

# Package ‘RcppDE’

May 7, 2026

**Version** 0.1.9

**Date** 2026-03-10

**Title** Global Optimization by Differential Evolution in C++

**Description** An efficient C++ based implementation of the 'DEoptim' function which performs global optimization by differential evolution. Its creation was motivated by trying to see if the old approximation ``easier, shorter, faster: pick any two" could in fact be extended to achieving all three goals while moving the code from plain old C to modern C++. The initial version did in fact do so, but a good part of the gain was due to an implicit code review which eliminated a few inefficiencies which have since been eliminated in 'DEoptim'.

**License** GPL (>= 2)

**Imports** Rcpp

**Suggests** inline, DEoptim, lattice

**LinkingTo** Rcpp, RcppArmadillo

**URL** <https://github.com/eddelbuettel/rcppde>,  
<https://dirk.eddelbuettel.com/code/rcpp.de.html>

**BugReports** <https://github.com/eddelbuettel/rcppde/issues>

**NeedsCompilation** yes

**Author** Dirk Eddelbuettel [aut, cre] (ORCID:  
<<https://orcid.org/0000-0001-6419-907X>>),  
David Ardia [aut] (Author of DEoptim),  
Katharine Mullen [aut] (Author of DEoptim),  
Brian Peterson [aut] (Author of DEoptim),  
Joshua Ulrich [aut] (Author of DEoptim),  
Kris Boudt [ctb] (Contributor to DEoptim),  
Rainer Storn [aut] (Principal author of DE-Engine)

**Maintainer** Dirk Eddelbuettel <edd@debian.org>

**Repository** CRAN

**Date/Publication** 2026-03-11 06:30:32 UTC

## Contents

DEoptim . . . . .	2
DEoptim-methods . . . . .	7
DEoptim.control . . . . .	9
<b>Index</b>	<b>13</b>

---

DEoptim	<i>Differential Evolution Optimization</i>
---------	--

---

### Description

Performs evolutionary global optimization via the Differential Evolution algorithm.

### Usage

```
DEoptim(fn, lower, upper, control = DEoptim.control(), ...)
```

### Arguments

fn	the function to be optimized (minimized). The function should have as its first argument the vector of real-valued parameters to optimize, and return a scalar real result. NA and NaN values are not allowed. Note that fn can also be an external pointer object encapsulating a C/C++-level function pointer to a compiled functions which may offer considerable speed improvements.
lower, upper	two vectors specifying scalar real lower and upper bounds on each parameter to be optimized, so that the i-th element of lower and upper applied to the i-th parameter. The implementation searches between lower and upper for the global optimum (minimum) of fn.
control	a list of control parameters; see <a href="#">DEoptim.control</a> .
...	further arguments to be passed to fn.

### Details

DEoptim performs optimization (minimization) of fn.

The control argument is a list; see the help file for [DEoptim.control](#) for details.

The R implementation of Differential Evolution (DE), **DEoptim**, was first published on the Comprehensive R Archive Network (CRAN) in 2005 by David Ardia. Early versions were written in pure R. Since version 2.0-0 (published to CRAN in 2009) the package has relied on an interface to a C implementation of DE, which is significantly faster on most problems as compared to the implementation in pure R. The C interface is in many respects similar to the MS Visual C++ v5.0 implementation of the Differential Evolution algorithm distributed with the book *Differential Evolution – A Practical Approach to Global Optimization* by Price, K.V., Storn, R.M., Lampinen J.A., Springer-Verlag, 2006. Since version 2.0-3 the C implementation dynamically allocates the memory required to store the population, removing limitations on the number of members in the population and length of the parameter vectors that may be optimized. Since becoming publicly available,

the package **DEoptim** has been used by several authors to solve optimization problems arising in diverse domains; see Mullen et al. (2009) for a review.

To perform a maximization (instead of minimization) of a given function, simply define a new function which is the opposite of the function to maximize and apply `DEoptim` to it.

To integrate additional constraints (than box constraints) on the parameters  $x$  of  $fn(x)$ , for instance  $x[1] + x[2]^2 < 2$ , integrate the constraint within the function to optimize, for instance:

```
fn <- function(x){
  if (x[1] + x[2]^2 < 2){
    r <- Inf
  } else{
    ...
  }
  return(r)
}
```

This simplistic strategy usually does not work all that well for gradient-based or Newton-type methods. It is likely to be alright when the solution is in the interior of the feasible region, but when the solution is on the boundary, optimization algorithm would have a difficult time converging. Furthermore, when the solution is on the boundary, this strategy would make the algorithm converge to an inferior solution in the interior. However, for methods such as DE which are not gradient based, this strategy might not be that bad.

