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.

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.

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

```
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. method specifies the solver to be used.

The benefits of a unified syntax are seen immediately if we look 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.

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

```
## Check if method  BFGS  from package  stats  is available
## Check if method  CG  from package  stats  is available
## Check if method  Nelder-Mead  from package  stats  is available
## Check if method  L-BFGS-B  from package  stats  is available
```

```
## Check if method  nlm  from package  stats  is available
## Check if method  nlminb  from package  stats  is available
## Check if method  lbfgsb3c  from package  lbfgsb3c  is available
## Check if method  Rcgmin  from package  optimx  is available
## Check if method  Rtnmin  from package  optimx  is available
## Check if method  Rvmmin  from package  optimx  is available
## Check if method  snewton  from package  optimx  is available
## Check if method  snewtonm  from package  optimx  is available
## Check if method  spg  from package  BB  is available
## Check if method  ucminf  from package  ucminf  is available
## Check if method  newuoa  from package  minqa  is available
## Check if method  bobyqa  from package  minqa  is available
## Check if method  uobyqa  from package  minqa  is available
## Check if method  nmkb  from package  dfoptim  is available
## Check if method  hjkb  from package  dfoptim  is available
## Check if method  hjn  from package  optimx  is available
## Check if method  lbfgs  from package  lbfgs  is available
## Check if method  subplex  from package  subplex  is available
## Check if method  ncg  from package  optimx  is available
## Check if method  nvm  from package  optimx  is available
## Check if method  mla  from package  marqLevAlg  is available
## Check if method  slsqp  from package  nloptr  is available
## Check if method  anms  from package  pracma  is available

## NULL
```

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

```
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)
```

```
## lower:[1] -Inf
## upper:[1] Inf
## par:[1] 0.1 0.1
```

```
proptimr(sol)
```

```
## Result  sol ( nvm  ->  (no_name) ) calc. min. = 2.46519e-31  at
## 1.5     2.25
## After  4  fn evals, and  3  gr evals and  0  hessian evals
## Termination code is  2 : NA
##
## ------------------------------------------------
```

The function `proptimr()` is used to give a tidy output to the result of running `optimr()`.

Extra [1D] solver

There is also method "Brent" in `optim()` that is a one-parameter, or one-dimensional ([1D]), minimizer. Separately, base R has the one-parameter function `optimize()`. In the development of `optimx`, [1D] problems have not been a priority, though there is the illustrative script `onepar_test.R` in the `demo` directory of package `optimx` and at the time of writing, we advise the use of `optimize()`.

More than one method at a time

The `method` argument must actually 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
```

In package `optimx`, the function `optimr()` takes one `method` at a time. However, the function `opm()` allows `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 `method`, namely "ALL" and "MOST". The former tries all currently available methods as listed in the returned object `allmeth` from function `ctrldefault()`, while the latter uses a more reasonable subset of these. Note that a call to `ctrldefault()` needs an integer argument that approximates the number of parameters in the objective function to be minimized.

```
solm <- opm(par=z0, fn=sqmod, gr=sqmod.g, method="MOST", xx=1.5)
```

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

```
## Warning in optimr(par, fn, gr, hess = hess, method = meth, lower = lower, :
## Successful convergence Restarts for stagnation =0
```

```
## lower:[1] -Inf
## upper:[1] Inf
## par:[1] 0.1 0.1
```

```
print(summary(solm, order=value))
```

```
##                  p1 s1       p2 s2        value fevals gevals hevals conv
## nlm         1.500000    2.250000    0.000000e+00      6      6      0    0
## lbfgsb3c    1.500000    2.250000    0.000000e+00      3      3      0    0
## Rtnmin      1.500000    2.250000    0.000000e+00      3      3      0    0
## spg         1.500000    2.250000    0.000000e+00     21      4      0    0
## subplex     1.500000    2.250000    0.000000e+00    337      0      0    0
## ncg         1.500000    2.250000    0.000000e+00      4      2      0    0
## slsqp       1.500000    2.250000    0.000000e+00      6      5      0    0
## nlminb      1.500000    2.250000    4.930381e-32      5      3      0    0
```

```
## ucminf      1.500000    2.250000    1.972152e-31      4      4      0    0
## 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
## bobyqa      1.500000    2.250000    7.038638e-20     35      0      0    0
## mla         1.500000    2.250000    5.466920e-17      7     20      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
##               kkt1 kkt2 xtime
## nlm           TRUE TRUE 0.001
## lbfgsb3c      TRUE TRUE 0.002
## Rtnmin        TRUE TRUE 0.002
## spg           TRUE TRUE 0.002
## subplex       TRUE TRUE 0.002
## ncg           TRUE TRUE 0.002
## slsqp         TRUE TRUE 0.004
## nlminb        TRUE TRUE 0.001
## ucminf        TRUE TRUE 0.001
## Rcgmin        TRUE TRUE 0.001
## nvm           TRUE TRUE 0.001
## Rvmmin        TRUE TRUE 0.002
## bobyqa        TRUE TRUE 0.001
## mla           TRUE TRUE 0.002
## Nelder-Mead   TRUE TRUE 0.002
## nmkb         FALSE TRUE 0.005
```

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 is considered the best. Timing results are generally unreliable when they are as small as these.

