Package 'DEoptimR'

June 7, 2026

Version 1.2-0

Date 2026-06-07

Title Differential Evolution Optimization in Pure R

URL svn://svn.r-forge.r-project.org/svnroot/robustbase/pkg/DEoptimR

Description Differential Evolution (DE) stochastic heuristic algorithms for global optimization of problems with and without general constraints. The aim is to curate a collection of its variants that (1) do not sacrifice simplicity of design, (2) are essentially tuning-free, and (3) can be efficiently implemented directly in the R language. Currently, it provides implementations of the algorithms 'jDE' by Brest et al. (2006) <doi:10.1109/TEVC.2006.872133> for single-objective optimization and 'NCDE' by Qu et al. (2012) <doi:10.1109/TEVC.2011.2161873> for multimodal optimization (single-objective problems with multiple solutions).

Imports stats, methods

Suggests mirai (>= 2.7.1)

Enhances robustbase

License GPL (>= 2)

Author Eduardo L. T. Conceicao [aut, cre], Martin Maechler [ctb] (ORCID: <https://orcid.org/0000-0002-8685-9910>)

Maintainer Eduardo L. T. Conceicao <mail@eduardoconceicao.org>

Repository CRAN

Repository/R-Forge/Project robustbase

Repository/R-Forge/Revision 1025

Repository/R-Forge/DateTimeStamp 2026-06-07 16:58:58

Date/Publication 2026-06-07 18:20:07 UTC

NeedsCompilation no

Contents

`JDEoptim` *Bound-Constrained and Nonlinear Constrained Single-Objective Optimization via Differential Evolution*

Description

A bespoke implementation of the 'jDE' variant by Brest *et al.* (2006) doi:10.1109/TEVC.2006.872133.

Usage

```
JDEoptim(lower, upper, fn,
         constr = NULL, meq = 0, eps = 1e-05,
         NP = 10*length(lower), Fl = 0.1, Fu = 1,
         tau_F = 0.1, tau_CR = 0.1, tau_pF = 0.1,
         jitter_factor = 0.001,
         tol = 1e-15, maxiter = 2000*length(lower), fnscale = 1,
         compare_to = c("median", "max"),
         add_to_init_pop = NULL,
         trace = FALSE, triter = 1,
         details = FALSE, ...)
```

Arguments

`lower, upper`	numeric vectors of *lower* and *upper* bounds for the parameters to be optimized over. Must be finite (`is.finite`) as they bound the hyper-rectangle of the initial random population.
`fn`	(nonlinear) objective `function` to be *minimized*. It takes as first argument the vector of parameters over which minimization is to take place. It must return the value of the function at that point.
`constr`	an optional `function` for specifying the *left-hand side* of nonlinear constraints under which we want to minimize `fn`. Nonlinear equalities should be given first and defined to equal zero ($h_j(X) = 0$), followed by nonlinear inequalities defined as lesser than or equal to zero ($g_i(X) \leq 0$). This function takes the vector of parameters as its first argument and returns a real vector with the length of the total number of constraints. It defaults to `NULL`, meaning that *bound-constrained* minimization is used.
`meq`	an optional positive integer specifying that the first `meq` constraints are treated as *equality* constraints, and all the remaining as *inequality* constraints. Defaults to `0` (inequality constraints only).

eps	maximal admissible constraint violation for equality constraints. An optional real vector of small positive tolerance values with length meq used in the transformation of equalities into inequalities of the form $\|h_j(X)\| - \epsilon \leq 0$. A scalar value is expanded to apply to all equality constraints. Default is 1e-5.
NP	an optional positive integer giving the number of candidate solutions in the randomly distributed initial population. Defaults to 10*length(lower).
Fl	an optional scalar which represents the minimum value that the *scaling factor* F could take. Default is 0.1, which is almost always satisfactory.
Fu	an optional scalar which represents the maximum value that the *scaling factor* F could take. Default is 1, which is almost always satisfactory.
tau_F	an optional scalar which represents the probability that the *scaling factor* F is updated. Defaults to 0.1, which is almost always satisfactory.
tau_CR	an optional constant value which represents the probability that the *crossover probability* CR is updated. Defaults to 0.1, which is almost always satisfactory.
tau_pF	an optional scalar which represents the probability that the *mutation probability* p_F in the mutation strategy DE/rand/1/either-or is updated. Defaults to 0.1.
jitter_factor	an optional tuning constant for *jitter*. If NULL only *dither* is used. Defaults to 0.001.
tol	an optional positive scalar giving the tolerance for the stopping criterion. Default is 1e-15.
maxiter	an optional positive integer specifying the maximum number of iterations that may be performed before the algorithm is halted. Defaults to 2000*length(lower).
fnscale	an optional positive scalar specifying the typical magnitude of fn. It is used only in the *stopping criterion*. Defaults to 1. See 'Details'.
compare_to	an optional character string controlling which function should be applied to the fn values of the candidate solutions in a generation to be compared with the so-far best one when evaluating the *stopping criterion*. If "median" the median function is used; else, if "max" the max function is used. It defaults to "median". See 'Details'.
add_to_init_pop	an optional real vector of length length(lower) or matrix with length(lower) rows specifying initial values of the parameters to be optimized which are appended to the randomly generated initial population. It defaults to NULL.
trace	an optional logical value indicating if a trace of the iteration progress should be printed. Default is FALSE.
triter	an optional positive integer giving the frequency of tracing (every triter iterations) when trace = TRUE. Default is triter = 1, in which case iteration : < value of stopping test > (value of best solution) best solution { index of violated constraints } is printed at each iteration.
details	an optional logical value. If TRUE the output will contain the parameters in the final population and their respective fn values. Defaults to FALSE.
...	optional additional arguments passed to fn and constr.

