GUIDE-II user manual

version 1.0

This manual describes GUIDE-II, a Graphical User Interface to ease the design of Evolutionary Algorithms: the only programming that the user cannot avoid writing is the fitness function – any other component being given default value as soon as the genotype has been (graphically) designed.

License: GUIDE is distributed under the GPL license

Downloading: At the moment, contact Marc.Schoenauer@inria.fr to obtain the full GUIDE package.

Installation: GUIDE relies on the C++ library Evolving Objects (see http://eodev.sourceforge.net), so EO must be installed before installing GUIDE.

GUIDE comes as a set of Java files that simply need to be compiled together (no additional package is required). The main file is **`guide/Guide.java`**

Table of Content

I) Introduction

Guide is a program generates evolutionary algorithms from a graphical user interface. It has been build in open source for research and education in science. The aim of Guide is to make evolutionary algorithm easy to create even if the result is quite standard.

II) Evolution Engine

The evolution engine allows the user to set all parameters independent of the genome. He defines the number of parents that are selected for reproduction or mutation and how the parents are selected and how individuals are reduced to fit the correct amount of individuals. The evolution engine gives the possibility to change the goal, ether minimisation or maximisation. The user can choose the number of generations the algorithm will go throw and so on.

1) The upper banner

The upper banner has four choices. The number of generations are the number of generations the algorithm will run before giving the result found. Pop is the number of individuals in the population. Evolution Engine is the strategy used for the algorithm. By changing this parameter, you may change parameters below. The goal allows you to choose between minimisation or maximisation of the evaluation function.

2) Selection and reduction

In this section the user defines how and how many individuals are selected for crossover and mutation, and how and how many children and parents are reduced to fit the good population size at the end of the generation. This section defines the probability of mutation and crossover of a selected parent.

i) The Init. Pop. Bar

The first bar is divides in three population : the elite population, the fertile population and the non elite population (N.E.P.). The elite population is the best individuals at the beginning of the generation. The elite population can be copied in the population at the end of the generation. If strong elitism the elite population will always be copied. If weak elitism the elite population will be copied if stronger of

the children. The fertile population is the population that can give parents. Elite population is always fertile. The non elite population is the population that is not in the elite. Further on they will be reduced between themselves and with the children. The size of these three populations can be determined on the bar or on the three input fields (Elite, Fert. And N.E.P). You can ether enter the population size or a percentage of the total population.

ii) The parent bar

The parent population is the population that gives children and that can mutate. To change the size of that population, you can move the bar or insert the size in the selector field. You can ether enter the population size or a percentage of the total population. This population is issued from the fertile population by a selector. You can choose the selector type and one of it's parameters in the Parents Selector box on the left of the bar.

iii) The offspring bar

The offspring population is issued from the parents. In a first time they are simply copied. But they may be modified by mutation or crossover. You can chose their size on the box or in the field "Offsp" on the left of the parent box. In that field you can choose between a fix number of a percentage of the total population.

iv) Probability of mutation and crossover

Between the parent and the offspring box, you have the two fields. Each child has a probability of mutation and a probability of crossover. When mutated or crossed over the child is modified. A child can be crossed over and mutated.

v) The intermediate bar

The intermediate bar is a first reduction. There is two sub-bars, a blue one and a pink one, symbolizing the N.E.P and the offspring populations. This first step reduces N.E.P to a fix number and offspring to a fix number. The two populations are reduced separately. On the left of the bar you can enter the intermediate size of both population (fix number or percentage) and choose the type of reduction (with a parameter).

vi) The end population

This population is the population is ready for the next generation. You reduce the intermediate population with a reducer in the box on the left of the bar. You can

use one parameter. The elite population can be copied into it. If you choose strong elitism (the box on the left of the bar), the elite population will always be copied. If you chose weak elitism, they will be copied if stronger of the resulting population.

vii) Selectors and reducers

The selectors and reducers are the following:

- Tournam(N) : chooses N individuals and returns the best.

