# Package ‘optextras’

December 20, 2019

**Version** 2019-12.4

**Date** 2019-12-04

**Title** Tools to Support Optimization Possibly with Bounds and Masks

**Maintainer** John C Nash <nashjc@uottawa.ca>

**Description** Tools to assist in safely applying user generated objective and derivative function to optimization programs. These are primarily function minimization methods with at most bounds and masks on the parameters. Provides a way to check the basic computation of objective functions that the user provides, along with proposed gradient and Hessian functions, as well as to wrap such functions to avoid failures when inadmissible parameters are provided. Check bounds and masks. Check scaling or optimality conditions. Perform an axial search to seek lower points on the objective function surface. Includes forward, central and backward gradient approximation codes.

**License** GPL-2

**Imports** numDeriv, utils

**NeedsCompilation** no

**Author** John C Nash [aut, cre]

**Repository** CRAN

**Date/Publication** 2019-12-20 13:20:07 UTC

## R topics documented:

---

`optextras-package` *Tools to Support Optimization Possibly with Bounds and Masks*

---

## Description

Provides tools that work with extensions of the optim() function to unify and streamline optimization capabilities in R for smooth, possibly box constrained functions of several or many parameters

There are three test functions, fnchk, grchk, and hesschk, to allow the user function to be tested for validity and correctness. However, no set of tests is exhaustive, and extensions and improvements are welcome. The package `numDeriv` is used for generation of numerical approximations to derivatives.

## Details

| | |
|---|---|
| Package: | optextras |
| Version: | 2012-6.18 |
| Date: | 2012-06-18 |
| License: | GPL-2 |
| Lazyload: | Yes |
| Depends: | numDeriv |
| Suggests: | BB, ucminf, Rcgmin, Rvmmin, minqa, setRNG, dfoptim |
| Repository: | R-Forge |
| Repository/R-Forge/Project: | optimizer |

Index:

```
axsearch     Perform an axial search optimality check
bmchk        Check bounds and masks for parameter constraints
bmstep       Compute the maximum step along a search direction.
fnchk        Test validity of user function
gHgen        Compute gradient and Hessian as a given
             set of parameters
gHgenb       Compute gradient and Hessian as a given
             set of parameters appying bounds and masks
grback       Backward numerical gradient approximation
grcentral    Central numerical gradient approximation
grchk        Check that gradient function evaluation
```

```
              matches numerical gradient
grfwd         Forward numerical gradient approximation
grnd          Gradient approximation using \code{numDeriv}
hesschk       Check that Hessian function evaluation
              matches numerical approximation
kktchk        Check the Karush-Kuhn-Tucker optimality conditions
optsp         An environment to hold some globally useful items
              used by optimization programs
scalechk      Check scale of initial parameters and bounds
```

## Author(s)

John C Nash <nashjc@uottawa.ca> and Ravi Varadhan <RVaradhan@jhmi.edu>

Maintainer: John C Nash <nashjc@uottawa.ca>

## References

Nash, John C. and Varadhan, Ravi (2011) Unifying Optimization Algorithms to Aid Software System Users: optimx for R, Journal of Statistical Software, publication pending.

## See Also

optim

---

# axsearch — *Perform axial search around a supposed minimum and provide diagnostics*

---

## Description

Nonlinear optimization problems often terminate at points in the parameter space that are not satisfactory optima. This routine conducts an axial search, stepping forward and backward along each parameter and computing the objective function. This allows us to compute the `tilt` and `radius of curvature` or `roc` along that parameter axis.

`axsearch` assumes that one is MINIMIZING the function `fn`. While we believe that it will work using the wrapper `ufn` from this package with the 'maximize=TRUE' setting, we believe it is much safer to write your own function that is to be minimized. That is minimize (-1)*(function to be maximized). All discussion here is in terms of minimization.

Axial search may find parameters with a function value lower than that at the supposed minimum, i.e., lower than `fmin`.

In this case `axsearch` exits immediately with the new function value and parameters. This can be used to restart an optimizer, as in the optimx wrapper.

## Usage

```
axsearch(par, fn=NULL, fmin=NULL, lower=NULL, upper=NULL, bdmsk=NULL, trace=0, ...)
```

## Arguments

- `par` A numeric vector of values of the optimization function parameters that are at a supposed minimum.
- `fn` The user objective function
- `fmin` The value of the objective function at the parameters `par`. ?? what if fmin==NULL?
- `lower` A vector of lower bounds on the parameters.
- `upper` A vector of upper bounds on the parameters.
- `bdmsk` An indicator vector, having 1 for each parameter that is "free" or unconstrained, and 0 for any parameter that is fixed or MASKED for the duration of the optimization. Partly for historical reasons, we use the same array during the progress of optimization as an indicator that a parameter is at a lower bound (bdmsk element set to -3) or upper bound (-1).
- `trace` If trace>0, then local output is enabled.
- `...` Extra arguments for the user function.

## Details

The axial search MAY give a lower function value, in which case, one can restart. Its primary use is in presenting some features of the function surface in the tilt and radius of curvature measures returned. However, better measures should be possible, and this function should be regarded as largely experimental.

## Value

A list with components:

- `bestfn` The lowest (best) function value found (??maximize??) during the axial search, else the original fmin value. (This is actively set in that case.)
- `par` The vector of parameters at the best function value.
- `details` A data frame reporting the original parameters, the forward step and backward step function values, the size of the step taken for a particular parameter, the tilt and the roc (radius of curvature). Some elements will be NA if we find a lower function value during the axial search.

