MECHTRON 2MP3: Programming for Mechatronics

Developing a Basic Genetic Optimization Algorithm in C

Ethan Otteson, 400433686

1 Introduction

A genetic algorithm was developed in C in order to find the global solution to the Ackley function. This was done by studying and translating a similar program created in Python. This model serves as a general algorithm which can solve other optimization functions, including the Rastrigin, Beale, and Easom functions which were tested to ensure the model was working correctly. Additionally, the model is able to optimize more than two variables as long as the corresponding hard coded variables are changed including the `NUM_VARIABLES`, `Lbound[]`, and `Ubound[]`. For each run, the user specifies the population size, max generation, crossover rate, mutation rate, and stop criteria they would like to use. Below the results are highlighted, the implementation of the code is discussed, the make file used is explained, and improvements to the program are discussed.

2 Programming Genetic Algorithm

2.1 Implementation (12 points)

Files you have downloaded are:

- `OF.c` (Do not change anything in it)
- `functions.h` (Do not change anything in it)
- `GA.c`
- `functions.c`

```
#include <math.h>
#include "functions.h"

double Objective_function(int NUM_VARIABLES, double x[
    NUM_VARIABLES])
{
```

```
    double sum1 = 0.0, sum2 = 0.0;
    for (int i = 0; i < NUM_VARIABLES; i++)
    {
     sum1 += x[i] * x[i];
     sum2 += cos(2.0 * M_PI * x[i]);
    }
    return -20.0 * exp(-0.2 * sqrt(sum1 / NUM_VARIABLES)) - exp(
       sum2 / NUM_VARIABLES) + 20.0 + M_E;
   }
```

The number of decision variables in the optimization problem is set to 2, with upper and lower bounds for both decision variables set to +5 and -5, respectively:

```
int NUM_VARIABLES = 2;
double Lbound[] = {-5.0, -5.0};
double Ubound[] = {5.0, 5.0};
```

```
Genetic Algorithm is initiated.
-------------------------------------------------
The number of variables: 2
Lower bounds: [-5.000000, -5.000000]
Upper bounds: [5.000000, 5.000000]

Population Size:   100
Max Generations:   10000
Crossover Rate:    0.500000
Mutation Rate:     0.100000
Stopping criteria: 0.0000000000000001

Results
-------------------------------------------------
CPU time: 1.143935 seconds
Best solution found: (-0.0001323239496775, -0.0000225231135396)
Best fitness: 0.0003801313729501
```

The original genetic algorithm tried to replicate the Python file given in C. With a stop criteria of zero, the results are shown in the table below.

Table 1: Results with Crossover Rate = 0.5, Mutation Rate = 0.05, and Stop Criteria = 0

Pop Size	Max Gen	Best Solution			CPU time (Sec)
		x_1	x_2	Fitness	
10	100	0.014261652256	-0.083113096227	0.417441789068	0.000000
100	100	-0.008276740558	-0.022906812384	0.084615586491	0.001950
1000	100	-0.000166881363	0.002803222743	0.008152724062	0.037665
10000	100	0.000780582883	-0.000268567819	0.000060347374	2.764599
1000	1000	-0.000108832027	0.000040542799	0.000328848179	0.369077
1000	10000	0.000028514768	0.000020943116	0.000100101554	3.599629
1000	100000	0.000000030267	0.000001175794	0.000003326788	36.834155
1000	1000000	-0.000000002328	-0.000000165309	0.000000467613	402.547965

Table 2: Results with Crossover Rate = 0.5, Mutation Rate = 0.2, and Stop Criteria = 0

Pop Size	Max Gen	Best Solution			CPU time (Sec)
		x_1	x_2	Fitness	
10	100	0.044109614120	0.015535417485	0.189550153589	0.000000
100	100	-0.006178442857	-0.002185970080	0.019680159825	0.001393
1000	100	0.000319427345	-0.000026787165	0.000909384348	0.042084
10000	100	-0.000045227352	0.000033534597	0.000159334724	2.779461
1000	1000	0.000002884771	-0.000008030328	0.000024136244	0.391559
1000	10000	0.000045052450	0.000002563465	-0.000015718396	3.787544
1000	100000	0.000000877771	0.000000477302	0.000002826049	38.340442
1000	1000000	-0.000000006984	0.000000048894	0.000000139698	447.637323

However, once a stop criteria is implemented as in the next table it is important to note how this functions. The provided Python code did not include a clear representation as to how the stop criteria was handled as I believe this was done within a library not available in C. This genetic algorithm handles the stop criteria in lines 141-153 of the `GA.c` file. It works by counting the number of generations that have elapsed without producing a better best solution. A better best solution is defined as a solution that has a fitness change greater than the specified stop criteria. This means that a generation whose best individual's fitness is less than the specified stop criteria better than the best recorded solution or a generation whose best individual's fitness does not improve at all or gets worse will be counted as a generation with no change. Previously, the program only required one generation like this before stopping, however, this produced poor accuracy as the algorithm would stop too soon. To fix this it was implemented that 10 generations in a row without significant improvement must elapse before the program will stop early. 10 was chosen as it met a good balance between accuracy and time not stopping too early as to sacrifice accuracy while also not wasting time. Additionally, 100 was also a good value and seemed to almost exactly increase the time by 10 times while decreasing the best solution and best fitness by about 10 times. Ultimately, 10 was chosen as it was deemed to produce an acceptable level of accuracy in a much shorter time. The results involving this stop criteria are shown in the table below.

