Package 'GA'

May 7, 2026

Version 3.2.5

Date 2026-01-08

Title Genetic Algorithms

Description Flexible general-purpose toolbox implementing genetic algorithms (GAs) for stochastic optimisation. Binary, real-valued, and permutation representations are available to optimize a fitness function, i.e. a function provided by users depending on their objective function. Several genetic operators are available and can be combined to explore the best settings for the current task. Furthermore, users can define new genetic operators and easily evaluate their performances. Local search using general-purpose optimisation algorithms can be applied stochastically to exploit interesting regions. GAs can be run sequentially or in parallel, using an explicit master-slave parallelisation or a coarse-grain islands approach. For more details see Scrucca (2013) <doi:10.18637/jss.v053.i04> and Scrucca (2017) <doi:10.32614/RJ-2017-008>.

Depends R (>= 3.4), methods, foreach, iterators

Imports stats, graphics, grDevices, utils, cli, crayon, Rcpp

LinkingTo Rcpp, RcppArmadillo

Suggests parallel, doParallel, doRNG (>= 1.6), knitr (>= 1.8), rmarkdown (>= 2.0)

License GPL (>= 2)

VignetteBuilder knitr

URL https://luca-scr.github.io/GA/

BugReports https://github.com/luca-scr/GA/issues

Repository CRAN

ByteCompile true

NeedsCompilation yes

Author Luca Scrucca [aut, cre] (ORCID: <https://orcid.org/0000-0003-3826-0484>)

Maintainer Luca Scrucca <luca.scrucca@unibo.it>

Date/Publication 2026-01-08 13:10:08 UTC

Contents

`GA-package` *Genetic Algorithms*

Description

Flexible general-purpose toolbox implementing genetic algorithms (GAs) for stochastic optimisation. Binary, real-valued, and permutation representations are available to optimize a fitness function, i.e. a function provided by users depending on their objective function. Several genetic operators are available and can be combined to explore the best settings for the current task. Furthermore, users can define new genetic operators and easily evaluate their performances. Local search using general-purpose optimisation algorithms can be applied stochastically to exploit interesting regions. GAs can be run sequentially or in parallel, using an explicit master-slave parallelisation or a coarse-grain islands approach.

Details

For a quick intro to GA package see the vignette accompanying the package. Further details are provided in the papers referenced below.

References

Scrucca L. (2013). GA: A Package for Genetic Algorithms in R. *Journal of Statistical Software*, 53(4), 1-37, doi:10.18637/jss.v053.i04.

Scrucca, L. (2017) On some extensions to GA package: hybrid optimisation, parallelisation and islands evolution. *The R Journal*, 9/1, 187-206. doi:10.32614/RJ2017008

Author(s)

Luca Scrucca <luca.scrucca@unibo.it>

`binary2decimal` *Binary encoding of decimal numbers and vice versa.*

Description

Functions for computing binary to decimal conversion of numbers and vice versa.

Usage

```
decimal2binary(x, length)
binary2decimal(x)
```

Arguments

`x`	input value.
`length`	an optional value giving the length of binary string to return.

Details

`decimal2binary` converts a numerical value (which is forced to be an integer) to a binary representation, i.e. a vector of 0s and 1s. For real numerical values see the example below.

`binary2decimal` converts a binary value, i.e. a vector of 0s and 1s, to a decimal representation.

Author(s)

Luca Scrucca

See Also

`binary2gray`

Examples

```
# for integer values
dval <- 12
(bval <- decimal2binary(dval))
binary2decimal(bval)

# for real values
dval <- 12.456
# use
(bval <- decimal2binary(dval*1000))
binary2decimal(bval)/1000
```

`binary2gray` *Gray encoding for binary strings*

Description

Functions for computing Gray encoding from/to binary strings.

Usage

```
binary2gray(x)
gray2binary(x)
```

Arguments

`x`	the string to be evaluated

Details

Gray encoding allows to obtain binary strings not affected by the well-known Hamming cliff problem. With Gray encoding the number of bit differences between any two consecutive values is one, whereas in binary strings this is not always true.

Author(s)

Luca Scrucca

See Also

`binary2decimal`

Examples

```
# Consider a five-bit encoding of values 15 and 16  using the standard
# binary coding
decimal2binary(15, 5)
decimal2binary(16, 5)
# Moving from 15 to 16 (or vice versa) all five bits need to be changed,
# but using Gray encoding the two binary strings differ by one bit.
binary2gray(decimal2binary(15, 5))
binary2gray(decimal2binary(16, 5))
```

---

de *Differential Evolution via Genetic Algorithms*

Description

Maximization of a fitness function using Differential Evolution (DE). DE is a population-based evolutionary algorithm for optimisation of fitness functions defined over a continuous parameter space.

Usage

```
de(fitness,
   lower, upper,
   popSize = 10*d,
   stepsize = 0.8,
   pcrossover = 0.5,
   ...)
```

Arguments

`fitness`	the fitness function, any allowable R function which takes as input a vector of values representing a potential solution, and returns a numerical value describing its "fitness".
`lower`	a vector of length equal to the decision variables providing the lower bounds of the search space.
`upper`	a vector of length equal to the decision variables providing the upper bounds of the search space.
`popSize`	the population size. By default is set at 10 times the number of decision variables.
`pcrossover`	the probability of crossover, by default set to 0.5.
`stepsize`	the stepsize or weighting factor. A value in the interval [0,2], by default set to 0.8. If set at `NA` a random value is selected in the interval [0.5, 1.0] (so called dithering).
`...`	additional arguments to be passed to the `ga` function.

Details

Differential Evolution (DE) is a stochastic evolutionary algorithm that optimises multidimensional real-valued fitness functions without requiring the optimisation problem to be differentiable.

This implementation follows the description in Simon (2013; Sec. 12.4, and Fig. 12.12) and uses the functionalities available in the `ga` function for Genetic Algorithms.

The DE selection operator is defined by `gareal_de` with parameters `p = pcrossover` and `F = stepsize`.

Value

Returns an object of class `de-class`. See `de-class` for a description of available slots information.

Author(s)

Luca Scrucca <luca.scrucca@unibo.it>

References

Scrucca L. (2013). GA: A Package for Genetic Algorithms in R. *Journal of Statistical Software*, 53(4), 1-37, doi:10.18637/jss.v053.i04.

Scrucca, L. (2017) On some extensions to GA package: hybrid optimisation, parallelisation and islands evolution. *The R Journal*, 9/1, 187-206, doi:10.32614/RJ2017008.

Simon D. (2013) *Evolutionary Optimization Algorithms*. John Wiley & Sons.

Price K., Storn R.M., Lampinen J.A. (2005) *Differential Evolution: A Practical Approach to Global Optimization*. Springer.

