gradmin: a general **R** routine for function minimization using gradients

John C. Nash

2017-04-10

Gradient function minimization algorithms

There is a huge number of methods for function minimization based on the use of gradients of the function to be minimized. This vignette presents **gradmin**, an **R** function that allows many, possibly most, such algorithms to be tried, at least those that are not of the **reverse communication** type. ??ref

Function minimization is the problem of finding the set of parameters x_* for which there is no lower value of the function $f(x_*)$. Note that we can **maximize** the function by minimizing $ - f(x)$. Also note that the set of parameters x_* may not be unique. There can be multiple or even an infinite number of sets of parameters which yield the lowest function value. There may also be no such set of parameters if the function has no lower bound, for example, $ f(x) = x $.

We will begin with **gradminu** for unconstrained minimization, that is, with no constraints. However, we will eventually wish to add various constraints on the parameters x, in particular, bounds or box constraints,

$$l_i <= x_i <= u_i$$

where i is an index over all the parameters.

It is also useful to consider **masks**, or fixed paramters. That is, we may wish to fix the value of some of the parameters and only minimize the function over the rest. This can be very useful in practice, since many problems include parameters which are set externally for a variety of reasons. At some stage, we may wish to allow these parameters to be changed.

Additional information

Almost all approaches to gradient minimization require not just the function $f(x)$, but also its gradient $g(x)$, a function that returns the vector of first derivatives of f with respect to the parameters at the specified set of paramters. Furthermore, we may also have the Hessian, the matrix of second derivatives, computed by a routine that returns $H(x)$. However, many methods do not require explicit functions for generating approximations or surrogates for these objects.

Furthermore, we shall usually require a set of starting parameters x_0, though some methods may generate this start from the bounds vectors l and u. Choice of the starting parameters may critically affect the performance of a minimization method.

The general structure

Minimization methods generally have the following structure:

- a **setup** phase, in which the function and possibly the gradient and hessian are provided, along with a set of controls that particularize the minimization method;

There follows an iteration of the following three components:

- the **termination** test, which allows a decision to continue the method and seek parameters with lower function values, or to stop and report our result;
- the generation of a **search** direction, that is a vector of values of which we add a multiple called the **steplength** to the current parameter vector to generate new trial points at which we hope to find lower function values;
- the **line search**, which chooses (gets) a particular step length along the search direction according to some rule. Generally the step length must be positive, and many methods test or rely on the assumption of the search direction being "downhill".

Finally, we may have a **report** phase. In this we may **test** as well as report results, for example, by computing the eigenvalues of the (possibly approximate) Hessian. In the context of the **gradmin** and most of my own packages, reporting is considered external to the minimization and our "report" is generally limited to returning the proposed best parameters and the function value at those parameters and possibly the gradient evaluated there, along with some counts of the number of functions, gradients and hessians computed.

Thus we have a structure:

```
SETUP

Iterate over:
     TERMTEST
     GENSRCH
     GETSTEP

 REPORT
```

Termination tests

Termination is often confused with **convergence**. The former applies to methods or programs, and is the way in which we stop our method. Termination tests do include tests that look at the mathematical conditions for a minimum, in particular a very small gradient. In some cases we may also test that the Hessian matrix at the suggested minimal set of parameters is positive definite, However, the computation of a Hessian, analytic or approximate, can require a large computational effort (in fact, greater than that of finding the supposed minimum), so it is common to leave this to the reporting stage of the minimization process.

Search direction generation

Generating a search direction is the stage of a method that is generally used to categorize it. Let us call the search direction t. Then a so-called Newton method will generate t as the solution of

$$H(x) * t = -g(x)$$

A quasi-Newton or variable metric method – and I ask to be excused for failing here to provide differentiating details – uses some rule to compute an initial Hessian approximation H_0 and sequentially alters this at each iteration or cycle to some matrix H_k where k is the iteration counter. The update generally purports to generate an approximation to the true Hessian, but the reality of such methods is that "approximation" is a very loose concept in this context.

A very simple search generation is given by the **steepest descent** direction, namely,

$$t = -g(x)$$

Despite the name, this is not generally a good method, as it tends to do poorly in practice, with successive search directions being too similar. **Conjugate gradient** methods do much better. These start with the steepest descents direction, then update the search by some simple rules in an attempt to generate directions which are not too similar to previous ones. However, the "rules" and formulas generally require some care to avoid numerical difficulty, and the cycle may need to be restarted at some point.

Some, but not all, methods require the solution of linear equations equations. There are many methods for doing this, some of which could be approximate. The choice of such a **solver** is a sub-classifier of our methods.

