Using and extending the optimx package

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

2023-06-15

Abstract

`optimx` is a wrapper package to allow the multiple methods for function minimization available in R and CRAN packages to be used through a consistent syntax. This vignette is a significant update of an earlier one that describes some of the issues in building and extending the package. However, it is intended mainly for developers, including ourselves, as an aid to and partial record of work done. Thus it is an incomplete and imperfect document not aimed at users.

NOTE: Some material also appears in the "Intro to optimx" vignette

Background

The base R distribution includes three tools that allow for function minimization, possibly with bounds constraints. These are `optim()`, `nlm()` and `nlminb()`. There is also the nonlinear least squares solver `nls()` that will **not** be discussed here except when there is overlap with particular matters at hand.

Because the syntax for `optim()` is relatively convenient and already spans five different multiparameter minimization methods (or **solvers**). There is also method "Brent" in `optim()` that is a one-parameter, or one-dimensional, minimizer. Separately, base R has the one-parameter function `optimize()`.

The `optim()` syntax, from the manual, is

```
optim(par, fn, gr = NULL, ...,
      method = c("Nelder-Mead", "BFGS", "CG", "L-BFGS-B", "SANN",
                 "Brent"),
      lower = -Inf, upper = Inf,
      control = list(), hessian = FALSE)
```

Here `par` is the vector of starting parameters for the function `fn` which has an (optional) gradient function `gr`. `method` specifies the solver to be used. It actually must be a **single** character string. An attempt to call two solvers by, say,

```
t2 <- optim(par=c(2,2), fn=fr, gr=grr, method=c("BFGS", "Nelder-Mead"))
```

generates the message

```
Error in match.arg(method) : 'arg' must be of length 1
```

The benefits of a unified syntax are seen immediately if we look how `nlm()` and `nlminb()` must be called.

```
nlm(f, p, ..., hessian = FALSE, typsize = rep(1, length(p)),
    fscale = 1, print.level = 0, ndigit = 12, gradtol = 1e-6,
    stepmax = max(1000 * sqrt(sum((p/typsize)^2)), 1000),
    steptol = 1e-6, iterlim = 100, check.analyticals = TRUE)
```

```
nlminb(start, objective, gradient = NULL, hessian = NULL, ...,
       scale = 1, control = list(), lower = -Inf, upper = Inf)
```

The reader can query the help for these functions (`?nlm` or `?nlminb`) to get the meaning of the arguments. Here I will note particularly

- the names, order and structure of the parameters and objective function are inconsistent across the three tools;
- the controls have different names and meanings and are sometimes arguments in the call, and sometimes elements of a control vector;
- the "answer" returned by the three tools is similar but not quite interchangeable in structure.

When we extend our interest to CRAN packages, these issues multiply.

Moreover, there are different features of the R tools and CRAN packages that are NOT general. It would be helpful to users if they were consistently available across a wider range of solver functions. Some of these are:

- to provide for parameter scaling for all methods via `control$parscale` or a similar (but consistent) mechanism;
- to return a consistent result structure that as far as possible matches the list `optim()` returns;
- to allow a number of methods to be called in a single multi-solver function, `opm()`;
- to permit some post-solution checks and extra optimization infrastructure to be consistently applied across a number of solvers;
- to allow for bounds constraints and fixed (masked) parameters to be specified, where the solvers allow;
- to provide a didactic version of a direct search Hooke-and-Jeeves solver;
- to provide for the use of explicit Hessian function in some existing solvers and two didactic safeguarded Newton solvers.

The updated package incorporates the earlier `optimx()` function which inspired the present work. That function uses a slightly different call and returns a slightly different result structure from the more recent `opm()` function. A number of CRAN and other packages use `optimx()`; see Nash and Varadhan (2009). Because we do not wish to cause breakages of working packages and tools, we have retained the `optimx()` function in the current version.

WARNING: only updates to correct egregious errors in `optimx()` are envisaged going forward. Please use `optimr()` and `opm()`.

It is fully intended that both developers and users may wish to extend the package, especially by adding solvers to the `optimr()` function. The file `ctrldefault.R` lists the optimizers currently available via this function in the code `optimr.R`. Note that there are multiple package lists in the `ctrldefault()` function for unconstrained, bounded and masked parameters. Users can add lists suitable to their needs. Indeed, this vignette is intended to assist in the the addition of other solvers, as well perhaps as extra functionality or features.

Ideally, such extensions should be possible for any R user reasonably proficient in programming R scripts. 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. I caution, however, that over 95% of the effort in building and improving `optimx` has been to try to ensure that errors and conflicts are purged, and that the code is resistant to mistakes in user inputs.