See Also

`summary,de-method`, `plot,de-method`, `de-class`

Examples

```
# 1) one-dimensional function
f <- function(x)  abs(x)+cos(x)
curve(f, -20, 20)

DE <- de(fitness = function(x) -f(x), lower = -20, upper = 20)
plot(DE)
summary(DE)

curve(f, -20, 20, n = 1000)
abline(v = DE@solution, lty = 3)

# 2) "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)

# from help("optim")
SANN <- optim(50, fn = wild, method = "SANN",
```

```
              control = list(maxit = 20000, temp = 20, parscale = 20))
unlist(SANN[1:2])

DE <- de(fitness = function(...) -wild(...), lower = -50, upper = 50)
plot(DE)
summary(DE)

# 3) two-dimensional Rastrigin function

Rastrigin <- function(x1, x2)
{
  20 + x1^2 + x2^2 - 10*(cos(2*pi*x1) + cos(2*pi*x2))
}

x1 <- x2 <- seq(-5.12, 5.12, by = 0.1)
f <- outer(x1, x2, Rastrigin)
persp3D(x1, x2, f, theta = 50, phi = 20, col.palette = bl2gr.colors)

DE <- de(fitness = function(x) -Rastrigin(x[1], x[2]),
         lower = c(-5.12, -5.12), upper = c(5.12, 5.12),
         popSize = 50)
plot(DE)
summary(DE)

filled.contour(x1, x2, f, color.palette = bl2gr.colors,
               plot.axes = { axis(1); axis(2);
                             points(DE@solution,
                                    col = "yellow", pch = 3, lwd = 2) })

# 4) two-dimensional Ackley function

Ackley <- function(x1, x2)
{
  -20*exp(-0.2*sqrt(0.5*(x1^2 + x2^2))) -
  exp(0.5*(cos(2*pi*x1) + cos(2*pi*x2))) + exp(1) + 20
}

x1 <- x2 <- seq(-3, 3, by = 0.1)
f <- outer(x1, x2, Ackley)
persp3D(x1, x2, f, theta = 50, phi = 20, col.palette = bl2gr.colors)

DE <- de(fitness = function(x) -Ackley(x[1], x[2]),
         lower = c(-3, -3), upper = c(3, 3),
         stepsize = NA)
plot(DE)
summary(DE)

filled.contour(x1, x2, f, color.palette = bl2gr.colors,
               plot.axes = { axis(1); axis(2);
                             points(DE@solution,
                                    col = "yellow", pch = 3, lwd = 2) })

# 5) Curve fitting example (see Scrucca JSS 2013)
```

```
## Not run:
# subset of data from data(trees, package = "spuRs")
tree <- data.frame(Age = c(2.44, 12.44, 22.44, 32.44, 42.44, 52.44, 62.44,
                           72.44, 82.44, 92.44, 102.44, 112.44),
                   Vol = c(2.2, 20, 93, 262, 476, 705, 967, 1203, 1409,
                           1659, 1898, 2106))
richards <- function(x, theta)
  { theta[1]*(1 - exp(-theta[2]*x))^theta[3] }
fitnessL2 <- function(theta, x, y)
  { -sum((y - richards(x, theta))^2) }
DE <- de(fitness = fitnessL2, x = tree$Age, y = tree$Vol,
         lower = c(3000, 0, 2), upper = c(4000, 1, 4),
         popSize = 500, maxiter = 1000, run = 100,
         names = c("a", "b", "c"))
summary(DE)

## End(Not run)
```

`de-class` *Class "de"*

Description

An S4 class for differential evolution algorithm

Objects from the Class

Objects can be created by calls to the de function.

Slots

`call` an object of class `"call"` representing the matched call;

`type` a character string specifying the type of genetic algorithm used;

`lower` a vector providing for each decision variable the lower bounds of the search space in case of real-valued or permutation encoded optimisations. Formerly this slot was named `min`;

`upper` a vector providing for each decision variable the upper bounds of the search space in case of real-valued or permutation encoded optimizations. Formerly this slot was named `max`;

`names` a vector of character strings providing the names of decision variables (optional);

`popSize` the population size;

`iter` the actual (or final) iteration of DE search;

`run` the number of consecutive generations without any improvement in the best fitness value before the DE is stopped;

`maxiter` the maximum number of iterations to run before the DE search is halted;

`suggestions` a matrix of user provided solutions and included in the initial population;

`population` the current (or final) population;

`elitism` the number of best fitness individuals to survive at each generation;

`stepsize` the stepsize or weighting factor;

`pcrossover` the crossover probability;

`pmutation` the mutation probability;

`optim` a logical specifying whether or not a local search using general-purpose optimisation algorithms should be used;

`fitness` the values of fitness function for the current (or final) population;

`summary` a matrix of summary statistics for fitness values at each iteration (along the rows);

`bestSol` if `keepBest = TRUE`, the best solutions at each iteration;

`fitnessValue` the best fitness value at the final iteration;

`solution` the value(s) of the decision variables giving the best fitness at the final iteration.

Author(s)

Luca Scrucca

See Also

For examples of usage see `de`.

`ga` *Genetic Algorithms*

Description

Maximization of a fitness function using genetic algorithms (GAs). Local search using general-purpose optimisation algorithms can be applied stochastically to exploit interesting regions. The algorithm can be run sequentially or in parallel using an explicit master-slave parallelisation.

Usage

```
ga(type = c("binary", "real-valued", "permutation"),
   fitness, ...,
   lower, upper, nBits,
   population = gaControl(type)$population,
   selection = gaControl(type)$selection,
   crossover = gaControl(type)$crossover,
   mutation = gaControl(type)$mutation,
   popSize = 50,
   pcrossover = 0.8,
   pmutation = 0.1,
   elitism = base::max(1, round(popSize*0.05)),
   updatePop = FALSE,
```

```
  postFitness = NULL,
  maxiter = 100,
  run = maxiter,
  maxFitness = Inf,
  names = NULL,
  suggestions = NULL,
  optim = FALSE,
  optimArgs = list(method = "L-BFGS-B",
                   poptim = 0.05,
                   pressel = 0.5,
                   control = list(fnscale = -1, maxit = 100)),
  keepBest = FALSE,
  parallel = FALSE,
  monitor = if(interactive()) gaMonitor else FALSE,
  seed = NULL)
```

Arguments

type	the type of genetic algorithm to be run depending on the nature of decision variables. Possible values are: "binary" for binary representations of decision variables. "real-valued" for optimization problems where the decision variables are floating-point representations of real numbers. "permutation" for problems that involves reordering of a list of objects.
fitness	the fitness function, any allowable R function which takes as input an individual string representing a potential solution, and returns a numerical value describing its "fitness".
...	additional arguments to be passed to the fitness function. This allows to write fitness functions that keep some variables fixed during the search.
lower	a vector of length equal to the decision variables providing the lower bounds of the search space in case of real-valued or permutation encoded optimizations. Formerly this argument was named min; its usage is allowed but deprecated.
upper	a vector of length equal to the decision variables providing the upper bounds of the search space in case of real-valued or permutation encoded optimizations. Formerly this argument was named max; its usage is allowed but deprecated.
nBits	a value specifying the number of bits to be used in binary encoded optimizations.
population	an R function for randomly generating an initial population. See ga_Population for available functions.
selection	an R function performing selection, i.e. a function which generates a new population of individuals from the current population probabilistically according to individual fitness. See ga_Selection for available functions.
crossover	an R function performing crossover, i.e. a function which forms offsprings by combining part of the genetic information from their parents. See ga_Crossover for available functions.

`mutation` an R function performing mutation, i.e. a function which randomly alters the values of some genes in a parent chromosome. See `ga_Mutation` for available functions.

`popSize` the population size.

