Package 'nloptr'

June 25, 2024

Type Package

Title R Interface to NLopt

Version 2.1.1

Description Solve optimization problems using an R interface to NLopt. NLopt is a free/open-source library for nonlinear optimization, providing a common interface for a number of different free optimization routines available online as well as original implementations of various other algorithms. See <https://nlopt.readthedocs.io/en/latest/NLopt_Algorithms/> for more information on the available algorithms. Building from included sources requires 'CMake'. On Linux and 'macOS', if a suitable system build of NLopt (2.7.0 or later) is found, it is used; otherwise, it is built from included sources via 'CMake'. On Windows, NLopt is obtained through 'rwinlib' for 'R <= 4.1.x' or grabbed from the appropriate toolchain for 'R >= 4.2.0'.

License LGPL (>= 3)

SystemRequirements cmake (>= 3.2.0) which is used only on Linux or macOS systems when no system build of nlopt (>= 2.7.0) can be found.

Biarch true

Encoding UTF-8

RoxygenNote 7.3.1

Suggests knitr, rmarkdown, covr, tinytest

VignetteBuilder knitr

URL https://github.com/astamm/nloptr, https://astamm.github.io/nloptr/

BugReports https://github.com/astamm/nloptr/issues

NeedsCompilation yes

Author Jelmer Ypma [aut],
Steven G. Johnson [aut] (author of the NLopt C library),
Aymeric Stamm [ctb, cre] (<https://orcid.org/0000-0002-8725-3654>),
Hans W. Borchers [ctb],
Dirk Eddelbuettel [ctb],

Brian Ripley [ctb] (build process on multiple OS),
Kurt Hornik [ctb] (build process on multiple OS),
Julien Chiquet [ctb],
Avraham Adler [ctb] (<https://orcid.org/0000-0002-3039-0703>),
Xiongtao Dai [ctb],
Jeroen Ooms [ctb]

Maintainer Aymeric Stamm <aymeric.stamm@cnrs.fr>

Repository CRAN

Date/Publication 2024-06-25 10:00:02 UTC

Contents

auglag *Augmented Lagrangian Algorithm*

Description

The Augmented Lagrangian method adds additional terms to the unconstrained objective function, designed to emulate a Lagrangian multiplier.

Usage

```
auglag(
  x0,
  fn,
  gr = NULL,
  lower = NULL,
  upper = NULL,
  hin = NULL,
  hinjac = NULL,
  heq = NULL,
  heqjac = NULL,
  localsolver = "COBYLA",
  localtol = 1e-06,
  ineq2local = FALSE,
  nl.info = FALSE,
  control = list(),
  deprecatedBehavior = TRUE,
  ...
)
```

Arguments

x0	starting point for searching the optimum.
fn	objective function that is to be minimized.
gr	gradient of the objective function; will be provided provided is NULL and the solver requires derivatives.
lower, upper	lower and upper bound constraints.
hin, hinjac	defines the inequality constraints, hin(x) >= 0
heq, heqjac	defines the equality constraints, heq(x) = 0.
localsolver	available local solvers: COBYLA, LBFGS, MMA, or SLSQP.
localtol	tolerance applied in the selected local solver.
ineq2local	logical; shall the inequality constraints be treated by the local solver?; not possible at the moment.
nl.info	logical; shall the original NLopt info been shown.
control	list of options, see nl.opts for help.

`deprecatedBehavior`
: logical; if `TRUE` (default for now), the old behavior of the Jacobian function is used, where the equality is ≥ 0 instead of ≤ 0. This will be reversed in a future release and eventually removed.

`...`
: additional arguments passed to the function.

Details

This method combines the objective function and the nonlinear inequality/equality constraints (if any) in to a single function: essentially, the objective plus a 'penalty' for any violated constraints.

This modified objective function is then passed to another optimization algorithm with no nonlinear constraints. If the constraints are violated by the solution of this sub-problem, then the size of the penalties is increased and the process is repeated; eventually, the process must converge to the desired solution (if it exists).

Since all of the actual optimization is performed in this subsidiary optimizer, the subsidiary algorithm that you specify determines whether the optimization is gradient-based or derivative-free.

The local solvers available at the moment are `COBYLA'' (for the derivative-free approach)` and LBFGS", `MMA'', or` SLSQP" (for smooth functions). The tolerance for the local solver has to be provided.

There is a variant that only uses penalty functions for equality constraints while inequality constraints are passed through to the subsidiary algorithm to be handled directly; in this case, the subsidiary algorithm must handle inequality constraints. (At the moment, this variant has been turned off because of problems with the NLOPT library.)

Value

List with components:

`par`
: the optimal solution found so far.

`value`
: the function value corresponding to `par`.

`iter`
: number of (outer) iterations, see `maxeval`.

`global_solver`
: the global NLOPT solver used.

`local_solver`
: the local NLOPT solver used, LBFGS or COBYLA.

`convergence`
: integer code indicating successful completion (> 0) or a possible error number (< 0).

`message`
: character string produced by NLopt and giving additional information.

Note

Birgin and Martinez provide their own free implementation of the method as part of the TANGO project; other implementations can be found in semi-free packages like LANCELOT.

Author(s)

Hans W. Borchers

References

Andrew R. Conn, Nicholas I. M. Gould, and Philippe L. Toint, "A globally convergent augmented Lagrangian algorithm for optimization with general constraints and simple bounds," SIAM J. Numer. Anal. vol. 28, no. 2, p. 545-572 (1991).

E. G. Birgin and J. M. Martinez, "Improving ultimate convergence of an augmented Lagrangian method," Optimization Methods and Software vol. 23, no. 2, p. 177-195 (2008).

See Also

`alabama::auglag`, `Rsolnp::solnp`

Examples

```
x0 <- c(1, 1)
fn <- function(x) (x[1] - 2) ^ 2 + (x[2] - 1) ^ 2
hin <- function(x) 0.25 * x[1]^2 + x[2] ^ 2 - 1  # hin <= 0
heq <- function(x) x[1] - 2 * x[2] + 1           # heq = 0
gr <- function(x) nl.grad(x, fn)
hinjac <- function(x) nl.jacobian(x, hin)
heqjac <- function(x) nl.jacobian(x, heq)

# with COBYLA
auglag(x0, fn, gr = NULL, hin = hin, heq = heq, deprecatedBehavior = FALSE)

# $par:   0.8228761 0.9114382
# $value:   1.393464
# $iter:  1001

auglag(x0, fn, gr = NULL, hin = hin, heq = heq, localsolver = "SLSQP",
       deprecatedBehavior = FALSE)

# $par:   0.8228757 0.9114378
# $value:   1.393465
# $iter   184

##  Example from the alabama::auglag help page
##  Parameters should be roughly (0, 0, 1) with an objective value of 1.

fn <- function(x) (x[1] + 3 * x[2] + x[3]) ^ 2 + 4 * (x[1] - x[2]) ^ 2
heq <- function(x) x[1] + x[2] + x[3] - 1
# hin restated from alabama example to be <= 0.
hin <- function(x) c(-6 * x[2] - 4 * x[3] + x[1] ^ 3 + 3, -x[1], -x[2], -x[3])

set.seed(12)
auglag(runif(3), fn, hin = hin, heq = heq, localsolver= "lbfgs",
       deprecatedBehavior = FALSE)

# $par:   4.861756e-08 4.732373e-08 9.999999e-01
# $value:   1
# $iter:  145
```

```
##  Powell problem from the Rsolnp::solnp help page
##  Parameters should be roughly (-1.7171, 1.5957, 1.8272, -0.7636, -0.7636)
##  with an objective value of 0.0539498478.

x0 <- c(-2, 2, 2, -1, -1)
fn1  <- function(x) exp(x[1] * x[2] * x[3] * x[4] * x[5])
eqn1 <-function(x)
c(x[1] * x[1] + x[2] * x[2] + x[3] * x[3] + x[4] * x[4] + x[5] * x[5] - 10,
  x[2] * x[3] - 5 * x[4] * x[5],
  x[1] * x[1] * x[1] + x[2] * x[2] * x[2] + 1)

auglag(x0, fn1, heq = eqn1, localsolver = "mma", deprecatedBehavior = FALSE)

# $par: -1.7173645  1.5959655  1.8268352 -0.7636185 -0.7636185
# $value:   0.05394987
# $iter:  916
```

---

bobyqa *Bound Optimization by Quadratic Approximation*

Description

BOBYQA performs derivative-free bound-constrained optimization using an iteratively constructed quadratic approximation for the objective function.

Usage

```
bobyqa(
  x0,
  fn,
  lower = NULL,
  upper = NULL,
  nl.info = FALSE,
  control = list(),
  ...
)
```

Arguments

x0	starting point for searching the optimum.
fn	objective function that is to be minimized.
lower, upper	lower and upper bound constraints.
nl.info	logical; shall the original NLOPT info be shown.
control	list of options, see nl.opts for help.
...	additional arguments passed to the function.

Details

This is an algorithm derived from the BOBYQA Fortran subroutine of Powell, converted to C and modified for the NLOPT stopping criteria.

