Intro to optimx

John C. Nash (nashjc at uottawa.ca)

2023-07-25

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 shows how it may be used to simplify the use of different solvers and to discover which ones may be suitable for particular types of problems.

This vignette was written to accompany the 2023 major overhaul of the package. However, it should be noted that the package is extensible and solvers and other features are being added regularly.

Background

Function minimization (or maximization), possibly constrained, is an important computational problem. The CRAN package optimx (John C. Nash and Varadhan (2011)) aims to provide a unified interface to optim(), nlm() and nlminb() from the base R distribution and a number of other solvers from CRAN packages or from optimx itself. The collection of solvers can be extended, as discussed in a companion vignette. There are also nonlinear least squares solvers nls() and some in CRAN packages such as nlsr and minpack.lm (John C Nash and Duncan Murdoch (2023), Elzhov et al. (2012)) that will not be discussed here except when there is overlap with particular matters at hand.

“Optimization”

The general nonlinear 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 solution of nonlinear equations overlaps many methods.

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), such features are likely inaccessible via this package without some customization of the code. Even masks are specified by bounds constraints where the upper and lower bounds coincide. This will generally “work” for methods that support bounds constraints, but a few methods explicitly recognize masks and avoid extra computational work. Where possible, the solvers supplied as part of optimx try to be efficient in this way.

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. The “SANN” method of optim(), a simulated annealing minimization tool, is excluded from the set of methods used by...
optimr() as it performs badly and simply runs for a specified number of function evaluations. That is, it does not necessarily return a solution.

Thus the solvers in optimx 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 conditions essentially require that a local minimum 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.

**Intent of package optimx**

The goals in preparing optimx are

- to provide a unified interface to allow the execution of many optimization solvers through a common syntax, thereby allowing the solver to be changed with the alteration of the method name
- to allow several solvers to be applied in a single call so that they may be compared in their performance on a single problem
- to provide for easier examination of different features of optimization solvers, for example, different gradient or Hessian approximations, or their use or omission. That is, to provide a framework for development of methods or programs.

Note that these goals point to different modes of use of package optimx.

Though unstated, a consideration throughout is to code the package and its components as much as possible in R and not call other languages. Given the large number of solvers that are built in other computing environments, this is aspirational rather than doctrinal.

**Unifying the syntax**

Because the syntax for optim() is relatively convenient and already spans five different multiparameter minimization methods (or solvers), this syntax has been adopted for optimx::optimr().

The optim() syntax is

```r
optim(par, fn, gr = NULL, ..., 
  method = "string_name_of_method", 
  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 specifies the solver to be used.

The benefits of a unified syntax are seen immediately if we examine how nlm() and nlminb() must be called, which we leave to the reader.

**Available solvers**

optimx provides access to a number of solvers. It is also relatively easy to extend, with almost all the work in the file optimr.R, though ctrldefault.R and NAMESPACE need modification too. The list of solvers available can be displayed as follows, in fact by testing to see if they are installed and available.

```r
library(optimx)
checkallsolvers()
```

```
## method BFGS ( BFGS ) from package stats is available
## method CG ( CG ) from package stats is available
## method Nelder-Mead ( Nelder-Mead ) from package stats is available
## method L-BFGS-B ( L-BFGS-B ) from package stats is available
```
## method nlm (nlm) from package stats is available
## method nlminb (nlminb) from package stats is available
## method lbfgsb3c (lbfgsb3c) from package lbfgsb3c is available
## method Rcgmin (Rcgmin) from package optimx is available
## method Rtnmin (Rtnmin) from package optimx is available
## method Rvmmin (Rvmmin) from package optimx is available
## method sneutron (snewton) from package optimx is available
## method sneutronm (snewtonm) from package optimx is available
## method spg (spg) from package BB is available
## method ucminf (ucminf) from package ucminf is available
## method newuoa (newuoa) from package minqa is available
## method bobyqa (bobyqa) from package minqa is available
## method uobyqa (uobyqa) from package minqa is available
## method nmkb (nmkb) from package dfoptim is available
## method hjkb (hjkb) from package dfoptim is available
## method hjn (hjn) from package optimx is available
## method lbfgs (lbfgs) from package lbfgs is available
## method subplex (subplex) from package subplex is available
## method ncv (ncg) from package optimx is available
## method nvm (nvm) from package optimx is available
## method mla (mla) from package marqLevAlg is available
## method slsqp (slsqp) from package nloptr is available
## method tnewt (tnewton) from package nloptr is available
## method anms (anms) from package pracma is available
## method pracmanm (nelder_mead) from package pracma is available
## method nlmm (neldermead) from package nloptr is available
## NULL

checkallsolvers returns a list of missing or uninstalled solvers, so a NULL tells us all are ready to use.

Note that solvers can generally be categorized by whether they use the function, the function and gradient,
or the function, gradient and hessian. Another nomenclature is “direct search”, “descent”, and “Newton-like”
methods.

Also note that many of the solvers available in optimx() are NOT very good, or are experimental, so replete
with weaknesses or errors. Part of the goal of the package is to allow for testing and comparison of solvers,
and to allow their improvement.

An example

