Using and extending the optimr package

John C. Nash (nashjc at *uottawa.ca)*

2016-07-08

Abstract

`optimr` and `optimrx` are wrapper R packages to allow the regular R `optim()` structure to be applied when using many different optimization packages available to R users. In particular, we want

- to use the structure of the `optim()` call
- to provide for parameter scaling for all methods via `control$parscale`
- to return a consistent result structure
- to allow a number of methods to be compared easily via the `opm()` function (as such, `optimr` replaces the `optimx` package).

The package `optimr` is intended to be available as an official CRAN package, so has a limited set of underlying optimizers. `optimrx` will be available on alternative package repositories (initially R-forge, in the **optimizer** project), and will allow the use of many optimizers. Since some of these optimizers may become deprecated or otherwise unavailable, or else not be part of the official R repositories, I expect that the `optimrx` package will, from time to time, suffer glitches and failures.

It is fully intended that users may wish to extend the packages, especially their `optimr()` functions. `optimr()` and `ctrldefault` are the only places where the functions in the packages differ, with `ctrldefault.R` listing the optimizers available in each package and `optimr.R` having relevant code to match. Note that there are multiple package lists in the `ctrldefault()` function for unconstrained, bounded and masked parameters.

Package `optimrx` may also be expected to contain extra tests, commentary, and other material of a developmental or experimental nature. Nevertheless, I believe users may find this larger collection more useful if they are having to attack difficult problems.

Overview

optimr is a package intended to provide improved and extended function minimization tools for R. Such facilities are commonly referred to as "optimization", but the original `optim()` function and its replacement in this package, which has the same name as the package, namely `optimr()`, only allow for the minimization or maximization of nonlinear functions of multiple parameters subject to at most bounds constraints. Many methods offer extra facilities, but apart from **masks** (fixed parameters) for the `hjn`, `Rcgmin` and `Rvmmin`, such features are likely inaccessible via this package without some customization of the code.

In general, we wish to find the vector of parameters `bestpar` that minimize an objective function specified by an R function `fn(par, ... )` where `par` is the general vector of parameters, initially provided as the vector `par0`, and the dot arguments are additional information needed to compute the function. Function minimization methods may require information on the gradient or Hessian of the function, which we will assume to be furnished, if required, by functions `gr(par, ...)` and `hess(par, ....)`. Bounds or box constraints, if they are to be imposed, are given in the vectors `lower` and `upper`.

As far as I am aware, all the optimizers included in the `optimr` package are **local** minimizers. That is, they attempt to find a local minimum of the objective function. Global optimization is a much bigger problem. Even finding a local minimum can often be difficult, and to that end, the package includes a function `kktchk()` to test the Kuhn-Karush-Tucker conditions. These essentially require that a local minumum has a zero gradient and all nearby points on the function surface have a greater function value. There are many details that are ignored in this very brief explanation.

It is intended that `optimr` can and should be extended with new optimizers and perhaps with extra functionality. Moreover, such extensions should be possible for an R user reasonably proficient in programming R scripts. These changes would, of course, be made by downloading the source of the package and modifying the R code to make a new, but only locally available, package. The author does, on the other hand, welcome suggestions for inclusion in the distributed package, especially if these have been well-documented and tested. Over 95% of the effort in building `optimr` has been to try to ensure that errors and conflicts are purged, and that the code is resistant to mistakes in user inputs.

How the optimr() function (generally) works

`optimr()` is an aggregation of wrappers for a number of individual function minimization ("optimization") tools available for R. The individual wrappers are selected by a sequence of `if()` statements using the argument `method` in the call to `optimr()`.

