Package ‘nloptr’

March 11, 2020

Type Package

Title R Interface to NLopt

Version 1.2.2.1

Date 2020-03-11

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 <http://ab-initio.mit.edu/wiki/index.php/NLopt_Introduction> for more information on the available algorithms. During installation of nloptr on Unix-based systems, the installer checks whether the NLopt library is installed on the system. If the NLopt library cannot be found, the code is compiled using the NLopt source included in the nloptr package.

SystemRequirements A system installation of NLopt >= 2.4.0 (with headers) will be used if available.

License LGPL-3

Suggests testthat (>= 0.8.1), knitr, rmarkdown, inline (>= 0.3.14)

LazyLoad yes

NeedsCompilation yes

VignetteBuilder knitr

RoxygenNote 6.1.0

Author Jelmer Ypma [aut, cre],
  Steven G. Johnson [aut] (author of the NLopt C library),
  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] (removal deprecated calls from tests,
  <https://orcid.org/0000-0002-3039-0703>)

Maintainer Jelmer Ypma <uctpjyy@ucl.ac.uk>

Repository CRAN

Date/Publication 2020-03-11 16:31:22 UTC
**R topics documented:**

- nloptr-package ........................................ 2
- auglag .................................................. 5
- bobyqa .................................................. 8
- ccsaq ................................................... 9
- check.derivatives ...................................... 11
- cobyla .................................................. 13
- crs2lm .................................................. 14
- direct .................................................... 16
- is.nloptr ............................................... 19
- isres ................................................... 19
- lbfgs .................................................... 21
- mlsl ..................................................... 23
- mma ....................................................... 25
- neldermead ............................................. 27
- newuoa .................................................. 29
- nl.grad .................................................. 30
- nl.opts ................................................ 31
- nloptr .................................................. 32
- nloptr.get.default.options ......................... 37
- nloptr.print.options ................................ 38
- print.nloptr ......................................... 39
- sbplx ................................................... 39
- slsqp .................................................... 41
- stogo .................................................... 43
- tnewton ............................................... 45
- varmetric .............................................. 46

**Index** .................................................................. 48

---

**nloptr-package**  
*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.
Details

NLopt addresses general nonlinear optimization problems of the form:
min f(x) \quad x \in \mathbb{R}^n
s.t. g(x) <= 0 \quad h(x) = 0 \quad lb <= x <= ub

where f 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.

An optimization problem can be solved with the general nloptr interface, or using one of the wrapper functions for the separate algorithms: auglag, bobyqa, cobyla, crs2lm, direct, lbfgs, mls1, mma, neldermead, newuoa, sbplx, slsqp, stogo, tnewton, varmetric.

Note

See ?nloptr for more examples.

Author(s)

Steven G. Johnson and others (C code)
Jelmer Ypma (R interface)
Hans W. Borchers (wrappers)

References


See Also

optim nlm nlmib Rsolnp::Rsolnp Rsolnp::solnp nloptr auglag bobyqa cobyla crs2lm direct isres lbfgs mls1 mma neldermead newuoa sbplx slsqp stogo tnewton varmetric

Examples

# Example problem, number 71 from the Hock-Schittkowski test suite.
#
# \min_{x} x_1x_4*(x_1 + x_2 + x_3) + x_3
# s.t.
# \( x_1 x_2 x_3 x_4 \geq 25 \)
# \( x_1^2 + x_2^2 + x_3^2 + x_4^2 = 40 \)
# \( 1 \leq x_1, x_2, x_3, x_4 \leq 5 \)

# we re-write the inequality as
# \( 25 - x_1 x_2 x_3 x_4 \leq 0 \)

# and the equality as
# \( x_1^2 + x_2^2 + x_3^2 + x_4^2 - 40 = 0 \)

# \( x_0 = (1, 5, 5, 1) \)

# optimal solution = (1.00000000, 4.74299963, 3.82114998, 1.37940829)

library('nloptr')

# f(x) = x_1 x_4 (x_1 + x_2 + x_3) + x_3
#
# eval_f <- function( x ) {
#                                  x[1] * x[4],
#                                  x[1] * x[4] + 1.0,
# }

# constraint functions
# inequalities
# eval_g_ineq <- function( x ) {
#   grad <- c( -x[2] * x[3] * x[4],
#              -x[1] * x[3] * x[4],
#              -x[1] * x[2] * x[4],
#   return( list( "constraints"=constr, "jacobian"=grad ) )
# }

# equalities
# eval_g_eq <- function( x ) {
#   grad <- c( 2.0 * x[1],
#              2.0 * x[2],
#              2.0 * x[3],
#              2.0 * x[4] )
#   return( list( "constraints"=constr, "jacobian"=grad ) )
# }