Value

List with components:

`par`	the optimal solution found so far.
`value`	the function value corresponding to `par`.
`iter`	number of (outer) iterations, see `maxeval`.
`convergence`	integer code indicating successful completion (> 0) or a possible error number (< 0).
`message`	character string produced by NLOPT and giving additional information.

Note

Because BOBYQA constructs a quadratic approximation of the objective, it may perform poorly for objective functions that are not twice-differentiable.

References

M. J. D. Powell. "The BOBYQA algorithm for bound constrained optimization without derivatives," Department of Applied Mathematics and Theoretical Physics, Cambridge England, technical reportNA2009/06 (2009).

See Also

cobyla, newuoa

Examples

```
## Rosenbrock Banana function

rbf <- function(x) {(1 - x[1]) ^ 2 + 100 * (x[2] - x[1] ^ 2) ^ 2}

## The function as written above has a minimum of 0 at (1, 1)

S <- bobyqa(c(0, 0), rbf)


S

## Rosenbrock Banana function with both parameters constrained to [0, 0.5]

S <- bobyqa(c(0, 0), rbf, lower = c(0, 0), upper = c(0.5, 0.5))

S
```

ccsaq *Conservative Convex Separable Approximation with Affine Approximation plus Quadratic Penalty*

Description

This is a variant of CCSA ("conservative convex separable approximation") which, instead of constructing local MMA approximations, constructs simple quadratic approximations (or rather, affine approximations plus a quadratic penalty term to stay conservative)

Usage

```
ccsaq(
  x0,
  fn,
  gr = NULL,
  lower = NULL,
  upper = NULL,
  hin = NULL,
  hinjac = NULL,
  nl.info = FALSE,
  control = list(),
  deprecatedBehavior = TRUE,
  ...
)
```

Arguments

x0	starting point for searching the optimum.
fn	objective function that is to be minimized.
gr	gradient of function fn; will be calculated numerically if not specified.
lower, upper	lower and upper bound constraints.
hin	function defining the inequality constraints, that is hin>=0 for all components.
hinjac	Jacobian of function hin; will be calculated numerically if not specified.
nl.info	logical; shall the original NLopt info been shown.
control	list of options, see nl.opts for help.
deprecatedBehavior	logical; if TRUE (default for now), the old behavior of the Jacobian function is used, where the equality is ≥ 0 instead of ≤ 0. This will be reversed in a future release and eventually removed.
...	additional arguments passed to the function.

Value

List with components:

par	the optimal solution found so far.
value	the function value corresponding to par.
iter	number of (outer) iterations, see maxeval.
convergence	integer code indicating successful completion (> 1) or a possible error number (< 0).
message	character string produced by NLopt and giving additional information.

Note

"Globally convergent" does not mean that this algorithm converges to the global optimum; it means that it is guaranteed to converge to some local minimum from any feasible starting point.

References

Krister Svanberg, "A class of globally convergent optimization methods based on conservative convex separable approximations," SIAM J. Optim. 12 (2), p. 555-573 (2002).

See Also

mma

Examples

```
##  Solve the Hock-Schittkowski problem no. 100 with analytic gradients
##  See https://apmonitor.com/wiki/uploads/Apps/hs100.apm

x0.hs100 <- c(1, 2, 0, 4, 0, 1, 1)
fn.hs100 <- function(x) {(x[1] - 10) ^ 2 + 5 * (x[2] - 12) ^ 2 + x[3] ^ 4 +
                         3 * (x[4] - 11) ^ 2 + 10 * x[5] ^ 6 + 7 * x[6] ^ 2 +
                         x[7] ^ 4 - 4 * x[6] * x[7] - 10 * x[6] - 8 * x[7]}

hin.hs100 <- function(x) {c(
2 * x[1] ^ 2 + 3 * x[2] ^ 4 + x[3] + 4 * x[4] ^ 2 + 5 * x[5] - 127,
7 * x[1] + 3 * x[2] + 10 * x[3] ^ 2 + x[4] - x[5] - 282,
23 * x[1] + x[2] ^ 2 + 6 * x[6] ^ 2 - 8 * x[7] - 196,
4 * x[1] ^ 2 + x[2] ^ 2 - 3 * x[1] * x[2] + 2 * x[3] ^ 2 + 5 * x[6] -
 11 * x[7])
}

gr.hs100 <- function(x) {
 c( 2 * x[1] - 20,
   10 * x[2] - 120,
    4 * x[3] ^ 3,
    6 * x[4] - 66,
   60 * x[5] ^ 5,
   14 * x[6] - 4 * x[7] - 10,
    4 * x[7] ^ 3 - 4 * x[6] - 8)
```

```
}

hinjac.hs100 <- function(x) {
  matrix(c(4 * x[1], 12 * x[2] ^ 3, 1, 8 * x[4], 5, 0, 0,
           7, 3, 20 * x[3], 1, -1, 0, 0,
           23, 2 * x[2], 0, 0, 0, 12 * x[6], -8,
           8 * x[1] - 3 * x[2], 2 * x[2] - 3 * x[1], 4 * x[3], 0, 0, 5, -11),
           nrow = 4, byrow = TRUE)
}

##  The optimum value of the objective function should be 680.6300573
##  A suitable parameter vector is roughly
##  (2.330, 1.9514, -0.4775, 4.3657, -0.6245, 1.0381, 1.5942)

# Results with exact Jacobian
S <- ccsaq(x0.hs100, fn.hs100, gr = gr.hs100,
      hin = hin.hs100, hinjac = hinjac.hs100,
      nl.info = TRUE, control = list(xtol_rel = 1e-8),
      deprecatedBehavior = FALSE)

# Results without Jacobian
S <- ccsaq(x0.hs100, fn.hs100, hin = hin.hs100,
      nl.info = TRUE, control = list(xtol_rel = 1e-8),
      deprecatedBehavior = FALSE)
```

`check.derivatives` *Check analytic gradients of a function using finite difference approximations*

Description

This function compares the analytic gradients of a function with a finite difference approximation and prints the results of these checks.

Usage

```
check.derivatives(
  .x,
  func,
  func_grad,
  check_derivatives_tol = 1e-04,
  check_derivatives_print = "all",
  func_grad_name = "grad_f",
  ...
)
```

Arguments

.x	point at which the comparison is done.
func	function to be evaluated.
func_grad	function calculating the analytic gradients.
check_derivatives_tol	option determining when differences between the analytic gradient and its finite difference approximation are flagged as an error.
check_derivatives_print	option related to the amount of output. 'all' means that all comparisons are shown, 'errors' only shows comparisons that are flagged as an error, and 'none' shows the number of errors only.
func_grad_name	option to change the name of the gradient function that shows up in the output.
...	further arguments passed to the functions func and func_grad.

Value

The return value contains a list with the analytic gradient, its finite difference approximation, the relative errors, and vector comparing the relative errors to the tolerance.

Author(s)

Jelmer Ypma

See Also

nloptr

Examples

```
library('nloptr')

# example with correct gradient
f <- function(x, a) sum((x - a) ^ 2)

f_grad <- function(x, a)  2 * (x - a)

check.derivatives(.x = 1:10, func = f, func_grad = f_grad,
          check_derivatives_print = 'none', a = runif(10))

# example with incorrect gradient
f_grad <- function(x, a)  2 * (x - a) + c(0, 0.1, rep(0, 8))

check.derivatives(.x = 1:10, func = f, func_grad = f_grad,
          check_derivatives_print = 'errors', a = runif(10))

# example with incorrect gradient of vector-valued function
g <- function(x, a) c(sum(x - a), sum((x - a) ^ 2))

g_grad <- function(x, a) {
```

```
   rbind(rep(1, length(x)) + c(0, 0.01, rep(0, 8)),
   2 * (x - a) + c(0, 0.1, rep(0, 8)))
}

check.derivatives(.x = 1:10, func = g, func_grad = g_grad,
          check_derivatives_print = 'all', a = runif(10))
```

---

cobyla *Constrained Optimization by Linear Approximations*

Description

COBYLA is an algorithm for derivative-free optimization with nonlinear inequality and equality constraints (but see below).

Usage

```
cobyla(
  x0,
  fn,
  lower = NULL,
  upper = NULL,
  hin = NULL,
  nl.info = FALSE,
  control = list(),
  deprecatedBehavior = TRUE,
  ...
)
```

Arguments

x0	starting point for searching the optimum.
fn	objective function that is to be minimized.
lower, upper	lower and upper bound constraints.
hin	function defining the inequality constraints, that is hin>=0 for all components.
nl.info	logical; shall the original NLOPT info be shown.
control	list of options, see nl.opts for help.
deprecatedBehavior	logical; if TRUE (default for now), the old behavior of the Jacobian function is used, where the equality is ≥ 0 instead of ≤ 0. This will be reversed in a future release and eventually removed.
...	additional arguments passed to the function.

Details

It constructs successive linear approximations of the objective function and constraints via a simplex of $n+1$ points (in n dimensions), and optimizes these approximations in a trust region at each step.

