Design document for Differential Evolution Algorithm

Ali Khudiyev, Kanan Jafarli, Ibrahim Yunuslu

November 2020

1 Introduction

1.1 Differential Evolution Approach

Differential Evolution (DE) is a metaheuristic that tries to solve an optimization problem without having any strong assumptions beforehand. The way DE does so, is by using better candidate solutions(agents) iteratively in each generation until a satisfactory solution is obtained. The candidate solutions are used systematically by predefined mathematical system. Although it is not guaranteed that satisfactory solution will be obtained after n generations, the algorithm tries to converge after each generation which gives us the hope that eventually, the final solution will satisfy the predefined threshold value.

The differential evolution algorithm (DEA) is usually implemented to find the global minimum of an arbitrary dimensional function. However, it can also be used to find the global maximum of function $f(\bar{x})$ if given as $-f(\bar{x})$, and for a specific point y_0 in space the function should be passed as $[f(\bar{x}) - y_0]^2$.

Algorithm. The differential evolution algorithm works with randomly generated population(also known as agents). The algorithm is mainly based on iteratively using agents of each generation to obtain hopefully a better population. So, three points(called **target** and two different agents P_1, P_2) are randomly picked from the current generation and a new point called **mutant** is obtained by shifting the **target** in the direction of $P_1 - P_2$.

$$v_{i,G+1} = x_{r_1,G} + F \cdot (x_{r_2,G} - x_{r_3,G})$$

After the process of mutation, we recombine the **mutant** and randomly chosen(from the original population) **parent** to obtain new **trial** and this stage is called the recombination.

$$u_{ji,G+1} = \begin{cases} v_{ji,G+1}, & \text{if randb}(j) \leq P_{\text{crossover}} \text{ or } j = \text{rnbr}(i) \\ x_{ji,G}, & \text{if randb}(j) > P_{\text{crossover}} \text{ and } j \neq \text{rnbr}(i) \end{cases}$$

$$\text{where } j = 1, 2, ..., D$$

The third stage involves selection of better candidate between the **parent** and **trial** according to a greedy approach(best candidate is the one which is closer to the global minimum). The selected point is replaced by the **parent** and after doing these three processes for all points, a new or the next generation is obtained.

$$x_{r_1,G} \leftarrow \begin{cases} u_{ji,G+1}, & \text{if } f(u_{ji,G+1}) < f(x_{r_1,G}) \\ x_{r_1,G}, & \text{otherwise} \end{cases}$$

After each generation, it is hoped to get better candidate solutions which are closer to the global minimum of the selected function.

Algorithm 1: DEA

```
Result: Global minimum of $f(X)$ has been approximated
parsing;
initialization;
while current number of runs $< N$ do
    while the criterion is not met do
        for each point $P$ do
            randomly pick $Target$, $P_1$, $P_2$;
            $Mutant$ = mutate($Target$, $P_1$, $P_2$);
            $Trial$ = recombine($Mutant$, $P$);
            $P$ = select($Trial$, $P$);
        end
    end
end
```

1.2 Genetic Approach

Genetic algorithm (GA) is a metaheuristic that has the same goal as DEA, however, the way GA operates is a bit different than the DEA. The difference between these approaches is mostly due to recombination and mutation stages. The way this algorithm does the recombination is exchanging some genes between the fittest pair of chromosomes and mutation is to insert or change the recombined chromosome to make sure that the candidate solutions are not get stuck in the local interval. Although it is not guaranteed that satisfactory solution will be obtained after n generations, the algorithm tries to converge after each generation which gives us the hope that eventually, the final solution will satisfy the predefined threshold value.

Note: The genetic algorithm can also be used to try to find the global maximum/minimum or even specific point of an arbitrary dimensional function by using the same tricks as used in the DEA.

