Package 'metaheuristicOpt'

May 8, 2026

Type Package

Title Metaheuristic for Optimization

Version 2.0.0

Author Lala Septem Riza [aut, cre],
Iip [aut],
Eddy Prasetyo Nugroho [aut],
Muhammad Bima Adi Prabowo [aut],
Enjun Junaeti [aut],
Ade Gafar Abdullah [aut]

Maintainer Lala Septem Riza <lala.s.riza@upi.edu>

Description An implementation of metaheuristic algorithms for continuous optimization. Currently, the package contains the implementations of 21 algorithms, as follows: particle swarm optimization (Kennedy and Eberhart, 1995), ant lion optimizer (Mirjalili, 2015 <doi:10.1016/j.advengsoft.2015.01.010>), grey wolf optimizer (Mirjalili et al., 2014 <doi:10.1016/j.advengsoft.2013.12.007>), dragonfly algorithm (Mirjalili, 2015 <doi:10.1007/s00521-015-1920-1>), firefly algorithm (Yang, 2009 <doi:10.1007/978-3-642-04944-6_14>), genetic algorithm (Holland, 1992, ISBN:978-0262581110), grasshopper optimisation algorithm (Saremi et al., 2017 <doi:10.1016/j.advengsoft.2017.01.004>), harmony search algorithm (Mahdavi et al., 2007 <doi:10.1016/j.amc.2006.11.033>), moth flame optimizer (Mirjalili, 2015 <doi:10.1016/j.knosys.2015.07.006>, sine cosine algorithm (Mirjalili, 2016 <doi:10.1016/j.knosys.2015.12.022>), whale optimization algorithm (Mirjalili and Lewis, 2016 <doi:10.1016/j.advengsoft.2016.01.008>), clonal selection algorithm (Castro, 2002 <doi:10.1109/TEVC.2002.1011539>), differential evolution (Das & Suganthan, 2011), shuffled frog leaping (Eusuff, Landsey & Pasha, 2006), cat swarm optimization (Chu et al., 2006), artificial bee colony algorithm (Karaboga & Akay, 2009), krill-herd algorithm (Gandomi & Alavi, 2012), cuckoo search (Yang & Deb, 2009), bat algorithm (Yang, 2012), gravitational based search (Rashedi et al., 2009) and black hole optimization (Hatamlou, 2013).

Depends R (>= 3.5.0)

License GPL (>= 2) | file LICENSE

Encoding UTF-8

LazyData true

RoxygenNote 6.1.1

NeedsCompilation no

Repository CRAN

Date/Publication 2019-06-19 12:10:10 UTC

Contents

ABC *Optimization using Artificial Bee Colony Algorithm*

Description

This is the internal function that implements Artificial Bee Colony Algorithm. It is used to solve continuous optimization tasks. Users do not need to call it directly, but just use `metaOpt`.

Usage

```
ABC(FUN, optimType = "MIN", numVar, numPopulation = 40,
  maxIter = 500, rangeVar, cycleLimit = as.integer(numVar *
  numPopulation))
```

Arguments

FUN	an objective function or cost function,
optimType	a string value that represent the type of optimization. There are two option for this arguments: "MIN" and "MAX". The default value is "MIN", which the function will do minimization. Otherwise, you can use "MAX" for maximization problem. The default value is "MIN".
numVar	a positive integer to determine the number variables.
numPopulation	a positive integer to determine the number populations. The default value is 40.
maxIter	a positive integer to determine the maximum number of iterations. The default value is 500.
rangeVar	a matrix $(2 \times n)$ containing the range of variables, where n is the number of variables, and first and second rows are the lower bound (minimum) and upper bound (maximum) values, respectively. If all variable have equal upper bound, you can define rangeVar as matrix (2×1).
cycleLimit	a positive integer to determine number of times allowed for candidate solution to not move. The default value is as.integer(numVar * numPopulation).

Details

This algorithm was proposed by (Karaboga & Akay, 2009). It inspired by type of bee. They are three types of bee employeed, onlooker and scout. Employed bee work by finding food source. Onlooker bee work by finding better food source other than foods that Employed bee found. Scout bee work by removing abandoned food source. Each candidate solution in ABC algorithm represent as bee and they will move in 3 phases employed, onlooker and scout.

In order to find the optimal solution, the algorithm follow the following steps.

- initialize population randomly.
- Employed bee phase (Perform local search and greedy algorithm for each candidate solution).
- Onlooker bee phase (Perform local search and greedy algorithm for some candidate solutions).
- Scout bee phase (Remove abandoned candidate solutions).
- If a termination criterion (a maximum number of iterations or a sufficiently good fitness) is met, exit the loop, else back to employed bee phase.

Value

Vector [v1, v2, ..., vn] where n is number variable and vn is value of n-th variable.

References

Karaboga, D., & Akay, B. (2009). A comparative study of artificial bee colony algorithm. Applied mathematics and computation, 214(1), 108-132.

See Also

metaOpt

Examples

```
#################################
## Optimizing the sphere function

# define sphere function as objective function
sphere <- function(x){
    return(sum(x^2))
}

## Define parameter
numVar <- 5
rangeVar <- matrix(c(-10,10), nrow=2)

## calculate the optimum solution using artificial bee colony algorithm
resultABC <- ABC(sphere, optimType="MIN", numVar, numPopulation=20,
                 maxIter=100, rangeVar)

## calculate the optimum value using sphere function
optimum.value <- sphere(resultABC)
```

ALO *Optimization using Ant Lion Optimizer*

Description

This is the internal function that implements Ant Lion Optimizer Algorithm. It is used to solve continuous optimization tasks. Users do not need to call it directly, but just use `metaOpt`.

Usage

```
ALO(FUN, optimType = "MIN", numVar, numPopulation = 40,
  maxIter = 500, rangeVar)
```

Arguments

`FUN`	an objective function or cost function,
`optimType`	a string value that represent the type of optimization. There are two option for this arguments: `"MIN"` and `"MAX"`. The default value is `"MIN"`, which the function will do minimization. Otherwise, you can use `"MAX"` for maximization problem. The default value is `"MIN"`.
`numVar`	a positive integer to determine the number variables.
`numPopulation`	a positive integer to determine the number populations. The default value is 40.
`maxIter`	a positive integer to determine the maximum number of iterations. The default value is 500.
`rangeVar`	a matrix $(2 \times n)$ containing the range of variables, where n is the number of variables, and first and second rows are the lower bound (minimum) and upper bound (maximum) values, respectively. If all variable have equal upper bound, you can define `rangeVar` as matrix (2×1).

Details

This algorithm was proposed by (Mirjalili, 2015). The Ant Lion Optimizer (ALO) algorithm mimics the hunting mechanism of antlions in nature. Five main steps of hunting prey such as the random walk of ants, building traps, entrapment of ants in traps, catching preys, and re-building traps are implemented.

In order to find the optimal solution, the algorithm follow the following steps.

- Initialization: Initialize the first population of ants and antlions randomly, calculate the fitness of ants and antlions and find the best antlions as the elite (determined optimum).
- Update Ants Position: Select an antlion using Roulette Whell then update ants position based on random walk around selected antlion and elite. Furthermore, calculate the fitness of all ants.
- Replace an antlion with its corresponding ant, if it becomes fitter
- Update elite if an antlion becomes fitter than the elite
- Check termination criteria, if termination criterion is satisfied, return the elite as the optimal solution for given problem. Otherwise, back to Update Ants Position steps.

Value

`Vector [v1, v2, ..., vn]` where `n` is number variable and `vn` is value of `n-th` variable.

References

Seyedali Mirjalili, The Ant Lion Optimizer, Advances in Engineering Software, Volume 83, 2015, Pages 80-98, ISSN 0965-9978, https://doi.org/10.1016/j.advengsoft.2015.01.010

See Also

`metaOpt`

Examples

```
###################################
## Optimizing the schewefel's problem 2.22 function

# define schewefel's problem 2.22 function as objective function
schewefels2.22 <- function(x){
   return(sum(abs(x)+prod(abs(x))))
}

## Define parameter
numVar <- 5
rangeVar <- matrix(c(-10,10), nrow=2)

## calculate the optimum solution using Ant Lion Optimizer
resultALO <- ALO(schewefels2.22, optimType="MIN", numVar, numPopulation=20,
                 maxIter=100, rangeVar)

## calculate the optimum value using schewefel's problem 2.22 function
```

```
optimum.value <- schewefels2.22(resultALO)
```

BA *Optimization using Bat Algorithm*

Description

This is the internal function that implements Bat Algorithm. It is used to solve continuous optimization tasks. Users do not need to call it directly, but just use `metaOpt`.

Usage

```
BA(FUN, optimType = "MIN", numVar, numPopulation = 40, maxIter = 500,
  rangeVar, maxFrequency = 0.1, minFrequency = -0.1, gama = 1,
  alphaBA = 0.1)
```

Arguments

`FUN`	an objective function or cost function,
`optimType`	a string value that represent the type of optimization. There are two option for this arguments: `"MIN"` and `"MAX"`. The default value is `"MIN"`, which the function will do minimization. Otherwise, you can use `"MAX"` for maximization problem. The default value is `"MIN"`.
`numVar`	a positive integer to determine the number variables.
`numPopulation`	a positive integer to determine the number populations. The default value is 40.
`maxIter`	a positive integer to determine the maximum number of iterations. The default value is 500.
`rangeVar`	a matrix $(2 \times n)$ containing the range of variables, where n is the number of variables, and first and second rows are the lower bound (minimum) and upper bound (maximum) values, respectively. If all variable have equal upper bound, you can define `rangeVar` as matrix (2×1).
`maxFrequency`	a numeric to determine maximum frequency. The default value is 0.1.
`minFrequency`	a numeric to determine minimum frequency. The default value is -0.1.
`gama`	a numeric greater than equal to 1. It use to increase pulse rate. The default value is 1.
`alphaBA`	a numeric between 0 and 1. It use to decrease loudness. The default value is 0.1.