```r
library(optimx)
sqmod <- function(z, xx) {
    nn <- length(z)
    yy <- xx^(1:nn)
    f <- sum((yy - z)^2)
    f
}
sqmod.g <- function(z, xx) {
    nn <- length(z)
    yy <- xx^(1:nn)
    gg <- 2*(z - yy)
}

z0 <- rep(0.1, 2) # initial parameters
sol <- optimr(par=z0, fn=sqmod, gr=sqmod.g, method="nvm", xx=1.5)
```
The function \texttt{proptimr()} is used to give a tidy output to the result of running \texttt{optimr()}. 

**Excluded [1D] solver**

There is also method “Brent” in \texttt{optim()} that is a one-parameter, or one-dimensional ([1D]), minimizer. Separately, base R has the one-parameter function \texttt{optimize()}. In the development of \texttt{optimx}, [1D] problems have not been a priority, though there is the illustrative script \texttt{onepar_test.R} in the inst/doc/legacy-demo directory of package \texttt{optimx}. At the time of writing, we advise the use of \texttt{optimize()}. 

**More than one method at a time**

The \texttt{method} argument to \texttt{optimr()} must actually be a \texttt{single} character string. An attempt to call two solvers by, say,

\begin{verbatim}
t2 <- optim(par=c(2,2), fn=fr, gr=grr, method=c("BFGS", "Nelder-Mead"))
\end{verbatim}

generates the message

\begin{verbatim}
Error in match.arg(method) : 'arg' must be of length 1
\end{verbatim}

In package \texttt{optimx}, the function \texttt{optimr()} takes one \texttt{method} at a time. However, the function \texttt{opm()} allows \texttt{method} to be a vector of character strings, and returns the results in a data frame, allowing easy comparison of different solvers when presented with a single problem. Moreover, there are special values that may be given to \texttt{method}, namely “ALL” and “MOST”. The former tries all currently available methods as listed in the returned object \texttt{allmeth} from function \texttt{ctrldefault()}, while the latter uses a more reasonable subset of these. Note that a call to \texttt{ctrldefault()} needs an integer argument that approximates the number of parameters in the objective function to be minimized.

\begin{verbatim}
solm <- opm(par=z0, fn=sqmod, gr=sqmod.g, method="MOST", xx=1.5)
\end{verbatim}

## Warning in opm(par = z0, fn = sqmod, gr = sqmod.g, method = "MOST", xx = 1.5): 
## 'snewtonm' removed from 'method' -- no hess()

\begin{verbatim}
cat("Result of applying opm() to ",attr(solm,"fname"))
\end{verbatim}

\begin{verbatim}
# Result of applying opm() to sqmod
print(summary(solm, order=value))
\end{verbatim}

<table>
<thead>
<tr>
<th>method</th>
<th>p1</th>
<th>s1</th>
<th>p2</th>
<th>s2</th>
<th>value</th>
<th>fevals</th>
<th>gevals</th>
<th>hevals</th>
<th>conv</th>
</tr>
</thead>
<tbody>
<tr>
<td>nlm</td>
<td>1.500000</td>
<td>2.250000</td>
<td>0.0000000e+00</td>
<td>6</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lbfgsb3c</td>
<td>1.500000</td>
<td>2.250000</td>
<td>0.0000000e+00</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rtnmin</td>
<td>1.500000</td>
<td>2.250000</td>
<td>0.0000000e+00</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spg</td>
<td>1.500000</td>
<td>2.250000</td>
<td>0.0000000e+00</td>
<td>21</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>subplex</td>
<td>1.500000</td>
<td>2.250000</td>
<td>0.0000000e+00</td>
<td>337</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ncg</td>
<td>1.500000</td>
<td>2.250000</td>
<td>0.0000000e+00</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>slsqp</td>
<td>1.500000</td>
<td>2.250000</td>
<td>0.0000000e+00</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tnewt</td>
<td>1.500000</td>
<td>2.250000</td>
<td>0.0000000e+00</td>
<td>5</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nlminb</td>
<td>1.500000</td>
<td>2.250000</td>
<td>4.930381e-32</td>
<td>5</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ucminf</td>
<td>1.500000</td>
<td>2.250000</td>
<td>4.930381e-32</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Rcgmin 1.500000 2.250000 2.465190e-31 4 2 0 0
## nvm 1.500000 2.250000 2.465190e-31 4 3 0 2
## Rvmmin 1.500000 2.250000 2.465190e-31 4 3 0 2
## pracmanm 1.500000 2.250000 4.435370e-28 199 0 0 0
## bobyqa 1.500000 2.250000 7.038638e-20 35 0 0 0
## mla 1.500000 2.250000 5.466920e-17 7 20 0 0
## nlmm 1.500001 2.250000 1.31518e-12 115 0 0 0
## Nelder-Mead 1.500161 2.249910 3.393324e-08 87 0 0 0
## nmkb 1.499130 2.250543 1.052278e-06 47 0 0 0

Note that the summary offers a way to order the results. Here we have chosen to order them from the smallest minimum found, so that the first result could be considered the best, though we have seen cases where the lowest solution value is for a set of parameters that is infeasible. We caution that timing results are generally unreliable when they are as small as these, and also we have only a single timing.

### Using named arguments

R allows a user to specify the arguments to a function like `optim` or, particularly, `optimr` by name, and to do so in any order. We recommend naming arguments, as using the position of arguments can, by very simple mistakes, get wrong answers. (We have done it ourselves!) One such error is to include the `gr` argument when the solver does not use gradients and the function is not defined. We do admin, however, positional arguments do allow for shorter calling statements.

An equally annoying error, however, is to use the WRONG argument name. For example, in one case it took me several hours to realize a very strange error was being caused by using `start=strt` rather than `par=strt` in calling the `snewtonm()` function. This may argue for calling the solvers via `optimr` as a way to avoid such naming errors.

### Examples of usage

There are a number of examples and tests in the directory `inst/doc/examples/`, as well as in the `tests` directory and included in the manual (.Rd) files for the functions in the `optimx` package. Most of these may be run via the script `runex.R` in this directory. The script `fhop.R` requires user input and also the non-CRAN package `funconstrain` from https://github.com/jlmelville/funconstrain. However, it provides a quite diverse set of well-established tests that include gradients and hessians in analytic form. The script
HessQuality.R uses the same examples and also requires user input to illustrate measures of quality for computed (possibly approximate) Hessians.

Additional resources

The directory inst/doc/legacy-demo/ contains the demonstration directory that was previously part of the package when the optimx() function was the primary (but now deprecated) tool. Currently, there is no demonstration directory, as I have found running the demo() takes too much keyboard entry for simple use. Instead the inst/doc/examples/ can be loaded and run. They can be executed line by line in Rstudio IDE (https://www.rstudio.com/categories/rstudio-ide/).

Developments of the optimx() function

optimx is now a mature package that has gone through several revisions since it was first set up in 2008. Users will note that there is much overlap between the original function optimx() (kept in the package for legacy uses) and the current opm(). The reasons for restructuring to two functions optimr() and opm() are as follows:

- optimx has a large set of features, some of which conflict, which increase 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, and these features are now deprecated.
- 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.

Running multiple methods – opm()

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. 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.

To allow for ease of use, opm() can take as its method argument the single character strings “ALL” or “MOST” (upper case). The first of these uses all available methods, the second a selection of “most” of the methods, in
both cases conditional on bounds and other characteristics of the problem. The use of “ALL” is disparaged, as there are a number of legacy methods that have been superceded by very similar solvers with (hopefully) superior performance. The set of solvers defined by “MOST” is intended to serve as a way to examine the performance of different classes of solver on a problem without excessive duplication.

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$maxfeval` and `control$maxit` 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.

My experience is that the hopes of users are generally much more optimistic than the eventual results of applying ideas like those in `polyopt()`.

