trustOptim: a trust region nonlinear optimizer for R

Michael Braun

MIT Sloan School of Management

August 21, 2012

Nonlinear optimization of continuous functions occurs frequently in statistics, most notably with maximum likelihood and *maximum a posteriori* (MAP) methods. Among users of R (R 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 that call among different optimization techniques, 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. Having such a large number of alternatives lets the practicing statistician choose the best available tool for the task at hand.

The problems with these methods arise when there is a large number of variables over which the objective function is 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 R matrix). Although L-BFGS-B is a limited-memory alternative to BFGS, neither is certain to offer a particularly 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 iterate x_t to iterate x_{t+1}, and then find the distance along that direction that yields the greatest improvement in the objective function. The simpler example of a line search method is "steepest descent," which follows the direction of the gradient at x_t, and stops at the best point along that ray. 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. 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 close 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 is both robust and efficient for certain kinds of large-scale problems that line search methods cannot solve. Like many other nonlinear optimizers, it is iterative, and uses gradient and Hessian estimates at each step to decide where it should move next. However, it differs from many other nonlinear optimizers in R in two important ways:

1. It uses a trust region algorithm, which is the other common class of nonlinear optimization algorithms (other than line search); and

2. It exploits the structure of the Hessian when it is sparse, which is often the case in hierarchical models.

Trust region methods work by first 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 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. We do not consider points outside of the trust region, so 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 insufficiently better than, or worse 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, but allow for steps that are large enough that the algorithm can 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. 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 be a lot faster when the objective function is expensive to compute. Although there is no guarantee that trust region algorithms will converge faster than other alternatives, they may work better for difficult optimization problems that other algorithms cannot solve.

Another attractive feature of the trustOptim package is how it exploits the structure of the objective function when its 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 parameters β_i and β_j vectors are conditionally independent, the cross-partial derivatives between all elements of β_i and β_j are zero. If the model includes a very large number of 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 only need to compute, store, and operate on 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 (Guennebaud et al. 2010), 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 the trustOptim package. We then introduce the `trust.optim` function, and describe how to use it.

1 Algorithmic details

Consider $f(x)$, an objective function over $x \in \mathbb{R}^p$ 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.

1.1 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.

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

The norm $\|\cdot\|_M$ is a Mahanalobis 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

$$\rho_t = \frac{f(x_t) - f(x_t + s_t)}{f^*(x_t) - f^*(x_t + s_t)} \tag{2}$$

This ratio is the improvement in the objective function that we would get from a move, relative to the improvement that is predicted by the quadratic approximation. Let η_1 be the minimum improvement that we need to see in $f(x)$ for us to move from x_t to x_{t+1}, and let η_2 be the maximum improvement in $f(x)$ 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_k 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 quadratic program. If B_k is positive definite, the Steihaug solution to the TRS will be exact, up to some level of numerical precision. However, if B_k is indefinite, there may be some directions in which the algorithm could move that have 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. (Conn et al. 2000) suggest using the Lanczos algorithm, instead of the Steihaug algorithm, to solve the TRS. 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 to rescale the TRS. Note that the constraint in the TRS is expressed as an M-norm, rather that a straight Euclidean norm. The matrix M is a positive definite matrix that 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 is B_t itself, but B_t is not necessarily positive definite. In this case, one could use a modified Cholesky decomposition, as described in Nocedal and Wright (2006), and this option is available in trustOptim. The package also has an option for a "diagonal" preconditioner, which is just the diagonal elements of B_t. Other preconditioners may be available in the future, but we have yet to see the need. Since the sparse matrix implementation of the Steihaug algorithm runs so quickly, the Cholesky preconditioners appear to run more slowly than just using the diagonal preconditioner, or even no preconditioner at all (e.g., the "identity" preconditioner).