## Examples

```
#####################
# require(optimx)
require(optextras)
# Simple bounds test for n=4
bt.f<-function(x){
  sum(x*x)
}

bt.g<-function(x){
  gg<-2.0*x
}
```

```
n<-4
lower<-rep(0,n)
upper<-lower # to get arrays set
bdmsk<-rep(1,n)
# bdmsk[(trunc(n/2)+1)]<-0
for (i in 1:n) {
  lower[i]<-1.0*(i-1)*(n-1)/n
  upper[i]<-1.0*i*(n+1)/n
}
xx<-0.5*(lower+upper)

cat("lower bounds:")
print(lower)
cat("start:       ")
print(xx)
cat("upper bounds:")
print(upper)

abtrvm <- list() # ensure we have the structure

cat("Rvmmin \n\n")
# Note: trace set to 0 below. Change as needed to view progress.

# Following can be executed if package optimx available
# abtrvm <- optimr(xx, bt.f, bt.g, lower=lower, upper=upper, method="Rvmmin",
#                  control=list(trace=0))
# Note: use lower=lower etc. because there is a missing hess= argument
# print(abtrvm)

abtrvm$par <- c(0.00, 0.75, 1.50, 2.25)
abtrvm$value <- 7.875
cat("Axial search")
axabtrvm <- axsearch(abtrvm$par, fn=bt.f, fmin=abtrvm$value, lower, upper, bdmsk=NULL,
                     trace=0)
print(axabtrvm)

abtrvm1 <- list() # set up structure
# Following can be executed if package optimx available
# cat("Now force an early stop\n")
# abtrvm1 <- optimr(xx, bt.f, bt.g, lower=lower, upper=upper, method="Rvmmin",
#                   control=list(maxit=1, trace=0))
# print(abtrvm1)

abtrvm1$value <- 8.884958
abtrvm1$par <- c(0.625, 1.625, 2.625, 3.625)

cat("Axial search")
axabtrvm1 <- axsearch(abtrvm1$par, fn=bt.f, fmin=abtrvm1$value, lower, upper, bdmsk=NULL,
                      trace=0)
print(axabtrvm1)

cat("Do NOT try axsearch() with maximize\n")
```

| bmchk | *Check bounds and masks for parameter constraints used in nonlinear optimization* |
|---|---|

## Description

Nonlinear optimization problems often have explicit or implicit upper and lower bounds on the parameters of the function to be miminized or maximized. These are called bounds or box constraints. Some of the parameters may be fixed for a given problem or for a temporary trial. These fixed, or masked, paramters are held at one value during a specific 'run' of the optimization.

It is possible that the bounds are inadmissible, that is, that at least one lower bound exceeds an upper bound. In this case we set the flag `admissible` to FALSE.

Parameters that are outside the bounds are moved to the nearest bound and the flag `parchanged` is set TRUE. However, we DO NOT change masked parameters, and they may be outside the bounds. This is an implementation choice, since it may be useful to test objective functions at point outside the bounds.

The package bmchk is essentially a test of the R function bmchk(), which is likely to be incorporated within optimization codes.

## Usage

```
bmchk(par, lower=NULL, upper=NULL, bdmsk=NULL, trace=0, tol=NULL, shift2bound=TRUE)
```

## Arguments

| | |
|---|---|
| `par` | A numeric vector of starting values of the optimization function parameters. |
| `lower` | A vector of lower bounds on the parameters. |
| `upper` | A vector of upper bounds on the parameters. |
| `bdmsk` | An indicator vector, having 1 for each parameter that is "free" or unconstrained, and 0 for any parameter that is fixed or MASKED for the duration of the optimization. Partly for historical reasons, we use the same array during the progress of optimization as an indicator that a parameter is at a lower bound (bdmsk element set to -3) or upper bound (-1). |
| `trace` | An integer that controls whether diagnostic information is displayed. A positive value displays information, 0 (default) does not. |
| `tol` | If provided, is used to detect a MASK, that is, lower=upper for some parameter. |
| `shift2bound` | If TRUE, non-masked paramters outside bounds are adjusted to the nearest bound. We then set parchanged = TRUE which implies the original parameters were infeasible. |

## Details

The bmchk function will check that the bounds exist and are admissible, that is, that there are no lower bounds that exceed upper bounds.

There is a check if lower and upper bounds are very close together, in which case a mask is imposed and maskadded is set TRUE. NOTE: it is generally a VERY BAD IDEA to have bounds close together in optimization, but here we use a tolerance based on the double precision machine epsilon. Thus it is not a good idea to rely on bmchk() to test if bounds constraints are well-posed.

## Value

A list with components:

| | |
|---|---|
| `bvec` | The vector of parameters, possibly adjusted for bounds. Parameters outside bounds are adjusted to the nearest bound. |
| `bdmsk` | adjusted input masks |
| `bchar` | indicator for humans – "-","L","F","U","+","M" for out-of-bounds-low, lower bound, free, upper bound, out-of-bounds-high, masked (fixed) |
| `lower` | (adjusted) lower bounds. If upper-lower<tol, we create a mask rather than leave bounds. In this case we could eliminate the bounds. At the moment, this change is NOT made, but a commented line of code is present in the file `bmchk.R`. |
| `upper` | (adjusted) upper bounds |
| `nolower` | TRUE if no lower bounds, FALSE otherwise |
| `noupper` | TRUE if no upper bounds, FALSE otherwise |
| `bounds` | TRUE if there are any bounds, FALSE otherwise |
| `admissible` | TRUE if bounds are admissible, FALSE otherwise This means no lower bound exceeds an upper bound. That is the bounds themselves are sensible. This condition has nothing to do with the starting parameters. |
| `maskadded` | TRUE when a mask has been added because bounds are very close or equal, FALSE otherwise. See the code for the implementation. |
| `parchanged` | TRUE if parameters are changed by bounds, FALSE otherswise. Note that parchanged = TRUE implies the input parameter values were infeasible, that is, violated the bounds constraints. |
| `feasible` | TRUE if parameters are within or on bounds, FALSE otherswise. |
| `onbound` | TRUE if any parameter is on a bound, FALSE otherswise. Note that parchanged = TRUE implies onbound = TRUE, but this is not used inside the function. This output value may be important, for example, in using the optimization function `nmkb` from package `dfoptim`. |

## Examples

```
#####################

cat("25-dimensional box constrained function\n")
flb <- function(x)
    { p <- length(x); sum(c(1, rep(4, p-1)) * (x - c(1, x[-p])^2)^2) }
```

```
start<-rep(2, 25)
cat("\n start:")
print(start)
lo<-rep(2,25)
cat("\n lo:")
print(lo)
hi<-rep(4,25)
cat("\n hi:")
print(hi)
bt<-bmchk(start, lower=lo, upper=hi, trace=1)
print(bt)
```

---

bmstep *Compute the maximum step along a search direction.*

---

### Description

Nonlinear optimization problems often have explicit or implicit upper and lower bounds on the parameters of the function to be miminized or maximized. These are called bounds or box constraints. Some of the parameters may be fixed for a given problem or for a temporary trial. These fixed, or masked, paramters are held at one value during a specific 'run' of the optimization.

