# Tutorial: Lesson 1

This lesson will let you

- run your first Evolutionary Algorithm written within EO Library, choosing between evolving bitstrings, or evolving vectors of real numbers,
- browse through the code of these algorithms, or
- follow the guided tour.

Later you will be asked to

- write your own fitness function,
- use different kinds of selection procedures in the framework of a generational GA evolution engine,
- check that EO let you separate the representation from the evolution engine.

## I want to run an Evolutionary Algorithm now

You can choose to run a standard bitstring Genetic Algorithm (as defined in Goldberg's book) or a standard real-valued genetic algorithm, as proposed in Micahlewicz's book.

If you have not already done what was recommended in the Tutorial main page , do it **NOW**. Then go to the Lesson1 sub-dir of the tutorial dir, and simply type at the system prompt

```
(myname@myhost) EOdir/Tutorial/Lesson1 % FirstRealGA
```
or
```
(myname@myhost) EOdir/Tutorial/Lesson1 % FirstBitGA
```

and something should happen.

## What is happening?

At the moment, the FirstBitGA maximizes the number of ones in the bitstring (also calls the OneMaxfunction, whose solution is, as you can guess, 11111111), and the FirstRealGA is maximizing (the default action for EO is to maximize) the inverse of the sum (i.e. minimizing the sum) of the square of its variables (also called the sphere function, whose solution is ... all zeroes).

And what you see on the screen when running one of these two programs is, in each case, the initial and final population of an Evolutionary run, one individual per line, its fitness first, then the number of items (bits or real numbers) of the genotype, and the genotype itself. The final population hopefully contains the solution in the discrete case, and is close to it in the continuous case.

You need now to take a look at either programs by browsing alone through the sources for FirstBitGA and FirstRealGA, or follow the guided tour below, or go directly to the exercises.

## Browsing the code:

- **General includes:**Like all C-like code, the file starts with include directives (Bit - Real). Apart from standard includes, the spedific file to include is **eo**: this is a file that contains the list of the most important EO files.

- **Representation**: you then have to delclare the type of individuals you will be handling. All evolution-related objects you will need are templatized w.r.t. the type of individuals.
  - Bit You say here that you will be handling **Bitstring genotypes**, whose fitness is a double. This makes Indi derive from the STL class vector<bool>
  - Real You say here that you will be handling **Real-valued genotypes**, whose fitness is a double. This makes Indi derive from the STL class vector<double>

- **Fitness function:** the code for the fitness function is included in the file. It must take as argument a reference to an individual (at the moment).
  - Bit This function simply computes the number of ones of the bitstring (it's called the OneMax function). The optimum is of course the all-ones bitstring.
  - Real This function simply computes the inverse of the sum of the squares of all variables (also called the sphere function). The optimum is of course the all-zeroes vector.

- **Parameters**: all parameters of the algorithm are declared here (Bit - Real), and their values and assigned. Of course, this means that you will need to recompile to change these values - see Lesson 3 to get rid of that heavy requirement.

- **Random seeding:** Random numbers play an important role in Evolutionary Algorithms. See in EO programming hints more details about how this is simulated in EO - but as far as you are concerned now, remember that the global Random Number Generator is called **`rng`** and should be used everywhere you need a realization of a random variable of known law. Moreover, this RNG requires a **`seed`**, which is set here (Bit - Real): everytime you run the algorithm with the same seed, you will get the same result. Hence, to test the robustness of your algorithm, you should run it with different seeds. This is rather time consuming in the present programs, so we suggest that you wait until Lesson 3 to do so.

- **Fitness function encapsulation:** EO is based on the notion of functors - hence you now need to encapsulate your fitness function into a functor object. This is what is done here (Bit - Real).

- **Initialization**: to initialize the population, first declare an empty object of class **`eoPop<Indi>`**, which is basically an STL **`vector<Indi>`**, then fill it with Indi's. And remember that **`v.push_back`** simply appends its argument at the end of STL vector **`v`**.
  - Bit **`rng.flip()`** return a random boolean
  - Real **`rng.uniform()`** returns a real value uniformly drawn in [0,1].

- **Output**: take a snapshot at the initial population (Bit - Real). Sort it first, so the best individuals are first, and display it. Note that an eoPop has a **<<** method, which means that a simple **`os << pop`** streams the **`pop`** onto the ostream **`os`**. This is true for all objects of of class **eoPrintable** (most EO objects) through the method **`printOn`** (which is then called by the **<<** operator).

- **Evolution engine:** The selection/replacement mechanism (Bit - Real) is a simple generational GA here: a simple selector, and a generational replacement. The **eoDetTournament** has been chosen as a robust selection, and the generational replacement (all parents are replaced by the offspring) is hard-coded in the eoSGA algorithm.