```r
defnR <- function(x, gs=100.0) {
  n <- length(x)
  x1 <- x[2:n]
  x2 <- x[1:(n - 1)]
  sum(gs * (x1 - x2)^2 + (1 - x2)^2)
}
defgrR <- function(x, gs=100.0) {
  n <- length(x)
  g <- rep(NA, n)
  if (n > 2) {
    ii <- 2:(n - 1)
    g[ii] <- 2 * (x[ii] - 1) + 4 * gs * x[ii] * (x[ii]^2 - x[ii + 1]) + 2 * gs * (x[ii] - x[ii - 1]^2)
  }
  g[n] <- 2 * gs * (x[n] - x[n - 1]^2)
  g
}
defx0 <- rep(pi, 4)
defmc <- data.frame(method=c("Nelder-Mead","Rvmmin"), maxit=c(1000, 100), maxfeval= c(1000, 1000))
defans <- polyopt(x0, fnR, grR, methcontrol=mc, control=list(trace=0))
```
## Method 1: Nelder-Mead
## Method 2: Rvmmin

<table>
<thead>
<tr>
<th></th>
<th>p1</th>
<th>p2</th>
<th>p3</th>
<th>p4</th>
<th>value</th>
<th>fevals</th>
<th>gevals</th>
<th>convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.236467</td>
<td>1.530031</td>
<td>2.34111</td>
<td>5.484277</td>
<td>2.136779e+00</td>
<td>129</td>
<td>NA</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1.000000</td>
<td>1.000000</td>
<td>1.000000</td>
<td>1.000000</td>
<td>1.360662e-28</td>
<td>59</td>
<td>39</td>
<td>0</td>
</tr>
</tbody>
</table>

mc <- data.frame(method=c("Nelder-Mead","Rvmmin"), maxit=c(100, 100), maxfeval=c(100, 1000))

ans <- polyopt(x0, fnR, grR, methcontrol=mc, control=list(trace=0))

## Method 1: Nelder-Mead
## Method 2: Rvmmin

<table>
<thead>
<tr>
<th></th>
<th>p1</th>
<th>p2</th>
<th>p3</th>
<th>p4</th>
<th>value</th>
<th>fevals</th>
<th>gevals</th>
<th>convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.237657</td>
<td>1.530116</td>
<td>2.342114</td>
<td>5.484497</td>
<td>2.139230e+00</td>
<td>101</td>
<td>NA</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1.000000</td>
<td>1.000000</td>
<td>1.000000</td>
<td>1.000000</td>
<td>4.461994e-30</td>
<td>60</td>
<td>40</td>
<td>0</td>
</tr>
</tbody>
</table>

mc <- data.frame(method=c("Nelder-Mead","Rvmmin"), maxit=c(10, 100), maxfeval=c(10, 1000))

ans <- polyopt(x0, fnR, grR, methcontrol=mc, control=list(trace=0))

## Method 1: Nelder-Mead
## Method 2: Rvmmin

<table>
<thead>
<tr>
<th></th>
<th>p1</th>
<th>p2</th>
<th>p3</th>
<th>p4</th>
<th>value</th>
<th>fevals</th>
<th>gevals</th>
<th>convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.905973</td>
<td>2.631084</td>
<td>3.495022</td>
<td>3.495022</td>
<td>1.217125e+04</td>
<td>11</td>
<td>NA</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1.000000</td>
<td>1.000000</td>
<td>1.000000</td>
<td>1.000000</td>
<td>6.236932e-30</td>
<td>87</td>
<td>63</td>
<td>0</td>
</tr>
</tbody>
</table>

### 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. The function `multistart()` allows 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. Note that only one method can be specified in the call.