The bmstep() function computes the maximum step possible (which could be infinite) along a particular search direction from current parameters to bounds.

### Usage

```
bmstep(par, srchdirn, lower=NULL, upper=NULL, bdmsk=NULL, trace=0)
```

### Arguments

| | |
|---|---|
| par | A numeric vector of starting values of the optimization function parameters. |
| srchdirn | A numeric vector giving the search direction. |
| lower | A vector of lower bounds on the parameters. |
| upper | A vector of upper bounds on the parameters. |
| bdmsk | An indicator vector, having 1 for each parameter that is "free" or unconstrained, and 0 for any parameter that is fixed or MASKED for the duration of the optimization. Partly for historical reasons, we use the same array during the progress of optimization as an indicator that a parameter is at a lower bound (bdmsk element set to -3) or upper bound (-1). |
| trace | An integer that controls whether diagnostic information is displayed. A positive value displays information, 0 (default) does not. |

### Details

The bmstep function will compute and return (as a double or Inf) the maximum step to the bounds.

## Value

A double precision value or Inf giving the maximum step to the bounds.

## Examples

```
#####################
xx <- c(1, 1)
lo <- c(0, 0)
up <- c(100, 40)
sdir <- c(4,1)
bm <- c(1,1) # both free
ans <- bmstep(xx, sdir, lo, up, bm, trace=1)
# stepsize
print(ans)
# distance
print(ans*sdir)
# New parameters
print(xx+ans*sdir)
```

---

`ctrldefault` *set control defaults*

---

## Description

Set control defaults.

## Usage

```
ctrldefault(npar)

dispdefault(ctrl)
```

## Arguments

| | |
|---|---|
| `npar` | Number of parameters to optimize. |
| `ctrl` | A list (likely generated by ‘ctrldefault‘) of default settings to ‘optimx‘. |

## Value

`ctrldefault` returns the default control settings for optimization tools.

`dispdefault` provides a compact display of the contents of a control settings list.

fnchk *Run tests, where possible, on user objective function*

## Description

fnchk checks a user-provided R function, ffn.

## Usage

```
fnchk(xpar, ffn, trace=0, ... )
```

## Arguments

| | |
|---|---|
| xpar | the (double) vector of parameters to the objective funcion |
| ffn | a user-provided function to compute the objective function |
| trace | set >0 to provide output from fnchk to the console, 0 otherwise |
| ... | optional arguments passed to the objective function. |

## Details

fnchk attempts to discover various errors in function setup in user-supplied functions primarily intended for use in optimization calculations. There are always more conditions that could be tested!

## Value

The output is a list consisting of list(fval=fval, infeasible=infeasible, excode=excode, msg=msg)

fval
: The calculated value of the function at parameters xpar if the function can be evaluated.

infeasible
: FALSE if the function can be evaluated, TRUE if not.

excode
: An exit code, which has a relationship to

msg
: A text string giving information about the result of the function check: Messages and the corresponding values of excode are:
  - fnchk OK; excode = 0; infeasible = FALSE
  - Function returns INADMISSIBLE; excode = -1; infeasible = TRUE
  - Function returns a vector not a scalar; excode = -4; infeasible = TRUE
  - Function returns a list not a scalar; excode = -4; infeasible = TRUE
  - Function returns a matrix list not a scalar; excode = -4; infeasible = TRUE
  - Function returns an array not a scalar; excode = -4; infeasible = TRUE
  - Function returned not length 1, despite not vector, matrix or array; excode = -4; infeasible = TRUE
  - Function returned non-numeric value; excode = 0; excode = -1; infeasible = TRUE
  - Function returned Inf or NA (non-computable); excode = -1; infeasible = TRUE

## Author(s)

John C. Nash <nashjc@uottawa.ca>

## Examples

```
# Want to illustrate each case.
# Ben Bolker idea for a function that is NOT scalar
# rm(list=ls())
# library(optimx)
sessionInfo()
benbad<-function(x, y){
  # y may be provided with different structures
  f<-(x-y)^2
} # very simple, but ...

y<-1:10
x<-c(1)
cat(”fc01: test benbad() with y=1:10, x=c(1)\n”)
fc01<-fnchk(x, benbad, trace=4, y)
print(fc01)

y<-as.vector(y)
cat(”fc02: test benbad() with y=as.vector(1:10), x=c(1)\n”)
fc02<-fnchk(x, benbad, trace=1, y)
print(fc02)

y<-as.matrix(y)
cat(”fc03: test benbad() with y=as.matrix(1:10), x=c(1)\n”)
fc03<-fnchk(x, benbad, trace=1, y)
print(fc03)

y<-as.array(y)
cat(”fc04: test benbad() with y=as.array(1:10), x=c(1)\n”)
fc04<-fnchk(x, benbad, trace=1, y)
print(fc04)

y<-”This is a string”
cat(”test benbad() with y a string, x=c(1)\n”)
fc05<-fnchk(x, benbad, trace=1, y)
print(fc05)

cat(”fnchk with Rosenbrock\n”)
fr <- function(x) {   ## Rosenbrock Banana function
  x1 <- x[1]
  x2 <- x[2]
  100 * (x2 - x1 * x1)^2 + (1 - x1)^2
}
xtrad<-c(-1.2,1)
ros1<-fnchk(xtrad, fr, trace=1)
print(ros1)
npar<-2
opros<-list2env(list(fn=fr, gr=NULL, hess=NULL, MAXIMIZE=FALSE, PARSCALE=rep(1,npar), FNSCALE=1,
```

```
                    KFN=0, KGR=0, KHESS=0, dots=NULL))
uros1<-fnchk(xtrad, fr, trace=1)
print(uros1)
```

