Package ‘nloptr’
February 20, 2015

Type     Package
Title    R interface to NLopt
Version  1.0.4
Date     2014-08-02
Author   Jelmer Ypma <uctpjyy@ucl.ac.uk>, with contributions by
          Hans W. Borchers <hwbarchers@gmail.com> and
          Dirk Eddelbuettel <edd@debian.org>
Maintainer Jelmer Ypma <uctpjyy@ucl.ac.uk>

Description nloptr is 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 on Unix the
NLopt code is downloaded and compiled from the NLopt website.

License   LGPL-3
Suggests  testthat (>= 0.8.1)
LazyLoad  yes
NeedsCompilation yes
Repository CRAN
Date/Publication 2014-08-04 15:35:43

R topics documented:

  nloptr-package ........................................... 2
  auglag ..................................................... 5
  bobyqa ..................................................... 7
  check.derivatives ......................................... 9
  cobyla ...................................................... 10
  csr2lm ...................................................... 12
  DIRECT ..................................................... 14
  grad, jacobian ........................................... 16
nloptr-package

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.

NLopt addresses general nonlinear optimization problems of the form:

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

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), \( \text{lb} \) and \( \text{ub} \). 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, mls, mma, neldermead, newuoa, sbplx, slsqp, stogo, tnewton, varmetric.

Details
Package: nloptr
Type: Package
Version: 0.9.9
Date: 2013-11-22
License: L-GPL
LazyLoad: yes

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 nlminb Rsolnp ssolnp 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-Schittkowsky test suite.
#
# \min_{x} x1\times x4*(x1 + x2 + x3) + x3
# s.t.
# \ x1*x2*x3*x4 >= 25
# \ x1^2 + x2^2 + x3^2 + x4^2 = 40
# \ 1 <= x1,x2,x3,x4 <= 5
#
# we re-write the inequality as
# 25 - x1\times x2\times x3\times x4 <= 0
#
# and the equality as
# x1^2 + x2^2 + x3^2 + x4^2 - 40 = 0
#
# x0 = (1,5,5,1)
#
# optimal solution = (1.00000000, 4.74299963, 3.82114998, 1.37940829)
```r
library('nloptr')

# f(x) = x1*x4*(x1 + x2 + x3) + x3
#
eval_f <- function(x) {
  return(list(objective = x[1]*x[4]*(x[1] + x[2] + x[3]) + x[3],
                           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],
            -x[1]*x[2]*x[3])
  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
x0 <- c(1, 5, 5, 1)

# lower and upper bounds of control
lb <- c(1, 1, 1)
ub <- c(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,
              constr=eval_g_ineq,
              eq=eval_g_eq,
              lb=lb,
              ub=ub,
              opts=opts)
```

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-6, 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 if 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, LBGFS, 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) into 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.

References


See Also

alabama::auglag, Rsolnp::solnp
**Examples**

```r
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
g <- 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 1.080000e-14 1.000000e+00
# $value: 1
# $iter: 289

## Powell problem from the Rsolnp::solnp help page
x0 <- c(-2, 2, -1, -1)
fn1 <- function(x) exp(x[1]*x[2]*x[3]*x[4]*x[5])
eqn1 <- function(x)
x[2]*x[3]-5*x[4]*x[5],

auglag(x0, fn1, heq = eqn1, localsolver = "mma")
# $par: -3.988458e-10 -1.654201e-08 -3.752028e-10 8.904445e-10 8.926363e-10
# $value: 1
# $iter: 1001
```

---

**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 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
check.derivatives

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)))
```

Description

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`.

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

```r
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
h <- function(x, a) {
  return( c( sum(x-a), sum((x-a)^2) ) )
}

h_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=h, func_grad=h_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).
cobyla

Usage

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 nlopts 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.

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.

