Specification for Differential Evolution and Genetic 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.

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

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

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.

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.

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.

1.5 Software and hardware constraints

The project is implemented mainly in two different programming languages. The graphical user interface is coded in Python while the command line interface (or the core algorithm) is implemented in C++. For that, there are different libraries (both from C++ and Python) which are used in the project.

Programming language	Library	Version
	cmath (for usage of advanced mathematical functions)	
	vector (for dynamic memory)	
	string	
C++	**getopt** (for argument parsing)	
	set	9+ (GNU g++)
	random	
	fstream (for files)	
	omp (for parallelism)	4+
	expreval (for expression evaluation)	
	numpy	
	matplotlib (for animations and plotting)	
Python	**tkinter** (for GUI)	3+
	os	
	re (for regex)	
Shell	-	-

Table 1: Software requirements

All libraries shown above is required to build the project successfully. However, there is only one library(**expreval**) which has to be implemented due to the requirements(evaluation of strings in C++).

Note: Hardware requirements are not shown because to run this program, any computer of today's market is acceptable.

2 Detailed requirements

2.1 Functional specification

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).

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.

2.2 Interface specification

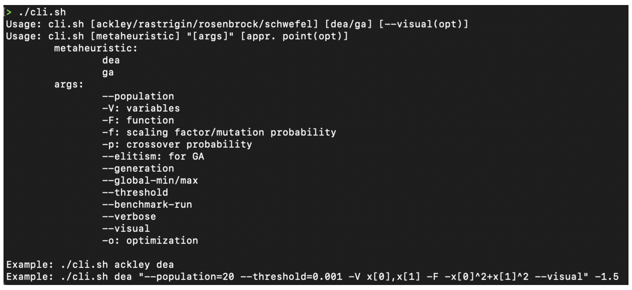
Running the program. A user can run the program by running either the shell script **cli.sh** or **metaheuristic-optimization.py** from terminal:

```
// for command line interface
./cli.sh [metaheuristic] "[args]" [appr. point(opt)]

// for graphical user interface
python metaheuristic-optimization.py
```

For further information about the usage of the program can also be obtained by running the **cli.sh** without any arguments or with the *help* option(check the output 1):

```
// About the program usage
./cli.sh
// or
./cli.sh help
```



```
> ./cli.sh
Usage: cli.sh [ackley/rastrigin/rosenbrock/schwefel] [dea/ga] [--visual(opt)]
Usage: cli.sh [metaheuristic] "[args]" [appr. point(opt)]
        metaheuristic:
                dea
                ga
        args:
                --population
                -V: variables
                -F: function
                -f: scaling factor/mutation probability
                -p: crossover probability
                --elitism: for GA
                --generation
                --global-min/max
                --threshold
                --benchmark-run
                --verbose
                --visual
                -o: optimization

Example: ./cli.sh ackley dea
Example: ./cli.sh dea "--population=20 --threshold=0.001 -V x[0],x[1] -F -x[0]^2+x[1]^2 --visual" -1.5
```

Figure 1: Output of the commands shown above

The [**function**] has to be given as a mathematical expression(i.e. 2^3). The user can use most of the mathematical functions included in the built-in **cmath** library(i.e. *sin()*, *cos()*, *gamma()*, *asinh()* and so on.). To use variables, the user should utilize a built-in variable vector called **x**.

For example,

Mathematical function	**String representation**
$f(x, y, z) = x + (yz)^2$	$x[0] + (x[1] * x[2]) \wedge 2$

Table 2: The function on the left side is passed to the program as the string(text) on the right side.

The function above can be passed to the program as shown below:

-V x[0],x[1],x[2] **-F** x[0]+(x[1]*x[2]) ∧ 2.

Note: All available mathematical operators and functions(which are supported by the expression evaluator engine - *expreval*) can be found in this link. Therefore, the user can use any of these operators or functions freely.

Population size. Population size is to indicate the number of total agents(points) and it has to be an integer greater than 0.

Hyper parameters.

[DEA]. If not given, scaling factor F and crossover probability P take the default values 0.8 and 0.9 respectively. In addition to this, F and P have to be in the intervals of $(0, 2]$ and $[0, 1]$ respectively.

[GA]. If not given, mutation probability(rate) P_{mutation} and crossover probability $P_{\text{crossover}}$ take the default values 0.1 and 0.5 respectively. In addition to this, these parameters have to be in the interval of $[0, 1]$ and to turn on *elitism* you can give the option [--elitism].