Note that `DEoptim` stops if any NA or NaN value is obtained. You have to redefine your function to handle these values (for instance, set NA to Inf in your objective function).

It is important to emphasize that the result of `DEoptim` is a random variable, i.e., different results will obtain when the algorithm is run repeatedly with the same settings. Hence, the user should set the random seed if they want to reproduce the results, e.g., by setting `set.seed(1234)` before the call of `DEoptim`.

`DEoptim` relies on repeated evaluation of the objective function in order to move the population toward a global minimum. Users interested in making `DEoptim` run as fast as possible should ensure that evaluation of the objective function is as efficient as possible. Using pure R code, this may often be accomplished using vectorization. Writing parts of the objective function in a lower-level language like C or Fortran may also increase speed.

Further details and examples of the R package **DEoptim** can be found in Mullen et al. (2009) and Ardia et al. (2010).

Please cite the package in publications.

## Value

The output of the function `DEoptim` is a member of the S3 class `DEoptim`. More precisely, this is a list (of length 2) containing the following elements:

`optim`, a list containing the following elements:

- `bestmem`: the best set of parameters found.

- `bestval`: the value of `fn` corresponding to `bestmem`.
- `nfeval`: number of function evaluations.
- `iter`: number of procedure iterations.

member, a list containing the following elements:

- `lower`: the lower boundary.
- `upper`: the upper boundary.
- `bestvalit`: the best value of `fn` at each iteration.
- `bestmemit`: the best member at each iteration.
- `pop`: the population generated at the last iteration.
- `storepop`: a list containing the intermediate populations.

Members of the class `DEoptim` have a `plot` method that accepts the argument `plot.type`. `plot.type = "bestmemit"` results in a plot of the parameter values that represent the lowest value of the objective function each generation. `plot.type = "bestvalit"` plots the best value of the objective function each generation. Finally, `plot.type = "storepop"` results in a plot of stored populations (which are only available if these have been saved by setting the `control` argument of `DEoptim` appropriately). Storing intermediate populations allows us to examine the progress of the optimization in detail. A summary method also exists and returns the best parameter vector, the best value of the objective function, the number of generations optimization ran, and the number of times the objective function was evaluated.

### Note

*Differential Evolution* (DE) is a search heuristic introduced by Storn and Price (1997). Its remarkable performance as a global optimization algorithm on continuous numerical minimization problems has been extensively explored; see Price et al. (2006). DE belongs to the class of genetic algorithms which use biology-inspired operations of crossover, mutation, and selection on a population in order to minimize an objective function over the course of successive generations (see Mitchell, 1998). As with other evolutionary algorithms, DE solves optimization problems by evolving a population of candidate solutions using alteration and selection operators. DE uses floating-point instead of bit-string encoding of population members, and arithmetic operations instead of logical operations in mutation. DE is particularly well-suited to find the global optimum of a real-valued function of real-valued parameters, and does not require that the function be either continuous or differentiable.

Let  $NP$  denote the number of parameter vectors (members)  $x \in R^d$  in the population. In order to create the initial generation,  $NP$  guesses for the optimal value of the parameter vector are made, either using random values between lower and upper bounds (defined by the user) or using values given by the user. Each generation involves creation of a new population from the current population members  $\{x_i \mid i = 1, \dots, NP\}$ , where  $i$  indexes the vectors that make up the population. This is accomplished using *differential mutation* of the population members. An initial mutant parameter vector  $v_i$  is created by choosing three members of the population,  $x_{r_0}$ ,  $x_{r_1}$  and  $x_{r_2}$ , at random. Then  $v_i$  is generated as

$$v_i \doteq x_{r_0} + F \cdot (x_{r_1} - x_{r_2})$$

where  $F$  is a positive scale factor, effective values for which are typically less than one. After the first mutation operation, mutation is continued until  $d$  mutations have been made, with a crossover probability  $CR \in [0, 1]$ . The crossover probability  $CR$  controls the fraction of the parameter values that are copied from the mutant. If an element of the trial parameter vector is found to violate the bounds after mutation and crossover, it is reset in such a way that the bounds are respected (with the specific protocol depending on the implementation). Then, the objective function values associated with the children are determined. If a trial vector has equal or lower objective function value than the previous vector it replaces the previous vector in the population; otherwise the previous vector remains. Variations of this scheme have also been proposed; see Price et al. (2006) and [DEoptim.control](#).