1.2 Gradients and Hessians

The trustOptim package provides four trust region methods that differ only in how the Hessian matrix B is computed and stored. Two methods, Sparse and SparseFD, are optimized for objective functions with sparse Hessians. The Sparse method 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 Sparse method may 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). The SparseFD method requires only a list of the row and column indices of the non-zero elements of the lower triangle of the Hessian, but not the values themselves. The trustOptim package will use the sparsity structure of the Hessian to estimate the Hessian through through finite differences of the gradient. Thus, SparseFD is preferred if you know the structure of the Hessian, but the actual values of the Hessian are not readily available or expensive to compute.

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. The quasi-Newton Hessians are stored as dense matrices, so they are not appropriate for large problems. In our experience, neither of these methods offers clear advantages of the BFGS or L-BFGS-B implementations in `optim`, but we include them for convenience and completeness. If the problem is small and dense, but still causes problems with `optim`, you could try BFGS or SR1 in trustOptim, or you could run the SparseFD method with a sparsity structure that reflects the density of the Hessian.

For both methods, the user does need to supply a function that evaluates the gradient. 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, and is subject to numerical error, especially near the optimum when elements of the gradient are close to zero. Using the SparseFD method with finite-differenced gradients means that the Hessian is "doubly differenced," and the resulting lack of numerical precision makes those Hessians next to worthless.

1.3 Sparse finite differencing

One of the distinguishing characteristics of trustOptim is that the SparseFD method allows the user to supply the sparsity structure of the Hessian, rather than a function that computes the Hessian exactly. Naive, finite differenced estimates of a dense Hessian require $p + 1$ evaluations of the gradient if using forward differences, and $2p$ estimates for central differences (and even more if more accuracy is needed). However, for certain sparsity structures, we can estimate a Hessian using many fewer gradient evaluations. The trick is to identify groups of variables for which perturbing any subset of the variables in the group together has the exact same effect on the gradient as perturbing any one of the elements in the group alone. Such groups will exist in models for which the cross-partial derivatives across a large number of pairs of variables are zero. Curtis et al. (1974) introduce the idea of reducing the number of evaluations to estimate sparse Jacobians, and Powell and Toint (1979) describe how to partition variables into appropriate groups, and how to recover Hessian information through back-substitution. Coleman and Moré (1983) show that the task of grouping the variables amounts to a classic graph-coloring problem. Gebremedhin et al. (2005) summarize more recent advances in this area.

Suppose that we have, in a hierarchical model, N units, k heterogeneous parameters per unit, and r population-level parameters. Since the cross-partial derivatives between an element in β_i and an element in β_j is zero, any element of $beta_i$ and β_j can be in the same group, but since the cross partials for elements with a single β_i are not zero, these elements cannot be in the same group. Furthermore, if we assume that any β_i could be correlated with the r population-level parameters, and that the r population-level parameters may be correlated amongst themselves, we can estimate the Hessian (with forward differences) with no more than $k + r + 1$ gradient evaluations. Note that this number *does not grow with* N. Thus, computing the Hessian for a dataset with, say, 100 heterogeneous units, is no more expensive than for a dataset with a million heterogeneous units, and the amount of storage required for the sparse Hessian grows only linearly in N. In fact, for large N and small $k + r$, finite differencing could even be faster than direct computation. This would happen if we needed to compute $\frac{\partial^2 f}{\partial\beta_{ik}\partial\beta_{il}}$ for all $i = 1 \ldots N$.

Although efficient use of finite differencing depends on being able to partition variables into a small number of groups, the inability to partition the variables in that way might not precluse the use of sparse matrix structures. For example, suppose there are some non-zero cross-partials across heterogeneous units, but not many of them (a spatial model might have this structure). Although finite differencing might not be efficient, the trustOptim algorithms can still take advantage of sparsity if the user provides a function that computes the exact sparse Hessian.

2 Using the package

To run the algorithms in the trustOptim package, you need to call the `trust.optim` function. Its signature is:

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

All of the arguments are documented in the manual for the package. 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, a function `hs` that 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.