COBYLA supports equality constraints by transforming them into two inequality constraints. This functionality has not been added to the wrapper. To use COBYLA with equality constraints, please use the full `nloptr` invocation.

Value

List with components:

`par`	the optimal solution found so far.
`value`	the function value corresponding to `par`.
`iter`	number of (outer) iterations, see `maxeval`.
`convergence`	integer code indicating successful completion (> 0) or a possible error number (< 0).
`message`	character string produced by NLopt and giving additional information.

Note

The original code, written in Fortran by Powell, was converted in C for the SCIPY project.

Author(s)

Hans W. Borchers

References

M. J. D. Powell, "A direct search optimization method that models the objective and constraint functions by linear interpolation," in Advances in Optimization and Numerical Analysis, eds. S. Gomez and J.-P. Hennart (Kluwer Academic: Dordrecht, 1994), p. 51-67.

See Also

bobyqa, newuoa

Examples

```
##  Solve the Hock-Schittkowski problem no. 100 with analytic gradients
##  See https://apmonitor.com/wiki/uploads/Apps/hs100.apm

x0.hs100 <- c(1, 2, 0, 4, 0, 1, 1)
fn.hs100 <- function(x) {(x[1] - 10) ^ 2 + 5 * (x[2] - 12) ^ 2 + x[3] ^ 4 +
                         3 * (x[4] - 11) ^ 2 + 10 * x[5] ^ 6 + 7 * x[6] ^ 2 +
                         x[7] ^ 4 - 4 * x[6] * x[7] - 10 * x[6] - 8 * x[7]}

hin.hs100 <- function(x) {c(
2 * x[1] ^ 2 + 3 * x[2] ^ 4 + x[3] + 4 * x[4] ^ 2 + 5 * x[5] - 127,
7 * x[1] + 3 * x[2] + 10 * x[3] ^ 2 + x[4] - x[5] - 282,
```

```
23 * x[1] + x[2] ^ 2 + 6 * x[6] ^ 2 - 8 * x[7] - 196,
4 * x[1] ^ 2 + x[2] ^ 2 - 3 * x[1] * x[2] + 2 * x[3] ^ 2 + 5 * x[6] -
 11 * x[7])
}

S <- cobyla(x0.hs100, fn.hs100, hin = hin.hs100,
      nl.info = TRUE, control = list(xtol_rel = 1e-8, maxeval = 2000),
      deprecatedBehavior = FALSE)

##  The optimum value of the objective function should be 680.6300573
##  A suitable parameter vector is roughly
##  (2.330, 1.9514, -0.4775, 4.3657, -0.6245, 1.0381, 1.5942)

S
```

`crs2lm` *Controlled Random Search*

Description

The Controlled Random Search (CRS) algorithm (and in particular, the CRS2 variant) with the 'local mutation' modification.

Usage

```
crs2lm(
  x0,
  fn,
  lower,
  upper,
  maxeval = 10000,
  pop.size = 10 * (length(x0) + 1),
  ranseed = NULL,
  xtol_rel = 1e-06,
  nl.info = FALSE,
  ...
)
```

Arguments

`x0`	initial point for searching the optimum.
`fn`	objective function that is to be minimized.
`lower`, `upper`	lower and upper bound constraints.
`maxeval`	maximum number of function evaluations.
`pop.size`	population size.
`ranseed`	prescribe seed for random number generator.

xtol_rel	stopping criterion for relative change reached.
nl.info	logical; shall the original NLOPT info be shown.
...	additional arguments passed to the function.

Details

The CRS algorithms are sometimes compared to genetic algorithms, in that they start with a random population of points, and randomly evolve these points by heuristic rules. In this case, the evolution somewhat resembles a randomized Nelder-Mead algorithm.

The published results for CRS seem to be largely empirical.

Value

List with components:

par	the optimal solution found so far.
value	the function value corresponding to par.
iter	number of (outer) iterations, see maxeval.
convergence	integer code indicating successful completion (> 0) or a possible error number (< 0).
message	character string produced by NLOPT and giving additional information.

Note

The initial population size for CRS defaults to $10x(n+1)$ in n dimensions, but this can be changed. The initial population must be at least $n + 1$.

References

W. L. Price, "Global optimization by controlled random search," J. Optim. Theory Appl. 40 (3), p. 333-348 (1983).

P. Kaelo and M. M. Ali, "Some variants of the controlled random search algorithm for global optimization," J. Optim. Theory Appl. 130 (2), 253-264 (2006).

Examples

```
## Minimize the Hartmann 6-Dimensional function
## See https://www.sfu.ca/~ssurjano/hart6.html

a <- c(1.0, 1.2, 3.0, 3.2)
A <- matrix(c(10,  0.05, 3, 17,
              3, 10, 3.5, 8,
              17, 17, 1.7, 0.05,
              3.5, 0.1, 10, 10,
              1.7, 8, 17, 0.1,
              8, 14, 8, 14), nrow = 4)

B  <- matrix(c(.1312, .2329, .2348, .4047,
```

```
                .1696, .4135, .1451, .8828,
                .5569, .8307, .3522, .8732,
                .0124, .3736, .2883, .5743,
                .8283, .1004, .3047, .1091,
                .5886, .9991, .6650, .0381), nrow = 4)

hartmann6 <- function(x, a, A, B) {
  fun <- 0
  for (i in 1:4) {
    fun <- fun - a[i] * exp(-sum(A[i, ] * (x - B[i, ]) ^ 2))
  }

  fun
}

## The function has a global minimum of -3.32237 at
## (0.20169, 0.150011, 0.476874, 0.275332, 0.311652, 0.6573)

S <- crs2lm(x0 = rep(0, 6), hartmann6, lower = rep(0, 6), upper = rep(1, 6),
            ranseed = 10L, nl.info = TRUE, xtol_rel=1e-8, maxeval = 10000,
            a = a, A = A, B = B)

S
```

---

direct *DIviding RECTangles Algorithm for Global Optimization*

Description

DIRECT is a deterministic search algorithm based on systematic division of the search domain into smaller and smaller hyperrectangles. The DIRECT_L makes the algorithm more biased towards local search (more efficient for functions without too many minima).

Usage

```
direct(
  fn,
  lower,
  upper,
  scaled = TRUE,
  original = FALSE,
  nl.info = FALSE,
  control = list(),
  ...
)

directL(
  fn,
```

```
  lower,
  upper,
  randomized = FALSE,
  original = FALSE,
  nl.info = FALSE,
  control = list(),
  ...
)
```

Arguments

fn	objective function that is to be minimized.
lower, upper	lower and upper bound constraints.
scaled	logical; shall the hypercube be scaled before starting.
original	logical; whether to use the original implementation by Gablonsky – the performance is mostly similar.
nl.info	logical; shall the original NLopt info been shown.
control	list of options, see nl.opts for help.
...	additional arguments passed to the function.
randomized	logical; shall some randomization be used to decide which dimension to halve next in the case of near-ties.

Details

The DIRECT and DIRECT-L algorithms start by rescaling the bound constraints to a hypercube, which gives all dimensions equal weight in the search procedure. If your dimensions do not have equal weight, e.g. if you have a "long and skinny" search space and your function varies at about the same speed in all directions, it may be better to use unscaled variant of the DIRECT algorithm.

The algorithms only handle finite bound constraints which must be provided. The original versions may include some support for arbitrary nonlinear inequality, but this has not been tested.

The original versions do not have randomized or unscaled variants, so these options will be disregarded for these versions.

Value

List with components:

par	the optimal solution found so far.
value	the function value corresponding to par.
iter	number of (outer) iterations, see maxeval.
convergence	integer code indicating successful completion (> 0) or a possible error number (< 0).
message	character string produced by NLopt and giving additional information.

Note

The DIRECT_L algorithm should be tried first.

Author(s)

Hans W. Borchers

References

D. R. Jones, C. D. Perttunen, and B. E. Stuckmann, “Lipschitzian optimization without the Lipschitz constant,” J. Optimization Theory and Applications, vol. 79, p. 157 (1993).

J. M. Gablonsky and C. T. Kelley, “A locally-biased form of the DIRECT algorithm," J. Global Optimization, vol. 21 (1), p. 27-37 (2001).

See Also

The `dfoptim` package will provide a pure R version of this algorithm.