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!) 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.

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/).

"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), 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 in `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, possibly apart from the "SANN" method of `optim()`, which 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.

The message is thus that 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 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 single 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.

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 this function is now deprecated. See below for how such ideas are implemented in separate functions.
- 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.

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.

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.

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.

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 packages attempts to provide a sane set of choices for the default values of controls. In many cases, `optimx` functions will NOT be able to use all the options available for some solvers, and users will need to access such functions directly.

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()`.

```
fnR <- function (x, gs=100.0)
{
    n <- length(x)
    x1 <- x[2:n]
    x2 <- x[1:(n - 1)]
    sum(gs * (x1 - x2^2)^2 + (1 - x2)^2)
}
grR <- function (x, gs=100.0)
{
    n <- length(x)
    g <- rep(NA, n)
    g[1] <- 2 * (x[1] - 1) + 4*gs * x[1] * (x[1]^2 - x[2])
    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
}

x0 <- rep(pi, 4)
mc <- data.frame(method=c("Nelder-Mead","Rvmmin"), maxit=c(1000, 100), maxfeval= c(1000, 1000))

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

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

```
ans
```

```
##         p1       p2      p3       p4        value fevals gevals convergence
## 1 1.236467 1.530031 2.34111 5.484277 2.136779e+00    129     NA           0
## 2 1.000000 1.000000 1.00000 1.000000 1.249728e-28     59     39           0
```

```
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
```

```
ans
```

```
##         p1       p2       p3       p4        value fevals gevals convergence
## 1 1.237657 1.530116 2.342114 5.484497 2.139230e+00    101     NA           1
## 2 1.000000 1.000000 1.000000 1.000000 4.461994e-30     60     40           0
```

```
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
```

```
ans
```

```
##         p1       p2       p3       p4        value fevals gevals convergence
## 1 2.905973 2.631084 3.495022 3.495022 1.217125e+04     11     NA           1
## 2 1.000000 1.000000 1.000000 1.000000 6.236932e-30     87     63           0
```

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.

```
# We use already loaded fnR and grR from previous chunk
pm <- rbind(rep(1,4), rep(pi, 4), rep(-2,4), rep(0,4), rep(20,4))
pm <- as.matrix(pm)
cat("multistart matrix:\n")
```

```
## multistart matrix:
```

```
print(pm)
```

```
##           [,1]      [,2]      [,3]      [,4]
## [1,]  1.000000  1.000000  1.000000  1.000000
## [2,]  3.141593  3.141593  3.141593  3.141593
## [3,] -2.000000 -2.000000 -2.000000 -2.000000
## [4,]  0.000000  0.000000  0.000000  0.000000
## [5,] 20.000000 20.000000 20.000000 20.000000
```

```
ans <- multistart(pm, fnR, grR, method="Rvmmin", control=list(trace=0))
ans
```

```
##           p1        p2        p3       p4        value fevals gevals
## 1  1.0000000 1.0000000 1.0000000 1.000000 0.000000e+00      1      1
## 2 -0.7756592 0.6130934 0.3820629 0.145972 3.701429e+00     78     48
## 3  1.0000000 1.0000000 1.0000000 1.000000 1.533348e-29     77     57
## 4  1.0000000 1.0000000 1.0000000 1.000000 4.979684e-30     58     40
## 5  1.0000000 1.0000000 1.0000000 1.000000 0.000000e+00    199    145
##   convergence
## 1           2
## 2           0
## 3           0
## 4           0
## 5           2
```

Personally, I have not been happy with the outputs of such approaches in the few cases where I have applied

them. They are an inelegant approach to such problems, though in some cases may be an appropriate 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.

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 (true) derivative-free methods, we may also choose to employ numerical approximations for derivatives with gradient-based solvers. 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. Generally, I advise explicitly setting the gradient function for methods that use this information, and `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 "grcentral" for forward and central derivative approximations. Direct codes for the gradient are included in the call by a name NOT in character form.

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

- as mentioned above, 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 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
##   0.003   0.000   0.003
```

```
proptimr(sola)
```

```
## Result  sola ( ncg  ->  (no_name) ) calc. min. = 6.688787e-16  at
## 1     1     1     1
## After  181  fn evals, and  80  gr evals and  0  hessian evals
## Termination code is  0 : NA
##
## ------------------------------------------------
```

```
system.time(solfwd <- optimr(par=x0, fn=fnR, gr="grfwd", method=mymeth))
```

```
##    user  system elapsed
##   0.001   0.000   0.002
```

```
proptimr(solfwd)
```

```
## Result  solfwd ( ncg  ->  (no_name) ) calc. min. = 677.6654  at
## 1.79385     0.8793545     -0.3571361     0.05541653
## After  38  fn evals, and  6  gr evals and  0  hessian evals
## Termination code is  0 : NA
##
## ------------------------------------------------
```