Details

This algorithm was proposed by (Yang, 2011). It was inspired by echolocation of bats. Candidate solutions in bat algorithm are represented by bat. They have flying speed, pulse rate, loudness and pulse frequency and they move based on them.

In order to find the optimal solution, the algorithm follow the following steps.

- initialize population randomly.
- move every candidate solutions based on velocity and pulse frequnecy.
- move some candidate solutions near globak best randomly.
- If a candidate solution have better fitness than global best replace it with new random candidate solution then increase its pulse rate and decrease its loudness.
- If a termination criterion (a maximum number of iterations or a sufficiently good fitness) is met, exit the loop, else back to move every candidate solutions.

Value

`Vector [v1, v2, ..., vn]` where `n` is number variable and `vn` is value of `n-th` variable.

References

Yang, X. S., (2011), Bat Algorithm for Multiobjective Optimization, Int. J. Bio-Inspired Computation, Vol. 3, No. 5, pp.267-274.

See Also

metaOpt

Examples

```
###################################
## Optimizing the schewefel's problem 1.2 function

# define schewefel's problem 1.2 function as objective function
schewefels1.2 <- function(x){
  dim <- length(x)
  result <- 0
    for(i in 1:dim){
       result <- result + sum(x[1:i])^2
   }
  return(result)
}

## Define parameter
numVar <- 5
rangeVar <- matrix(c(-10,10), nrow=2)

## calculate the optimum solution using bat algorithm
resultBA <- BA(schewefels1.2, optimType="MIN", numVar, numPopulation=20,
                maxIter=100, rangeVar)

## calculate the optimum value using schewefel's problem 1.2 function
optimum.value <- schewefels1.2(resultBA)
```

BHO *Optimization using Black Hole Optimization Algorithm*

Description

This is the internal function that implements Black-Hole based Optimization Algorithm. It is used to solve continuous optimization tasks. Users do not need to call it directly, but just use `metaOpt`.

Usage

```
BHO(FUN, optimType = "MIN", numVar, numPopulation = 40,
  maxIter = 500, rangeVar)
```

Arguments

`FUN`	an objective function or cost function,
`optimType`	a string value that represent the type of optimization. There are two option for this arguments: `"MIN"` and `"MAX"`. The default value is `"MIN"`, which the function will do minimization. Otherwise, you can use `"MAX"` for maximization problem. The default value is `"MIN"`.
`numVar`	a positive integer to determine the number variables.
`numPopulation`	a positive integer to determine the number populations. The default value is 40.
`maxIter`	a positive integer to determine the maximum number of iterations. The default value is 500.
`rangeVar`	a matrix $(2 \times n)$ containing the range of variables, where n is the number of variables, and first and second rows are the lower bound (minimum) and upper bound (maximum) values, respectively. If all variable have equal upper bound, you can define `rangeVar` as matrix (2×1).

Details

This algorithm was proposed by (Hatamlou, 2013). The main inspiration for BHO algorithm originates from black hole that swallow all nearest star. Black hole represent candidate solution with best fitness and other candidate solutions as star, so all star search new best candidate solution while moving towards black-hole. if star reaches better fitness than black hole, exchange its position. star that too close to black hole (pass event horizon) wiil be replace by new random candidate solution.

In order to find the optimal solution, the algorithm follow the following steps.

- initialize population randomly.
- select best candidate solution as black hole other as stars.
- change each star location to moving toward black hole.
- If a star reaches a location with lower cost than the black hole, exchange their locations.
- If a star crosses the event horizon of the black hole, replace it with a new star in a random location in the search space.
- If a termination criterion (a maximum number of iterations or a sufficiently good fitness) is met, exit the loop.

Value

Vector [v1, v2, ..., vn] where n is number variable and vn is value of n-th variable.

References

Hatamlou, A. (2013). Black hole: A new heuristic optimization approach for data clustering. Information Sciences, 222(December), 175–184. https://doi.org/10.1016/j.ins.2012.08.023

See Also

metaOpt

Examples

```
##################################
## Optimizing the step function

# define step function as objective function
step <- function(x){
    result <- sum(abs((x+0.5))^2)
    return(result)
}

## Define parameter
numVar <- 5
rangeVar <- matrix(c(-100,100), nrow=2)

## calculate the optimum solution using black hole optimization
resultBHO <- BHO(step, optimType="MIN", numVar, numPopulation=20,
                 maxIter=100, rangeVar)

## calculate the optimum value using step function
optimum.value <- step(resultBHO)
```

CLONALG *Optimization using Clonal Selection Algorithm*

Description

This is the internal function that implements Clonal Selection Algorithm. It is used to solve continuous optimization tasks. Users do not need to call it directly, but just use metaOpt.

Usage

```
CLONALG(FUN, optimType = "MIN", numVar, numPopulation = 40,
  maxIter = 500, rangeVar, selectionSize = as.integer(numPopulation/4),
  multipicationFactor = 0.5, hypermutationRate = 0.1)
```

Arguments

FUN	an objective function or cost function,
optimType	a string value that represent the type of optimization. There are two option for this arguments: "MIN" and "MAX". The default value is "MIN", which the function will do minimization. Otherwise, you can use "MAX" for maximization problem. The default value is "MIN".
numVar	a positive integer to determine the number variables.
numPopulation	a positive integer to determine the number populations. The default value is 40.
maxIter	a positive integer to determine the maximum number of iterations. The default value is 500.
rangeVar	a matrix $(2 \times n)$ containing the range of variables, where n is the number of variables, and first and second rows are the lower bound (minimum) and upper bound (maximum) values, respectively. If all variable have equal upper bound, you can define rangeVar as matrix (2×1).
selectionSize	a positive integer between 0 and numVar to determine selection size (see details). The default value is as.integer(numPopulation/4).
multipicationFactor	a positive numeric between 0 and 1 to determine number of clones. The default value is 0.5.
hypermutationRate	a positive numeric between 0 and 1 to determine probabilty of variable in clone candidate solutions to be mutated, close to 1 probability is high and vice versa. The default value is 0.1.

Details

This algorithm was proposed by (Castro & Zuben, 2002). The Clonal Selection Algorithm (CLONALG) mimics maturation proses of imumune system. CLONALG consist 5 step initialize, selection, clonal, hypermutation and maturation.

In order to find the optimal solution, the algorithm follow the following steps.

- initialize population randomly.
- select top selectionSize candidate solutions from population with best fitness.
- clone each selected candidate solutions.
- hypermutation each variable in cloned candidate solutions.
- maturation combine each hypermutated candidate solution with population. Select top n candidate solution from population as new population.
- If a termination criterion (a maximum number of iterations or a sufficiently good fitness) is met, exit the loop.

Value

Vector [v1, v2, ..., vn] where n is number variable and vn is value of n-th variable.

References

Castro, L. & Zuben, F. J. V. (2002). Learning and optimization using the clonal selection principle. IEEE Transactions on Evolutionary Computation, Special Issue on Artificial. Immune Systems, 6(3), 239–251. https://doi.org/10.1109/TEVC.2002.1011539

See Also

metaOpt

Examples

```
###################################
## Optimizing the quartic with noise function

# define Quartic with noise function as objective function
quartic <- function(x){
    dim <- length(x)
    result <- sum(c(1:dim)*(x^4))+runif(1)
    return(result)
}

## Define parameter
numVar <- 5
rangeVar <- matrix(c(-1.28, 1.28), nrow=2)

## calculate the optimum solution clonal selection algorithm
resultCLONALG <- CLONALG(quartic, optimType="MIN", numVar, numPopulation=20,
                 maxIter=100, rangeVar)

## calculate the optimum value using quartic with noise function
optimum.value <- quartic(resultCLONALG)
```

CS *Optimization using Cuckoo Search algorithm*

Description

This is the internal function that implements cuckoo search Algorithm. It is used to solve continuous optimization tasks. Users do not need to call it directly, but just use metaOpt.

Usage

```
CS(FUN, optimType = "MIN", numVar, numPopulation = 40, maxIter = 500,
  rangeVar, abandonedFraction = 0.5)
```

Arguments

FUN	an objective function or cost function,
optimType	a string value that represent the type of optimization. There are two option for this arguments: "MIN" and "MAX". The default value is "MIN", which the function will do minimization. Otherwise, you can use "MAX" for maximization problem. The default value is "MIN".
numVar	a positive integer to determine the number variables.
numPopulation	a positive integer to determine the number populations. The default value is 40.
maxIter	a positive integer to determine the maximum number of iterations. The default value is 500.
rangeVar	a matrix $(2 \times n)$ containing the range of variables, where n is the number of variables, and first and second rows are the lower bound (minimum) and upper bound (maximum) values, respectively. If all variable have equal upper bound, you can define rangeVar as matrix (2×1).
abandonedFraction	a positive numeric between 0 and 1 to determine fraction of population to be replaced. The default value is 0.5.

Details

This algorithm was proposed by (Yang & Deb, 2009). This algorithhm was inspired by behaviour of cuckoo birds which place its egg on other bird nest. While cuckoo birds putting the eggs in the nests of other birds they are two possible outcome. First the owner of the nest will stay on the nest. Second the owner of the nest will abandon the nest.

In order to find the optimal solution, the algorithm follow the following steps.

- initialize population randomly.
- create a mutant vector.
- select a candidate solution in population randomly then compare it with mutant vector. if mutant vector have better fitness than candidate solution replace candidate solution with mutant vector.
- replace fraction of population with worst fitness with new random candidate solutions.
- If a termination criterion (a maximum number of iterations or a sufficiently good fitness) is met, exit the loop, else back to create a mutant vector.

Value

Vector [v1, v2, ..., vn] where n is number variable and vn is value of n-th variable.

