

# GenLib v. 0.0.1

## ***Introduction***

GenLib stands for Genetic algorithm Library, whose main purpose is to provide the basis for applications which use genetic algorithms for solving optimization problems. Original idea was to come up with the design, which would support variety of types of objectives as well as single and multiple optimization within the same framework. Another goal would be to allow easy (once one managed to understand the library design) assembling and testing of variety of algorithms by combining genetic operators. I hope, that in this I have, up to certain extent, succeeded.

In order to achieve the aforementioned objectives, the combination of class hierarchy, with wide use of virtual methods, and templates was used. I did not want to create fully templated library (like STL) for the sake of components interaction flexibility. The tradeoff, of course, is the loss of efficiency for calls of virtual methods. I believe, however, that such loss is minor, in comparison with what was achieved in terms of potential extendability. The templates also were used where there was a clear advantage for the design simplicity (see, for example `GLOrganismVector <T>class`).

The library in its current stage is usable, but by far not complete (well, this is version 0.0.1 after all) . Three major parts are missing:

1. Implementation of multi-objective optimization.
2. Saving and restoring of optimizer's status.
3. Multithreading.

I plan to implement all this with time, but the speed and the order will be determined largely by the interest in this library (if any). Also, wider collection of basic genetic operators (like mutators and crossovers) would be useful. Again, if somebody will show an interest in particular feature – please, do contact me and I shall see what I can do.

## ***Design***

I am not going to boast, that this library is easy to understand and to use. I hope it is, but I am not in position to comment on this – after all it is for user to decide. What I can claim, however, that it has (in most parts) a consistent design with clear ideas behind it, which, in my experience, is the key component to understanding any library.

### **General conceptions and notations.**

As mentioned before, the flexibility of the library is achieved through build of class hierarchy. There are four main types of classes in GenLib:

- Pure abstract classes, which have form `GLBase<Type>`. Normally you should not be able to instantiate the object of base classes. `<Type>` refers to an operation, which this class's children are supposed to implement (see below). Example of such class is `GLBaseOrganism`, which is a prototype for all organisms which may be handled by the Genetic Algorithm in this library.

Interfaces, with prefixes GLInterface<>. Normally they do not contain any data and their purpose is similar to that in Java – specify an interface that class must implement. For example, GLInterfaceSwapTails is the interface for organisms which are able to swap tails. In order to be used with CROSSOVER\_ONE\_POINT the organism must implement it.

Stand-alone 'real' classes. Those are classes which do not have abstract class as their parent (apart, may be, from interface(s)). Quite often they have many static methods. Example of such class is GLConstants.

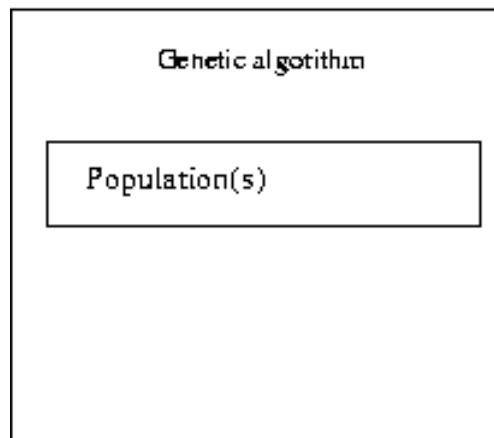
Finally, the last group of classes are those, which are derived from the base classes. They typically have name like GL<Type><Characteristic>, where <Type> is the same as in GLBase<Type> and <Characteristics> is (ideally) makes clear of the class's specifics. So, the GLMutatorGenesSwapping is child of GLBaseMutator class and swaps two arbitrary genes in organism's chromosome.

## Genetic operator's design

If you looked for this library, you are probably familiar with main concepts which are used in Genetic Algorithm optimization. I, therefore, shall explain them only to extent, which is necessary to understand the design principles used in the library.