To add a new optimizer, we need in general terms to carry out the following:

- Ensure the new function is available, that is, the package containing it is installed, and the functions `import`ed into `optimr`;
- Add an appropriate `if()` statement to select the new "method";
- Translate the `control` list elements of `optimr()` into the corresponding control arguments (possibly not in a list of that name but in one or more other structures, or even arguments or environment variables) for the new "method";
- If necessary, redefine the R function or functions to compute the value of the function, gradient and possibly Hessian of the objective function so that the output is suited to the method at hand.
- When derivative information is required by a method, we may also need to incorporate the possibility of numerical approximations to the derivative information.
- Add code to check for situations where the new method cannot be applied, and in such cases return a result with appropriate diagnostic information so that the user can either adjust the inputs or else choose a different method.
- Provide, if required, appropriate links to modified function and gradient routines that allow the parameter scaling `control$parscale` to be applied if this funtionality is not present in the methods. To my knowledge, only the base `optim()` function methods do not need such special scaling provisions.
- As needed, back-transform scaled parameters and other output of the different optimization methods.

Bounds

A number of methods support the inclusion of box (or bounds) constraints. This includes the function `nmkb()` from package `dfoptim`. Unfortunately, this method uses the transfinite transformation of the objective function to impose the bounds (Chapter 11 of Nash, 2014), which causes an error if any of the initial parameters are on one of the bounds.

There are several improvements in the `optimr` and `optimrx` packages relating to bounds that would be especially nice to see, but I do not have any good ideas yet how to implement them all. Among these unresolved improvements are:

- to use the transfinite approach to permit bounds to be supplied for all the unconstrained optimization methods;
- to automatically adjust the bounds or the parameters very slightly to allow initial parameter sets to be provided with the initial parameters on a bound. I think it would be important to issue a warning in such cases.
- to flag or otherwise indicate to the user which approach has been used, and also to allow control of the approach. For example, the transfinite approach could be used with unconstrained versions of some of the methods that allow bounds to permit comparisons of the effectiveness of transfinite versus active-set approaches. Note that these possibilities increase the complexity of the code and may be prone to bugs.

Masks (fixed parameters)

The methods `Rcgmin` and `Rvmmin` (and possibly others, but not obviously accessible via this package) also permit fixed (masked) parameters. This is useful when we want to establish an objective function where one or more of the parameters is supplied a value in most situations, or for which we want to fix a value while we optimize the other parameters. At another time, we may want to allow such parameters to be part of the optimization process.

In principle, we could fix parameters in methods that allow bounds constraints by simply setting the lower and upper bounds equal for the parameters to be masked. As a computational approach, this is generally a very bad idea, but in the present `optimr()` this is permitted for methods `Rcgmin` and `Rvmmin` but should give an error for other bounds-capable methods. In the case of the two methods mentioned, the bounds signal that masks are active, and the parameters in question are more or less excluded from the computations except for their role in evaluating function and gradient values.

Issues in adding a new method

Adjusting the objective function for different methods

The method `nlm()` provides a good example of a situation where the default `fn()` and `gr()` are inappropriate to the method to be added to `optimr()`. We need a function that returns not only the function value at the parameters but also the gradient and possibly the hessian. Don't forget the dot arguments which are the exogenous data for the function!

```
nlmfn <- function(spar, ...){
   f <- efn(spar, ...)
   g <- egr(spar, ...)
   attr(f,"gradient") <- g
   attr(f,"hessian") <- NULL # ?? maybe change later
   f
}
```

Note that we have defined `nlmfn` using the scaled parameters `spar` and the scaled function `efn` and gradient `egr`. That is, we develop the unified objective plus gradient AFTER the parameters are scaled.

In the present `optimr()`, the definition of `nlmfn` is put near the top of `optimr()` and it is always loaded. It is the author's understanding that such functions will always be loaded/interpreted no matter where they are in the code of a function. For ease of finding the code, and as a former Pascal programmer, I have put it near the top, as the structure can be then shared across several similar optimizers. There are other methods that compute the objective function and gradient at the same set of parameters. Though `nlm()` can make use of Hessian information, we have chosen here to omit the computation of the Hessian.

Parameter scaling

Parameter scaling is a feature of the original `optim()` but generally not provided in many other optimizers. It has been included (at times with some difficulty) in the `optimr()` function. The construct is to provide a vector of scaling factors via the `control` list in the element `parscale`.