- Stoch.Trn(p) : chooses two individuals and returns the best with a probability of p. The parameter p should be better than 0.5.

- E.P. Trn(T) : for every individual I, it chooses T opponents. It then counts (c)the number of opponents I is better. The individuals having the best c are kept.

- Ranking : does a roulette on the rank of the individuals. The best individuals have the rank popSize.

- Roul.Wh.: does a roulette on the fitness (on maximisation).

- Sequent.: returns the individuals in order from the best to the worth.

- Uniform : chooses the individuals on a random base.

Some of them are invalid as selectors or reducers. They can not be selected by the user.

III) Genome creation

To create the genome of your algorithm, you have to use chromosomes. Each chromosome must have a unique name. A chromosome is ether a container (a nod) or a basic type (a leaf). A container may be a multiple children container or a single child container.

1) Types of chromosomes

A single child container has one nod but the child can represent several elements. For example a vector of ten booleans has one child boolean but that child represent ten elements. These multiple elements container are divides in four types. Ether the container has a fixes number of elements or not. Ether the container is ordered (order is important) or not. The four types are bag (fixes size, unordered), list (unfixed size, ordered), set (unfixed size, unordered) and vect (fixed size, ordered).

The four basic types are boolean, integer, real and permutation. The type

boolean takes the values true and false. The integer takes all the integer values between this minimum and maximum. The minimum and maximum may be infinite. The real is in fact a double variable that takes all values between his minimum and maximum. The minimum and maximum may be infinite. The permutation is a array of integer values. Each value from 0 to N-1(N is the size of the permutation) is present is the array one and only once.

There are two chromosomes that handles multiples children. The root type and the tuple type. Each child has one element. The root is always at the top of the genome, but the tuple can be anywhere else.

Each chromosome has a types and a unique type. The unique type depends on the unique types of the children and some times of the names of the children. If a list of integers is changed to real, the type of the list stays “list”, but its unique type change. If the user defines an operator on list using the integers, it can become incorrect if the child is changed to real. For that reason when unique types are changed a message is appears asking you if you want to keep these operators.

2) Creating the genome

In the "Genome Creation" tab, it is possible to modify and create the genome structure. By clicking with the right button over a chromosome, a menu pops up. With that menu you can add a chromosome over or under the selected chromosome. An option allows you to delete the subtree. And the last two options allows you to copy and paste a subtree. You can modify a chromosome by clicking on its button. To modify the parameters of a chromosome press the button and choose the same type of chromosome.

If a chromosome is modified or if the user is creating a new chromosome, a dialog opens. That dialog asks for a name and a type of chromosome. The name of the chromosome must be unique. If parameters are needed they're asked by a specific dialog.

IV) Operators manipulation

There are three sorts of operators : the initialisation operators, the mutation operators and the crossover operators. There are three tabs, one for each sort of operator. Each chromosome must have one operator of each sort.

An operator is local to a chromosome. However it is possible to write an operator that doesn't obey that rule. The initialisation operators on container (bag,

list, set and vector) types and the permutation type are important. The objects created don't have any elements. It is only at initialisation time that these objects get their elements.

Operators might be predefined or user defined. A user defined operator is one that the user build by his own. He has to set the code for it. The predefined operators have been written by the developers and can be selected in a list of operators.

The operator dialog opens when you click on the button of the chromosome. You can add a new operator, change an operator for an other, delete an operator. The import button allows you to copy the code of an existing operator to create a new one. To change the parameters of an operator choose modify and select the same type of operator.

You can have more than one operator for a chromosome. When the chromosome is executed, one operator will be chosen depending on it's weight. The probability of being chosen is equal to this weight divided by the total of all weights on that chromosome. You can modify the weight of the operators in the operators dialog.

1) user defined operators

There are two ways of creating a new operator type. Ether by selecting the "user defined" line in the adding list, or by using the import button. To modify the code of an user defined operator you have to click modify and choose to modify that operator by himself. In this situation all operators will be modified. A new operator type needs a name unused for that chromosome type.