```r
# edited from inst/doc/examples/StyblinskiTang2308.R
require(optimx, quietly=TRUE)
fnStyblinskiTang <- function(x) 0.5 * sum(x^4 - 16*x^2 + 5*x)
grST <- function(x) {
  2*x^3 - 16*x + 2.5
}
n <- 4
cat("Globalmin ",n," = ",(39.16599)*length(x0),"\n")

## Globalmin 4 = -156.664
methlist<-c("hjn", "nvm", "Nelder-Mead")
stmat<matrix(runif(6*n, -5, 5), nrow=6, ncol=n)
cat("Starting parameters (rows of matrix):
")

## Starting parameters (rows of matrix):
```
```r
for (mstr in methlist){
  cat("Method ",mstr,":\n")
  mtst <- multistart(stmat, fnStyblinskiTang, grST, method=mstr)
  print(mtst)
}

# Try a single start
```
expedient.

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.

**Bounds**

A number of methods support the inclusion of box (or bounds) constraints. These are specified via the arguments `lower` and `upper` to either `opm()` or `optimr()`. It there are \( n \) parameters, these arguments may either be assigned a single value or else a vector of \( n \) elements. A single value is replicated in an \( n \)-vector.

**Masks** are bounds constraints where the upper and lower bound are equal, so essentially fixed parameters.

The function `optimx::bmchk()` has a rich set of tests for bounds and parameters. In particular, it returns information on whether the supplied parameters and bounds are

- **admissible** meaning no lower bound exceeds an upper bound
- **feasible** meaning all parameters are in the closed interval between the lower and upper bounds
- **onbound** to flag cases where a parameter is on an upper or lower bound, as some methods, `nmkb` in particular, fails if this is the case
- **maskadded** when upper and lower bounds are close together

Generally it is a poor idea to specify bounds that are too close together. For masks, make the lower and upper bounds equal, otherwise give the methods some space to adjust parameters, as most methods rely on various tolerances and heuristics. Moreover, approximations to derivatives using finite difference have to take steps along the parameters. I know of no generally available program that checks that finite difference steps do not violate bounds.

**Infeasible parameters**

If a user specifies infeasible parameters, `optimx::optimr()` will stop. However, there is an easy way to adjust initial parameters to be within bounds. This does, of course, assume that the bounds are specified correctly and the parameters are incorrect. Here is an example of how to make the adjustment.

```r
# set initial parameters in vector init, lower and upper bounds in lo and up
chkpar <- bmchk(init, lower=lo, upper=up)
if (chkpar$parchanged) init <- chkpar$bvec
```

**Display of solution status w.r.t. bounds**

A recent addition to the output summary is a set of status flags showing which parameters are on lower or upper bounds or are masked. The following example illustrates this. At the time of writing, I have not diagnosed why method `nmkb` does not indicate that the solution is on a lower bound. However, `nmkb` from the `dfoptim` package uses a transfinite transformation of the parameters to keep parameters within bounds, so the parameter returned may be very slightly different from the bound, so not getting flagged.

The example also suppresses a warning for a lack of symmetry in the estimated Hessian at solution in order to keep this presentation tidy. The warning is very common and I have yet to find a suitable test for asymmetry that works well across the range of different problems.