References

Yang, X. S., & Deb, S. (2009, December). Cuckoo search via Lévy flights. In 2009 World Congress on Nature & Biologically Inspired Computing (NaBIC) (pp. 210-214). IEEE.

See Also

metaOpt

Examples

```
#################################
## Optimizing the sphere function

# define sphere function as objective function
sphere <- function(x){
    return(sum(x^2))
}

## Define parameter
numVar <- 5
rangeVar <- matrix(c(-10,10), nrow=2)

## calculate the optimum solution cuckoo search
resultCS <- CS(sphere, optimType="MIN", numVar, numPopulation=20,
                maxIter=100, rangeVar)

## calculate the optimum value using sphere function
optimum.value <- sphere(resultCS)
```

CSO *Optimization using Cat Swarm Optimization Algorithm*

Description

This is the internal function that implements Cat Swarm Optimization Algorithm. It is used to solve continuous optimization tasks. Users do not need to call it directly, but just use `metaOpt`.

Usage

```
CSO(FUN, optimType = "MIN", numVar, numPopulation = 40,
  maxIter = 500, rangeVar, mixtureRatio = 0.5, tracingConstant = 0.1,
  maximumVelocity = 1, smp = as.integer(20), srd = 20,
  cdc = as.integer(numVar), spc = TRUE)
```

Arguments

`FUN`	an objective function or cost function,
`optimType`	a string value that represent the type of optimization. There are two option for this arguments: `"MIN"` and `"MAX"`. The default value is `"MIN"`, which the function will do minimization. Otherwise, you can use `"MAX"` for maximization problem. The default value is `"MIN"`.
`numVar`	a positive integer to determine the number variables.
`numPopulation`	a positive integer to determine the number populations. The default value is 40.
`maxIter`	a positive integer to determine the maximum number of iterations. The default value is 500.

`rangeVar` a matrix ($2 \times n$) containing the range of variables, where n is the number of variables, and first and second rows are the lower bound (minimum) and upper bound (maximum) values, respectively. If all variable have equal upper bound, you can define `rangeVar` as matrix (2×1).

`mixtureRatio` a positive numeric between 0 and 1 to determine flaging proportion. higher mixtureRatio increase number of candidate solutions in seeking mode and vice versa. The default value is 0.5.

`tracingConstant` a positive numeric between 0 and 1 to determine tracingConstant. The default value is 0.1.

`maximumVelocity` a positive numeric to determine maximumVelocity while candidate solutions in tracing mode performing local search. The default value is 1.

`smp` a positive integer to determine number of duplication in genetic operator. The default value is `as.integer(20)`.

`srd` a positive numeric between 0 and 100 to determine mutation length in genetic operator. The default value is 20.

`cdc` a positive integer between 0 and numVar to determine number of variabel in candidate solutions in seeking mode to be mutated during mutation step in genetic operator. The default value is `as.integer(numVar)`.

`spc` a logical. if spc is TRUE smp = smp else smp = smp - 1. The default value is TRUE.

Details

This algorithm was proposed by (Chu, Tsai & Pan, 2006). This algorithm was inspired by behaviours of felyne. Behaviours of felyne can be devided into two seeking mode (when flyne rest) and tracing mode (when felyne chase its prey). candidate solutions divided into seeking and tracing mode. candidate solution in seeking mode move using local search while candidate solution in tracing mode move using genetic operator.

In order to find the optimal solution, the algorithm follow the following steps.

- initialize population randomly.
- flaging (tracing or seeking) every candidate solution in population based on mixtureRatio randomly.
- candidate solutions in seeking mode move using local search
- candidate solutions in tracing mode move using genetic operator
- If a termination criterion (a maximum number of iterations or a sufficiently good fitness) is met, exit the loop, else back to flaging candidate solutions.

Value

`Vector [v1, v2, ..., vn]` where `n` is number variable and `vn` is value of `n-th` variable.

References

Chu, S. C., Tsai, P. W., & Pan, J. S. (2006, August). Cat swarm optimization. In Pacific Rim international conference on artificial intelligence (pp. 854-858). Springer, Berlin, Heidelberg.

See Also

metaOpt

Examples

```
##################################
## Optimizing the schewefel's problem 2.22 function

# define schewefel's problem 2.22 function as objective function
schewefels2.22 <- function(x){
   return(sum(abs(x)+prod(abs(x))))
}

## Define parameter
numVar <- 5
rangeVar <- matrix(c(-10,10), nrow=2)

## calculate the optimum solution using Ant Lion Optimizer
resultCSO <- CSO(schewefels2.22, optimType="MIN", numVar, numPopulation=20,
                 maxIter=100, rangeVar)

## calculate the optimum value using schewefel's problem 2.22 function
optimum.value <- schewefels2.22(resultCSO)
```

DA *Optimization using Dragonfly Algorithm*

Description

This is the internal function that implements Dragonfly Algorithm. It is used to solve continuous optimization tasks. Users do not need to call it directly, but just use metaOpt.

Usage

```
DA(FUN, optimType = "MIN", numVar, numPopulation = 40, maxIter = 500,
  rangeVar)
```

Arguments

FUN	an objective function or cost function,
optimType	a string value that represent the type of optimization. There are two option for this arguments: "MIN" and "MAX". The default value is "MIN", which the function will do minimization. Otherwise, you can use "MAX" for maximization problem. The default value is "MIN".
numVar	a positive integer to determine the number variables.
numPopulation	a positive integer to determine the number populations. The default value is 40.

`maxIter` a positive integer to determine the maximum number of iterations. The default value is 500.

`rangeVar` a matrix ($2 \times n$) containing the range of variables, where n is the number of variables, and first and second rows are the lower bound (minimum) and upper bound (maximum) values, respectively. If all variable have equal upper bound, you can define `rangeVar` as matrix (2×1).

Details

This algorithm was proposed by (Mirjalili, 2015). The main inspiration of the DA algorithm originates from the static and dynamic swarming behaviours of dragonflies in nature. Two essential phases of optimization, exploration and exploitation, are designed by modelling the social interaction of dragonflies in navigating, searching for foods, and avoiding enemies when swarming dynamically or statistically.

In order to find the optimal solution, the algorithm follow the following steps.

- Initialization: Initialize the first population of dragonflies randomly, calculate the fitness of dragonflies and find the best dragonfly as food source and the worst dragonfly as enemy position.
- Calculating Behaviour Weight that affecting fly direction and distance. First, find the neighbouring dragonflies for each dragonfly then calculate the behaviour weight. The behaviour weight consist of separation, alignment, cohesion, attracted toward food sources and distraction from enemy. The neighbouring dragonfly determined by the neighbouring radius that increasing linearly for each iteration.
- Update the position each dragonfly using behaviour weight and the delta (same as velocity in PSO).
- Calculate the fitness and update food and enemy position
- Check termination criteria, if termination criterion is satisfied, return the food position as the optimal solution for given problem. Otherwise, back to Calculating Behaviour Weight steps.

Value

`Vector [v1, v2, ..., vn]` where `n` is number variable and `vn` is value of `n-th` variable.

References

Seyedali Mirjalili. 2015. Dragonfly algorithm: a new meta-heuristic optimization technique for solving single-objective, discrete, and multi-objective problems. Neural Comput. Appl. 27, 4 (May 2015), 1053-1073. DOI=https://doi.org/10.1007/s00521-015-1920-1

See Also

`metaOpt`

Examples

```
#################################
## Optimizing the schewefel's problem 1.2 function

# define schewefel's problem 1.2 function as objective function
schewefels1.2 <- function(x){
  dim <- length(x)
  result <- 0
    for(i in 1:dim){
       result <- result + sum(x[1:i])^2
   }
  return(result)
}

## Define parameter
numVar <- 5
rangeVar <- matrix(c(-10,10), nrow=2)

## calculate the optimum solution using dragonfly algorithm
resultDA <- DA(schewefels1.2, optimType="MIN", numVar, numPopulation=20,
                 maxIter=100, rangeVar)

## calculate the optimum value using schewefel's problem 1.2 function
optimum.value <- schewefels1.2(resultDA)
```

DE *Optimization using Differential Evolution Algorithm*

Description

This is the internal function that implements Differential Evolution Algorithm. It is used to solve continuous optimization tasks. Users do not need to call it directly, but just use `metaOpt`.

Usage

```
DE(FUN, optimType = "MIN", numVar, numPopulation = 40, maxIter = 500,
  rangeVar, scalingVector = 0.8, crossOverRate = 0.5,
  strategy = "best 1")
```

Arguments

`FUN`	an objective function or cost function
`optimType`	a string value that represent the type of optimization. There are two option for this arguments: `"MIN"` and `"MAX"`. The default value is `"MIN"`, which the function will do minimization. Otherwise, you can use `"MAX"` for maximization problem. The default value is `"MIN"`.
`numVar`	a positive integer to determine the number variables.

`numPopulation` a positive integer to determine the number populations. The default value is 40.

`maxIter` a positive integer to determine the maximum number of iterations. The default value is 500.

`rangeVar` a matrix $(2 \times n)$ containing the range of variables, where n is the number of variables, and first and second rows are the lower bound (minimum) and upper bound (maximum) values, respectively. If all variable have equal upper bound, you can define `rangeVar` as matrix (2×1).

`scalingVector` a positive numeric between 0 and 1 to determine scalingVector for mutation operator. The default value is 0.8.

`crossOverRate` a positive numeric between 0 and 1 to determine crossOver probability. The default value is 0.5.

`strategy` characters to determine mutation method. They are six methods to choose:

- "classical".
- "best 1"
- "target to best"
- "best 2"
- "rand 2"
- "rand 2 dir"

details of the mutation methods are on the references. The default value is "best 1".

Details

This Differential Evolution algorithm based on jurnal by (Das & Suganthan, 2011). Differential Evolution algorithm use genetic operator for optimization such as mutation, crossover and selection.