`updatePop` a logical defaulting to `FALSE`. If set at `TRUE` the first attribute attached to the value returned by the user-defined fitness function is used to update the population. *Be careful though, this is an experimental feature!*

`postFitness` a user-defined function which, if provided, receives the current `ga-class` object as input, performs post fitness-evaluation steps, then returns an updated version of the object which is used to update the GA search. *Be careful though, this is an experimental feature!*

`pcrossover` the probability of crossover between pairs of chromosomes. Typically this is a large value and by default is set to 0.8.

`pmutation` the probability of mutation in a parent chromosome. Usually mutation occurs with a small probability, and by default is set to 0.1.

`elitism` the number of best fitness individuals to survive at each generation. By default the top 5% individuals will survive at each iteration.

`maxiter` the maximum number of iterations to run before the GA search is halted.

`run` the number of consecutive generations without any improvement in the best fitness value before the GA is stopped.

`maxFitness` the upper bound on the fitness function after that the GA search is interrupted.

`names` a vector of character strings providing the names of decision variables.

`suggestions` a matrix of solutions strings to be included in the initial population. If provided the number of columns must match the number of decision variables.

`optim` a logical defaulting to `FALSE` determining whether or not a local search using general-purpose optimisation algorithms should be used. See argument `optimArgs` for further details and finer control.

`optimArgs` a list controlling the local search algorithm with the following components:

- `method` a string specifying the general-purpose optimisation method to be used, by default is set to `"L-BFGS-B"`. Other possible methods are those reported in `optim`.
- `poptim` a value in the range [0,1] specifying the probability of performing a local search at each iteration of GA (default 0.1).
- `pressel` a value in the range [0,1] specifying the pressure selection (default 0.5). The local search is started from a random solution selected with probability proportional to fitness. High values of `pressel` tend to select the solutions with the largest fitness, whereas low values of `pressel` assign quasi-uniform probabilities to any solution.
- `control` a list of control parameters. See 'Details' section in `optim`.

`keepBest` a logical argument specifying if best solutions at each iteration should be saved in a slot called `bestSol`. See `ga-class`.

`parallel` An optional argument which allows to specify if the Genetic Algorithm should be run sequentially or in parallel.

For a single machine with multiple cores, possible values are:

- a logical value specifying if parallel computing should be used (TRUE) or not (FALSE, default) for evaluating the fitness function;
- a numerical value which gives the number of cores to employ. By default, this is obtained from the function detectCores;
- a character string specifying the type of parallelisation to use. This depends on system OS: on Windows OS only "snow" type functionality is available, while on Unix/Linux/Mac OSX both "snow" and "multicore" (default) functionalities are available.

In all the cases described above, at the end of the search the cluster is automatically stopped by shutting down the workers.

If a cluster of multiple machines is available, evaluation of the fitness function can be executed in parallel using all, or a subset of, the cores available to the machines belonging to the cluster. However, this option requires more work from the user, who needs to set up and register a parallel back end. In this case the cluster must be explicitly stopped with stopCluster.

monitor — a logical or an R function which takes as input the current state of the ga-class object and show the evolution of the search. By default, for interactive sessions the function gaMonitor prints the average and best fitness values at each iteration. If set to plot these information are plotted on a graphical device. Other functions can be written by the user and supplied as argument. In non interactive sessions, by default monitor = FALSE so any output is suppressed.

seed — an integer value containing the random number generator state. This argument can be used to replicate the results of a GA search. Note that if parallel computing is required, the **doRNG** package must be installed.

Details

Genetic algorithms (GAs) are stochastic search algorithms inspired by the basic principles of biological evolution and natural selection. GAs simulate the evolution of living organisms, where the fittest individuals dominate over the weaker ones, by mimicking the biological mechanisms of evolution, such as selection, crossover and mutation.

The **GA** package is a collection of general purpose functions that provide a flexible set of tools for applying a wide range of genetic algorithm methods.

The ga function enables the application of GAs to problems where the decision variables are encoded as "binary", "real-valued", or "permutation" strings.

Default genetic operators are set via gaControl. To retrieve the currently set operators:

```
gaControl("binary")

gaControl("real-valued")

gaControl("permutation")
```

Value

Returns an object of class ga-class. See ga-class for a description of available slots information.

Author(s)

Luca Scrucca <luca.scrucca@unibo.it>

References

Back T., Fogel D., Michalewicz Z. (2000). *Evolutionary Computation 1: Basic Algorithms and Operators*. IOP Publishing Ltd., Bristol and Philadelphia.

Back T., Fogel D., Michalewicz Z. (2000b). *Evolutionary Computation 2: Advanced Algorithms and Operators*. IOP Publishing Ltd., Bristol and Philadelphia.

Coley D. (1999). *An Introduction to Genetic Algorithms for Scientists and Engineers*. World Scientific Pub. Co. Inc., Singapore.

Eiben A., Smith J. (2003). *Introduction to Evolutionary Computing*. Springer-Verlag, Berlin Heidelberg.

Goldberg D. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Professional, Boston, MA.

Haupt R. L., Haupt S. E. (2004). *Practical Genetic Algorithms*. 2nd edition. John Wiley & Sons, New York.

Luke S. (2013) *Essentials of Metaheuristics*, 2nd edition. Lulu.

Scrucca L. (2013). GA: A Package for Genetic Algorithms in R. *Journal of Statistical Software*, 53(4), 1-37, doi:10.18637/jss.v053.i04.

Scrucca, L. (2017) On some extensions to GA package: hybrid optimisation, parallelisation and islands evolution. *The R Journal*, 9/1, 187-206, doi:10.32614/RJ2017008.

Simon D. (2013) *Evolutionary Optimization Algorithms*. John Wiley & Sons.

Sivanandam S., Deepa S. (2007). *Introduction to Genetic Algorithms*. Springer-Verlag, Berlin Heidelberg.

Yu X., Gen M. (2010). *Introduction to Evolutionary Algorithms*. Springer-Verlag, Berlin Heidelberg.

See Also

summary,ga-method, plot,ga-method, ga-class, ga_Population, ga_Selection, ga_Crossover, ga_Mutation, gaControl.

