Package 'OSCARS'

November 21, 2026

Type Package

Title Global Bounded Optimization by the OSCARS-II Algorithm

Version 0.2.2

Maintainer Chris Price <chrisj.price@canterbury.ac.nz>

Description A collection of general optimization routines based on variants of the One Side Cut Accelerated Random Search (OSCARS-II) algorithm (Price et al., 2020, <doi:10.1007/s10898-020-00928-6>). The main function , 'oscars()', performs black-box optimization of a general (including nonsmooth or discontinuous) function subject to simple bounds on the unknowns. If all bounds are finite, oscars searches globally. The main method implements a stochastic direct search method and is derivative free. Testing shows the OSCARS-II algorithm usually finds extrema with fewer function evaluations than similar global derivative-free methods.

License Apache License (>= 2)

Encoding UTF-8

Imports stats, cli

Config/roxygen2/version 8.1.1

Suggests testthat (>= 3.0.0)

Config/testthat/edition 3

NeedsCompilation no

Author Chris Price [aut, cre],
Trent McDonald [aut, ctb] (R packaging)

Repository CRAN

Date/Publication 2026-11-21 19:30:23 UTC

Contents

oscars 2
oscars.control 5
oscarsQN 6
oscarsQN.control 10

poscars 12
print.oscars 15
print.oscarsQN 16
summary.oscars 17
summary.oscarsQN 18

Index **20**

`oscars` *OSCARS-II bound constrained global optimization*

Description

Performs black-box optimization of a general function subject to bounds on the unknown parameters using a variant of the OSCARS-II algorithm (Price, Reale and Robertson (2020) <doi.org/10.1007/s10898-020-00928-6>). If all bounds are finite, Oscars acts as a global optimization algorithm. It has been adapted to handle infinite upper and lower bounds, in which case the method has the characteristics of a local method for nonsmooth problems. Oscars does not use or assume the existence of derivatives of the objective function. It is a low overhead method for cheaply evaluated black-box functions. Black-box optimization methods for arbitrary functions do not and cannot provide certificates of optimality if halted after a finite amount of time.

Oscars is a stochastic direct search method which uses only function values at selected points. It generates a finite sequence of nested boxes around a control point, and randomly samples each box once, in turn. A new set of nested boxes is formed if the current set is exhausted or a point better than the control point is found. In the latter case the better point replaces the control. Initially the control point is set to the better of an internal initial point and a user supplied start point (if given).

From time to time the control is reset alternately to a random point, or to the best known point. Each reset marks the end of one cycle and the start of the next. All even numbered cycles start with control points chosen randomly from the feasible region. All odd numbered cycles (other than the first) set the control point equal to the best known point.

Oscars either performs a fixed number of function evaluations, or it halts if progress stalls for a significant period of time. In both cases it returns the best known point and the function evaluated at that point.

Usage

```
oscars(
  fname,
  n,
  lwr,
  upr,
  ...,
  start = NULL,
  controls = oscars.control(),
  progress = TRUE
)
```

Arguments

fname	An R function to be minimized. This function must take a vector of parameter values as its first argument, and return a scalar. Additional arguments can be supplied via ... Missing (NaN and NA) function values are acceptable as they are replaced with Inf when minimizing (or -Inf when maximizing).
n	The number of parameters with which fname is minimized.
lwr	A vector of lower bounds for the parameters of fname. If a single value lwr is supplied, this value will be used for all lower bounds. Lower bounds of minus infinity are acceptable. In order to maximize oscars effectiveness, it is recommended that the gap between upper and lower bounds not be unnecessarily wide.
upr	A vector of upper bounds for the parameters of fname. If a single value upr is supplied, this value will be used for all upper bounds. Upper bounds of infinity are acceptable. It is suggested that Inf be used for parameters that are unbounded above rather than a very large finite number as this signals the method to operate as a local search rather than attempting to cover all values between the bounds.
...	Additional parameters supplied to function fname.
start	This is an optional start point for the algorithm. It allows the user to direct the method to a region the user considers promising. If the start point is infeasible (i.e. violates some bounds) the closest feasible point to it is used. If a single value is provided, it is used for all dimensions. Default is null. The algorithm also generates an internal initial point as follows. When all bounds are finite, this is the centre point of the box. Otherwise each parameter is started at the average of its bounds when both are finite; if one bound is finite, it uses a feasible value near that bound; otherwise it uses the user supplied start value (if one is given) for that parameter, or zero otherwise.
controls	A list of oscar control parameters, such as iteration budget, tolerance, etc. See oscars.control for the full list and descriptions.
progress	If TRUE, a progress bar is drawn in the console. The bar appears after one to two seconds, so quick runs finish without a bar. Set to FALSE to suppress the bar entirely. Default is TRUE.