The code of an operator is displayed as a function, in reality it is a method. The parameter that is passes to that method are the parameters of the chromosome and of the operator. To access an element of a multiple element chromosome put [] after the chromosome name. To access a child of the chromosome of type tuple or root, you simply put a point and the chromosome name after the variable. The name of the child is encapsulated but the name is pre-processed into an access method (see the evaluation function section). If you put a space before the point, there will be no pre-processing into a method. To see the way to access a child or a element, look at the evaluation function tab. The difference is that in the evaluation you access a chromosome by a chain from the root of the genome. In an operator you access a chromosome by a chain from the chromosome of that operator.

Lets take the following example. The operator is user defined and contradicts

the chromosomes independence. The genome is a vect of reals. This is a operator on the vect chromosome. It adds an random real between 0 and 1, to all elements of the vector. This random number is the same for all the elements.

```
bool mut_vect_plus_rand(vect& _var0)
  bool modified=false;
  double rand = eo::rng.uniform(1);
  for(int i =0;i<var0.getSize();i++) _var0[i]+=rand;
  modified=true;
  return modified;
};
```

The types bag, list, set and vect are descend of the C++ std::vector class. So the methods of that class are available for these chromosomes. A method getSize() returns the number of elements of the container.

In the following example, the operator is on a tuple chromosome, the child integer is a integer number and the positive child is a boolean that is true when the integer is positive. The do_children is a primitive that asks all children to execute an operation on themselves. After the do_children primitive the boolean has a random value. It must be recalculated, in order to correspond to the integer value.

```
bool mut_tuple_MyBoolean(tuple& _var0){
  bool modified=false;
  do_children;
  if(_var0.integer<0)  _var0.positif= false;
  else _var0.positif=true;
  modified=true;
  return modified;
};
```

2) operators code

The code of an operator is the definition of it's actions. This section applies to user defined as well to predefined operators code. When defined by the user a dialog

appears simulating a function. In reality the code is inserted in the method of a functor. This functor is a C++ class.

To apply an operator on a chromosome, the C++ calls the appropriate functor operator (C++) and passes the chromosome as the parameter. The result of that operation is ether void, for initialisation, or bool for mutation or crossover.

The code used for an operator uses normal C++ instructions. You can include all instructions linked to the type of the chromosome. If the chromosome is a integer you can use the + operator, the -- operator, the * operator and so on. It is up to you to verify that the integer isn't over the maximal or under the minimum. In addition to normal C++ instructions, you may use (and it is advised) primitives. The primitives are special commands that are expanded in instructions during pre-processing.

i) parameters

Operators may have parameters. The constructor of the functor object takes one operator for each child, all the parameters of the chromosome and of the operator, the parser (where parameters can be overwritten) and the name of the chromosome as parameters. The mutation and crossover functors of multiple elements chromosomes take the initialisation operator of the child chromosome too. If the chromosome has children, the functor has a link towards the operator of each child. The name of this attribute is apply_on_childX, where X is the rank of the child. Each child must have a operator, even if it is never used. Otherwise the generated code will bug. The functor class has as private attribute for all the parameters of the chromosome and of the operator. All operators of the same chromosome type must have different names. Be aware that conflics between attributes names or C++ key words and variables names will create a bug during compilation.

The value of the parameters passed to the constructor are the default values for that operator. It is possible to modify these values by passing a parameter file at the execution of the algorithm (see the run button section).

Parameters can be used freely in the code. They are attributes that are initialised during the construction of the functor object.

a) The eoGeneralIntBounds and eoGeneralRealBounds

Both are chromosomes parameters of integers and real for the minimum and maximum of the chromosome. They are passed to the operators as explained on the upper text.

- virtual bool isBounded (void)

Self-Test: true if both a min and a max exists.

- virtual bool hasNoBoundAtAll (void) const=0

Self-Test: true if no min and no max.

