Explaining Gradient Minimizers in R

John C. Nash

12/02/2022

Abstract

This vignette is intended to aid understanding and maintenance of the function minimization methods available to users of R. R users commonly, if incorrectly, call this optimization, though most base or packaged methods for "optimization" carry out unconstrained or bounds constrained function minimization.

General Approach

The general problem is to find the vector of parameters x that gives at least a local minimum of an objective function $f(\mathbf{x}, Y)$, where Y is a collection of exogenous data that is needed to compute the function, possibly subject to constraints that for generality could be expressed as a vector of functions that are positive. That is

$$\mathbf{c}(\mathbf{x}, Y) \geq \mathbf{0}$$

We do not attempt to consider here the annoying detail of a constraint requiring strictly "greater than" constraints. In such cases, the expedient is to use a tiny quantity in place of the relevant zero. Furthermore, maximization of $f(\mathbf{x}, Y)$ can be achieved by minimizing $(-1) * f(\mathbf{x}, Y)$.

From $f(\mathbf{x}, Y)$, we can generally compute two important related objects:

- the **gradient** of $f(\mathbf{x}, Y)$ is the vector of first partial derivatives with respect to the parameters $\mathbf{x}$
- the **Hessian** of $f(\mathbf{x}, Y)$ is the matrix of second partial derivatives with respect to the parameters $\mathbf{x}$. In one dimension, the Hessian is the slope of the gradient.

Gradient minimizers are minimization methods that require the gradient or an approximation to be available. Other minimizers are either **direct search** approaches or **stochastic** methods that use pseudo-random trial sets of parameters. From gradient-related calculations, gradient minimizers propose a **search direction t** then try to find new sets of parameters with lower function value by moving from $\mathbf{x}$ along $\mathbf{t}$. At some such lower point, a new search direction is computed and a new search begun. This iterative process stops when no more progress can be made.

As with many similar tasks, the details give rise to a great many particular algorithms and programs. There can be no proof that any particular choice is "best", and we rely on accumulated experience with test and other problems to suggest the best approaches in particular situations.

Line Search

The **line search** is a key sub-problem for gradient minimization methods, and for convenience we will discuss it first, then move on to the issue of choosing a method to compute search directions.

The line search is an attempted one-dimensional minimization of

$$z(s) = f(\mathbf{x} + s * \mathbf{t}, Y)$$

since s is a scalar. We assume that $\mathbf{t}$ is a descent direction, and indeed this should be verified within programs, since rounding and other errors may cause that assumption to be violated. Typically we look for a negative gradient projection $\mathbf{g}^t * \mathbf{t}$. Since we are only looking at positive s values, 0 is a lower bound to the line search. We also may need to contend with lower and upper bounds (box constraints) on the parameters $\mathbf{x}$ that provide an upper limit s_{max}.

Line search methods follow a couple of standard themes:

- generate trial values of s by some reasonable heuristic and stop when the derived parameters $\mathbf{x}$ satisfy some **acceptable point** criterion.
- choose a suitable set of values of s so that a **model** $m(s)$ of the function $z(s)$ can be approximated. This model should be one for which the minimizing value s_m is easily computed, and we compute $z(s_m)$ in the hope that it is lower than any objective function value so far found.

"Acceptable" is a slippery concept, but commonly some version of the Wolfe or Armijo conditions (see https://en.wikipedia.org/wiki/Wolfe_conditions) are used. This is slightly stronger than requiring that the function value has simply been lowered in our search. However, we have not got any assurance that we have "minimized" $z(s)$. It is important to at least check that the new point is "lower" than any other so far, and to use the lowest point found so far. Depending on the manner in which the search direction $\mathbf{t}$ has been generated, we then may try a different search direction such as the negative gradient $\mathbf{t} = -\mathbf{g}$ or, if we are satisfied no further progress is possible, terminate the minimization.