gHgen *Generate gradient and Hessian for a function at given parameters.*

## Description

gHgen is used to generate the gradient and Hessian of an objective function used for optimization. If a user-provided gradient function gr is available it is used to compute the gradient, otherwise package numDeriv is used. If a user-provided Hessian function hess is available, it is used to compute a Hessian. Otherwise, if gr is available, we use the function jacobian() from package numDeriv to compute the Hessian. In both these cases we check for symmetry of the Hessian. Computational Hessians are commonly NOT symmetric. If only the objective function fn is provided, then the Hessian is approximated with the function hessian from package numDeriv which guarantees a symmetric matrix.

## Usage

```
gHgen(par, fn, gr=NULL, hess=NULL,
    control=list(ktrace=0), ...)
```

## Arguments

| | |
|---|---|
| par | Set of parameters, assumed to be at a minimum of the function fn. |
| fn | Name of the objective function. |
| gr | (Optional) function to compute the gradient of the objective function. If present, we use the Jacobian of the gradient as the Hessian and avoid one layer of numerical approximation to the Hessian. |
| hess | (Optional) function to compute the Hessian of the objective function. This is rarely available, but is included for completeness. |
| control | A list of controls to the function. Currently asymptol (default of 1.0e-7 which tests for asymmetry of Hessian approximation (see code for details of the test); ktrace, a logical flag which, if TRUE, monitors the progress of gHgen (default FALSE), and stoponerror, defaulting to FALSE to NOT stop when there is an error or asymmetry of Hessian. Set TRUE to stop. |
| ... | Extra data needed to compute the function, gradient and Hessian. |

## Details

None

## Value

`ansout` a list of four items,

- `gn` The approximation to the gradient vector.
- `Hn` The approximation to the Hessian matrix.
- `gradOK` TRUE if the gradient has been computed acceptably. FALSE otherwise.
- `hessOK` TRUE if the gradient has been computed acceptably and passes the symmetry test. FALSE otherwise.
- `nbm` Always 0. The number of active bounds and masks. Present to make function consistent with `gHgenb`.

## Examples

```
# genrose function code
genrose.f<- function(x, gs=NULL){ # objective function
## One generalization of the Rosenbrock banana valley function (n parameters)
n <- length(x)
        if(is.null(gs)) { gs=100.0 }
fval<-1.0 + sum (gs*(x[1:(n-1)]^2 - x[2:n])^2 + (x[2:n] - 1)^2)
        return(fval)
}

genrose.g <- function(x, gs=NULL){
# vectorized gradient for genrose.f
# Ravi Varadhan 2009-04-03
n <- length(x)
        if(is.null(gs)) { gs=100.0 }
gg <- as.vector(rep(0, n))
tn <- 2:n
tn1 <- tn - 1
z1 <- x[tn] - x[tn1]^2
z2 <- 1 - x[tn]
gg[tn] <- 2 * (gs * z1 - z2)
gg[tn1] <- gg[tn1] - 4 * gs * x[tn1] * z1
return(gg)
}

genrose.h <- function(x, gs=NULL) { ## compute Hessian
   if(is.null(gs)) { gs=100.0 }
n <- length(x)
hh<-matrix(rep(0, n*n),n,n)
for (i in 2:n) {
z1<-x[i]-x[i-1]*x[i-1]
# z2<-1.0-x[i]
                hh[i,i]<-hh[i,i]+2.0*(gs+1.0)
                hh[i-1,i-1]<-hh[i-1,i-1]-4.0*gs*z1-4.0*gs*x[i-1]*(-2.0*x[i-1])
                hh[i,i-1]<-hh[i,i-1]-4.0*gs*x[i-1]
                hh[i-1,i]<-hh[i-1,i]-4.0*gs*x[i-1]
}
        return(hh)
```

```
}

trad<-c(-1.2,1)
ans100fgh<-  gHgen(trad, genrose.f, gr=genrose.g, hess=genrose.h,
      control=list(ktrace=1))
print(ans100fgh)
ans100fg<-  gHgen(trad, genrose.f, gr=genrose.g,
      control=list(ktrace=1))
print(ans100fg)
ans100f<-  gHgen(trad, genrose.f, control=list(ktrace=1))
print(ans100f)
ans10fgh<-   gHgen(trad, genrose.f, gr=genrose.g, hess=genrose.h,
      control=list(ktrace=1), gs=10)
print(ans10fgh)
ans10fg<-   gHgen(trad, genrose.f, gr=genrose.g,
      control=list(ktrace=1), gs=10)
print(ans10fg)
ans10f<-   gHgen(trad, genrose.f, control=list(ktrace=1), gs=10)
print(ans10f)
```

---

gHgenb — *Generate gradient and Hessian for a function at given parameters.*

---

## Description

gHgenb is used to generate the gradient and Hessian of an objective function used for optimization. If a user-provided gradient function `gr` is available it is used to compute the gradient, otherwise package `numDeriv` is used. If a user-provided Hessian function `hess` is available, it is used to compute a Hessian. Otherwise, if `gr` is available, we use the function `jacobian()` from package `numDeriv` to compute the Hessian. In both these cases we check for symmetry of the Hessian. Computational Hessians are commonly NOT symmetric. If only the objective function `fn` is provided, then the Hessian is approximated with the function `hessian` from package `numDeriv` which guarantees a symmetric matrix.