In the tests of the package, and as an example of the use and utility of scaling, we use the Hobbs weed infestation problem (./tests/hobbs15b.R). This is a nonlinear least squares problem to estimate a three-parameter logistic function using data for 12 periods. This problem has a solution near the parameters c(196, 49, 0.3). In the test, we try starting from c(300, 50, 0.3) and from the much less informed c(1,1,1). In both cases, the scaling lets us find the solution more reliably. The timings and number of function and gradient evaluations are, however, not necessarily improved for the methods that "work" (though these measures are

all somewhat unreliable because they may be defined or evaluated differently in different methods – we use the information returned by the packages rather than insert counters into functions). However, what values of these measures should we apply for a failed method?

As a warning – having made the mistake myself – scaling must be applied to bounds when calling a bounds-capable method.

Function scaling

`optim()` uses `control$fnscale` to "scale" the value of the function or gradient computed by `fn` or `gr` respectively. In practice, the only use for this scaling is to convert a maximization to a minimization. Most of the methods applied are function **minimization** tools, so that if we want to maximize a function, we minimize its negative. Some methods actually have the possibility of maximization, and include a `maximize` control. In these cases having both `fnscale` and `maximize` could create a conflict. We check for this in `optimr()` and try to ensure both controls are set consistently.

Modified, unused or unwanted controls

Because different methods use different control parameters, and may even put them into arguments rather than the `control` list, A lot of the code in `optimr()` is purely for translating or transforming the names and values to achieve the desired result. This is sometimes not possible precisely. A method which uses `control$trace = TRUE` (a logical element) has only "on" or "off" for controlling output. Other methods use an integer for this `trace` object, or call it something else that is an integer, in which case there are more levels of output possible.

I have found that it is important to remove (i.e., set NULL) controls that are not used for a method. Moreover, since R can leave objects in the workspace, I find it important to set any unused or unwanted control to NULL both before and after calling a method.

Thus, if `print.level` is the desired control, and it more or less matches the `optimr() control$trace`, we need to set

```
print.level <- control$trace
control$trace <- NULL
```

After the method has run, we may need to reset `control$trace`.

Methods in other computing languages

When the method we wish to call is not written in R, special care is generally needed to get a reliable and consistent operation. Typically we call an R routine from `optimr()`. Let us call this routine `myop()`. then `myop()` will set up and call the underlying optimizer.

For FORTRAN programs, Nash (2014), Chapter 18, has some suggestions. Particular issues concern the dot arguments (ellipsis or ... entries to allow exogenous data for the objective function and gradient), which can raise difficulties. In package `optimrx` the interface to the `lbfgs` shows one approach, which consolidates the arguments for `lbfgs()` into a list, then converts the list to an environment.

Running multiple methods

It is often convenient to be able to run multiple optimization methods on the same function and gradient. To this end, the function `opm()` is supplied. The output of this by default includes the KKT tests and other

informatin in a data frame, and there are convenience methods `summary()` and `coef()` to allow for display or extraction of results.

`opm()` is extremely useful for comparing methods easily. I caution that it is not an efficient way to run problems, even though it can be extremely helpful in deciding which method to apply to a class of problems.

An important use of `opm()` is to discover cases where methods fail on particular problems and initial conditions of parameters and settings. This has proven over time to help discover weaknesses and bugs in codes for different methods. If you find that such cases, and your code and data can be rendered as an easily executed example, I strongly recommend posting it to one of the R lists or communicating with the package maintainers. That really is one of the few ways that our codes come to be improved.

Polyalgorithms – multiple methods in sequence

Function `polyopt()` is intended to allow for attempts to optimize a function by running different methods in sequence. The call to polyopt() differs from that of `optimr()` or `opm()` in the following respects:

- The `method` character argument or character vector is replaced by the `methcontrol` array which has a set of triplets consisting of a method name (character), a function evaluation count and an iteration count.
- the `control$maxit` and `control$maxfeval` are replaced, if present, with values from the `methcontrol` argument list.