Similarly annoying is the possibility that our model does not do a good enough job of approximating $z()$. The blackboard description of the process leaves out a great deal of the software safeguards needed to avoid silly answers being proposed to problems.

Backtracking line search

Let us consider a simple but quite effective strategy that is used in the `Rvmmin` Variable Metric minimizer (which uses the same underlying algorithm as `optim::BFGS` in base R).

Given a downhill search direction t, the gradient projection $q = \mathbf{g}^t * \mathbf{t}$, a small number (e.g., 0.0001) which we call α and some initial value of s, we

A) compute $z(s)$

B) If $z(s) < z(0) + s * q * \alpha$, accept s as an acceptable point; otherwise reduce s and repeat from (A).

If there are bounds constraints, the initial s may be limited by the distance to the closest bound. If this is smaller than the default initial stepsize, it may reduce the number of trial sets of parameters in the backtracking stepsize reduction.

Typically s is reduced by some fixed ratio, e.g., multiplying by a number such as 0.5 (halving) or some other fraction. I like 0.15 or 0.2 in order to reduce the stepsize quickly when that is important.

Quadratic inverse interpolation line search

A fairly obvious model for $z()$ is a quadratic one. That is,

$$m(s) = as^2 + bs + z(0) \approx z(s)$$

whose minimum should be found at $-b/(2a)$.

We note that we need two more conditions to find a and b. The gradient projection provides one of these, since $m'(0) = b \approx q$. A third piece of information comes from evaluating $z(s_0)$ for some choice of s_0.

In practice, poor choices of s_0 give poor results. A better approach, in my experience at least, is to first carry out a backtracking line search to an acceptable point (or at least a lower one), and use the stepsize to that point as s_0. This means that $z(s_0) < z(0)$. We then can solve our model approximation to find

$$a = (z(s_0) - z(0) - q * s_0)/{s_0}^2$$

so that

$$s_{min} = -q/(2 * a)$$

In software, we need to be careful how things are computed to avoid digit cancellation or division by very small quantities, but by and large this quadratic approximation approach is quite reliable and efficient.

Approximate derivatives

The calculation of the gradient and possibly the Hessian are an important aspect of all gradient-based methods, but an error-prone and often computationally demanding task. The choices for the user of optimization methods are:

- to put in the intellectual effort to develop formulas and code to compute the derivatives needed. Note that analytic expressions for deriviatives may be available in all, or even most, cases.
- to use symbolic or automatic differentiation (Margossian (2019)). R has some built-in tools in base R and in packages such as "Deriv" by Clausen and Sokol (2018). However, these are, in my experience, difficult to use. While symbolic differentiation, if it works, results in analytic expressions that can generally be implemented concisely, automatic differentiation can generate a large amount of stored data.
- numerical approximation to derivatives builds on the introductory calculus idea that the first derivative of a function $f(x)$ of a single variable x being defined as

  $$f'(x) = \lim_{h \to 0}((f(x + h) - f(x))/h)$$

  Finite differences lead to several formulas, some as simple as forward or backward difference approximations, or the more accurate central difference formula, but at the expense of an extra function evaluation. Greater accuracy can be obtained using more function evaluations, as in Gilbert (2009). For the gradient and the Hessian the number of parameters n means we need at least n and n^2 evaluations respectively. Generally evaluation of an expression for the gradient has a computational cost not hugely larger than that of a single function evaluation. Automatic differentiation is more complicated, but numerical approximation is often the most costly.

If approximations are used, minimization software generally follows a somewhat different trajectory of iterates – the list of sets of parameters $\mathbf{x}$ – than the same program using analytic gradients and Hessian. On the other hand, it has often be observed that approximations do well in early iterations when the initial parameters are "far" from the solution.

Where analytic gradients are particularly helpful is in deciding to terminate a method, since we can be more confident that a "small" gradient norm is correctly reported. This is especially so if the function value is quite large, since the derivative approximations must subtract nearly equal numbers.