-virtual bool isMinBounded (void)

Self-Test: bounded from below.

-virtual bool isMaxBounded (void)

Self-Test: bounded from above.

-virtual bool isInBounds (double)

Test on a value: is it in bounds?

-virtual void foldsInBounds (double &)

Put value back into bounds - by folding back and forth.

-virtual void foldsInBounds (long int &i) const

foldsInBounds for ints: call the method for double and convert back. (remember to cast int& to long int&)

-virtual void truncate (double &)

Put value back into bounds - by truncating to a boundary value.

-virtual void truncate (long int &i) const

truncate for ints: call the method for double and convert back (or doubles)

-virtual long int minimum () (or double)

get minimum value ::exception if does not exist

-virtual long int maximum ()

get maximum value ::exception if does not exist (or double)

-virtual long int range ()

get range ::exception if unbounded

-virtual double uniform (eoRng &_rng=eo::rng)

random generator of uniform numbers in bounds uses same naming convention than eo::rng ::exception if unbounded

-virtual long int random (eoRng &_rng=eo::rng)

ii) primitives

The code of operators have primitives. These primitives are used in the user defined code and in the predefined code in the same way. A primitive must be on a single line to be recognized and expanded. The only characters allowed on the line are spaces and the ';' all other characters included comments must not appear. A primitive can be expanded into several instruction so be aware of the brackets.

```
if(condition) primitive; //is incorrect the primitive is not on a single line


if(condition) //incorrect take care of the brackets
     primitive;


this can be expanded into
//only the first instruction depends on the condition
if(condition)
    primitive_instruction1;
primitive_instruction2
primitive_instruction3;

if(condition) {//correct
   primitive;
}
```

this can be expanded into

```
//all instruction depends on the condition
if(condition) {
  primitive_instruction1;
  primitive_instruction2
  primitive_instruction3;
}

if(condition) {
   primitive; //incorrect because of the comment
}
```

a) general primitives

·do_children

This primitive may be applied to all operators on all chromosomes. It means that the operator action is transmitted to all children. A operator action is ether initialisation, mutation or crossover. Then an operator action is asked on a child, the appropriated operator is executed on it. If a chromosome is a collection of elements, this operator executed a operator on each element. All of its elements are applied.

b) collection types primitives

The primitives in this section are available only on chromosomes of type collection. This chromosomes have type bag, list, set and vect. They can't be used on other chromosomes operators. The type of the elements of theses chromosomes are X.

·apply_on_index(i)

This primitive applied the action operator (initialisation, mutation or crossover) on the element of index i. An appropriate operator will be applied on that element and on no other. In the case of a crossover action operator, the two children parents are the two elements of index i. If that i is too big (crossover and

initialisation, mutation) nothing happens. For crossovers you can use apply_on_index(i,j). The index i is on the first child and j is the index on the second child.

·*init_child(x)*

Reinitialises the element x. In this situation x is a variable.

·*setSize and setSize(x)*

After construction a collection chromosome has no elements. Even if the type is bag or vect. In order to give the good size of elements you must use this primitive during initialisation. For chromosomes bag and vect, that have a fixed size, use setSize with no parameters. In that case the size will be fixed to the size parameter of the chromosome. For chromosomes list and set, that have no fixed size, use setSize(x) with a parameter x, which contains the number of elements to initialise the chromosome. The parameter x can be a variable or a number. The elements are created with a no parameter constructor. These elements are not initialised. To do that call primitive do_children after this the primitive setSize.

iii) collection types instructions

All collection chromosome is a template descendant eoAbstractVector who is itself descendant of of std::vector.

a) std::vector methods

·*push_back*

void push_back(const T& x);

Adds the element x at the end of the vector. (T is the data type of the vector's elements.)

·*begin*

iterator begin();

Returns an iterator (a special kind of object) that references the beginning of the vector. Although the iterator can be used for many things, for now we will just consider its use with erase and sort.