Details

Overview: The setting of the *control parameters* of canonical Differential Evolution (DE) is crucial for the algorithm's performance. Unfortunately, when the generally recommended values for these parameters (see, *e.g.*, Storn and Price, 1997) are unsuitable for use, their determination is often difficult and time consuming. The jDE algorithm proposed in Brest *et al.* (2006) employs a simple self-adaptive scheme to perform the automatic setting of control parameters scale factor `F` and crossover rate `CR`.

This implementation differs from the original description, most notably in the use of the *DE/rand/1/either-or* mutation strategy (Price *et al.*, 2005), combination of *jitter with dither* (Storn, 2008), and the random initialization of `F` and `CR`. The mutation operator brings an additional control parameter, the mutation probability p_F, which is self-adapted in the same manner as `CR`.

As done by jDE and its variants (Brest *et al.*, 2021) each worse parent in the current population is *immediately replaced* (asynchronous update) by its newly generated better or equal offspring (Babu and Angira, 2006) instead of updating the current population with all the new solutions at the same time as in classical DE (synchronous update).

As the algorithm subsamples via `sample()` which from R version 3.6.0 depends on `RNGkind(*, sample.kind)`, exact reproducibility of results from R versions 3.5.3 and earlier requires setting `RNGversion("3.5.0")`. In any case, do use `set.seed()` additionally for reproducibility!

Constraint Handling: Constraint handling is done using the approach described in Zhang and Rangaiah (2012), but with a *different reduction updating scheme* for the constraint relaxation value (μ). Instead of doing it once for every generation or iteration, the reduction is triggered for two cases when the *constraints only contain inequalities*. Firstly, every time a feasible solution is selected for replacement in the next generation by a new feasible trial candidate solution with a better objective function value. Secondly, whenever a current infeasible solution gets replaced by a feasible one. If the constraints *include equalities*, then the reduction is not triggered in this last case. This constitutes an original feature of the implementation.

The performance of any constraint handling technique for metaheuristics is severely impaired by a small feasible region. Therefore, equality constraints are particularly difficult to handle due to the tiny feasible region they define. So, instead of explicitly including all equality constraints in the formulation of the optimization problem, it might prove advantageous to eliminate some of them. This is done by expressing one variable x_k in terms of the remaining others for an equality constraint $h_j(X) = 0$ where $X = [x_1, \ldots, x_k, \ldots, x_d]$ is the vector of solutions, thereby obtaining a relationship as $x_k = R_{k,j}([x_1, \ldots, x_{k-1}, x_{k+1}, \ldots, x_d])$. In this way both the variable x_k and the equality constraint $h_j(X) = 0$ can be removed altogether from the original optimization formulation, since the value of x_k can be calculated during the search process by the relationship $R_{k,j}$. Notice, however, that two additional inequalities

$$l_k \leq R_{k,j}([x_1, \ldots, x_{k-1}, x_{k+1}, \ldots, x_d]) \leq u_k,$$

where the values l_k and u_k are the lower and upper bounds of x_k, respectively, must be provided in order to obtain an equivalent formulation of the problem. For guidance and examples on applying this approach see Wu *et al.* (2015).

Bound constraints are enforced by the *midpoint base* approach (see, *e.g.*, Biedrzycki *et al.*, 2019).

Discrete and Integer Variables: Any DE variant is easily extended to deal with *mixed integer nonlinear programming* problems using a small variation of the technique presented by Lampinen

and Zelinka (1999). Integer values are obtained by means of the `floor()` function *only* in the evaluation of the objective function and constraints, whereas DE itself still uses continuous variables. Additionally, each upper bound of the integer variables should be added by `1`.

Notice that the final solution needs to be converted with `floor()` to obtain its *integer* elements.