Bounds and masks constraints

Bounds or box constraints complicate our programs.

In particular, the step size in the line search may be limited by bounds. If $\mathbf{l} \leq \mathbf{x} \leq \mathbf{u}$

$$s_{max} = min((\mathbf{l} - \mathbf{x})/\mathbf{t}, (\mathbf{u} - \mathbf{x})/\mathbf{t})$$

This is then used as the initial step in the line search if it is greater than the chosen default starting step. Since we use

$$z(s) = f(\mathbf{x} + s * \mathbf{t}, Y)$$

we need to compute $\mathbf{x} + s * \mathbf{t}$ for a value of s that should bring some elements of $\mathbf{x}$ to a bound. Unfortunately, rounding errors can cause elements of the parameter to very slightly violate the bound. Finite difference derivative approximation calculations can also cause such violations. These are difficult to properly control in software.

Masks or fixed parameters are more easily accommodated, since we can simply force the appropriate gradient elements to zero for the fixed parameters. That is, we apply a "mask" to the gradient to zero out those elements. However, as a means to indicate that a parameter is fixed, we could set lower and upper bounds for the fixed parameters as equal.

Checking input information

The idea of telling our minimization program that parameters are fixed by setting lower and upper bounds equal raises the related question of inputs that are invalid. To clarify the vocabulary, I prefer to say a specification of a problem is **infeasible** if a parameter is not on or between lower and upper bounds. Here it is the parameters that violate constraints. However, a problem is **inadmissible** if a lower bound exceeds an upper bound, in that such a problem makes no sense.

Within software, it is sometimes a great nuisance if the starting parameters $\mathbf{x}$ violate bounds. For example, we have updated bounds but forgotten to adjust the starting parameters. Some programs then simply move parameters to the nearest bound and continue with the minimization, since users generally want a solution. However, users should be informed that initial parameters have been changed. Package `optimx` (for which this article was originally written; see J. C. Nash and Varadhan (2011)) provides functionality of this sort, along with a number of other checks for inputs that could be thought of as "silly" and usually the result of simple user errors.

Yet another check is that the objective function or constraints can be evaluated at the initial parameters. R provides a useful mechanism for such checks in the `try()` function. On the other hand, if users know that their objective function should not be evaluated in certain regions of the parameter space, it is sometimes worth explicitly coding for such eventualities. Here the fact that R objects can be given **attributes** is helpful. Another approach is to return a very large value when a function cannot be evaluated. In `optimx` the standard `control` list contains the element `bigval` preset to `.Machine$double.xmax*0.01` or approximately 1.797693e+306. This is set smaller than the biggest representable number in case any computations are performed on the returned value, but it can be debated whether 0.01 is an appropriate reduction factor, and I welcome discussion with users who actually apply this idea.

Steepest descents

Steepest descents (or gradient descent) is one of the simplest ideas for a minimizer. We use $\mathbf{t} = -\mathbf{g}$, perform a line search, then repeat.

While one can demonstrate that the gradient provides the most "downhill" direction of search, the iterative process tends to be quite inefficient in that it takes a long time to converge. Nevertheless, the negative gradient is often a useful tool to restart other methods when a method suffers some failure.

Newton-like methods

A long-standing approach to optimization is Newton's method, though it is likely Newton would not recognize modern forms of it. The idea is to seek a zero (or root) of the gradient $\mathbf{g}$. For this we need the Hessian H, i.e., the slope of the gradient in 1 dimension. We evaluate `H` and `g` at our current point `x` and then solve

$$H\mathbf{t} = -\mathbf{g}$$

to generate the search direction. The classic Newton method iterates with a full step of $\mathbf{t}$, but safeguarded Newton methods use some form of line search to ensure $f(\mathbf{x} + s * \mathbf{t}, Y)$ is lower than $f(\mathbf{x}, Y)$. Iterating hopefully finds us a stationary point of the gradient, which should be a flat point on the surface of $f()$. Details and constraints complicate the implementation of these ideas, and the computation of the gradient and especially the Hessian are often very tricky.