·*end*

iterator end();

Returns an iterator (a special kind of object) that references a position past the end of the vector. Like begin(), we will just consider its use with erase and sort.

•erase

void erase(iterator first, iterator last);

Erase (remove) elements from a vector. For now, we will consider only the case of removing all elements from a vector (see clear() for an alternate way to do the same thing):

vector<int> a;

...

a.erase(a.begin(), a.end()); // Remove all elements.

a.erase(a.begin()+2, a.begin()+3); // Remove the second element.

•insert

void insert (iterator pos, X x);

a.insert(a.begin()+i, x)

Inserts element x after at position i.

•clear

void clear ();

Erase all elements from a vector.

vector<int> a;

...

a.clear(); // Remove all elements.

b) std::vector operators

•=operator

The assignment operator replaces the target vector's contents with that of the source vector:

vector<int> a;

vector<int> b;

a.push_back(5);

a.push_back(10);

b.push_back(3);

b = a;

// The vector b now contains two elements: 5, 10

·[] operator

The subscript operator returns a reference to an element of the vector. A subscript value of zero returns a reference to the first element, and so on. The subscript must be between zero and size()-1. See the example above to see how the subscript operator is used in a loop to access elements of a vector. The subscripted vector may appear on the left or right sides of an assignment (the returned reference is an lvalue):

vector<double> vec;

vec.push_back(1.2);

vec.push_back(4.5);

vec[1] = vec[0] + 5.0;

vec[0] = 2.7; // Vector now has two elements: 2.7, 6.2

c) eoAbstractVector methods

·shuffle

void suffle()

Shuffle all elements around. It is a good thing to shuffle bas and sets at the beginning or the code. This cuts the idea of order in these containers.

·printOn

void printOn(ostream& os) const

Prints the collectors on the os stream.

·readFrom

void readFrom(istream& is)

Reads he collection from the is stream.

·getSize

unsigned int getSize() const

Returns the number of elements of the container.

·className

std::string className()=0

Return the name of the class

·!= operator

bool operator!=(const eoAbstractVector<X>& _x)

Is the opposite of operator ==.

·== operator

bool operator==(const eoAbstractVector<X>& _x)=0

Returns true is the two chromosomes are equal.

d) Multiple elements containers types

The types of multiple containers are:

- template<class X> class bag
- template<class X> class list
- template<class X> class set
- template<class X> class vect

iv) permutation type instructions

·getSize

unsigned int getSize() const

Returns the number of elements

·setSize

void setSize(unsigned int _size)

Initialises the permutation with _size elements. The value of these elements are

incoherent. They are integers with any od value.

- ***printOn***

void printOn(ostream& os) const

Prints on the stream os.

- ***readFrom***

void readFrom(istream& is)

Creates a new permutation by reading the stream is.

- ***[] operators***

unsigned int & operator[](int k)

unisgned int operator[](int k) const

Access tho the element k. Can be a left or a right value.

For example lets exchange index k1 and k2.

```
unsigned int temp = p[k1];
p[k1]=p[k2];
p[k2]=temp;
```

- ***== operator***

bool operator==(const permut &p)

Returns true if both permutations are equal.

- ***!= operator***

bool operator!=(const permut &p)

Returns true if both permutations are different.

v) the eo::rng global variable

The eo::rng is a global variable available anywhere in the code. It generates random numbers depending on a probability law. The following methods may be used.

- eoRng (uint32 s) : ctor takes a random seed; if you want another seed, use reseed.

- void reseed(uint32 s) : Re-initializes the Random Number Generator.

- void oldReseed (uint32 s) : Re-initializes the Random Number Generator - old version.

- double uniform (double m=1.0) : uniform(m = 1.0) returns a random double in the range [0, m).

- uint32 random (uint32 m) : random() returns a random integer in the range [0, m).

- bool flip (float bias=0.5) : flip() tosses a biased coin such that flip(x/100.0) will returns true x% of the time.