```r
XRosenbrock.f <- function (x)
{
  n <- length(x)
  x1 <- x[2:n]
  x2 <- x[1:(n - 1)]
  sum(100 * (x1 - x2^2)^2 + (1 - x2)^2)
}
XRosenbrock.g <- function (x)
```
```r
n <- length(x)
g <- rep(NA, n)
if (n > 2) {
  ii <- 2:(n - 1)
g[ii] <- 2 * (x[ii] - 1) + 400 * x[ii] * (x[ii]^2 - x[ii]) + 200 * (x[ii] - x[ii - 1]^2)
}
g[n] <- 200 * (x[n] - x[n - 1]^2)
g
}

lo <- (2:5)
up <- (4:7)
st <- 0.5*lo+up
brose <- optim(st, XRosenbrock.f, XRosenbrock.g, method="MOST", lower=lo, upper=up,
               control=list(dowarn=FALSE))

## Warning in optim(st, XRosenbrock.f, XRosenbrock.g, method = "MOST", lower = lo, :
## method requested does not handle bounds
## Warning in optim(st, XRosenbrock.f, XRosenbrock.g, method = "MOST", lower = lo, :
## 'snewtonm' removed from 'method' -- no hess()
# warnings turned off to avoid kktchk Hessian symmetry warning on most methods
summary(brose, order=TRUE)
```

```
## p1 s1 p2 s2 p3 s3 p4 s4 value fevals gevals hevals conv kkt1 kkt2
## nlminb   2 L   3   L   4 L   7 U    10714   5   3   0   0 FALSE TRUE
## lbfgsb3c 2 L   3   L   4 L   7 U    10714   3   3   0   0 FALSE TRUE
## Rtnmin   2 L   3   L   4 L   7 U    10714  13  13   0   0 FALSE TRUE
## bobyqa   2 L   3   L   4 L   7 U    10714  30  30   0   0 FALSE TRUE
## nmkb     2   3   L   4 L   7 U    10714  34  34   0   0 FALSE TRUE
## ncg      2 L   3   L   4 L   7 U    10714   9   8   0   0 FALSE TRUE
## Rcgmin   2 L   3   L   4 L   7 U    10714   9   8   0   0 FALSE TRUE
## nvm      2 L   3   L   4 L   7 U    10714  10  10   0   0 FALSE TRUE
## Rvmmin   2 L   3   L   4 L   7 U    10714  10  10   0   0 FALSE TRUE
## slsqp    2 L   3   L   4 L   7 U    10714   4   3   0   0 FALSE TRUE
## tnewt    2 L   3   L   4 L   7 U    10714   7   6   0   0 FALSE TRUE
## nlins    2 L   3   L   4 L   7 U    10714  67  67   0   2 FALSE TRUE
##
## xtime
## nlminb   0.018
## lbfgsb3c 0.001
## Rtnmin   0.003
## bobyqa   0.001
## nmkb     0.002
## ncg      0.001
## Rcgmin   0.001
## nvm      0.001
## Rvmmin   0.002
## slsqp    0.002
## tnewt    0.001
## nlins    0.002
```
Function and parameter 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. `optimr()` and `opm()` include a maximize control, and we would recommend avoiding `fnscale` use. Note that 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.

As general advice, I believe users should recast their problem as a minimization, either manually or with some pre-processing code, and then use post-solution processing to return the answer in a fashion relevant to the context. Unfortunately, there are many places where automated scaling by a negative number can create havoc.

Access to solver-specific 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.

In package `optimx` we have had to navigate such obstacles:

- I have found 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.
- The package attempts to provide a sane set of choices for the default values of controls. In many cases, `optimx` functions, particularly `opm()`, will NOT be able to use all the options available for some solvers, and users will need to access such functions directly.

In some cases, it is possible to access solver-specific controls, but we caution that this is still a developmental feature. We illustrate with an example.

Example of use of local control parameters

Since we have the source code to `optimr()` which is where all the solvers are called, it is possible to alter local control parameters. This is, however, not intended for general users. On the other hand, we do provide an illustration of how to make the modifications. The example is the `nelder_mead()` solver in package `pracma`. This is set by default to “converge” when the variance of the function values is no greater than `tol`, for which the default is 1.0e-8.

tol is, unfortunately, a common name, and is one of the `optimx` default controls. Thus we have allowed a special control named `pracmanmtol` for routine `optimr()`. Note that this is not intended to be applied to `opm()`. Special controls should be used for the solver for which they are appropriate. We believe that it is sensible to use a mechanism like that suggested here so that the solver’s original default settings are preserved, but can be changed by the user. `pracmanmtol` could also be added to `ctrldefault()` if it is generally important to use it. Below in the discussion of `ctrldefault()` further examples are referenced.

```r
library(optimx)
fnR <- function (x, gs=100.0)
{
    n <- length(x)
    x1 <- x[2:n]
    x2 <- x[1:(n - 1)]
```

12
```r
sum(gs * (x1 - x2^2)^2 + (1 - x2)^2)
}
x0 <- rep(pi, 4)
apnm0 <- optimr(x0, fnR, method="pracmanm")
proptimr(apnm0)
```

```
## Result apnm0 ( pracmanm -> (no_name) ) calc. min. = 2.4803e-06 at
## 0.99999 1.0001 1.0003 1.0006
## After 523 fn evals, and 0 gr evals and 0 hessian evals
## Termination code is 0 :
##
## apnm1 <- optimr(x0, fnR, method="pracmanm", control=list(pracmanmtol=1e-13))
```

```
## Warning in optimr(x0, fnR, method = "pracmanm", control = list(pracmanmtol =
## 1e-13)): Special controls present for optimr with method pracmanm
proptimr(apnm1)
```

```
## Result apnm1 ( pracmanm -> (no_name) ) calc. min. = 2.733e-14 at
## 1 1 1 1
## After 705 fn evals, and 0 gr evals and 0 hessian evals
## Termination code is 0 :
##
## Derivatives

Derivative information is used by many optimization methods. In particular, the \texttt{gradient} is the vector of first derivatives of the objective function and the \texttt{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 (true) derivative-free methods, we may also choose to employ numerical approximations for derivatives with gradient-based solvers. Some of the optimizers called by \texttt{optimr} automatically provide a numerical approximation if the gradient function (typically called \texttt{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. Generally, I advise explicitly setting the gradient function for methods that use this information, and \texttt{optimx} has tools to make this possible.