Number of generations. This parameter indicates the termination point for the program and therefore, has to be greater than 0. If not given the program checks if the **error threshold** has been provided and it terminates according to the error threshold(halting when the approximated solution has an error less than or equal to the threshold), if given, and it sets the number of generations to default value 1000, otherwise.

Approximation point. This parameter is to indicate the goal of the program. Since the algorithm is capable of finding global minimum, we can use to to approximate to any point of a function. There are 2 options which can be used:

- --global-min
- --global-max

If none of the options above is given then the program asks for a custom/specific point. This point indicates what value the program is going to approximate to.

Error threshold. This value indicates maximum acceptable error in the final solution. However, if the approximation point is chosen as global min/max then such threshold cannot be given(since there would not be any ground truth to compare the program's results with) and therefore, the program needs to decide when to terminate according to the number of generations.

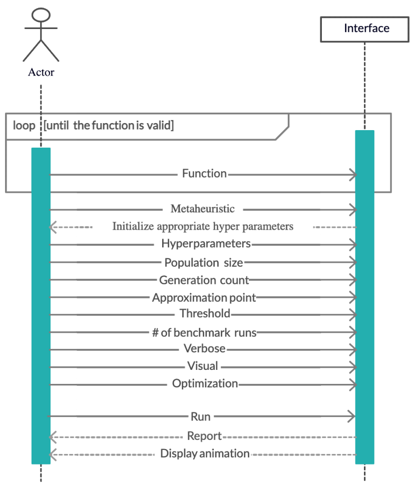
Number of benchmark runs. If you want to run the program several times to observe some statistics then this option can be used for such purpose.

Verbose log. If you are not satisfied with the final result and want to see some more details during the process then this option has to be set. It gives some additional information about each generation.

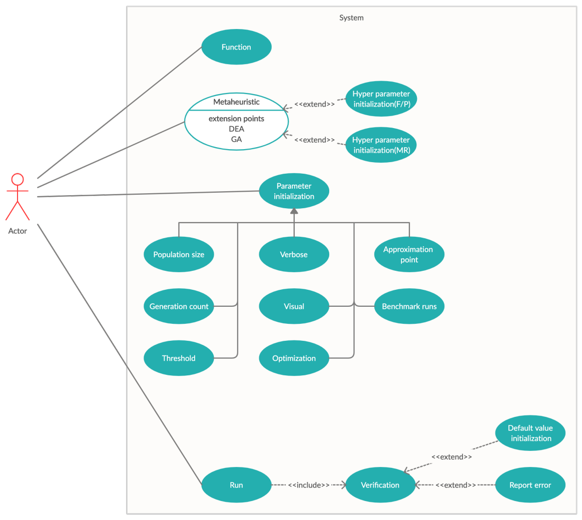
Visualization. To visualize the approximation process of DEA/GA you can use this option. However, this option works for only 2/3D functions.

Optimization. If you believe that the program is going to take so much time to terminate or you will not get even better result after the termination then you can use optimization flag. There are 4 options:

- -o0 : for no optimization (single threaded run, non-adaptive hyperparameters)
- -o1 : optimization level 1 (single threaded run, self-adaptive hyperparameters)
- -o2 : optimization level 2 (multi threaded run, non-adaptive hyperparameters)
- -o3 : optimization level 3 (multi threaded run, self-adaptive hyperparameters)



(a) System sequence



(b) Use case

Figure 2: Diagrams

2.3 Performance and safety

The performance of the program is heavily dependent on the given function as well as the hyper-parameters. The program should have the ability to use CPU cores in order to achieve better overall performance. To do that, OpenMP can be used to parallelize the benchmark runs since each run is independent from any other run.

Note: By using only 10-12 CPU cores at the speed of 2.3GHZ we were able to decrease the run time about 10 times than before.

Some safety cases have been developed in order to maintain the system when there is an issue in the program. However, there are some cases that the system immediately shuts down and this happens when there is either no way or no meaning to keep the program up and running(i.e. what if the log file is not accessible to visualize behaviour of the chosen algorithm?). There are also some other cases when the program can continue to run even if some internal states are not satisfactory. This type of situation may occur when the user enters an invalid function or inappropriate argument which does not satisfy the precondition(s).

Errors	**Description**	**Handling**
STATUS_INVALID_FUNCTION	Ill-formed function	User has to provide properly formed mathematical function
STATUS_WRONG_ARGUMENT	The given argument(s) does not satisfy the preconditions	Invalid argument(s) is replaced by a default value(s)

Table 3: User errors

Errors	**Description**	**Handling**
MEMORY_ALLOCATION	Bad (dynamic) memory allocation	-
FILE_ACCESS	File does not exist or cannot be opened	Immediate program termination

Table 4: System errors