Package 'DEoptimR'

October 19, 2014

Version 1.0-2

Date 2014-10-15

Title Differential Evolution Optimization in pure R

Author Eduardo L. T. Conceicao and Martin Maechler [ctb]

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

Description An implementation of the jDE variant of the Differential Evolution stochastic algorithm for global optimization of nonlinear programming problems.

Enhances robustbase

License GPL (>= 2)

ByteCompile yes

NeedsCompilation no

Repository CRAN

Date/Publication 2014-10-19 11:53:04

R topics documented:

JDEoptim *Nonlinear Constrained Optimization via Differential Evolution*

Description

An implementation of the jDE variant of the Differential Evolution stochastic algorithm for global optimization of nonlinear programming problems.

Usage

```
JDEoptim(lower, upper, fn,
    constr = NULL, meq = 0, eps = 1e-05,
    NP = 10*d, Fl = 0.1, Fu = 1,
    tau1 = 0.1, tau2 = 0.1, tau3 = 0.1,
    jitter_factor = 0.001,
    tol = 1e-15, maxiter = 200*d, fnscale = 1,
    FUN = c("median", "max"),
    add_to_init_pop = NULL, trace = FALSE, triter = 1,
    details = FALSE, ...)
```

Arguments

`lower, upper`	numeric vectors of *lower* or *upper* bounds, respectively, 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 nonlinear constraints under which we want to minimize `fn`. They should be given in the form $h_i(x) = 0, g_i(x) \leq 0$. 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, all the remaining as *inequality* constraints. Defaults to `0` (inequality constraints only).
`eps`	an optional real vector of small positive tolerance values with length `meq` used in the transformation of equalities into inequalities of the form $\|h_i(x)\| - \epsilon \leq 0$. 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.

tau1	an optional scalar which represents a probability in the mutation strategy DE/rand/1/either-or. Defaults to 0.1.
tau2	an optional scalar which represents the probability that the *scaling factor* F is updated. Defaults to 0.1, which is almost always satisfactory.
tau3	an optional constant value which represents the probability that the *crossover probability* CR is updated. Defaults to 0.1, which is almost always satisfactory.
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 200*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'.
FUN	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 that controls the frequency of tracing when trace = TRUE. Default is triter = 1, which means that iteration : < value of stopping test > (value of bes is printed at every 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() if that is not NULL.

Details

The setting of the *control parameters* of standard 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 its use of only a *single population* (Babu and Angira 2006) instead of separate current and child populations as in classical DE.

Constraint handling is done using the approach described in Zhang and Rangaiah (2012).

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* for the evaluation of the objective function. This is because DE itself works with 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.

The algorithm is stopped if

$$\frac{\mathrm{FUN}\{[\mathrm{fn}(x_1), \ldots, \mathrm{fn}(x_{\mathrm{npop}})]\} - \mathrm{fn}(x_{\mathrm{best}})}{\mathrm{fnscale}} \leq \mathrm{tol}$$

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)`. This is a variant of the *Diff* criterion studied by Zielinski and Laur (2008), 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.

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.

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, pp. 117–118.

Storn, R. (2008) Differential evolution research — trends and open questions; in Chakraborty, U. K., Ed., *Advances in differential evolution*. SCI 143, Springer-Verlag, Berlin, pp. 11–12.

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.

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.

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-Verlag, Berlin, pp. 111–138.

See Also

Function `DEoptim()` in the **DEoptim** package has many more options than `JDEoptim()`, but does not allow constraints in the same flexible manner.

Examples

```
# 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.
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
```

```
#   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.
fcn <-
    list(obj = function(x) {
             35*x[1]^0.6 + 35*x[2]^0.6
         },
         eq = 2,
         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
#
#   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])
```

Index