Examples

```
# 1) one-dimensional function
f <- function(x)  abs(x)+cos(x)
curve(f, -20, 20)

fitness <- function(x) -f(x)
GA <- ga(type = "real-valued", fitness = fitness, lower = -20, upper = 20)
summary(GA)
plot(GA)

curve(f, -20, 20)
abline(v = GA@solution, lty = 3)
```

```
# 2) one-dimensional function
f <- function(x)  (x^2+x)*cos(x) # -10 < x < 10
curve(f, -10, 10)

# write your own tracing function
monitor <- function(obj)
{
  curve(f, -10, 10, main = paste("iteration =", obj@iter))
  points(obj@population, obj@fitness, pch = 20, col = 2)
  rug(obj@population, col = 2)
  Sys.sleep(0.2)
}
## Not run:
GA <- ga(type = "real-valued", fitness = f, lower = -10, upper = 10, monitor = monitor)

## End(Not run)
# or if you want to suppress the tracing
GA <- ga(type = "real-valued", fitness = f, lower = -10, upper = 10, monitor = NULL)
summary(GA)

monitor(GA)
abline(v = GA@solution, lty = 3)

# 3) two-dimensional Rastrigin function

Rastrigin <- function(x1, x2)
{
  20 + x1^2 + x2^2 - 10*(cos(2*pi*x1) + cos(2*pi*x2))
}

x1 <- x2 <- seq(-5.12, 5.12, by = 0.1)
f <- outer(x1, x2, Rastrigin)
persp3D(x1, x2, f, theta = 50, phi = 20, col.palette = bl2gr.colors)
filled.contour(x1, x2, f, color.palette = bl2gr.colors)

GA <- ga(type = "real-valued", fitness =  function(x) -Rastrigin(x[1], x[2]),
         lower = c(-5.12, -5.12), upper = c(5.12, 5.12),
         popSize = 50, maxiter = 100)
summary(GA)
plot(GA)

# 4) Parallel GA
# Simple example of an expensive fitness function obtained artificially by
# introducing a pause statement.
## Not run:
Rastrigin <- function(x1, x2)
{
  Sys.sleep(0.1)
  20 + x1^2 + x2^2 - 10*(cos(2*pi*x1) + cos(2*pi*x2))
}

system.time(GA1 <- ga(type = "real-valued",
```

```
                        fitness =  function(x) -Rastrigin(x[1], x[2]),
                        lower = c(-5.12, -5.12), upper = c(5.12, 5.12),
                        popSize = 50, maxiter = 100, monitor = FALSE,
                        seed = 12345))

system.time(GA2 <- ga(type = "real-valued",
                        fitness =  function(x) -Rastrigin(x[1], x[2]),
                        lower = c(-5.12, -5.12), upper = c(5.12, 5.12),
                        popSize = 50, maxiter = 100, monitor = FALSE,
                        seed = 12345, parallel = TRUE))

## End(Not run)

# 5) Hybrid GA
# Example of GA with local search

Rastrigin <- function(x1, x2)
{
  20 + x1^2 + x2^2 - 10*(cos(2*pi*x1) + cos(2*pi*x2))
}

GA <- ga(type = "real-valued",
         fitness =  function(x) -Rastrigin(x[1], x[2]),
         lower = c(-5.12, -5.12), upper = c(5.12, 5.12),
         popSize = 50, maxiter = 100,
         optim = TRUE)
summary(GA)
```

`ga-class` *Class "ga"*

Description

An S4 class for genetic algorithms

Objects from the Class

Objects can be created by calls to the `ga` function.

Slots

`call` an object of class `"call"` representing the matched call;

`type` a character string specifying the type of genetic algorithm used;

`lower` a vector providing for each decision variable the lower bounds of the search space in case of real-valued or permutation encoded optimisations. Formerly this slot was named `min`;

`upper` a vector providing for each decision variable the upper bounds of the search space in case of real-valued or permutation encoded optimizations. Formerly this slot was named `max`;

`nBits` a value specifying the number of bits to be used in binary encoded optimizations;

`names` a vector of character strings providing the names of decision variables (optional);

`popSize` the population size;

`iter` the actual (or final) iteration of GA search;

`run` the number of consecutive generations without any improvement in the best fitness value before the GA is stopped;

`maxiter` the maximum number of iterations to run before the GA search is halted;

`suggestions` a matrix of user provided solutions and included in the initial population;

`population` the current (or final) population;

`elitism` the number of best fitness individuals to survive at each generation;

`pcrossover` the crossover probability;

`pmutation` the mutation probability;

`optim` a logical specifying whether or not a local search using general-purpose optimisation algorithms should be used;

`fitness` the values of fitness function for the current (or final) population;

`summary` a matrix of summary statistics for fitness values at each iteration (along the rows);

`bestSol` if `keepBest = TRUE`, the best solutions at each iteration;

`fitnessValue` the best fitness value at the final iteration;

`solution` the value(s) of the decision variables giving the best fitness at the final iteration.

Author(s)

Luca Scrucca

See Also

For examples of usage see `ga`.

`gaControl` *A function for setting or retrieving defaults genetic operators*

Description

Default settings for genetic operators used in the GA package.

Usage

```
gaControl(...)
```

Arguments

`...`	no arguments, a single character vector, or a named list with components.

Details

If the function is called with no arguments returns the current default settings, i.e., a list with the following default components:

`binary`
- `population` = `"gabin_Population"`
- `selection` = `"gabin_lrSelection"`
- `crossover` = `"gabin_spCrossover"`
- `mutation` = `"gabin_raMutation"`

`real-valued`
- `population` = `"gareal_Population"`
- `selection` = `"gareal_lsSelection"`
- `crossover` = `"gareal_laCrossover"`
- `mutation` = `"gareal_raMutation"`

`permutation`
- `population` = `"gaperm_Population"`
- `selection` = `"gaperm_lrSelection"`
- `crossover` = `"gaperm_oxCrossover"`
- `mutation` = `"gaperm_simMutation"`

`eps` the tolerance value used by the package functions. By default set at `sqrt(.Machine$double.eps)`.

`useRcpp` a logical specifying if a faster C++ implementation of genetic operators should be used (`TRUE` by default), or an R implementation.

The function may be called with a single string specifying the name of the component. In this case the function returns the current default settings.

To change the default values, a named component must be followed by a single value (in case of `eps` or `useRcpp`) or a list of component(s) specifying the name of the function for a genetic operator. See the Examples section.

Value

If the argument list is empty the function returns the current list of values. If the argument list is not empty, the returned list is invisible.

Note

The parameter values set via a call to this function will remain in effect for the rest of the session, affecting the subsequent behaviour of the functions for which the given parameters are relevant.

Author(s)

Luca Scrucca

See Also

`ga`

Examples

```
# get and save defaults
defaultControl <- gaControl()
print(defaultControl)
# get current defaults only for binary search
gaControl("binary")
# set defaults for selection operator of binary search
gaControl("binary" = list(selection = "gabin_tourSelection"))
gaControl("binary")
# set defaults for selection and crossover operators of binary search
gaControl("binary" = list(selection = "ga_rwSelection",
                          crossover = "gabin_uCrossover"))
gaControl("binary")
# restore defaults
gaControl(defaultControl)
gaControl()
```

`gaisl` *Islands Genetic Algorithms*

Description

Maximization of a fitness function using islands genetic algorithms (ISLGAs). This is a distributed multiple-population GA, where the population is partitioned into several subpopulations and assigned to separated islands. Independent GAs are executed in each island, and only occasionally sparse exchanges of individuals are performed among the islands. In principle islands can evolve sequentially, but increased computational efficiency is obtained by running GAs in parallel on each island. The latter is called island parallel GAs (ISLPGAs) and it is used by default.