See Also

bobyqa, newuoa
Examples

```r
### Solve Hock-Schittkowski 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 <- cobyla(x0.hs100, fn.hs100, hin = hin.hs100,
  nl.info = TRUE, control = list(xtol_rel = 1e-8, maxeval = 2000))
```

---

### csr2lm

**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-6, 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 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 $10^x(n+1)$ 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,
                .6124, .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**

```r
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.

- `randomized`  
  logical; shall some randomization be used to decide which dimension to halve next in the case of near-ties.

- `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.
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.

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

---

**grad, jacobian**

**Numerical Gradients and Jacobians**

**Description**

Provides numerical gradients and jacobians.

**Usage**

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

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

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

Usage

```r
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-6, 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>=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 \( n+1 \).

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

**nlbfgs**

*Low-storage BFGS*

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

Usage
```r
nlbfgs(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.

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.opt` 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 ftol_rel=1e-15 and xtol_rel=1e-7 for the local searches.

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,.1084,.3847,.1091,
                .5886,.9991,.6650,.0331), 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(xs0 = 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....: 1000
# Termination conditions:
#  stopval: -Inf, xtol_rel: 1e-08, maxeval: 1000, ftol_rel: 0, ftol_abs: 0
# Number of inequality constraints: 0
```
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

```markdown
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.

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,
  hin = hin.hs100, hinjac = hinjac.hs100,
  nl.info = TRUE, control = list(xtol_rel = 1e-8))

# correct result with inexact jacobian
S <- mma(x0.hs100, fn.hs100, hin = hin.hs100,
  nl.info = TRUE, control = list(xtol_rel = 1e-8))
```
**Description**

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

**Usage**

```lisp
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.
References


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 +
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)
neldermead(x0, psf) # 0 0 0 0, needs maximum number of function calls

## Not run:
# Bounded version of Nelder-Mead
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)
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.

References


See Also

bobyqa, cobyla

Examples

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

Examples

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

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.

NLopt addresses general nonlinear optimization problems of the form:

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

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), \( \text{lb} \) and \( \text{ub} \). 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.

Usage

```r
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 "objective" 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 "objective" 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.

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

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 op 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


See Also

nloptr.print.options check.derivatives optim nlm nlminb Rsolnp ssolnp

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) {
  )
}

# 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.

```r
eval_f_list <- function(x) {
  return( list( "objective" = 100 * (x[2] - x[1] * x[1])^2 + (1 - x[1])^2,
}
```

# 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(x^2)
# s.t. x^2 >= 0
#     x^2 >= ( a1*x1 + b1 )^3
#     x^2 >= ( 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 - x^2 <= 0
#     ( a2*x1 + b2 )^3 - x^2 <= 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]*x[1]+a[1]*x[1]^2, -1.0 ),
                      c( 3*a[2]*x[1]+a[2]*x[1]^2, -1.0 ) )
)
C functions with gradients in objective and constraint function
C this can be useful if the same calculations are needed for
C the function value and the gradient

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

eval_g <- function(x, a, b)
{
  return( list("constraints"=(a*x[1] + b)^3 - x[2],
               "jacobian"=rbind( c( 3*a[1]*x[1] + b[1])^2, -1.0 ),
                    c( 3*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 \texttt{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 \texttt{nloptr.print.options}.

Usage

\texttt{nloptr.get.default.options()}

Value

The return value contains a \texttt{data.frame} with the following elements

- \texttt{name}: name of the option
- \texttt{type}: type (numeric, logical, integer, character)
- \texttt{possible_values}: string explaining the values the option can take
- \texttt{default}: default value of the option (as a string)
- \texttt{is_termination_condition}: is this option part of the termination conditions?
- \texttt{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

\texttt{nloptr nloptr.print.options}
nloptr.print.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

---

**sbplx**  

*Subplex Algorithm*

**Description**

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

**Usage**

```r
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.
- \( nl\text{.info} \) logical; shall the original NLopt info been shown.
- \( control \) list of options, see \texttt{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 \texttt{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 \texttt{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

\texttt{subplex::subplex}
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)
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 +
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**

