Package ‘gena’

October 13, 2022

Type Package

Title Genetic Algorithm and Particle Swarm Optimization

Version 1.0.0

Date 2022-08-08

Description Implements genetic algorithm and particle swarm algorithm for real-valued functions. Various modifications (including hybridization and elitism) of these algorithms are provided. Implemented functions are based on ideas described in S. Katoch, S. Chauhan, V. Kumar (2020) <doi:10.1007/s11042-020-10139-6> and M. Clerc (2012) <`https://hal.archives-ouvertes.fr/hal-00764996`>.

Imports Rcpp (>= 1.0.6)

LinkingTo Rcpp, RcppArmadillo

License GPL (>= 2)

RoxygenNote 7.2.1

NeedsCompilation yes

Author Bogdan Potanin [aut, cre, ctb]

Maintainer Bogdan Potanin <`bogdanpotanin@gmail.com`>

Repository CRAN

Date/Publication 2022-08-15 08:20:02 UTC

R topics documented:

gena *Genetic Algorithm*

Description

This function allows to use genetic algorithm for numeric global optimization of real-valued functions.

Usage

```
gena(
  fn,
  gr = NULL,
  lower,
  upper,
  pop.n = 100,
  pop.initial = NULL,
  pop.method = "uniform",
  mating.method = "rank",
  mating.par = NULL,
  mating.self = FALSE,
  crossover.method = "local",
  crossover.par = NULL,
  crossover.prob = 0.8,
  mutation.method = "constant",
  mutation.par = NULL,
  mutation.prob = 0.2,
  mutation.genes.prob = 1/length(lower),
  elite.n = min(10, 2 * round(pop.n/20)),
  elite.duplicates = FALSE,
  hybrid.method = "rank",
  hybrid.par = 2,
  hybrid.prob = 0,
  hybrid.opt.par = NULL,
  hybrid.n = 1,
  constr.method = NULL,
  constr.par = NULL,
  maxiter = 100,
  is.max = TRUE,
  info = TRUE,
  ...
)
```

Arguments

`fn`	function to be maximized i.e. fitness function.
`gr`	gradient of the `fn`.
`lower`	lower bound of the search space.
`upper`	upper bound of the search space.
`pop.n`	integer representing the size of the population.
`pop.initial`	numeric matrix which rows are chromosomes to be included into the initial population. Numeric vector will be coerced to single row matrix.
`pop.method`	the algorithm to be applied for a creation of the initial population. See 'Details' for additional information.
`mating.method`	the algorithm to be applied for a mating i.e. selection of parents. See 'Details' for additional information.
`mating.par`	parameters of the mating (selection) algorithm.
`mating.self`	logical; if `TRUE` then the chromosome may mate with itself i.e. both parents may be the same chromosome.
`crossover.method`	an algorithm to be applied for crossover i.e. creation of the children. See 'Details' for additional information.
`crossover.par`	parameters of the crossover algorithm.
`crossover.prob`	probability of the crossover for each pair of parents.
`mutation.method`	algorithm to be applied for mutation i.e. random change in some genes of the children. See 'Details' for additional information.
`mutation.par`	parameters of the mutation algorithm.
`mutation.prob`	mutation probability for the chromosomes.
`mutation.genes.prob`	mutation probability for the genes.
`elite.n`	number of the elite children i.e. those which have the highest function value and will be preserved for the next population.
`elite.duplicates`	logical; if `TRUE` then some elite children may have the same genes.
`hybrid.method`	hybrids selection algorithm i.e. mechanism determining which chromosomes should be subject to local optimization. See 'Details' for additional information.
`hybrid.par`	parameters of the hybridization algorithm.

`hybrid.prob`	probability of generating the hybrids each iteration.
`hybrid.opt.par`	parameters of the local optimization function to be used for hybridization algorithm (including `fn` and `gr`).
`hybrid.n`	number of hybrids that appear if hybridization should take place during the iteration.
`constr.method`	the algorithm to be applied for imposing constraints on the chromosomes. See 'Details' for additional information.
`constr.par`	parameters of the constraint algorithm.
`maxiter`	maximum number of iterations of the algorithm.
`is.max`	logical; if `TRUE` (default) then fitness function will be maximized. Otherwise it will be minimized.
`info`	logical; if `TRUE` (default) then some optimization related information will be printed each iteration.
`...`	additional parameters to be passed to `fn` and `gr` functions.

Details

To find information on particular methods available via `pop.method`, `mating.method`, `crossover.method`, `mutation.method`, `hybrid.method` and `constr.method` arguments please see 'Details' section of `gena.population`, `gena.crossover`, `gena.mutation`, `gena.hybrid` and `gena.constr` correspondingly. For example to find information on possible values of `mutation.method` and `mutation.par` arguments see description of `method` and `par` arguments of `gena.mutation` function.

It is possible to provide manually implemented functions for population initialization, mating, crossover, mutation and hybridization. For example manual mutation function may be provided through `mutation.method` argument. It should have the same signature (arguments) as `gena.mutation` function and return the same object i.e. the matrix of chromosomes of the appropriate size. Manually implemented functions for other operators (crossover, mating and so on) may be provided in a similar way.

By default function does not impose any constraints upon the parameters. If `constr.method = "bounds"` then `lower` and `upper` constraints will be imposed. Lower bounds should be strictly smaller then upper bounds.

Currently the only available termination condition is `maxiter`. We are going to provide some additional termination conditions during future updates.