2.1 Methods and Hessians

There are four `methods` available in `trust.optim`: SR1, BFGS, Sparse and SparseFD. The methods differ in the way the algorithm updates an estimate of the Hessian. The Sparse method requires a user-supplied function that returns an exact Hessian as an object of the `dgCMatrix` class, as defined in the R Matrix package (Bates and Maechler 2012). The `dgCMatrix` class stores sparse matrices in a compressed format, so only the non-zero elements of the matrix are needed. The function that returns this Hessian matrix is specified in the `hs` argument in the call to `trust.optim`.

The Sparse method can still be used if the Hessian is dense, but the algorithm will not be as efficient, and the Hessian still needs to be returned as a `dgCMatrix` matrix. Use this method if you have computed an analytic gradient and an analytic Hessian, or if you are computing your objective function in another language (e.g., C++), and are using algorithmic differentiation software, such as CppAD (Bell 2012).

For the SparseFD method, you must supply `hess.struct`, a list that contains two integer vectors: `iRow` and `jCol`. These integer vectors contain the row and column indices of the non-zero elements of the lower triangle of the Hessian. The length of each of these vectors is equal to the number of non-zeros in the lower triangle of the Hessian. Do *not* include any elements from the upper triangle. Entries must be in order, first by column, and then by row within each column. Indexing starts at 1.

The `Matrix.to.Coord` function is a convenience function that converts a matrix to a list that can be used as the hess.struct argument to `trust.optim`. The only argument is M, a matrix of either of the R or Matrix variety (an R matrix will be coerced to a Matrix matrix). The function returns the row and column indices of the non-zero elements of M, ignores anything that is not in the lower triangle, and cares only if the cell is zero or not (the values to not matter). If you can construct a matrix that has the same pattern as the Hessian for your objective function, this tool is quite helpful.

Here is an example of how to supply the structure of a sparse Hessian to the SparseFD method.

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

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

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

Note that `Matrix.to.Coord` considers only the lower triangle of M.

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

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

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

Notice that M2 is only lower-triangular. Even though M was symmetric, H contains only the indices of the non-zero elements in the lower triangle. To recover the pattern of the *symmetric* matrix, do the following.

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

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

SR1 and BFGS are quasi-Newton updates that construct an approximation to the Hessian from successive estimates of the gradient. You do not need to supply any Hessian information to use them, but the approximations of the Hessian may not be very good, and they are stored as dense matrices. It is not clear that the quasi-Newton implementations in trustOptim are better than either the BFGS or conjugate gradient implementations in the `optim` function in base R, but I include them in the package for completeness.

2.2 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.

2.2.1 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 $\sqrt{\text{.Machine\$double.eps}}$, 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 "Cannot reach tolerance in gradient." 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}. Then, suppose 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.

2.2.2 Preconditioners

Currently, the package offers three preconditioners. The identity preconditioner (no preconditioning), a diagonal preconditioner (just the diagonal of the Hessian) and a modified Cholesky preconditioner. The identity and diagonal preconditioners are available for all of the methods. For the Sparse and SparseFD methods, 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 the BFGS algorithm, the Cholesky preconditioner is available because BFGS updates are always positive definite. At this time, if you select a Cholesky preconditioner for the SR1 method, the algorithm will use a diagonal 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.

3 Example: Hierarchical Binary Choice

An example of how to use the package is the file `inst/examples/ex1.R`. Suppose we have a dataset of N households, each with T opportunities to purchase a particular product. Let y_i be the number of times household i purchases the product, out of the T purchase opportunities. Furthermore, let p_i be the probability of purchase; p_i is the same for all T opportunities, so we can treat y_i as a binomial random variable. The purchase probability p_i is heterogeneous, and depends on both k continuous covariates x_i, and a heterogeneous coefficient vector β_i, such that

$$p_i = \frac{\exp(x_i'\beta_i)}{1+\exp(x_i'\beta_i)},\ i = 1\dots N \tag{3}$$

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

The log posterior density, ignoring any normalization constants, is

$$\log \pi(\beta_{1:N}, \mu | Y, X, \Sigma_0, \Omega_0) = \sum_{i=1}^{N} p_i^{y_i}(1-p_i)^{T-y_i} \left(\beta_i - \mu\right)' \Sigma^{-1} \left(\beta_i - \mu\right) + \mu'\Omega_0^{-1}\mu \tag{4}$$

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 following pattern:

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

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 in the the inst/examples/ex1.R file in the trustOptim package (the objective function `log.f`, gradient function `get.grad` and Hessian function `get.hess` are defined in the ex_funcs.R file in the same directory. What follows is a discussion of the ex1.R file.