Value

A list containing results of the optimization. This list consists of the following components:

- par: vector containing the best known parameters
- value: The minimized (or maximized) function value
- evaluations: The number of function evaluations used
- cycles: The number of cycles used
- convergence: 0 if the function and parameter tolerances have been reached; 1 if tolerances have not been reached but function evaluation budget has been exhausted; 2 if bounds are inconsistent.
- message: A text string explaining the value in convergence.
- controls: The values of the controls provided to oscars

Examples

```
# Camel function with global minima of f = -1.0316 at
# (0.0898,0.7127) and (0.0898,-0.7127) with four other local minima
camel <- function(par) {
  x = par[1]
  y = par[2]
  f = 4*x^2 - 2.1*x^4 + (1/3)*x^6 + x*y + 4*(y^4-y^2)
  return(f) }
out <- oscars(camel, n = 2, lwr = c(-5,-5), upr = c(5,5))

# How to use repeated upper and lower bounds.
# Bird function in 2 dimensions.  Global minimum = -106.7645367198
bird <- function(par)  {
  x1 = par[1];  x2 = par[2]
  f = sin(x1)*exp((1-cos(x2))^2) + cos(x2)*exp((1-sin(x1))^2) + (x1-x2)^2
  return(f)
} # end of bird function
out <- oscars(bird, 2, -10, 50)

# Hosaki function with global minimum of -2.3458 at (4,2) and one local minimum
hosaki <- function(par)  {
  x = par[1]
  y = par[2]
  f = (1 - 8*x + 7*x^2 - (7/3)*x^3 + (1/4)*x^4)*y*y*exp(-y)
  return(f) }
out <- oscars(hosaki, 2, 0, upr = c(5,6))

# The proper way to specify control parameters.
out <- oscars(hosaki, 2, lwr = c(0,0), upr = c(5,6),
  controls = oscars.control(nfmax = 100000, fTol=10*oscars.control()$fTol))

# An example of where the full function evaluation budget is used.
out <- oscars(hosaki,2,0,5,controls = oscars.control(nfmax=10000,fTol=-1))

# how to pass other values to the objective function
# Rosenbrocks "banana" function with global minimum of zero at (a, a^2)
rosenbrock <- function(par, a = 1, b = 100){
   f = (a - par[1])^2 + b*(par[2] - par[1]^2)^2
   return(f)
}
out <- oscars(rosenbrock, 2, -3, 3, a = 0.5)

# Providing a user start point to the algorithm.
# Weka_1 function with global minimum of 0 wherever x[1] = -1 and a local
# minimum at the origin.  Dimension n is arbitrary with bounds -1 <= x <= 2
weka_1 <- function(par)   {
  f1 = 1 + sqrt( sum( par^2 ))
  f2 = 4*par[1] + 4
  f = min(f1,f2)
  return(f)
}
out <- oscars(weka_1, 10, -1, 2, start = 0)
```

```
# An example of the use of infinite bounds.
# Active faces is a nonsmooth function with global minimum = 0 at the
# origin.   Standard bounds are 0 <= par <= 5.  Solution is on boundary.
activefaces <- function(par)  {
f1 = max( log( abs(par) + 1) )
f2 = log( abs( sum(par) ) + 1)
f = max(f1,f2)
return(f)
}
out <- oscars(activefaces, 10, 0, Inf, start = 4)
```

`oscars.control` *Control parameters for oscars routine*

Description

Provides control over oscar parameters, such as number of iterations, tolerance, etc.