Infinite values in `lower` and `upper` are substituted with `-(.Machine$double.xmax * 0.9)` and `.Machine$double.xmax * 0.9` correspondingly.

By default if `gr` is provided then BFGS algorithm will be used inside `optim` during hybridization. Otherwise `Nelder-Mead` will be used. Manual values for `optim` arguments may be provided (as a list) through `hybrid.opt.par` argument.

Arguments `pop.n` and `elite.n` should be even integers and `elite.n` should be greater then 2. If these arguments are odd integers then they will be coerced to even integers by adding 1. Also `pop.n` should be greater then `elite.n` at least by 2.

For more information on the genetic algorithm please see Katoch et. al. (2020).

Value

This function returns an object of class `gena` that is a list containing the following elements:

- `par` - chromosome (solution) with the highest fitness (objective function) value.
- `value` - value of `fn` at `par`.
- `population` - matrix of chromosomes (solutions) of the last iteration of the algorithm.
- `counts` - a two-element integer vector giving the number of calls to `fn` and `gr` respectively.
- `is.max` - identical to `is.max` input argument.
- `fitness.history` - vector which i-th element is fitness of the best chromosome in i-th iteration.
- `iter` - last iteration number.

References

S. Katoch, S. Chauhan, V. Kumar (2020). A review on genetic algorithm: past, present, and future. *Multimedia Tools and Applications*, 80, 8091-8126. <doi:10.1007/s11042-020-10139-6>

Examples

```
## Consider Ackley function

fn <- function(par, a = 20, b = 0.2)
{
  val <- a * exp(-b * sqrt(0.5 * (par[1] ^ 2 + par[2] ^ 2))) +
         exp(0.5 * (cos(2 * pi * par[1]) + cos(2 * pi * par[2]))) -
         exp(1) - a
  return(val)
}

# Maximize this function using classical
# genetic algorithm setup
set.seed(123)
lower <- c(-5, -100)
upper <- c(100, 5)
opt <- gena(fn = fn,
            lower = lower, upper = upper,
            hybrid.prob = 0,
            a = 20, b = 0.2)
print(opt$par)

# Replicate optimization using hybridization
opt <- gena(fn = fn,
            lower = lower, upper = upper,
            hybrid.prob = 0.2,
            a = 20, b = 0.2)
print(opt$par)


## Consider Rosenbrock function
```

```
fn <- function(par, a = 100)
{
  val <- -(a * (par[2] - par[1] ^ 2) ^ 2 + (1 - par[1]) ^ 2 +
           a * (par[3] - par[2] ^ 2) ^ 2 + (1 - par[2]) ^ 2)
  return(val)
}

# Apply genetic algorithm
lower <- rep(-10, 3)
upper <- rep(10, 3)
set.seed(123)
opt <- gena(fn = fn,
            lower = lower, upper = upper,
            a = 100)
print(opt$par)


# Improve the results by hybridization
opt <- gena(fn = fn,
            lower = lower, upper = upper,
            hybrid.prob = 0.2,
            a = 100)
print(opt$par)


# Provide manually implemented mutation function
# which simply randomly sorts genes.
# Note that this function should have the same
# arguments as gena.mutation.
mutation.my <- function(children, lower, upper,
                        prob, prob.genes,
                        method, par, iter)
{
  # Get dimensional data
  children.n <- nrow(children)
  genes.n <- ncol(children)

  # Select chromosomes that should mutate
  random_values <- runif(children.n, 0, 1)
  mutation_ind <- which(random_values <= prob)

  # Mutate chromosomes by randomly sorting
  # their genes
  for (i in mutation_ind)
  {
    children[i, ] <- children[i, sample(1:genes.n)]
  }

  # Return mutated chromosomes
  return(children)
}

opt <- gena(fn = fn,
            lower = lower, upper = upper,
            mutation.method = mutation.my,
            a = 100)
print(opt$par)
```

`gena.constr` *Constraints*

Description

Impose constraints on chromosomes.

Usage

```
gena.constr(population, method = "bounds", par, iter)
```

Arguments

`population`	numeric matrix which rows are chromosomes i.e. vectors of parameters values.
`method`	method used to impose constraints.
`par`	additional parameters to be passed depending on the `method`.
`iter`	iteration number of the genetic algorithm.

Details

If `method = "bounds"` then chromosomes will be bounded between `par$lower` and `par$upper`.

Value

The function returns a list with the following elements:

- `population` - matrix which rows are chromosomes after constraints have been imposed.
- `constr.ind` - matrix of logical values which (i, j)-th elements equals `TRUE` (`FALSE` otherwise) if j-th jene of i-th chromosome is a subject to constraint.

Examples

```
# Randomly initialize population
set.seed(123)
population <- gena.population(pop.n = 10,
                              lower = c(-5, -5),
                              upper = c(5, 5))

# Impose lower and upper bounds constraints
pop.constr <- gena.constr(population,
                          method = "bounds",
                          par = list(lower = c(-1, 2),
                                   upper = c(1, 5)))
print(pop.constr)
```

gena.crossover *Crossover*

Description

Crossover method (algorithm) to be used in the genetic algorithm.

Usage

```
gena.crossover(
  parents,
  fitness = NULL,
  prob = 0.8,
  method = "local",
  par = NULL,
  iter = NULL
)
```

Arguments

`parents`	numeric matrix which rows are parents i.e. vectors of parameters values.
`fitness`	numeric vector which `i`-th element is the value of `fn` at point `population[i, ]`.
`prob`	probability of crossover.
`method`	crossover method to be used for making children.
`par`	additional parameters to be passed depending on the `method`.
`iter`	iteration number of the genetic algorithm.

