Package 'Rsolnp'

October 12, 2022

Type Package

Title General Non-Linear Optimization

Version 1.16

Date 2015-07-02

Author Alexios Ghalanos and Stefan Theussl

Maintainer Alexios Ghalanos <alexios@4dscape.com>

Depends R (>= 2.10.0)

Imports truncnorm, parallel, stats

Description General Non-linear Optimization Using Augmented Lagrange Multiplier Method.

LazyLoad yes

License GPL

Repository CRAN

Repository/R-Forge/Project rino

Repository/R-Forge/Revision 102

Repository/R-Forge/DateTimeStamp 2015-07-04 02:08:32

Date/Publication 2015-12-28 09:01:11

NeedsCompilation no

R topics documented:

Rsolnp-package *The Rsolnp package*

Description

The Rsolnp package implements Y.Ye's general nonlinear augmented Lagrange multiplier method solver (SQP based solver).

Details

Package:	Rsolnp
Type:	Package
Version:	1.15
Date:	2013-04-10
License:	GPL
LazyLoad:	yes
Depends:	stats, truncnorm, parallel

Author(s)

Alexios Ghalanos and Stefan Theussl

References

Y.Ye, *Interior algorithms for linear, quadratic, and linearly constrained non linear programming*, PhD Thesis, Department of EES Stanford University, Stanford CA.

benchmark *The Rsolnp Benchmark Problems Suite.*

Description

The function implements a set of benchmark problems against the MINOS solver of Murtagh and Saunders.

Usage

```
benchmark(id = "Powell")
```

Arguments

id	The name of the benchmark problem. A call to the function benchmarkids will return the available benchmark problems.

Details

The benchmarks were run on dual xeon server with 24GB of memory and windows 7 operating system. The MINOS solver was used via the tomlab interface.

Value

A data.frame containing the benchmark data. The description of the benchmark problem can be accessed throught the `description` attribute of the data.frame.

Author(s)

Alexios Ghalanos and Stefan Theussl
Y.Ye (original matlab version of solnp)

References

W.Hock and K.Schittkowski, *Test Examples for Nonlinear Programming Codes*, Lecture Notes in Economics and Mathematical Systems. Springer Verlag, 1981.
Y.Ye, *Interior algorithms for linear, quadratic, and linearly constrained non linear programming*, PhD Thesis, Department of EES Stanford University, Stanford CA.
B.A.Murtagh and M.A.Saunders, *MINOS 5.5 User's Guide, Report SOL 83-20R*, Systems Optimization Laboratory, Stanford University (revised July 1998).
P. E. Gill, W. Murray, and M. A. Saunders, *SNOPT An SQP algorithm for large-scale constrained optimization*, SIAM J. Optim., 12 (2002), pp.979-1006.

Examples

```
## Not run:
benchmarkids()
benchmark(id = "Powell")
benchmark(id = "Alkylation")
benchmark(id = "Box")
benchmark(id = "RosenSuzuki")
benchmark(id = "Wright4")
benchmark(id = "Wright9")
benchmark(id = "Electron")
benchmark(id = "Permutation")
# accessing the description
test = benchmark(id = "Entropy")
attr(test, "description")

## End(Not run)
```

benchmarkids *The Rsolnp Benchmark Problems Suite problem id's.*

Description

Returns the id's of available benchmark in the Rsolnp Benchmark Problems Suite.

Usage

```
benchmarkids()
```

Value

A character vector of problem id's.

Author(s)

Alexios Ghalanos and Stefan Theussl
Y.Ye (original matlab version of solnp)

References

W.Hock and K.Schittkowski, *Test Examples for Nonlinear Programming Codes*, Lecture Notes in Economics and Mathematical Systems. Springer Verlag, 1981.
Y.Ye, *Interior algorithms for linear, quadratic, and linearly constrained non linear programming*, PhD Thesis, Department of EES Stanford University, Stanford CA.

Examples

```
benchmarkids()
```

gosolnp *Random Initialization and Multiple Restarts of the solnp solver.*

Description

When the objective function is non-smooth or has many local minima, it is hard to judge the optimality of the solution, and this usually depends critically on the starting parameters. This function enables the generation of a set of randomly chosen parameters from which to initialize multiple restarts of the solver (see note for details).