Usage

```
oscars.control(
  nfmax = 50000,
  infol = 1,
  DoMax = FALSE,
  fTol = 1e-06,
  xTol = 1e-08
)
```

Arguments

`nfmax`	The maximum number of function evaluations to perform. Default for `nfmax` is 50000.
`infol`	Verbosity during iterations. If `infol` is greater than 1, each new best function value is printed. 0 prints nothing. Default is 1.
`DoMax`	logical variable set to TRUE if the objective is to be maximized. Default is FALSE.
`fTol`	Stopping tolerance for the objective function f. This tolerance is multiplied by the larger of the absolute value of the current objective function value and 1. This gives a relative tolerance for large f, and and absolute one otherwise. If `fTol` is negative the algorithm will do the maximum `nfmax` of allowed function evaluations and then stop. Default is 1e-6.
`xTol`	Tolerance in the decision variables which is used to define the minimum sampling box size along each axis. For each decision variable `xTol` is scaled by the larger of 1 and the magnitude of the current value of that variable. This yields a

relative tolerance of `xTol` for large magnitude decision variables, and an absolute tolerance for small ones. Once the sampling box is less than tolerance along all axes, the sequence of nested sample boxes is ended. `xTol` must be positive and the algorithm will impose a minimum value of 1e-12. Difference between current and previous best known points must be within relative or absolute tolerance of `xTol` for oscars to halt before the function budget is exhausted. Default is 1e-8.

Details

A subset of parameters can be specified. All non-specified parameters revert to their defaults. No parameter abbreviations.

Value

A named list of control parameters for oscars.

Examples

```
oscars.control()  # default values
oscars.control(nfmax = 100000) # bump iteration budget
oscars.control(xTol = 10*oscars.control()$xTol) # increase xTol
```

`oscarsQN` *OSCARS-II-quasi-Newton for bound constrained global optimization*

Description

Performs global optimization of a general function subject to bounds on the unknown parameters using a variant of the algorithm in (Price, Reale and Robertson (2025) <doi.org/10.1007/s43069-024-00403-y>). The method is designed for functions which are continuously differentiable, but will run on black-box functions where only function values are available. It incorporates aspects of the direct search method OSCARS-II, guaranteeing eventual convergence even on black-box continuous functions. From time to time quasi-Newton steps are performed to accelerate convergence and improve the accuracy of the estimated optimizer. The algorithm will use analytic gradients if provided, otherwise it will estimate them via finite differences.

If all bounds are finite, oscarsQN acts as a global optimization algorithm. It has been adapted to handle infinite upper and lower bounds, in which case the method has the characteristics of a local method. Black-box methods for global optimization of arbitrary functions do not and cannot provide certificates of optimality if halted after a finite amount of time, even if the gradient is available at sample points.

OscarsQN is a stochastic direct search method which uses function values and gradients at selected points. It generates a finite sequence of nested boxes around a control point, and randomly samples each box once, in turn. Once the current set is exhausted or a point better than the control point is found the algorithm performs one quasi-Newton step and constructs a new set of nested boxes. If a

better point than the control is found, it replaces the control. Initially the control point is set to the better of an internal initial point and a user supplied start point (if given).

From time to time the control is reset alternately to a random point, or to the best known point. Each reset marks the end of one cycle and the start of the next.

OscarsQN either performs a fixed number of function evaluations, or it halts if a user specified target value has been reached, or the same best locally optimal point has been seen a prescribed number of times. It returns the best known point and the function value at that point.

Usage

```
oscarsQN(
  fname,
  gname = NULL,
  n,
  lwr,
  upr,
  ...,
  start = NULL,
  progress = TRUE,
  controls = oscarsQN.control()
)
```

Arguments

fname	An R function to be minimized. This function must take a vector of parameter values as its first argument, and return a scalar. Additional arguments can be supplied via ... Missing (NaN and NA) function values are acceptable as they are replaced with Inf when minimizing (or -Inf when maximizing).
gname	An R function which returns the gradient vector of the function fname. Default is NULL which results in finite differences being used to estimate gradients. A setting is available in the controls which calculates both the finite difference and analytic gradients for comparison. When both are calculated, the finite difference gradients are used. Additional arguments can be supplied via ...
n	The number of parameters with which fname is minimized.
lwr	A vector of lower bounds for the parameters of fname. If a single value lwr is supplied, this value will be used for all lower bounds. Lower bounds of minus infinity are acceptable. In order to maximize oscarsQN's effectiveness, it is recommended that the gap between upper and lower bounds not be unnecessarily wide. It is suggested that -Inf be used for parameters that are unbounded below rather than a very large (negative) number as this signals the method to act as a local search rather than attempting to cover all values between the bounds.
upr	A vector of upper bounds for the parameters of fname. If a single value upr is supplied, this value will be used for all upper bounds. Upper bounds of infinity are acceptable. It is suggested that Inf be used for parameters that are unbounded above rather than a very large finite number as this signals the method to operate as a local search rather than attempting to cover all values between the bounds.
...	Additional parameters supplied to the functions fname and gname.