Usage

```
gaisl(type = c("binary", "real-valued", "permutation"),
      fitness, ...,
      lower, upper, nBits,
      population = gaControl(type)$population,
      selection = gaControl(type)$selection,
      crossover = gaControl(type)$crossover,
      mutation = gaControl(type)$mutation,
      popSize = 100,
      numIslands = 4,
      migrationRate = 0.1,
      migrationInterval = 10,
      pcrossover = 0.8,
      pmutation = 0.1,
      elitism = base::max(1, round(popSize/numIslands*0.05)),
      updatePop = FALSE,
      postFitness = NULL,
      maxiter = 1000,
```

```
        run = maxiter,
        maxFitness = Inf,
        names = NULL,
        suggestions = NULL,
        optim = FALSE,
        optimArgs = list(method = "L-BFGS-B",
                         poptim = 0.05,
                         pressel = 0.5,
                         control = list(fnscale = -1, maxit = 100)),
        parallel = TRUE,
        monitor = if(interactive()) gaislMonitor else FALSE,
        seed = NULL)
```

Arguments

`type`	the type of genetic algorithm to be run depending on the nature of decision variables. Possible values are: `"binary"` for binary representations of decision variables. `"real-valued"` for optimization problems where the decision variables are floating-point representations of real numbers. `"permutation"` for problems that involves reordering of a list.
`fitness`	the fitness function, any allowable R function which takes as input an individual `string` representing a potential solution, and returns a numerical value describing its "fitness".
`...`	additional arguments to be passed to the fitness function. This allows to write fitness functions that keep some variables fixed during the search.
`lower`	a vector of length equal to the decision variables providing the lower bounds of the search space in case of real-valued or permutation encoded optimizations. Formerly this argument was named `min`; its usage is allowed but deprecated.
`upper`	a vector of length equal to the decision variables providing the upper bounds of the search space in case of real-valued or permutation encoded optimizations. Formerly this argument was named `max`; its usage is allowed but deprecated.
`nBits`	a value specifying the number of bits to be used in binary encoded optimizations.
`population`	an R function for randomly generating an initial population. See `ga_Population` for available functions.
`numIslands`	an integer value specifying the number of islands to be used in a *ring topology*, in which each island is connected unidirectionally with another island, hence forming a single continuous pathway.
`migrationRate`	a value in the range $[0,1]$ providing the proportion of individuals that should migrate between the islands.
`migrationInterval`	an integer value specifying the number of iterations at which exchange of individuals takes place.
`selection`	an R function performing selection, i.e. a function which generates a new population of individuals from the current population probabilistically according to individual fitness. See `ga_Selection` for available functions.

`crossover` an R function performing crossover, i.e. a function which forms offsprings by combining part of the genetic information from their parents. See `ga_Crossover` for available functions.

`mutation` an R function performing mutation, i.e. a function which randomly alters the values of some genes in a parent chromosome. See `ga_Mutation` for available functions.

`popSize` the population size.

`updatePop` a logical defaulting to `FALSE`. If set at `TRUE` the first attribute attached to the value returned by the user-defined fitness function is used to update the population. *Be careful though, this is an experimental feature!*

`postFitness` a user-defined function which, if provided, receives the current `ga-class` object as input, performs post fitness-evaluation steps, then returns an updated version of the object which is used to update the GA search. *Be careful though, this is an experimental feature!*

`pcrossover` the probability of crossover between pairs of chromosomes. Typically this is a large value and by default is set to 0.8.

`pmutation` the probability of mutation in a parent chromosome. Usually mutation occurs with a small probability, and by default is set to 0.1.

`elitism` the number of best fitness individuals to survive at each generation. By default the top 5% individuals in each island will survive at each iteration.

`maxiter` the maximum number of iterations to run before the GA search is halted.

`run` the number of consecutive generations without any improvement in the best fitness value before the GA is stopped.

`maxFitness` the upper bound on the fitness function after that the GA search is interrupted.

`names` a vector of character strings providing the names of decision variables.

`suggestions` a matrix of solutions strings to be included in the initial population. If provided the number of columns must match the number of decision variables.

`optim` a logical defaulting to `FALSE` determining whether or not a local search using general-purpose optimisation algorithms should be used. See argument `optimArgs` for further details and finer control.

`optimArgs` a list controlling the local search algorithm with the following components:

- `method` a string specifying the general-purpose optimisation method to be used, by default is set to `"L-BFGS-B"`. Other possible methods are those reported in `optim`.
- `poptim` a value in the range [0,1] specifying the probability of performing a local search at each iteration of GA (default 0.1).
- `pressel` a value in the range [0,1] specifying the pressure selection (default 0.5). The local search is started from a random solution selected with probability proportional to fitness. High values of `pressel` tend to select the solutions with the largest fitness, whereas low values of `pressel` assign quasi-uniform probabilities to any solution.
- `control` a list of control parameters. See 'Details' section in `optim`.

`parallel` An optional argument which allows to specify if the Islands Genetic Algorithm should be run sequentially or in parallel.

For a single machine with multiple cores, possible values are:

- a logical value specifying if parallel computing should be used (`TRUE`) or not (`FALSE`, default) for running GAs on each island;
- a numerical value which gives the number of cores to employ. By default, this is obtained from the function `detectCores`;
- a character string specifying the type of parallelisation to use. This depends on system OS: on Windows OS only `"snow"` type functionality is available, while on Unix/Linux/Mac OSX both `"snow"` and `"multicore"` (default) functionalities are available.

In all the cases described above, at the end of the search the cluster is automatically stopped by shutting down the workers.

If a cluster of multiple machines is available, evolution of GAs on each island can be executed in parallel using all, or a subset of, the cores available to the machines belonging to the cluster. However, this option requires more work from the user, who needs to set up and register a parallel back end. In this case the cluster must be explicitly stopped with `stopCluster`.

`monitor` a logical or an R function which takes as input the current state of the `gaisl-class` object and show the evolution of the search in different epochs. By default, for interactive sessions, the function `gaislMonitor` prints the average and best fitness values at each epoch for each island. In non interactive sessions, by default `monitor = FALSE` so any output is suppressed.

`seed` an integer value containing the random number generator state. This argument can be used to replicate the results of a ISLGA search. Note that if parallel computing is required, the **doRNG** package must be installed.

Details

Genetic algorithms (GAs) are stochastic search algorithms inspired by the basic principles of biological evolution and natural selection. GAs simulate the evolution of living organisms, where the fittest individuals dominate over the weaker ones, by mimicking the biological mechanisms of evolution, such as selection, crossover and mutation.

The `gaisl` function implements the islands GAs approach, where the population is partitioned into several subpopulations and assigned to separated islands. Independent GAs are executed in each island, and only occasionally sparse exchanges of individuals are performed among the islands. The algorithm can be run in parallel or sequentially. For more information on GAs see `ga`.

Value

Returns an object of class `gaisl-class`. See `gaisl-class` for a description of available slots information.

Author(s)

Luca Scrucca <luca.scrucca@unibo.it>

References

Luque G., Alba E. (2011) *Parallel Genetic Algorithms: Theory and Real World Applications*. Springer.

Luke S. (2013) *Essentials of Metaheuristics*, 2nd edition. Lulu.

Scrucca, L. (2017) On some extensions to GA package: hybrid optimisation, parallelisation and islands evolution. *The R Journal*, 9/1, 187-206. doi:10.32614/RJ2017008

See Also

summary,gaisl-method, plot,gaisl-method, gaisl-class, ga

Examples

```
## Not run:
# two-dimensional Rastrigin function
Rastrigin <- function(x1, x2)
{
  20 + x1^2 + x2^2 - 10*(cos(2*pi*x1) + cos(2*pi*x2))
}

x1 <- x2 <- seq(-5.12, 5.12, by = 0.1)
f <- outer(x1, x2, Rastrigin)
persp3D(x1, x2, f, theta = 50, phi = 20)
filled.contour(x1, x2, f, color.palette = jet.colors)

GA <- gaisl(type = "real-valued",
            fitness =  function(x) -Rastrigin(x[1], x[2]),
            lower = c(-5.12, -5.12), upper = c(5.12, 5.12),
            popSize = 80, maxiter = 500,
            numIslands = 4, migrationInterval = 50)
summary(GA)
plot(GA)

## End(Not run)
```

---

gaisl-class *Class "gaisl"*

Description

An S4 class for islands genetic algorithms (ISLGAs)

Objects from the Class

Objects can be created by calls to the gaisl function.