Details

Denote `parents` by C^{parent} which `i`-th row `parents[i, ]` is a chromosome c_i^{parent} i.e. the vector of parameter values of the function being optimized $f(.)$ that is provided via `fn` argument of `gena`. The elements of chromosome c_{ij}^{parent} are genes representing parameters values.

Crossover algorithm determines the way parents produce children. During crossover each of randomly selected pairs of parents c_i^{parent}, c_{i+1}^{parent} produce two children c_i^{child}, c_{i+1}^{child}, where i is odd. Each pair of parents is selected with probability `prob`. If pair of parents have not been selected for crossover then corresponding children and parents are coincide i.e. $c_i^{child} = c_i^{parent}$ and $c_{i+1}^{child} = c_{i+1}^{parent}$.

Argument `method` determines particular crossover algorithm to be applied. Denote by τ the vector of parameters used by the algorithm. Note that τ corresponds to `par`.

If `method = "split"` then each gene of the first child will be equiprobably picked from the first or from the second parent. So c_{ij}^{child} may be equal to c_{ij}^{parent} or $c_{(i+1)j}^{parent}$ with equal probability. The second child is the reversal of the first one in a sense that if the first child gets particular gene of the first (second) parent then the second child gets this gene from the first (second) parent i.e. if $c_{ij}^{child} = c_{ij}^{parent}$ then $c_{(i+1)j}^{child} = c_{(i+1)j}^{parent}$; if $c_{ij}^{child} = c_{(i+1)j}^{parent}$ then $c_{(i+1)j}^{child} = c_{ij}^{parent}$.

If `method = "arithmetic"` then:

$$c_i^{child} = \tau_1 c_i^{parent} + (1 - \tau_1)\, c_{i+1}^{parent}$$

$$c_{i+1}^{child} = (1 - \tau_1)\, c_i^{parent} + \tau_1 c_{i+1}^{parent}$$

where τ_1 is `par[1]`. By default `par[1] = 0.5`.

If `method = "local"` then the procedure is the same as for "arithmetic" method but τ_1 is a uniform random value between 0 and 1.

If `method = "flat"` then c_{ij}^{child} is a uniform random number between c_{ij}^{parent} and $c_{(i+1)j}^{parent}$. Similarly for the second child $c_{(i+1)j}^{child}$.

For more information on crossover algorithms please see Kora, Yadlapalli (2017).

Value

The function returns a matrix which rows are children.

References

P. Kora, P. Yadlapalli. (2017). Crossover Operators in Genetic Algorithms: A Review. *International Journal of Computer Applications*, 162 (10), 34-36, <doi:10.5120/ijca2017913370>.

Examples

```
# Randomly initialize the parents
set.seed(123)
parents.n <- 10
parents <- gena.population(pop.n = parents.n,
                           lower = c(-5, -5),
                           upper = c(5, 5))

# Perform the crossover
children <- gena.crossover(parents = parents,
                           prob = 0.6,
                           method = "local")
print(children)
```

gena.hybrid *Hybridization*

Description

Hybridization method (algorithm) to be used in the genetic algorithm.

Usage

```
gena.hybrid(
  population,
  fitness,
  hybrid.n = 1,
  method,
  par,
  opt.par,
  info = FALSE,
  iter = NULL,
  ...
)
```

Arguments

`population`	numeric matrix which rows are chromosomes i.e. vectors of parameters values.
`fitness`	numeric vector which `i`-th element is the value of `fn` at point `population[i, ]`.
`hybrid.n`	positive integer representing the number of hybrids.
`method`	hybridization method to improve chromosomes via local search.
`par`	additional parameters to be passed depending on the `method`.
`opt.par`	parameters of the local optimization function to be used for hybridization algorithm (including `fn` and `gr`).
`info`	logical; if `TRUE` then some optimization related information will be printed each iteration.
`iter`	iteration number of the genetic algorithm.
`...`	additional parameters to be passed to `fn` and `gr` functions.

Details

This function uses `gena.mating` function to select hybrids. Therefore `method` and `par` arguments will be passed to this function. If some chromosomes selected to become hybrids are duplicated then these duplicates will not be subject to local optimization i.e. the number of hybrids will be decreased by the number of duplicates (actual number of hybrids during some iterations may be lower than `hybrid.n`).

Currently `optim` is the only available local optimizer. Therefore `opt.par` is a list containing parameters that should be passed to `optim`.

For more information on hybridization please see El-mihoub et. al. (2006).

Value

The function returns a list with the following elements:

- `population` - matrix which rows are chromosomes including hybrids.
- `fitness` - vector which i-th element is the fitness of the i-th chromosome.
- `hybrids.ind` - vector of indexes of chromosomes selected for hybridization.
- `counts` a two-element integer vector giving the number of calls to `fn` and `gr` respectively.

References

T. El-mihoub, A. Hopgood, L. Nolle, B. Alan (2006). Hybrid Genetic Algorithms: A Review. *Engineering Letters*, 13 (3), 124-137.