In order to find the optimal solution, the algorithm follow the following steps.

- initialize population randomly.
- create some mutation vectors as new candidate solutions (mutation operator).
- perform crossover operator.
- perform selection operator.
- If a termination criterion (a maximum number of iterations or a sufficiently good fitness) is met, exit the loop, else back to create some mutation vector.

Value

`Vector [v1, v2, ..., vn]` where `n` is number variable and `vn` is value of `n-th` variable.

References

Das, S., & Suganthan, P. N. (2011). Differential evolution: A survey of the state-of-the-art. IEEE transactions on evolutionary computation, 15(1), 4-31.

See Also

`metaOpt`

Examples

```
##################################
## Optimizing the step function

# define step function as objective function
step <- function(x){
    result <- sum(abs((x+0.5))^2)
    return(result)
}

## Define parameter
numVar <- 5
rangeVar <- matrix(c(-100,100), nrow=2)

## calculate the optimum solution using differential evolution
resultDE <- DE(step, optimType="MIN", numVar, numPopulation=20,
                maxIter=100, rangeVar)

## calculate the optimum value using step function
optimum.value <- step(resultDE)
```

FFA *Optimization using Firefly Algorithm*

Description

This is the internal function that implements Firefly Algorithm. It is used to solve continuous optimization tasks. Users do not need to call it directly, but just use `metaOpt`.

Usage

```
FFA(FUN, optimType = "MIN", numVar, numPopulation = 40,
  maxIter = 500, rangeVar, B0 = 1, gamma = 1, alphaFFA = 0.2)
```

Arguments

`FUN`	an objective function or cost function,
`optimType`	a string value that represent the type of optimization. There are two option for this arguments: `"MIN"` and `"MAX"`. The default value is `"MIN"`, which the function will do minimization. Otherwise, you can use `"MAX"` for maximization problem. The default value is `"MIN"`.
`numVar`	a positive integer to determine the number variables.
`numPopulation`	a positive integer to determine the number populations. The default value is 40.
`maxIter`	a positive integer to determine the maximum number of iterations. The default value is 500.

rangeVar	a matrix ($2 \times n$) containing the range of variables, where n is the number of variables, and first and second rows are the lower bound (minimum) and upper bound (maximum) values, respectively. If all variable have equal upper bound, you can define rangeVar as matrix (2×1).
B0	a positive integer to determine the attractiveness firefly at r=0. The default value is 1.
gamma	a positive integer to determine light absorption coefficient. The default value is 1.
alphaFFA	a positive integer to determine randomization parameter. The default value is 0.2.

Details

This algorithm was proposed by (Yang, 2009). The firefly algorithm (FFA) mimics the behavior of fireflies, which use a kind of flashing light to communicate with other members of their species. Since the intensity of the light of a single firefly diminishes with increasing distance, the FFA is implicitly able to detect local solutions on its way to the best solution for a given objective function.

In order to find the optimal solution, the algorithm follow the following steps.

- Initialization: Initialize the first population of fireflies randomly, calculate the fitness of fireflies and assumes fitness values as Light Intensity.
- Update the firefly position based on the attractiveness. The firefly that have higher light intensity will tend to attract other fireflies. The attracted firefly will move based on the parameter that given by user.
- Calculate the fitness and update the best firefly position.
- Check termination criteria, if termination criterion is satisfied, return the best position as the optimal solution for given problem. Otherwise, back to Update firefly position steps.

Value

Vector [v1, v2, ..., vn] where n is number variable and vn is value of n-th variable.

References

X.-S. Yang, Firefly algorithms for multimodal optimization, in: Stochastic Algorithms: Foundations and Applications, SAGA 2009, Lecture Notes in Computer Sciences, Vol. 5792, pp. 169-178 (2009).

See Also

metaOpt

Examples

```
##################################
## Optimizing the quartic with noise function
# define Quartic with noise function as objective function
quartic <- function(x){
```

```
    dim <- length(x)
    result <- sum(c(1:dim)*(x^4))+runif(1)
    return(result)
}

## Define parameter
B0 <- 1
gamma <- 1
alphaFFA <- 0.2
numVar <- 5
rangeVar <- matrix(c(-1.28,1.28), nrow=2)

## calculate the optimum solution using Firefly Algorithm
resultFFA <- FFA(quartic, optimType="MIN", numVar, numPopulation=20,
                 maxIter=100, rangeVar, B0, gamma, alphaFFA)

## calculate the optimum value using sphere function
optimum.value <- quartic(resultFFA)
```

GA *Optimization using Genetic Algorithm*

Description

This is the internal function that implements Genetic Algorithm. It is used to solve continuous optimization tasks. Users do not need to call it directly, but just use `metaOpt`.

Usage

```
GA(FUN, optimType = "MIN", numVar, numPopulation = 40, maxIter = 500,
  rangeVar, Pm = 0.1, Pc = 0.8)
```

Arguments

`FUN`	an objective function or cost function,
`optimType`	a string value that represent the type of optimization. There are two option for this arguments: `"MIN"` and `"MAX"`. The default value is `"MIN"`, which the function will do minimization. Otherwise, you can use `"MAX"` for maximization problem. The default value is `"MIN"`.
`numVar`	a positive integer to determine the number variables.
`numPopulation`	a positive integer to determine the number populations. The default value is 40.
`maxIter`	a positive integer to determine the maximum number of iterations. The default value is 500.
`rangeVar`	a matrix $(2 \times n)$ containing the range of variables, where n is the number of variables, and first and second rows are the lower bound (minimum) and upper bound (maximum) values, respectively. If all variable have equal upper bound, you can define `rangeVar` as matrix (2×1).

Pm	a positive integer to determine mutation probability. The default value is 0.1.
Pc	a positive integer to determine crossover probability. The default value is 0.8.

Details

Genetic algorithms (GA) were invented by John Holland in the 1960 and were developed by Holland and his students and colleagues at the University of Michigan in the 1960 and the 1970. GA are commonly used to generate high-quality solutions to optimization and search problems by relying on bio-inspired operators such as mutation, crossover and selection.

In order to find the optimal solution, the algorithm follow the following steps.

- Initialization: Initialize the first population randomly, calculate the fitness and save the best fitness as bestPopulation.
- Selection: Select set of individual parent for doing crossover. Number of parent determined by the crossover probability which defined by user. In this work, we use method called Roulette Whell Selection.
- Crossover: Doing crossover between two parent from Selection step. This step done by selecting two point randomly and switching the values between them.
- Mutation : All individu in population have a chance to mutate. When mutation occurs, we generate the random values to replace the old one.
- Calculate the fitness of each individual and update bestPopulation.
- Check termination criteria, if termination criterion is satisfied, return the bestPopulation as the optimal solution for given problem. Otherwise, back to Selection steps.

Value

`Vector [v1, v2, ..., vn]` where `n` is number variable and `vn` is value of `n-th` variable.

References

Holland, J. H. 1975. Adaptation in Natural and Artificial Systems. University of Michigan Press. (Second edition: MIT Press, 1992.)

Melanie Mitchell. 1998. An Introduction to Genetic Algorithms. MIT Press, Cambridge, MA, USA.

See Also

`metaOpt`

Examples

```
#################################
## Optimizing the sphere function

# define sphere function as objective function
sphere <- function(x){
    return(sum(x^2))
}
```

```
## Define parameter
Pm <- 0.1
Pc <- 0.8
numVar <- 5
rangeVar <- matrix(c(-10,10), nrow=2)

## calculate the optimum solution using Genetic Algorithm
resultGA <- GA(sphere, optimType="MIN", numVar, numPopulation=20,
                 maxIter=100, rangeVar, Pm, Pc)

## calculate the optimum value using sphere function
optimum.value <- sphere(resultGA)
```

GBS *Optimization using Gravitational Based Search Algorithm.*

Description

This is the internal function that implements Gravitational Based Search Algorithm. It is used to solve continuous optimization tasks. Users do not need to call it directly, but just use `metaOpt`.

Usage

```
GBS(FUN, optimType = "MIN", numVar, numPopulation = 40,
  maxIter = 500, rangeVar, gravitationalConst = max(rangeVar),
  kbest = 0.1)
```

Arguments

`FUN`	an objective function or cost function,
`optimType`	a string value that represent the type of optimization. There are two option for this arguments: `"MIN"` and `"MAX"`. The default value is `"MIN"`, which the function will do minimization. Otherwise, you can use `"MAX"` for maximization problem. The default value is `"MIN"`.
`numVar`	a positive integer to determine the number variables.
`numPopulation`	a positive integer to determine the number populations. The default value is 40.
`maxIter`	a positive integer to determine the maximum number of iterations. The default value is 500.
`rangeVar`	a matrix $(2 \times n)$ containing the range of variables, where n is the number of variables, and first and second rows are the lower bound (minimum) and upper bound (maximum) values, respectively. If all variable have equal upper bound, you can define `rangeVar` as matrix (2×1).
`gravitationalConst`	a numeric to determine gravitational constant while calculating total force. The default value is `max(rangeVar)`.

kbest — a positive numeric between 0 and 1 to determine fraction of population with best fitness which will affect every candidate solution in population. The default value is 0.1.

Details

This algorithm was proposed by (Rashedi, 2009). GBS use newton law of universal gravitation and second law of motion to optimize. Every candidate solution in population consider having mass and it move using newton law of universal gravitation and second law of motion.

In order to find the optimal solution, the algorithm follow the following steps.

- initialize population randomly.
- calculate gravitational mass of every candidate solution in population.
- calculate total force of every candidate solution in population using newton law of universal gravitation.
- calculate acceleration of every candidate solution in population using newton second law of motion.
- update velocity of every candidate solution in population based on its acceleration.
- move every candidate solution in population based on its velocity.
- If a termination criterion (a maximum number of iterations or a sufficiently good fitness) is met, exit the loop, else back to calculate gravitational mass.