- double normal (void) : normal() zero mean gaussian deviate with standard deviation of 1.

- double normal (double stdev) : normal(stdev) zero mean gaussian deviate with user defined standard deviation.

- double normal (double mean, double stdev) : normal(mean, stdev) user defined mean gaussian deviate with user defined standard deviation.

- double negexp (double mean) : Generates random numbers using a negative exponential distribution.

- uint32 rand () : rand() returns a random number in the range [0, rand_max)

- uint32 rand_max (void) const : rand_max() the maximum returned by rand()

- template<class T> int roulette_wheel (const std::vector< T > &vec, T total=0): roulette_wheel(vec, total = 0) does a roulette wheel selection on the input std::vector vec.

- void printOn (std::ostream &_os) const : Write object.

- void readFrom (std::istream &_is) : Read object.

- std::string className (void) const : Return the class id.

V) Evaluation Function

The evaluation function is the numeric evaluation of a individual. To insert a evaluation function you must go on the evaluation function tag. The way that it is appears on the screen, it looks like a C++ function, but in fact a method is created. The function on the screen give access to a genome variable, creates a fit variable and returns fit. Theses lines added to the comments can not be changed. Do not use the return instruction in your code.

1) access to sub elements and children

The genome variable contains the root of the genome. To access a specific chromosome or element you must go down the genome. To access a child chromosome of a the root chromosome or a tuple, you must add a point and the name of that chromosome. It is important not to put a space between the point and the name. The access of a child on a chromosome by that way is private. The file is pre-processed to change the private access into the public access method. A access genome.var is pre-processed into genome.getvar0(). If you put a space after the point, your command is not pre-processed and the private access stays in place. This is useful to disable pre-processing. To access elements of bag, list, set and vect, you must put the index of that element between brackets.

For each chromosome, a line of comment is generated. That comment gives the access chain to that chromosome from the genome root, its name, and its type (C++). In the access chain brackets are shows without indexes for elements of a container.

2) the fit variable

The fit variable, of type double contains the value of the evaluation function at the end of the function. It is initialised to zero at the beginning. At the end it must contain the value of the evaluation. Do not write your own return instruction, a bug will appear.

3) User code

Between the fictive declaration of the fit variable and the return instruction, you can enter your own C++ code. You can use all methods of the chromosomes you handle in this aim.

VI) Menu and Buttons

1) open menu and button

Opens and loads an existing file XML project. The name before the XML extension becomes the project name.

2) Save (menu and button) and SaveAs(menu)

The project must have a name. To generate files, build and run them, you must have a name. This name comes from the XML file name. To set a name you must ether open a project or use saveAs. The name of the project is the name of before the the XML extension. The save menu and button saves the project in the file it's issued. As other files will be generated it is a good habit to save the project in its own directory.

3) The refresh button

Loads the project as it was the last time it was saved.

4) The Generate Button

This button calls the save action and the file generation action. You must have a project name (see saveAs or Open). The file generation action creates all C++ (EO) files for the algorithm, the makefile and the *.prm file. The C++ files are created but are not yet compiled so bugs are possible. If a different makefile is present in the project directory a question will ask you to erase or not the old makefile. This makes it possible for the user to change the makefile, to include other files for example, and doesn't systematically destroy it. The *.prm (* = name of the project) file is the file that contains all the parameters. It contains all the parameters of the evolution engine

(except minimum or maximum goals) and can contain the parameters of chromosomes and operators. You can change these parameters for the algorithm without recompiling (see the run section).

The following file are generated (XXX is the project name):

- eoXXX.h : chromosomes code
- eoXXXInit.h : initialisation operators code.
- eoXXXMut.h : mutation operators code.
- eoXXXQuadCrossover.h : crossover operators code.
- eoXXXEvalFunc.h : evaluation function code.
- EoXXXStat.h : code for the statistics on the population (best, mean).
- XXXEA.cpp : main part.
- XXX.prm : parameters file.