Usage

```
gosolnp(pars = NULL, fixed = NULL, fun, eqfun = NULL, eqB = NULL, ineqfun = NULL,
ineqLB = NULL, ineqUB = NULL, LB = NULL, UB = NULL, control = list(),
distr = rep(1, length(LB)), distr.opt = list(), n.restarts = 1, n.sim = 20000,
cluster = NULL, rseed = NULL, ...)
```

Arguments

pars	The starting parameter vector. This is not required unless the fixed option is also used.
fixed	The numeric index which indicates those parameters which should stay fixed instead of being randomly generated.
fun	The main function which takes as first argument the parameter vector and returns a single value.
eqfun	(Optional) The equality constraint function returning the vector of evaluated equality constraints.
eqB	(Optional) The equality constraints.
ineqfun	(Optional) The inequality constraint function returning the vector of evaluated inequality constraints.
ineqLB	(Optional) The lower bound of the inequality constraints.
ineqUB	(Optional) The upper bound of the inequality constraints.
LB	The lower bound on the parameters. This is not optional in this function.
UB	The upper bound on the parameters. This is not optional in this function.
control	(Optional) The control list of optimization parameters. The eval.type option in this control list denotes whether to evaluate the function as is and exclude inequality violations in the final ranking (default, value = 1), else whether to evaluate a penalty barrier function comprised of the objective and all constraints (value = 2). See solnp function documentation for details of the remaining control options.
distr	A numeric vector of length equal to the number of parameters, indicating the choice of distribution to use for the random parameter generation. Choices are uniform (1), truncated normal (2), and normal (3).
distr.opt	If any choice in distr was anything other than uniform (1), this is a list equal to the length of the parameters with sub-components for the mean and sd, which are required in the truncated normal and normal distributions.
n.restarts	The number of solver restarts required.
n.sim	The number of random parameters to generate for every restart of the solver. Note that there will always be significant rejections if inequality bounds are present. Also, this choice should also be motivated by the width of the upper and lower bounds.
cluster	If you want to make use of parallel functionality, initialize and pass a cluster object from the parallel package (see details), and remember to terminate it!
rseed	(Optional) A seed to initiate the random number generator, else system time will be used.
...	(Optional) Additional parameters passed to the main, equality or inequality functions

Details

Given a set of lower and upper bounds, the function generates, for those parameters not set as fixed, random values from one of the 3 chosen distributions. Depending on the `eval.type` option of the `control` argument, the function is either directly evaluated for those points not violating any inequality constraints, or indirectly via a penalty barrier function jointly comprising the objective and constraints. The resulting values are then sorted, and the best N (N = random.restart) parameter vectors (corresponding to the best N objective function values) chosen in order to initialize the solver. Since version 1.14, it is up to the user to prepare and pass a cluster object from the parallel package for use with gosolnp, after which the parLapply function is used. If your function makes use of additional packages, or functions, then make sure to export them via the `clusterExport` function of the parallel package. Additional arguments passed to the solver via the ... option are evaluated and exported by gosolnp to the cluster.

Value

A list containing the following values:

`pars`	Optimal Parameters.
`convergence`	Indicates whether the solver has converged (0) or not (1).
`values`	Vector of function values during optimization with last one the value at the optimal.
`lagrange`	The vector of Lagrange multipliers.
`hessian`	The Hessian at the optimal solution.
`ineqx0`	The estimated optimal inequality vector of slack variables used for transforming the inequality into an equality constraint.
`nfuneval`	The number of function evaluations.
`elapsed`	Time taken to compute solution.
`start.pars`	The parameter vector used to start the solver

Note

The choice of which distribution to use for randomly sampling the parameter space should be driven by the user's knowledge of the problem and confidence or lack thereof of the parameter distribution. The uniform distribution indicates a lack of confidence in the location or dispersion of the parameter, while the truncated normal indicates a more confident choice in both the location and dispersion. On the other hand, the normal indicates perhaps a lack of knowledge in the upper or lower bounds, but some confidence in the location and dispersion of the parameter. In using choices (2) and (3) for `distr`, the `distr.opt` list must be supplied with `mean` and `sd` as subcomponents for those parameters not using the uniform (the examples section hopefully clarifies the usage).

Author(s)

Alexios Ghalanos and Stefan Theussl
Y.Ye (original matlab version of solnp)

References

Y.Ye, *Interior algorithms for linear, quadratic, and linearly constrained non linear programming*, PhD Thesis, Department of EES Stanford University, Stanford CA.
Hu, X. and Shonkwiler, R. and Spruill, M.C. *Random Restarts in Global Optimization*, 1994, Georgia Institute of technology, Atlanta.