# initial values
auglag

x0 <- c( 1, 5, 5, 1 )

# lower and upper bounds of control
lb <- c( 1, 1, 1, 1 )
ub <- c( 5, 5, 5, 5 )

local_opts <- list( "algorithm" = "NLOPT_LD_MMA",
                    "xtol_rel" = 1.0e-7 )
opts <- list( "algorithm" = "NLOPT_LD_AUGLAG",
              "xtol_rel" = 1.0e-7,
              "maxeval" = 1000,
              "local_opts" = local_opts )

res <- nloptr( x0=x0,
               eval_f=eval_f,
               lb=lb,
               ub=ub,
               eval_g_ineq=eval_g_ineq,
               eval_g_eq=eval_g_eq,
               opts=opts)

print( res )

---

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 = c("COBYLA"), localtol = 1e-06, ineq2local = FALSE,
       nl.info = FALSE, control = list(), ...)```

**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.
... 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

See Also
alabama::auglag, Rsolnp::solnp

Examples

```r
x0 <- c(1, 1)
fn <- function(x) { x[1]^2 + x[2]^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)

auglag(x0, fn, gr = NULL, hin = hin, heq = heq) # with COBYLA
# $par: 0.8228761 0.9114382
# $value: 1.393464
# $iter: 1001

auglag(x0, fn, gr = NULL, hin = hin, heq = heq, localsolver = "SLSQP")
# $par: 0.8228757 0.9114378
# $value: 1.393465
# $iter: 173

## Example from the alabama::auglag help page
hin <- function(x) { c(6*x[2] + 4*x[3] - x[1]^3 - 3, x[1], x[2], x[3]) }

auglag(runif(3), fn, hin = hin, heq = heq, localsolver="lbfgs")
# $par: 2.380000e-09 1.086082e-14 1.000000e+00
# $value: 1
# $iter: 289

## Powell problem from the Rsolnp::solnp help page
x0 <- c(-2, 2, 2, -1, -1)
fn1 <- function(x) { exp(x[1]*x[2]*x[3]*x[4]*x[5]) }
bound Optimization by Quadratic Approximation

Description

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

Usage

```r
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 been 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


See Also

cobyla, newuoa

Examples

```r
fr <- function(x) {  # Rosenbrock Banana function
  100 * (x[2] - x[1]^2)^2 + (1 - x[1])^2
}
(S <- bobyqa(c(0, 0, 0), fr, lower = c(0, 0, 0), upper = c(0.5, 0.5, 0.5)))
```

---

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(), ...)`

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.
... 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

See Also
mma

Examples
## Solve the Hock-Schittkowski problem no. 100 with analytic gradients
x0.hs100 <- c(1, 2, 0, 4, 0, 1, 1)
fn.hs100 <- function(x) {
}
hin.hs100 <- function(x) {
  h <- numeric(4)
  return(h)
}
gr.hs100 <- function(x) {
  c( 2 * x[1] - 20,
      10 * x[2] - 120,
      4 * x[3]*3,
Check analytic gradients of a function using finite difference approximations

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

Usage

```r
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`. 
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 ) {
  return( sum( ( x - a )^2 ) )
}

f_grad <- function( x, a ) {
  return( 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 ) {
  return( 2*( x - a ) + c(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 ) {
  return( c( sum(x-a), sum( (x-a)^2 ) ) )
}

g_grad <- function( x, a ) {
  return( rbind( rep(1,length(x)) + c(0,.01,rep(0,8)), 2*(x-a) + c(0,1,rep(0,8)) ) )
}

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

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

**Usage**

```r
cobyla(x0, fn, lower = NULL, upper = NULL, hin = 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.
- `hin`: function defining the inequality constraints, that is `hin >= 0` for all components.
- `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**

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. As this does not give full satisfaction with the implementation in NLOPT, it has not been made available here.

**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


See Also

bobyqa, newuoa

Examples

### Solve Hock-Schittkowski no. 100

```r
x0.hs100 <- c(1, 2, 0, 4, 0, 1, 1)
fn.hs100 <- function(x) {
}

hin.hs100 <- function(x) {
  h <- numeric(4)
  return(h)
}

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

## Optimal value of objective function: 680.630057374431
```

---

**crs2lm**

*Controlled Random Search*

**Description**

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

**Usage**

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

- \( x_0 \): initial point for searching the optimum.
- \( f_n \): 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 been 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 \( 10 \times (n+1) \) in \( n \) dimensions, but this can be changed; the initial population must be at least \( n+1 \).