Stopping Criterion: The algorithm is stopped if

$$\frac{\text{compare\_to}\{[\text{fn}(X_1), \ldots, \text{fn}(X_{\text{npop}})]\} - \text{fn}(X_{\text{best}})}{\text{fnscale}} \leq \text{tol},$$

where the "best" individual X_{best} is the *feasible* solution with the lowest objective function value in the population and the total number of elements in the population, `npop`, is `NP+NCOL(add_to_init_pop)`. For `compare_to = "max"` this is the *Diff* criterion studied by Zielinski and Laur (2008) among several other alternatives, which was found to yield the best results.

Value

A list with the following components:

`par`	The best set of parameters found.
`value`	The value of `fn` corresponding to `par`.
`iter`	Number of iterations taken by the algorithm.
`convergence`	An integer code. `0` indicates successful completion. `1` indicates that the iteration limit `maxiter` has been reached.

and if `details = TRUE`:

`poppar`	Matrix of dimension `(length(lower), npop)`, with columns corresponding to the parameter vectors remaining in the population.
`popcost`	The values of `fn` associated with `poppar`, vector of length `npop`.

Note

It is possible to perform a warm start, *i.e.*, starting from the previous run and resume optimization, using `NP = 0` and the component `poppar` for the `add_to_init_pop` argument.

Author(s)

Eduardo L. T. Conceicao <mail@eduardoconceicao.org>

References

Babu, B. V. and Angira, R. (2006) Modified differential evolution (MDE) for optimization of non-linear chemical processes. *Computers and Chemical Engineering* **30**, 989–1002. doi:10.1016/j.compchemeng.2005.12.020.

Biedrzycki, R., Arabas, J. and Jagodzinski, D. (2019) Bound constraints handling in differential evolution: An experimental study. *Swarm and Evolutionary Computation* **50**, 100453. doi:10.1016/j.swevo.2018.10.004.

Brest, J., Greiner, S., Boskovic, B., Mernik, M. and Zumer, V. (2006) Self-adapting control parameters in differential evolution: A comparative study on numerical benchmark problems. *IEEE Transactions on Evolutionary Computation* **10**, 646–657. doi:10.1109/TEVC.2006.872133.

Brest, J., Maucec, M. S. and Boskovic, B. (2021) Self-adaptive differential evolution algorithm with population size reduction for single objective bound-constrained optimization: Algorithm j21; in *2021 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, pp. 817–824. doi:10.1109/CEC45853.2021.9504782.

Lampinen, J. and Zelinka, I. (1999). Mechanical engineering design optimization by differential evolution; in Corne, D., Dorigo, M. and Glover, F., Eds., *New Ideas in Optimization*. McGraw-Hill, pp. 127–146.

Price, K. V., Storn, R. M. and Lampinen, J. A. (2005) *Differential evolution: A practical approach to global optimization*. Springer, Berlin, Heidelberg, pp. 117–118. doi:10.1007/3540313060_2.

Storn, R. (2008) Differential evolution research — Trends and open questions; in Chakraborty, U. K., Ed., *Advances in differential evolution*. SCI 143, Springer, Berlin, Heidelberg, pp. 11–12. doi:10.1007/9783540688303_1.

Storn, R. and Price, K. (1997) Differential evolution - A simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization* **11**, 341–359. doi:10.1023/A:1008202821328.

Wu, G., Pedrycz, W., Suganthan, P. N. and Mallipeddi, R. (2015) A variable reduction strategy for evolutionary algorithms handling equality constraints. *Applied Soft Computing* **37**, 774–786. doi:10.1016/j.asoc.2015.09.007.

Zhang, H. and Rangaiah, G. P. (2012) An efficient constraint handling method with integrated differential evolution for numerical and engineering optimization. *Computers and Chemical Engineering* **37**, 74–88. doi:10.1016/j.compchemeng.2011.09.018.

Zielinski, K. and Laur, R. (2008) Stopping criteria for differential evolution in constrained single-objective optimization; in Chakraborty, U. K., Ed., *Advances in differential evolution*. SCI 143, Springer, Berlin, Heidelberg, pp. 111–138. doi:10.1007/9783540688303_4.

See Also

Function `DEoptim()` in the **DEoptim** package implements both canonical DE and the JADE algorithm, but does not allow constraints in the same flexible manner as `JDEoptim()`.

