starting values
```

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
ex_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)
```

An additional advantage of using `new.sparse.hessian.obj` is that when we pass addi-
tional 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.

We now run the algorithm, recording the time it takes to converge.

```
cat("running ",method, "\n")
t1 <- Sys.time()
opt <- trust.optim(start, fn=obj$fn,
                 gr = obj$gr,
                 hs = obj$hessian,
                 method = method,
                 control = control.list)

running  Sparse
Beginning optimization

iter      f  nrm_gr                     status   rad  CG iter       CG result
 1  15938.2  1803.7   Continuing - TR expand   15.0       8  Intersect TR bound
 2   6425.8  2735.6   Continuing - TR expand   45.0       8  Intersect TR bound
```

```
3     759.4    65.4                   Continuing    45.0    164    Reached tolerance
4     547.5    24.6                   Continuing    45.0    157    Reached tolerance
5     435.8    11.0                   Continuing    45.0    167    Reached tolerance
6     365.1     5.3                   Continuing    45.0    183    Reached tolerance
7     336.9     2.6                   Continuing    45.0    204    Reached tolerance
8     330.9     0.8                   Continuing    45.0    220    Reached tolerance
9     330.6     0.1                   Continuing    45.0    229    Reached tolerance
10    330.6     0.0                   Continuing    45.0    229    Reached tolerance
11    330.6     0.0                   Continuing    45.0    231    Reached tolerance


Iteration has terminated
11    330.6     0.0                   Success

t2 <- Sys.time()
td <- difftime(t2,t1)
print(td,units="secs")

Time difference of 1.655042 secs
```

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 or contracting (or neither) 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 four iterations, the solution to the TRS was reached after only one conjugate gradient step, 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 fifth 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.

Note that this problem has 1,005 parameters, and converged in less than half of a second.

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 (in sparse compressed format) of the objective function, the number of iterations, the final trust radius, the number of non-zeros in the Hessian, and the method used.

If we do not use **sparseHessianFD** to compute the Hessian, we can supply R functions are arguments for `fn,` `gr` and `hs`. In that case, we would pass the additional names arguments to the objective function in the `trust.optim` step.

4.1 Comparison to alternatives

Next, we compare the performance of `trust.optim` to some alternative nonlinear optimizers in R. The methods are summarized in Table **??**. The **trust** package (Geyer., 2009) is another stable and robust implementation of a trust region optimizer, and we found that it works well for modestly-sized problems (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.

Package	method	Type	Requires gradient	Requires Hessian
optim	CG	Line search	No, but preferred	No
optim	BFGS	Line search	No, but preferred	No
trust		trust region	Yes	Yes
trustOptim	Sparse	trust region	Yes	Yes

Table 1: Description of optimization algorithms

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 conditions, determined by crossing the number of heterogeneous units ($N \in (25, 250, 1000)$) and number of parameters per unit ($k \in (2, 8)$). 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 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`. For both **trustOptim** methods, we applied the modified Cholesky preconditioner.

With respect to run time, we see that for very small datasets (e.g., $N = 25$), there is no

N	method	k=2				k=8			
		secs	$\|g\|_2$	$\|g\|_\infty$	iters	secs	$\|g\|_2$	$\|g\|_\infty$	iters
25	Sparse	0.1	9.3e-06	4.3e-06	5	0.3	1.3e-06	4.9e-07	6
25	optim-BFGS	0.1	0.012	8.2e-03	31	0.2	0.015	4.8e-03	83
25	optim-CG	0.6	7.1e-06	4.2e-06	407	2.2	1.5e-05	7.6e-06	1464
25	trust	0.1	6.6e-12	4.0e-12	6	0.7	2.0e-09	1.2e-09	8
250	Sparse	0.2	1.3e-05	3.6e-06	8	1.5	1.2e-04	2.9e-05	8
250	optim-BFGS	0.3	0.03	0.023	39	8.0	0.087	0.036	118
250	optim-CG	6.8	2.5e-05	1.3e-05	1687	68.2	4.9e-05	3.0e-05	14578
250	trust	2.9	1.2e-08	3.4e-09	10	31.7	6.0e-09	9.9e-10	11
1000	Sparse	0.7	8.4e-05	1.0e-05	8	21.6	2.4e-04	2.4e-05	9
1000	optim-BFGS	3.2	0.20	0.14	38	271.3	0.73	0.35	111
1000	optim-CG	28.8	4.9e-05	2.7e-05	2182	1022.2	1.2e-04	5.7e-05	32489
1000	trust	49.0	6.0e-11	7.7e-12	10	3128.0	1.9e-08	2.0e-09	12

Table 2: Convergence times and gradient norms for hierarchical binary choice example. The method Sparse refers to the main method in **trustOptim**. The methods optim-BFGS and optim-CG refer to methods called via the optim function in base R. The method trust refers to the **trust** package.

clear reason to prefer **trustOptim** over the other packages. However, when the datasets get large, Sparse is clearly the fastest. The $N = 1000$, $k = 8$ case has more than 8,000 parameters, yet the Sparse method converges in about 100 seconds. One reason that optim.BFGS appears to run quickly, even for small problems, is that it is prone to stopping before the gradient is sufficiently flat. In fact, one may question whether BFGS has even found a local optimum. Thus, when the run times of Sparse are close to those of BFGS, the level of confidence that the result is correct seems like a reasonable tiebreaker.

5   Implementation details

The **trustOptim** package was written primarily in C++, using the Eigen Numerical Library (Guennebaud *et al.*, 2012). The **trustOptim** package links to the **RcppEigen** R package (Bates *et al.*, 2012), so the user does not need to install Eigen separately in order to compile **trustOptim**. The user will call the trust.optim function from R (defined in the callTrust.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 defines the C++ functions that collect data from R, passes them to the optimization routines, and return the results. There is one function for Sparse and another for 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 (defined in the files `inst/Rfunc.h` and `inst/RfuncHess.h`), are responsible for returning 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 immense frustration about not being able to find modes of posterior densities of hierarchical models. Existing tools in R 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 is 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, this may prove to be a useful tool in the statisticians toolbox.

References

Bates D, Francois R, Eddelbuettel D (2012). *RcppEigen: Rcpp integration for the Eigen templated linear algebra library.* R package version 0.3.1, URL `http://CRAN.R-project.org/package=RcppEigen`.

Bates D, Maechler M (2012). "Matrix: Sparse and Dense Matrix Classes and Methods." *R package version 1.0-6*.

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

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

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/`.

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.

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.