Examples

```
# Consider the following fitness function
fn <- function(x)
{
  val <- x[1] * x[2] - x[1] ^ 2 - x[2] ^ 2
}

# Also let's provide it's gradient (optional)
gr <- function(x)
{
  val <- c(x[2] - 2 * x[1],
           x[1] - 2 * x[2])
}

# Randomly initialize the population
set.seed(123)
n_population <- 10
population <- gena.population(pop.n = n_population,
                              lower = c(-5, -5),
                              upper = c(5, 5))

# Calculate fitness of each chromosome
fitness <- rep(NA, n_population)
for(i in 1:n_population)
{
  fitness[i] <- fn(population[i, ])
}

# Perform hybridization
hybrids <- gena.hybrid(population = population,
                       fitness = fitness,
                       opt.par = list(fn = fn,
                                      gr = gr,
                                      method = "BFGS",
                                      control = list(fnscale = -1,
                                                     abstol = 1e-10,
                                                     reltol = 1e-10,
                                                     maxit = 1000)),
                       hybrid.n = 2,
                       method = "rank",
                       par = 0.8)
print(hybrids)
```

gena.mating *Mating*

Description

Mating (selection) method (algorithm) to be used in the genetic algorithm.

Usage

```
gena.mating(
  population,
  fitness,
  parents.n,
  method = "rank",
  par = NULL,
  self = FALSE,
  iter = NULL
)
```

Arguments

`population`	numeric matrix which rows are chromosomes i.e. vectors of parameters values.
`fitness`	numeric vector which `i`-th element is the value of `fn` at point `population[i, ]`.
`parents.n`	even positive integer representing the number of parents.
`method`	mating method to be used for selection of parents.
`par`	additional parameters to be passed depending on the `method`.
`self`	logical; if `TRUE` then chromosome may mate itself. Otherwise mating is allowed only between different chromosomes.
`iter`	iteration number of the genetic algorithm.

Details

Denote `population` by C which `i`-th row `population[i, ]` is a chromosome c_i i.e. the vector of parameter values of the function being optimized $f(.)$ that is provided via `fn` argument of `gena`. The elements of chromosome c_{ij} are genes representing parameters values. Argument `fitness` is a vector of function values at corresponding chromosomes i.e. `fitness[i]` corresponds to $f_i = f(c_i)$. Total number of chromosomes in population $n_{population}$ equals to `nrow(population)`.

Mating algorithm determines selection of chromosomes that will become parents. During mating each iteration one of chromosomes become a parent until there are $n_{parents}$ (i.e. `parents.n`) parents selected. Each chromosome may become a parent multiple times or not become a parent at all.

Denote by c_i^s the i-th of selected parents. Parents c_i^s and c_{i+1}^s form a pair that will further produce a child (offspring), where i is odd. If `self = FALSE` then for each pair of parents (c_i^s, c_{i+1}^s) it is insured that $c_i^s \neq c_{i+1}^s$ except the case when there are several identical chromosomes in population. However `self` is ignored if `method` is `"tournament"`, so in this case self-mating is always possible.

Denote by p_i the probability of a chromosome to become a parent. Remind that each chromosome may become a parent multiple times. Probability $p_i\left(f_i\right)$ is a function of fitness f_i. Usually this function is non-decreasing so more fitted chromosomes have higher probability of becoming a parent. There is also an intermediate value w_i called weight such that:

$$p_i = \frac{w_i}{\sum\limits_{j=1}^{n_{population}} w_j}$$

Therefore all weights w_i are proportional to corresponding probabilities p_i by the same factor (sum of weights).

Argument `method` determines particular mating algorithm to be applied. Denote by τ the vector of parameters used by the algorithm. Note that τ corresponds to `par`. The algorithm determines a particular form of the $w_i\left(f_i\right)$ function which in turn determines $p_i\left(f_i\right)$.

If `method = "constant"` then all weights and probabilities are equal:

$$w_i = 1 => p_i = \frac{1}{n_{population}}$$

If `method = "rank"` then each chromosome receives a rank r_i based on the fitness f_i value. So if `j`-th chromosome is the fittest one and `k`-th chromosome has the lowest fitness value then $r_j = n_{population}$ and $r_k = 1$. The relationship between weight w_i and rank r_i is as follows:

$$w_i = \left(\frac{r_i}{n_{population}}\right)^{\tau_1}$$

The greater value of τ_1 the greater portion of probability will be delivered to more fitted chromosomes. Default value is $\tau_1 = 0.5$ so `par = 0.5`.

If `method = "fitness"` then weights are calculated as follows:

$$w_i = \left(f_i - \min\left(f_1, ..., f_{n_{population}}\right) + \tau_1\right)^{\tau_2}$$

By default $\tau_1 = 10$ and $\tau_2 = 0.5$ i.e. `par = c(10, 0.5)`. There is a restriction $\tau_1 \geq 0$ insuring that expression in brackets is non-negative.

If `method = "tournament"` then τ_1 (i.e. `par`) chromosomes will be randomly selected with equal probabilities and without replacement. Then the chromosome with the highest fitness (among these selected chromosomes) value will become a parent. It is possible to provide representation of this algorithm via probabilities p_i but the formulas are numerically unstable. By default `par = min(5, ceiling(parents.n * 0.1))`.

Validation and default values assignment for `par` is performed inside `gena` function not in `gena.mating`. It allows to perform validation a single time instead of repeating it each iteration of genetic algorithm.

For more information on mating (selection) algorithms please see Shukla et. al. (2015).

Value

The function returns a list with the following elements:

- `parents` - matrix which rows are parents. The number of rows of this matrix equals to `parents.n` while the number of columns is `ncol(population)`.
- `fitness` - vector which i-th element is the fitness of the i-th parent.
- `ind` - vector which i-th element is the index of i-th parent in population so `$parents[i, ]` equals to `population[ind[i], ]`.