Value

`Vector [v1, v2, ..., vn]` where `n` is number variable and `vn` is value of `n-th` variable.

References

Rashedi, E., Nezamabadi-Pour, H., & Saryazdi, S. (2009). GSA: a gravitational search algorithm. Information sciences, 179(13), 2232-2248.

See Also

metaOpt

Examples

```
##################################
## Optimizing the schewefel's problem 2.22 function

# define schewefel's problem 2.22 function as objective function
schewefels2.22 <- function(x){
   return(sum(abs(x)+prod(abs(x))))
}

## Define parameter
numVar <- 5
rangeVar <- matrix(c(-10,10), nrow=2)
```

```
## calculate the optimum solution using Gravitational Based Search
resultGBS <- GBS(schewefels2.22, optimType="MIN", numVar, numPopulation=20,
                 maxIter=100, rangeVar)

## calculate the optimum value using schewefel's problem 2.22 function
optimum.value <- schewefels2.22(resultGBS)
```

GOA *Optimization using Grasshopper Optimisation Algorithm*

Description

This is the internal function that implements Grasshopper Algorithm. It is used to solve continuous optimization tasks. Users do not need to call it directly, but just use `metaOpt`.

Usage

```
GOA(FUN, optimType = "MIN", numVar, numPopulation = 40,
  maxIter = 500, rangeVar)
```

Arguments

`FUN`	an objective function or cost function,
`optimType`	a string value that represent the type of optimization. There are two option for this arguments: `"MIN"` and `"MAX"`. The default value is `"MIN"`, which the function will do minimization. Otherwise, you can use `"MAX"` for maximization problem. The default value is `"MIN"`.
`numVar`	a positive integer to determine the number variables.
`numPopulation`	a positive integer to determine the number populations. The default value is 40.
`maxIter`	a positive integer to determine the maximum number of iterations. The default value is 500.
`rangeVar`	a matrix $(2 \times n)$ containing the range of variables, where n is the number of variables, and first and second rows are the lower bound (minimum) and upper bound (maximum) values, respectively. If all variable have equal upper bound, you can define `rangeVar` as matrix (2×1).

Details

Grasshopper Optimisation Algorithm (GOA) was proposed by (Mirjalili et al., 2017). The algorithm mathematically models and mimics the behaviour of grasshopper swarms in nature for solving optimisation problems.

Value

`Vector [v1, v2, ..., vn]` where `n` is number variable and `vn` is value of `n-th` variable.

References

Shahrzad Saremi, Seyedali Mirjalili, Andrew Lewis, Grasshopper Optimisation Algorithm: Theory and application, Advances in Engineering Software, Volume 105, March 2017, Pages 30-47, ISSN 0965-9978, https://doi.org/10.1016/j.advengsoft.2017.01.004

See Also

metaOpt

Examples

```
###################################
## Optimizing the schewefel's problem 1.2 function

# define schewefel's problem 1.2 function as objective function
schewefels1.2 <- function(x){
  dim <- length(x)
  result <- 0
    for(i in 1:dim){
       result <- result + sum(x[1:i])^2
   }
  return(result)
}

## Define parameter
numVar <- 5
rangeVar <- matrix(c(-10,10), nrow=2)

## calculate the optimum solution using grasshoper algorithm
resultGOA <- GOA(schewefels1.2, optimType="MIN", numVar, numPopulation=20,
                 maxIter=100, rangeVar)

## calculate the optimum value using schewefel's problem 1.2 function
optimum.value <- schewefels1.2(resultGOA)
```

GWO *Optimization using Grey Wolf Optimizer*

Description

This is the internal function that implements Grey Wolf Optimizer Algorithm. It is used to solve continuous optimization tasks. Users do not need to call it directly, but just use metaOpt.

Usage

```
GWO(FUN, optimType = "MIN", numVar, numPopulation = 40,
  maxIter = 500, rangeVar)
```

Arguments

FUN	an objective function or cost function,
optimType	a string value that represent the type of optimization. There are two option for this arguments: "MIN" and "MAX". The default value is "MIN", which the function will do minimization. Otherwise, you can use "MAX" for maximization problem. The default value is "MIN".
numVar	a positive integer to determine the number variables.
numPopulation	a positive integer to determine the number populations. The default value is 40.
maxIter	a positive integer to determine the maximum number of iterations. The default value is 500.
rangeVar	a matrix $(2 \times n)$ containing the range of variables, where n is the number of variables, and first and second rows are the lower bound (minimum) and upper bound (maximum) values, respectively. If all variable have equal upper bound, you can define rangeVar as matrix (2×1).

Details

This algorithm was proposed by (Mirjalili, 2014), inspired by the behaviour of grey wolf (Canis lupus). The GWO algorithm mimics the leadership hierarchy and hunting mechanism of grey wolves in nature. Four types of grey wolves such as alpha, beta, delta, and omega are employed for simulating the leadership hierarchy. In addition, the three main steps of hunting, searching for prey, encircling prey, and attacking prey, are implemented.

In order to find the optimal solution, the algorithm follow the following steps.

- Initialization: Initialize the first population of grey wolf randomly, calculate their fitness and find the best wolf as alpha, second best as beta and third best as delta. The rest of wolf assumed as omega.
- Update Wolf Position: The position of the wolf is updated depending on the position of three wolfes (alpha, betha and delta).
- Replace the alpha, betha or delta if new position of wolf have better fitness.
- Check termination criteria, if termination criterion is satisfied, return the alpha as the optimal solution for given problem. Otherwise, back to Update Wolf Position steps.

Value

Vector [v1, v2, ..., vn] where n is number variable and vn is value of n-th variable.

References

Seyedali Mirjalili, Seyed Mohammad Mirjalili, Andrew Lewis, Grey Wolf Optimizer, Advances in Engineering Software, Volume 69, 2014, Pages 46-61, ISSN 0965-9978, https://doi.org/10.1016/j.advengsoft.2013.12.007

See Also

metaOpt

Examples

```
##################################
## Optimizing the step function

# define step function as objective function
step <- function(x){
    result <- sum(abs((x+0.5))^2)
    return(result)
}

## Define parameter
numVar <- 5
rangeVar <- matrix(c(-100,100), nrow=2)

## calculate the optimum solution using grey wolf optimizer
resultGWO <- GWO(step, optimType="MIN", numVar, numPopulation=20,
                 maxIter=100, rangeVar)

## calculate the optimum value using step function
optimum.value <- step(resultGWO)
```

HS *Optimization using Harmony Search Algorithm*

Description

This is the internal function that implements Improved Harmony Search Algorithm. It is used to solve continuous optimization tasks. Users do not need to call it directly, but just use `metaOpt`.

Usage

```
HS(FUN, optimType = "MIN", numVar, numPopulation = 40, maxIter = 500,
  rangeVar, PAR = 0.3, HMCR = 0.95, bandwith = 0.05)
```

Arguments

`FUN`	an objective function or cost function,
`optimType`	a string value that represent the type of optimization. There are two option for this arguments: `"MIN"` and `"MAX"`. The default value is `"MIN"`, which the function will do minimization. Otherwise, you can use `"MAX"` for maximization problem. The default value is `"MIN"`.
`numVar`	a positive integer to determine the number variables.
`numPopulation`	a positive integer to determine the number populations. The default value is 40.
`maxIter`	a positive integer to determine the maximum number of iterations. The default value is 500.

`rangeVar`	a matrix $(2 \times n)$ containing the range of variables, where n is the number of variables, and first and second rows are the lower bound (minimum) and upper bound (maximum) values, respectively. If all variable have equal upper bound, you can define `rangeVar` as matrix (2×1).
`PAR`	a positive integer to determine the value of Pinch Adjusting Ratio. The default value is 0.3.
`HMCR`	a positive integer to determine the Harmony Memory Consideration Rate. The default value is 0.95.
`bandwith`	a positive integer to determine the bandwith. The default value is 0.05.

Details

Harmony Search (HS) was proposed by (Geem et al., 2001) mimicking the improvisation of music players. Furthermore, Improved Harmny Search (HS), proposed by Mahdavi, employs a method for generating new solution vectors that enhances accuracy and convergence rate of harmony search algorithm.

In order to find the optimal solution, the algorithm follow the following steps.

- Step 1. Initialized the problem and algorithm parameters
- Step 2. Initialize the Harmony Memory, creating the Harmony memory and give random rumber for each memory.
- Step 3. Improvise new Harmony, Generating new Harmony based on parameter defined by user
- Step 4. Update the Harmony Memory, If new harmony have better fitness than the worst harmony in Harmony Memory, then replace the worst harmony with new Harmony.
- Step 5. Check termination criteria, if termination criterion is satisfied, return the best Harmony as the optimal solution for given problem. Otherwise, back to Step 3.

Value

`Vector [v1, v2, ..., vn]` where `n` is number variable and `vn` is value of `n-th` variable.

References

Geem, Zong Woo, Joong Hoon Kim, and G. V. Loganathan (2001). "A new heuristic optimization algorithm: harmony search." Simulation 76.2: pp. 60-68.

M. Mahdavi, M. Fesanghary, E. Damangir, An improved harmony search algorithm for solving optimization problems, Applied Mathematics and Computation, Volume 188, Issue 2, 2007, Pages 1567-1579, ISSN 0096-3003, https://doi.org/10.1016/j.amc.2006.11.033

See Also

`metaOpt`

Examples

```
##################################
## Optimizing the quartic with noise function

# define Quartic with noise function as objective function
quartic <- function(x){
    dim <- length(x)
    result <- sum(c(1:dim)*(x^4))+runif(1)
    return(result)
}

## Define parameter
numVar <- 5
rangeVar <- matrix(c(-10,10), nrow=2)
PAR <- 0.3
HMCR <- 0.95
bandwith <- 0.05

## calculate the optimum solution using Harmony Search algorithm
resultHS <- HS(quartic, optimType="MIN", numVar, numPopulation=20,
                maxIter=100, rangeVar, PAR, HMCR, bandwith)

## calculate the optimum value using quartic with noise function
optimum.value <- quartic(resultHS)
```

KH *Optimization using Krill-Herd Algorithm*

Description

This is the internal function that implements Krill-Herd Algorithm. It is used to solve continuous optimization tasks. Users do not need to call it directly, but just use `metaOpt`.