- **Variation operators**: in the simple algorithm considered here, individuals undergo crossover and mutation. In EO, these operators are (functor) objects of class **eoQuadOp** (binary operator that modifies both its arguments) and **eoMonOp** (unary operator). These operators are applied in turn to all selected parents, according to user-defined probabilities. These probabilities are defined with all other parameters, and will be passed to the **`eoSGA`**algorithm.
    - Bit The crossover **eoBinCrossover** is the standard 1-point crossover, and **eoBinMutation** is the standard bit-flip mutation that randomly flips all bits with a given probability **`P_MUT_PER_BIT`**.
      **Warning**: the **`P_MUT_PER_BIT`** probability is an internal parameter of the **`eoBinMutation`**, it is **NOT** the probability of mutation at the individual level. EO corrects what can be viewed as an inconsistency in Holland's original work, further used in Goldberg's book by separating the probability of mutation for each individual (independent of the type of mutation that will be applied) from the probability of flipping each bit, which is specific of the bit-flip mutation. Hence, to run the same algorithm as Goldberg's SGA, the mutation probability (at individual level) is 1, and the probability of flipping each bit is **`P_MUT_PER_BIT`**.
    - Real The crossover **eoArithmeticCrossover** is the standard arithmetic crossover for real-valued vectors, that chooses a point randomly on the segment between both parents (also termed BLX-0). **eoUniformMutation** is the uniform mutation for real-valued vectors that choses a new value for each variable uniformly on an interval centered on the parent value. The width of the interval is an internal parameter of the object, here called **`EPSILON`**.

- **Stopping criterion:** Specify a maximum number of generations to run (Bit - Real): the simplest of all stopping criteria at the moment, using an object of a sub-class of class **eoContinue**.

- **The algorithm:** the simple algorithm that is used here, called **`eoSGA`** requires as parameters a selector, a crossover and the associated crossover rate, a mutation and the associated mutation rate, and a stopping criterion. Take a look at the corresponding constructor of the class **`eoSGA`**: it only initializes its private data with the parameters. Now look at the operator() method - the one that is called in the code for FirstBitGA or FirstRealGA - and you'll find out that is is as simple as it sounds.

- **Output**: After running the algorithm, output the sorted final population (Bit - Real) - and look at the best individual: this is the result of the algorithm.

- **Main body:** for technical reasons (intercepting the exceptions), we need a main like this one (Bit - Real)., and you should not touch it unless you know what you are doing. Simply note that this main calls the function main_function, which we have been discussing up to now!

## Exercise 1: maximize your own function

This is very easy - if your search space is that of bitstring or of unbounded real numbers.

- Go to the tutorial directory, and copy the program you want to modify onto **`mytest.cpp.`**
- Edit **`mytest.cpp`** with any text editor:
- Modify the fitness function itself (binary_value in **`FirstBitGA`**,real_value in **`FirstRealGA`**)
- Compile the program by typing **`make mytest`** at system prompt
- Run the new program by entering the command **`mytest`** at system prompt.

### Exercise 2: check the differences between both programs

Go and take a look at the code for these programs (Bit - Real). Use the symbolic representation of an Evolutionary Algorithm (you should understand that figure now, otherwise go there and come back) to understand how each part of the EA is coded. Try to spot the differences between both codes: there are not so many!
After you've tried that alone, take a look at the solution :-)

### Exercise 3: change the selection procedure

This is rather straightforward ... if you know what other types of selection are available!
At the moment, let's only consider only the following simple ones:

- You already know the tournament selection
  Syntax: **`eoDetTournament<Indi> select(T_SIZE);`** `// T_SIZE in [2,POP_SIZE)`
- Try the well-known roulette wheel
  Syntax: **`eoProportional<Indi> select;`**
- Or the stochastic binary tournament
  Syntax: **`eoStochTournament<Indi> select(RATE);`** `// RATE in ]0.5,1]`
- and of course the random selection should give bad results!
  Syntax: **`eoSelectRandom<Indi> select;`**

Note that all these classes of eoObjects are derived from the abstract class **eoSelectOne.**

# Lessons:

- in EO, all actions are performed by functor objects (this section is the last time in this tutorial that there is a direct link to the EO Programming hints page - though the link at top and bottom of all pages will remain there).
- in EO, all object you will usually need to manipulate are templatized **w.r.t. the type of the individual** you are handling.
- The type of the individual is itself templatized **w.r.t. the type of fitness** (double by default).
- In EO (actually, in EC!) initialization and variation operators are representation-dependent, while the evolution engine is representation-independent (well, like any rule, this one does have some exceptions).
- Changing the fitness function, or the selection procedure inside the generational GA evolution engine is straightforward.
- remember, all solutions to exercises are in the same sub-dir of dir Tutorial than the lesson itself (see here).

---

documentation

---

*Marc Schoenauer*

Last modified: Fri Nov 3 18:49:12 CET 2000