References

A. Shukla, H. Pandey, D. Mehrotra (2015). Comparative review of selection techniques in genetic algorithm. *2015 International Conference on Futuristic Trends on Computational Analysis and Knowledge Management (ABLAZE)*, 515-519, <doi:10.1109/ABLAZE.2015.7154916>.

Examples

```
# Consider the following fitness function
fn <- function(x)
{
  val <- x[1] * x[2] - x[1] ^ 2 - x[2] ^ 2
}

# Randomly initialize the population
set.seed(123)
pop.nulation <- 10
population <- gena.population(pop.n = pop.nulation,
                              lower = c(-5, -5),
                              upper = c(5, 5))

# Calculate fitness of each chromosome
fitness <- rep(NA, pop.nulation)
for(i in 1:pop.nulation)
{
  fitness[i] <- fn(population[i, ])
}

# Perform mating to select parents
parents <- gena.mating(population = population,
                       fitness = fitness,
                       parents.n = pop.nulation,
                       method = "rank",
                       par = 0.8)
print(parents)
```

gena.mutation *Mutation*

Description

Mutation method (algorithm) to be used in the genetic algorithm.

Usage

```
gena.mutation(
  children,
  lower,
  upper,
  prob = 0.2,
  prob.genes = 1/nrow(children),
  method = "constant",
  par = 1,
  iter = NULL
)
```

Arguments

`children`	numeric matrix which rows are children i.e. vectors of parameters values.
`lower`	lower bound of the search space.
`upper`	upper bound of the search space.
`prob`	probability of mutation for a child.
`prob.genes`	numeric vector or numeric value representing the probability of mutation of a child's gene. See 'Details'.
`method`	mutation method to be used for transforming genes of children.
`par`	additional parameters to be passed depending on the `method`.
`iter`	iteration number of the genetic algorithm.

Details

Denote `children` by C^{child} which `i`-th row `children[i, ]` is a chromosome c_i^{child} i.e. the vector of parameter values of the function being optimized $f(.)$ that is provided via `fn` argument of `gena`. The elements of chromosome c_{ij}^{child} are genes representing parameters values.

Mutation algorithm determines random transformation of children's genes. Each child may be selected for mutation with probability `prob`. If i-th child is selected for mutation and `prob.genes` is a vector then j-th gene of this child is transformed with probability `prob.genes[j]`. If `prob.genes` is a constant then this probability is the same for all genes.

Argument `method` determines particular mutation algorithm to be applied. Denote by τ the vector of parameters used by the algorithm. Note that τ corresponds to `par`. Also let's denote by c_{ij}^{mutant} the value of gene c_{ij}^{child} after mutation.

If `method = "constant"` then c_{ij}^{mutant} is a uniform random variable between `lower[j]` and `upper[j]`.

If `method = "normal"` then c_{ij}^{mutant} equals to the sum of c_{ij}^{child} and normal random variable with zero mean and standard deviation `par[j]`. By default `par` is identity vector of length `ncol(children)` so `par[j] = 1` for all `j`.

If `method = "percent"` then c_{ij}^{mutant} is generated from c_{ij}^{child} by equiprobably increasing or decreasing it by q percent, where q is a uniform random variable between 0 and `par[j]`. Note that `par` may also be a constant then all genes have the same maximum possible percentage change. By default `par = 20`.

For more information on mutation algorithms please see Patil, Bhende (2014).

Value

The function returns a matrix which rows are children (after mutation has been applied to some of them).

References

S. Patil, M. Bhende. (2014). Comparison and Analysis of Different Mutation Strategies to improve the Performance of Genetic Algorithm. *International Journal of Computer Science and Information Technologies*, 5 (3), 4669-4673.

Examples

```
# Randomly initialize some children
set.seed(123)
children.n <- 10
children <- gena.population(pop.n = children.n,
                            lower = c(-5, -5),
                            upper = c(5, 5))

# Perform the mutation
mutants <- gena.mutation(children = children,
                         prob = 0.6,
                         prob.genes = c(0.7, 0.8),
                         par = 30,
                         method = "percent")
print(mutants)
```

`gena.population` *Population*

Description

Initialize the population of chromosomes.

Usage

```
gena.population(pop.n, lower, upper, pop.initial = NULL, method = "uniform")
```

Arguments

`pop.n`	positive integer representing the number of chromosomes in population.
`lower`	numeric vector which i-th element determines the minimum possible value for i-th gene.
`upper`	numeric vector which i-th element determines the maximum possible value for i-th gene.
`pop.initial`	numeric matrix which rows are initial chromosomes suggested by user.
`method`	string representing the initialization method to be used. For a list of possible values see Details.

Details

If `"method = uniform"` then i-th gene of each chromosome is randomly (uniformly) chosen between `lower[i]` and `upper[i]` bounds. If `"method = normal"` then i-th gene is generated from a truncated normal distribution with mean `(upper[i] + lower[i]) / 2` and standard deviation `(upper[i] - lower[i]) / 6` where `lower[i]` and `upper[i]` are lower and upper truncation bounds correspondingly. If `"method = hypersphere"` then population is simulated uniformly from the hypersphere with center `upper - lower` and radius `sqrt(sum((upper - lower) ^ 2))` via `rhypersphere` function setting `type = "inside"`.

Value

This function returns a matrix which rows are chromosomes.