Algorithm. The genetic algorithm works with randomly generated population(also known as chromosomes). The algorithm is mainly based on iteratively using a pair chromosomes which have the highest fitness value to obtain hopefully a better population. So, two fittest points(called parents P_1 and P_2) are picked from the current generation and a new point called an **offspring** is obtained by exchanging some genes between the parents.

$$v_{ji,G+1} = \begin{cases} x^{(1)}_{ji,G}, & \text{if rand}(0,1) < P_{\text{crossover}} \\ x^{(2)}_{ji,G}, & \text{otherwise} \end{cases}$$

After the process of recombination, we mutate the **offsprings** according to the given probability of mutation to get a little bit of diversity.

$$u_{ji,G+1} = \begin{cases} v_{ji,G+1}, & \text{if rand}() \geq P_{\text{mutation}} \\ z_{random}(\in \mathbb{R}), & \text{otherwise} \end{cases}$$

$$\text{where } j = 1, 2, ..., D$$

The third stage makes sure that all newly obtained chromosomes or points replace their parents and the fitness value of each individual is calculated for the next generation.

$$x^{(i)}_G \leftarrow u^{(i)}_{G+1}$$

After each generation, it is hoped to get better candidate solutions which are closer to the global minimum of the selected function.

Algorithm 2: GA

```
Result: Global minimum of $f(X)$ has been approximated
parsing;
initialization;
while current number of runs $< N$ do
    while the criterion is not met do
        calculate the fitness of all chromosomes(points);
        for each fittest pair ($P_1$, $P_2$) in sorted chromosomes do
            Offsprings = recombine($P_1$, $P_2$);
            Mutants = mutate(Offsprings);
            replace the pair ($P_1$, $P_2$) with Mutants;
        end
    end
end
```

1.3 A list of the main functions

- **parse_arguments(...)** - Parsing the arguments
- **get_random_inputs(...)** - Getting random inputs in the search space (for DEA)
- **sort_for_fittest(...)** - Sorts the search space according to outputs of the fitness function (for GA)
- **mutate(...)** - Mutation by 3 randomly picked agents (DEA); Mutation of offsprings (GA)
- **crossover(...)** - Recombination of parent and mutant agents (DEA); Recombination of two parents to obtain offsprings (GA)
- **select(...)** - Selection of the best candidate
- **visualize(...)** - Illustrating the process

1.4 The different users and their characteristics

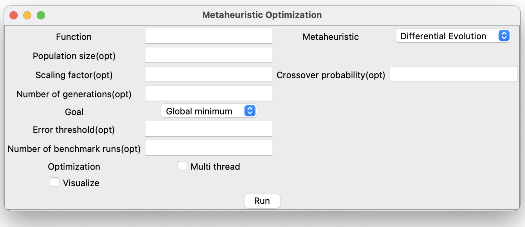
There are not several types of users, and in fact, all users share the same characteristics. They can manipulate the same system properties, and to do so, they can change several arguments which are shown below:

- Function with an arbitrary dimension (**-F [function] -V [variables]**)
- Number of agents or population size (optional, default: 10) [**--population=10**]
- Hyper parameters
  - [**DEA**] F and $P_{\text{crossover}}$ (optional, default: $F = 0.8$, $P_{\text{crossover}} = 0.9$) [**-f 0.8 -p 0.9**]
  - [**GA**] P_{mutation}, $P_{\text{crossover}}$ and *Elitism* (optional, default: $P_{\text{mutation}} = 0.1$, $P_{\text{crossover}} = 0.5$, *elitism* is turned off) [**-f 0.1 -p 0.5**]
- Number of generations (optional, default: 1000) [**--generation=1000**]
- Approximation point (optional, default: custom point) []
- Error threshold (optional, default: -1) [**--threshold=-1**]
- Number of benchmark runs (optional, default: 1) [**--benchmark-run=1**]
- Verbose run (optional, default: false) [**--verbose**]
- Visualization (optional, default: false) [**--visual**]
- Optimization (optional, default: [single-thread, manual]) [**-o0**]

Any user can adjust the settings however (s)he wants. This parameters can be modified from both command line and graphical user interface.

