Package ‘Rsolnp’

June 20, 2025

Type Package

Title General Non-Linear Optimization

Version 2.0.0

Date 2025-06-01

Maintainer Alexios Galanos <alexios@4dscape.com>

Depends R (>= 3.0.2)

LinkingTo Rcpp (>= 0.10.6), RcppArmadillo

Imports Rcpp, truncnorm, parallel, stats, numDeriv, future.apply

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

LazyLoad yes

License GPL-2

Encoding UTF-8

NeedsCompilation yes

RoxygenNote 7.3.2

Suggests knitr, rmarkdown

VignetteBuilder knitr

Author Alexios Galanos [aut, cre, cph] (ORCID: <https://orcid.org/0009-0000-9308-0457>),
Stefan Theussl [ctb] (ORCID: <https://orcid.org/0000-0002-6523-4620>),
Yinyu Ye [aut] (Original author of the solnp Matlab code, Stanford University)

Repository CRAN

Date/Publication 2025-06-20 09:20:02 UTC

Contents

`csolnp` *Nonlinear optimization using augmented Lagrange method (C++ version)*

Description

Nonlinear optimization using augmented Lagrange method (C++ version)

Usage

```
csolnp(
  pars,
  fn,
  gr = NULL,
  eq_fn = NULL,
  eq_b = NULL,
  eq_jac = NULL,
  ineq_fn = NULL,
  ineq_lower = NULL,
  ineq_upper = NULL,
  ineq_jac = NULL,
  lower = NULL,
  upper = NULL,
  control = list(),
  use_r_version = FALSE,
  ...
)
```

Arguments

`pars`	an numeric vector of decision variables (length n).
`fn`	the objective function (must return a scalar).
`gr`	an optional function for computing the analytic gradient of the function (must return a vector of length n).
`eq_fn`	an optional function for calculating equality constraints.
`eq_b`	a vector of the equality bounds (if eq_fn provided).
`eq_jac`	an optional function for computing the analytic jacobian of the equality. function (a matrix with number of columns n and number of rows the same length as the number of equalities).

ineq_fn	an optional function for calculating inequality constraints.
ineq_lower	the lower bounds for the inequality (must be finite)
ineq_upper	the upper bounds for the inequalitiy (must be finite)
ineq_jac	an optional function for computing the analytic jacobian of the inequality (a matrix with number of columns n and number of rows the same length as the number of inequalities).
lower	lower bounds for the parameters. This is strictly required.
upper	upper bounds for the parameters. This is strictly required.
control	a list of solver control parameters (see details).
use_r_version	(logical) used for debugging and validation. Uses the R version of the solver rather than the C++ version. Will be deprecated in future releases.
...	additional arguments passed to the supplied functions (common to all functions supplied).

Details

The optimization problem solved by `csolnp` is formulated as:

$$\begin{aligned} \min_{x \in \mathbb{R}^n} \quad & f(x) \\ \text{s.t.} \quad & g(x) = b \\ & h_l \le h(x) \le h_u \\ & x_l \le x \le x_u \end{aligned}$$

where $f(x)$ is the objective function, $g(x)$ is the vector of equality constraints with target value b, $h(x)$ is the vector of inequality constraints bounded by h_l and h_u, with parameter bounds x_l and x_u. Internally, inequality constraints are converted into equality constraints using slack variables and solved using an augmented Lagrangian approach. This function is based on the original R code, but converted to C++, making use of `Rcpp` and `RcppArmadillo`. Additionally, it allows the user to pass in analytic gradient and Jacobians, else finite differences using functions from the `numDeriv` package are used.

The control list consists of the following options:

rho Numeric. Initial penalty parameter for the augmented Lagrangian. Controls the weight given to constraint violation in the objective. Default is `1`.

max_iter Integer. Maximum number of major (outer) iterations allowed. Default is `400`.

min_iter Integer. Maximum number of minor (inner) iterations (per major iteration) for the quadratic subproblem solver. Default is `800`.

tol Numeric. Convergence tolerance for both feasibility (constraint violation) and optimality (change in objective). The algorithm terminates when changes fall below this threshold. Default is `1e-8`.

trace Integer If `1`, prints progress, `2` includes diagnostic information during optimization. Default is `0`.