`start` This is an optional start point for the algorithm. It allows the user to direct the method to a region the user considers promising. If the start point is infeasible (i.e. violates some bounds) the closest feasible point to it is used. If a single value is provided, it is used for all dimensions. Default is null.

The algorithm also generates an internal initial point as follows. When all bounds are finite, this is the centre point of the box. Otherwise each parameter is started at the average of its bounds when both are finite; if one bound is finite, it uses a feasible value near that bound; otherwise it uses the user supplied start value (if one is given) for that parameter, or zero otherwise.

`progress` If TRUE, a progress bar is drawn in the console. The bar appears after one to two seconds, so quick runs finish without a bar. Set to FALSE to suppress the bar entirely. Default is TRUE.

`controls` A list of oscar control parameters, such as iteration budget, tolerance, etc. See `oscarsQN.control` for the full list and descriptions.

Value

A list containing results of the optimization. This list consists of the following components:

- `par`: vector containing the best known parameters
- `value`: The minimized (or maximized) function value
- `evaluations`: The number of function evaluations used. Function evaluations used in calculating finite difference estimates of the gradient are included in this total, but any analytic gradient calculations are not.
- `cycles`: The number of cycles used.
- `convergence`: 0 if the function and parameter tolerances have been reached; 1 if tolerances have not been reached but function evaluation budget has been exhausted; 2 if bounds are inconsistent.
- `message`: A text string explaining the value in `convergence`.
- `numberKKTpoints`: Number of Karush-Kuhn-Tucker (KKT) points found. KKT points are potential minimizers.
- `numberbestKKTpoints`: Number of KKT points found which take the best known function value (within tolerance).
- `controls`: The values of the controls provided to oscarsQN.