Examples

```
## Not run:
# [Example 1]
# Distributions of Electrons on a Sphere Problem:
# Given n electrons, find the equilibrium state distribution (of minimal Coulomb
# potential) of the electrons positioned on a conducting sphere. This model is
# from the COPS benchmarking suite. See http://www-unix.mcs.anl.gov/~more/cops/.
gofn = function(dat, n)
{

x = dat[1:n]
y = dat[(n+1):(2*n)]
z = dat[(2*n+1):(3*n)]
ii = matrix(1:n, ncol = n, nrow = n, byrow = TRUE)
jj = matrix(1:n, ncol = n, nrow = n)
ij = which(ii<jj, arr.ind = TRUE)
i = ij[,1]
j = ij[,2]
#  Coulomb potential
potential = sum(1.0/sqrt((x[i]-x[j])^2 + (y[i]-y[j])^2 + (z[i]-z[j])^2))
potential
}

goeqfn = function(dat, n)
{
x = dat[1:n]
y = dat[(n+1):(2*n)]
z = dat[(2*n+1):(3*n)]
apply(cbind(x^2, y^2, z^2), 1, "sum")
}

n = 25
LB = rep(-1, 3*n)
UB = rep(1,  3*n)
eqB = rep(1, n)
ans = gosolnp(pars  = NULL, fixed = NULL, fun = gofn, eqfun = goeqfn, eqB = eqB,
LB = LB, UB = UB, control = list(outer.iter = 100, trace = 1),
distr = rep(1, length(LB)), distr.opt = list(), n.restarts = 2, n.sim = 20000,
rseed = 443, n = 25)
# should get a function value around 243.813

# [Example 2]
# Parallel functionality for solving the Upper to Lower CVaR problem (not properly
# formulated...for illustration purposes only).
```

```
mu =c(1.607464e-04, 1.686867e-04, 3.057877e-04, 1.149289e-04, 7.956294e-05)
sigma = c(0.02307198,0.02307127,0.01953382,0.02414608,0.02736053)
R = matrix(c(1, 0.408, 0.356, 0.347, 0.378,  0.408, 1, 0.385, 0.565, 0.578, 0.356,
0.385, 1, 0.315, 0.332, 0.347, 0.565, 0.315, 1, 0.662, 0.378, 0.578,
0.332, 0.662, 1), 5,5, byrow=TRUE)
# Generate Random deviates from the multivariate Student distribution
set.seed(1101)
v = sqrt(rchisq(10000, 5)/5)
S = chol(R)
S = matrix(rnorm(10000 * 5), 10000) %*% S
ret = S/v
RT = as.matrix(t(apply(ret, 1, FUN = function(x) x*sigma+mu)))
# setup the functions
.VaR = function(x, alpha = 0.05)
{
VaR = quantile(x, probs = alpha, type = 1)
VaR
}

.CVaR = function(x, alpha = 0.05)
{
VaR = .VaR(x, alpha)
X = as.vector(x[, 1])
CVaR = VaR - 0.5 * mean(((VaR-X) + abs(VaR-X))) / alpha
CVaR
}
.fn1 = function(x,ret)
{
port=ret%*%x
obj=-.CVaR(-port)/.CVaR(port)
return(obj)
}

# abs(sum) of weights ==1
.eqn1  = function(x,ret)
{
sum(abs(x))
}

LB=rep(0,5)
UB=rep(1,5)
pars=rep(1/5,5)
ctrl = list(delta = 1e-10, tol = 1e-8, trace = 0)
cl = makePSOCKcluster(2)
# export the auxilliary functions which are used and cannot be seen by gosolnp
clusterExport(cl, c(".CVaR", ".VaR"))
ans = gosolnp(pars, fun = .fn1, eqfun = .eqn1, eqB = 1, LB = LB, UB = UB,
n.restarts = 2, n.sim=500, cluster = cl, ret = RT)
ans
# don't forget to stop the cluster!
stopCluster(cl)

## End(Not run)
```

solnp *Nonlinear optimization using augmented Lagrange method.*

Description

The solnp function is based on the solver by Yinyu Ye which solves the general nonlinear programming problem:

$$\min f(x)$$
$$\text{s.t.}$$
$$g(x) = 0$$
$$l_h \leq h(x) \leq u_h$$
$$l_x \leq x \leq u_x$$

where, $f(x)$, $g(x)$ and $h(x)$ are smooth functions.