Examples

```
# NOTE: Examples were excluded from testing
#       to reduce package check time.

# Use a preset seed so test values are reproducible.
set.seed(1234)

# Bound-constrained optimization --------------------

#   Griewank function
#
#   -600 <= xi <= 600, i = {1, 2, ..., n}
#   The function has a global minimum located at
```

```
#   x* = (0, 0, ..., 0) with f(x*) = 0. Number of local minima
#   for arbitrary n is unknown, but in the two dimensional case
#   there are some 500 local minima.
#
#   Source:
#     Ali, M. Montaz, Khompatraporn, Charoenchai, and
#     Zabinsky, Zelda B. (2005).
#     A numerical evaluation of several stochastic algorithms
#     on selected continuous global optimization test problems.
#     Journal of Global Optimization 31, 635-672.
#     https://doi.org/10.1007/s10898-004-9972-2
griewank <- function(x) {
    1 + crossprod(x)/4000 - prod( cos(x/sqrt(seq_along(x))) )
}

JDEoptim(rep(-600, 10), rep(600, 10), griewank,
         tol = 1e-7, trace = TRUE, triter = 50)

# Nonlinear constrained optimization ---------------------

#   f(x) = 35*x[1]^0.6 + 35*x[2]^0.6
#   subject to:
#   h1(x) = 600*x[1] - 50*x[3] - x[1]*x[3] + 5000 = 0
#   h2(x) = 600*x[2] + 50*x[3] - 15000 = 0
#
#   0 <= x1 <= 34, 0 <= x2 <= 17, 100 <= x3 <= 300
#   The global optimum is
#   (x1, x2, x3; f) = (0, 16.666667, 100; 189.311627).
#
#   Source:
#     Westerberg, Arthur W., and Shah, Jigar V. (1978).
#     Assuring a global optimum by the use of an upper bound
#     on the lower (dual) bound.
#     Computers and Chemical Engineering 2, 83-92.
#     https://doi.org/10.1016/0098-1354(78)80012-X
fcn <-
    list(obj = function(x) {
             35*x[1]^0.6 + 35*x[2]^0.6
         },
         eq = 2,                   # two equality constraints
         con = function(x) {
             x1 <- x[1]; x3 <- x[3]
             c(600*x1 - 50*x3 - x1*x3 + 5000,
               600*x[2] + 50*x3 - 15000)
         })

JDEoptim(c(0, 0, 100), c(34, 17, 300),
         fn = fcn$obj, constr = fcn$con, meq = fcn$eq,
         tol = 1e-7, trace = TRUE, triter = 50)

#   Designing a pressure vessel
#   Case A: all variables are treated as continuous
#
```

```
#   f(x) = 0.6224*x[1]*x[3]*x[4] + 1.7781*x[2]*x[3]^2 +
#              3.1611*x[1]^2*x[4] + 19.84*x[1]^2*x[3]
#   subject to:
#   g1(x) = 0.0193*x[3] - x[1] <= 0
#   g2(x) = 0.00954*x[3] - x[2] <= 0
#   g3(x) = 750.0*1728.0 - pi*x[3]^2*x[4] - 4/3*pi*x[3]^3 <= 0
#
#   1.1 <= x1 <= 12.5*, 0.6 <= x2 <= 12.5*,
#   0.0 <= x3 <= 240.0*, 0.0 <= x4 <= 240.0
#   Roughly guessed*
#   The global optimum is (x1, x2, x3, x4; f) =
#   (1.100000, 0.600000, 56.99482, 51.00125; 7019.031).
#
#   Source:
#     Lampinen, Jouni, and Zelinka, Ivan (1999).
#     Mechanical engineering design optimization
#     by differential evolution.
#     In: David Corne, Marco Dorigo and Fred Glover (Editors),
#     New Ideas in Optimization, McGraw-Hill, pp 127-146
pressure_vessel_A <-
    list(obj = function(x) {
             x1 <- x[1]; x2 <- x[2]; x3 <- x[3]; x4 <- x[4]
             0.6224*x1*x3*x4 + 1.7781*x2*x3^2 +
                 3.1611*x1^2*x4 + 19.84*x1^2*x3
         },
         con = function(x) {
             x1 <- x[1]; x2 <- x[2]; x3 <- x[3]; x4 <- x[4]
             c(0.0193*x3 - x1,
               0.00954*x3 - x2,
               750.0*1728.0 - pi*x3^2*x4 - 4/3*pi*x3^3)
         })

JDEoptim(c( 1.1,  0.6,   0.0,   0.0),
         c(12.5, 12.5, 240.0, 240.0),
         fn = pressure_vessel_A$obj,
         constr = pressure_vessel_A$con,
         tol = 1e-7, trace = TRUE, triter = 50)

# Mixed integer nonlinear programming --------------------

#   Designing a pressure vessel
#   Case B: solved according to the original problem statements
#           steel plate available in thicknesses multiple
#           of 0.0625 inch
#
#   wall thickness of the
#   shell 1.1 [18*0.0625] <= x1 <= 12.5 [200*0.0625]
#   heads 0.6 [10*0.0625] <= x2 <= 12.5 [200*0.0625]
#         0.0 <= x3 <= 240.0, 0.0 <= x4 <= 240.0
#   The global optimum is (x1, x2, x3, x4; f) =
#   (1.125 [18*0.0625], 0.625 [10*0.0625],
#    58.29016, 43.69266; 7197.729).
pressure_vessel_B <-
```

```
    list(obj = function(x) {
             x1 <- floor(x[1])*0.0625
             x2 <- floor(x[2])*0.0625
             x3 <- x[3]; x4 <- x[4]
             0.6224*x1*x3*x4 + 1.7781*x2*x3^2 +
                 3.1611*x1^2*x4 + 19.84*x1^2*x3
         },
         con = function(x) {
             x1 <- floor(x[1])*0.0625
             x2 <- floor(x[2])*0.0625
             x3 <- x[3]; x4 <- x[4]
             c(0.0193*x3 - x1,
               0.00954*x3 - x2,
               750.0*1728.0 - pi*x3^2*x4 - 4/3*pi*x3^3)
         })

res <- JDEoptim(c( 18,    10,     0.0,   0.0),
                c(200+1, 200+1, 240.0, 240.0),
                fn = pressure_vessel_B$obj,
                constr = pressure_vessel_B$con,
                tol = 1e-7, trace = TRUE, triter = 50)
res
# Now convert to integer x1 and x2
c(floor(res$par[1:2]), res$par[3:4])
```