"Optimization"

The general nonliear optimization problem considers the minimization or maximization of an objective function subject to equality and inequality constraints that are both linear and nonlinear. It comprehends the field of mathematical programming, and borders on nonlinear equations.

In R, what is referred to as "optimization", including the original `optim()` function and its replacement in this package, only allows 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` methods, such features are likely inaccessible via this package without some customization of the code. Even masks are just bounds constraints where the upper and lower bounds coincide.

For our purposes, we wish to find the vector of (real valued) 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 a vector that is either the first argument to `optimr()` or else specified by a `par=` argument. 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 `optimx` package are **local** minimizers, possibly apart from the "SANN" method of `optim()`. That is, the solvers 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 `optimx` package offers the 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.

Developments of the `optimx()` function

Users will note that there is much overlap between functions `optimx()` and `opm()`. The reasons for the restructuring to `optimr()` and `opm()` are as follows:

- `optimx` has a large set of features, which increases the complexity of maintenance;
- `opm()` is built upon `optimr()` which is designed to call any one of a set of solvers, so extensions reside largely in the `optimr()` function;
- there are small but possibly important differences in the result structures from `optimx()` and `opm()`. For example, `opm()` does not report the number of "iterations" in the variable `niter` because such a value is not returned by either the original `optim()` or newer `optimr()` functions.
- `optimr()` is structured so it can use the original `optim()` syntax and result format, together with the `parscale` control allowing parameter scaling for all solvers. `optimx()` only allows `parscale` for some solvers.
- `optimx()` was intended to have features to allow for calling solvers sequentially to create polyalgorithms, as well as for using multiple vectors of starting parameters. The combination of such options may cause unexpected behaviour.
- `polyopt()` (built on `optimr()`) allows for running solvers sequentially to create a polyalgorithm. For example, one may want to run a number of cycles of method `Nelder-Mead`, followed by the gradient method `Rvmmin`.
- `multistart()` allows multiple starting vectors to be used in a single call.

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 `imported` into `optimr`;
- Include the function in one or more of the lists of methods in the `ctrldefault()` function;
- 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, update `ctrldefault()`;
- 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 (see the section of `optimr()` for the `nlm()` function, for example);
- 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 functionality 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, and reset any `control` items.

An internal structure for bounds and masks

The `Rvmmin` and `Rcgmin` packages use the (integer) vector `bdmsk` to indicate the status of each parameter with respect to bounds and masks.

- `bdmsk[i] =  1` indicates parameter `i` is free
- `bdmsk[i] =  0` indicates parameter `i` is fixed at its current value (masked)
- `bdmsk[i] = -1` indicates parameter `i` is at its upper bound (`upper[i]`)
- `bdmsk[i] = -3` indicates parameter `i` is at its lower bound (`lower[i]`)

This particular structure was developed for use with the BASIC programs in Nash and Walker-Smith (1987), primarily for active set methods, and returned in the `bdmsk` list item of the answer for `Rvmmin()` and `Rcgmin()`. These need to know the direction in which a step can be taken for each parameter. If the gradient component `i` is positive, then the step needs to be a negative one, which it can only be if there is no lower bound in the way. A free parameter is clearly fine, or one that is on an upper bound. Adding 2 to `bdmsk[i]` must be positive, but if it is negative, we are at a lower bound and cannot move. Similarly we can use a negative gradient component and step forward from a lower bound. The codes `Rvmminb.R` and `Rcgminb.R` provide details, which are fairly straightforward, but tedious.

I believe that the same structure could be ported to other gradient-based codes, for example, `snewton` or `snewtonm`.

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` package relating to bounds that would be especially nice to see, but I do not have 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. It would be important to issue a warning in such cases.
- to flag or otherwise indicate to the user which approach to bounds 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 `hjn`, `Rcgmin` and `Rvmmin` (and possibly others, but not obviously accessible yet via this package) also permit fixed (masked) parameters. See Nash and Walker-Smith (1987). 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 as a way to **signal** that a parameter is fixed. Internally a different representation is used.

The method `hjn` is a Hooke and Jeeves axial search method that allows masks and bounds. It is coded using explicit loops, so will generally be much slower than an implementation (e.g., `hjkb` from `dfoptim` or the similar code in package `pracma`) that try to employ vectorized computations. `hjn` was included in `optimr` to provide an example of a direct search method with masks. I do NOT recommend it for general use.

There is a possibility that masks could be implemented globally in optimr.

- masked parameters can be selected by `mskd <- which(lower >= upper)`
- if `idx <- 1:length(par)`, then 'idx <- idx[which(lower<=upper)] are parameters that take part in the optimization
- `idx` can be passed into the working function and gradient routines `efn` and `egr` in the same way scaling is performed.