References

B. Kazimipour, X. Li, A. Qin (2014). A review of population initialization techniques for evolutionary algorithms. *2014 IEEE Congress on Evolutionary Computation*, 2585-2592, <doi:10.1109/CEC.2014.6900618>.

Examples

```
set.seed(123)
gena.population(pop.n = 10,
                lower = c(-1, -2, -3),
                upper = c(1, 0, -1),
                pop.initial = rbind(c(0, -1, -2),
                                    c(0.1, -1.2, -2.3)),
                method = "normal")
```

`genaDiff` *Numeric Differentiation*

Description

Numeric estimation of the gradient and Hessian.

Usage

```
gena.grad(
  fn,
  par,
  eps = sqrt(.Machine$double.eps) * abs(par),
  method = "central-difference",
  fn.args = NULL
)

gena.hessian(
  fn = NULL,
  gr = NULL,
  par,
  eps = sqrt(.Machine$double.eps) * abs(par),
  fn.args = NULL,
  gr.args = NULL
)
```

Arguments

`fn`	function for which gradient or Hessian should be calculated.
`par`	point (parameters' value) at which `fn` should be differentiated.
`eps`	numeric vector representing increment of the `par`. So `eps[i]` represents increment of `par[i]`. If `eps` is a constant then all increments are the same.
`method`	numeric differentiation method: "central-difference" or "forward-difference".
`fn.args`	list containing arguments of `fn` except `par`.
`gr`	gradient function of `fn`.
`gr.args`	list containing arguments of `gr` except `par`.

Details

It is possible to substantially improve numeric Hessian accuracy by using analytical gradient `gr`. If both `fn` and `gr` are provided then only `gr` will be used. If only `fn` is provided for `gena.hessian` then eps will be transformed to `sqrt(eps)` for numeric stability purposes.

Value

Function `gena.grad` returns a vector that is a gradient of `fn` at point `par` calculated via `method` numeric differentiation approach using increment `eps`.

Function `gena.hessian` returns a matrix that is a Hessian of `fn` at point `par`.

Examples

```
# Consider the following function
fn <- function(par, a = 1, b = 2)
{
  val <- par[1] * par[2] - a * par[1] ^ 2 - b * par[2] ^ 2
}

# Calculate the gradient at point (2, 5) respect to 'par'
# when 'a = 1' and 'b = 1'
par <- c(2, 5)
fn.args = list(a = 1, b = 1)
gena.grad(fn = fn, par = par, fn.args = fn.args)

# Calculate Hessian at the same point
gena.hessian(fn = fn, par = par, fn.args = fn.args)

# Repeat calculation of the Hessian using analytical gradient
gr <- function(par, a = 1, b = 2)
{
  val <- c(par[2] - 2 * a * par[1],
           par[1] - 2 * b * par[2])
}
gena.hessian(gr = gr, par = par, gr.args = fn.args)
```

plot.gena — *Plot best found fitnesses during genetic algorithm*

Description

Plot best found fitnesses during genetic algorithm

Usage

```
## S3 method for class 'gena'
plot(x, y = NULL, ...)
```

Arguments

x	Object of class "gena"
y	this parameter currently ignored
...	further arguments (currently ignored)

Value

This function does not return anything.

`plot.pso` *Plot best found fitnesses during genetic algorithm*

Description

Plot best found fitnesses during genetic algorithm

Usage

```
## S3 method for class 'pso'
plot(x, y = NULL, ...)
```

Arguments

`x`	Object of class "pso"
`y`	this parameter currently ignored
`...`	further arguments (currently ignored)

Value

This function does not return anything.

`print.gena` *Print method for "gena" object*

Description

Print method for "gena" object

Usage

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

Arguments

`x`	Object of class "gena"
`...`	further arguments (currently ignored)

Value

This function does not return anything.

`print.pso` *Print method for "pso" object*

Description

Print method for "pso" object

Usage

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

Arguments

`x`	Object of class "pso"
`...`	further arguments (currently ignored)

Value

This function does not return anything.

`print.summary.gena` *Summary for "gena" object*

Description

Summary for "gena" object

Usage

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

Arguments

`x`	Object of class "gena"
`...`	further arguments (currently ignored)

Value

This function returns `x` input argument.

print.summary.pso *Summary for "pso" object*

Description

Summary for "pso" object

Usage

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

Arguments

x	Object of class "pso"
...	further arguments (currently ignored)

Value

This function returns x input argument.

pso *Particle Swarm Optimization*

Description

This function allows to use particle swarm algorithm for numeric global optimization of real-valued functions.

Usage

```
pso(
  fn,
  gr = NULL,
  lower,
  upper,
  pop.n = 40,
  pop.initial = NULL,
  pop.method = "uniform",
  nh.method = "random",
  nh.par = 3,
  nh.adaptive = TRUE,
  velocity.method = "hypersphere",
  velocity.par = list(w = 1/(2 * log(2)), c1 = 0.5 + log(2), c2 = 0.5 + log(2)),
  hybrid.method = "rank",
  hybrid.par = 2,
  hybrid.prob = 0,
  hybrid.opt.par = NULL,
  hybrid.n = 1,
  constr.method = NULL,
  constr.par = NULL,
  random.order = TRUE,
  maxiter = 100,
  is.max = TRUE,
  info = TRUE,
  ...
)
```