The methods in `methcontrol` are executed in the sequence in which they appear. Each method runs until either the specified number of iterations (typically gradient evaluations) or function evaluations have been completed, after which the best set of parameters so far is passed to the next method specified, or the function exits.

Polyalgorithms may be useful because some methods such as Nelder-Mead are fairly robust and efficient in finding the region in which a minimum exists, but then very slow to obtain an accurate set of parameters. Gradients at points far from a solution may be such that gradient-based methods do poorly when started far away from a solution, but are very efficient when started "nearby". Caution, however, is recommended. Such approaches need to be tested for particular applications.

Multiple sets of starting parameters

For problems with multiple minima, or which are otherwise difficult to solve, it is sometimes helpful to attempt an optimization from several starting points. `multistart()` is a simple wrapper to allow this to be carried out. Instead of the vector `par` for the starting parameters argument, however, we now have a matrix `parmat`, each **row** of which is a set of starting parameters.

In setting up this functionality, I chose NOT to allow mixing of multiple starts with a polyalgorithm or multiple methods. For users really wishing to do this, I believe the available source codes `opm.R`, `polyopt.R` and `multistart.R` provide a sufficient base that the required tools can be fashioned fairly easily.

Counting function, gradient and hessian evaluations

Different methods take different approaches to counting the computational effort of performing optimizations. Sometimes this can make it difficult to compare methods.

- When using numerical gradient approximations, it would be more sensible to report 0 gradient evaluations, but count each function evaluation.
- "Iterations" may be used as a measure of effort for some methods, but an "iteration" may not be comparable across methods.

Derivatives

Derivative information is used by many optimization methods. In particular, the **gradient** is the vector of first derivatives of the objective function and the **hessian** is its second derivative. It is generally non-trivial to write a function for a gradient, and generally a lot of work to write the hessian function.

While there are derivative-free methods, we may also choose to employ numerical approximations for derivatives. Some of the optimizers called by `optimr` automatically provide a numerical approximation if the gradient function (typically called `gr`) is not provided or set NULL. However, I believe this is open to abuse and also a source of confusion, since we may not be informed in the results what approximation has been used.

For example, the package `numDeriv` has functions for the gradient and hessian, and offers three methods, namely a Richardson extrapolation (the default), a complex step method, and a "simple" method. The last is either a forward of backward approximation controlled by an extra argument `side` to the `grad()` function. The complex step method, which offers essentially analytic precision from a very efficient computation, is unfortunately only applicable if the objective function has specific properties. That is, according to the documentation:

```
This method requires that the function be able to handle complex valued
arguments and return the appropriate complex valued result, even though
the user may only be interested in the real-valued derivatives.  It also
requires that the complex function be analytic.
```

The default method of `numDeriv` generally require multiple evaluations of the objective function to approximate a derivative. The simpler choices, namely, the forward, backward and central approximations, require respectively 1, 1, and 2 function evaluations.

To keep the code straightforward, I decided that if an approximate gradient is to be used by `optimr()` (or by extension the multiple method routine `opm()`), then the user should specify the name of the approximation routine as a character string in quotations marks. The package supplies four gradient approximation functions for this purpose, namely "grfwd" and "grback" for the forward and backward simple approximations, "grcentral" for the central approximation, and "grnd" for the default Richardson method via `numDeriv`. It should be fairly straightforward for a user to copy the structure of any of these routines and call their own gradient, but at the time of writing this (2016-6-29) I have not tried to do so. An example using the complex step derivative would also be useful to include in this vignette.

Functions besides `optimr()` in the package

opm()

As mentioned above, this routine allows a vector of methods to be applied to a given function and (optionally) gradient. The pseudo-method "ALL" (upper case) can be given on its own (not in a vector) to run all available methods. If bounds are given, "ALL" restricts the set to the methods that can deal with bounds.

optchk()

This routine is an attempt to consolidate the function, gradient and scale checks.

ctrldefault()

This routine provides default values for the `control` vector that is applicable to all the methods for a given size of problem. The single argument to this function is the number of parameters, which is used to compute

values for termination tolerances and limits on function and gradient evaluations. However, while I believe the values computed are "reasonable" in general, for specific problems they may be wildly inappropriate.