Usage

```
solnp(pars, fun, eqfun = NULL, eqB = NULL, ineqfun = NULL, ineqLB = NULL,
ineqUB = NULL, LB = NULL, UB = NULL, control = list(), ...)
```

Arguments

pars	The starting parameter vector.
fun	The main function which takes as first argument the parameter vector and returns a single value.
eqfun	(Optional) The equality constraint function returning the vector of evaluated equality constraints.
eqB	(Optional) The equality constraints.
ineqfun	(Optional) The inequality constraint function returning the vector of evaluated inequality constraints.
ineqLB	(Optional) The lower bound of the inequality constraints.
ineqUB	(Optional) The upper bound of the inequality constraints.
LB	(Optional) The lower bound on the parameters.
UB	(Optional) The upper bound on the parameters.
control	(Optional) The control list of optimization parameters. See below for details.
...	(Optional) Additional parameters passed to the main, equality or inequality functions. Note that the main and constraint functions must take the exact same arguments, irrespective of whether they are used by all of them.

Details

The solver belongs to the class of indirect solvers and implements the augmented Lagrange multiplier method with an SQP interior algorithm.

Value

A list containing the following values:

`pars`	Optimal Parameters.
`convergence`	Indicates whether the solver has converged (0) or not (1 or 2).
`values`	Vector of function values during optimization with last one the value at the optimal.
`lagrange`	The vector of Lagrange multipliers.
`hessian`	The Hessian of the augmented problem at the optimal solution.
`ineqx0`	The estimated optimal inequality vector of slack variables used for transforming the inequality into an equality constraint.
`nfuneval`	The number of function evaluations.
`elapsed`	Time taken to compute solution.

Control

rho This is used as a penalty weighting scaler for infeasibility in the augmented objective function. The higher its value the more the weighting to bring the solution into the feasible region (default 1). However, very high values might lead to numerical ill conditioning or significantly slow down convergence.

outer.iter Maximum number of major (outer) iterations (default 400).

inner.iter Maximum number of minor (inner) iterations (default 800).

delta Relative step size in forward difference evaluation (default 1.0e-7).

tol Relative tolerance on feasibility and optimality (default 1e-8).

trace The value of the objective function and the parameters is printed at every major iteration (default 1).

Note

The control parameters `tol` and `delta` are key in getting any possibility of successful convergence, therefore it is suggested that the user change these appropriately to reflect their problem specification.
The solver is a local solver, therefore for problems with rough surfaces and many local minima there is absolutely no reason to expect anything other than a local solution.

Author(s)

Alexios Ghalanos and Stefan Theussl
Y.Ye (original matlab version of solnp)

References

Y.Ye, *Interior algorithms for linear, quadratic, and linearly constrained non linear programming*, PhD Thesis, Department of EES Stanford University, Stanford CA.

Examples

```
# From the original paper by Y.Ye
# see the unit tests for more....
#--------------------------------------------------------------------------------
# POWELL Problem
fn1=function(x)
{
exp(x[1]*x[2]*x[3]*x[4]*x[5])
}

eqn1=function(x){
z1=x[1]*x[1]+x[2]*x[2]+x[3]*x[3]+x[4]*x[4]+x[5]*x[5]
z2=x[2]*x[3]-5*x[4]*x[5]
z3=x[1]*x[1]*x[1]+x[2]*x[2]*x[2]
return(c(z1,z2,z3))
}


x0 = c(-2, 2, 2, -1, -1)
powell=solnp(x0, fun = fn1, eqfun = eqn1, eqB = c(10, 0, -1))
```

`startpars` *Generates and returns a set of starting parameters by sampling the parameter space based on the evaluation of the function and constraints.*

Description

A simple penalty barrier function is formed which is then evaluated at randomly sampled points based on the upper and lower parameter bounds (when `eval.type` = 2), else the objective function directly for values not violating any inequality constraints (when `eval.type` = 1). The sampled points can be generated from the uniform, normal or truncated normal distributions.

Usage

```
startpars(pars = NULL, fixed = NULL, fun, eqfun = NULL, eqB = NULL,
ineqfun = NULL, ineqLB = NULL, ineqUB = NULL, LB = NULL, UB = NULL,
distr = rep(1, length(LB)), distr.opt = list(), n.sim = 20000, cluster = NULL,
rseed = NULL, bestN = 15, eval.type = 1, trace = FALSE, ...)
```

Arguments

`pars`	The starting parameter vector. This is not required unless the fixed option is also used.
`fixed`	The numeric index which indicates those parameters which should stay fixed instead of being randomly generated.
`fun`	The main function which takes as first argument the parameter vector and returns a single value.