At the time of writing (2016-7-11) this has yet to be tried. However, we have got a test of the transformation.

```
## test masked transformation

# set of 10 parameters
par <- 3*(1:10)
cat("par:")
```

```
## par:
```

```
print(par)
```

```
##  [1]  3  6  9 12 15 18 21 24 27 30
```

```
cat("Mask 3rd, 5th, 7th\n")
```

```
## Mask 3rd, 5th, 7th
```

```
bdmsk<-rep(1,10) # indicator of parameters that are free
bdmsk[3] <- 0
bdmsk[5] <- 0
bdmsk[7] <- 0
cat("bdmsk:")
```

```
## bdmsk:
```

```
print(bdmsk)
```

```
##  [1] 1 1 0 1 0 1 0 1 1 1
```

```
# want to produce xpar which are the reduced parameters
iactive <- which(bdmsk == 1)
cat("iactive (length=",length(iactive),"):")
```

```
## iactive (length= 7 ):
```

```
print(iactive)
```

```
## [1]  1  2  4  6  8  9 10
```

```
xpar <- par[iactive]
cat("xpar:")
```

```
## xpar:
```

```
print(xpar)
```

```
## [1]  3  6 12 18 24 27 30
```

```
xpar <- - xpar
print("altered xpar:")
```

```
## [1] "altered xpar:"
```

```
print(xpar)
```

```
## [1]  -3  -6 -12 -18 -24 -27 -30
```

```
cat("expand back to newpar\n")
```

```
## expand back to newpar
```

```
newpar <- par
newpar[iactive] <- xpar
cat("newpar:")
```

```
## newpar:
```

```
print(newpar)
```

```
##  [1]  -3  -6   9 -12  15 -18  21 -24 -27 -30
```

```
# Need to combine with scaling to get full setup for optimr()

# Then also think of the transfinite approach for bounds on unconstrained
## Or even for bounds methods but using unconstrained part.
```

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. There is a further risk of conflict with the use of numerical approximation of derivatives, as discussed later.

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`.

There are some programming issues in the package with method controls and these are discussed in a separate section.

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 the interface to this "foreign" language solver `myop()`.
Then `myop()` will set up and call the underlying optimizer and return (supposedly) an answer.

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 `optimr` the interface to the `lbfgs` shows one approach, which consolidates the arguments for `lbfgs()` into a list, then converts the list to an environment.

Using separate unconstrained and bounds constrained functions

As an attempt to avoid executing unnecessary code, for some years I built separate unconstrained and constrained routines with a common umbrella routine. Hence,

- `Rvmmin()` calls either `Rvmminu()` or `Rvmminb()`
- `Rcgmin()` calls either `Rcgminu()` or `Rcgminb()`
- the method specification "Rtnmin" calls either `tn()` or `tnbc()`

More recently, with improvements in the just-in-time compiler for R, I am exploring consolidated codes, e.g., `nvm()` and `ncg()` though these are far from fully developed. However, they also are designed to be called from `optimr()` and not run independently, whereas the routines mentioned above originally were part of independent R packages. The newer codes rely on `optimr()` to perform sanity checks on supplied parameters, bounds and functions. In particular, `bmchk()` is called to check parameters and bounds.

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 information 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 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, or termination tests cause the method to be exited, after which the best set of parameters so far is passed to the next method specified. If there is no further method, `polyopt()` 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 matrix. It is generally non-trivial to write a function for a gradient, and a huge amount 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. There is also the question of counting function and gradient evaluations correctly.

Specifying that numerical gradients are to be used, and which method for such approximation, presents a software interfacing issue. Around 2010 I introduced the idea of making the name of a gradient approximation program a character string e.g., "grfwd" and "grcentral" for forward and central derivative approximations. Direct codes for the gradient are included in the call by name NOT in character form.

This idea has been helpful, but it has raised some concerns.

- as described below, there may be conflicts with the `maximize` control;

- there is an inefficiency if the (already evaluated) current best function value is not used in computing the gradient approximation;
- bounds constraints can be violated by the argument step to compute the gradient approximation by finite differences;
- some approximations may be inappropriate or inadmissible for the current objective function.

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 requires 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 (extra) function evaluations for each parameter.

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 `optimx` package supplies five gradient approximation functions for this purpose, namely "grfwd" and "grback" for the forward and backward simple approximations, "grcentral" for the central approximation, "grnd" for the default Richardson method via `numDeriv`, and "grpracma" for function `grad` from package `pracma`. It should be fairly straightforward for a user to copy the structure of any of these routines and build their own gradient approximation, but I have not tried to document how to do this.