Functions used from the `optextras` package

Optimization methods share a lot of common infrastructure, and much of this has been collected in my `optextras` package. The routines used in the current packages are as follows.

kktchk()

This routine, which can be called independently for checking the results of other optimization tools, checks the KKT conditions for a given set of parameters that are thought to describe a local optimum.

grfwd(), grback(), grcentral() and grnd()

These have be discussed above under Derivatives.

fnchk(), grchk() and hesschk()

These functions are provided to allow for detection of user errors in supplied function, gradient or hessian functions. Though we do not yet use hessians in the optimizers called, it is hoped that eventually they can be incorporated.

`fnchk()` is mainly a tool to ensure that the supplied function returns a finite scalar value when evaluated at the supplied parameters.

The other routines use numerical approximations (from `numDeriv`) to check the derivative functions supplied by the user.

bmchk()

This routine is intended to trap errors in setting up bounds and masks for function minimization problems. In particular, we are looking for situations where parameters are outside the bounds or where bounds are impossible to satisfy (e.g., lower > upper). This routine creates an indicator vector called bdmsk whose values are 1 for free parameters, 0 for masked (fixed) parameters, -3 for parameters at their lower bound and -1 for those at their upper bound. (The particular values are related to a coding trick for BASIC in the early 1980s.)

scalechk()

This routine is an attempt to check if the parameters and bounds are roughly similar in their scale. Unequal scaling can result in poor outcomes when trying to optimize functions using derivative free methods that try to search the paramter space. Note that the attempt to include parameter scaling for all methods is intended to provide a work-around for such bad scaling.

Program inefficiencies

One of the unfortunate, and only partially avoidable, inefficiencies in a wrapper function such as `optimr()` is that there will be duplication of much of the setup and error-avoidance that a properly constructed optimization program requires. That is, both the wrapper and the called programs will have code to accomplish similar goals. Some of these relate to the following.

- Bounds constraints should be checked to ensure that lower bounds do not exceed upper bounds.
- Parameters should be feasible with respect to the bounds.
- Function and gradient evaluations should be counted.
- Function and gradient code should satisfy some minimal tests.
- Timing of execution may be performed surrounding different aspects of the computations, and the positioning of the timing code may be awkward to place so the timing measures equivalent operations. For example, if there is a significant setup within some routines and not others, we should try to time either setup and optimization, or just optimization. However, there is often some awkwardness that prevents a clean placement of the timing.

Besides these sources of inefficiency, there is a potential cost in both human effort and program execution if we "specialize" variants of a code. For example, there can be separate unconstrained and constrained routines, and the wrapper should call the appropriate version. `Rvmmin` has a top-level routine to decide between `Rvmminu` and `Rvmminb`, but `optimr()` takes over this selection. A similar choice exists within `dfoptim` for the Hooke and Jeeves codes. While previously, I would have chosen to separate the bounded and unconstrained routines, I am now leaning towards a combined routine for the Hooke and Jeeves. First, I discovered that the separation seems to have introduced a bug, since the code was structured to allow a similar organization for both choices, where possibly a different structure would have been better adapted for efficient R. Note, however, that I have not performed appropriate timings to support this conjecture. Second, I managed to implement a bounds constrained HJ code from a description in one of may own books in less time than it took to try (not fully successfully) to correct the code from `dfoptim`, in part because the development and stable versions of the latter are quite different, though both failed the test function `bt.f()` example. I did get a modified `dfoptim` code to function correctly, but am not fully sure why my changes worked.

Note that this is not a criticism of the creators of `dfoptim`. I have made similar choices myself with other packages. And it is challenging to balance clarity, maintainability, efficiency, and common structure for a suite of related program codes.

Testing the package

- individual method tests
- problem tests

References

Nash, John C. (2014) Nonlinear Parameter Optimization Using R Tools, Wiley: Chichester.