Examples

```
# Camel function with global minima of f = -1.0316 at
# (0.0898,0.7127) and (0.0898,-0.7127) with four other local minima
camel <- function(par) {
  x = par[1]
  y = par[2]
  f = 4*x^2 - 2.1*x^4 + (1/3)*x^6 + x*y + 4*(y^4-y^2)
  return(f) }

camelgrad <- function(par) {
  x = par[1]
```

```
  y = par[2]
  g = c(0, 0)
  g[1] = 8*x - 8.4*x^3 + 2*x^5 + y
  g[2] = x + 16*y^3 - 8*y
  return(g) }
out <- oscarsQN(camel, camelgrad, n = 2, lwr = c(-5,-5), upr = c(5,5))


# Bird function in 2 dimensions.  Global minimum = -106.7645367198
bird <- function(par)  {
  x1 = par[1];  x2 = par[2]
  f = sin(x1)*exp((1-cos(x2))^2) + cos(x2)*exp((1-sin(x1))^2) + (x1-x2)^2
  return(f) }

birdgrad <- function(par)  {
  x1 = par[1];  x2 = par[2]
  g = c(0, 0)
 g[1] = cos(x1)*exp((1-cos(x2))^2) - 2*cos(x2)*exp((1-sin(x1))^2)*(1-sin(x1))*cos(x1) + 2*(x1-x2)
 g[2] = 2*sin(x1)*exp((1-cos(x2))^2)*(1-cos(x2))*sin(x2) - sin(x2)*exp((1-sin(x1))^2) + 2*(x2-x1)
  return(g) }
out <- oscarsQN(bird, birdgrad, 2, -10, 50)

# Hosaki function with global minimum of -2.3458 at (4,2) and one local minimum
hosaki <- function(par)  {
  x = par[1]
  y = par[2]
  f = (1 - 8*x + 7*x^2 - (7/3)*x^3 + (1/4)*x^4)*y*y*exp(-y)
  return(f) }

hosakigrad <- function(par)  {
  x = par[1]
  y = par[2]
  g = c(0, 0)
  g[1] = (-8 + 14*x - 7*x^2 + x^3)*y*y*exp(-y)
  g[2] = (1 - 8*x + 7*x^2 - (7/3)*x^3 + (1/4)*x^4)*(2-y)*y*exp(-y)
  return(g) }
out <- oscarsQN(hosaki, hosakigrad, 2, 0, upr = c(5,6))

# Rosenbrocks "banana" function with global minimum of zero at (a, a^2)
rosenbrock <- function(par, a = 1, b = 100) {
  f = (a - par[1])^2 + b*(par[2] - par[1]^2)^2
  return(f)  }

rosenbrockgrad <- function(par, a = 1, b = 100) {
  g = c(0, 0)
  g[1] = -2*(a - par[1]) + 2*b*(par[2] - par[1]^2)*(-2*par[1])
  g[2] = 2*b*(par[2] - par[1]^2)
  return(g)  }
out <- oscarsQN(rosenbrock, rosenbrockgrad, 2, -3, 3, a = 0.5)

# Schwefel function with global min of -418.9829n at x_i = 420.97...
# in n dimensions, where n is arbitrary.
schwefel <- function(par) {
```

```
  f = - sum(par*sin(sqrt(abs(par))))
  return(f)  }

schwefelgrad <- function(par) {
  rootpar = sqrt(abs(par))
  g = -sin(rootpar) - 0.5*rootpar*cos(rootpar)
  return(g)  }
out <- oscarsQN(schwefel, schwefelgrad, n = 3, -500, 500)
# This problem is solved in n = 3 dimensions here.

# vardim function with global min of 0 at par[i] = 1 in n dimensions.
vardim <- function(par) {
  n = length(par)
  temp = c(1:n)
  fn1 = sum(temp*(par-1))
  f = sum((par-1)^2) + fn1^2 + fn1^4
  return(f)   }

vardimgrad <- function(par) {
  n = length(par)
  temp = c(1:n)
  fn1 = sum(temp*(par-1))
  g = 2*(par-1) + (2*fn1 + 4*fn1^3)*temp
  return(g)  }
out <- oscarsQN(vardim, vardimgrad, n = 5, 0, 2.7182818)
# dixon function with global min of 0 in n dimensions at par[i] = 1.
dixon <- function(par) {
  n = length(par)
  xlo = par[1:n-1]
  xhi = par[2:n]
  f = (1-par[1])^2 + (1-par[n])^2 + sum( (xlo^2 - xhi)^2 )
  return(f)   }

dixongrad <- function(par) {
  n = length(par)
  x = par
  g = rep(0, times = n)
  g[1] = -2*(1-x[1]) + 2*(x[1]^2 - x[2])*2*x[1]
  jk = 2
  while (jk < n)  {
    #cat(sprintf("j = %i \n",jk))
    g[jk] = 2*(x[jk]^2 - x[jk+1])*2*x[jk] - 2*(x[jk-1]^2 - x[jk])
    jk = jk+1
  }
  g[n] = -2*(1-x[n]) + 2*(x[n-1]^2 - x[n])*(-1)
  return(g)  }
out <- oscarsQN(dixon, dixongrad, n = 4, -2, 2)
```

`oscarsQN.control` *Control parameters for oscarsQN routine*

Description

Provides control over oscarsQN parameters, such as number of iterations, tolerance, etc.

Usage

```
oscarsQN.control(
  nfmax = 50000,
  infol = 1,
  DoMax = FALSE,
  fTol = 1e-05,
  xTol = 1e-08,
  kktTol = 1e-05,
  kktstopcount = 3,
  fTarget = NULL,
  CompareGrad = FALSE
)
```

Arguments

`nfmax`	The maximum number of function evaluations to perform. Default for `nfmax` is 50000, but is normally adjusted by the user.
`infol`	Verbosity during iterations. If `infol` is greater than 1, each new best function value is printed. 0 prints nothing. Default is 1.
`DoMax`	logical variable set to TRUE if the objective is to be maximized. Default is `FALSE`.
`fTol`	Tolerance for comparing f values at KKT points. This tolerance is multiplied by the larger of the absolute value of the current objective function value and 1. This gives a relative tolerance for large f, and and absolute one otherwise.
`xTol`	Tolerance in the decision variables which is used to define the minimum sampling box size along each axis. For each decision variable `xTol` is scaled by the larger of 1 and the magnitude of the current value of that variable. This yields a relative tolerance of `xTol` for large magnitude decision variables, and an absolute tolerance for small ones. Once the sampling box is less than tolerance along all axes, the sequence of nested sample boxes is ended. `xTol` must be positive and the algorithm will impose a minimum value of 1e-12. Difference between current and previous best known points must be within relative or absolute tolerance of `xTol` for oscars to halt before the function budget is exhausted. Default is 1e-8.
`kktTol`	Tolerance for the Karush-Kuhn-Tucker conditions to determine a potential local minimizer of the problem. Default = 1e-5.
`kktstopcount`	Number of times a KKT point must be identified with objective function value f being within tolerance of the best known value. Maximum accepted variation in f value governed by `fTol`. Default = 3.
`fTarget`	Target value for the objective function which, once reached, halts the algorithm immediately. Default = NULL, which means no target is set and the method will not halt by this means.