An example using the complex step derivative would also be useful to include in this vignette. I welcome contributions!

Note As at 2018-7-8 Changcheng Li and John Nash had preliminary success in using the `autodiffr` package at https://github.com/Non-Contradiction/autodiffr/ to generate gradient, hessian and jacobian functions via the Julia language automatic differentiation capabilities. There are some issues of slow performance compared to native-R functions to compute these collections of derivative information, but the ease of generation of the functions and the fact that they generate analytic results i.e., as good as R can compute the information, allows for some improvements in results and also makes feasible the application of some Newton-like methods. Unfortunately, rapid changed in Julia functions have made it very difficult to keep this capability operational.

Conflict between numerical gradients and maximization controls

A nasty inconsistency that has caused me quite a lot of difficulty in developing and maintaining `optimx` is that the use of a character string `gr` function, implying a numerical approximation to the gradient, comes into conflict with either of the controls `maximize=TRUE` or `fnscale=-1` (or other negative number).

The issue arises because we need to negate the `gr` function when it is analytic, but the numerical approximation works on the **already** scaled objective function. Great care is needed to avoid trouble, especially as individual solvers may also try to apply scalings.

Duplications and other efficiency considerations

Most optimization routines perform preliminary checks on the inputs. Clearly any wrapper function such as `optimr()` or `opm()` also has such checks. There is naturally duplication. While this may be minor in time and effort cost, it would be more elegant to be able to use just one set of checks.

The obvious simple gradient approximations “grfwd” and “grback” compare the function at a step away from a base point given by `par`. Looking at the code, we have

```
grfwd <- function(par, userfn, fbase=NULL, env=optsp, ...) {
   # Forward different gradient approximation
   eps<-env$deps
   if (is.null(fbase)) fbase <- userfn(par, ...)
   # ensure we have function value at par
   df <- rep(NA, length(par))
   teps <- eps * (abs(par) + eps)
   for (i in 1:length(par)) {
      dx <- par
      dx[i] <- dx[i] + teps[i]
      df[i] <- (userfn(dx, ...) - fbase)/teps[i]
   }
   df
}
```

In many solvers, we would generally use the current “best” point for the vector `par` as our base for computing the gradient. Thus if `n = length(par)`, we compute `fbase n` times when we already have the value unless the solver code is clever enough to provide it. Because other approximations, “grcentral” in particular, do not use `fbase` (even though it appears in the function specification), avoiding recalculation would require our solvers to be aware of the particular approximation that has been given in the call. Nevertheless, it is clear there are opportunities for improved efficiency.

Structuring the `optimr()` function

Given the `optimr` has been set up to

- prepare internal functions `efn` and `egr` that are to be **minimized**. This means that **internal** calls to solvers within `optimr()` have `maximize` set to `FALSE` and `fnscale` set to `1`.
- pass NULL to internal function `egr` if `gr` is NULL; this may cause some methods to fail
- `parscale` is applied to the objective function parameters and we minimize **scaled** parameters and transform the values back to the original scale when returning an answer on exiting `optimr()`.

It is open to discussion wheter we should flag (i.e., explicitly report in the output result or elsehwere) that an approximation is being used for the gradient or hessian.

A test script was written to explore this structure and is part of the package in `inst/doc/`testefnegr.R‘.

Special cases such as `nlm()`

nlm() needs an objective function that has gradient and hessian functions as attributes. As illustrated in `inst/doc/examples/woodtest.R`, this takes a form like

```
wood.fgh <- function(x){ # For info only
   fval <- wood.f(x)
   gval <- wood.g(x)
   hval <- wood.h(x)
   attr(fval,"gradient") <- gval
   attr(fval,"hessian") <- hval
   fval
}
```

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 as a single character string (not in a vector) to run all available methods. A more useful pseudo-method is “MOST” which leaves out some less-useful methods. If bounds are given, the list of methods specified is restricted 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.

`ctrldefault()` was developed by recording the different controls of many functions and packages in a spreadsheet to allow for their recording and comparison. This spreadsheet, `optcontrol.xls` also records the returned answer structure. It is part of the `inst/doc/` collection of information in the package.

dispdefault()

This routine (in file `ctrldefault.R`) is intended to allow a compact display of the current default settings used within `optimx`.

Functions that were formerly in the `optextras` package

Optimization methods share a lot of common infrastructure, and much of this was collected in my retired `optextras` package. (Now in the `optimx` package.) The routines used in the current package 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(), grnd() and grpracma()

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 that was used in Nash and Walker-Smith (1987).) Note that this function allows parameter values outside the bounds to be shifted to the nearest bound (when `shift2bound=TRUE`, which is the default behaviour).