Slots

`call` an object of class `"call"` representing the matched call;

`type` a character string specifying the type of genetic algorithm used;

`lower` a vector providing for each decision variable the lower bounds of the search space in case of real-valued or permutation encoded optimisations. Formerly this slot was named `min`;

`upper` a vector providing for each decision variable the upper bounds of the search space in case of real-valued or permutation encoded optimizations. Formerly this slot was named `max`;

`nBits` a value specifying the number of bits to be used in binary encoded optimizations;

`names` a vector of character strings providing the names of decision variables (optional);

`popSize` the population size;

`numIslands` the number of islands;

`migrationRate` the migration rate;

`migrationInterval` the migration interval;

`maxiter` the maximum number of ISLGA iterations before the search is halted;

`run` the number of consecutive generations without any improvement in the best fitness value before the ISLGA is stopped;

`maxiter` the maximum number of iterations to run before the GA search is halted;

`suggestions` a matrix of user provided solutions and included in the initial population;

`elitism` the number of best fitness individuals to survive at each generation;

`pcrossover` the crossover probability;

`pmutation` the mutation probability;

`optim` a logical specifying whether or not a local search using general-purpose optimisation algorithms should be used;

`islands` a list containing the objects of class `ga` corresponding to each island GA evolution;

`summary` a list of matrices of summary statistics for fitness values at each iteration (along the rows). Each element of the list corresponds to the evolution of an island;

`fitnessValues` a list of best fitness values found in each island at the final iteration;

`solutions` a list of matrices, one for each island, containing the values of the decision variables giving the best fitness at the final iteration;

`fitnessValue` the best fitness value at the final iteration;

`solution` a matrix containing the values of the decision variables giving the best fitness at the final iteration.

Author(s)

Luca Scrucca

See Also

For examples of usage see `gaisl`.

gaMonitor *Monitor genetic algorithm evolution*

Description

Functions to print summary statistics of fitness values at each iteration of a GA search.

Usage

```
gaMonitor(object, digits = getOption("digits"), ...)

gaislMonitor(object, digits = getOption("digits"), ...)
```

Arguments

object	an object of class ga-class or gaisl-class, usually resulting from a call to function ga or gaisl, respectively.
digits	minimal number of significant digits.
...	further arguments passed to or from other methods.

Value

These functions print a summary of current GA step on the console.
By default, gaMonitor is called in interactive sessions by ga. The old monitoring function, used as the default until version 2.2 of **GA** package, is provided in gaMonitor2.
By default, gaislMonitor is called in interactive sessions by gaisl.

Author(s)

Luca Scrucca

gaSummary *Summarize genetic algorithm evolution*

Description

A function which returns fitness summary statistics at each iteration of GA search.

Usage

```
gaSummary(x, ...)
```

Arguments

x	a vector of fitness values for which summary statistics should be computed.
...	further arguments passed to or from other methods.

Details

This function computes summary statistics for a vector of fitness values at current iteration of GA search.

Value

A vector with the following values: `(max, mean, median, min)`

Author(s)

Luca Scrucca

See Also

ga

`ga_Crossover` *Crossover operators in genetic algorithms*

Description

Functions implementing crossover genetic operator.

Usage

```
ga_spCrossover(object, parents, ...)

gabin_spCrossover(object, parents, ...)
gabin_uCrossover(object, parents, ...)

gareal_spCrossover(object, parents, ...)
gareal_waCrossover(object, parents, ...)
gareal_laCrossover(object, parents, ...)
gareal_blxCrossover(object, parents, a = 0.5, ...)
gareal_laplaceCrossover(object, parents, a = 0, b = 0.15, ...)

gaperm_cxCrossover(object, parents, ...)
gaperm_pmxCrossover(object, parents, ...)
gaperm_oxCrossover(object, parents, ...)
gaperm_pbxCrossover(object, parents, ...)
```

Arguments

`object`	An object of class `"ga"`, usually resulting from a call to function ga.
`parents`	A two-rows matrix of values indexing the parents from the current population.
`...`	Further arguments passed to or from other methods.
`a, b`	Parameters of genetic operators.

Value

Return a list with two elements:

children	a matrix of dimension 2 times the number of decision variables containing the generated offsprings;
fitness	a vector of length 2 containing the fitness values for the offsprings. A value NA is returned if an offspring is different (which is usually the case) from the two parents.

Author(s)

Luca Scrucca

See Also

ga

---

ga_Mutation *Mutation operators in genetic algorithms*

Description

Functions implementing mutation genetic operator.

Usage

```
gabin_raMutation(object, parent, ...)

gareal_raMutation(object, parent, ...)
gareal_nraMutation(object, parent, ...)
gareal_rsMutation(object, parent, ...)
gareal_powMutation(object, parent, pow = 10, ...)

gaperm_simMutation(object, parent, ...)
gaperm_ismMutation(object, parent, ...)
gaperm_swMutation(object, parent, ...)
gaperm_dmMutation(object, parent, ...)
gaperm_scrMutation(object, parent, ...)
```

Arguments

object	An object of class "ga", usually resulting from a call to function ga.
parent	A vector of values for the parent from the current population where mutation should occur.
...	Further arguments passed to or from other methods.
pow	Parameters of genetic operators.

Value

Return a vector of values containing the mutated string.

Author(s)

Luca Scrucca

`ga_pmutation` *Variable mutation probability in genetic algorithms*

Description

A function which calculates the mutation probability for the current iteration. This enables to use GAs with variable mutation rate (see examples).

Usage

```
ga_pmutation(object, p0 = 0.5, p = 0.01, T = round(object@maxiter/2), ...)
```

Arguments

`object`	An object of class `"ga"`, usually resulting from a call to function `ga`.
`p0`	initial probability of mutation.
`p`	limiting probability of mutation.
`T`	maximum iteration after which it should converges to `p`.
`...`	Further arguments passed to or from other methods.

Value

Return a numeric value in the range (0,1).

Author(s)

Luca Scrucca

See Also

`ga`, `ga_Mutation`

Examples

```
## Not run:
Rastrigin <- function(x1, x2)
{
  20 + x1^2 + x2^2 - 10*(cos(2*pi*x1) + cos(2*pi*x2))
}

GA <- ga(type = "real-valued",
         fitness =  function(x) -Rastrigin(x[1], x[2]),
         lower = c(-5.12, -5.12), upper = c(5.12, 5.12),
         popSize = 50, maxiter = 500, run = 100,
         pmutation = ga_pmutation)
plot(GA)

GA <- ga(type = "real-valued",
         fitness =  function(x) -Rastrigin(x[1], x[2]),
         lower = c(-5.12, -5.12), upper = c(5.12, 5.12),
         popSize = 50, maxiter = 500, run = 100,
         pmutation = function(...) ga_pmutation(..., p0 = 0.1))
plot(GA)

## End(Not run)
```

---

ga_Population *Population initialization in genetic algorithms*

Description

Functions for creating a random initial population to be used in genetic algorithms.

Usage

```
gabin_Population(object, ...)

gareal_Population(object, ...)

gaperm_Population(object, ...)
```

Arguments

`object`	An object of class `"ga"`, usually resulting from a call to function `ga`.
`...`	Further arguments passed to or from other methods.