Tracing information provides the following:

Iter The current major iteration number.

Obj The value of the objective function $f(x)$ at the current iterate.

||Constr|| The norm of the current constraint violation, summarizing how well all constraints (equality and inequality) are satisfied. Typically the Euclidean or infinity norm.

RelObj The relative change in the objective function value compared to the previous iteration, i.e., $|f_k - f_{k-1}|/max(1, |f_{k-1}|)$.

Step The norm of the parameter update taken in this iteration, i.e., $||x_k - x_{k-1}||$.

Penalty The current value of the penalty parameter (ρ) in the augmented Lagrangian. This parameter is adaptively updated to balance objective minimization and constraint satisfaction.

Value

A list with the following slot:

pars The parameters at the optimal solution found.

objective The value of the objective at the optimal solution found.

objective_history A vector of objective values obtained at each outer iteration.

out_iterations The number of outer iterations used to arrive at the solution.

convergence The convergence code (0 = converged).

message The convergence message.

kkt_diagnostics A list of optimal solution diagnostics.

lagrange The vector of Lagrange multipliers at the optimal solution found.

n_eval The number of function evaluations.

elapsed The time taken to find a solution.

hessian The Hessian at the optimal solution.

Author(s)

Alexios Galanos

`csolnp_ms` *Multi-start version of csolnp*

Description

Runs the csolnp solver from multiple diverse feasible starting values and returns the best solution found.

Usage

```
csolnp_ms(
  fn,
  gr = NULL,
  eq_fn = NULL,
  eq_b = NULL,
  eq_jac = NULL,
  ineq_fn = NULL,
  ineq_lower = NULL,
  ineq_upper = NULL,
  ineq_jac = NULL,
  lower = NULL,
  upper = NULL,
  control = list(),
  n_candidates = 20,
  penalty = 10000,
  eq_tol = 1e-06,
  ineq_tol = 1e-06,
  seed = NULL,
  return_all = FALSE,
  ...
)
```

Arguments

fn	the objective function (must return a scalar).
gr	an optional function for computing the analytic gradient of the function (must return a vector of length n).
eq_fn	an optional function for calculating equality constraints.
eq_b	a vector of the equality bounds (if eq_fn provided).
eq_jac	an optional function for computing the analytic Jacobian of the equality function (a matrix with number of columns n and number of rows equal to the number of equalities).
ineq_fn	an optional function for calculating inequality constraints.
ineq_lower	the lower bounds for the inequality constraints (must be finite).
ineq_upper	the upper bounds for the inequality constraints (must be finite).
ineq_jac	an optional function for computing the analytic Jacobian of the inequality function (a matrix with number of columns n and number of rows equal to the number of inequalities).
lower	lower bounds for the parameters. This is strictly required.
upper	upper bounds for the parameters. This is strictly required.
control	a list of solver control parameters (see details).
n_candidates	integer. The number of initial feasible candidate points to generate for multi-start optimization. Default is 20.

penalty	numeric. The penalty parameter used when projecting to feasibility for candidate generation. Default is 1e4.
eq_tol	Numeric. Tolerance for equality constraint violation (default is 1e-6). Candidate solutions with kkt_diagnostics\$eq_violation less than or equal to this value are considered feasible with respect to equality constraints.
ineq_tol	Numeric. Tolerance for inequality constraint violation (default is 1e-6). Candidate solutions with kkt_diagnostics\$ineq_violation less than or equal to this value are considered feasible with respect to inequality constraints.
seed	an optional random seed used to initialize the random number generator for the random samples.
return_all	logical. Whether to return all solutions as a list. This may be useful for debugging.
...	additional arguments passed to the supplied functions (common to all functions supplied).

Details

This function automates the process of generating multiple feasible starting points (using lower, upper, and constraint information), runs `csolnp` from each, and returns the solution with the lowest objective value. It is useful for problems where local minima are a concern or the objective surface is challenging.