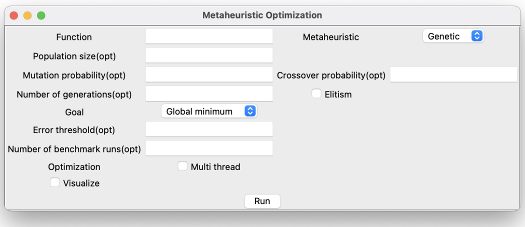
2 Preliminary design

The preliminary design of user interface is shown in below pictures. The GUI(Graphical User Interface) tends to be very intuitive for the user as much as possible.



Metaheuristic: Differential Evolution
Goal: Global minimum

(a) DEA


Metaheuristic Optimization
Function
Population size(opt)
Mutation probability(opt)
Number of generations(opt)
Goal
Error threshold(opt)
Number of benchmark runs(opt)
Optimization
Visualize
Multi thread
Metaheuristic
Crossover probability(opt)
Elitism
Run

Metaheuristic: Genetic
Goal: Global minimum

(b) GA

Figure 1: Mock up diagrams (demo)

2.1 Packages and external signatures

Some external packages have been used in this project to make its different parts up and running. Most of these packages and libraries are well known C++ and Python libraries such as *vector*, *set*, *matplotlib*, *tkinter* and so on. There is only one external library(**expreval**) that has been used in the project to be able to calculate any mathematical function(as long as expressed with pre-defined mathematical operators which are mentioned in this document via the external link) and to give the user freedom to optimize any arbitrary function that he/she wants. Therefore, a brief breakdown about what and how this library works is provided in the next paragraph.

ExprEval library. The library has been implemented in C++ and is used to calculate mathematical functions and expressions. Throwing exceptions and handling internal errors are the responsibility of this library in case of ill-formed expressions. It has a very simple API, so that the main project does not have difficulties linking and using this library.

In a mathematical expression, all operators have their own priorities to indicate the order of calculation. It means, that the priorities tell us which operator has to be handled before than the other. Here is the list of operators and their corresponding priorities which have been defined and are usable in this project.

Operator	Priority
$+$	9
$-$	9
$\times$	8
/	8
$\wedge$	7
mod	7
ln/log2/log	6
sin/cos/tan	6
sinh/cosh/tanh	6
arcsin/arccos/arctan	6
arcsinh/arccosh/arctanh	6
round/ceil/floor	6
abs/rand/gamma	6
and/or/not/xor/shiftl/shiftr	6
pi/e/T/F	5
()	0

Table 1: Operator priority table

Note: More information about this library can be found in this document.

3 Detailed design

3.1 Different classes and association among them

There are several abstract concepts used in this project work such as *metaheuristics*, *population*, *agents* and so on. These concepts have concrete forms and associations among them. Each concept is described individually in the table below as well as its attributes and methods(if any).

Class	**Description**	**Attributes**	**Methods**
Agent	It can be a chromosome(made of genes) or a single point. In this context, it is more appropriate to think of it as a point.	Coordinates of a point in each axis of any arbitrary dimension (in a double format since the space is $\mathbb{R}$)	-
Population	A set of agents which have been uniformly randomly generated over the search space in $\mathbb{R}$	All generated agents	-
Metaheuristic algorithm	An algorithm or approach to optimize the given mathematical function in $\mathbb{R}$	Hyper-parameters for two optimization algorithms (DEA+GA)	Abstract methods for mutation, recombination(crossover) and selection
Differential Evolution Algorithm	One of the two metaheuristic algorithms which is inherited from *Metaheuristic*	Scaling factor and crossover probability	-(overrided the parent class)
Genetic Algorithm	The other metaheuristic algorithm which is inherited from *Metaheuristic*	Mutation and crossover probability (+elitism, optional)	-(overrided from the parent class)
Command Line Interface(CLI)	A command line tool for the user to run and interact with the program	Options to pass in	Running the metaheuristic algorithm with the given parameters
Graphical User Interface(GUI)	A graphical interface for the user to run and interact with the program; it tends to be more intuitive and therefore, user-friendly	The application window and widgets to be able to make the program interactive	Running the metaheuristic algorithm with the given parameters

Table 2: Different classes, their attributes and methods

The class diagram is used to describe all of the above mentioned details more visually and it is shown in the figure below.