Details

`gabin_Population` generates a random population of `object@nBits` binary values;

`gareal_Population` generates a random (uniform) population of real values in the range [`object@min`, `object@max`];

gaperm_Population generates a random (uniform) population of integer values in the range [object@min, object@max].

Value

Return a matrix of dimension object@popSize times the number of decision variables.

Author(s)

Luca Scrucca

See Also

ga

---

ga_Selection *Selection operators in genetic algorithms*

Description

Functions implementing selection genetic operator.

Usage

```
ga_lrSelection(object, r = 2/(object@popSize * (object@popSize - 1)),
                       q = 2/object@popSize, ...)
ga_nlrSelection(object, q = 0.25, ...)
ga_rwSelection(object, ...)
ga_tourSelection(object, k = 3, ...)

gabin_lrSelection(object, r = 2/(object@popSize * (object@popSize - 1)),
                          q = 2/object@popSize, ...)
gabin_nlrSelection(object, q = 0.25, ...)
gabin_rwSelection(object, ...)
gabin_tourSelection(object, k = 3, ...)

gareal_lrSelection(object, r = 2/(object@popSize * (object@popSize - 1)),
                           q = 2/object@popSize, ...)
gareal_nlrSelection(object, q = 0.25, ...)
gareal_rwSelection(object, ...)
gareal_tourSelection(object, k = 3, ...)
gareal_lsSelection(object, ...)
gareal_sigmaSelection(object, ...)

gaperm_lrSelection(object, r = 2/(object@popSize * (object@popSize - 1)),
                           q = 2/object@popSize, ...)
gaperm_nlrSelection(object, q = 0.25, ...)
gaperm_rwSelection(object, ...)
```

```
gaperm_tourSelection(object, k = 3, ...)

gareal_de(object, F = 0.8, p = 0.5, ...)
```

Arguments

object	An object of class "ga", usually resulting from a call to function ga.
r	A tuning parameter for the GA selection operator.
q	A tuning parameter for the GA selection operator.
k	A tuning parameter for the GA selection operator.
F, p	Tuning parameters for the DE selection operator.
...	Further arguments passed to or from other methods.

Value

Return a list with two elements:

population	a matrix of dimension object@popSize times the number of decision variables containing the selected individuals or strings;
fitness	a vector of length object@popSize containing the fitness values for the selected individuals.

Author(s)

Luca Scrucca

See Also

ga, de.

---

numericOrNA-class *Virtual Class "numericOrNA" - Simple Class for sub-assignment Values*

Description

The class "numericOrNA" is a simple class union (setClassUnion) of "numeric" and "logical".

Objects from the Class

Since it is a virtual Class, no objects may be created from it.

Examples

```
showClass("numericOrNA")
```

palettes *Colours palettes*

Description

Functions for creating a vector of colours from pre-specified palettes.

Usage

```
jet.colors(n)

spectral.colors(n)

bl2gr.colors(n)
```

Arguments

n	a numerical value specifying the number of colours in the palette.

Details

`jet.colors()` creates a palette of colours which tend to have high brightness and not uniform luminance. Furthermore, the brightest colours, yellow and cyan, are used for intermediate data values, and this has the effect of emphasizing uninteresting (and arbitrary) values while de-emphasizing the extremes. For these reasons this popular palette is not recommended.
`spectral.colors()` creates a palette based on ColorBrewer https://colorbrewer2.org, so the resulting colours have a much uniform luminance.
The `bl2gr.colors()` palette returns a palette of colours from blue to green.

Value

Returns a character vector of colours encoded in hexadecimal values.

See Also

colors, rgb.

Examples

```
jet.colors(9)
spectral.colors(9)
bl2gr.colors(9)

par(mfrow = c(3,1), mar = c(1,1,1,1))
n = 21
image(1:21, 1, as.matrix(1:21), col = jet.colors(21),
      ylab = "", xlab = "", xaxt = "n", yaxt = "n", bty = "n")
image(1:21, 1, as.matrix(1:21), col = spectral.colors(21),
      ylab = "", xlab = "", xaxt = "n", yaxt = "n", bty = "n")
```

```
image(1:21, 1, as.matrix(1:21), col = bl2gr.colors(21),
      ylab = "", xlab = "", xaxt = "n", yaxt = "n", bty = "n")
```

---

parNames-methods *Parameters or decision variables names from an object of class* ga-class.

Description

A method for obtaining the names of parameters or decision variables from an object of class ga-class.

Usage

```
parNames(object, ...)
## S4 method for signature 'ga'
parNames(object, ...)
```

Arguments

object	An object of class "ga", usually resulting from a call to function ga.
...	Further arguments, currently not used.

Value

A list of character values providing the names of parameters or decision variables.

Author(s)

Luca Scrucca

See Also

ga

persp3D *Perspective plot with colour levels*

Description

This function draws a perspective plot of a surface with different levels in different colours.

Usage

```
persp3D(x, y, z, theta = 30, phi = 20, d = 5, expand = 2/3,
        xlim = range(x, finite = TRUE), ylim = range(y, finite = TRUE),
        zlim = range(z, finite = TRUE), levels = pretty(zlim, nlevels),
        nlevels = 20, col.palette = jet.colors, border = NA,
        ticktype = "detailed", xlab = NULL, ylab = NULL, zlab = NULL,
        ...)
```

Arguments

x, y	locations of grid lines at which the values in z are measured. These must be in ascending order. By default, equally spaced values from 0 to 1 are used. If x is a list, its components x$x and x$y are used for x and y, respectively.
z	a matrix containing the values to be plotted (NAs are allowed).
theta, phi	angles defining the viewing direction. theta gives the azimuthal direction and phi the colatitude.
d	a value which can be used to vary the strength of the perspective transformation.
expand	a expansion factor applied to the z coordinates.
xlim, ylim, zlim	x-, y- and z-limits for the axes.
levels	a vector of values specifying the levels to be used for plotting the surface with different colours.
nlevels	a value specifying the number of levels to be used for plotting. This value is used if levels argument is not specified.
col.palette	the colour palette used for plotting.
border	the colour of the line drawn around the surface facets. By default is set to NA so no borders are drawn.
ticktype	a character specifying the type of axes tickmarks. By default "detailed" ticks are drawn.
xlab, ylab, zlab	character strings specifying the titles for the axes.
...	Further arguments passed to the function persp.

Details

This function enhances the default perspective plot for drawing 3-dimensional surfaces.

Value

Return a list with the following elements:

persp	the viewing transformation matrix (see persp);
levels	a vector of values giving the levels used for plotting the surface;
colors	a vector of strings giving the colour used for plotting the surface.

Author(s)

Luca Scrucca

See Also

persp

Examples

```
y <- x <- seq(-10, 10, length=60)
f <- function(x,y) { r <- sqrt(x^2+y^2); 10 * sin(r)/r }
z <- outer(x, y, f)
persp3D(x, y, z, theta = 30, phi = 30, expand = 0.5)
persp3D(x, y, z, col.palette = heat.colors, phi = 30, theta = 225,
        box = TRUE, border = NA, shade = .4)
x1 <- seq(-3,3,length=50)
x2 <- seq(-3,3,length=50)
y <- function(x1, x2) sin(x1)+cos(x2)
persp3D(x1, x2, outer(x1,x2,y), zlab="y", theta = 150, phi = 20, expand = 0.6)
```

---

plot.de-method *Plot of Differential Evolution search path*

Description

The plot method for de-class objects gives a plot of best and average fitness values found during the iterations of the DE search.