First, the Genetic Algorithm (GA) is population-based search technique, i. e. the prospective solutions (organisms) form population(s) which change (evolve) with time, hopefully improving the average fitness of its elements and eventually finding (close to) optimal solution. The GA task, therefore, is to manage one or more populations (some variety of GAs use several populations which evolve separately and occasionally exchange the organisms) – initialize them with initial solutions, start the evolution and decide when to stop.

Therefore, it is logical, that GA should own population(s):



**Fig. 1: Genetic algorithm owns one or more populations.**

The population is responsible for

- a) storing current generation of organisms.

creating the new generation.

Population contains its own storage (type of the storage used is characteristic of population), so it is not

assumed that storage type can change during runtime.

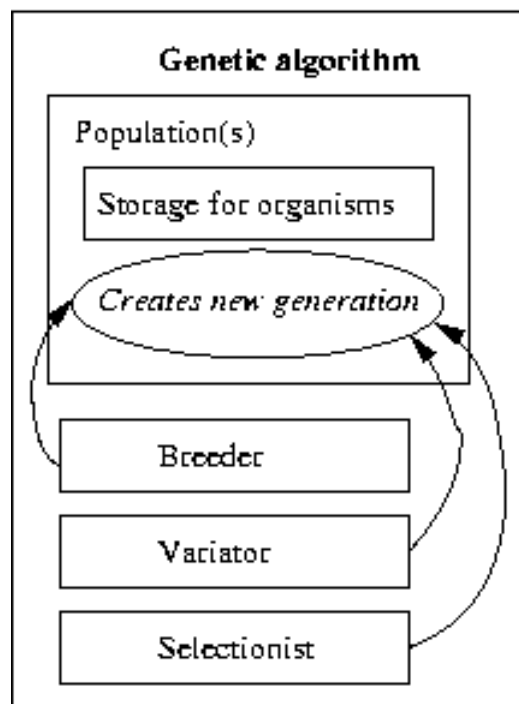
For creating new generation population applies three major genetic operators:

- 1) Breeding – choosing the organisms for breeding and applying crossover to them.

Variation – choosing the organisms to be mutated (which may or may not include newly born organisms) and applying mutation operator to them.

Selection – ranking old and new organisms and deciding which will survive for next generation.

Those population's characteristic is *how* it applies those operators, but operators itself belong to GA and are passed to population when new generation is to be formed. Therefore, GA decides when new generation will be created and which operators are to be used for this generation, but it delegates the job of creating new generation to Population class.



**Fig. 2: Genetic algorithm delegates work of creating new generation to Population passing its own genetic operators.**

Such approach allows using the same population class in different genetic algorithms.

Consider genetic operators now. Let us start with Breeder. It can be divided in two parts

- to decide which organisms to breed and with whom;

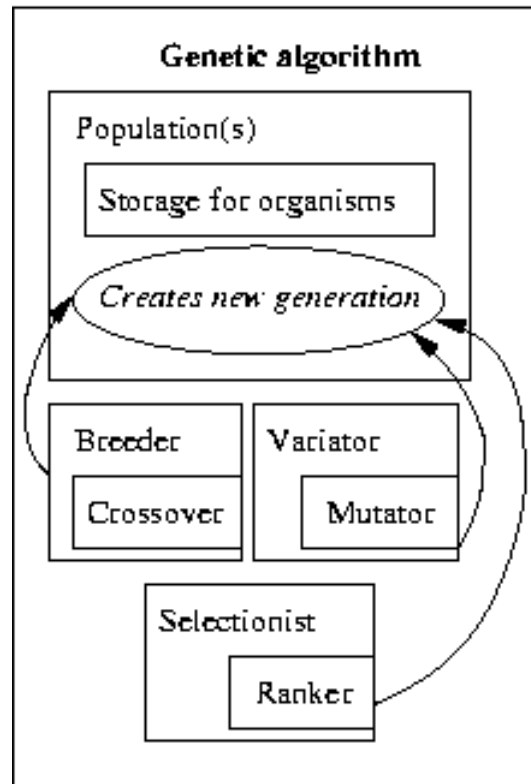
to create new children using crossover operator;

The first part usually is general and requires, may be, information about organisms' fitness, while the second operation is organism specific (crossover, in principle, must use information about internal structure of organism). That is to say, the same breeding strategy may be applied to various types of organisms, while crossover, normally, can be used with much narrower set. Therefore, each breeder uses crossover, which was passed to it in constructor.