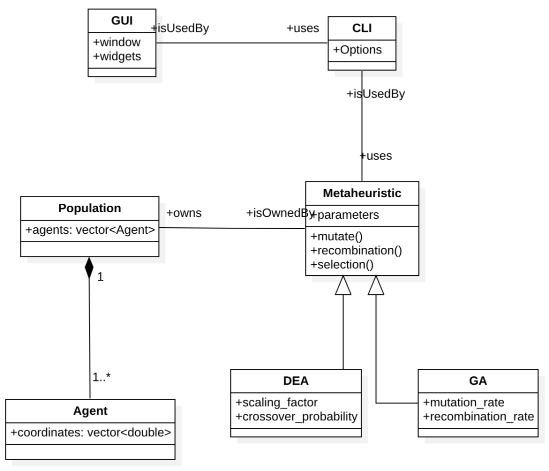


Figure 2: Class diagram

3.2 System interaction

The program takes a function and several arguments(check the section 1.4) from user. Then it needs to parse those arguments properly in order to set up the settings for the initial run. The settings have to be set automatically if the given arguments do not satisfy the preconditions(i.e. population size has to be an integer greater than 0), therefore, we need to have some default values for each parameter.

After setting up the parameters, the function given by the user is used and evaluated by the third-party library(a.k.a. **expreval**) and if the function fails to be evaluated then it returns **INVALID_FUNCTION** exit code. Due to the ill-formed function the system prints an error message to the terminal(”**Incorrect function!**” and some additional information about the function) and waits for a new valid user input. If the function is correct then one of the metaheuristics (*DEA*/*GA*) is invoked with the initialized settings(i.e. number of generations, population size, scaling factor, crossover probability and so on).

The user parameters are passed to the core program via the interface(the shell or/and python program). The program runs the selected metaheuristic optimization algorithm(DEA or GA) and it may store the history of evolution or the logs in a separate file in order to use it for the visualization purpose afterwards. The file is opened and written into only once per the program run regardless of the several running threads(if optimization is selected by the user) to avoid the race condition in the file stream. When the core program terminates the interface may run the visualizer which is another small program for a specific purpose(illustration of the selected algorithm in an action). The visualizer fetches the logs and handles the internal errors such as bad file access or non-visualizable data in case of higher dimensions(3+). If a problem occurs it prints out the error message and terminates immediately. Otherwise, it opens up a new window and displays the evolution of generations(not necessarily one-by-one).

Finally, some technical/numerical parts of the results are displayed in the terminal(i.e. number of generations and approximated function arguments) while the animations can be shown in a separate window(if specified). In case the given function has higher dimension than 3, the program does not plot the graph(and show the animation) and prints ”**Function has to be 2/3D!**” to the terminal.

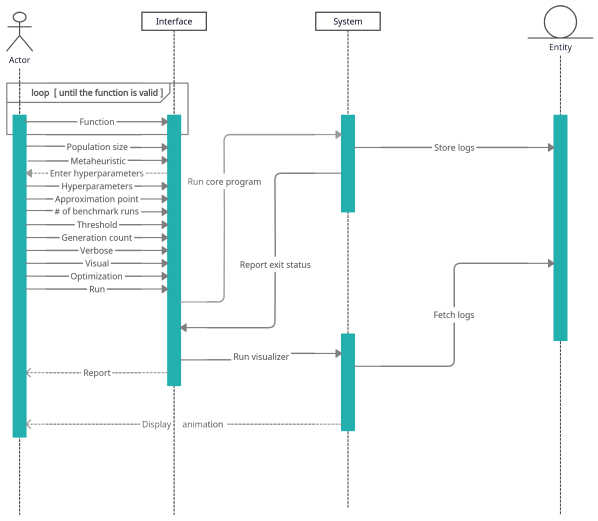


Figure 3: Interaction sequence diagram