eqfun	(Optional) The equality constraint function returning the vector of evaluated equality constraints.
eqB	(Optional) The equality constraints.
ineqfun	(Optional) The inequality constraint function returning the vector of evaluated inequality constraints.
ineqLB	(Optional) The lower bound of the inequality constraints.
ineqUB	(Optional) The upper bound of the inequality constraints.
LB	The lower bound on the parameters. This is not optional in this function.
UB	The upper bound on the parameters. This is not optional in this function.
distr	A numeric vector of length equal to the number of parameters, indicating the choice of distribution to use for the random parameter generation. Choices are uniform (1), truncated normal (2), and normal (3).
distr.opt	If any choice in distr was anything other than uniform (1), this is a list equal to the length of the parameters with sub-components for the mean and sd, which are required in the truncated normal and normal distributions.
bestN	The best N (less than or equal to n.sim) set of parameters to return.
n.sim	The number of random parameter sets to generate.
cluster	If you want to make use of parallel functionality, initialize and pass a cluster object from the parallel package (see details), and remember to terminate it!
rseed	(Optional) A seed to initiate the random number generator, else system time will be used.
eval.type	Either 1 (default) for the direction evaluation of the function (excluding inequality constraint violations) or 2 for the penalty barrier method.
trace	(logical) Whether to display the progress of the function evaluation.
...	(Optional) Additional parameters passed to the main, equality or inequality functions

Details

Given a set of lower and upper bounds, the function generates, for those parameters not set as fixed, random values from one of the 3 chosen distributions. For simple functions with only inequality constraints, the direct method (eval.type = 1) might work better. For more complex setups with both equality and inequality constraints the penalty barrier method (eval.type = 2)might be a better choice.

Value

A matrix of dimension bestN x (no.parameters + 1). The last column is the evaluated function value.

Note

The choice of which distribution to use for randomly sampling the parameter space should be driven by the user's knowledge of the problem and confidence or lack thereof of the parameter distribution. The uniform distribution indicates a lack of confidence in the location or dispersion of the parameter,

while the truncated normal indicates a more confident choice in both the location and dispersion. On the other hand, the normal indicates perhaps a lack of knowledge in the upper or lower bounds, but some confidence in the location and dispersion of the parameter. In using choices (2) and (3) for `distr`, the `distr.opt` list must be supplied with `mean` and `sd` as subcomponents for those parameters not using the uniform.

Author(s)

Alexios Ghalanos and Stefan Theussl

Examples

```
## Not run:
library(Rsolnp)
library(parallel)
# Windows
cl = makePSOCKcluster(2)
# Linux:
# makeForkCluster(nnodes = getOption("mc.cores", 2L), ...)

gofn = function(dat, n)
{

x = dat[1:n]
y = dat[(n+1):(2*n)]
z = dat[(2*n+1):(3*n)]
ii = matrix(1:n, ncol = n, nrow = n, byrow = TRUE)
jj = matrix(1:n, ncol = n, nrow = n)
ij = which(ii<jj, arr.ind = TRUE)
i = ij[,1]
j = ij[,2]
#  Coulomb potential
potential = sum(1.0/sqrt((x[i]-x[j])^2 + (y[i]-y[j])^2 + (z[i]-z[j])^2))
potential
}

goeqfn = function(dat, n)
{
x = dat[1:n]
y = dat[(n+1):(2*n)]
z = dat[(2*n+1):(3*n)]
apply(cbind(x^2, y^2, z^2), 1, "sum")
}
n = 25
LB  = rep(-1, 3*n)
UB  = rep( 1, 3*n)
eqB = rep( 1,   n)

sp = startpars(pars = NULL, fixed = NULL, fun = gofn , eqfun = goeqfn,
eqB = eqB, ineqfun = NULL, ineqLB = NULL, ineqUB = NULL, LB = LB, UB = UB,
distr = rep(1, length(LB)), distr.opt = list(), n.sim = 2000,
```

```
cluster = cl, rseed = 100, bestN = 15, eval.type = 2, n = 25)
#stop cluster
stopCluster(cl)
# the last column is the value of the evaluated function (here it is the barrier
# function since eval.type = 2)
print(round(apply(sp, 2, "mean"), 3))
# remember to remove the last column
ans = solnp(pars=sp[1,-76],fun = gofn , eqfun = goeqfn , eqB = eqB, ineqfun = NULL,
ineqLB = NULL, ineqUB = NULL, LB = LB, UB = UB, n = 25)
# should get a value of around 243.8162

## End(Not run)
```

Index