Newton-like ideas inspire many other gradient methods. Commonly we wish to avoid storing or explicitly computing H.

Variable Metric and quasi-Newton methods

In this article, at the risk of offending some purists, these terms will be treated as synonymous, though some workers do make a distiction.

We have seen that Newton's method gets a search direction $\mathbf{t}$ from solving

$$H\mathbf{t} = -\mathbf{g}$$

If the Hessian H is invertible, then this is equivalently written

$$\mathbf{t} = -H^{-1}\mathbf{g}$$

The central idea of variable metric (VM) or quasi-Newton (QN) methods is to replace the initial Hessian or its inverse with some approximation that is much easier to compute and which (hopefully) allows a simpler solution of the equations that define $\mathbf{t}$. In the kth iteration we perform a search along the computed $\mathbf{t}$ to get $\mathbf{x_{k+1}}$ starting from $\mathbf{x_k}$. Using "old" and "new" gradients, parameters and function values, we derive "better" approximations of either H or H^{-1}. The goal is to have updates that are efficient to calculate and preserve some desirable properties for the approximate Hessian or Hessian inverse such as symmetry and positive-definiteness. The actual Hessian may not be positive definite, though it will be symmetric.

The first such update was by Davidon in 1959, circulated as a mimeograph report that was rejected by journal referees. It was finally published by SIAM as Davidon (1991) when thousands of references to an essentially unpublished but critical piece of research work became an embarrassment to the field. https://en.wikipedia.org/wiki/Davidon%E2%80%93Fletcher%E2%80%93Powell_formula gives a reasonable discussion of the DFP algorithm.

https://en.wikipedia.org/wiki/Broyden%E2%80%93Fletcher%E2%80%93Goldfarb%E2%80%93Shanno_algorithm presents the main alternative to the DFP approach. The BFGS family of approaches has become dominant for the variable metric class of minimizers. As usual, programming details can heavily influence practical behaviour of software and there are thousands of papers and programs. In base R, `optim()` has methods labelled `BFGS` and `L-BFGS-B`. The former is an adaptation (I did not program it however) of my Algorithm 21 from J. C. Nash (1979), which itself was developed after a January 1976 discussion with Roger Fletcher of the method described in Fletcher (1970) to try to find a very compact code. At the time 8K bytes was considered a reasonable amount of memory for a small computer. `L-BFGS-B` is based on ideas in Byrd et al. (1995) and a quite complicated code that I feel generally uneasy discussing, despite having worked to provide an updated version of it for R users (see Fidler et al. (2018)).

Rvmmin

`Rvmmin()` is an all-R implementation of a bounds-constrained version of my own Algorithm 21 from J. C. Nash (1979) that is now part of package `optimx`, though it was earlier a separate package. While the base R `optim::BFGS` can be faster, being coded in C, it is not as easy to read, maintain or modify as the R code of `Rvmmin`.

Conjugate Gradient methods

Extending the conjugate gradients method for solving **linear** equations of Hestenes and Stiefel (1952) to minimizing nonlinear objective functions, Fletcher and Reeves (1964) started a busy field of endeavour that aims to accomplish the minimization with as small a working memory as possible.

The idea is to start with some descent direction – the negative gradient is traditional – and perform a line search. A new gradient is then combined with information from the present line search information to produce a new **t** that is somehow "conjugate" to, and linearly independent of, previous search directions. For a problem with n parameters, we can generate at most n such directions, so must restart our process after n **cycles**, or before if there are indications of "loss of conjugacy".

The brilliance of Fletcher and Reeves (1964) is that the formulas for their "new" search direction are extremely simple. However, there are alternative formulations of the algorithm which, depending on various assumptions and approximations, are almost equally simple but give somewhat different sets of search directions and minimization performance.