```r
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


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) {
  (x[1]-10)^2 + 5*(x[2]-12)^2 + x[3]^4 + 3*(x[4]-11)^2 + 10*x[5]*6 +
}
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)
```
StoGO is a global optimization algorithm that works by systematically dividing the search space into smaller hyper-rectangles.

Usage

\[
\text{stogo}(x_0, \text{fn}, \text{gr} = \text{NULL}, \text{lower} = \text{NULL}, \text{upper} = \text{NULL}, \\
\text{maxeval} = 10000, \text{xtol\_rel} = 1e-6, \text{randomized} = \text{FALSE}, \\
\text{nl.info} = \text{FALSE}, \ldots)
\]

Arguments

- \(x_0\) initial point for searching the optimum.
- \(\text{fn}\) objective function that is to be minimized.
- \(\text{gr}\) optional gradient of the objective function.
- \(\text{lower, upper}\) lower and upper bound constraints.
- \(\text{maxeval}\) maximum number of function evaluations.
- \(\text{xtol\_rel}\) stopping criterion for relative change reached.
- \(\text{randomized}\) logical; shall a randomizing variant be used?
- \(\text{nl.info}\) logical; shall the original NLopt info been shown.
- \(\ldots\) 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:

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>par</td>
<td>the optimal solution found so far.</td>
</tr>
<tr>
<td>value</td>
<td>the function value corresponding to par.</td>
</tr>
<tr>
<td>iter</td>
<td>number of (outer) iterations, see maxeval.</td>
</tr>
<tr>
<td>convergence</td>
<td>integer code indicating successful completion (&gt; 0) or a possible error number (&lt; 0).</td>
</tr>
<tr>
<td>message</td>
<td>character string produced by NLopt and giving additional information.</td>
</tr>
</tbody>
</table>

Note

Only bound-constrained problems are supported by this algorithm.

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)
```

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

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

- \(x_0\): starting point for searching the optimum.
- \(f_n\): objective function that is to be minimized.
- \(g_r\): gradient of function \(f_n\); will be calculated numerically if not specified.
- \(\text{lower, upper}\): lower and upper bound constraints.
- \(\text{precond}\): logical; preset L-BFGS with steepest descent.
- \(\text{restart}\): logical; restarting L-BFGS with steepest descent.
- \(\text{n1.info}\): logical; shall the original NLopt info been shown.
- \(\text{control}\): list of options, see \(\text{n1.opts}\) 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:

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

Note

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

References


See Also

- \(\text{lbfgs}\)
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.

**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.

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 2.189093 4
```
Index

*Topic interface
  check.derivatives, 9
  is.nloptr, 17
  nloptr, 29
  nloptr-package, 2
  nloptr.get.default.options, 34
  nloptr.print.options, 35
  print.nloptr, 36

*Topic optimize
  check.derivatives, 9
  is.nloptr, 17
  nloptr, 29
  nloptr-package, 2
  nloptr.get.default.options, 34
  nloptr.print.options, 35
  print.nloptr, 36

auglag, 3, 5
bobyqa, 3, 7, 11, 27
check.derivatives, 9, 31
cobyla, 3, 8, 10, 27
crs2lm, 3
crs2lm(csr2lm), 12
csr2lm, 12

DIRECT, 14
direct, 3, 22
direct(DIRECT), 14
directl(DIRECT), 14

ggrad, jacobian, 16

is.nloptr, 17
isres, 3, 18

lbfgs, 3, 19, 42, 44

mlsl, 3, 21
mma, 3, 23

neldermead, 3, 25
newuoa, 3, 8, 11, 26
nl.grad(grad, jacobian), 16
nl.jacobian(grad, jacobian), 16
nl.opts(NLOpt options), 28
nlm, 3, 31
nlminb, 3, 31
NLOpt options, 28
nloptr, 3, 10, 17, 29, 34–36
nloptr-package, 2
nloptr.get.default.options, 34
nloptr.print.options, 31, 34, 35

optim, 3, 20, 31

print.nloptr, 36

Rsolnp, 3, 31

sbplx, 3, 36
slsqp, 3, 24, 38
ssolnp, 3, 31
stogo, 3, 40

tnewton, 3, 41

varmetric, 3, 43