NCDEoptim *Bound-Constrained and Nonlinear Constrained Multimodal Optimization via Differential Evolution*

Description

A bespoke implementation of the 'NCDE' (neighborhood based crowding DE) algorithm by Qu *et al.* (2012) doi:10.1109/TEVC.2011.2161873, assisted with the dynamic archive mechanism of Epitropakis *et al.* (2013) doi:10.1109/CEC.2013.6557556.

Usage

```
NCDEoptim(lower, upper, fn,
          constr = NULL, meq = 0, eps = 1e-5,
          crit = 1e-5, niche_radius = NULL, archive_size = 100,
          reinit_if_solu_in_arch = TRUE,
          NP = 100, Fl = 0.1, Fu = 1, CRl = 0, CRu = 1.1,
          nbngbrsl = NP/20, nbngbrsu = NP/5,
          tau_F = 0.1, tau_CR = 0.1, tau_pF = 0.1,
          tau_nbngbrs = 0.1,
          jitter_factor = 0.001,
          maxiter = 2000,
          add_to_init_pop = NULL,
```

```
            trace = FALSE, triter = 1,
            ...)
```

Arguments

lower, upper	numeric vectors, the lower and upper bounds of the search space (*box constraints*); must be finite (is.finite).
fn	a function to be **minimized** that takes a numeric vector X_i as first argument and returns the value of the objective.
constr	a vector function specifying the **left-hand side** of equality constraints defined to equal zero ($h_j(X_i) = 0,\ j = 1, \ldots, \text{meq}$), followed by inequality constraints defined as lesser than or equal to zero ($g_j(X_i) \leq 0,\ j = \text{meq} + 1, \ldots$). This function takes X_i as its first argument and returns a numeric vector with the same length of the total number of constraints. It defaults to NULL, which means that **bound-constrained** minimization is used.
meq	an integer, the first meq constraints are *equality* constraints whereas the remaining ones are *inequality* constraints. Defaults to 0 (inequality constraints only).
eps	the maximal admissible constraint violation for equality constraints. A numeric vector of small positive tolerance values with length meq used in the transformation of equalities into inequalities of the form $\lvert h_j(X_i)\rvert - \epsilon \leq 0$. A scalar value is expanded to apply to all equality constraints. Default is 1e-5.
crit	a numeric, the acceptance threshold on the archive strategy. If isTRUE(all.equal(fn(X_best_so_far_i fn(X_i), tolerance = crit)), a solution X_i is checked for possible insertion into the dynamic archive. Defaults to 1e-5.
niche_radius	a numeric, the absolute tolerance used to decide whether the solution X_i is *identical* to an already existing local or global solution *in the archive*. It defaults to NULL, meaning that the niche radius is adaptively chosen during the search. Results are **much better** if one is able to provide a reasonable value.
archive_size	an integer, the maximum number of solutions that can be kept in the archive; entries above this limit are discarded. Default is 100.
reinit_if_solu_in_arch	a logical, if TRUE, any solution X_i already in the archive **reinitializes** its nearest neighbor *in the population* within the range $[\text{lower}, \text{upper}]$. Default is TRUE.
NP	an integer, the population size. Defaults to 100.
Fl	a numeric, the minimum value that the *scaling factor* F could take. It defaults to 0.1.
Fu	a numeric, the maximum value that the *scaling factor* F could take. It defaults to 1.
CRl	a numeric, the minimum value to be used for the *crossover constant* CR. It defaults to 0.
CRu	a numeric, the maximum value to be used for the *crossover constant* CR. It defaults to 1.1.
nbngbrsl	an integer, the lower limit for the *neighborhood size* nbngbrs. It defaults to 1/20 of the population size.