In the 1970s I explored several such formulas for the updated search direction, and included three possibilities in Algorithm 22 of J. C. Nash (1979), but chose that of Polak and Ribière (1969) as the default. Unfortunately, my experience with this algorithm, which is implemented in base R as `optim::CG`, has been very unsatisfactory.

By contrast, an approach that melds the update ideas by Dai and Yuan (2001) has led me to build `Rcgmin()`, now part of package `optimx` (J. C. Nash and Varadhan (2011)), though originally an independent R package. This has proven remarkably strong in performance. I added bounds and masks constraints.

Rcgmin and ncg

NOTE: In early 2022 I discovered a coding glitch in the bounds constrained code that ignored the possibility that a quadratic inverse interpolation would suggest a negative stepsize, which would be "uphill". A crude correction to double the best steplength so far in such cases was added to block possible failure.

Other ideas for simplifying Newton's method

The quasi-Newton/variable metric methods used one approximation of the Hessian or its inverse to provide a Newton-like search direction. Another approach is to approximately solve the Newton equations rather than approximate the Hessian. Such methods are typically called **inexact Newton** or **truncated Newton** methods. They often do NOT explicitly generate the Hessian, but use clever ideas which generate, at least approximately, the effect of multiplying a vector by the Hessian without actually having the matrix itself. Having such a matrix-vector product allows for solution of the Newton equations using the **linear** conjugate gradient or similar methods. These are iterative, and stopping them "early" may still give a useful search direction while saving computational effort. Hence the "inexact" or "truncated" nomenclature. See S. G. Nash (2000) for a survey.

Rtnmin

The method "Rtnmin" calls the functions `tn()` or `tnbc()` (for unconstrained or bounds constrained problems). These functions are R translations of Matlab functions due to S. G. Nash (1983). The translation is not always well-coded for R, and these routines could be improved, but often work quite well, especially for problems with large numbers of parameters.

Limited-memory quasi-Newton methods

`optimx` offers several versions of these ideas as the bounds constrained solvers "L-BFGS-B" from base-R `optim()` and its revision in C as `lbfgsb3c` (Fidler2018), as well as the unconstrained solver "lbfgs()" (Coppola2014). The idea behind these methods is to avoid storing the full Hessian or inverse Hessian, but instead to keep only a few of the vectors of information needed to create them via the updating formulas. The details require care and attention. Moreover, all the "work" of these codes is programmed in languages other

than R. They are particularly helpful for problems with large numbers of parameters such as the variable dimension problem number 25 of Moré, Garbow, and Hillstrom (1981). In the example below, warnings about the lack of symmetry of the returned Hessian are suppressed.

```
# example using var_dim function from More et al. (or funconstrain)
bfn = function(par) {
  n <- length(par)
  if (n < 1) {stop("Variably Dimensioned: n must be positive")}
  fsum <- 0;  fn1 <- 0;
  for (j in 1:n) {
    fj <- par[j] - 1; fsum <- fsum + fj * fj; fn1 <- fn1 + j * fj
  }
  fn1_fn1 <- fn1 * fn1  # f_n+1 and f_n+2
  fsum <- fsum + fn1_fn1 + fn1_fn1 * fn1_fn1
  fsum
}
bgr = function(par) {
  n <- length(par)
#  if (n < 1) {stop("Variably Dimensioned: n must be positive")  }
  fsum <- 0;  grad <- rep(0, n);  fn1 <- 0
  for (j in 1:n) {
    fj <- par[j] - 1; fn1 <- fn1 + j * fj; grad[j] <- grad[j] + 2 * fj
  }
  fn1_2 <- fn1 * 2; fn13_4 <- fn1_2 * fn1_2 * fn1
  grad <- grad + 1:n * (fn1_2 + fn13_4)
  grad
}
x0<-rep(pi,100)
library(optimx)
defaultW <- getOption("warn")
options(warn = -1)
mm<-c("ncg","Rcgmin","lbfgs","L-BFGS-B","Rtnmin")
res1<-opm(x0, bfn, bgr, method=mm)
options(warn = defaultW)
res1[,100:108]
```

```
##          p100        value fevals gevals hevals convergence kkt1 kkt2 xtime
## ncg         1 5.902531e-25     58     13      0           0 TRUE TRUE 0.028
## Rcgmin      1 1.239035e-23     73     17      0           0 TRUE TRUE 0.002
## lbfgs       1 4.845483e-16     62     62      0           0 TRUE TRUE 0.048
## L-BFGS-B    1 8.548800e-18     60     60      0           0 TRUE TRUE 0.002
## Rtnmin      1 3.714564e-13     86     86      0           0 TRUE TRUE 0.006
```