Usage

```
KH(FUN, optimType = "MIN", numVar, numPopulation = 40, maxIter = 500,
  rangeVar, maxMotionInduced = 0.01,
  inertiaWeightOfMotionInduced = 0.01, epsilon = 1e-05,
  foragingSpeed = 0.02, inertiaWeightOfForagingSpeed = 0.01,
  maxDifussionSpeed = 0.01, constantSpace = 1, mu = 0.1)
```

Arguments

`FUN`	an objective function or cost function,
`optimType`	a string value that represent the type of optimization. There are two option for this arguments: `"MIN"` and `"MAX"`. The default value is `"MIN"`, which the function will do minimization. Otherwise, you can use `"MAX"` for maximization problem. The default value is `"MIN"`.

`numVar` a positive integer to determine the number variables.

`numPopulation` a positive integer to determine the number populations. The default value is 40.

`maxIter` a positive integer to determine the maximum number of iterations. The default value is 500.

`rangeVar` a matrix $(2 \times n)$ containing the range of variables, where n is the number of variables, and first and second rows are the lower bound (minimum) and upper bound (maximum) values, respectively. If all variable have equal upper bound, you can define `rangeVar` as matrix (2×1).

`maxMotionInduced`
a positive numeric between 0 and 1 to determine maximum motion induced. The default value is 0.01.

`inertiaWeightOfMotionInduced`
a positive numeric between 0 and 1 to determine how much motion induced affect krill (candidate solution) movement. the greater the value the greater the affect of motion induced on krill movement. The default value is 0.01.

`epsilon` a positive numeric between 0 and 1 to determine epsilon constant. The default value is 1e-05.

`foragingSpeed` a positive numeric between 0 and 1 to determine foraging speed. The default value is 0.02

`inertiaWeightOfForagingSpeed`
a positive numeric between 0 and 1 to determine how much foraging speed affect krill (candidate solution) movement. the greater the value the greater the affect of foraging speed on krill movement. The default value is 0.01.

`maxDifussionSpeed`
a positive numeric between 0 and 1 to determine maximum difussion speed. The default value is 0.01.

`constantSpace` a numeric between 0 and 1 to determine how much range affect krill movement. The default value is 1.

`mu` a numeric between 0 and 1 to determine constant number for mutation operator. The default value is 0.1.

Details

This algorithm was proposed by (Gandomi & Alavi, 2012). It was inspired by behaviours of swarm of krill. Every krill move based on motion induced (such as obstacle, predators), foraging speed (food source) and physical difussion (swarm density). In KH algorithm candidate solution represented by krill. KH algorithm also use genetic operator mutation and crossover.

In order to find the optimal solution, the algorithm follow the following steps.

- initialize population randomly.
- calculate total motion based on motion induced, foraging speed and physical difussion for each candidate solutions and move it based on total motion.
- perform genetic operator crossover and mutation
- If a termination criterion (a maximum number of iterations or a sufficiently good fitness) is met, exit the loop, else back to calculate total motion.

Value

Vector [v1, v2, ..., vn] where n is number variable and vn is value of n-th variable.

References

Gandomi, A. H., & Alavi, A. H. (2012). Krill herd: a new bio-inspired optimization algorithm. Communications in nonlinear science and numerical simulation, 17(12), 4831-4845.

See Also

metaOpt

Examples

```
##################################
## Optimizing the sphere function

# define sphere function as objective function
sphere <- function(x){
    return(sum(x^2))
}

## Define parameter
numVar <- 5
rangeVar <- matrix(c(-10,10), nrow=2)

## calculate the optimum solution
resultKH <- KH(sphere, optimType="MIN", numVar, numPopulation=20,
                maxIter=100, rangeVar)

## calculate the optimum value using sphere function
optimum.value <- sphere(resultKH)
```

---

metaOpt *metaOpt The main function to execute algorithms for getting optimal solutions*

Description

A main funtion to compute the optimal solution using a selected algorithm.

Usage

```
metaOpt(FUN, optimType = "MIN", algorithm = "PSO", numVar, rangeVar,
  control = list(), seed = NULL)
```

Arguments

FUN	an objective function or cost function,
optimType	a string value that represents the type of optimization. There are two options for this arguments: "MIN" and "MAX". The default value is "MIN", referring the minimization problem. Otherwise, you can use "MAX" for maximization problem.
algorithm	a vector or single string value that represent the algorithm used to do optimization. There are currently twenty one implemented algorithm: • "PSO": Particle Swarm Optimization. See PSO; • "ALO": Ant Lion Optimizer. See ALO; • "GWO": Grey Wolf Optimizer. See GWO • "DA" : Dragonfly Algorithm. See DA • "FFA": Firefly Algorithm. See FFA • "GA" : Genetic Algorithm. See GA • "GOA": Grasshopper Optimisation Algorithm. See GOA • "HS": Harmony Search Algorithm. See HS • "MFO": Moth Flame Optimizer. See MFO • "SCA": Sine Cosine Algorithm. See SCA • "WOA": Whale Optimization Algorithm. See WOA • "CLONALG": Clonal Selection Algorithm. See CLONALG • "DE": Differential Evolution Algorithm. See DE • "SFL": Shuffled Frog Leaping Algorithm. See SFL • "CSO": Cat Swarm Optimization Algorithm. See CSO • "ABC": Artificial Bee Colony Algorithm. See ABC • "KH": Krill-Herd Algorithm. See KH • "CS": Cuckoo Search Algorithm. See CS • "BA": Bat Algorithm. See BA • "GBS": Gravitation Based Search Algorithm. See GBS • "BHO": Black Hole Based Optimization Algorithm. See BHO
numVar	a positive integer to determine the number variables.
rangeVar	a matrix $(2 \times n)$ containing the range of variables, where n is the number of variables, and first and second rows are the lower bound (minimum) and upper bound (maximum) values, respectively. If all variable have equal upper bound, you can define rangeVar as matrix (2×1).
control	a list containing all arguments, depending on the algorithm to use. The following list are parameters required for each algorithm. • PSO: list(numPopulation, maxIter, Vmax, ci, cg, w) • ALO: list(numPopulation, maxIter) • GWO: list(numPopulation, maxIter)

- DA:
  `list(numPopulation, maxIter)`
- FFA:
  `list(numPopulation, maxIter, B0, gamma, alphaFFA)`
- GA:
  `list(numPopulation, maxIter, Pm, Pc)`
- GOA:
  `list(numPopulation, maxIter)`
- HS:
  `list(numPopulation, maxIter, PAR, HMCR, bandwith)`
- MFO:
  `list(numPopulation, maxIter)`
- SCA:
  `list(numPopulation, maxIter)`
- WOA:
  `list(numPopulation, maxIter)`
- CLONALG:
  `list(numPopulation, maxIter, selectionSize, multipicationFactor, hypermutationRate)`
- DE:
  `list(numPopulation, maxIter, scalingVector, crossOverRate, strategy)`
- SFL:
  `list(numPopulation, maxIter, numMemeplex, frogLeapingIteration)`
- CSO:
  `list(numPopulation, maxIter, mixtureRatio, tracingConstant, maximumVelocity, smp, srd, cdc, spc)`
- ABC:
  `list(numPopulation, maxIter, cycleLimit)`
- KH:
  `list(numPopulation, maxIter, maxMotionInduced, inertiaWeightOfMotionInduced, epsilon, foragingSpeed, inertiaWeightOfForagingSpeed, maxDifussionSpeed, constantSpace, mu)`
- CS:
  `list(numPopulation, maxIter, abandonedFraction)`
- BA:
  `list(numPopulation, maxIter, maxFrequency, minFrequency, gama, alphaBA)`
- GBS:
  `list(numPopulation, maxIter, gravitationalConst, kbest)`
- BHO:
  `list(numPopulation, maxIter)`

  Description of the `control` **Parameters**

  - `numPopulation`: a positive integer to determine the number populations. The default value is 40.
  - `maxIter`: a positive integer to determine the maximum number of iterations. The default value is 500.

- `Vmax`: a positive integer to determine the maximum velocity of particle. The default value is 2.
- `ci`: a positive integer to determine the individual cognitive. The default value is 1.49445.
- `cg`: a positive integer to determine the group cognitive. The default value is 1.49445.
- `w`: a positive integer to determine the inertia weight. The default value is 0.729.
- `B0`: a positive integer to determine the attractiveness firefly at r=0. The default value is 1.
- `gamma`: a positive integer to determine light absorption coefficient. The default value is 1.
- `alphaFFA`: a positive integer to determine randomization parameter. The default value is 0.2.
- `Pm`: a positive integer to determine mutation probability. The default value is 0.1.
- `Pc`: a positive integer to determine crossover probability. The default value is 0.8.
- `PAR`: a positive integer to determine Pinch Adjusting Rate. The default value is 0.3.
- `HMCR`: a positive integer to determine Harmony Memory Considering Rate. The default value is 0.95.
- `bandwith`: a positive integer to determine distance bandwith. The default value is 0.05.
- `selectionSize`: a positive integer between 0 and numVar to determine selection size. The default value is `as.integer(numPopulation/4)`.
- `multipicationFactor`: a positive numeric between 0 and 1 to determine number of clones. The default value is 0.5.
- `hypermutationRate`: a positive numeric between 0 and 1 to determine probabilty of variable in clone candidate solutions to be mutated, close to 1 probability is high and vice versa. The default value is 0.1.
- `scalingVector`: a positive numeric between 0 and 1 to determine scalingVector for mutation operator. The default value is 0.8.
- `crossOverRate`: a positive numeric between 0 and 1 to determine crossOver probability. The default value is 0.5.
- `strategy`: characters to determine mutation method. They are six methods to choose:
  - "classical".
  - "best 1"
  - "target to best"
  - "best 2"
  - "rand 2"
  - "rand 2 dir"

  The default value is "best 1".