Examples

```
### Minimize the Hartmann6 function
hartmann6 <- function(x) {
  a <- c(1.0, 1.2, 3.0, 3.2)
  A <- matrix(c(10.0,  0.05, 3.0, 17.0,
         3.0, 10.0,  3.5,  8.0,
          17.0, 17.0,  1.7,  0.05,
         3.5,  0.1, 10.0, 10.0,
         1.7,  8.0, 17.0,  0.1,
         8.0, 14.0,  8.0, 14.0), nrow=4, ncol=6)
  B  <- matrix(c(.1312,.2329,.2348,.4047,
         .1696,.4135,.1451,.8828,
         .5569,.8307,.3522,.8732,
         .0124,.3736,.2883,.5743,
         .8283,.1004,.3047,.1091,
         .5886,.9991,.6650,.0381), nrow=4, ncol=6)
  fun <- 0
  for (i in 1:4) {
    fun <- fun - a[i] * exp(-sum(A[i,] * (x - B[i,]) ^ 2))
  }
  fun
}
S <- directL(hartmann6, rep(0, 6), rep(1, 6),
       nl.info = TRUE, control = list(xtol_rel = 1e-8, maxeval = 1000))
## Number of Iterations....: 1000
## Termination conditions:  stopval: -Inf
##   xtol_rel: 1e-08,  maxeval: 1000,  ftol_rel: 0,  ftol_abs: 0
## Number of inequality constraints:  0
## Number of equality constraints:  0
## Current value of objective function:  -3.32236800687327
## Current value of controls:
##   0.2016884 0.1500025 0.4768667 0.2753391 0.311648 0.6572931
```

is.nloptr *R interface to NLopt*

Description

is.nloptr preforms checks to see if a fully specified problem is supplied to nloptr. Mostly for internal use.

Usage

```
is.nloptr(x)
```

Arguments

x	object to be tested.

Value

Logical. Return TRUE if all tests were passed, otherwise return FALSE or exit with Error.

Author(s)

Jelmer Ypma

See Also

nloptr

isres *Improved Stochastic Ranking Evolution Strategy*

Description

The Improved Stochastic Ranking Evolution Strategy (ISRES) is an algorithm for nonlinearly constrained global optimization, or at least semi-global, although it has heuristics to escape local optima.

Usage

```
isres(
  x0,
  fn,
  lower,
  upper,
  hin = NULL,
  heq = NULL,
```

```
  maxeval = 10000,
  pop.size = 20 * (length(x0) + 1),
  xtol_rel = 1e-06,
  nl.info = FALSE,
  deprecatedBehavior = TRUE,
  ...
)
```

Arguments

x0	initial point for searching the optimum.
fn	objective function that is to be minimized.
lower, upper	lower and upper bound constraints.
hin	function defining the inequality constraints, that is hin <= 0 for all components.
heq	function defining the equality constraints, that is heq = 0 for all components.
maxeval	maximum number of function evaluations.
pop.size	population size.
xtol_rel	stopping criterion for relative change reached.
nl.info	logical; shall the original NLOPT info be shown.
deprecatedBehavior	logical; if TRUE (default for now), the old behavior of the Jacobian function is used, where the equality is ≥ 0 instead of ≤ 0. This will be reversed in a future release and eventually removed.
...	additional arguments passed to the function.

Details

The evolution strategy is based on a combination of a mutation rule—with a log-normal step-size update and exponential smoothing—and differential variation—a Nelder-Mead-like update rule). The fitness ranking is simply via the objective function for problems without nonlinear constraints, but when nonlinear constraints are included the stochastic ranking proposed by Runarsson and Yao is employed.

This method supports arbitrary nonlinear inequality and equality constraints in addition to the bounds constraints.

Value

List with components:

par	the optimal solution found so far.
value	the function value corresponding to par.
iter	number of (outer) iterations, see maxeval.
convergence	integer code indicating successful completion (> 0) or a possible error number (< 0).
message	character string produced by NLopt and giving additional information.

Note

The initial population size for CRS defaults to $20x(n+1)$ in n dimensions, but this can be changed. The initial population must be at least $n + 1$.

Author(s)

Hans W. Borchers

References

Thomas Philip Runarsson and Xin Yao, "Search biases in constrained evolutionary optimization," IEEE Trans. on Systems, Man, and Cybernetics Part C: Applications and Reviews, vol. 35 (no. 2), pp. 233-243 (2005).

Examples

```
## Rosenbrock Banana objective function

rbf <- function(x) {(1 - x[1]) ^ 2 + 100 * (x[2] - x[1] ^ 2) ^ 2}

x0 <- c(-1.2, 1)
lb <- c(-3, -3)
ub <- c(3,  3)

## The function as written above has a minimum of 0 at (1, 1)

isres(x0 = x0, fn = rbf, lower = lb, upper = ub)

## Now subject to the inequality that x[1] + x[2] <= 1.5

hin <- function(x) {x[1] + x[2] - 1.5}

S <- isres(x0 = x0, fn = rbf, hin = hin, lower = lb, upper = ub,
           maxeval = 2e5L, deprecatedBehavior = FALSE)

S

sum(S$par)
```

---

lbfgs *Low-storage BFGS*

Description

Low-storage version of the Broyden-Fletcher-Goldfarb-Shanno (BFGS) method.

Usage

```
lbfgs(
  x0,
  fn,
  gr = NULL,
  lower = NULL,
  upper = NULL,
  nl.info = FALSE,
  control = list(),
  ...
)
```

Arguments

x0	initial point for searching the optimum.
fn	objective function to be minimized.
gr	gradient of function fn; will be calculated numerically if not specified.
lower, upper	lower and upper bound constraints.
nl.info	logical; shall the original NLopt info been shown.
control	list of control parameters, see nl.opts for help.
...	further arguments to be passed to the function.

Details

The low-storage (or limited-memory) algorithm is a member of the class of quasi-Newton optimization methods. It is well suited for optimization problems with a large number of variables.

One parameter of this algorithm is the number m of gradients to remember from previous optimization steps. NLopt sets m to a heuristic value by default. It can be changed by the NLopt function set_vector_storage.

Value

List with components:

par	the optimal solution found so far.
value	the function value corresponding to par.
iter	number of (outer) iterations, see maxeval.
convergence	integer code indicating successful completion (> 0) or a possible error number (< 0).
message	character string produced by NLopt and giving additional information.

Note

Based on a Fortran implementation of the low-storage BFGS algorithm written by L. Luksan, and posted under the GNU LGPL license.

Author(s)

Hans W. Borchers

References

J. Nocedal, "Updating quasi-Newton matrices with limited storage," Math. Comput. 35, 773-782 (1980).

D. C. Liu and J. Nocedal, "On the limited memory BFGS method for large scale optimization," Math. Programming 45, p. 503-528 (1989).

See Also

optim

Examples

```
flb <- function(x) {
  p <- length(x)
  sum(c(1, rep(4, p-1)) * (x - c(1, x[-p])^2)^2)
}
# 25-dimensional box constrained: par[24] is *not* at the boundary
S <- lbfgs(rep(3, 25), flb, lower=rep(2, 25), upper=rep(4, 25),
     nl.info = TRUE, control = list(xtol_rel=1e-8))
## Optimal value of objective function:  368.105912874334
## Optimal value of controls: 2  ...  2  2.109093  4
```

---

mlsl *Multi-level Single-linkage*

Description

The "Multi-Level Single-Linkage" (MLSL) algorithm for global optimization searches by a sequence of local optimizations from random starting points. A modification of MLSL is included using a low-discrepancy sequence (LDS) instead of pseudorandom numbers.

Usage

```
mlsl(
  x0,
  fn,
  gr = NULL,
  lower,
  upper,
  local.method = "LBFGS",
  low.discrepancy = TRUE,
  nl.info = FALSE,
```

```
  control = list(),
  ...
)
```

Arguments

x0	initial point for searching the optimum.
fn	objective function that is to be minimized.
gr	gradient of function fn; will be calculated numerically if not specified.
lower, upper	lower and upper bound constraints.
local.method	only BFGS for the moment.
low.discrepancy	logical; shall a low discrepancy variation be used.
nl.info	logical; shall the original NLOPT info be shown.
control	list of options, see nl.opts for help.
...	additional arguments passed to the function.

Details

MLSL is a 'multistart' algorithm: it works by doing a sequence of local optimizations—using some other local optimization algorithm—from random or low-discrepancy starting points. MLSL is distinguished, however, by a 'clustering' heuristic that helps it to avoid repeated searches of the same local optima and also has some theoretical guarantees of finding all local optima in a finite number of local minimizations.

The local-search portion of MLSL can use any of the other algorithms in NLOPT, and, in particular, can use either gradient-based or derivative-free algorithms. For this wrapper only gradient-based LBFGS is available as local method.

Value

List with components:

par	the optimal solution found so far.
value	the function value corresponding to par.
iter	number of (outer) iterations, see maxeval.
convergence	integer code indicating successful completion (> 0) or a possible error number (< 0).
message	character string produced by NLOPT and giving additional information.

Note

If you don't set a stopping tolerance for your local-optimization algorithm, MLSL defaults to ftol_rel = 1e-15 and xtol_rel = 1e-7 for the local searches.

Author(s)

Hans W. Borchers

References

A. H. G. Rinnooy Kan and G. T. Timmer, "Stochastic global optimization methods" Mathematical Programming, vol. 39, p. 27-78 (1987).

Sergei Kucherenko and Yury Sytsko, "Application of deterministic low-discrepancy sequences in global optimization", Computational Optimization and Applications, vol. 30, p. 297-318 (2005).