`CompareGrad` If set to TRUE the algorithm with calculate both the analytic gradient and the finite difference gradient estimate. Both will be printed out for comparison and checking of the analytic gradient code listed under the function `gname`. Execution will proceed with the finite difference gradient.

Details

A subset of parameters can be specified. All non-specified parameters revert to their defaults. No parameter abbreviations.

Value

A named list of control parameters for oscars.

Examples

```
oscarsQN.control()  # default values
oscarsQN.control(nfmax = 100000) # bump iteration budget
oscarsQN.control(xTol = 10*oscars.control()$xTol) # increase xTol
```

`poscars` *Parallel OSCARS-II bound constrained global optimization*

Description

`poscars` is a parallel wrapper around `oscars` that runs several independent OSCARS-II searches at the same time, one per processor core, and returning the best result found.

Usage

```
poscars(
  fname,
  n,
  lwr,
  upr,
  ...,
  start = NULL,
  controls = oscars.control(),
  ncores = 2,
  divide.budget = TRUE,
  seed = NULL,
  cl = NULL
)
```

Arguments

fname	An R function to be minimized (or maximized). It must take a vector of parameter values as its first argument and return a scalar. Additional arguments can be supplied via .... Missing (NaN or NA) function values are acceptable; they are replaced with Inf when minimizing (or -Inf when maximizing). Because the runs execute in separate R processes, any helper objects or functions that fname references from the global environment should either be defined inside fname or passed to it through ... (see Details).
n	The number of parameters with which fname is minimized.
lwr	A vector of lower bounds for the parameters of fname. If a single value lwr is supplied, this value will be used for all lower bounds. Lower bounds of minus infinity are acceptable. In order to maximize oscars effectiveness, it is recommended that the gap between upper and lower bounds not be unnecessarily wide.
upr	A vector of upper bounds for the parameters of fname. If a single value upr is supplied, this value will be used for all upper bounds. Upper bounds of infinity are acceptable. It is suggested that Inf be used for parameters that are unbounded above rather than a very large finite number as this signals the method to operate as a local search rather than attempting to cover all values between the bounds.
...	Additional parameters supplied to function fname.
start	This is an optional start point for the algorithm. It allows the user to direct the method to a region the user considers promising. If the start point is infeasible (i.e. violates some bounds) the closest feasible point to it is used. If a single value is provided, it is used for all dimensions. Default is null. The algorithm also generates an internal initial point as follows. When all bounds are finite, this is the centre point of the box. Otherwise each parameter is started at the average of its bounds when both are finite; if one bound is finite, it uses a feasible value near that bound; otherwise it uses the user supplied start value (if one is given) for that parameter, or zero otherwise.
controls	A list of oscar control parameters, such as iteration budget, tolerance, etc. See oscars.control for the full list and descriptions.
ncores	The number of processor cores (parallel workers) to use. Must be a positive integer, or Inf to use every core detected on the machine. If ncores <= 1 the optimization is run serially by calling oscars directly (no cluster is created). If ncores exceeds the number of cores detected on the machine it is reduced to that number and a message is printed. Default is 2 per CRAN policies.
divide.budget	Logical. If TRUE (the default) the total evaluation budget controls$nfmax is divided among the ncores workers. In this case, each worker performs a max of ceiling(nfmax / ncores) evaluations. If FALSE, every worker is given the full nfmax budget.
seed	An optional integer used to seed the parallel random number streams. Setting this makes the entire parallel run reproducible. Regardless, each worker receives its own seed to ensure independent OSCARS streams across workers. Default is NULL, which draws non-reproducible streams.
cl	An optional pre-existing parallel cluster object created by makeCluster. If supplied it is used for the runs and left running on exit (the caller is responsible for

stopping it), which avoids cluster start-up cost across repeated calls. If NULL (the default) a temporary cluster of ncores workers is created and stopped automatically.