Usage

```
## S4 method for signature 'de'
plot(x, y, ylim, cex.points = 0.7,
     col = c("green3", "dodgerblue3",  adjustcolor("green3", alpha.f = 0.1)),
     pch = c(16, 1), lty = c(1,2), legend = TRUE, grid = graphics::grid, ...)
```

Arguments

x	An object of class "ga".
y	Not used.
ylim	A vector of two values specifying the limits on the y-axis.
cex.points	The magnification to be used for points.
col	The colours to be used for best and average fitness values.
pch	The type of points to be used for best and average fitness values.
lty	The type of lines to be used for best and average fitness values.
legend	A logical specifying if a legend should be included.
grid	A function for grid drawing of NULL to avoid drawing one.
...	Further arguments, currently not used.

Details

Plot best and average fitness values at each iteration of DE search.

Value

The method invisibly return a data.frame with the iterations and summary statistics for the fitness function evaluated at each iteration.

Author(s)

Luca Scrucca

See Also

de, de-class, plot,de-method.

Examples

```
# See examples in help(de)
```

---

plot.ga-method *Plot of Genetic Algorithm search path*

Description

The plot method for ga-class objects gives a plot of best and average fitness values found during the iterations of the GA search.

Usage

```
## S4 method for signature 'ga'
plot(x, y, ylim, cex.points = 0.7,
     col = c("green3", "dodgerblue3",  adjustcolor("green3", alpha.f = 0.1)),
     pch = c(16, 1), lty = c(1,2), legend = TRUE, grid = graphics::grid, ...)
```

Arguments

x	An object of class "ga".
y	Not used.
ylim	A vector of two values specifying the limits on the y-axis.
cex.points	The magnification to be used for points.
col	The colours to be used for best and average fitness values.
pch	The type of points to be used for best and average fitness values.
lty	The type of lines to be used for best and average fitness values.
legend	A logical specifying if a legend should be included.
grid	A function for grid drawing of NULL to avoid drawing one.
...	Further arguments, currently not used.

Details

Plot best and average fitness values at each iteration of GA search.

Value

The method invisibly return a `data.frame` with the iterations and summary statistics for the fitness function evaluated at each iteration.

Author(s)

Luca Scrucca

See Also

ga, ga-class.

Examples

```
# See examples in help(ga)

# The following code shows how to obtain graphs using the
# ggplot2 plotting system
## Not run:
GA <- ga(type = "real-valued",
         fitness = function(x) -(abs(x)+cos(x)),
         lower = -20, upper = 20,
         popSize = 20, pmutation = 0.2, maxiter = 50)
```

```
out <- plot(GA)
library(reshape2)
df <- melt(out[,c(1:3,5)], id.var = "iter")
library(ggplot2)
ggplot(out) +
  geom_ribbon(aes(x = iter, ymin = median, ymax = max,
                  colour = "median", fill = "median")) +
  geom_line(aes(x = iter, y = max, colour = "max")) +
  geom_point(aes(x = iter, y = max, colour = "max")) +
  geom_line(aes(x = iter, y = mean, colour = "mean"), lty = 2) +
  geom_point(aes(x = iter, y = mean, colour = "mean"), pch = 1) +
  xlab("Generation") +  ylab("Fitness values") +
  scale_colour_manual(breaks = c("max", "mean", "median"),
              values = c("green3", "dodgerblue3", adjustcolor("green3", alpha.f = 0.1))) +
  scale_fill_manual(breaks = "median",
                    values = adjustcolor("green3", alpha.f = 0.1)) +
  guides(fill = "none",
         colour = guide_legend(override.aes =
                             list(fill = c(NA, NA, adjustcolor("green3", alpha.f = 0.1)),
                                  pch = c(19,1,NA)))) +
  theme_bw() +
  theme(legend.title = element_blank(),
        legend.pos = "top",
        legend.background = element_blank())

## End(Not run)
```

`plot.gaisl-method` *Plot of Islands Genetic Algorithm search path*

Description

The `plot` method for `gaisl-class` objects gives a plot of best fitness values found in each island during the GA iterations.

Usage

```
## S4 method for signature 'gaisl'
plot(x, y, ...)
```

Arguments

`x`	An object of class `"gaisl"`.
`y`	Not used.
`...`	Further arguments passed to `plot.default`, such as `ylim`, `ylab`, etc., or to `matplot`, such as `col`, `lty`, and `lwd`.

Details

Plot best fitness values found in each island during the GA iterations.

Value

The method invisibly return a list with the following components:

iter	a vector of values specifying the iteration.
summary	a matrix of best fitness values for each island along the columns.

Author(s)

Luca Scrucca

See Also

gaisl, gaisl-class.

Examples

```
# See examples in help(gaisl)
```

---

summary.de-method *Summary for Differential Evolution*

Description

Summary method for class de-class.

Usage

```
## S4 method for signature 'de'
summary(object, ...)

## S3 method for class 'summary.de'
print(x, digits = getOption("digits"), ...)
```

Arguments

object	an object of class de-class.
x	an object of class summary.de.
digits	number of significant digits.
...	further arguments passed to or from other methods.

Value

The summary function returns an object of class summary.de which can be printed by the corresponding print method. The function also returns invisibly a list with the information from the differential evolution search.

Author(s)

Luca Scrucca

See Also

de

Examples

```
f <- function(x) abs(x)+cos(x)
DE <- de(fitness = function(x) -f(x),
         lower = -20, upper = 20, run = 50)
out <- summary(DE)
print(out)
str(out)
```

---

summary.ga-method *Summary for Genetic Algorithms*

Description

Summary method for class ga-class.

Usage

```
## S4 method for signature 'ga'
summary(object, ...)

## S3 method for class 'summary.ga'
print(x, digits = getOption("digits"), ...)
```

Arguments

object	an object of class ga-class.
x	an object of class summary.ga.
digits	number of significant digits.
...	further arguments passed to or from other methods.

Value

The summary function returns an object of class summary.ga which can be printed by the corresponding print method. The function also returns invisibly a list with the information from the genetic algorithm search.

Author(s)

Luca Scrucca

See Also

ga

Examples

```
f <- function(x)  abs(x)+cos(x)
GA <- ga(type = "real-valued",
         fitness = function(x) -f(x),
         lower = -20, upper = 20, run = 50)
out <- summary(GA)
print(out)
str(out)
```

---

summary.gaisl-method *Summary for Islands Genetic Algorithms*

Description

Summary method for class gaisl-class.

Usage

```
## S4 method for signature 'gaisl'
summary(object, ...)

## S3 method for class 'summary.gaisl'
print(x, digits = getOption("digits"), ...)
```

Arguments

object	an object of class gaisl-class.
x	an object of class summary.gaisl.
digits	number of significant digits.
...	further arguments passed to or from other methods.

Value

The summary function returns an object of class summary.gaisl which can be printed by the corresponding print method. The function also returns invisibly a list with the information from the islands genetic algorithm search.

Author(s)

Luca Scrucca

See Also

gaisl

Examples

```
## Not run:
f <- function(x)  abs(x)+cos(x)
GA <- gaisl(type = "real-valued",
            fitness = function(x) -f(x),
            lower = -20, upper = 20, run = 10,
            numIslands = 4)
out <- summary(GA)
print(out)
str(out)

## End(Not run)
```

Index