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)
```

```
source("ex_funcs.R")  ## fn, gr and hs functions
set.seed(123)

N <- 200
k <- 5
T <- 10
```

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, 1 for a diagonal preconditioner, and 2 for a modified Cholesky preconditioner).

```
method <- "SparseFD"
control.list <- list(start.trust.radius=5,
                     stop.trust.radius = 1e-5,
                     prec=1e-7,
                     report.freq=1L,
                     report.level=4L,
                     report.precision=3L,
                     maxit=2000000L,
                     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.

```
x.mean <- rep(0,k)
x.cov <- diag(k)
mu <- rnorm(k,0,10)
Omega <- diag(k)
inv.Sigma <- rWishart(1,k+5,diag(k))[,,1]
inv.Omega <- solve(Omega)
X <- t(rmvnorm(N, mean=x.mean, sigma=x.cov))
B <- t(rmvnorm(N, mean=mu, sigma=Omega))
XB <- colSums(X * B)
log.p <- XB - log1p(exp(XB))
Y <- laply(log.p, function(q) return(rbinom(1,T,exp(q))))
```

```
nvars <- N*k + k
start <- rnorm(nvars) ## random starting values
```

The `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, and is problem-dependent.

```
hess.struct <- get.hess.struct(N, k)  ## for SparseFD method only
```