Details

OSCARS-II is a stochastic direct search: each run draws random sample points inside a sequence of nested, shrinking boxes and periodically restarts from a random or the best known point (see oscars for full details). Each run depends on its own chain of random draws and the inner search loop cannot be split across cores. However, *independent whole runs* are possible and poscars launches ncores independent runs, each seeded with its own reproducible random number stream, and keeps whichever run finds the best objective value.

Each worker run stops using the ordinary oscars stopping rules. OSCARS stopping rules are governed by tolerances fTol and xTol (see oscars.control).

By default, poscars divides the total evaluation budget nfmax evenly among the ncores workers (see divide.budget). This keeps the total number of function evaluations roughly the same as a single serial oscars call, but should execute faster by roughly ncores times. Set divide.budget = FALSE to give every worker the full budget, which does more total work but improves the chance of locating the global optimum.

The objective function fname, its ... arguments, the bounds and the controls are sent to each worker. Any objects that fname uses from the global workspace are *not* automatically exported to the workers. To be safe, make fname self-contained or pass everything it needs through ....

Value

A list of class "oscars" containing results of the optimization. This list consists of the following components:

- par: vector containing the best known parameters
- value: The minimized (or maximized) function value
- evaluations: The number of function evaluations used
- cycles: The number of cycles used
- convergence: 0 if the function and parameter tolerances have been reached; 1 if tolerances have not been reached but function evaluation budget has been exhausted; 2 if bounds are inconsistent.
- message: A text string explaining the value in convergence.
- controls: The values of the controls provided to oscars.
- ncores: The number of cores used.
- all.values: A vector containing the best values returned by every worker.

See Also

oscars for the underlying algorithm and the meaning of the return fields; oscars.control for the control parameters; makeCluster for supplying your own cluster.

Examples

```
# Per CRAN policies, these examples run on only 2 cores.
# Set ncores = Inf to utilize all cores.
# Setting ncores = parallel::detectCores()-1 is a good choice.

# Camel function with global minima of f = -1.0316 at
# (0.0898, 0.7127) and (0.0898, -0.7127) plus four other local minima.
camel <- function(par) {
  x <- par[1]
  y <- par[2]
  4*x^2 - 2.1*x^4 + (1/3)*x^6 + x*y + 4*(y^4 - y^2)
}
# Run four independent searches in parallel and keep the best.
out <- poscars(camel, n = 2, lwr = c(-5, -5), upr = c(5, 5), ncores = 2)
out

# Reproducible parallel run via the seed argument.
out1 <- poscars(camel, 2, -5, 5, ncores = 2, seed = 42)
out2 <- poscars(camel, 2, -5, 5, ncores = 2, seed = 42)
identical(out1$value, out2$value)

# Passing extra arguments to the objective function.
# Rosenbrock's "banana" function, global minimum 0 at (a, a^2).
rosenbrock <- function(par, a = 1, b = 100) {
  (a - par[1])^2 + b*(par[2] - par[1]^2)^2
}
out <- poscars(rosenbrock, 2, -3, 3, a = 0.5, ncores = 2)

# Best-of-ncores multi-start: give every worker the full budget instead
# of dividing it, trading more total work for a more thorough search.
out <- poscars(camel, 2, -5, 5, ncores = 2, divide.budget = FALSE,
               controls = oscars.control(nfmax = 20000, infol = 0))

# Reuse a single cluster across several calls to avoid start-up cost.
cl <- parallel::makeCluster(2)
o1 <- poscars(camel, 2, -5, 5, cl = cl)
o2 <- poscars(rosenbrock, 2, -3, 3, a = 0.5, cl = cl)
parallel::stopCluster(cl)
```

`print.oscars` *Print method for 'oscars' objects*

Description

Prints an 'oscars' object showing minimized (or maximized) parameters and the optimization message.

Usage

```
## S3 method for class 'oscars'
print(x, ...)
```

Arguments

x	An 'oscars' object returned by oscars.
...	Included for compatibility with other print methods. Ignored here.

Value

No return value, called for side effects. Technically, NULL is returned invisibly.