- numMemeplex: a positive integer (as.integer()) between 0 and numVar to determine number of memeplex (see details).The default value is as.integer(numPopulation/3).
- frogLeapingIteration: a positive integer (as.integer()) to determine number of iteration for each memeplex. The default value is as.integer(10).
- mixtureRatio: a positive numeric between 0 and 1 to determine flaging proportion. higher mixtureRatio increase number of candidate solutions in seeking mode and vice versa. The default value is 0.5.
- tracingConstant: a positive numeric between 0 and 1 to determine tracingConstant. The default value is 0.1.
- maximumVelocity: a positive numeric to determine maximumVelocity while candidate solutions in tracing mode performing local search. The default value is 1.
- smp: a positive integer to determine number of duplication in genetic operator. The default value is as.integer(20).
- srd: a positive numeric between 0 and 100 to determine mutation length in genetic operator. The default value is 20.
- cdc: a positive integer between 0 and numVar to determine number of variabel in candidate solutions in seeking mode to be mutated during mutation step in genetic operator. The default value is as.integer(numVar).
- spc: a logical. if spc is TRUE smp = smp else smp = smp - 1. The default value is TRUE.
- cycleLimit: a positive integer to determine number of times allowed for candidate solution to not move. The default value is as.integer(numVar * numPopulation).
- maxMotionInduced: a positive numeric between 0 and 1 to determine maximum motion induced. The default value is 0.01.
- inertiaWeightOfMotionInduced: a positive numeric between 0 and 1 to determine how much motion induced affect krill (candidate solution) movement. the greater the value the greater the affect of motion induced on krill movement. The default value is 0.01.
- epsilon: a positive numeric between 0 and 1 to determine epsilon constant. The default value is 1e-05.
- foragingSpeed: a positive numeric between 0 and 1 to determine foraging speed. The default value is 0.02
- inertiaWeightOfForagingSpeed: a positive numeric between 0 and 1 to determine how much foraging speed affect krill (candidate solution) movement. the greater the value the greater the affect of foraging speed on krill movement. The default value is 0.01.
- maxDifussionSpeed: a positive numeric between 0 and 1 to determine maximum difussion speed. The default value is 0.01.
- constantSpace: a numeric between 0 and 1 to determine how much range affect krill movement. The default value is 1.
- mu: a numeric between 0 and 1 to determine constant number for mutation operator. The default value is 0.1.

- abandonedFraction: a positive numeric between 0 and 1 to determine fraction of population to be replaced. The default value is 0.5.
- maxFrequency: a numeric to determine maximum frequency. The default value is 0.1.
- minFrequency: a numeric to determine minimum frequency. The default value is -0.1.
- gama: a numeric greater than equal to 1. It use to increase pulse rate. The default value is 1.
- alphaBA: a numeric between 0 and 1. It use to decrease loudness. The default value is 0.1.
- gravitationalConst: a numeric to determine gravitational constant while calculating total force. The default value is max(rangeVar).
- kbest: a positive numeric between 0 and 1 to determine fraction of population with best fitness which will affect every candidate solution in population. The default value is 0.1.

seed a number to determine the seed for RNG.

Details

This function makes accessible all algorithm that are implemented in this package. All of the algorithm use this function as interface to find the optimal solution, so users do not need to call other functions. In order to obtain good results, users need to adjust some parameters such as the objective function, optimum type, number variable or dimension, number populations, the maximal number of iterations, lower bound, upper bound, or other algorithm-dependent parameters which are collected in the control parameter.

Value

List that contain list of variable, optimum value and execution time.

Examples

```
##################################
## Optimizing the sphere function

## Define sphere function as an objective function
sphere <- function(X){
    return(sum(X^2))
}

## Define control variable
control <- list(numPopulation=40, maxIter=100, Vmax=2, ci=1.49445, cg=1.49445, w=0.729)

numVar <- 5
rangeVar <- matrix(c(-10,10), nrow=2)

## Define control variable
best.variable <- metaOpt(sphere, optimType="MIN", algorithm="PSO", numVar,
                         rangeVar, control)
```

MFO *Optimization using Moth Flame Optimizer*

Description

This is the internal function that implements Moth Flame Optimization Algorithm. It is used to solve continuous optimization tasks. Users do not need to call it directly, but just use `metaOpt`.

Usage

```
MFO(FUN, optimType = "MIN", numVar, numPopulation = 40,
  maxIter = 500, rangeVar)
```

Arguments

`FUN`	an objective function or cost function,
`optimType`	a string value that represent the type of optimization. There are two option for this arguments: `"MIN"` and `"MAX"`. The default value is `"MIN"`, which the function will do minimization. Otherwise, you can use `"MAX"` for maximization problem. The default value is `"MIN"`.
`numVar`	a positive integer to determine the number variables.
`numPopulation`	a positive integer to determine the number populations. The default value is 40.
`maxIter`	a positive integer to determine the maximum number of iterations. The default value is 500.
`rangeVar`	a matrix $(2 \times n)$ containing the range of variables, where n is the number of variables, and first and second rows are the lower bound (minimum) and upper bound (maximum) values, respectively. If all variable have equal upper bound, you can define `rangeVar` as matrix (2×1).

Details

This algorithm was proposed (Mirjalili, 2015). The main inspiration of this optimizer is the navigation method of moths in nature called transverse orientation. Moths fly in night by maintaining a fixed angle with respect to the moon, a very effective mechanism for travelling in a straight line for long distances. However, these fancy insects are trapped in a useless/deadly spiral path around artificial lights.

In order to find the optimal solution, the algorithm follow the following steps.

- Initialization: Initialize the first population of moth randomly, calculate the fitness of moth and find the best moth as the best flame obtained so far The flame indicate the best position obtained by motion of moth. So in this step, position of flame will same with the position of moth.
- Update Moth Position: All moth move around the corresponding flame. In every iteration, the number flame is decreasing over the iteration. So at the end of iteration all moth will move around the best solution obtained so far.

- Replace a flame with the position of moth if a moth becomes fitter than flame
- Check termination criteria, if termination criterion is satisfied, return the best flame as the optimal solution for given problem. Otherwise, back to Update Moth Position steps.

Value

`Vector [v1, v2, ..., vn]` where `n` is number variable and `vn` is value of `n-th` variable.

References

Seyedali Mirjalili, Moth-flame optimization algorithm: A novel nature-inspired heuristic paradigm, Knowledge-Based Systems, Volume 89, 2015, Pages 228-249, ISSN 0950-7051, https://doi.org/10.1016/j.knosys.2015.07.00

See Also

`metaOpt`

Examples

```
###################################
## Optimizing the schewefel's problem 2.22 function

# define schewefel's problem 2.22 function as objective function
schewefels2.22 <- function(x){
   return(sum(abs(x)+prod(abs(x))))
}

## Define parameter
numVar <- 5
rangeVar <- matrix(c(-10,10), nrow=2)

## calculate the optimum solution using Moth Flame Optimizer
resultMFO <- MFO(schewefels2.22, optimType="MIN", numVar, numPopulation=20,
                 maxIter=100, rangeVar)

## calculate the optimum value using schewefel's problem 2.22 function
optimum.value <- schewefels2.22(resultMFO)
```

PSO *Optimization using Prticle Swarm Optimization*

Description

This is the internal function that implements Particle Swarm Optimization Algorithm. It is used to solve continuous optimization tasks. Users do not need to call it directly, but just use `metaOpt`.

Usage

```
PSO(FUN, optimType = "MIN", numVar, numPopulation = 40,
  maxIter = 500, rangeVar, Vmax = 2, ci = 1.49445, cg = 1.49445,
  w = 0.729)
```

Arguments

FUN	an objective function or cost function,
optimType	a string value that represent the type of optimization. There are two option for this arguments: "MIN" and "MAX". The default value is "MIN", which the function will do minimization. Otherwise, you can use "MAX" for maximization problem. The default value is "MIN".
numVar	a positive integer to determine the number variables.
numPopulation	a positive integer to determine the number populations. The default value is 40.
maxIter	a positive integer to determine the maximum number of iterations. The default value is 500.
rangeVar	a matrix $(2 \times n)$ containing the range of variables, where n is the number of variables, and first and second rows are the lower bound (minimum) and upper bound (maximum) values, respectively. If all variable have equal upper bound, you can define rangeVar as matrix (2×1).
Vmax	a positive integer to determine the maximum particle's velocity. The default value is 2.
ci	a positive integer to determine individual cognitive. The default value is 1.49445.
cg	a positive integer to determine group cognitive. The default value is 1.49445.
w	a positive integer to determine inertia weight. The default value is 0.729.

Details

This algorithm was proposed by (Kennedy & Eberhart, 1995), inspired by the behaviour of the social animals/particles, like a flock of birds in a swarm. The inertia weight that proposed by Shi and Eberhart is used to increasing the performance of PSO.

In order to find the optimal solution, the algorithm follow the following steps.

- Initialization: Initialize the first population of particles and its corresponding velocity. Then, calculate the fitness of particles and find the best position as Global Best and Local Best.
- Update Velocity: Every particle move around search space with specific velocity. In every iteration, the velocity is depend on two things, Global best and Local best. Global best is the best position of particle obtained so far, and Local best is the best solution in current iteration.
- Update particle position. After calculating the new velocity, then the particle move around search with the new velocity.
- Update Global best and local best if the new particle become fitter.
- Check termination criteria, if termination criterion is satisfied, return the Global best as the optimal solution for given problem. Otherwise, back to Update Velocity steps.