Intuitively, the effect of the scheme is that the shape of the distribution of the population in the search space is converging with respect to size and direction towards areas with high fitness. The closer the population gets to the global optimum, the more the distribution will shrink and therefore reinforce the generation of smaller difference vectors.

As a general advice regarding the choice of  $NP$ ,  $F$  and  $CR$ , Storn et al. (2006) state the following: Set the number of parents  $NP$  to 10 times the number of parameters, select weighting factor  $F = 0.8$  and crossover constant  $CR = 0.9$ . Make sure that you initialize your parameter vectors by exploiting their full numerical range, i.e., if a parameter is allowed to exhibit values in the range  $[-100, 100]$  it is a good idea to pick the initial values from this range instead of unnecessarily restricting diversity. If you experience misconvergence in the optimization process you usually have to increase the value for  $NP$ , but often you only have to adjust  $F$  to be a little lower or higher than 0.8. If you increase  $NP$  and simultaneously lower  $F$  a little, convergence is more likely to occur but generally takes longer, i.e., DE is getting more robust (there is always a convergence speed/robustness trade-off).

DE is much more sensitive to the choice of  $F$  than it is to the choice of  $CR$ .  $CR$  is more like a fine tuning element. High values of  $CR$  like  $CR = 1$  give faster convergence if convergence occurs. Sometimes, however, you have to go down as much as  $CR = 0$  to make DE robust enough for a particular problem. For more details on the DE strategy, we refer the reader to Storn and Price (1997) and Price et al. (2006).

#### Author(s)

For **RcppDE**: Dirk Eddelbuettel.

For **DEoptim**: David Ardia, Katharine Mullen <katharine.mullen@nist.gov>, Brian Peterson and Joshua Ulrich.

#### References

- Storn, R. and Price, K. (1997) Differential Evolution – A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces, *Journal of Global Optimization*, 11:4, 341–359.
- Price, K.V., Storn, R.M., Lampinen J.A. (2006) *Differential Evolution - A Practical Approach to Global Optimization*. Berlin Heidelberg: Springer-Verlag. ISBN 3540209506.
- Mitchell, M. (1998) *An Introduction to Genetic Algorithms*. The MIT Press. ISBN 0262631857.
- Mullen, K.M., Ardia, D., Gil, D.L., Windover, D., Cline, J. (2011) **DEoptim**: An R Package for Global Optimization by Differential Evolution. [doi:10.18637/jss.v040.i06](https://doi.org/10.18637/jss.v040.i06)
- Ardia, D., Boudt, K., Carl, P., Mullen, K.M., Peterson, B.G. (2010) Differential Evolution (**DEoptim**) for Non-Convex Portfolio Optimization. [doi:10.32614/RJ2011005](https://doi.org/10.32614/RJ2011005)

**See Also**

[DEoptim.control](#) for control arguments, [DEoptim-methods](#) for methods on DEoptim objects, including some examples in plotting the results; [optim](#) or [constrOptim](#) for alternative optimization algorithms.

**Examples**

```
## Rosenbrock Banana function
## The function has a global minimum  $f(x) = 0$  at the point  $(0,0)$ .
## Note that the vector of parameters to be optimized must be the first
## argument of the objective function passed to DEoptim.
Rosenbrock <- function(x){
  x1 <- x[1]
  x2 <- x[2]
  100 * (x2 - x1 * x1)^2 + (1 - x1)^2
}

## DEoptim searches for minima of the objective function between
## lower and upper bounds on each parameter to be optimized. Therefore
## in the call to DEoptim we specify vectors that comprise the
## lower and upper bounds; these vectors are the same length as the
## parameter vector.
lower <- c(-10,-10)
upper <- -lower

## run DEoptim and set a seed first for replicability
set.seed(1234)
DEoptim(Rosenbrock, lower, upper)

## increase the population size
DEoptim(Rosenbrock, lower, upper, DEoptim.control(NP = 100))

## change other settings and store the output
outDEoptim <- DEoptim(Rosenbrock, lower, upper, DEoptim.control(NP = 80,
  itermax = 400, F = 1.2, CR = 0.7))

## plot the output
plot(outDEoptim)

## 'Wild' function, global minimum at about -15.81515
Wild <- function(x)
  10 * sin(0.3 * x) * sin(1.3 * x^2) +
  0.00001 * x^4 + 0.2 * x + 80

plot(Wild, -50, 50, n = 1000, main = "'Wild function'")

outDEoptim <- DEoptim(Wild, lower = -50, upper = 50,
  control = DEoptim.control(trace = FALSE))

plot(outDEoptim)

DEoptim(Wild, lower = -50, upper = 50,
```

```
control = DEoptim.control(NP = 50)
```