Acknowledgement

References

Byrd, Richard H., Peihuang Lu, Jorge Nocedal, and Ci You Zhu. 1995. "A Limited Memory Algorithm for Bound Constrained Optimization." *SIAM Journal on Scientific Computing* 16 (5): 1190–1208. https://doi.org/10.1137/0916069.

Clausen, Andrew, and Serguei Sokol. 2018. *Deriv: R-Based Symbolic Differentiation*. https://CRAN.R-project.org/package=Deriv.

Dai, Y. H., and Y. Yuan. 2001. "An Efficient Hybrid Conjugate Gradient Method for Unconstrained Optimization." *Annals of Operations Research* 103 (1-4): 33–47.

Davidon, William C. 1991. "Variable Metric Method for Minimization." *SIAM Journal on Optimization* 1

(1): 1–17.

Fidler, Matthew L, John C Nash, Ciyou Zhu, Richard Byrd, Jorge Nocedal, and Jose Luis Morales. 2018. *Lbfgsb3c: Limited Memory BFGS Minimizer with Bounds on Parameters with Optim() 'c' Interface*. https://CRAN.R-project.org/package=lbfgsb3c.

Fletcher, R. 1970. "A New Approach to Variable Metric Algorithms." *Computer Journal* 13 (3): 317–22.

Fletcher, R., and C. M. Reeves. 1964. "Function Minimization by Conjugate Gradients." *The Computer Journal* 7 (2): 149–54. https://doi.org/10.1093/comjnl/7.2.149.

Gilbert, Paul. 2009. *numDeriv: Accurate Numerical Derivatives*. R Foundation for Statistical Computing. http://www.bank-banque-canada.ca/pgilbert.

Hestenes, M. R., and E. Stiefel. 1952. "Methods of Conjugate Gradients for Solving Linear Systems." *Journal of Research of the National Bureau of Standards* 49 (6): 409–36.

Margossian, Charles C. 2019. "A Review of Automatic Differentiation and Its Efficient Implementation." *WIREs Data Mining and Knowledge Discovery* 9 (4). https://doi.org/10.1002/widm.1305.

Moré, Jorge J., Burton S. Garbow, and Kenneth E. Hillstrom. 1981. "Testing Unconstrained Optimization Software." *J-Toms* 7 (1): 17–41.

Nash, John C. 1979. *Compact Numerical Methods for Computers: Linear Algebra and Function Minimisation*. Bristol: Adam Hilger.

Nash, John C., and Ravi Varadhan. 2011. "Unifying Optimization Algorithms to Aid Software System Users: Optimx for R." *Journal of Statistical Software* 43 (9): 1–14. https://doi.org/10.18637/jss.v043.i09.

Nash, Stephen G. 1983. "Truncated-Newton Methods for Large-Scale Minimization." In *Applications of Nonlinear Programming to Optimization and Control*, edited by H. E. Rauch, 91–100.

———. 2000. "A Survey of Truncated-Newton Methods." *Journal of Computational and Applied Mathematics* 124: 45–59.

Polak, E., and G. Ribière. 1969. "Note sur la convergence de méthodes de directions conjuguées." *Rev. Française Informat Recherche Opérationelle* 3 (1): 35–43.