## Usage

```
gHgenb(par, fn, gr=NULL, hess=NULL, bdmsk=NULL, lower=NULL, upper=NULL,
    control=list(ktrace=0), ...)
```

## Arguments

| | |
|---|---|
| `par` | Set of parameters, assumed to be at a minimum of the function `fn`. |
| `fn` | Name of the objective function. |
| `gr` | (Optional) function to compute the gradient of the objective function. If present, we use the Jacobian of the gradient as the Hessian and avoid one layer of numerical approximation to the Hessian. |

| | |
|---|---|
| hess | (Optional) function to compute the Hessian of the objective function. This is rarely available, but is included for completeness. |
| bdmsk | An integer vector of the same length as par. When an element of this vector is 0, the corresponding parameter value is fixed (masked) during an optimization. Non-zero values indicate a parameter is free (1), at a lower bound (-3) or at an upper bound (-1), but this routine only uses 0 values. |
| lower | Lower bounds for parameters in par. |
| upper | Upper bounds for parameters in par. |
| control | A list of controls to the function. Currently asymptol (default of 1.0e-7 which tests for asymmetry of Hessian approximation (see code for details of the test); ktrace, a logical flag which, if TRUE, monitors the progress of gHgenb (default FALSE), and stoponerror, defaulting to FALSE to NOT stop when there is an error or asymmetry of Hessian. Set TRUE to stop. |
| ... | Extra data needed to compute the function, gradient and Hessian. |

## Details

None

## Value

ansout a list of four items,

- gn The approximation to the gradient vector.
- Hn The approximation to the Hessian matrix.
- gradOK TRUE if the gradient has been computed acceptably. FALSE otherwise.
- hessOK TRUE if the gradient has been computed acceptably and passes the symmetry test. FALSE otherwise.
- nbm The number of active bounds and masks.

## Examples

```
require(numDeriv)
# genrose function code
genrose.f<- function(x, gs=NULL){ # objective function
## One generalization of the Rosenbrock banana valley function (n parameters)
n <- length(x)
        if(is.null(gs)) { gs=100.0 }
fval<-1.0 + sum (gs*(x[1:(n-1)]^2 - x[2:n])^2 + (x[2:n] - 1)^2)
        return(fval)
}

genrose.g <- function(x, gs=NULL){
# vectorized gradient for genrose.f
# Ravi Varadhan 2009-04-03
n <- length(x)
        if(is.null(gs)) { gs=100.0 }
gg <- as.vector(rep(0, n))
```

```
tn <- 2:n
tn1 <- tn - 1
z1 <- x[tn] - x[tn1]^2
z2 <- 1 - x[tn]
gg[tn] <- 2 * (gs * z1 - z2)
gg[tn1] <- gg[tn1] - 4 * gs * x[tn1] * z1
return(gg)
}

genrose.h <- function(x, gs=NULL) { ## compute Hessian
   if(is.null(gs)) { gs=100.0 }
n <- length(x)
hh<-matrix(rep(0, n*n),n,n)
for (i in 2:n) {
z1<-x[i]-x[i-1]*x[i-1]
z2<-1.0-x[i]
                hh[i,i]<-hh[i,i]+2.0*(gs+1.0)
                hh[i-1,i-1]<-hh[i-1,i-1]-4.0*gs*z1-4.0*gs*x[i-1]*(-2.0*x[i-1])
                hh[i,i-1]<-hh[i,i-1]-4.0*gs*x[i-1]
                hh[i-1,i]<-hh[i-1,i]-4.0*gs*x[i-1]
}
        return(hh)
}


maxfn<-function(x, top=10) {
       n<-length(x)
ss<-seq(1,n)
f<-top-(crossprod(x-ss))^2
f<-as.numeric(f)
return(f)
}

negmaxfn<-function(x) {
f<-(-1)*maxfn(x)
return(f)
}

parx<-rep(1,4)
lower<-rep(-10,4)
upper<-rep(10,4)
bdmsk<-c(1,1,0,1) # masked parameter 3
fval<-genrose.f(parx)
gval<-genrose.g(parx)
Ahess<-genrose.h(parx)
gennog<-gHgenb(parx,genrose.f)
cat("results of gHgenb for genrose without gradient code at ")
print(parx)
print(gennog)
cat("compare to g =")
print(gval)
cat("and Hess\n")
print(Ahess)
```

```
cat("\n\n")
geng<-gHgenb(parx,genrose.f,genrose.g)
cat("results of gHgenb for genrose at ")
print(parx)
print(gennog)
cat("compare to g =")
print(gval)
cat("and Hess\n")
print(Ahess)
cat("****************************************\n")
parx<-rep(0.9,4)
fval<-genrose.f(parx)
gval<-genrose.g(parx)
Ahess<-genrose.h(parx)
gennog<-gHgenb(parx,genrose.f,control=list(ktrace=TRUE), gs=9.4)
cat("results of gHgenb with gs=",9.4," for genrose without gradient code at ")
print(parx)
print(gennog)
cat("compare to g =")
print(gval)
cat("and Hess\n")
print(Ahess)
cat("\n\n")
geng<-gHgenb(parx,genrose.f,genrose.g, control=list(ktrace=TRUE))
cat("results of gHgenb for genrose at ")
print(parx)
print(gennog)
cat("compare to g =")
print(gval)
cat("and Hess\n")
print(Ahess)
gst<-5
cat("\n\nTest with full calling sequence and gs=",gst,"\n")
gengall<-gHgenb(parx,genrose.f,genrose.g,genrose.h, control=list(ktrace=TRUE),gs=gst)
print(gengall)


top<-25
x0<-rep(2,4)
cat("\n\nTest for maximization and top=",top,"\n")
cat("Gradient and Hessian will have sign inverted")
maxt<-gHgen(x0, maxfn, control=list(ktrace=TRUE), top=top)
print(maxt)

cat("test against negmaxfn\n")
gneg <- grad(negmaxfn, x0)
Hneg<-hessian(negmaxfn, x0)
# gdiff<-max(abs(gneg-maxt$gn))/max(abs(maxt$gn))
# Hdiff<-max(abs(Hneg-maxt$Hn))/max(abs(maxt$Hn))
# explicitly change sign
gdiff<-max(abs(gneg-(-1)*maxt$gn))/max(abs(maxt$gn))
Hdiff<-max(abs(Hneg-(-1)*maxt$Hn))/max(abs(maxt$Hn))
cat("gdiff = ",gdiff,"  Hdiff=",Hdiff,"\n")
```

## grback — *Backward difference numerical gradient approximation.*

### Description

`grback` computes the backward difference approximation to the gradient of user function `userfn`.