---

DEoptim-methods

*DEoptim-methods*


---

## Description

Methods for DEoptim objects.

## Usage

```
## S3 method for class 'DEoptim'
summary(object, ...)
## S3 method for class 'DEoptim'
plot(x, plot.type = c("bestmemit", "bestvalit", "storepop"), ...)
```

## Arguments

object	an object of class DEoptim; usually, a result of a call to <a href="#">DEoptim</a> .
x	an object of class DEoptim; usually, a result of a call to <a href="#">DEoptim</a> .
plot.type	should we plot the best member at each iteration, the best value at each iteration or the intermediate populations?
...	further arguments passed to or from other methods.

## Details

Members of the class DEoptim have a plot method that accepts the argument plot.type. plot.type = "bestmemit" results in a plot of the parameter values that represent the lowest value of the objective function each generation. plot.type = "bestvalit" plots the best value of the objective function each generation. Finally, plot.type = "storepop" results in a plot of stored populations (which are only available if these have been saved by setting the control argument of DEoptim appropriately). Storing intermediate populations allows us to examine the progress of the optimization in detail. A summary method also exists and returns the best parameter vector, the best value of the objective function, the number of generations optimization ran, and the number of times the objective function was evaluated.

## Note

Further details and examples of the R package **DEoptim** can be found in Mullen et al. (2009) and Ardia et al. (2010).

Please cite the package in publications.

## Author(s)

For **RcppDE**: Dirk Eddelbuettel.

For **DEoptim**: David Ardia, Katharine Mullen <katharine.mullen@nist.gov>, Brian Peterson and Joshua Ulrich.

## References

Mullen, K.M., Ardia, D., Gil, D.L, Windover, D., Cline, J. (2011) **DEoptim**: An R Package for Global Optimization by Differential Evolution. doi:10.18637/jss.v040.i06

Ardia, D., Boudt, K., Carl, P., Mullen, K.M., Peterson, B.G. (2011) Differential Evolution (**DEoptim**) for Non-Convex Portfolio Optimization. doi:10.32614/RJ2011005

## See Also

[DEoptim](#) and [DEoptim.control](#).

## Examples

```
## Rosenbrock Banana function
## The function has a global minimum  $f(x) = 0$  at the point  $(0,0)$ .
## Note that the vector of parameters to be optimized must be the first
## argument of the objective function passed to DEoptim.
Rosenbrock <- function(x){
  x1 <- x[1]
  x2 <- x[2]
  100 * (x2 - x1 * x1)^2 + (1 - x1)^2
}

lower <- c(-10, -10)
upper <- -lower

set.seed(1234)
outDEoptim <- DEoptim(Rosenbrock, lower, upper)

## print output information
summary(outDEoptim)

## plot the best members
plot(outDEoptim, type = 'b')

## plot the best values
dev.new()
plot(outDEoptim, plot.type = "bestvalit", type = 'b', col = 'blue')

## rerun the optimization, and store intermediate populations
outDEoptim <- DEoptim(Rosenbrock, lower, upper,
  DEoptim.control(itermax = 500,
    storepopfrom = 1, storepopfreq = 2))
summary(outDEoptim)

## plot intermediate populations
dev.new()
plot(outDEoptim, plot.type = "storepop")

## Wild function
Wild <- function(x)
  10 * sin(0.3 * x) * sin(1.3 * x^2) +
```

```

0.00001 * x^4 + 0.2 * x + 80

outDEoptim = DEoptim(Wild, lower = -50, upper = 50,
                    DEoptim.control(trace = FALSE, storepopfrom = 50,
                                    storepopfreq = 1))

plot(outDEoptim, type = 'b')

dev.new()
plot(outDEoptim, plot.type = "bestvalit", type = 'b')

## Not run:
## an example with a normal mixture model: requires package mvtnorm
library(mvtnorm)

## neg value of the density function
negPdfMix <- function(x) {
  tmp <- 0.5 * dmvnorm(x, c(-3, -3)) + 0.5 * dmvnorm(x, c(3, 3))
  -tmp
}

## wrapper plotting function
plotNegPdfMix <- function(x1, x2)
  negPdfMix(cbind(x1, x2))

## contour plot of the mixture
x1 <- x2 <- seq(from = -10.0, to = 10.0, by = 0.1)
thexlim <- theylim <- range(x1)
z <- outer(x1, x2, FUN = plotNegPdfMix)

contour(x1, x2, z, nlevel = 20, las = 1, col = rainbow(20),
        xlim = thexlim, ylim = theylim)

set.seed(1234)
outDEoptim <- DEoptim(negPdfMix, c(-10, -10), c(10, 10),
                    DEoptim.control(NP = 100, itermax = 100, storepopfrom = 1,
                                    storepopfreq = 5))

## convergence plot
dev.new()
plot(outDEoptim)

## the intermediate populations indicate the bi-modality of the function
dev.new()
plot(outDEoptim, plot.type = "storepop")

## End(Not run)