See Also

direct

Examples

```
## Minimize the Hartmann 6-Dimensional function
## See https://www.sfu.ca/~ssurjano/hart6.html

a <- c(1.0, 1.2, 3.0, 3.2)
A <- matrix(c(10,  0.05, 3, 17,
              3, 10, 3.5, 8,
              17, 17, 1.7, 0.05,
              3.5, 0.1, 10, 10,
              1.7, 8, 17, 0.1,
              8, 14, 8, 14), nrow = 4)

B  <- matrix(c(.1312, .2329, .2348, .4047,
               .1696, .4135, .1451, .8828,
               .5569, .8307, .3522, .8732,
               .0124, .3736, .2883, .5743,
               .8283, .1004, .3047, .1091,
               .5886, .9991, .6650, .0381), nrow = 4)

hartmann6 <- function(x, a, A, B) {
  fun <- 0
  for (i in 1:4) {
    fun <- fun - a[i] * exp(-sum(A[i, ] * (x - B[i, ]) ^ 2))
  }

  fun
}

## The function has a global minimum of -3.32237 at
## (0.20169, 0.150011, 0.476874, 0.275332, 0.311652, 0.6573)

S <- mlsl(x0 = rep(0, 6), hartmann6, lower = rep(0, 6), upper = rep(1, 6),
      nl.info = TRUE, control = list(xtol_rel = 1e-8, maxeval = 1000),
      a = a, A = A, B = B)
```

mma *Method of Moving Asymptotes*

Description

Globally-convergent method-of-moving-asymptotes (MMA) algorithm for gradient-based local optimization, including nonlinear inequality constraints (but not equality constraints).

Usage

```
mma(
  x0,
  fn,
  gr = NULL,
  lower = NULL,
  upper = NULL,
  hin = NULL,
  hinjac = NULL,
  nl.info = FALSE,
  control = list(),
  deprecatedBehavior = TRUE,
  ...
)
```

Arguments

x0	starting point for searching the optimum.
fn	objective function that is to be minimized.
gr	gradient of function fn; will be calculated numerically if not specified.
lower, upper	lower and upper bound constraints.
hin	function defining the inequality constraints, that is hin <= 0 for all components.
hinjac	Jacobian of function hin; will be calculated numerically if not specified.
nl.info	logical; shall the original NLopt info been shown.
control	list of options, see nl.opts for help.
deprecatedBehavior	logical; if TRUE (default for now), the old behavior of the Jacobian function is used, where the equality is ≥ 0 instead of ≤ 0. This will be reversed in a future release and eventually removed.
...	additional arguments passed to the function.

Details

This is an improved CCSA ("conservative convex separable approximation") variant of the original MMA algorithm published by Svanberg in 1987, which has become popular for topology optimization.

Value

List with components:

par	the optimal solution found so far.
value	the function value corresponding to par.
iter	number of (outer) iterations, see maxeval.
convergence	integer code indicating successful completion (> 1) or a possible error number (< 0).
message	character string produced by NLopt and giving additional information.

Note

"Globally convergent" does not mean that this algorithm converges to the global optimum; rather, it means that the algorithm is guaranteed to converge to some local minimum from any feasible starting point.

Author(s)

Hans W. Borchers

References

Krister Svanberg, "A class of globally convergent optimization methods based on conservative convex separable approximations", SIAM J. Optim. 12 (2), p. 555-573 (2002).

See Also

slsqp

Examples

```
#  Solve the Hock-Schittkowski problem no. 100 with analytic gradients
#  See https://apmonitor.com/wiki/uploads/Apps/hs100.apm

x0.hs100 <- c(1, 2, 0, 4, 0, 1, 1)
fn.hs100 <- function(x) {(x[1] - 10) ^ 2 + 5 * (x[2] - 12) ^ 2 + x[3] ^ 4 +
                         3 * (x[4] - 11) ^ 2 + 10 * x[5] ^ 6 + 7 * x[6] ^ 2 +
                         x[7] ^ 4 - 4 * x[6] * x[7] - 10 * x[6] - 8 * x[7]}

hin.hs100 <- function(x) {c(
2 * x[1] ^ 2 + 3 * x[2] ^ 4 + x[3] + 4 * x[4] ^ 2 + 5 * x[5] - 127,
7 * x[1] + 3 * x[2] + 10 * x[3] ^ 2 + x[4] - x[5] - 282,
23 * x[1] + x[2] ^ 2 + 6 * x[6] ^ 2 - 8 * x[7] - 196,
4 * x[1] ^ 2 + x[2] ^ 2 - 3 * x[1] * x[2] + 2 * x[3] ^ 2 + 5 * x[6] -
 11 * x[7])
}

gr.hs100 <- function(x) {
 c( 2 * x[1] - 20,
   10 * x[2] - 120,
```

```
    4 * x[3] ^ 3,
    6 * x[4] - 66,
   60 * x[5] ^ 5,
   14 * x[6] - 4 * x[7] - 10,
    4 * x[7] ^ 3 - 4 * x[6] - 8)
}

hinjac.hs100 <- function(x) {
  matrix(c(4 * x[1], 12 * x[2] ^ 3, 1, 8 * x[4], 5, 0, 0,
           7, 3, 20 * x[3], 1, -1, 0, 0,
           23, 2 * x[2], 0, 0, 0, 12 * x[6], -8,
           8 * x[1] - 3 * x[2], 2 * x[2] - 3 * x[1], 4 * x[3], 0, 0, 5, -11),
           nrow = 4, byrow = TRUE)
}

#  The optimum value of the objective function should be 680.6300573
#  A suitable parameter vector is roughly
#  (2.330, 1.9514, -0.4775, 4.3657, -0.6245, 1.0381, 1.5942)

# Using analytic Jacobian
S <- mma(x0.hs100, fn.hs100, gr = gr.hs100,
      hin = hin.hs100, hinjac = hinjac.hs100,
      nl.info = TRUE, control = list(xtol_rel = 1e-8),
      deprecatedBehavior = FALSE)

# Using computed Jacobian
S <- mma(x0.hs100, fn.hs100, hin = hin.hs100,
      nl.info = TRUE, control = list(xtol_rel = 1e-8),
      deprecatedBehavior = FALSE)
```

`neldermead` *Nelder-Mead Simplex*

Description

An implementation of almost the original Nelder-Mead simplex algorithm.

Usage

```
neldermead(
  x0,
  fn,
  lower = NULL,
  upper = NULL,
  nl.info = FALSE,
  control = list(),
  ...
)
```

Arguments

x0	starting point for searching the optimum.
fn	objective function that is to be minimized.
lower, upper	lower and upper bound constraints.
nl.info	logical; shall the original NLopt info been shown.
control	list of options, see nl.opts for help.
...	additional arguments passed to the function.

Details

Provides explicit support for bound constraints, using essentially the method proposed in Box. Whenever a new point would lie outside the bound constraints the point is moved back exactly onto the constraint.

Value

List with components:

par	the optimal solution found so far.
value	the function value corresponding to par.
iter	number of (outer) iterations, see maxeval.
convergence	integer code indicating successful completion (> 0) or a possible error number (< 0).
message	character string produced by NLopt and giving additional information.

Note

The author of NLopt would tend to recommend the Subplex method instead.

Author(s)

Hans W. Borchers

References

J. A. Nelder and R. Mead, "A simplex method for function minimization," The Computer Journal 7, p. 308-313 (1965).

M. J. Box, "A new method of constrained optimization and a comparison with other methods," Computer J. 8 (1), 42-52 (1965).

See Also

dfoptim::nmk

Examples

```
# Fletcher and Powell's helic valley
fphv <- function(x)
  100*(x[3] - 10*atan2(x[2], x[1])/(2*pi))^2 +
    (sqrt(x[1]^2 + x[2]^2) - 1)^2 +x[3]^2
x0 <- c(-1, 0, 0)
neldermead(x0, fphv)  #  1 0 0

# Powell's Singular Function (PSF)
psf <- function(x)  (x[1] + 10*x[2])^2 + 5*(x[3] - x[4])^2 +
          (x[2] - 2*x[3])^4 + 10*(x[1] - x[4])^4
x0 <- c(3, -1, 0, 1)
neldermead(x0, psf)   #  0 0 0 0, needs maximum number of function calls

## Not run:
# Bounded version of Nelder-Mead
rosenbrock <- function(x) { ## Rosenbrock Banana function
  100 * (x[2] - x[1]^2)^2 + (1 - x[1])^2 +
  100 * (x[3] - x[2]^2)^2 + (1 - x[2])^2
}
lower <- c(-Inf, 0,   0)
upper <- c( Inf, 0.5, 1)
x0 <- c(0, 0.1, 0.1)
S <- neldermead(c(0, 0.1, 0.1), rosenbrock, lower, upper, nl.info = TRUE)
# $xmin = c(0.7085595, 0.5000000, 0.2500000)
# $fmin = 0.3353605
## End(Not run)
```

---

newuoa *New Unconstrained Optimization with quadratic Approximation*

Description

NEWUOA solves quadratic subproblems in a spherical trust region via a truncated conjugate-gradient algorithm. For bound-constrained problems, BOBYQA should be used instead, as Powell developed it as an enhancement thereof for bound constraints.