nbngbrsu	an integer, the upper limit for the *neighborhood size* nbngbrs. It defaults to 1/5 of the population size.
tau_F	a numeric, the probability that the *scaling factor* F is updated. Defaults to 0.1.
tau_CR	a numeric, the probability that the *crossover constant* CR is updated. Defaults to 0.1.
tau_pF	a numeric, the probability that the *mutation probability* p_F in the mutation strategy DE/rand/1/either-or is updated. Defaults to 0.1.
tau_nbngbrs	a numeric, the probability that the *neighborhood size* nbngbrs is updated. Defaults to 0.1.
jitter_factor	a numeric, the tuning constant for *jitter*. If NULL only *dither* is used. Default is 0.001.
maxiter	an integer, the maximum number of iterations allowed which is the **stopping condition**. Default is 2000.
add_to_init_pop	numeric vector of length length(lower) or column-wise matrix with length(lower) rows specifying initial candidate solutions which are appended to the randomly generated initial population. Default is NULL.
trace	a logical, determines whether or not to monitor the iteration process. Default is FALSE.
triter	an integer, trace output is printed at every triter iterations. Default is 1.
...	additional arguments passed to fn and constr.

Details

This implementation differs mainly from the original 'NCDE' algorithm of Qu *et al.* (2012) by employing the archiving procedure proposed in Epitropakis *et al.* (2013) and the adaptive 'jDE' strategy instead of canonical Diferential Evolution. The key reason for archiving good solutions during the search process is to prevent them from being lost during evolution. Constraints are tackled through the ε-constrained method as proposed in Poole and Allen (2019). The 'jDE' and ε-constrained mechanisms are applied in the same way as in JDEoptim, but with *synchronous* mode of population update. In contrast, the reinitialization in the current population triggered by already found solutions is done *asynchronously*.

Each line of trace output follows the format of:

```
iteration : < value of niche radius > population>> ( value of best solution ) best solution
{ index of violated constraints } archive>> [ number of solutions found ] ( value of best
solution ) best solution
```

Value

A list with the following components:

solution_arch	a matrix whose columns are the local and global minima stored in the **archive** of feasible solutions in ascending order of the objective function values.
objective_arch	the values of $\mathrm{fn}(X_i)$ for the corresponding columns of solution_arch.

`solution_pop` a `matrix` whose columns are the local and global minima stored in the **final population** in ascending order of the objective function values; feasible solutions come first followed by the infeasible ones.

`objective_pop` the values of $\mathrm{fn}(X_i)$ for the corresponding columns of `solution_pop`.

`iter` the number of iterations used.

and if there are general constraints present:

`constr_value_arch`
: a `matrix` whose columns contain the values of the constraints for `solution_arch`.

`constr_value_pop`
: a `matrix` whose columns contain the values of the constraints for `solution_pop`.

Note

This function is in an experimental stage.

Author(s)

Eduardo L. T. Conceicao <mail@eduardoconceicao.org>

References

Epitropakis, M. G., Li, X. and Burke, E. K. (2013) A dynamic archive niching differential evolution algorithm for multimodal optimization; in *2013 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, pp. 79–86. doi:10.1109/CEC.2013.6557556.

Poole, D. J. and Allen, C. B. (2019) Constrained niching using differential evolution. *Swarm and Evolutionary Computation* **44**, 74–100. doi:10.1016/j.swevo.2018.11.004.

Qu, B. Y., Suganthan, P. N. and Liang, J. J. (2012) Differential evolution with neighborhood mutation for multimodal optimization. *IEEE Transactions on Evolutionary Computation* **16**, 601–614. doi:10.1109/TEVC.2011.2161873.

Examples

```
# NOTE: Examples were excluded from testing
#       to reduce package check time.

# Use a preset seed so test values are reproducible.
set.seed(1234)

# Warning: the examples are run using a very small number of
# iterations to decrease execution time.

# Bound-constrained optimization --------------------

#   Vincent function
#
#   f(x) = -mean(sin(10*log(x)))
#
#   0.25 <= xi <= 10, i = {1, 2, ..., n}
```

```
#   The function has 6^n global minima without local minima.

NCDEoptim(c(0.25, 0.25), c(10, 10),
          function(x) -mean(sin(10*log(x))),
          niche_radius = 0.2,
          maxiter = 200, trace = TRUE, triter = 20)

# Nonlinear constrained optimization --------------------

#   Function F10 of Poole and Allen (2019)
#
#   f(x) = -sin(5*pi*x)^6 + 1
#   subject to:
#   g(x) = -cos(10*pi*x) <= 0
#
#   0 <= x <= 1
#   The 10 global optima are
#   (x1*, ..., x10*; f*) = ((2*(1:10) - 1)/20; 0.875).

NCDEoptim(0, 1,
          function(x) -sin(5*pi*x)^6 + 1,
          function(x) -cos(10*pi*x),
          niche_radius = 0.05,
          maxiter = 200, trace = TRUE, triter = 20)
```

SPJDEoptim *Simple Parallel Version of* JDEoptim()

Description

Parallel evaluation of the objective function and/or constraints for each candidate solution in the population.