```

**Description**

Allow the user to set some characteristics of the Differential Evolution optimization algorithm implemented in DEoptim.

**Usage**

```
DEoptim.control(VTR = -Inf, strategy = 2, bs = FALSE, NP = 50,
  itermax = 200, CR = 0.5, F = 0.8, trace = TRUE,
  initialpop = NULL, storepopfrom = itermax + 1,
  storepopfreq = 1, p = 0.2, c = 0, reltol = sqrt(.Machine$double.eps),
  steptol = itermax)
```

**Arguments**

VTR	the value to be reached. The optimization process will stop if either the maximum number of iterations <code>itermax</code> is reached or the best parameter vector <code>bestmem</code> has found a value <code>fn(bestmem) &lt;= VTR</code> . Default to <code>-Inf</code> .
strategy	defines the Differential Evolution strategy used in the optimization procedure: 1: DE / rand / 1 / bin (classical strategy) 2: DE / local-to-best / 1 / bin (default) 3: DE / best / 1 / bin with jitter 4: DE / rand / 1 / bin with per-vector-dither 5: DE / rand / 1 / bin with per-generation-dither 6: DE / current-to-p-best / 1 any value not above: variation to DE / rand / 1 / bin: either-or-algorithm. Default strategy is currently 2. See <i>*Details*</i> .
bs	if <code>FALSE</code> then every mutant will be tested against a member in the previous generation, and the best value will proceed into the next generation (this is standard trial vs. target selection). If <code>TRUE</code> then the old generation and <code>NP</code> mutants will be sorted by their associated objective function values, and the best <code>NP</code> vectors will proceed into the next generation (best of parent and child selection). Default is <code>FALSE</code> .
NP	number of population members. Defaults to 50. For many problems it is best to set <code>NP</code> to be at least 10 times the length of the parameter vector.
itermax	the maximum iteration (population generation) allowed. Default is 200.
CR	crossover probability from interval <code>[0,1]</code> . Default to 0.5.
F	step-size from interval <code>[0,2]</code> . Default to 0.8.
trace	Printing of progress occurs? Default to <code>TRUE</code> . If numeric, progress will be printed every <code>trace</code> iterations.
initialpop	an initial population used as a starting population in the optimization procedure. May be useful to speed up the convergence. Default to <code>NULL</code> . If given, each member of the initial population should be given as a row of a numeric matrix, so that <code>initialpop</code> is a matrix with <code>NP</code> rows and a number of columns equal to the length of the parameter vector to be optimized.
storepopfrom	from which generation should the following intermediate populations be stored in memory. Default to <code>itermax + 1</code> , i.e., no intermediate population is stored.

storepopfreq	the frequency with which populations are stored. Default to 1, i.e., every intermediate population is stored.
p	when strategy = 6, the top (100 * p)% best solutions are used in the mutation. p must be defined in (0,1].
c	when c > 0, crossover probability(CR) and step-size(F) are randomized at each mutation as an implementation of the JADE algorithm . c must be defined in [0,1].
reltol	relative convergence tolerance. The algorithm stops if it is unable to reduce the value by a factor of reltol * (abs(val) + reltol) after steptol steps. Defaults to sqrt(.Machine\$double.eps), typically about 1e-8.
steptol	see reltol. Defaults to itermax.