Variator is very similar to breeder – it needs to make general decision which organisms to mutate (or which organisms have higher probability for mutation) and then to apply organism-specific mutation operator.

The task of Selectionist is to decide which organisms (old and new) are to survive for the next generation. For this it needs some way to rank them, usually according to their fitness.

So, the following structure emerged:



**Fig. 3: Breeder, Variator and Selectionist use Crossover, Mutator and Ranker operators correspondingly.**

What is described above is the use of Bridge design pattern. In case you have not read it yet, I strongly recommend you the following book: *Design Patterns: Elements of Reusable Object-Oriented Software*.

## Initializing the Genetic Algorithm.

All the elements of GA (population, breeder, variator, selectionist etc.) are created in GA's constructor using GLParametersGeneticAlgorithm and GLFactory classes. The GLParametersGeneticAlgorithm is the simple structure, containing constants corresponding to different operators, pointers to structure which contain additional data for some of the operators and constants which determine properties of the Genetic Algorithm. For example, field `m_mutator_type` may have value `MUTATOR_GENES_SWAPPING` which means that if parameters object, containing this field, will be passed to GA's constructor the GA will be created with mutator of GLMutatorGenesSwapping type. This particular mutator randomly swaps to genes in chromosome and does not require any additional paremeters, so field `m_mutator_params` will be set to NULL. If `m_mutator_type==MUTATOR_GENES_SHUFFLING` then

GLMutatorShuffling will be created. This mutator randomly shuffle certain number of genes in organism's chromosome. It requires to know the minimum and maximum number of genes it is allowed to shuffle and therefore `m_mutator_params` field must be point to object of class `GLMutatorShuffling::TParameters` in order to provide mutator with necessary information.

GLFactory creates the classes based on the constants and additional parameters provided by `GLParametersGeneticAlgorithm`. Because the GA's constructor takes the pointer to the factory and all the factory's methods are virtual, it can be extended by inheritance to handle the genetic operators which are problem-specific and defined outside the library.

## **Solving problems with Genetic Algorithm.**

In order to apply Genetic Algorithm to particular problem one needs to feed it with:

1. Fitness function evaluator;
2. Initial population;
3. Stopping criteria.

The fitness function evaluator is your objective function, or what you want to optimize. The GenLib is designed to find the minimum, so if you need to find the maximum of you will need to minimize the negative value of your objective function.

The evaluator, you pass to the solver or, rather, the fitness value object (class inherited from `GLBaseFitness`) which it returns, will determine what type of optimization you are going to use (e.g. integer or real arithmetics, single or multiple objective optimization). Use of pointer to the class as a fitness value instead of template slows things down but adds the flexibility to the design. In principle, with such approach, single and multiple objective optimizations can be done using almost identical algorithm. The only thing you need to change is Ranker and/or Selector.

Initial population is obtained from 'organism initializer'. This is class which should be inherited from `GLBaseOrganismsInitialiser` and produces the initialized organisms. Specifying the particular initializer will automatically determine the representation of your solution – e.g. vector of integers, tree, or bitmap.

Stopping criteria determines when the optimization process should stop. As GA is stochastic method usually there is no strict criteria that optimum was reached. The common criteria for stopping is number of generations passed since the start of optimization. It also can be number of generations since the last improvement to the fitness function was detected, when the list of optimal solutions was amended etc. It also could be the combination of 'simple' criteria – for example, “total number of generations reached 1000 OR number of generations since the fitness value was improved is 50 OR the best found so far value of fitness function is less than certain value'.