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 “gcentral” for forward and central derivative approximations. Direct codes for the gradient are included in the call by a name \texttt{NOT} in character form.

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

- as mentioned above, there may be conflicts with the \texttt{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 \texttt{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 several 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 the grad function 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 do so.

mymeth<="ncg"

# using fnR and grR from earlier chunk
system.time(sola <- optimr(par=x0, fn=fnR, gr=grR, method=mymeth))

## user system elapsed
## 0.001 0.000 0.002

proptimr(sola)

## Result sola ( ncg -> fnR ) calc. min. = 6.6888e-16 at
## 1 1 1 1
## After 181 fn evals, and 80 gr evals and 0 hessian evals
## Termination code is 0 : NA
##
## proptimr(solback)

system.time(solback <- optimr(par=x0, fn=fnR, gr="grpracma", method=mymeth))

## user system elapsed
## 0.016 0.000 0.017

proptimr(solback)
## Result solback ( ncg -> fnR ) calc. min. = 5.287e-09 at  
## 0.99999 0.99997 0.99994 0.99988  
## After 2243 fn evals, and 264 gr evals and 0 hessian evals  
## Termination code is 1 : NA  
##  
```r
system.time(solctl <- optimr(par=x0, fn=fnR, gr="grcentral", method=mymeth))
```

## user system elapsed  
## 0.004 0.000 0.004

**proptimr(solctl)**

## Result solctl ( ncg -> fnR ) calc. min. = 7.154e-15 at  
## 1 1 1 1  
## After 181 fn evals, and 80 gr evals and 0 hessian evals  
## Termination code is 0 : NA  
##  
```r
system.time(solnd <- optimr(par=x0, fn=fnR, gr="grnd", method=mymeth))
```

## user system elapsed  
## 0.024 0.000 0.024

**proptimr(solnd)**

## Result solnd ( ncg -> fnR ) calc. min. = 6.6535e-16 at  
## 1 1 1 1  
## After 181 fn evals, and 80 gr evals and 0 hessian evals  
## Termination code is 0 : NA  
##  
```r
system.time(solpra <- optimr(par=x0, fn=fnR, gr="grpracma", method=mymeth))
```

## user system elapsed  
## 0.003 0.000 0.003

**proptimr(solpra)**

## Result solpra ( ncg -> fnR ) calc. min. = 1.6956e-15 at  
## 1 1 1 1  
## After 133 fn evals, and 57 gr evals and 0 hessian evals  
## Termination code is 0 : NA  
##  
It is worth mentioning that the package **pracma** (Borchers (2017)) has functions **gradient** and **hessian**, and these routines can be substituted for the **numDeriv** versions. Within **optimr()** and **optim()** the gradient, for example, can use the quoted string “grpracma” to get the numeric gradient function of **pracma**, while specifying the control list elements **hesspkg="pracma"** or **jacpkg="pracma"** will substitute the appropriate Hessian or Jacobian approximations.

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

**Note** In 2018 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 changes in Julia functions have made it difficult to keep this capability operational.

Legacy optimx() function

The functions optimx(), optimx.setup(), optimx.check(), and optimx.run() are included in the package to allow for existing legacy uses. However, these functions will NOT be updated or improved, and any deficiencies that do not cause the package to be archived will NOT be corrected.

Illustrations of how optimx() is used are included in the files in directory inst/doc/legacy-demo/. Note that some of these examples have long run times. Moreover, the objective functions are such that some of the solvers fail to find a satisfactory optimum.

Solvers provided with the optimx package

Over time, I developed a number of solvers, particularly ones written entirely using the R language. Some of these were in separate packages, but it has made sense to consolidate them with optimx, and the individual packages are now deprecated. That is to say, it makes no sense to me to have separate packages, each of which needs to be maintained and coordinated with the others.

Moreover, the most recent solvers are intended to be called from the wrapper function optimr() so that common needs for checking of suitability of parameters, bounds and functions and derivatives can be shared in a single block of code. It makes no sense to call these new, streamlined programs independently.

Hooke and Jeeves method

A very simple pattern search method from Hooke and Jeeves (1961) is easily programmed. The function hjn() is intended to be a didactic tool and NOT an operational routine. Note that specifying method="hjkb" gets a more sophisticated version from the package dfoptim.