## Details

This defines the Differential Evolution strategy used in the optimization procedure, described below in the terms used by Price et al. (2006); see also Mullen et al. (2009) for details.

- strategy = 1: DE / rand / 1 / bin.  
This strategy is the classical approach for DE, and is described in [DEoptim](#).
- strategy = 2: DE / local-to-best / 1 / bin.  
In place of the classical DE mutation the expression

$$v_{i,g} = old_{i,g} + (best_g - old_{i,g}) + x_{r0,g} + F \cdot (x_{r1,g} - x_{r2,g})$$

is used, where  $old_{i,g}$  and  $best_g$  are the  $i$ -th member and best member, respectively, of the previous population. This strategy is currently used by default.

- strategy = 3: DE / best / 1 / bin with jitter.  
In place of the classical DE mutation the expression

$$v_{i,g} = best_g + jitter + F \cdot (x_{r1,g} - x_{r2,g})$$

is used, where  $jitter$  is defined as  $0.0001 * rand + F$ .

- strategy = 4: DE / rand / 1 / bin with per vector dither.  
In place of the classical DE mutation the expression

$$v_{i,g} = x_{r0,g} + dither \cdot (x_{r1,g} - x_{r2,g})$$

is used, where  $dither$  is calculated as  $F + rand * (1 - F)$ .

- strategy = 5: DE / rand / 1 / bin with per generation dither.  
The strategy described for 4 is used, but  $dither$  is only determined once per-generation.
- any value not above: variation to DE / rand / 1 / bin: either-or algorithm.  
In the case that  $rand < 0.5$ , the classical strategy strategy = 1 is used. Otherwise, the expression

$$v_{i,g} = x_{r0,g} + 0.5 \cdot (F + 1) \cdot (x_{r1,g} + x_{r2,g} - 2 \cdot x_{r0,g})$$

is used.

**Value**

The default value of `control` is the return value of `DEoptim.control()`, which is a list (and a member of the S3 class `DEoptim.control`) with the above elements.

**Note**

Further details and examples of the R package **DEoptim** can be found in Mullen et al. (2009) and Ardia et al. (2010).

Please cite the package in publications.

**Author(s)**

For **RcppDE**: Dirk Eddelbuettel.

For **DEoptim**: David Ardia, Katharine Mullen <katharine.mullen@nist.gov>, Brian Peterson and Joshua Ulrich.

**References**

Price, K.V., Storn, R.M., Lampinen J.A. (2006) *Differential Evolution - A Practical Approach to Global Optimization*. Berlin Heidelberg: Springer-Verlag. ISBN 3540209506.

Mullen, K.M., Ardia, D., Gil, D.L, Windover, D., Cline, J. (2011) **DEoptim**: An R Package for Global Optimization by Differential Evolution. [doi:10.18637/jss.v040.i06](https://doi.org/10.18637/jss.v040.i06)

Ardia, D., Boudt, K., Carl, P., Mullen, K.M., Peterson, B.G. (2011) Differential Evolution (**DEoptim**) for Non-Convex Portfolio Optimization. [doi:10.32614/RJ2011005](https://doi.org/10.32614/RJ2011005)

**See Also**

[DEoptim](#) and [DEoptim-methods](#).

**Examples**

```
## set the population size to 20
DEoptim.control(NP = 20)

## set the population size, the number of iterations and don't
## display the iterations during optimization
DEoptim.control(NP = 20, itermax = 100, trace = FALSE)
```

# Index

- \* **methods**
  - DEoptim-methods, [7](#)
- \* **minimization**
  - DEoptim, [2](#)
- \* **nonlinear**
  - DEoptim, [2](#)
  - DEoptim.control, [9](#)
- \* **optimize**
  - DEoptim, [2](#)
  - DEoptim.control, [9](#)

constrOptim, [6](#)

DEoptim, [2](#), [7](#), [8](#), [11](#), [12](#)  
DEoptim-methods, [7](#)  
DEoptim.control, [2](#), [5](#), [6](#), [8](#), [9](#)

optim, [6](#)

plot.DEoptim (DEoptim-methods), [7](#)

summary.DEoptim (DEoptim-methods), [7](#)