The line search

While it is perfectly possible to use a unit step along the search direction, we generally want to try to reduce the function value by searching along t, and try to find a good set of new parameters

$$x_{new} = x + step * t$$

Thus we can had a "naive" method where we set $step = 1$, the choice of the traditional Newton method, but in practice, this can sometimes mean that the function value is not reduced. Generally, we try to apply some computations that reduce the function value by a suitable choice for *step*.

One possibility is to use a method that seeks the minimum of a function of one variable. Thus we find the minimum of $f(x + step * t)$ with respect to *step*. There are several possibilities for doing this which can be explored, but in most cases we do not require a very precise step length.

A generally more efficient method uses a simple backtrack choice. We start with $step = 1$. If the function is suitably reduced, we replace *step* with $ r * step $ where r is less than 1. I often use 0.1 or 0.2. We repeat until x_t is "suitable".

Suitablity is generally decided using the "sufficient decrease" condition,

$$f(x_t + step * s) < f(x_t) + c * step * g(x_t)^T * s$$

where c is some number less than 1. Typically $ c = 1e-4 = 0.0001 $. We refer to this as the Armijo condition. Note that the product of gradient times search vector is negative for any reasonable situation, since we are trying to go "downhill". The condition can be made slightly more stringent by taking the absolute value of the gradient projection, giving the Wolfe conditions. See https://en.wikipedia.org/wiki/Wolfe_conditions.

However, for the Marquardt approach above, it is common to use a unit step, but in the event we fail to reduce the function, we increase λ. This is repeated until either the function is reduced or the parameters are unchanged. (If λ gets very large, the search vector becomes tiny, so there is no change in x_t.) Otherwise, it is usual to have some sort of line search. There are many possibilities.

So-called **Newton** methods are among the most commonly mentioned in the solution of nonlinear equations or function minimization. However, as discussed in https://en.wikipedia.org/wiki/Newton%27s_method#History, the **Newton** or **Newton-Raphson** method as we know it today was not what either of its supposed originators knew.

This vignette discusses the development of some safeguarded variants of Newton methods for function minimization in **R**. Note that there are some resources in **R** for solving nonlinear equations by Newton-like methods in the packages **nleqslv** and **pracma**.

The basic approach

If we have a function $f(x)$, with gradient $g(x)$ and second derivative (Hessian) $H(x)$ the first order condition for an extremum (min or max) is

$g(x) = 0$

To ensure a minimum, we want

$H(x) > 0$ $

The first order condition leads to a root-finding problem.

It turns out that x need not be a scalar. We can consider it to be a vector of parameters to be determined. This renders $g(x)$ a vector also, and $H(x)$ a matrix. The conditions of optimality then require a zero gradient and positive-definite Hessian.

The Newton approach to such equations is to provide a guess to the root x_try and to then solve the equation

$H(x_t) * s = -g(x_t)$

for the search vector s. We update x_t to $x_t + s$ and repeat until we have a very small gradient $g(x_t)$. If $H(x)$ is positive definite, we have a reasonable approximation to a (local) minimum.

Motivations

A particular interest in Newton-like methods its theoretical quadratic convergence. See https://en.wikipedia.org/wiki/Newton%27s_method. That is, the method will converge in one step for a quadratic function $f(x)$, and for "reasonable" functions will converge very rapidly. There are, however, a number of conditions, and practical programs need to include safequards against mis-steps in the iterations.

The principal issues concern the possiblity that $H(x)$ may not be positive definite, at least in some parts of the domain, and that the curvature may be such that a unit step $x_t + s$ does not reduce the function f. We therefore get a number of possible variants of the method when different possible safeguards are applied.

Algorithm possibilities

There are many choices we can make in building a practical code to implement the ideas above. In tandem with the two main issues expressed above, we will consider

- the modification of the solution of the main equation $ H(x_t) * s = - g(x_t)$ so that a reasonable search vector s is always generated

Some choices to compute the search vector

The primary concern in solving for s is that the Hessian may not be positive definite. This means that we cannot apply fast and stable methods like the Cholesky decomposition to the matrix. At the time of writing, we consider the following approaches:

- Attempt to solve $H(x_t) * s = -g(x_t)$ with **R** directly, and rely on internal checks to catch any cases where the solution fails. We can use `try()` to stop the program in this case.
- Use a Levenberg-Marquardt (??ref) stabilization to ensure that we have an augmented Hessian that is positive definite. Essentially, we create $H_aug = H + \lambda * I$ where I is the unit matrix of the size of H, and λ is a scalar chosen to ensure the resulting matrix is positive definite.

- Use the singular value decomposition and drop any singular planes where the singular values fall below some threshhold. Note that deciding the threshhold is possibly a non-trivial matter.

Note that within the above general choices for solution, we could try to specify how the solution is obtained, since there are various ways to solve linear equations and to find the singular value decomposition. (??refs, discussion??)

?? nf, ng, nh etc.

?? Marquardt stabilization ??