Arguments

`fn`	function to be maximized i.e. fitness function.
`gr`	gradient of the `fn`.
`lower`	lower bound of the search space.
`upper`	upper bound of the search space.
`pop.n`	integer representing the size of the population.
`pop.initial`	numeric matrix which rows are particles to be included into the initial population. Numeric vector will be coerced to single row matrix.
`pop.method`	the algorithm to be applied for a creation of the initial population. See 'Details' for additional information.
`nh.method`	string representing the method (topology) to be used for the creation of neighbourhoods. See 'Details' for additional information.
`nh.par`	parameters of the topology algorithm.
`nh.adaptive`	logical; if `TRUE` (default) then neighbourhoods change every time when the best known (to the swarm) fitnesses value have not increased. Neighbourhoods are updated according to the topology defined via `nh.method` argument.
`velocity.method`	string representing the method to be used for the update of velocities.
`velocity.par`	parameters of the velocity formula.
`hybrid.method`	hybrids selection algorithm i.e. mechanism determining which particles should be subject to local optimization. See 'Details' for additional information.
`hybrid.par`	parameters of the hybridization algorithm.
`hybrid.prob`	probability of generating the hybrids each iteration.
`hybrid.opt.par`	parameters of the local optimization function to be used for hybridization algorithm (including `fn` and `gr`).
`hybrid.n`	number of hybrids that appear if hybridization should take place during the iteration.
`constr.method`	the algorithm to be applied for imposing constraints on the particles. See 'Details' for additional information.

`constr.par`	parameters of the constraint algorithm.
`random.order`	logical; if `TRUE` (default) then particles related routine will be implemented in a random order.
`maxiter`	maximum number of iterations of the algorithm.
`is.max`	logical; if `TRUE` (default) then fitness function will be maximized. Otherwise it will be minimized.
`info`	logical; if `TRUE` (default) then some optimization related information will be printed each iteration.
`...`	additional parameters to be passed to `fn` and `gr` functions.

Details

Default arguments have been set in accordance with SPSO 2011 algorithm proposed by M. Clerc (2012).

To find information on particular methods available via `pop.method`, `nh.method`, `velocity.method`, `hybrid.method` and `constr.method` arguments please see 'Details' section of `gena.population`, `pso.nh`, `pso.velocity`, `gena.hybrid` and `gena.constr` correspondingly.

It is possible to provide manually implemented functions for population initialization, neighbourhoods creation, velocity updated, hybridization and constraints in a similar way as for `gena`.

By default function does not impose any constraints upon the parameters. If `constr.method = "bounds"` then `lower` and `upper` constraints will be imposed. Lower bounds should be strictly smaller then upper bounds.

Currently the only available termination condition is `maxiter`. We are going to provide some additional termination conditions during future updates.

Infinite values in `lower` and `upper` are substituted with `-(.Machine$double.xmax * 0.9)` and `.Machine$double.xmax * 0.9` correspondingly.

By default if `gr` is provided then BFGS algorithm will be used inside `optim` during hybridization. Otherwise `Nelder-Mead` will be used. Manual values for `optim` arguments may be provided (as a list) through `hybrid.opt.par` argument.

For more information on particle swarm optimization please see M. Clerc (2012).

Value

This function returns an object of class `pso` that is a list containing the following elements:

- `par` - particle (solution) with the highest fitness (objective function) value.
- `value` - value of `fn` at `par`.
- `population` - matrix of particles (solutions) of the last iteration of the algorithm.
- `counts` - a two-element integer vector giving the number of calls to `fn` and `gr` respectively.
- `is.max` - identical to `is.max` input argument.
- `fitness.history` - vector which i-th element is fitness of the best particle in i-th iteration.
- `iter` - last iteration number.

References

M. Clerc (2012). Standard Particle Swarm Optimisation. *HAL archieve*.

Examples

```
## Consider Ackley function

fn <- function(par, a = 20, b = 0.2)
{
  val <- a * exp(-b * sqrt(0.5 * (par[1] ^ 2 + par[2] ^ 2))) +
         exp(0.5 * (cos(2 * pi * par[1]) + cos(2 * pi * par[2]))) -
         exp(1) - a
  return(val)
}

# Maximize this function using particle swarm algorithm

set.seed(123)
lower <- c(-5, -100)
upper <- c(100, 5)
opt <- pso(fn = fn,
           lower = lower, upper = upper,
           a = 20, b = 0.2)
print(opt$par)


## Consider Bukin function number 6

fn <- function(x, a = 20, b = 0.2)
{
  val <- 100 * sqrt(abs(x[2] - 0.01 * x[1] ^ 2)) + 0.01 * abs(x[1] + 10)
  return(val)
}

# Minimize this function using initially provided
# position for one of the particles
set.seed(777)
lower <- c(-15, -3)
upper <- c(-5, 3)
opt <- pso(fn = fn,
           pop.init = c(8, 2),
           lower = lower, upper = upper,
           is.max = FALSE)
print(opt$par)
```

pso.nh *Neighbourhood*

Description

Constructs a neighbourhood of each particle using particular topology.

Usage

```
pso.nh(pop.n = 40, method = "ring", par = 3, iter = 1)
```

Arguments

pop.n	integer representing the size of the population.
method	string representing the topology to be used for construction of the neighbourhood. See 'Details' for additional information.
par	additional parameters to be passed depending on the method.
iter	iteration number of the genetic algorithm.