Table 3: Results with Crossover Rate = 0.5, Mutation Rate = 0.2, and Stop Criteria 1e-16

Pop Size	Max Gen	Best Solution			CPU time (Sec)	Gens Elapsed
		x_1	x_2	Fitness		
10	100	0.032107441	-0.016534470	0.136555946	0.000000	26
100	100	0.015212830	-0.001576982	0.049475859	0.000000	26
1000	100	-0.002512440	-0.000505975	0.007423823	0.003940	23
10000	100	0.000026880	-0.000104473	0.000305429	0.572999	21
1000	1000	-0.000563699	0.001957875	0.005873192	0.025086	46
1000	10000	-0.000531917	0.001605621	0.004860284	0.005670	32
1000	100000	0.000076240	-0.000816269	0.002336703	0.007425	30
1000	1000000	-0.000233950	0.000893815	0.002635988	0.010758	32

2.2 Report and `Makefile` (3 points)

Analyzing the tables above shows us some interesting results. First, look at the first two tables prior to the addition of a stop criteria. Increasing the population by X10 seems to increase the CPU time taken by about X100 while increasing the maximum generation by X10 only increased the CPU time taken by X10. This is an increasing result and suggests that population size is used more frequently in something like a for loop compared to the maximum generation which is true. Although, this is not to say there is not a balance between these two variables. There is also little change between the two tables with very different crossover and mutation rates, this suggests to me that there is a lot of work that could be done to improve these functions and increase the effect they have on our results. We also see in all the tables that the first few results register a CPU time of 0 seconds; I believe this is a limitation within these C functions and their level of precision.

Comparing the third table to the other two shows some increasing effects about the inclusion of the stop condition. We can see for the four tests which have population size changing that this stop condition has a very positive effect, as in a fraction of the time of the other scenarios a result is achieved that is within the margin of error between different runs. This suggests that the stop condition has a very positive effect on our results when only changing population size as we are not wasting time on the last 70-80 generations which are not producing significantly better results. However, this conclusion does not hold true for the other four test cases where only maximum generation is being changed. Instead, these results do not change between tests which is to be expected as adding more possible generations does not matter if we never get to them. This suggests to me that a change to how the stop condition is handled may be useful. For instance, having the no change counter mentioned above as a function of maximum generations (such as `no_change_limit = MAX_GENERATIONS / 10`) would still give the advantage of not needing extra generations when the population is big enough but would also not cause the obstacle we are currently experiencing as maximum generations increase.

The makefile is used in order to simply compile the genetic algorithm with one command as without it one would have to compile each file separately because of the multi-file dependency. The makefile I created and used for this assignment is shown and explained below.

```
CC = gcc
CFLAGS = -Wall -Wextra -g -O3 -pg
TARGET = GA
SRC = GA.c functions.c OF.c
OBJ = $(SRC:.c=.o)

$(TARGET): $(OBJ)
    $(CC) $(CFLAGS) -o $(TARGET) $(OBJ) -lm

%.o: %.c
    $(CC) $(CFLAGS) -c $<

clean:
 rm -f $(OBJ) $(TARGET)
```

- The first line defines 'gcc' to be used as the C compiler.
- The second line sets the flags which will be used to compile each file. `-Wall` and `-Wextra` enable most warning messages. While `-g` and `-pg` generate debugging and extra information so that the gdb and gprof tools can be used, respectively. `-O3` enables a high level of optimization to improve run time.
- The third line specifies the name of the generated executable.
- The fourth line specifies all of the source files.
- The fifth line creates a list of object files replacing the .c extension with a .o extension for each of the files defined in the previous line
- The sixth line gives rules as to how to build the target executable from the object files.
- The seventh line is the exact rules that need to be followed when building the executable. Using the specified compiler and compiler flags create the target output file using the list of object files. Additionally, I added the flag for the math library here as when I included it in the list of flags defined above, I would receive an error that certain elementary math functions were not defined.
- The eighth line is the generic rule for building the object files from their source file.

- The ninth line is the exact generic rule for the line above. Use the compiler and flags defined above, and "$ < " allows the use of automatic variables for each source file.
- The final couple of lines are used to remove all the files created by the make file, if the command 'make clean' is run in the terminal it will automatically remove all of the object and target files created.

2.3 Improving the Performance - Bonus (+1 points)

I intended to do this by modifying the crossover function to take into account each parent's fitness and potentially introduce elitism, as gprof indicated that crossover was taking up a large percentage of CPU time. However, due to time constraints, this was not possible.

2.4 Bonus (+3 points) - Only the fastest program!

3 Appendix