### Usage

```
grback(par, userfn, fbase=NULL, env=optsp, ...)
```

### Arguments

| | |
|---|---|
| `par` | parameters to the user objective function userfn |
| `userfn` | User-supplied objective function |
| `fbase` | The value of the function at the parameters, else NULL. This is to save recomputing the function at this point. |
| `env` | Environment for scratchpad items (like `deps` for approximation control in this routine). Default `optsp`. |
| `...` | optional arguments passed to the objective function. |

### Details

| | |
|---|---|
| Package: | grback |
| Depends: | R (>= 2.6.1) |
| License: | GPL Version 2. |

### Value

`grback` returns a single vector object `df` which approximates the gradient of userfn at the parameters par. The approximation is controlled by a global value `optderiveps` that is set when the package is attached.

### Author(s)

John C. Nash

## Examples

```
cat("Example of use of grback\n")

myfn<-function(xx, shift=100){
    ii<-1:length(xx)
    result<-shift+sum(xx^ii)
}

xx<-c(1,2,3,4)
ii<-1:length(xx)
print(xx)
gn<-grback(xx,myfn, shift=0)
print(gn)
ga<-ii*xx^(ii-1)
cat("compare to analytic gradient:\n")
print(ga)

cat("change the step parameter to 1e-4\n")
optsp$deps <- 1e-4
gn2<-grback(xx,myfn, shift=0)
print(gn2)
```

---

`grcentral` *Central difference numerical gradient approximation.*

---

## Description

`grcentral` computes the central difference approximation to the gradient of user function `userfn`.

## Usage

```
grcentral(par, userfn, fbase=NULL, env=optsp, ...)
```

## Arguments

| | |
|---|---|
| `par` | parameters to the user objective function userfn |
| `userfn` | User-supplied objective function |
| `fbase` | The value of the function at the parameters, else NULL. This is to save recomputing the function at this point. |
| `env` | Environment for scratchpad items (like `deps` for approximation control in this routine). Default `optsp`. |
| `...` | optional arguments passed to the objective function. |

## Details

| | |
|---|---|
| Package: | grcentral |
| Depends: | R (>= 2.6.1) |
| License: | GPL Version 2. |

## Value

`grcentral` returns a single vector object `df` which approximates the gradient of userfn at the parameters par. The approximation is controlled by a global value `optderiveps` that is set when the package is attached.

## Author(s)

John C. Nash

## Examples

```
cat("Example of use of grcentral\n")

myfn<-function(xx, shift=100){
    ii<-1:length(xx)
    result<-shift+sum(xx^ii)
}
xx<-c(1,2,3,4)
ii<-1:length(xx)
print(xx)
gn<-grcentral(xx,myfn, shift=0)
print(gn)
ga<-ii*xx^(ii-1)
cat("compare to\n")
print(ga)
```

---

`grchk` *Run tests, where possible, on user objective function and (optionally) gradient and hessian*

---

## Description

`grchk` checks a user-provided R function, `ffn`.

## Usage

```
grchk(xpar, ffn, ggr, trace=0, testtol=(.Machine$double.eps)^(1/3), ...)
```

### Arguments

| | |
|---|---|
| xpar | parameters to the user objective and gradient functions ffn and ggr |
| ffn | User-supplied objective function |
| ggr | User-supplied gradient function |
| trace | set >0 to provide output from grchk to the console, 0 otherwise |
| testtol | tolerance for equality tests |
| ... | optional arguments passed to the objective function. |

### Details

| | |
|---|---|
| Package: | grchk |
| Depends: | R (>= 2.6.1) |
| License: | GPL Version 2. |

`numDeriv` is used to numerically approximate the gradient of function `ffn` and compare this to the result of function `ggr`.

### Value

`grchk` returns a single object `gradOK` which is true if the differences between analytic and approximated gradient are small as measured by the tolerance `testtol`.

This has attributes "ga" and "gn" for the analytic and numerically approximated gradients.

At the time of preparation, there are no checks for validity of the gradient code in `ggr` as in the function `fnchk`.

### Author(s)

John C. Nash

### Examples

```
# Would like examples of success and failure. What about "near misses"??
cat("Show how grchk works\n")
require(optextras)
require(numDeriv)
# require(optimx)

jones<-function(xx){
  x<-xx[1]
  y<-xx[2]
  ff<-sin(x*x/2 - y*y/4)*cos(2*x-exp(y))
  ff<- -ff
}

jonesg <- function(xx) {
  x<-xx[1]
```

```
  y<-xx[2]
  gx <-  cos(x * x/2 - y * y/4) * ((x + x)/2) * cos(2 * x - exp(y)) -
    sin(x * x/2 - y * y/4) * (sin(2 * x - exp(y)) * 2)
  gy <- sin(x * x/2 - y * y/4) * (sin(2 * x - exp(y)) * exp(y)) - cos(x *
             x/2 - y * y/4) * ((y + y)/4) * cos(2 * x - exp(y))
  gg <- - c(gx, gy)
}

jonesg2 <- function(xx) {
  gx <- 1
  gy <- 2
  gg <- - c(gx, gy)
}


xx <- c(1, 2)

gcans <- grchk(xx, jones, jonesg, trace=1, testtol=(.Machine$double.eps)^(1/3))
gcans

gcans2 <- grchk(xx, jones, jonesg2, trace=1, testtol=(.Machine$double.eps)^(1/3))
gcans2
```

grfwd *Forward difference numerical gradient approximation.*

## Description

grfwd computes the forward difference approximation to the gradient of user function userfn.