References


Examples

```r
### Minimize the Hartmann6 function
hartmann6 <- function(x) {
  n <- length(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 <- 0.0
  for (i in 1:4) {
    fun <- fun - a[i] * exp(-sum(A[i,]*(x-B[i,])^2))
  }
  return(fun)
}

S <- mls1(x0 = rep(0, 6), hartmann6, lower = rep(0,6), upper = rep(1,6),
          nl.info = TRUE, control=list(xtol_rel=1e-8, maxeval=1000))
## Number of Iterations....: 4050
## Termination conditions: maxeval: 10000 xtol_rel: 1e-06
## Number of inequality constraints: 0
## Number of equality constraints: 0
## Optimal value of objective function: -3.32236801141328
## Optimal value of controls:
## 0.2016893 0.1500105 0.4768738 0.2753326 0.3116516 0.6573004
```

**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


See Also

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

Examples

```r
### Minimize the Hartmann6 function
hartmann6 <- function(x) {
  n <- length(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 <- 0.0
  for (i in 1:4) {
    fun <- fun - a[i] * exp(-sum(A[i,]*(x-B[i,])^2))
  }
  return(fun)
}
S <- directL(hartmann6, rep(0,6), rep(1,6),
             nl.info = TRUE, control=list(xtol_rel=1e-8, maxeval=1000))
## Number of Iterations.....: 500
## Termination conditions: stopval: -Inf
## xtol_rel: 1e-08, maxeval: 500, 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) 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, ...)

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 $hin \geq 0$ 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 been shown.
- **...** 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 bound 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 $20 \times (n+1)$ in $n$ dimensions, but this can be changed; the initial population must be at least $n+1$.

Author(s)

Hans W. Borchers

References

### Rosenbrock Banana objective function

```r
fn <- function(x)
    return( 100 * (x[2] - x[1] * x[1])^2 + (1 - x[1])^2 )
```

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

```r
isres(x0 = x0, fn = fn, lower = lb, upper = ub)
```

---

**lbfgs**

*Low-storage BFGS*

**Description**

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

**Usage**

```r
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


See Also

`optim`

Examples

```r
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**

```r
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 been 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 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 L-BFGS 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 \texttt{ftol\_rel=1e-15} and \texttt{xtol\_rel=1e-7} for the local searches.

Author(s)

Hans W. Borchers

References


See Also

direct

Examples

```r
### Minimize the Hartmann6 function
hartmann6 <- function(x) {
  n <- length(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 <- 0.0
  for (i in 1:4) {
    fun <- fun - a[i] * exp(-sum(A[i,]*(x-B[i,])^2))
  }
  return(fun)
}
S <- mlsl(x0 = rep(0, 6), hartmann6, lower = rep(0,6), upper = rep(1,6),
mma

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: -3.32236801141552
## Current value of controls:
## 0.2016895 0.1500107 0.476874 0.2753324 0.3116516 0.6573005

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(), ...)

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.
... 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. Note:
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.

Author(s)

Hans W. Borchers

References


See Also

`slsqp`

Examples

```r
## Solve the Hock-Schittkowski problem no. 100 with analytic gradients
x0.hs100 <- c(1, 2, 0, 4, 0, 1, 1)
fn.hs100 <- function(x) {
}
hin.hs100 <- function(x) {
  h <- numeric(4)
  return(h)
}
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,  

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), 4, 7, byrow=TRUE)
}

# incorrect result with exact jacobian
S <- mma(x0.hs100, fn.hs100, gr = gr.hs100,  
in = hin.hs100, hinjac = hinjac.hs100,  
nl.info = TRUE, control = list(xtol_rel = 1e-8))

# This example is put in donttest because it runs for more than  
# 40 seconds under 32-bit Windows. The difference in time needed  
# to execute the code between 32-bit Windows and 64-bit Windows  
# can probably be explained by differences in rounding/truncation  
# on the different systems. On Windows 32-bit more iterations  
# are needed resulting in a longer runtime.  
# correct result with inexact jacobian
S <- mma(x0.hs100, fn.hs100, hin = hin.hs100,  
nl.info = TRUE, control = list(xtol_rel = 1e-8))

---

**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


See Also

dfoptim::nmk

Examples

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


See Also

bobyqa, cobyqa

Examples

fr <- function(x) {  # Rosenbrock Banana function
  100 * (x[2] - x[1]^2)^2 + (1 - x[1])^2
}
(S <- newuoa(c(1, 2), fr))

---

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

```r
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
```

Description

Sets and changes the NLOPT options.