Newton-like methods

Newton-like methods are a traditional approach to optimization and solution of nonlinear equations. The idea is to find a stationary (i.e., flat) point on the function surface, which will be either a maximum, minimum or saddle point. There are also a number of interesting theorems that show the approach is in some way “best”. However, the assumptions that underly such theorems generally do not apply when we would like them to, and Newton-like methods, and in particular the classical approach, generally perform dismally on real-life problems.

This classic approach starts with a set of parameters $x_0$ for our objective function $f(x)$. We will evaluate the Hessian matrix of second partial derivatives at $x_0$ and call it $H$ and the gradient as the vector of first derivatives and call it $g$. Then we solve the Newton equations

$$H\delta = -g$$

and create

$$x_1 = x_0 + \delta$$

and repeat the iteration until, hopefully, we “converge” to a set of parameters where the gradient is (nearly) zero.
optimx includes two Newton-like solvers that try to use this classic algorithm but add some safeguards to avoid a few of the potential hazards. A major one of these is a singular Hessian, which prevents progress and generally results in program failure, sometimes with output that is difficult to interpret.

snewtm() is a streamlined version of the earlier snewtonm() that stabilizes the Newton equations following ideas of Levenberg (1944) and Marquardt (1963). It allows bounds to be specified, and is likely as good as safeguarded Newton methods get unless a lot of fancy tweaks are added.

snewton() uses a back-track line search along the direction δ rather than a full step. This is much less successful as a strategy, but serves as a didactic tool. Since the source code is provided, it can be used for experimentation. A streamlined version of this code has not been prepared as I have not found this approach very promising.

Note that some other solvers available to optimx from other packages use Newton-like ideas, in particular nlm().

A significant cost to using Newton-like methods is the generation of the Hessian. This requires a lot of human effort to code, with many opportunities for mistakes. It also is computationally costly in most situations. Approximations can be used, but these may multiply the computational cost.

Relevant files are snewtonm.R and snewton.R, as well as the vignette SNewton.Rmd (SNewton: safeguarded Newton methods for function minimization).

Conjugate gradient methods

The base R function optim() offers a method “CG” which is a translation of Algorithm 22 from John C. Nash (1979). (Actually, it is a semi-automated translation of the Pascal version of this from the Second edition, 1990.) I was never happy with this algorithm, and STRONGLY disparage its use. However, a very minor alteration, following some suggestions in Dai and Yuan (2001) yielded the function Rcgmin(). This has gone through a number of transformations, largely in an attempt to simplify the code rather than change the algorithm. The current RECOMMENDED version is the streamlined version ncg(), but previous versions are available in the package. Relevant files are ncg.R, Rcgmin.R, Rcgminb.R, and Rcgminu.R, with the last two files intended for separate application to bounds constrained or unconstrained problems.

Note that conjugate gradient methods are very lean on memory use and can be applied to functions of very large numbers of parameters. My experience is that the performance varies markedly with the test function and particular details of the internals of the method. “ncg” is quite good, but often is outperformed by “Rcgmin”. The differences are largely in the line search. “ncg” has more checks and safeguards, and combines bounded and unbounded codes. It could, however, likely be further improved.

Variable metric methods

The base R function optim() offers a method “BFGS” which is a translation of Algorithm 21 from John C. Nash (1979). (Actually, it is a semi-automated translation of the Pascal version of this from the Second edition, 1990.) This algorithm was developed from Fletcher (1970) when I visited Roger Fletcher in Dundee in January 1976 and we used a pencil to scratch out lines of Fortran code that were not critical to the method, as I needed to shoe-horn the code into an extremely small memory. It has proved exceptionally robust. However, the compiled code in base R does not allow for experimentation with variations on the algorithm, nor did it allow for bounds constraints. Rvmmin(), entirely in R, does have bounds constraints. More recently, I have endeavoured to simplify the code by merging bounded and unconstrained solver versions (Rvmminb() and Rvmminu()) into the streamlined code nvm(). Method nvm is now recommended, but the previous functions are still available, since they are used in a number of legacy applications. Relevant files are nvm.R, Rvmminb.R, Rvmmin.R, and Rvmminu.R.

Variable metric methods are very similar (some would say equivalent to) quasi-Newton methods. The approach is to approximate the Hessian with some inexpensive proxy. A large proportion of such approaches have been found to be very efficient and reliable. The particular approach of Fletcher (1970) starts with a unit matrix as the “approximate inverse Hessian”. This means the initial δ is a steepest descent direction.
Then at each iteration, the approximate inverse Hessian is updated by a quite inexpensive process, avoiding an actual solution of the matrix equations. Note, however, that there are myriad variants on these themes. The `optim()` choice “L-BFGS-B” is a particular and much tweaked routine (Byrd et al. (1995), Zhu et al. (1997)). In 2011, there was a revision by Morales and Nocedal (2011), and Fidler et al. (2018) is the result, which is available via `optimx` as solver `lbfgsb3c`. (It is NOT part of the source code of the package.) There are few cases where `lbfgsb3c` and L-BFGS-B get different answers.