Value

Vector [v1, v2, ..., vn] where n is number variable and vn is value of n-th variable.

References

Kennedy, J. and Eberhart, R. C. Particle swarm optimization. Proceedings of IEEE International Conference on Neural Networks, Piscataway, NJ. pp. 1942-1948, 1995

Shi, Y. and Eberhart, R. C. A modified particle swarm optimizer. Proceedings of the IEEE Congress on Evolutionary Computation (CEC 1998), Piscataway, NJ. pp. 69-73, 1998

See Also

metaOpt

Examples

```
###################################
## Optimizing the schewefel's problem 1.2 function

# define schewefel's problem 1.2 function as objective function
schewefels1.2 <- function(x){
  dim <- length(x)
  result <- 0
    for(i in 1:dim){
       result <- result + sum(x[1:i])^2
   }
  return(result)
}

## Define parameter
Vmax <- 2
ci <- 1.5
cg <- 1.5
w <- 0.7
numVar <- 5
rangeVar <- matrix(c(-10,10), nrow=2)

## calculate the optimum solution using Particle Swarm Optimization Algorithm
resultPSO <- PSO(schewefels1.2, optimType="MIN", numVar, numPopulation=20,
                 maxIter=100, rangeVar, Vmax, ci, cg, w)

## calculate the optimum value using schewefel's problem 1.2 function
optimum.value <- schewefels1.2(resultPSO)
```

SCA *Optimization using Sine Cosine Algorithm*

Description

This is the internal function that implements Sine Cosine Algorithm. It is used to solve continuous optimization tasks. Users do not need to call it directly, but just use `metaOpt`.

Usage

```
SCA(FUN, optimType = "MIN", numVar, numPopulation = 40,
  maxIter = 500, rangeVar)
```

Arguments

`FUN`	an objective function or cost function,
`optimType`	a string value that represent the type of optimization. There are two option for this arguments: `"MIN"` and `"MAX"`. The default value is `"MIN"`, which the function will do minimization. Otherwise, you can use `"MAX"` for maximization problem. The default value is `"MIN"`.
`numVar`	a positive integer to determine the number variables.
`numPopulation`	a positive integer to determine the number populations. The default value is 40.
`maxIter`	a positive integer to determine the maximum number of iterations. The default value is 500.
`rangeVar`	a matrix $(2 \times n)$ containing the range of variables, where n is the number of variables, and first and second rows are the lower bound (minimum) and upper bound (maximum) values, respectively. If all variable have equal upper bound, you can define `rangeVar` as matrix (2×1).

Details

This algorithm was proposed by (Mirjalili, 2016). The SCA creates multiple initial random candidate solutions and requires them to fluctuate outwards or towards the best solution using a mathematical model based on sine and cosine functions. Several random and adaptive variables also are integrated to this algorithm to emphasize exploration and exploitation of the search space in different milestones of optimization.

In order to find the optimal solution, the algorithm follow the following steps.

- Initialization: Initialize the first population of candidate solution randomly, calculate the fitness of candidate solution and find the best candidate.
- Update Candidate Position: Update the position with the equation that represent the behaviour of sine and cosine function.
- Update the best candidate if there are candidate solution with better fitness.
- Check termination criteria, if termination criterion is satisfied, return the best candidate as the optimal solution for given problem. Otherwise, back to Update Candidate Position steps.

Value

Vector [v1, v2, ..., vn] where n is number variable and vn is value of n-th variable.

References

Seyedali Mirjalili, SCA: A Sine Cosine Algorithm for solving optimization problems, Knowledge-Based Systems, Volume 96, 2016, Pages 120-133, ISSN 0950-7051, https://doi.org/10.1016/j.knosys.2015.12.022

See Also

metaOpt

Examples

```
##################################
## Optimizing the step function

# define step function as objective function
step <- function(x){
    result <- sum(abs((x+0.5))^2)
    return(result)
}

## Define parameter
numVar <- 5
rangeVar <- matrix(c(-100,100), nrow=2)

## calculate the optimum solution using Sine Cosine Algorithm
resultSCA <- SCA(step, optimType="MIN", numVar, numPopulation=20,
                 maxIter=100, rangeVar)

## calculate the optimum value using step function
optimum.value <- step(resultSCA)
```

SFL *Optimization using Shuffled Frog Leaping Algorithm*

Description

This is the internal function that implements Shuffled Frog Leaping Algorithm. It is used to solve continuous optimization tasks. Users do not need to call it directly, but just use metaOpt.

Usage

```
SFL(FUN, optimType = "MIN", numVar, numPopulation = 40,
  maxIter = 500, rangeVar, numMemeplex = as.integer(numPopulation/3),
  frogLeapingIteration = as.integer(10))
```

Arguments

FUN	an objective function or cost function,
optimType	a string value that represent the type of optimization. There are two option for this arguments: "MIN" and "MAX". The default value is "MIN", which the function will do minimization. Otherwise, you can use "MAX" for maximization problem. The default value is "MIN".
numVar	a positive integer to determine the number variables.
numPopulation	a positive integer to determine the number populations. The default value is 40.
maxIter	a positive integer to determine the maximum number of iterations. The default value is 500.
rangeVar	a matrix $(2 \times n)$ containing the range of variables, where n is the number of variables, and first and second rows are the lower bound (minimum) and upper bound (maximum) values, respectively. If all variable have equal upper bound, you can define rangeVar as matrix (2×1).
numMemeplex	a positive integer (as.integer()) between 0 and numVar to determine number of memeplexes (see details). The default value is as.integer(numPopulation/3).
frogLeapingIteration	a positive integer (as.integer()) to determine number of iterations for each memeplex. The default value is as.integer(10).

Details

This algorithm was proposed by (Eusuff, Lansey & Pasha, 2006). The main inspiration for SFL algorithm originates from how swarm of frogs finding foods.

In order to find the optimal solution, the algorithm follow the following steps.

- initialize population randomly.
- separate population into "numMemeplex" memeplexes.
- update worst candidate solution using best candidate solution on each memeplex as much as "frogLeaping Iteration".
- Shuffled back each memeplexes into population.
- Sort population based on fitness.
- If a termination criterion (a maximum number of iterations or a sufficiently good fitness) is met, exit the loop, else back to separate population into memeplexes.

Value

Vector [v1, v2, ..., vn] where n is number variable and vn is value of n-th variable.

References

Eusuff, M., Lansey, K., & Pasha, F. (2006). Shuffled frog-leaping algorithm: a memetic meta-heuristic for discrete optimization. Engineering Optimization, 38(2), 129–154.

See Also

metaOpt

Examples

```
###################################
## Optimizing the quartic with noise function

# define Quartic with noise function as objective function
quartic <- function(x){
    dim <- length(x)
    result <- sum(c(1:dim)*(x^4))+runif(1)
    return(result)
}

## Define parameter
numVar <- 5
rangeVar <- matrix(c(-1.28, 1.28), nrow=2)

## calculate the optimum solution shuffled frog leaping algorithm
resultSFL <- SFL(quartic, optimType="MIN", numVar, numPopulation=20,
                 maxIter=100, rangeVar)

## calculate the optimum value using quartic with noise function
optimum.value <- quartic(resultSFL)
```

WOA *Optimization using Whale Optimization Algorithm*

Description

This is the internal function that implements Whale Optimization Algorithm. It is used to solve continuous optimization tasks. Users do not need to call it directly, but just use metaOpt.

Usage

```
WOA(FUN, optimType = "MIN", numVar, numPopulation = 40,
  maxIter = 500, rangeVar)
```

Arguments

FUN	an objective function or cost function,
optimType	a string value that represent the type of optimization. There are two option for this arguments: "MIN" and "MAX". The default value is "MIN", which the function will do minimization. Otherwise, you can use "MAX" for maximization problem. The default value is "MIN".
numVar	a positive integer to determine the number variables.

numPopulation	a positive integer to determine the number populations. The default value is 40.
maxIter	a positive integer to determine the maximum number of iterations. The default value is 500.
rangeVar	a matrix $(2 \times n)$ containing the range of variables, where n is the number of variables, and first and second rows are the lower bound (minimum) and upper bound (maximum) values, respectively. If all variable have equal upper bound, you can define rangeVar as matrix (2×1).

Details

This algorithm was proposed by (Mirjalili, 2016), which mimics the social behavior of humpback whales. The algorithm is inspired by the bubble-net hunting strategy.

In order to find the optimal solution, the algorithm follow the following steps.

- Initialization: Initialize the first population of whale randomly, calculate the fitness of whale and find the best whale position as the best position obtained so far.
- Update Whale Position: Update the whale position using bubble-net hunting strategy. The whale position will depend on the best whale position obtained so far. Otherwise random whale choosen if the specific condition meet.
- Update the best position if there are new whale that have better fitness
- Check termination criteria, if termination criterion is satisfied, return the best position as the optimal solution for given problem. Otherwise, back to Update Whale Position steps.

Value

Vector [v1, v2, ..., vn] where n is number variable and vn is value of n-th variable.

References

Seyedali Mirjalili, Andrew Lewis, The Whale Optimization Algorithm, Advances in Engineering Software, Volume 95, 2016, Pages 51-67, ISSN 0965-9978, https://doi.org/10.1016/j.advengsoft.2016.01.008

See Also

metaOpt

Examples

```
###################################
## Optimizing the sphere function

# define sphere function as objective function
sphere <- function(x){
    return(sum(x^2))
}

## Define parameter
numVar <- 5
rangeVar <- matrix(c(-10,10), nrow=2)
```

```
## calculate the optimum solution using Ant Lion Optimizer
resultWOA <- WOA(sphere, optimType="MIN", numVar, numPopulation=20,
                 maxIter=100, rangeVar)

## calculate the optimum value using sphere function
optimum.value <- sphere(resultWOA)
```

Index