Usage

```r
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. 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:

\[
\begin{align*}
\min \ f(x) & \ \text{in } \mathbb{R}^n \\
\text{s.t. } g(x) & \leq 0 \ h(x) = 0 \ lb \leq x \leq ub
\end{align*}
\]

where \( f \) 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, [http://ab-initio.mit.edu/nlopt](http://ab-initio.mit.edu/nlopt)

**See Also**

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

**Examples**

```r
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) {
}
```
# 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,
            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
#
library('nloptr')

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

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

# gradient of objective function

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

# jacobian of constraint

```r
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

```r
eval_f1 <- function( x, a, b ){
  return( list("objective"=sqrt(x[2]),
                     "gradient"=c(0,.5/sqrt(x[2])) ) )
}
```

```r
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

```r
a <- c(2,-1)
b <- c(0, 1)
```

# Solve using NLOPT_LD_MMA with gradient information supplied in separate function

```r
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

```r
res1 <- nloptr( x0=c(1.234,5.678),
                eval_f=eval_f0,
                nloptr
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

```r
## 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**

---

**sbpx**

Subplex Algorithm

### Description

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

### Usage

```r
sbpx(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


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 +
x0 <- c(-1, 0, 0)
spbx(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 +
x0 <- c(3, -1, 0, 1)
spbx(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(), ...)

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.
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.
... 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


Author(s)

Hans W. Borchers

References


See Also

- alabama::auglag
- Rsolnp::solnp
- Rdonlp2::donlp2

Examples

```r
## Solve the Hock-Schittkowski problem no. 100
x0.hs100 <- c(1, 2, 0, 4, 0, 1, 1)
fn.hs100 <- function(x) {
}

hin.hs100 <- function(x) {
  h <- numeric(4)
  return(h)
}
```
S <- slsqp(x0.hs100, fn = fn.hs100, # no gradients and jacobians provided
          hin = hin.hs100,
          control = list(xtol_rel = 1e-8, check_derivatives = TRUE))
S
## Optimal value of objective function: 690.622270249131 *** WRONG ***

# Even the numerical derivatives seem to be too tight.
# Let's try with a less accurate jacobian.

hinjac.hs100 <- function(x) nl.jacobian(x, hin.hs100, heps = 1e-2)
S <- slsqp(x0.hs100, fn = fn.hs100,
          hin = hin.hs100, hinjac = hinjac.hs100,
          control = list(xtol_rel = 1e-8))
S
## Optimal value of objective function: 680.630057392593 *** CORRECT ***

---

**stogo**

**Stochastic Global Optimization**

**Description**

StoGO is a global optimization algorithm that works by systematically dividing the search space into smaller hyper-rectangles.

**Usage**

```r
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 been shown.
- **...**: additional arguments passed to the function.
Details

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 by a gradient-based local-search algorithm (a BFGS variant), optionally including some randomness.

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 bound-constrained problems are supported by this algorithm.

Author(s)

Hans W. Borchers

References


Examples

```r
### Rosenbrock Banana objective function
fn <- function(x)
  return( 100 * (x[2] - x[1] * x[1])^2 + (1 - x[1])^2 )

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

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


See Also

lbfgs

Examples

```r
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, 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
```

**varmetric**

*Shifted Limited-memory Variable-metric*

Description

Shifted limited-memory variable-metric algorithm.

Usage

```r
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.
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


See Also

`lbfgs`

Examples

```r
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

∗Topic interface
  check.derivatives, 11
  is.nloptr, 19
  nloptr, 32
  nloptr-package, 2
  nloptr.get.default.options, 37
  nloptr.print.options, 38
  print.nloptr, 39

∗Topic optimize
  check.derivatives, 11
  is.nloptr, 19
  nloptr, 32
  nloptr-package, 2
  nloptr.get.default.options, 37
  nloptr.print.options, 38
  print.nloptr, 39
  optim, 3, 22, 34
  print.nloptr, 39
  sbplx, 3, 39
  slsqp, 3, 26, 41
  stogo, 3, 43
  tnewton, 3, 45
  varmetric, 3, 46

auglag, 3, 5
bobyqa, 3, 8, 14, 30
ccsaq, 9
check.derivatives, 11, 34
cobyla, 3, 9, 13, 30
crs2lm, 3, 14
direct, 3, 16, 24
directL (direct), 16
is.nloptr, 19
isres, 3, 19
lbfgs, 3, 21, 46, 47
mlsl, 3, 23
mma, 3, 10, 25
neldermead, 3, 27
newuoa, 3, 9, 14, 29
nl.grad, 30
nl.jacobian (nl.grad), 30
nl.opts, 31
nlm, 3, 34
nlminb, 3, 34
nloptr, 3, 12, 19, 32, 37–39
nloptr-package, 2
nloptr.get.default.options, 37
nloptr.print.options, 34, 37, 38, 38
optim, 3, 22, 34
print.nloptr, 39
sbplx, 3, 39
slsqp, 3, 26, 41
stogo, 3, 43
 tnewton, 3, 45
varmetric, 3, 46