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 parameter space. Note that the attempt to include parameter scaling for all methods is intended to provide a work-around for such bad scaling. Because nonlinear functions have scaling that changes across the parameter domain, `scalechk()` should be treated with caution.

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 are often coding peculiarities that prevent 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, the separate unconstrained and constrained routines, with the wrapper calling 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 my 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. Ravi Varadhan has corrected the codes on CRAN.

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

Providing controls to different algorithms

We have already noted that it is important to provide control settings for the different methods. This can be a challenge, largely because some methods (or at least their instantiations in **R** codes or wrappers to other languages) only permit selected controls. The `ctrldefault` function provides as many of the controls as possible, and provides them via the `control` list. However, many methods put the contols into separate arguments of their function call. For example, the `control$maxit` iteration limit is `iterlim` for the function `nlm()`.

A more subtle difficulty is that the multiple-method wrapper `opm()` will generally be called with the same iteration and function count limits across different solvers. However, we may wish to compare the underlying codes with their own, rather particular, limits. Again `nlm()` provides an example when we call the Wood 4-parameter test function starting at `w0 <- c(-3,-1,-3,-1)`. The default `iterlim` value in `nlm()` is 100, but `ctrldefault(npar=4)` returns a value for `maxit` of 1000. This has consequences, as seen in the following example, which also shows how to call `nlm` through `optimr()` and `opm()` using the internal default.

```
require(optimx)
```

```
## Loading required package: optimx
```

```
npar <- 4
control<-list(maxit=NULL)
ctrl <- ctrldefault(npar)
ncontrol <- names(control)
nctrl <- names(ctrl)
for (onename in ncontrol) {
   if (onename %in% nctrl) {
     if (! is.null(control[onename]) || ! is.na(control[onename]) )
     ctrl[onename]<-control[onename]
   }
}
control <- ctrl # note the copy back! control now has a FULL set of values
print(control$maxit)
```

```
## NULL
```

```
control<-list()
ctrl <- ctrldefault(npar)
ncontrol <- names(control)
nctrl <- names(ctrl)
for (onename in ncontrol) {
   if (onename %in% nctrl) {
     if (! is.null(control[onename]) || ! is.na(control[onename]) )
     ctrl[onename]<-control[onename]
   }
}
control <- ctrl # note the copy back! control now has a FULL set of values
print(control$maxit)
```

```
## [1] 1000
```

Testing the package

There are examples and tests within the package. These include

- simple examples to illustrate the usage
- individual method tests
- problem tests over many methods

At some future opportunity, I hope to be able to document these tests more fully.

Needed tests or examples to provide proper coverage

For `optimr()`, which runs individual solvers, we need to run the following test combinations for those methods where the tests are appropriate:

- unconstrained
  - minimization
  - maximization
- bounds constrained
  - minimization
  - maximization
  - different ways of inputting bounds (single value, only lower, etc.)
- masks (fixed parameters)
  - plus bounds
  - no bounds
  - possibly with maximization

For `axsearch(), an example of use. Currently a small example in`tests/bdstest.R‘. ?? put example in Rd file.

For `bmchk()` ?? improve example in Rd file.

For `bmstep()` an example in the Rd file is needed. This is intended as an internal function that computes the maximum allowable step along a search direction (or returns Inf).

- checksolver.R
- ctrldefault.R
- fnchk.R
- gHgenb.R
- gHgen.R
- grback.R
- grcentral.R
- grchk.R
- grfwd.R
- grnd.R
- grpracma.R
- hesschk.R
- hjn.R
- kktchk.R
- multistart.R
- opm.R
- optchk.R
- optimr.R
- optimx.check.R
- optimx-package.R
- optimx.R
- optimx.run.R
- optimx.setup.R
- polyopt.R
- scalechk.R
- snewtonm.R
- snewton.R
- zzz.R

References

Nash, John C. 2014. *Nonlinear Parameter Optimization Using R Tools.* Book. John Wiley & Sons: Chichester. http://www.wiley.com//legacy/wileychi/nash/.

Nash, John C., and Ravi Varadhan. 2009. "Updating & Improving Optim(): Unifying Optimization Algorithms in R for Smooth, Nonlinear Problems." In *UseR!2009 Program.* http://www.r-project.org/conferences/useR-2009/abstracts/book-of-abstracts.pdf.

Nash, John C., and Mary Walker-Smith. 1987. *Nonlinear Parameter Estimation: An Integrated System in BASIC.* Book. New York: Marcel Dekker.