**Truncated Newton methods**

In a similar fashion, **Truncated Newton** methods (sometimes also called **Inexact Newton methods**) try to simplify the solution of the Newton equations by solving them with a linear conjugate gradients approach. There are a number of implementations of such ideas. In `optimx`, the Matlab codes of my brother Stephen Nash (S. G. Nash (1983), S. G. Nash (2000)) have been translated to R for unconstrained (`tn()`) or bounds constrained (`tnbc()`) problems. Relevant files – the transformation from Matlab needed some mildly awkward coding – are Rtnmin-package.R, tnbc.R, tn.R and 3zz.R. To use the functions within `optimr()` or `opm()`, the solvers are specified simply as “Rtnmin” and the programs will select the internal function.

Of the descent methods, the truncated Newton codes in `optimx` are likely the ones most in need of review and rework, as they have been the least revised. Moreover, the codes were largely research programs rather than production ones. From the package `nloptr` the `tnewton()` function is available in `optimx` using the method name `tnewt`, and this performs quite well.

**Some other functions in the package**

`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 will be part of the .inst/doc/ collection of information in the package.

Note that calling `optimr()` with `control=list( )` containing controls that are NOT in the default set will try to cause these to be passed to the relevant solver. However, no special checks that these are valid is performed. CAUTION! Here is an example where we want to give the solver `minqa::bobyqa()` some parameters that alter the search strategy. A longer example is given in inst/doc/examples/specctrlhobbs.R.

```r
require(optimx)

hobbs.f <- function(x){ # # Hobbs weeds problem -- function
  if (abs(12*x[3]) > 500) { # check computability
    fbad <- .Machine$dmax
    return(fbad)
  }
  res <- hobbs.res(x)
  f <- sum(res * res)
}

hobbs.res <- function(x){ # Hobbs weeds problem -- residual
  # This variant uses looping
  if (length(x) != 3) stop("hobbs.res -- parameter vector n!=3")
  y <- c(5.308, 7.24, 9.638, 12.866, 17.069, 23.192, 31.443, 38.558, 50.156, 62.948,
```
t <- 1:12
if (abs(12*x[3]) > 50) {
    res <- rep(Inf, 12)
} else {
    res <- 100*x[1]/(1+10*x[2]*exp(-0.1*x[3]*t)) - y
}
}
x0 <- c(1, 1, 1)
cat("default rhobeg and rhoend in bobyqa\n")

## default rhobeg and rhoend in bobyqa
tdef <- optimr(x0, hobbs.f, method="bobyqa")
proptimr(tdef)

## Result tdef ( bobyqa -> (no_name) ) calc. min. = 2.5873 at
## 1.9619 4.9092 3.1357
## After 513 fn evals, and 0 gr evals and 0 hessian evals
## Termination code is 0 : Normal exit from bobyqa
##
## -------------------------------------------------

tsp3 <- optimr(x0, hobbs.f, method="bobyqa", control=list(rhobeg=10, rhoend=1e-4))

## Warning in optimr(x0, hobbs.f, method = "bobyqa", control = list(rhobeg = 10, :
## Special controls present for optimr with method bobyqa
proptimr(tsp3)

## Result tsp3 ( bobyqa -> (no_name) ) calc. min. = 8528.4 at
## 1.221 1 1.0115
## After 20 fn evals, and 0 gr evals and 0 hessian evals
## Termination code is 0 : Normal exit from bobyqa
##
## -------------------------------------------------

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. These have now been consolidated in the optimx package. The routines used are as
follows.

axsearch()

A crude and old-fashioned form of post-solution analysis is to take small steps from the proposed solution in
a positive and negative direction along each parameter axis and check if the function surface is symmetric
and consider its curvature.
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. A persistent issue with kktchk() is that it finds the returned Hessian matrix is very often asymmetric, which it should not be. The degree of asymmetry is rarely extreme, but so far a reasonable test has proved elusive.

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

These have been discussed above under Derivatives.

fchck(), grchk() and hesschk()

These functions are provided to allow for detection of user errors in supplied function, gradient or hessian functions.

fchck() 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 (e.g., from numDeriv) to check the derivative functions supplied by the user.

bmchck()

This routine (mentioned above) 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 John C. 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.

checkallsolvers()

Running this function (no argument is needed) will test if all required solvers are available in the current computing environment. This is provided as a tool to be used BEFORE trying to run opm() or optimr(). We used it to list the available solvers.

proptimr()

This little function is intended to provide compact output for reporting the results of running optimr(). Where available, the “status” flags are reported. These are an indicator for humans – “-”, “L”, “F”, “U”, “+”, “M”, “?”,”!” for out-of-bounds-low (-), lower bound (L), free (F or blank), upper bound (U), out-of-bounds-high (+), masked or fixed (M), unknown (?), and inadmissible (!). This function can be used on other solver outputs, but the “status” flags may be missing. Moreover, some solvers do not return information for some of the information proptimr() attempts to display. For example, the “nlm” solver (nlm() from base R) only returns “iterations” (likely computed gradient evaluations) but not function evaluations.
Where possible, `proptimr()` uses extra function, gradient and hessian count information generated by `optimr()`. Note that the `counts` vector of function and/or gradient or iteration counts is still returned by `optimr` but many methods return only partial information and none appear to give the hessian evaluation count. `opm()` also returns this information and `summary.opm()` displays this information.

References