Usage

```
newuoa(x0, fn, nl.info = FALSE, control = list(), ...)
```

Arguments

x0	starting point for searching the optimum.
fn	objective function that is to be minimized.
nl.info	logical; shall the original NLOPT info be shown.
control	list of options, see nl.opts for help.
...	additional arguments passed to the function.

Details

This is an algorithm derived from the NEWUOA Fortran subroutine of Powell, converted to C and modified for the NLOPT stopping criteria.

Value

List with components:

`par`	the optimal solution found so far.
`value`	the function value corresponding to `par`.
`iter`	number of (outer) iterations, see `maxeval`.
`convergence`	integer code indicating successful completion (> 0) or a possible error number (< 0).
`message`	character string produced by NLopt and giving additional information.

Note

NEWUOA may be largely superseded by BOBYQA.

Author(s)

Hans W. Borchers

References

M. J. D. Powell. "The BOBYQA algorithm for bound constrained optimization without derivatives," Department of Applied Mathematics and Theoretical Physics, Cambridge England, technical reportNA2009/06 (2009).

See Also

bobyqa, cobyla

Examples

```
## Rosenbrock Banana function

rbf <- function(x) {(1 - x[1]) ^ 2 + 100 * (x[2] - x[1] ^ 2) ^ 2}

S <- newuoa(c(1, 2), rbf)

## The function as written above has a minimum of 0 at (1, 1)

S
```

`nl.grad` *Numerical Gradients and Jacobians*

Description

Provides numerical gradients and Jacobians.

Usage

```
nl.grad(x0, fn, heps = .Machine$double.eps^(1/3), ...)
```

Arguments

`x0`	point as a vector where the gradient is to be calculated.
`fn`	scalar function of one or several variables.
`heps`	step size to be used.
`...`	additional arguments passed to the function.

Details

Both functions apply the "central difference formula" with step size as recommended in the literature.

Value

`grad` returns the gradient as a vector; `jacobian` returns the Jacobian as a matrix of usual dimensions.

Author(s)

Hans W. Borchers

Examples

```
fn1 <- function(x) sum(x ^ 2)
nl.grad(seq(0, 1, by = 0.2), fn1)
## [1] 0.0  0.4  0.8  1.2  1.6  2.0
nl.grad(rep(1, 5), fn1)
## [1] 2  2  2  2  2

fn2 <- function(x) c(sin(x), cos(x))
x <- (0:1) * 2 * pi
nl.jacobian(x, fn2)
##    [,1] [,2]
## [1,]  1  0
## [2,]  0  1
## [3,]  0  0
## [4,]  0  0
```

nl.opts *Setting NL Options*

Description

Sets and changes the NLOPT options.

Usage

```
nl.opts(optlist = NULL)
```

Arguments

optlist	list of options, see below.

Details

The following options can be set (here with default values):

```
stopval = -Inf, # stop minimization at this value
xtol_rel = 1e-6, # stop on small optimization step
maxeval = 1000, # stop on this many function evaluations
ftol_rel = 0.0, # stop on change times function value
ftol_abs = 0.0, # stop on small change of function value
check_derivatives = FALSE
```

Value

returns a list with default and changed options.

Note

There are more options that can be set for solvers in NLOPT. These cannot be set through their wrapper functions. To see the full list of options and algorithms, type `nloptr.print.options()`.

Author(s)

Hans W. Borchers

Examples

```
nl.opts(list(xtol_rel = 1e-8, maxeval = 2000))
```

`nloptr` *R interface to NLopt*

Description

nloptr is an R interface to NLopt, a free/open-source library for nonlinear optimization started by Steven G. Johnson, providing a common interface for a number of different free optimization routines available online as well as original implementations of various other algorithms. The NLopt library is available under the GNU Lesser General Public License (LGPL), and the copyrights are owned by a variety of authors. Most of the information here has been taken from the NLopt website, where more details are available.

Usage

```
nloptr(
  x0,
  eval_f,
  eval_grad_f = NULL,
  lb = NULL,
  ub = NULL,
  eval_g_ineq = NULL,
  eval_jac_g_ineq = NULL,
  eval_g_eq = NULL,
  eval_jac_g_eq = NULL,
  opts = list(),
  ...
)
```

Arguments

`x0`	vector with starting values for the optimization.
`eval_f`	function that returns the value of the objective function. It can also return gradient information at the same time in a list with elements "objective" and "gradient" (see below for an example).
`eval_grad_f`	function that returns the value of the gradient of the objective function. Not all of the algorithms require a gradient.
`lb`	vector with lower bounds of the controls (use `-Inf` for controls without lower bound), by default there are no lower bounds for any of the controls.
`ub`	vector with upper bounds of the controls (use `Inf` for controls without upper bound), by default there are no upper bounds for any of the controls.
`eval_g_ineq`	function to evaluate (non-)linear inequality constraints that should hold in the solution. It can also return gradient information at the same time in a list with elements "constraints" and "jacobian" (see below for an example).
`eval_jac_g_ineq`	function to evaluate the Jacobian of the (non-)linear inequality constraints that should hold in the solution.

`eval_g_eq` function to evaluate (non-)linear equality constraints that should hold in the solution. It can also return gradient information at the same time in a list with elements "constraints" and "jacobian" (see below for an example).

`eval_jac_g_eq` function to evaluate the Jacobian of the (non-)linear equality constraints that should hold in the solution.

`opts` list with options. The option "`algorithm`" is required. Check the NLopt website for a full list of available algorithms. Other options control the termination conditions (`minf_max`, `ftol_rel`, `ftol_abs`, `xtol_rel`, `xtol_abs`, `maxeval`, `maxtime`). Default is `xtol_rel` = 1e-4. More information here. #nolint A full description of all options is shown by the function `nloptr.print.options()`.

Some algorithms with equality constraints require the option `local_opts`, which contains a list with an algorithm and a termination condition for the local algorithm. See `?`nloptr-package`` for an example.

The option `print_level` controls how much output is shown during the optimization process. Possible values:

0 (default)	no output
1	show iteration number and value of objective function
2	1 + show value of (in)equalities
3	2 + show value of controls

The option `check_derivatives` (default = `FALSE`) can be used to run to compare the analytic gradients with finite difference approximations. The option `check_derivatives_print` (`'all'` (default), `'errors'`, `'none'`) controls the output of the derivative checker, if it is run, showing all comparisons, only those that resulted in an error, or none. The option `check_derivatives_tol` (default = 1e-04), determines when a difference between an analytic gradient and its finite difference approximation is flagged as an error.

`...` arguments that will be passed to the user-defined objective and constraints functions.

Details

NLopt addresses general nonlinear optimization problems of the form:

$$\min f(x) \quad x \in R^n$$

$$\text{s.t. } g(x) \leq 0 h(x) = 0 lb \leq x \leq ub$$

where $f(x)$ is the objective function to be minimized and x represents the n optimization parameters. This problem may optionally be subject to the bound constraints (also called box constraints), lb and ub. For partially or totally unconstrained problems the bounds can take `-Inf` or `Inf`. One may also optionally have m nonlinear inequality constraints (sometimes called a nonlinear programming problem), which can be specified in $g(x)$, and equality constraints that can be specified in $h(x)$. Note that not all of the algorithms in NLopt can handle constraints.

Value

The return value contains a list with the inputs, and additional elements

`call`	the call that was made to solve
`status`	integer value with the status of the optimization (0 is success)
`message`	more informative message with the status of the optimization
`iterations`	number of iterations that were executed
`objective`	value if the objective function in the solution
`solution`	optimal value of the controls
`version`	version of NLopt that was used

Note

See `?`nloptr-package`` for an extended example.

Author(s)

Steven G. Johnson and others (C code)
Jelmer Ypma (R interface)

References

Steven G. Johnson, The NLopt nonlinear-optimization package, https://github.com/stevengj/nlopt

See Also

`nloptr.print.options` `check.derivatives` `optim` `nlm` `nlminb` `Rsolnp::Rsolnp` `Rsolnp::solnp`