Usage

```
SPJDEoptim(lower, upper, fn,
           constr = NULL, meq = 0, eps = 1e-5,
           sequential_eval = c("none", "constraints",
                               "objective", "both"),
           ..., further_args = list(),
           NP = 10*length(lower),
           Fl = 0.1, Fu = 1, CRl = 0, CRu = 1.1,
           tau_F = 0.1, tau_CR = 0.1, tau_pF = 0.1,
           jitter_factor = 0.001,
           tol = 1e-15, maxiter = 2000*length(lower),
           fnscale = 1, compare_to = c("median", "max"),
           add_to_init_pop = NULL,
           trace = FALSE, triter = 1,
           details = FALSE)
```

Arguments

`lower`, `upper`	numeric vectors, the lower and upper bounds of the search space (*box constraints*); must be finite (`is.finite`).
`fn`	a `function` to be **minimized** that takes a numeric vector X_i as first argument and returns the value of the objective.
`constr`	a vector `function` specifying the **left-hand side** of equality constraints defined to equal zero ($h_j(X_i) = 0,\ j = 1, \ldots, \text{meq}$), followed by inequality constraints defined as lesser than or equal to zero ($g_j(X_i) \leq 0,\ j = \text{meq} + 1, \ldots$). This function takes X_i as its first argument and returns a numeric vector with the same length of the total number of constraints. It defaults to `NULL`, which means that **bound-constrained** minimization is used.
`meq`	an integer, the first `meq` constraints are *equality* constraints whereas the remaining ones are *inequality* constraints. Defaults to `0` (inequality constraints only).
`eps`	the maximal admissible constraint violation for equality constraints. A numeric vector of small positive tolerance values with length `meq` used in the transformation of equalities into inequalities of the form $\lvert h_j(X_i)\rvert - \epsilon \leq 0$. A scalar value is expanded to apply to all equality constraints. Default is `1e-5`.
`sequential_eval`	a character string that indicates when either the objective function (if `"objective"`) or the constraints (if `"constraints"`) should be evaluated in a serial instead of parallel manner whenever their computational cost is low. If `"both"` the computation is purely sequential, being intended for debugging and testing. Defaults to `"none"`.
`...`	[*to be used only when run in parallel mode*] named arguments specifying **locally-defined** functions and other objects required but not defined in `fn` and/or `constr`.
`further_args`	a list of any further arguments passed to `fn` and `constr`. Defaults to an empty list.
`NP`	an integer, the population size. Defaults to `10*length(lower)`.
`Fl`	a numeric, the minimum value that the *scaling factor* `F` could take. It defaults to `0.1`.
`Fu`	a numeric, the maximum value that the *scaling factor* `F` could take. It defaults to `1`.
`CRl`	a numeric, the minimum value to be used for the *crossover constant* `CR`. It defaults to `0`.
`CRu`	a numeric, the maximum value to be used for the *crossover constant* `CR`. It defaults to `1.1`.
`tau_F`	a numeric, the probability that the *scaling factor* `F` is updated. Defaults to `0.1`.
`tau_CR`	a numeric, the probability that the *crossover constant* `CR` is updated. Defaults to `0.1`.
`tau_pF`	a numeric, the probability that the *mutation probability* p_F in the mutation strategy DE/rand/1/either-or is updated. Defaults to `0.1`.
`jitter_factor`	a numeric, the tuning constant for *jitter*. If `NULL` only *dither* is used. Default is `0.001`.

`tol`	a numeric, the tolerance for the stopping criterion. Default is `1e-15`.
`maxiter`	an integer, the maximum number of iterations allowed. Default is `2000*length(lower)`.
`fnscale`	a numeric, the typical magnitude of `fn` used in the *stopping criterion*. Defaults to `1`. See 'Details' in `JDEoptim()`.
`compare_to`	a character string controlling which function should be applied when evaluating the *stopping criterion*. It defaults to `"median"`. See 'Details' in `JDEoptim()`.
`add_to_init_pop`	numeric vector of length `length(lower)` or column-wise `matrix` with `length(lower)` rows specifying initial candidate solutions which are appended to the randomly generated initial population. Default is `NULL`.
`trace`	logical indicating whether or not to monitor the iteration process. Default is `FALSE`.
`triter`	an integer, trace output is printed at every `triter` iterations. Default is `1`.
`details`	logical indicating if the output will contain the solutions in the final population and their respective `fn` values. Defaults to `FALSE`.