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

```
cat("running ",method, "\n")
t1 <- Sys.time()
opt <- trust.optim(start, fn=log.f,
                   gr = get.grad,
                   hs = get.hess,  ## used only for Sparse method
                   hess.struct = hess.struct, ## used only for SparseFD method
                   method = method,
                   control = control.list,
                   Y=Y, X=X, inv.Omega=inv.Omega, inv.Sigma=inv.Sigma
                   )
t2 <- Sys.time()
td <- difftime(t2,t1)
print(td,units="secs")
```

```
running  SparseFD
Beginning optimization

iter      f    nrm_gr                  status     rad  CG iter        CG result
  1  12888.0   6114.2  Continuing - TR expand    15.0      1  Intersect TR bound
  2  10011.2   4774.7  Continuing - TR expand    45.0      1  Intersect TR bound
  3   4491.7   1422.1  Continuing - TR expand   135.0      1  Intersect TR bound
  4    822.6    133.1  Continuing - TR expand   405.0      7  Intersect TR bound
  5    393.1     17.2              Continuing   405.0     55  Reached tolerance
  6    304.2      7.1              Continuing   405.0     61  Reached tolerance
  7    260.2      3.2              Continuing   405.0     58  Reached tolerance
  8    248.6      1.1              Continuing   405.0     62  Reached tolerance
  9    247.6      0.1              Continuing   405.0     63  Reached tolerance
 10    247.6      0.0              Continuing   405.0     65  Reached tolerance
 11    247.6      0.0              Continuing   405.0     63  Reached tolerance

Iteration has terminated
 11     247.58      0.00                Success

Time difference of 0.3053319 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 the latter case, "Reached Tolerance" suggests a solution to the TRS that is in the interior of the trust region.

Note that this problem has 1,005 parameters, and converged in less than one third 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 use the Sparse method instead of SparseFD, the trust region iterations are exactly the same, except that the algorithm takes almost 10 seconds to run. This is because it takes longer to construct a block diagonal Hessian with $N = 200$ blocks than it does to compute a sparse Hessian using finite differencing and 10 partitions. This may not always be the case if there are a large number of population-level parameters.

4 Discussion

The motivation behind the trustOptim package was immense frustration about not being able to find posterior 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 saddlepoints 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 that 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.

4.1 Alternatives

There is another trust region package for R, called “trust,” written by Geyer. (2009). This package is a robust implementation, and works well for modestly-sized problems for which the user can provide an exact estimate of the Hessian can be computed efficiently. The basic trust region algorithm is similar to the one used in trustOptim, with some important differences.

1. To solve the TRS, trustOptim package uses the Steihaug conjugate gradient algorithm, as opposed to the eigenvalue decompositions used in the trust package.
2. The trustOptim package does not require the user to supply an exact estimate of the Hessian, as long as the structure of the Hessian is provided;
3. If the Hessian is sparse, trustOptim makes special arrangements to exploit that sparsity; and
4. The trustOptim package uses the norm of the gradient as the stopping criterion, instead of the relative change in the objective function (more on that below)

In some cases, we found that the trust package can actually outperform trustOptim, and for many problems, trust may be the preferred tool. The motivation behind trustOptim was to make some additional trade-offs and enhancements to deal with large-scale problems, especially when the Hessian is sparse.

Naturally, there are many other optimization tools available for R users. These are described in the R Task View on Optimization and Mathematical Programming. Table 1 summarizes the differences of the alternatives that were discussed in this paper.

Package	method	Type	Requires gradient	Requires Hessian
optim	CG	Line search	No, but preferred	No
optim	BFGS	Line search	No, but preferred	No
optim	Nelder-Mead	Simplex	No	No
trust		trust region	Yes	Yes
trustOptim	BFGS/SR1	trust region	Yes	No
trustOptim	Sparse	trust region	Yes	Yes
trustOptim	SparseFD	trust region	Yes	structure only

Table 1: Summary of some popular nonlinear optimization algorithms for R.

4.2 Implementation details

The trustOptim package was written primarily in C++. The code makes extensive use of the Eigen Numerical Library (Guennebaud et al. 2010), which contains classes for sparse matrices and a complete set of optimized linear algebra routines. The finite differencing algorithm for the SparseFD method is described in Coleman et al. (1985a). The actual Fortran code was published by Coleman et al. (1985b) as Algorithm 636 in the ACM Transactions on Mathematical Software. I then converted this code to C, using `f2c` (this is to avoid having to call a Fortran compiler, or link to Fortran libraries, when installing the package). I offer special thanks to Tom Coleman for giving me permission to include his code in the package.

The trustOptim package is freely available for use. In general, the code is released under the Mozilla Public License, version 2.0 (MPL), with the exception of the source code to the finite differencing algorithms by Coleman et al. (1985b). The copyright for that code is held by the Association for Computing Machinery (ACM). Details of both licenses are in the LICENSE file that is included with the package.

If you do use the trustOptim package for work that leads to a publication, please let me know, and please cite this paper. I put a lot of work into it, and I would like to get some credit for it.

4.3 Future plans

In the future, more features will be added to trustOptim. Here is a list of some possibilities, in no particular order:

1. additional preconditioners to accelerate convergence to solutions of the trust region subproblem;
2. implementation of the Lanczos algorithm for solving the trust region subproblem (Conn et al. 2000, ch. 5.2);
3. the ability to handle constrained optimization problems; and
4. an interface with an algorithmic differentiation package, once one is available for R.

References

Douglas Bates and Martin Maechler. Matrix: Sparse and Dense Matrix Classes and Methods. *R package version 1.0-6*, 2012.

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

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

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

Thomas F Coleman, Burton S Garbow, and Jorge J Moré. Algorithm 636: FORTRAN Subroutines for Estimating Sparse Hessian Matrices. *ACM Transactions on Mathematical Software*, 11(4):378, 1985b.

Andrew R Conn, Nicholas I M Gould, and Philippe L Toint. *Trust-Region Methods*. SIAM-MPS, Philadelphia, 2000.

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

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

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

Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

Jorge Nocedal and Stephen J Wright. *Numerical Optimization*. Springer, second edition edition, 2006.

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

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

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