5) The build button and menu

This builds the project into a executable. The result of the compilation is shown in the Experience Monitor tab. If you change only the evolution engine parameters, you don't have to recompile (except for the goal), just press the generate button (see the run section). To compile in a monitor, in the project directory type:

>make XXXEA

Where XXX is the project name.

To build a project in Guide or in a terminal, you must have the EO library installed. If you run it in Guide, you must have told Guide were in witch directory the library is. To change the directory use the Build/Settings menu.

6) The Run button.

This button runs the experience and shows the result in the Experience Monitor tab. However if a bug exists at the execution level this process might block. To recover you must close and reopen Guide. To execute in a monitor, in the project directory type:

> ./XXXEA @XXX.prm

Where XXX is the project name.

XXXEA is the executable. The XXX.prm is a file containing all the evolution

engine parameters. Without this file the algorithm will ignore all these parameters. You can overwrite the chromosomes and operators parameters by adding a line with there name and value.

For a parameter on a chromosome the name is :

--var0_size = 5 (on a vect for example)

YYY_NNN (YYY is the name of the chromosome and NNN the name of the parameter)

For a parameter on an operator the name is :

--init_var0_tosscoin_proba = 0.5 (the proba value on the tosscoin operator on a boolean is 0.5) (the probability it becomes true)

SSS_YYY_MMM_NNN (SSS is the sort of operator (init, mut or cross) YYY is the name of the chromosome, MMM the name of the operator and NNN the name of the parameter)

For a weight on a operator on a multiple operator chromosome :

--initOp_var0_1 = 0.5 (the weight of the first operator of initialisation on var0 is 0.5)

SSSOp_YYY_Z (SSS is the sort of operator (init, mut or cross) YYY is the name of the chromosome and Z is the rank of the operator 1,2…)

If you execute XXXEA without the XXX.prm file the evolution engine parameters will be ignored and the chromosomes and operators parameters will be the ones set by the user in Guide.

7) Quick Run.

If you only have modified the evolution engine, you don't need to rebuild the project. At the exception of the minimal or maximal goal. You just can generate the files and run.

VII) Special issue

1) the userXXXEA.cpp

If a file called userXXXEA.cpp (XXX is the name of the project) is present it will be includes in the generated files. So if you want to include your files in the project create a userXXXEA.cpp file and wite:

userXXXEA.cpp :

#include “myFiles.h”

The header file “myFile.h” will be included in the generated files. It will be visible in the hole project, in all chromosomes, operators and evaluation code. You can include as many lines you want. Bewared to compiles the myFiles.h code. To do this change the makefile.

You can call the functions and objects created in the evaluation function (and operators) as follows:

Evaluation function :

```
double eval_function(GenomeClass &genome)
{
  double fit=0;
  //BEGIN YOUR CODE
  myFile myObject = new myFile();//create your external object
  myObject->myMethode();//call your method
  //END OF YOUR CODE
  return fit;
}
```

2) The userXXXEvalFunc.h

You can include methods in the evaluation function class by creating a userXXXEvalFunc.h file (XXX is the name of the project). The content will be included in the class as follows:

userXXXEvalFunc.h :

```
private :
   int c=0;
public:
  void count(){
     c++;
  }
```

You can call the methods and attributs created in the evaluation function as follows:

Evaluation function :

```
double eval_function(GenomeClass &genome)
{
  double fit=0;
  //BEGIN YOUR CODE
  count();
  if(c%100=0) cout<<c<<”evaluations”<<endl;//writes a line every 100 evaluations
  //code
  //END OF YOUR CODE
  return fit;
}
```

VIII) Conclusion

Guide and its practical user interface makes Evolutionary Algorithm easy to develop. This allows scientists that have little knowledge in Evolutinary Computation, or even in computer programming, to create their own program for a functional optimisation. It gives the possibility to computer scientists to create quickly a generic evolutionary algorithm that they can later modify easily.

Contact: Marc.Schoenauer@inria.fr.