Details

The optimizer is intended for dealing with computationally expensive objective functions and/or constraints (see, *e.g.*, Storn and Price, 2025). The algorithm has been parallelized simply by distributing the evaluation of the candidate solutions in the population over the available processors/cores with `mirai_map()` from package **mirai**.

It is important to note this approach and its implementation **is not equivalent** to the purely sequential version `JDEoptim()`, the major reason being the fact that an asynchronous update of the current population is not possible anymore. This in turn almost always has a **deleterious effect** on the efficiency of DE. For this reason although it can run **without mirai installed**, for example using `sequential_eval = "both"`, the use of `JDEoptim()` is preferred over `SPJDEoptim()` in this case.

Note that all support for the parallel computation is provided by package **mirai** and controlled directly by end-users; in particular, resources are declared with `mirai::daemons()` **before invoking** `SPJDEoptim()`.

Value

A list with the following components:

`par`	the best solution X_i found.
`value`	the value of $\mathrm{fn}(X_i)$ corresponding to `par`.
`iter`	the number of iterations used.
`convergence`	an integer code; `0` indicates successful completion, whereas `1` indicates that the iteration limit `maxiter` has been reached.

and if `details = TRUE`:

`poppar`	a `matrix` whose columns are the solutions remaining in the final population.
`popcost`	the values of $\mathrm{fn}(X_i)$ for the corresponding columns of `poppar`.

Note

Additional arguments are passed to `fn` and `constr` through `further_args` instead of `...` such as in `JDEoptim()`.

Author(s)

Eduardo L. T. Conceicao <mail@eduardoconceicao.org>

References

https://mirai.r-lib.org

Storn, Rainer and Price, Kenneth V. (2025) Multi-child DE — a massively parallel differential evolution algorithm; in *2025 IEEE Symposium on Computational Intelligence on Engineering/Cyber Physical Systems (CIES)*, Trondheim, Norway. doi:10.1109/CIES64955.2025.11007629.

Examples

```
if (requireNamespace("mirai", quietly = TRUE)) {
# NOTE: Examples were excluded from testing
#       to reduce package check time.

# stays within 2-core limit
mirai::daemons(1, dispatcher = FALSE)
set.seed(42)

# Warning: the examples are run using a very small number of
# iterations to decrease execution time. Because they are very
# cheap to compute, parallel evaluation is actually detrimental.

#   Shekel 5 problem
#
#   0 <= xj <= 10, j = {1, 2, 3, 4}
#   There are five local minima and the global minimizer is
#   located at x* = (4.00, 4.00, 4.00, 4.00) with
#   f(x*) ~~ -10.1499.
#
#   Source:
#     Ali, M. Montaz, Khompatraporn, Charoenchai, and
#     Zabinsky, Zelda B. (2005).
#     A numerical evaluation of several stochastic algorithms
#     on selected continuous global optimization test problems.
#     Journal of Global Optimization 31, 635-672.
#     https://doi.org/10.1007/s10898-004-9972-2
a_ij <- matrix(c(4, 4, 4, 4,
                 1, 1, 1, 1,
                 8, 8, 8, 8,
                 6, 6, 6, 6,
                 3, 7, 3, 7), ncol = 4, byrow = TRUE)
c_i <- c(0.1, 0.2, 0.2, 0.4, 0.4)
a_ji <- t(a_ij)
shekel5 <- function(x) {
```

```
    -sum(1/( colSums((x - a_ji)^2) + c_i ))
}

SPJDEoptim(rep(0, 4), rep(10, 4), shekel5,
           a_ji = a_ji, c_i = c_i,
           maxiter = 10, trace = TRUE)

#   Sinusoidal problem
#
#   0 <= xi <= 180, i = {1, 2, ..., n}
#   The variable x is in degrees. Parameter A affects the
#   amplitude of the global optimum; B affects the peridiocity
#   and hence the number of local minima; z shifts the location
#   of the global minimum. The location of the global solution
#   is at x* = (90+z, 90+z, ..., 90+z) with the global optimum
#   value of f(x*) = -(A+1).
#
#   Source:
#     Ali, M. Montaz, Khompatraporn, Charoenchai, and
#     Zabinsky, Zelda B. (2005).
#     A numerical evaluation of several stochastic algorithms
#     on selected continuous global optimization test problems.
#     Journal of Global Optimization 31, 635-672.
#     https://doi.org/10.1007/s10898-004-9972-2
sinusoidal <- function(x, A, B, z) {
    var_in_rad <- (x - z)*pi/180
    -( A*prod(sin(var_in_rad)) + prod(sin(B*var_in_rad)) )
}

SPJDEoptim(rep(0, 10), rep(180, 10), sinusoidal,
           further_args = list(A = 2.5, B = 5, z = 30),
           maxiter = 10, trace = TRUE)

mirai::daemons(0) # reset
Sys.sleep(1)
}
```

Index