Examples

```
library('nloptr')

## Rosenbrock Banana function and gradient in separate functions
eval_f <- function(x) {
  return(100 * (x[2] - x[1] * x[1])^2 + (1 - x[1])^2)
}

eval_grad_f <- function(x) {
  return(c(-400 * x[1] * (x[2] - x[1] * x[1]) - 2 * (1 - x[1]),
         200 * (x[2] - x[1] * x[1])))
}


# initial values
x0 <- c(-1.2, 1)

opts <- list("algorithm"="NLOPT_LD_LBFGS",
       "xtol_rel"=1.0e-8)
```

```
# solve Rosenbrock Banana function
res <- nloptr(x0=x0,
       eval_f=eval_f,
       eval_grad_f=eval_grad_f,
       opts=opts)
print(res)


## Rosenbrock Banana function and gradient in one function
# this can be used to economize on calculations
eval_f_list <- function(x) {
  return(
  list(
    "objective" = 100 * (x[2] - x[1] * x[1]) ^ 2 + (1 - x[1]) ^ 2,
    "gradient"  = c(-400 * x[1] * (x[2] - x[1] * x[1]) - 2 * (1 - x[1]),
            200 * (x[2] - x[1] * x[1]))))
}

# solve Rosenbrock Banana function using an objective function that
# returns a list with the objective value and its gradient
res <- nloptr(x0=x0,
       eval_f=eval_f_list,
       opts=opts)
print(res)



# Example showing how to solve the problem from the NLopt tutorial.
#
# min sqrt(x2)
# s.t. x2 >= 0
#    x2 >= (a1*x1 + b1)^3
#    x2 >= (a2*x1 + b2)^3
# where
# a1 = 2, b1 = 0, a2 = -1, b2 = 1
#
# re-formulate constraints to be of form g(x) <= 0
#    (a1*x1 + b1)^3 - x2 <= 0
#    (a2*x1 + b2)^3 - x2 <= 0

library('nloptr')


# objective function
eval_f0 <- function(x, a, b) {
  return(sqrt(x[2]))
}

# constraint function
eval_g0 <- function(x, a, b) {
  return((a*x[1] + b)^3 - x[2])
}
```

```
# gradient of objective function
eval_grad_f0 <- function(x, a, b) {
  return(c(0, .5/sqrt(x[2])))
}

# Jacobian of constraint
eval_jac_g0 <- function(x, a, b) {
  return(rbind(c(3*a[1]*(a[1]*x[1] + b[1])^2, -1.0),
         c(3*a[2]*(a[2]*x[1] + b[2])^2, -1.0)))
}


# functions with gradients in objective and constraint function
# this can be useful if the same calculations are needed for
# the function value and the gradient
eval_f1 <- function(x, a, b) {
  return(list("objective"=sqrt(x[2]),
         "gradient"=c(0,.5/sqrt(x[2]))))
}

eval_g1 <- function(x, a, b) {
  return(list("constraints"=(a*x[1] + b)^3 - x[2],
          "jacobian"=rbind(c(3*a[1]*(a[1]*x[1] + b[1])^2, -1.0),
                  c(3*a[2]*(a[2]*x[1] + b[2])^2, -1.0))))
}


# define parameters
a <- c(2,-1)
b <- c(0, 1)

# Solve using NLOPT_LD_MMA with gradient information supplied in separate
# function.
res0 <- nloptr(x0=c(1.234,5.678),
        eval_f=eval_f0,
        eval_grad_f=eval_grad_f0,
        lb = c(-Inf,0),
        ub = c(Inf,Inf),
        eval_g_ineq = eval_g0,
        eval_jac_g_ineq = eval_jac_g0,
        opts = list("algorithm"="NLOPT_LD_MMA"),
        a = a,
        b = b)
print(res0)

# Solve using NLOPT_LN_COBYLA without gradient information
res1 <- nloptr(x0=c(1.234,5.678),
        eval_f=eval_f0,
        lb = c(-Inf, 0),
        ub = c(Inf, Inf),
        eval_g_ineq = eval_g0,
        opts = list("algorithm" = "NLOPT_LN_COBYLA"),
```

```
        a = a,
        b = b)
print(res1)


# Solve using NLOPT_LD_MMA with gradient information in objective function
res2 <- nloptr(x0=c(1.234, 5.678),
        eval_f=eval_f1,
        lb = c(-Inf, 0),
        ub = c(Inf, Inf),
        eval_g_ineq = eval_g1,
        opts = list("algorithm"="NLOPT_LD_MMA",
              "check_derivatives" = TRUE),
        a = a,
        b = b)
print(res2)
```

`nloptr.get.default.options` *Return a data.frame with all the options that can be supplied to nloptr.*

Description

This function returns a data.frame with all the options that can be supplied to `nloptr`. The data.frame contains the default values of the options and an explanation. A user-friendly way to show these options is by using the function `nloptr.print.options`.

Usage

```
nloptr.get.default.options()
```

Value

The return value contains a `data.frame` with the following elements

`name`	name of the option
`type`	type (numeric, logical, integer, character)
`possible_values`	string explaining the values the option can take
`default`	default value of the option (as a string)
`is_termination_condition`	is this option part of the termination conditions?
`description`	description of the option (taken from NLopt website if it's an option that is passed on to NLopt).

Author(s)

Jelmer Ypma

See Also

nloptr nloptr.print.options

---

nloptr.print.options *Print description of nloptr options*

Description

This function prints a list of all the options that can be set when solving a minimization problem using nloptr.

Usage

```
nloptr.print.options(opts.show = NULL, opts.user = NULL)
```

Arguments

opts.show	list or vector with names of options. A description will be shown for the options in this list. By default, a description of all options is shown.
opts.user	object containing user supplied options. This argument is optional. It is used when nloptr.print.options is called from nloptr. In that case options are listed if print_options_doc is set to TRUE when passing a minimization problem to nloptr.

Author(s)

Jelmer Ypma

See Also

nloptr

Examples

```
library('nloptr')
nloptr.print.options()

nloptr.print.options(opts.show = c("algorithm", "check_derivatives"))

opts <- list("algorithm"="NLOPT_LD_LBFGS",
       "xtol_rel"=1.0e-8)
nloptr.print.options(opts.user = opts)
```

print.nloptr *Print results after running nloptr*

Description

This function prints the nloptr object that holds the results from a minimization using `nloptr`.

Usage

```
## S3 method for class 'nloptr'
print(x, show.controls = TRUE, ...)
```

Arguments

`x`	object containing result from minimization.
`show.controls`	Logical or vector with indices. Should we show the value of the control variables in the solution? If `show.controls` is a vector with indices, it is used to select which control variables should be shown. This can be useful if the model contains a set of parameters of interest and a set of nuisance parameters that are not of immediate interest.
`...`	further arguments passed to or from other methods.

Author(s)

Jelmer Ypma

See Also

`nloptr`

sbplx *Subplex Algorithm*

Description

Subplex is a variant of Nelder-Mead that uses Nelder-Mead on a sequence of subspaces.

Usage

```
sbplx(
  x0,
  fn,
  lower = NULL,
  upper = NULL,
  nl.info = FALSE,
  control = list(),
  ...
)
```

Arguments

x0	starting point for searching the optimum.
fn	objective function that is to be minimized.
lower, upper	lower and upper bound constraints.
nl.info	logical; shall the original NLopt info been shown.
control	list of options, see nl.opts for help.
...	additional arguments passed to the function.

Details

SUBPLEX is claimed to be much more efficient and robust than the original Nelder-Mead while retaining the latter's facility with discontinuous objectives.

This implementation has explicit support for bound constraints via the method in the Box paper as described on the neldermead help page.

Value

List with components:

par	the optimal solution found so far.
value	the function value corresponding to par.
iter	number of (outer) iterations, see maxeval.
convergence	integer code indicating successful completion (> 0) or a possible error number (< 0).
message	character string produced by NLopt and giving additional information.

Note

It is the request of Tom Rowan that reimplementations of his algorithm shall not use the name 'subplex'.

References

T. Rowan, "Functional Stability Analysis of Numerical Algorithms", Ph.D. thesis, Department of Computer Sciences, University of Texas at Austin, 1990.

See Also

subplex::subplex

Examples

```
# Fletcher and Powell's helic valley
fphv <- function(x)
  100*(x[3] - 10*atan2(x[2], x[1])/(2*pi))^2 +
    (sqrt(x[1]^2 + x[2]^2) - 1)^2 +x[3]^2
x0 <- c(-1, 0, 0)
sbplx(x0, fphv)  #  1 0 0

# Powell's Singular Function (PSF)
psf <- function(x)  (x[1] + 10*x[2])^2 + 5*(x[3] - x[4])^2 +
          (x[2] - 2*x[3])^4 + 10*(x[1] - x[4])^4
x0 <- c(3, -1, 0, 1)
sbplx(x0, psf, control = list(maxeval = Inf, ftol_rel = 1e-6)) #  0 0 0 0 (?)
```

---

slsqp *Sequential Quadratic Programming (SQP)*

Description

Sequential (least-squares) quadratic programming (SQP) algorithm for nonlinearly constrained, gradient-based optimization, supporting both equality and inequality constraints.

Usage

```
slsqp(
  x0,
  fn,
  gr = NULL,
  lower = NULL,
  upper = NULL,
  hin = NULL,
  hinjac = NULL,
  heq = NULL,
  heqjac = NULL,
  nl.info = FALSE,
  control = list(),
  deprecatedBehavior = TRUE,
  ...
)
```

Arguments

x0	starting point for searching the optimum.
fn	objective function that is to be minimized.
gr	gradient of function fn; will be calculated numerically if not specified.
lower, upper	lower and upper bound constraints.

hin	function defining the inequality constraints, that is hin <= 0 for all components. This is new behavior in line with the rest of the nloptr arguments. To use the old behavior, please set deprecatedBehavior to TRUE.
hinjac	Jacobian of function hin; will be calculated numerically if not specified.
heq	function defining the equality constraints, that is heq = 0 for all components.
heqjac	Jacobian of function heq; will be calculated numerically if not specified.
nl.info	logical; shall the original NLopt info been shown.
control	list of options, see nl.opts for help.
deprecatedBehavior	logical; if TRUE (default for now), the old behavior of the Jacobian function is used, where the equality is ≥ 0 instead of ≤ 0. This will be reversed in a future release and eventually removed.
...	additional arguments passed to the function.