## Usage

```
grfwd(par, userfn, fbase=NULL, env=optsp, ...)
```

## Arguments

| | |
|---|---|
| par | parameters to the user objective function userfn |
| userfn | User-supplied objective function |
| fbase | The value of the function at the parameters, else NULL. This is to save recomputing the function at this point. |
| env | Environment for scratchpad items (like deps for approximation control in this routine). Default optsp. |
| ... | optional arguments passed to the objective function. |

## Details

| | |
|---|---|
| Package: | grfwd |
| Depends: | R (>= 2.6.1) |
| License: | GPL Version 2. |

## Value

`grfwd` returns a single vector object `df` which approximates the gradient of userfn at the parameters par. The approximation is controlled by a global value `optderiveps` that is set when the package is attached.

## Author(s)

John C. Nash

## Examples

```
cat("Example of use of grfwd\n")

myfn<-function(xx, shift=100){
    ii<-1:length(xx)
    result<-shift+sum(xx^ii)
}
xx<-c(1,2,3,4)
ii<-1:length(xx)
print(xx)
gn<-grfwd(xx,myfn, shift=0)
print(gn)
ga<-ii*xx^(ii-1)
cat("compare to\n")
print(ga)
```

---

`grnd` *A reorganization of the call to numDeriv grad() function.*

---

## Description

Provides a wrapper for the numDeriv approximation to the gradient of a user supplied objective function `userfn`.

## Usage

```
grnd(par, userfn, ...)
```

### Arguments

| | |
|---|---|
| `par` | A vector of parameters to the user-supplied function `fn` |
| `userfn` | A user-supplied function |
| `...` | Other data needed to evaluate the user function. |

### Details

The Richardson method is used in this routine.

### Value

`grnd` returns an approximation to the gradient of the function userfn

### Examples

```
cat("Example of use of grnd\n")
require(numDeriv)
myfn<-function(xx, shift=100){
    ii<-1:length(xx)
    result<-shift+sum(xx^ii)
}
xx<-c(1,2,3,4)
ii<-1:length(xx)
print(xx)
gn<-grnd(xx,myfn, shift=0)
print(gn)
ga<-ii*xx^(ii-1)
cat("compare to\n")
print(ga)
```

---

## `hesschk` *Run tests, where possible, on user objective function and (optionally) gradient and hessian*

---

### Description

`hesschk` checks a user-provided R function, `ffn`.

### Usage

```
hesschk(xpar, ffn, ggr, hhess, trace=0, testtol=(.Machine$double.eps)^(1/3), ...)
```

## Arguments

| | |
|---|---|
| `xpar` | parameters to the user objective and gradient functions ffn and ggr |
| `ffn` | User-supplied objective function |
| `ggr` | User-supplied gradient function |
| `hhess` | User-supplied Hessian function |
| `trace` | set >0 to provide output from hesschk to the console, 0 otherwise |
| `testtol` | tolerance for equality tests |
| `...` | optional arguments passed to the objective function. |

## Details

| | |
|---|---|
| Package: | hesschk |
| Depends: | R (>= 2.6.1) |
| License: | GPL Version 2. |

`numDeriv` is used to compute a numerical approximation to the Hessian matrix. If there is no analytic gradient, then the `hessian()` function from `numDeriv` is applied to the user function `ffn`. Otherwise, the `jacobian()` function of `numDeriv` is applied to the `ggr` function so that only one level of differencing is used.

## Value

The function returns a single object `hessOK` which is TRUE if the analytic Hessian code returns a Hessian matrix that is "close" to the numerical approximation obtained via `numDeriv`; FALSE otherwise.

`hessOK` is returned with the following attributes:

- "nullhess"Set TRUE if the user does not supply a function to compute the Hessian.
- "asym"Set TRUE if the Hessian does not satisfy symmetry conditions to within a tolerance. See the `hesschk` for details.
- "ha"The analytic Hessian computed at paramters `xpar` using `hhess`.
- "hn"The numerical approximation to the Hessian computed at paramters `xpar`.
- "msg"A text comment on the outcome of the tests.

## Author(s)

John C. Nash

## Examples

```
# genrose function code
genrose.f<- function(x, gs=NULL){ # objective function
## One generalization of the Rosenbrock banana valley function (n parameters)
n <- length(x)
```

```
        if(is.null(gs)) { gs=100.0 }
fval<-1.0 + sum (gs*(x[1:(n-1)]^2 - x[2:n])^2 + (x[2:n] - 1)^2)
        return(fval)
}

genrose.g <- function(x, gs=NULL){
# vectorized gradient for genrose.f
# Ravi Varadhan 2009-04-03
n <- length(x)
        if(is.null(gs)) { gs=100.0 }
gg <- as.vector(rep(0, n))
tn <- 2:n
tn1 <- tn - 1
z1 <- x[tn] - x[tn1]^2
z2 <- 1 - x[tn]
gg[tn] <- 2 * (gs * z1 - z2)
gg[tn1] <- gg[tn1] - 4 * gs * x[tn1] * z1
return(gg)
}

genrose.h <- function(x, gs=NULL) { ## compute Hessian
   if(is.null(gs)) { gs=100.0 }
n <- length(x)
hh<-matrix(rep(0, n*n),n,n)
for (i in 2:n) {
z1<-x[i]-x[i-1]*x[i-1]
# z2<-1.0-x[i]
                hh[i,i]<-hh[i,i]+2.0*(gs+1.0)
                hh[i-1,i-1]<-hh[i-1,i-1]-4.0*gs*z1-4.0*gs*x[i-1]*(-2.0*x[i-1])
                hh[i,i-1]<-hh[i,i-1]-4.0*gs*x[i-1]
                hh[i-1,i]<-hh[i-1,i]-4.0*gs*x[i-1]
}
        return(hh)
}

trad<-c(-1.2,1)
ans100<-hesschk(trad, genrose.f, genrose.g, genrose.h, trace=1)
print(ans100)
ans10<-hesschk(trad, genrose.f, genrose.g, genrose.h, trace=1, gs=10)
print(ans10)
```

---

kktchk *Check Kuhn Karush Tucker conditions for a supposed function minimum*

---

## Description

Provide a check on Kuhn-Karush-Tucker conditions based on quantities already computed. Some of these used only for reporting.