See Also

oscars

Examples

```
# Branins camel function with global minimum of f = -1.0316 at
# (0.0898,0.7127) and (0.0898,-0.7127) with four other local minimizers
camel <- function(par) {
  x = par[1]
  y = par[2]
  f = 4*x^2 - 2.1*x^4 + (1/3)*x^6 + x*y + 4*(y^4-y^2)
  return(f) }
out <- oscars(camel, n = 2, lwr = c(-5,-5), upr = c(5,5))
out
```

---

print.oscarsQN *Print method for 'oscars' objects*

Description

Prints an 'oscars' object showing minimized (or maximized) parameters and the optimization message.

Usage

```
## S3 method for class 'oscarsQN'
print(x, ...)
```

Arguments

x	An 'oscars' object returned by oscars.
...	Included for compatibility with other print methods. Ignored here.

Value

No return value, called for side effects. Technically, NULL is returned invisibly.

See Also

oscars

Examples

```
# Hosaki function with global minimum of -2.3458 at (4,2) and one local minimum
hosaki <- function(par)  {
  x = par[1]
  y = par[2]
  f = (1 - 8*x + 7*x^2 - (7/3)*x^3 + (1/4)*x^4)*y*y*exp(-y)
  return(f) }

hosakigrad <- function(par)  {
  x = par[1]
  y = par[2]
  g = c(0, 0)
  g[1] = (-8 + 14*x - 7*x^2 + x^3)*y*y*exp(-y)
  g[2] = (1 - 8*x + 7*x^2 - (7/3)*x^3 + (1/4)*x^4)*(2-y)*y*exp(-y)
  return(g) }
out <- oscarsQN(hosaki, hosakigrad, 2, 0, upr = c(5,6))
out
```

---

summary.oscars *Summary method for 'oscars' objects*

Description

Summarizes an 'oscars' object. Shows an 'oscars' object's minimized (or maximized) parameters, optimization message, iterations, etc..

Usage

```
## S3 method for class 'oscars'
summary(object, ...)
```

Arguments

object	An 'oscars' object returned by oscars.
...	Ignored here. Included for use by other methods.

Value

No return value, called for side effects. Technically, NULL is returned invisibly.

See Also

oscars

Examples

```
# Branins camel function with global minimum of f = -1.0316 at
# (0.0898,0.7127) and (0.0898,-0.7127) with four other local minimizers
camel <- function(par) {
  x = par[1]
  y = par[2]
  f = 4*x^2 - 2.1*x^4 + (1/3)*x^6 + x*y + 4*(y^4-y^2)
  return(f) }
out <- oscars(camel, n = 2, lwr = c(-5,-5), upr = c(5,5))
summary(out)
```

---

summary.oscarsQN *Summary method for 'oscarsQN' objects*

Description

Summarizes an 'oscarsQN' object. Shows an 'oscarsQN' object's minimized (or maximized) parameters, optimization message, iterations, etc..

Usage

```
## S3 method for class 'oscarsQN'
summary(object, ...)
```

Arguments

object	An 'oscarsQN' object returned by oscarsQN.
...	Ignored here. Included for use by other methods.

Value

No return value, called for side effects. Technically, NULL is returned invisibly.

See Also

oscarsQN

Examples

```
# Hosaki function with global minimum of -2.3458 at (4,2) and one local minimum
hosaki <- function(par)  {
  x = par[1]
  y = par[2]
  f = (1 - 8*x + 7*x^2 - (7/3)*x^3 + (1/4)*x^4)*y*y*exp(-y)
  return(f) }

hosakigrad <- function(par)  {
  x = par[1]
  y = par[2]
  g = c(0, 0)
  g[1] = (-8 + 14*x - 7*x^2 + x^3)*y*y*exp(-y)
  g[2] = (1 - 8*x + 7*x^2 - (7/3)*x^3 + (1/4)*x^4)*(2-y)*y*exp(-y)
  return(g) }
out <- oscarsQN(hosaki, hosakigrad, 2, 0, upr = c(5,6))
summary(out)
```

Index

makeCluster, *13, 14*

oscars, 2, *12–14*, *16–18*
oscars.control, *3*, 5, *13, 14*
oscarsQN, 6, *18*
oscarsQN.control, *8*, 10

poscars, 12
print.oscars, 15
print.oscarsQN, 16

summary.oscars, 17
summary.oscarsQN, 18