Providing all three components as parameters to `runGA` method allows to solve completely different types of problems.

## **Calibration of GA**

The results of optimization by GA heavily depend on numerous parameters. Changing the size of the initial population, mutation rate, type of genetic operator etc. may lead to completely different results. At the same time, as GA is stochastic method it is very difficult to judge the effects of parameter's

change after a single run. To help user with calibration GenLib includes classes GLMultipleGARunner and GLStatisticsSimple. Class GLMultipleGARunner runs takes the vector of GAs (defined with various sets of parameters) and performs a specified number of runs for each GA from the list. The results are stored in vector of pointers to class GLBaseGaStatus, containing various useful statistics. Class GLStatisticsSimple analyses the results and prints a lot of useful information – e.g. mean and standard deviation of number of generations before convergence etc.

## Example of using the GA

Examples of GA's use are contained in TestGL.cpp file. Consider one of them – testFullGa. This functions finds the minimum of  $f(\mathbf{x}) = \sum_{i=1}^{n-1} |x_{i+1} - x_i|$  where  $\mathbf{x}$  is vector of integers of size  $n$  from 0 to  $n-1$  inclusive. The solution for this problem is trivial. In fact it has two solutions:  $x_i = i - 1$  end  $x_i = n - 1$ . In both cases the minimum value of objective function is  $f(\mathbf{x}) = n - 1$ . Note, that size of the solution space (total number of possible solutions) is equal to  $n!$ .

Go through the example, omitting the debug output.

```
GLFactory factory;
```

This statement defines the factory which will be used for instantiating objects required for Genetic Algorithm.

```
//GLRandomNumbersGenerator::initGenerator(1241186119);
```

This commented line requires some explanations. Genetic library uses class GLRandomNumberGenerator to obtain random numbers and to perform relevant operations like random shuffling. If not initialized, this class will take an initial value from the timer when used for the first time. It can also be initialized at any time with the specific number. When initialized from the timer the GLRandomNumberGenerator will print the initial value, allowing user to repeat the run if any problems were encountered. This is very useful feature, taking into account the stochastic nature of the GA algorithm.

```
GLParametersGeneticAlgorithm *parameters =  
    new GLParametersGeneticAlgorithm();
```

Creates the class for storing the genetic algorithm parameters.

```
parameters->m_mutator_type = MUTATOR_GENES_SWAPPING;
```

Sets the type of mutator operators to be used in GA

```
parameters->m_initial_population_size = 50;
```

Initial size of the population. Genetic algorithms can be with constant population or with the population which varies in time.

```
GLGeneticAlgorithmStandard test_ga(*parameters, &factory);  
delete parameters;
```

Creates the GA.

```
GLBaseOrganismInitialiser *initialiser =
    new GLInitialiserShuffleIntVector(40);
```

Creates an initialiser for initializing the initial population with randomly shuffled vector of integers between 0 and 39.

```
GLBaseEvaluator *evaluator = new GLTestEvaluator();
```

Objective function.

```
TListOfStoppers stoppers;
stoppers.push_back(new GLGaStopperMaxGenerations(1000));
stoppers.push_back(new GLGaStopperNoNew(5));
```

Creates the list of stoppers – criteria for Genetic algorithm to stop. In this case GA will run no more than 1000 generations and stop as long as no new organisms appeared in population for 5 generations.

```
test_ga.runGA(initialiser, evaluator, stoppers);
```

Run GA to get the solution

```
cout << "Total " << evaluator->getCounter()
      << " evaluations were done\n";
```

Evaluator has a counter, which is updated each time the objective function is evaluated. One of the way to measure the performance of particular type of GA if objective function's evaluations have high overheads.

```
cout << *(test_ga.getStatus());
```

After finishing the status is available from the GA which contains, among other things, optimal solution(s) found during and optimization process. It is printed here.

Note, that if we decide to find the minimum among the vectors of size 10 all what is necessary is to change the parameter passed to GLInitialiserShuffleIntVector constructor.