Details

The algorithm optimizes successive second-order (quadratic/least-squares) approximations of the objective function (via BFGS updates), with first-order (affine) approximations of the constraints.

Value

List with components:

par	the optimal solution found so far.
value	the function value corresponding to par.
iter	number of (outer) iterations, see maxeval.
convergence	integer code indicating successful completion (> 1) or a possible error number (< 0).
message	character string produced by NLopt and giving additional information.

Note

See more infos at https://nlopt.readthedocs.io/en/latest/NLopt_Algorithms/.

Author(s)

Hans W. Borchers

References

Dieter Kraft, "A software package for sequential quadratic programming", Technical Report DFVLR-FB 88-28, Institut fuer Dynamik der Flugsysteme, Oberpfaffenhofen, July 1988.

See Also

alabama::auglag, Rsolnp::solnp, Rdonlp2::donlp2

Examples

```
##  Solve the Hock-Schittkowski problem no. 100 with analytic gradients
##  See https://apmonitor.com/wiki/uploads/Apps/hs100.apm

x0.hs100 <- c(1, 2, 0, 4, 0, 1, 1)
fn.hs100 <- function(x) {(x[1] - 10) ^ 2 + 5 * (x[2] - 12) ^ 2 + x[3] ^ 4 +
                         3 * (x[4] - 11) ^ 2 + 10 * x[5] ^ 6 + 7 * x[6] ^ 2 +
                         x[7] ^ 4 - 4 * x[6] * x[7] - 10 * x[6] - 8 * x[7]}

hin.hs100 <- function(x) {c(
2 * x[1] ^ 2 + 3 * x[2] ^ 4 + x[3] + 4 * x[4] ^ 2 + 5 * x[5] - 127,
7 * x[1] + 3 * x[2] + 10 * x[3] ^ 2 + x[4] - x[5] - 282,
23 * x[1] + x[2] ^ 2 + 6 * x[6] ^ 2 - 8 * x[7] - 196,
4 * x[1] ^ 2 + x[2] ^ 2 - 3 * x[1] * x[2] + 2 * x[3] ^ 2 + 5 * x[6] -
 11 * x[7])
}

S <- slsqp(x0.hs100, fn = fn.hs100,   # no gradients and jacobians provided
     hin = hin.hs100,
     nl.info = TRUE,
     control = list(xtol_rel = 1e-8, check_derivatives = TRUE),
     deprecatedBehavior = FALSE)

##  The optimum value of the objective function should be 680.6300573
##  A suitable parameter vector is roughly
##  (2.330, 1.9514, -0.4775, 4.3657, -0.6245, 1.0381, 1.5942)

S
```

---

stogo *Stochastic Global Optimization*

Description

STOGO is a global optimization algorithm that works by systematically dividing the search space—which must be bound-constrained—into smaller hyper-rectangles via a branch-and-bound technique, and searching them using a gradient-based local-search algorithm (a BFGS variant), optionally including some randomness.

Usage

```
stogo(
  x0,
  fn,
  gr = NULL,
  lower = NULL,
  upper = NULL,
  maxeval = 10000,
```

```
  xtol_rel = 1e-06,
  randomized = FALSE,
  nl.info = FALSE,
  ...
)
```

Arguments

x0	initial point for searching the optimum.
fn	objective function that is to be minimized.
gr	optional gradient of the objective function.
lower, upper	lower and upper bound constraints.
maxeval	maximum number of function evaluations.
xtol_rel	stopping criterion for relative change reached.
randomized	logical; shall a randomizing variant be used?
nl.info	logical; shall the original NLOPT info be shown.
...	additional arguments passed to the function.

Value

List with components:

par	the optimal solution found so far.
value	the function value corresponding to par.
iter	number of (outer) iterations, see maxeval.
convergence	integer code indicating successful completion (> 0) or a possible error number (< 0).
message	character string produced by NLOPT and giving additional information.

Note

Only bounds-constrained problems are supported by this algorithm.

Author(s)

Hans W. Borchers

References

S. Zertchaninov and K. Madsen, "A C++ Programme for Global Optimization," IMM-REP-1998-04, Department of Mathematical Modelling, Technical University of Denmark.

Examples

```
## Rosenbrock Banana objective function

rbf <- function(x) {(1 - x[1]) ^ 2 + 100 * (x[2] - x[1] ^ 2) ^ 2}

x0 <- c(-1.2, 1)
lb <- c(-3, -3)
ub <- c(3,  3)

## The function as written above has a minimum of 0 at (1, 1)

stogo(x0 = x0, fn = rbf, lower = lb, upper = ub)
```

---

tnewton *Preconditioned Truncated Newton*

Description

Truncated Newton methods, also called Newton-iterative methods, solve an approximating Newton system using a conjugate-gradient approach and are related to limited-memory BFGS.

Usage

```
tnewton(
  x0,
  fn,
  gr = NULL,
  lower = NULL,
  upper = NULL,
  precond = TRUE,
  restart = TRUE,
  nl.info = FALSE,
  control = list(),
  ...
)
```

Arguments

x0	starting point for searching the optimum.
fn	objective function that is to be minimized.
gr	gradient of function fn; will be calculated numerically if not specified.
lower, upper	lower and upper bound constraints.
precond	logical; preset L-BFGS with steepest descent.
restart	logical; restarting L-BFGS with steepest descent.
nl.info	logical; shall the original NLopt info been shown.
control	list of options, see nl.opts for help.
...	additional arguments passed to the function.

Details

Truncated Newton methods are based on approximating the objective with a quadratic function and applying an iterative scheme such as the linear conjugate-gradient algorithm.

Value

List with components:

`par`	the optimal solution found so far.
`value`	the function value corresponding to `par`.
`iter`	number of (outer) iterations, see `maxeval`.
`convergence`	integer code indicating successful completion (> 1) or a possible error number (< 0).
`message`	character string produced by NLopt and giving additional information.

Note

Less reliable than Newton's method, but can handle very large problems.

Author(s)

Hans W. Borchers

References

R. S. Dembo and T. Steihaug, "Truncated Newton algorithms for large-scale optimization," Math. Programming 26, p. 190-212 (1982).

See Also

lbfgs

Examples

```
flb <- function(x) {
  p <- length(x)
  sum(c(1, rep(4, p - 1)) * (x - c(1, x[-p]) ^ 2) ^ 2)
}
# 25-dimensional box constrained: par[24] is *not* at boundary
S <- tnewton(rep(3, 25L), flb, lower = rep(2, 25L), upper = rep(4, 25L),
       nl.info = TRUE, control = list(xtol_rel = 1e-8))
## Optimal value of objective function:  368.105912874334
## Optimal value of controls: 2  ...  2  2.109093  4
```

varmetric *Shifted Limited-memory Variable-metric*

Description

Shifted limited-memory variable-metric algorithm.

Usage

```
varmetric(
  x0,
  fn,
  gr = NULL,
  rank2 = TRUE,
  lower = NULL,
  upper = NULL,
  nl.info = FALSE,
  control = list(),
  ...
)
```

Arguments

x0	initial point for searching the optimum.
fn	objective function to be minimized.
gr	gradient of function fn; will be calculated numerically if not specified.
rank2	logical; if true uses a rank-2 update method, else rank-1.
lower, upper	lower and upper bound constraints.
nl.info	logical; shall the original NLopt info been shown.
control	list of control parameters, see nl.opts for help.
...	further arguments to be passed to the function.

Details

Variable-metric methods are a variant of the quasi-Newton methods, especially adapted to large-scale unconstrained (or bound constrained) minimization.

Value

List with components:

par	the optimal solution found so far.
value	the function value corresponding to par.
iter	number of (outer) iterations, see maxeval.
convergence	integer code indicating successful completion (> 0) or a possible error number (< 0).
message	character string produced by NLopt and giving additional information.

Note

Based on L. Luksan's Fortran implementation of a shifted limited-memory variable-metric algorithm.

Author(s)

Hans W. Borchers

References

J. Vlcek and L. Luksan, "Shifted limited-memory variable metric methods for large-scale unconstrained minimization," J. Computational Appl. Math. 186, p. 365-390 (2006).

See Also

lbfgs

Examples

```
flb <- function(x) {
  p <- length(x)
  sum(c(1, rep(4, p-1)) * (x - c(1, x[-p])^2)^2)
}
# 25-dimensional box constrained: par[24] is *not* at the boundary
S <- varmetric(rep(3, 25), flb, lower=rep(2, 25), upper=rep(4, 25),
     nl.info = TRUE, control = list(xtol_rel=1e-8))
## Optimal value of objective function:  368.105912874334
## Optimal value of controls: 2  ...  2  2.109093  4
```

Index