Genetic Algorithm for Laser Wakefield Accelerator (GALWA)* Documentation Version 3.0

Sam Close, Jaspreet Aujla, Dave Scott & Elisabetta Boella

July 26, 2021

Contents

1 Introduction

1.1 Principle of the Genetic Algorithm

Here is the pseudo-code for the genetic algorithm. N.B.: names provided here are placeholder and to aid understanding only.

1. Read in the initial data $\rightarrow$ `preferences`
2. Generate the population as dictated by `preferences`

*Work in progress name

    (a) For each individual created, randomly assign parameters as per `preferences`
    (b) Write these individuals to a list for the generation → `generation`

3. Run the simulation using `preferences` to adjust
    (a) For each individual, calculate a merit function with the data produced from its simulation
    (b) Write this merit function to the individual, in `generation`
4. Providing all individuals were written to, sort `generation` as required (min, or max)
5. Run the genetic algorithm as per `preferences`
    (a) Select the top 50%, and add them to the next generation → `children`
    (b) Copy the genes of `children` into → `genepool`
    (c) Randomly permute these characteristic to build the other 50% of the population, with the possibility of mutation. Add these to `children`
6. Write (over) `children` → `generation`
7. Return to step 2, and repeat as many times as dictated by `preferences`

1.2 Notation

Python data types will be qualified in the definition of attributes and methods by `::$type`. For example, `generation::int`: the generation number is an integer.

2 Files

2.1 `ga_inputs.json`

One edits this file to change the input data for the genetic algorithm.

2.2 `main.py`

As the name suggests, this script acts as the main execution of the package.

2.3 `individualclass.py`

This files defines the `Individual` class.

2.3.1 Attributes

`parameter_list` Contains the relevant parameters one wishes to test on the genetic algorithm, input as a csv list.

2.3.2 Methods

`merit_calc(input_file)`	Calculates the merit function for the individual, given their result after simulation.
`change_file(input_file)`	Opens and reads the input file into parameters objects.
`run_simulation`	Currently runs `pass`.
`list_files(directory)`	Maybe puts the files into a directory of your choosing???
`extract_merit`	Evaluates the merit of the individual and assigns it to them.
`reverse_change(input_file)`	Undoes the effect of `change_file` on the input file.

2.4 `gaclass.py`

This files defines the `GeneticAlgorithm` class.

2.4.1 Attributes

`max_generation_number::int`	Stores the maximum number of generations we will test.

2.4.2 Methods

`run`	Actually runs the genetic algorithm. Outputs a plot and can output data, but prints to terminal giving useful information regarding merit function.
`data_saver`	Takes the data from `run` and saves it in a useful format.
`data_plotter`	Takes the data from `run` and plots it using MatPlotLib.

2.5 `generationclass.py`

This file defines the `Generation` class.

2.5.1 Attributes

`generation::int`	Stores the generation number, i.e., generation zero is the original set of individuals.
`num_of_individuals::int`	Stores the number of individuals in the generation.
`mutation_rate::int`	Stores the mutation rate for the generation. The mutation rate is presently the number of individuals in the generation, i.e., that many individuals will be randomly mutated.
`population::list`	Stores the individuals for the generation.
`newborn::list`	Stores the individuals for the next generation, the have been produced from mating and mutating the present generation.
`input_file_list::list`	Stores the names of all input files

2.5.2 Methods

`__init__` and `__str__` are quite self-explanatory and as such will be neglected here.

`populate(History::list)` Creates a generation based on class attributes. `History` is a list of all unique individuals to have lived.

`output_current_status` Updates the user on the status of the simulation, by printing to terminal the number of the simulation (i.e., how many individuals have been simulated), and the details of the present individual being simulated.

`repopulate(NewPop::list, History::list)` Generates a new generation, given a list of individuals, `NewPop`. It then also writes these individuals to `History`.

`mating_stage(History::list)` As the name provocatively suggests, this stage creates a new population. The population is sorted by merit function, and the top 50% are saved for the next generation (essentially, cloned). The other remaining spaces for the newborns are generated either by `mutation_stage` or `crossover_stage`, depending on the `mutation_rate`.

`ossover_stage(History::list)` Assuming the individual suffers no mutation, they simply experience crossover. From a pool of parameters, the new individuals have randomly chosen characteristics. The `History` input here allows us to check the new individuals against those who came before; if we have already seen them, we need not add them to the list since we already know what they can do.

`utation_stage(History::list)` If the individual suffers a mutation, they still have most characteristics randomly determined as in `crossover_stage`, however, randomly, one of their characteristics is changed. Once again, `History` is parsed so we can see if they are a duplicate member.

2.6 `DataAnalysis.py`

Will be redundant in the next version.

3 Usage

1. Run `main.py`