Candidate Generation:

The `generate_feasible_starts` approach creates a diverse set of initial parameter vectors (candidates) that are feasible with respect to box and (optionally) nonlinear constraints. The process is as follows:

1. For each candidate, a random point is sampled inside the parameter box constraints (`lower` and `upper`) but a small distance away from the boundaries, to avoid numerical issues. This is achieved by a helper function that applies a user-specified buffer (`eps`).
2. If nonlinear inequality constraints (`ineq_fn`) are provided, each sampled point is projected towards the feasible region using a fast penalized minimization. This step does not solve the feasibility problem exactly, but quickly produces a point that satisfies the constraints to within a specified tolerance, making it suitable as a starting point for optimization.
3. If only box constraints are present, the sampled point is used directly as a feasible candidate.
4. The set of feasible candidates is ranked by the objective with lower values considered better. This allows prioritization of candidates that start closer to optimality.

This method efficiently creates a diverse set of robust initial values, improving the chances that multi-start optimization will identify the global or a high-quality local solution, especially in the presence of non-convexities or challenging constraint boundaries. **Solution Selection:** For each candidate starting point, `csolnp_ms` runs the `csolnp` solver and collects the resulting solutions and their associated KKT diagnostics. If equality or inequality constraints are present, candidate solutions are first filtered to retain only those for which the maximum violation of equality (`kkt_diagnostics\$eq_violation`) and/or inequality (`kkt_diagnostics\$ineq_violation`) constraints are less than or equal to user-specified tolerances (`eq_tol` and `ineq_tol`). Among the feasible solutions (those satisfying all constraints within tolerance), the solution with the lowest objective value is selected and returned as the best result. If no candidate fully satisfies the constraints, the solution with the smallest total constraint violation is returned, with a warning issued to indicate that strict feasibility was not achieved. This two-stage selection process ensures that the final result is both feasible (when possible) and optimally minimizes the objective function among all feasible candidates.

Value

A list containing the best solution found by multi-start, with elements analogous to those returned by `csolnp`. If `return_all` is TRUE, then a list of all solutions is returned instead.

Author(s)

Alexios Galanos

See Also

`csolnp`

`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 Galanos 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 (original version)*

Description

Nonlinear optimization using augmented Lagrange method (original version)

Usage

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

Arguments

`pars`	an numeric vector of decision variables (length n).
`fun`	the objective function (must return a scalar).
`eqfun`	an optional function for calculating equality constraints.
`eqB`	a vector of the equality bounds (if eq_fn provided).
`ineqfun`	an optional function for calculating inequality constraints.
`ineqLB`	the lower bounds for the inequality (must be finite)
`ineqUB`	the upper bounds for the inequalitiy (must be finite)
`LB`	lower bounds on decision variables
`UB`	upper bounds on decision variables
`control`	a list of solver control parameters (see details).
`...`	additional arguments passed to the supplied functions (common to all functions supplied).

Details

The optimization problem solved by `csolnp` is formulated as:

$$\begin{aligned} \min_{x \in \mathbb{R}^n} \quad & f(x) \\ \text{s.t.} \quad & g(x) = b \\ & h_l \le h(x) \le h_u \\ & x_l \le x \le x_u \end{aligned}$$

where $f(x)$ is the objective function, $g(x)$ is the vector of equality constraints with target value b, $h(x)$ is the vector of inequality constraints bounded by h_l and h_u, with parameter bounds x_l and x_u. Internally, inequality constraints are converted into equality constraints using slack variables and solved using an augmented Lagrangian approach. The control is a list with the following options:

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).

Value

An list with the following slot:

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.