Details

If method = "ring" then each particle will have par[1] neighbours. By default par[1] = 3. See section 3.2.1 of M. Clerc (2012) for additional details. If method = "wheel" then there is a single (randomly selected) particle which informs (and informed by) other particles while there is no direct communication between other particles. If method = "random" then each particle randomly informs other par[1] particles and itself. Note that duplicates are possible so sometimes each particle may inform less then par[1] particles. By default par[1] = 3. See section 3.2.2 of M. Clerc (2012) for more details. If method = "star" then all particles are fully informed by each other. If method = "random2" then each particle will be self-informed and informed by the j-th particle with probability par[1] (value between 0 and 1). By default par[1] = 0.1.

Value

This function returns a list which i-th element is a vector of particles' indexes which inform i-th particle i.e. neighbourhood of the i-th particle.

References

Maurice Clerc (2012). Standard Particle Swarm Optimisation. *HAL archieve*.

Examples

```
# Prepare random number generator
set.seed(123)

# Ring topology with 5 neighbours
pso.nh(pop.n = 10, method = "ring", par = 5)

# Wheel topology
pso.nh(pop.n = 10, method = "wheel")

# Star topology
pso.nh(pop.n = 10, method = "star")
```

```
# Random topology where each particle
# randomly informs 3 other particles
pso.nh(pop.n = 10, method = "random", par = 3)

# Random2 topology wehere each particle could
# be informed by the other with probability 0.2
pso.nh(pop.n = 10, method = "random2", par = 0.2)
```

pso.velocity *Velocity*

Description

Calculates (updates) velocities of the particles.

Usage

```
pso.velocity(
  population,
  method = "hypersphere",
  par = list(w = 1/(2 * log(2)), c1 = 0.5 + log(2), c2 = 0.5 + log(2)),
  velocity,
  best.pn,
  best.nh,
  best.pn.fitness,
  best.nh.fitness,
  iter = 1
)
```

Arguments

`population`	numeric matrix which rows are particles i.e. vectors of parameters values.
`method`	string representing method to be used for velocities calculation. See 'Details' for additional information.
`par`	additional parameters to be passed depending on the `method`.
`velocity`	matrix which i-th row is a velocity of the i-th particle.
`best.pn`	numeric matrix which i-th row is a best personal position known by the i-th particle.
`best.nh`	numeric matrix which i-th row is a best personal position in a neighbourhood of the i-th particle.
`best.pn.fitness`	numeric vector which i-th row is the value of a fitness function at point `best.pn[i, ]`.

`best.nh.fitness`	numeric vector which i-th row is the value of a fitness function at point `best.nh[i, ]`.
`iter`	iteration number of the genetic algorithm.

Details

If `method = "classic"` then classical velocity formula is used:

$$v_{i,j,(t+1)} = w \times v_{i,j,t} + c_1 \times u_{1,i,j} \times b^{pn}_{i,j,t} + c_2 \times u_{2,i,j} \times b^{nh}_{i,j,t}$$

where $v_{i,j,t}$ is a velocity of the i-th particle respect to the j-th component at time t. Random variables $u_{1,i,j}$ and $u_{2,i,j}$ are i.i.d. respect to all indexes and follow standard uniform distribution $U(0,1)$. Variable $b^{pn}_{i,j,t}$ is j-th component of the best known particle's (personal) position up to time period t. Similarly $b^{nh}_{i,j,t}$ is j-th component of the best of best known particle's position in a neighbourhood of the i-th particle. Hyperparameters w, c_1 and c_2 may be provided via `par` argument as a list with elements `par$w`, `par$c1` and `par$c2` correspondingly.

If `method = "hypersphere"` then rotation invariant formula from sections 3.4.2 and 3.4.3 of M. Clerc (2012) is used with arguments identical to the classical method. To simulate a random variate from the hypersphere function `rhypersphere` is used setting `type = "non-uniform"`.

In accordance with M. Clerc (2012) default values are `par$w = 1/(2 * log(2))`, `par$c1 = 0.5 + log(2)` and `par$c2 = 0.5 + log(2)`.

Value

This function returns a matrix which i-th row represents updated velocity of the i-th particle.

References

Maurice Clerc (2012). Standard Particle Swarm Optimisation. *HAL archieve*.

`rhypersphere` *Hypersphere*

Description

Simulates uniform random variates from the hypersphere.

Usage

```
rhypersphere(n, dim = 2, radius = 1, center = rep(0, dim), type = "boundary")
```

Arguments

`n`	number of observations to simulate.
`dim`	dimensions of hypersphere.
`radius`	radius of hypersphere.
`center`	center of hypersphere.
`type`	character; if `"boundary"` (default) then random variates are simulated from the hypersphere. If `"inside"` random variates are points lying inside the hypersphere. If `"non-uniform"` then random variates are non-uniform and simulated from the inner part of the hypersphere simply by making radius a uniform random variable between `0` and `radius`.

Value

The function returns a vector of random variates.

Examples

```
set.seed(123)
# Get 5 random uniform variates from 3D hypersphere
# of radius 10 centered at (2, 3, 1)
rhypersphere(n = 5, dim = 3, radius = 10, center = c(2, 3, 1))
```

`summary.gena` *Summarizing gena Fits*

Description

Summarizing gena Fits

Usage

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

Arguments

`object`	Object of class "gena"
`...`	further arguments (currently ignored)

Value

This function returns the same list as `gena` function changing its class to "summary.gena".

`summary.pso` *Summarizing pso Fits*

Description

Summarizing pso Fits

Usage

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

Arguments

`object`	Object of class "pso"
`...`	further arguments (currently ignored)

Value

This function returns the same list as `pso` function changing its class to "summary.pso".

Index