

Package: QGA (via r-universe)

October 19, 2026

Type Package

Title Quantum Genetic Algorithm

Version 1.1

Date 2024-09-23

Description Function that implements the Quantum Genetic Algorithm, first proposed by Han and Kim in 2000. This is an R implementation of the 'python' application developed by Lahoz-Beltra (<https://github.com/ResearchCodesHub/QuantumGeneticAlgorithms>). Each optimization problem is represented as a maximization one, where each solution is a sequence of (qu)bits. Following the quantum paradigm, these qubits are in a superposition state: when measuring them, they collapse in a 0 or 1 state. After measurement, the fitness of the solution is calculated as in usual genetic algorithms. The evolution at each iteration is oriented by the application of two quantum gates to the amplitudes of the qubits: (1) a rotation gate (always); (2) a Pauli-X gate (optionally). The rotation is based on the theta angle values: higher values allow a quicker evolution, and lower values avoid local maxima. The Pauli-X gate is equivalent to the classical mutation operator and determines the swap between alfa and beta amplitudes of a given qubit. The package has been developed in such a way as to permit a complete separation between the engine, and the particular problem subject to combinatorial optimization.

License GPL (>= 2)

Encoding UTF-8

LazyLoad yes

Depends R (>= 3.5.0), parallel

Suggests knitr, rmarkdown, numDeriv

NeedsCompilation no

URL <https://barcaroli.github.io/QGA/>,
<https://github.com/barcaroli/QGA/>

BugReports <https://github.com/barcaroli/QGA/issues>
VignetteBuilder knitr
RoxygenNote 7.3.3
Repository <https://barcaroli.r-universe.dev>
Date/Publication 2026-10-19 15:38:56 UTC
RemoteUrl <https://github.com/barcaroli/qga>
RemoteRef HEAD
RemoteSha e9702bac802db6e68c11ec04e3f55d81dbbe2b6f

Contents

QGA	2
Index	6

QGA	<i>Quantum Genetic Algorithm</i>
-----	----------------------------------

Description

Runs a Quantum Genetic Algorithm (QGA) to optimize a user-defined fitness function over discrete decision variables. Internally it initializes a quantum population, iterates measurement, rotation, and optional mutation and interference, repairs invalid genes, evaluates fitness (in parallel if ‘parallel = TRUE’), and tracks best/average fitness across generations.

Usage

```
QGA(  
  popsize = 20,  
  generation_max = 200,  
  nvalues_sol,  
  Genome,  
  thetainit = 3.1415926535 * 0.05,  
  thetaend = 3.1415926535 * 0.025,  
  pop_mutation_rate_init = NULL,  
  pop_mutation_rate_end = NULL,  
  mutation_rate_init = NULL,  
  mutation_rate_end = NULL,  
  mutation_flag = TRUE,  
  plotting = TRUE,  
  verbose = TRUE,  
  progress = TRUE,  
  eval_fitness,  
  eval_func_inputs,  
  stop_limit = NULL,
```

```

stop_iters = NULL,
interference_flag = FALSE,
interference_rate = 0.1,
interference_type = c("diagonal", "amplitude"),
interference_lambda = 0.5,
epsilon = 0,
restart_iters = NULL,
restart_strength = 0.5,
attractor_type = c("generation", "personal"),
migration_group_size = 2,
migration_local_period = 1,
migration_global_period = 50,
parallel = FALSE
)

```

Arguments

popsize	Integer. Population size.
generation_max	Integer. Maximum number of generations to run.
nvalues_sol	Integer. Number of allowed discrete values per gene (1..nvalues_sol).
Genome	Integer. Number of genes (decision variables).
thetainit	Numeric (radians). Initial rotation step for the update rule.
thetaend	Numeric (radians). Final rotation step (linearly decayed to this value).
pop_mutation_rate_init	Numeric in [0,1]. Initial per-individual mutation probability (default '1/(pop-size+1)' when 'mutation_flag').
pop_mutation_rate_end	Numeric in [0,1]. Final per-individual mutation probability (default '1/(pop-size+1)' when 'mutation_flag').
mutation_rate_init	Numeric in [0,1]. Initial per-bit mutation probability (default '1/(Genome+1)' when 'mutation_flag').
mutation_rate_end	Numeric in [0,1]. Final per-bit mutation probability (default '1/(Genome+1)' when 'mutation_flag').
mutation_flag	Logical. Whether to apply the mutation operator.
plotting	Logical. If TRUE, plots fitness history over generations.
verbose	Logical. If TRUE, prints per-generation summary to console.
progress	Logical. If TRUE, shows a text progress bar.
eval_fitness	Function. User-provided function with signature 'function(solution_integers, eval_func_inputs)' returning a numeric fitness value.
eval_func_inputs	Any. Second argument passed through to 'eval_fitness'.
stop_limit	Numeric. Early-stop threshold; stops when best fitness >= 'stop_limit'.