## Usage

```
kktchk(par, fn, gr, hess=NULL, upper=NULL, lower=NULL,
            maximize=FALSE, control=list(), ...)
```

## Arguments

| | |
|---|---|
| `par` | A vector of values for the parameters which are supposedly optimal. |
| `fn` | The objective function |
| `gr` | The gradient function |
| `hess` | The Hessian function |
| `upper` | Upper bounds on the parameters |
| `lower` | Lower bounds on the parameters |
| `maximize` | Logical TRUE if function is being maximized. Default FALSE. |
| `control` | A list of controls for the function |
| `...` | The dot arguments needed for evaluating the function and gradient and hessian |

## Details

kktchk computes the gradient and Hessian measures for BOTH unconstrained and bounds (and masks) constrained parameters, but the kkt measures are evaluated only for the constrained case.

## Value

The output is a list consisting of

| | |
|---|---|
| `gmax` | The absolute value of the largest gradient component in magnitude. |
| `evratio` | The ratio of the smallest to largest Hessian eigenvalue. Note that this may be negative. |
| `kkt1` | A logical value that is TRUE if we consider the first (i.e., gradient) KKT condition to be satisfied. WARNING: The decision is dependent on tolerances and scaling that may be inappropriate for some problems. |
| `kkt2` | A logical value that is TRUE if we consider the second (i.e., positive definite Hessian) KKT condition to be satisfied. WARNING: The decision is dependent on tolerances and scaling that may be inappropriate for some problems. |
| `hev` | The calculated hessian eigenvalues, sorted largest to smallest?? |
| `ngatend` | The computed (unconstrained) gradient at the solution parameters. |
| `nnatend` | The computed (unconstrained) hessian at the solution parameters. |

## See Also

`optim`

## Examples

```
cat("Show how kktc works\n")

# require(optimx)
require(optextras)

jones<-function(xx){
  x<-xx[1]
  y<-xx[2]
  ff<-sin(x*x/2 - y*y/4)*cos(2*x-exp(y))
  ff<- -ff
}

jonesg <- function(xx) {
  x<-xx[1]
  y<-xx[2]
  gx <-  cos(x * x/2 - y * y/4) * ((x + x)/2) * cos(2 * x - exp(y)) -
    sin(x * x/2 - y * y/4) * (sin(2 * x - exp(y)) * 2)
  gy <- sin(x * x/2 - y * y/4) * (sin(2 * x - exp(y)) * exp(y)) - cos(x *
             x/2 - y * y/4) * ((y + y)/4) * cos(2 * x - exp(y))
  gg <- - c(gx, gy)
}

ans <- list() # to ensure structure available
# If optimx package available, the following can be run.
# xx<-0.5*c(pi,pi)
# ans <- optimr(xx, jones, jonesg, method="Rvmmin")
# ans

ans$par <- c(3.154083, -3.689620)

kkans <- kktchk(ans$par, jones, jonesg)
kkans
```

---

`scalechk` *Check the scale of the initial parameters and bounds input to an optimization code used in nonlinear optimization*

---

## Description

Nonlinear optimization problems often have different scale for different parameters. This function is intended to explore the differences in scale. It is, however, an imperfect and heuristic tool, and could be improved.

At this time scalechk does NOT take account of masks. (?? should 110702)

## Usage

```
scalechk(par, lower = lower, upper = upper, bdmsk=NULL, dowarn = TRUE)
```

## Arguments

| | |
|---|---|
| `par` | A numeric vector of starting values of the optimization function parameters. |
| `lower` | A vector of lower bounds on the parameters. |
| `upper` | A vector of upper bounds on the parameters. |
| `bdmsk` | An indicator vector, having 1 for each parameter that is "free" or unconstrained, and 0 for any parameter that is fixed or MASKED for the duration of the optimization. |
| `dowarn` | Set TRUE to issue warnings. Othwerwise this is a silent routine. Default TRUE. |

## Details

The scalechk function will check that the bounds exist and are admissible, that is, that there are no lower bounds that exceed upper bounds.

NOTE: Free paramters outside bounds are adjusted to the nearest bound. We then set parchanged = TRUE which implies the original parameters were infeasible.

There is a check if lower and upper bounds are very close together, in which case a mask is imposed and maskadded is set TRUE. NOTE: it is generally a VERY BAD IDEA to have bounds close together in optimization, but here we use a tolerance based on the double precision machine epsilon. Thus it is not a good idea to rely on scalechk() to test if bounds constraints are well-posed.

## Value

A list with components:

# Returns: # list(lpratio, lbratio) – the log of the ratio of largest to smallest parameters # and bounds intervals (upper-lower) in absolute value (ignoring Inf, NULL, NA)

| | |
|---|---|
| `lpratio` | The log of the ratio of largest to smallest parameters in absolute value (ignoring Inf, NULL, NA) |
| `lbration` | The log of the ratio of largest to smallest bounds intervals (upper-lower) in absolute value (ignoring Inf, NULL, NA) |

## Examples

```
#####################
  par <- c(-1.2, 1)
  lower <- c(-2, 0)
  upper <- c(100000, 10)
  srat<-scalechk(par, lower, upper,dowarn=TRUE)
  print(srat)
  sratv<-c(srat$lpratio, srat$lbratio)
  if (max(sratv,na.rm=TRUE) > 3) { # scaletol from ctrldefault in optimx
     warnstr<-"Parameters or bounds appear to have different scalings.\n
     This can cause poor performance in optimization. \n
```

```
    It is important for derivative free methods like BOBYQA, UOBYQA, NEWUOA."
    cat(warnstr,"\n")
}
```

# Index