Author(s)

Alexios Galanos

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))
```

`solnp_problems_table` *List of Valid Test Problems for the SOLNP Suite*

Description

Returns a data.frame of known and registered test problems used with the SOLNP solver. The list includes problems from the Hock-Schittkowski suite as well as a selection of other classic optimization problems.

Usage

```
solnp_problems_table()
```

Details

- All problem functions are expected to follow the naming convention 'Problem_problem' (e.g., 'hs01_problem').
- For Hock-Schittkowski problems, numbers range from 1 to 50, with a few selected extras (e.g., 110, 118, 119).
- The "Other" suite includes named problems like 'box', 'alkylation', 'entropy', 'garch', etc., and are numbered sequentially.

Value

A data.frame with the following columns:

Suite A character string indicating the suite the problem belongs to. One of "Hock-Schittkowski" or "Other".

Problem The base name of the problem function (without the '_problem' suffix).

Number An integer identifier used to index or request problems programmatically.

See Also

solnp_problem_suite()

Examples

```
# View all known problems
tail(solnp_problems_table())

# Filter only HS problems
head(subset(solnp_problems_table(), Suite == "Hock-Schittkowski"))
```

---

solnp_problem_suite *Retrieve Implemented Test Problems for the SOLNP Suite*

Description

Returns a list (or a single object) of implemented test problems corresponding to a selected suite. Problem functions must follow the naming convention 'problem_name_problem' and return a list describing the optimization problem (e.g., objective, constraints, bounds).

Usage

```
solnp_problem_suite(
  suite = "Hock-Schittkowski",
  number = 1,
  return_all = FALSE
)
```

Arguments

suite	Character. The test suite to draw from. Must be one of "Hock-Schittkowski" or "Other". Default is "Hock-Schittkowski".
number	Integer or vector of integers. One or more problem numbers to retrieve. Ignored if return_all = TRUE.
return_all	Logical. If TRUE, returns all implemented problems in the specified suite. Default is FALSE.

Details

- Problems are matched by number within the selected suite, using the table from solnp_problems_table().
- If a requested problem is valid but not yet implemented (i.e., the corresponding function does not exist), a message will inform the user.
- If a problem number exceeds the allowable range (e.g., > 306 for Hock-Schittkowski), an error is raised.

Value

If one problem is requested and implemented, the evaluated problem object is returned directly. Otherwise, an unnamed list of evaluated problem objects is returned.

See Also

solnp_problems_table()

Examples

```
## Not run:
# Retrieve a single HS problem
prob <- solnp_problem_suite(number = 1)

# Retrieve multiple HS problems
probs <- solnp_problem_suite(number = c(1, 2, 3))

# Retrieve problem in "Other" suite
other_prob <- solnp_problem_suite(suite = "Other", number = 1)

## End(Not run)
```

---

solnp_standardize_problem *Standardize an Optimization Problem to NLP Standard Form*

Description

Converts a problem specified with two-sided inequalities and nonzero equality right-hand sides to the standard nonlinear programming (NLP) form.

Usage

```
solnp_standardize_problem(prob)
```

Arguments

prob	A list specifying the problem in SOLNP-compatible format, with components `fn`, `eq_fn`, `eq_jac`, `eq_b`, `ineq_fn`, `ineq_jac`, `ineq_lower`, `ineq_upper`, and others.

Details

The standard form given by the following set of equations:

$$\min_{x} \; f(x)$$

$$\text{subject to} \quad e(x) = 0$$
$$g(x) \leq 0$$

Specifically:

- All equality constraints are standardized to $e(x) = e(x) - b = 0$
- Each two-sided inequality $l \leq g(x) \leq u$ is converted to one or two one-sided constraints: $l - g(x) \leq 0$, $g(x) - u \leq 0$

The returned problem object has all equalities as $e(x) = 0$, all inequalities as $g(x) \leq 0$, and any right-hand side or bounds are absorbed into the standardized constraint functions.

Value

A list with the same structure as the input, but with `eq_fn` and `ineq_fn` standardized to the forms $e(x) = 0$ and $g(x) \leq 0$, and with `eq_b`, `ineq_lower`, and `ineq_upper` removed.

See Also

solnp_problem_suite

Examples

```
# Alkylation problem
p <- solnp_problem_suite(suite = "Other", number = 1)
ps <- solnp_standardize_problem(p)
ps$eq_fn(ps$start)    # standardized equalities: e(x) = 0
ps$ineq_fn(ps$start)  # standardized inequalities: g(x) <= 0
```

---

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.

`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.
`bestN`	The best N (less than or equal to n.sim) set of parameters to return.
`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 Galanos 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