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

```
##    user  system elapsed
##   0.017   0.000   0.017
```

```
proptimr(solback)
```

```
## Result  solback ( ncg  ->  (no_name) ) calc. min. = 5.287042e-09  at
## 0.9999856     0.9999705     0.9999394     0.9998773
## After  2243  fn evals, and  264  gr evals and  0  hessian evals
## Termination code is  1 : NA
##
## ------------------------------------------------
```

```
system.time(solctl <- optimr(par=x0, fn=fnR, gr="grcentral", method=mymeth))
```

```
##    user  system elapsed
##   0.003   0.000   0.004
```

```
proptimr(solctl)
```

```
## Result  solctl ( ncg  ->  (no_name) ) calc. min. = 7.153971e-15  at
## 1     1     1     1
## After  181  fn evals, and  80  gr evals and  0  hessian evals
## Termination code is  0 : NA
##
## ------------------------------------------------
```

```
system.time(solnd <- optimr(par=x0, fn=fnR, gr="grnd", method=mymeth))
```

```
##    user  system elapsed
##   0.022   0.000   0.023
```

```
proptimr(solnd)
```

```
## Result  solnd ( ncg  ->  (no_name) ) 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
##
## ------------------------------------------------
```

```
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  ->  (no_name) ) calc. min. = 1.695638e-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 `opm()` 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 faile 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 $x0$ for our objective function $f(x)$. We will evaluate the **Hessian** matrix of second partial derivatives at $x0$ 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

$$x1 = x0 + \delta$$

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 algoritm, 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 by 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 zzz.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

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 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.

```
require(optimx)
hobbs.f<- function(x){ # # Hobbs weeds problem -- function
    if (abs(12*x[3]) > 500) { # check computability
       fbad<-.Machine$double.xmax
       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,
         75.995, 91.972)
    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. = 2.587277  at
## 1.961864     4.909168     3.135697
## 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. = 8528.395  at
## 1.221039     1     1.011529
## 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. (Now in the `optimx` package.) The routines used in the current package 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 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 (e.g., from `numDeriv`) to check the derivative functions supplied by the user.

bmchk()

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, gradiend 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

Borchers, Hans Werner. 2017. *Pracma: Practical Numerical Math Functions.* https://CRAN.R-project.org/package=pracma.

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

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

Elzhov, Timur V., Katharine M. Mullen, Andrej-Nikolai Spiess, and Ben Bolker. 2012. *Minpack.lm: R Interface to the Levenberg-Marquardt Nonlinear Least-Squares Algorithm Found in MINPACK, Plus Support for Bounds.* R Project for Statistical Computing. http://CRAN.R-project.org/package=minpack.lm.

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

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

Hooke, Robert, and T. A. Jeeves. 1961. "'Direct Search' Solution of Numerical and Statistical Problems." *J. ACM* 8 (2): 212–29.

John C Nash, and Duncan Murdoch. 2023. *nlsr: Functions for Nonlinear Least Squares Solutions*.

Levenberg, Kenneth. 1944. "A Method for the Solution of Certain Non-Linear Problems in Least Squares." *Quarterly of Applied Mathematics* 2: 164--168.

Marquardt, Donald W. 1963. "An Algorithm for Least-Squares Estimation of Nonlinear Parameters." *SIAM Journal on Applied Mathematics* 11 (2): 431–41.

Morales, José Luis, and Jorge Nocedal. 2011. "Remark on Algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound constrained optimization." *ACM Trans. Math. Softw.* 38 (1): 7:1–4. http://doi.acm.org/10.1145/2049662.2049669.

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

Nash, John C, and Ravi Varadhan. 2011. Optimx*: A Replacement and Extension of the optim() Function*. Nash Information Services Inc.; Johns Hopkins University.

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

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

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

Zhu, Ciyou, Richard H. Byrd, Peihuang Lu, and Jorge Nocedal. 1997. "Algorithm 778: L-BFGS-b: Fortran Subroutines for Large-Scale Bound-Constrained Optimization." *ACM Trans. Math. Softw.* 23 (4): 550–60. https://doi.org/10.1145/279232.279236.