<code>stop_iters</code>	Integer or NULL. If set, stops when there is no improvement over ‘stop_iters’ generations.
<code>interference_flag</code>	Logical. Whether to apply the interference operator after mutation (see ‘interference_type’). Default FALSE.
<code>interference_rate</code>	Numeric in [0,1]. Probability of applying the interference operator to the population at each iteration. Default 0.1.
<code>interference_type</code>	Character. Type of interference operator: “diagonal” (default) copies the amplitudes of each qubit from another individual following a diagonal scheme (Narayanan & Moore, 1996), without using fitness; “amplitude” adds to each qubit the fitness-weighted sum of the amplitudes of the whole population, so that amplitudes of individuals above the average fitness interfere constructively and those below the average destructively. See ‘interference()’.
<code>interference_lambda</code>	Numeric ≥ 0 . Strength of the amplitude interference (used only with ‘interference_type = “amplitude”’). Default 0.5.
<code>epsilon</code>	Numeric in $\backslash[0, 0.5)$. H-epsilon gate (Han & Kim, 2004): the probability of each bit value is kept in $\backslash[‘epsilon’, 1 - ‘epsilon’]$, so that qubits never collapse to a deterministic state. Default 0 (no gate).
<code>restart_iters</code>	Integer or NULL. If set, when the best fitness does not improve for ‘restart_iters’ generations, the population (except the best individual) is partially brought back towards equal probabilities. Should be smaller than ‘stop_iters’. Default NULL (no restart).
<code>restart_strength</code>	Numeric in $\backslash[0, 1]$. Fraction of the way each qubit angle is moved towards $\pi/4$ at a restart (1 = full reset). Default 0.5.
<code>attractor_type</code>	Character. What each individual is rotated towards: “generation” (default), the best individual of the current generation; “personal”, its own attractor, i.e. the best solution it has found so far, shared through local and global migration (Han & Kim, 2002).
<code>migration_group_size</code>	Integer. Size of the groups of consecutive individuals for local migration (‘attractor_type = “personal”’). Default 2.
<code>migration_local_period</code>	Integer or NULL. Every this many generations, the best attractor of each group is copied to the group. Default 1.
<code>migration_global_period</code>	Integer or NULL. Every this many generations, the best attractor is copied to all individuals. Default 50.
<code>parallel</code>	Logical. If TRUE, fitness evaluation is distributed over ‘detectCores() - 1’ workers via ‘foreach’/‘doParallel’. Default FALSE (sequential evaluation, no cluster is created).

Details

Solutions are encoded as bitstrings and decoded to integer values in the range 1..`nvalues_sol` for each of the `'Genome'` genes. The user must supply a fitness function `'eval_fitness(solution_integers, eval_func_inputs)'` that returns a numeric score to maximize.

Value

A list with two elements: - `'best_solution_integers'`: Integer vector of length `'Genome'` with values in 1..`nvalues_sol`. - `'fitness_history_df'`: Data frame with columns `'generation'`, `'fitness_average'`, `'fitness_best'`.

See Also

`'rotation()'`, `'mutation()'`, `'measure()'`, `'repair()'`

Examples

```
# Minimal example (toy fitness: sum of gene values)
set.seed(1)
f <- function(sol, data) sum(sol)

# This call is illustrative; tune popsize/generation_max for real problems
# out <- QGA(popsize = 10, generation_max = 5,
#           nvalues_sol = 4, Genome = 3,
#           eval_fitness = f, eval_func_inputs = NULL,
#           plotting = FALSE, verbose = FALSE, progress = FALSE)
```

Index

QGA, [2](#)