I. Optimization of the FLC

A. Membership Functions

The first step to optimizing the membership functions is to take the already existing membership functions from the FLC for both inputs and output. Each membership function is a gene in the chromosome, with it being represented by 2 or 3 bits to determine the size and shape of the function [9]. For the Gaussian functions, it’s 2 bits, 3 for the sigmoid and 3 for the triangular. All these genes are added together as one long chromosome, which in the case of the FLC already implemented its 39 genes of 80 bits. 2 for the tactile, 8 for each ultrasonic for 48 in total, 6 for the GPS, and 12 for each wheel for 24 in total.

For each of the bits within a gene, they are converted into an 8-bit floating point binary, with the first bit being the sign (positive or negative), with 4 bits being for the number before the decimal point (maximum of 15) and the 3 remaining bits are for after the decimal point (precision of 1 1/8th). This would make the new length of the chromosome 640 bits with 39 genes. The chromosome is split into partitions of what sensor that gene belongs to.

The maximum value for the system is 10, so only 4 bits were needed to represent that, however, if larger numbers are needed then more bits are required. This is also the case for the precision of the binary, as its only 1/8th which for this system is fine but if greater precision is needed then a new format is needed. A suitable replacement would be using IEEE 754 single or double-precision formats which use 32 bits and 64 bits respectively. These two formats would allow for both a much greater precision and a much higher number than the maximum of 15, however, this format would also need a lot more bits and in turn higher computational requirements.

With one chromosome being the configuration of the membership functions, a population is randomised. The raw sensor data from the dataset is used to evaluate the membership functions of the configuration, with it also using the rules already implemented to evaluate against the actual outcome, using the fitness function below [9] with Ci being the calculated output and Ai being the actual output.



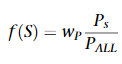
Once each chromosome is evaluated, a roulette wheel is used to select the parent chromosomes. Those parent chromosome swap genes with each other within the same respective sensor, to make sure genes from sensor 1 don’t swap with genes from sensor 2. In addition to the crossover operation, the mutation operator is also potentially run to mutate genes to ensure the chromosomes avoid local minimums. Once the offspring chromosomes are created, then they are evaluated like before and the entire GA process starts again until the termination criteria (i.e. maximum generations) is met.

B. Rule Set

For the creation of the rule set, the first step is creating a population of all feasible rules. A fuzzy partition is (grid showing the potential rules) created by taking the inputs and mapping them onto an input space with the raw data from the data set, in the case of the FLC it’s an 8-D dimensional space (8 input sensors). The input space is split into fuzzy subspaces by the membership functions of the inputs, creating a 2x46 subspaces with each subspace representing a different rule with a total of 8192 potential rules [8].

Each gene of the chromosome is then each potential rule totally to 8192 bits. Like mentioned before, the raw data from the data set is mapped onto the input space, with each piece given a classification to mirror the difference output states (i.e. reverse or forward). If a subspace doesn’t have any data within it, the rule is considered a dummy rile, as the outcome of the rule cannot be determined, and thus doesn’t affect the performance of the system. A dummy rule is given a value of 0 with it being unable to be changed from either crossover or mutation [8]. Any remaining rules are given either a value of 1 (the rule is used) and -1 (the rule isn’t used), and during the creation of the initial population, these genes are randomly given a value of either 1 or -1.

Once the initial population is created they are evaluated against the raw data, the data is run through the configuration of rules and tested against the actual input using the fitness function below. With Ps being the number of data classified correctly, Pall being the total number of data, and Wp is a weigh [8].



Once the chromosomes are evaluated, a roulette wheel is used to determine the parent chromosomes. Those chromosomes perform the crossover and mutation operators to create new offspring’s, a new configuration of rules. This process is repeated until the termination criteria are met.

As optimizing both the membership functions and the rule set required the other (i.e. membership function optimization requires the use of the ruleset), after optimizing the rule set the membership function optimization could be run again to optimize it to the new rule set, and the same could then be done with the optimization of the rule set. This process could be repeated, with each iteration improving the membership functions and rule set to a convergence.

C. Limiting Number of Rules

The fitness function from part I.B could be adjusted to take into account the total amount of rules used when it evaluates the chromosome. The new fitness function also takes away a score from the original fitness, with the score being the efficiency of the rule set [8]. With Ns being the number of rules used, Nall being the total number of rules, and Wn being a weight. As can be seen, the fitness function calculates the efficiency of the rule set by dividing the number of rules used by the total number of rules, this ensures that the chromosomes that use the least number of rules become the parent chromosomes and pass on their genes.



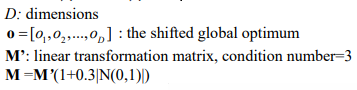
However, while this fitness function is effective at reducing the number of rules, it doesn’t have a limit to how many rules it can use. One potential solution is to make the weight (Wn) for the rule set much greater than the weight (Wp) for the classification so that the number of rules is reduced to the desired number, but this still wouldn’t fully limit it. Another solution is to use the fitness function from part I.B and adapt the encoding, crossover and mutation operators to limit the chromosome from setting more than the desired number of rules.

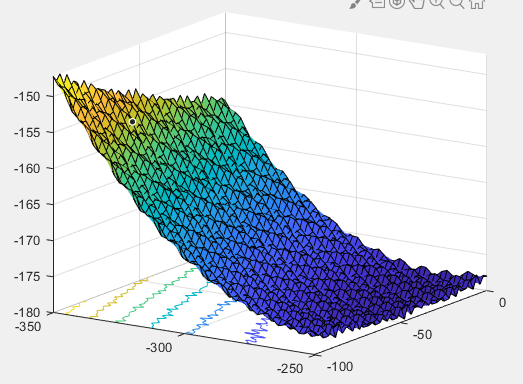
A final solution is to edit the fitness function above to take into account a limit to the number of rules. Instead of Ns/Nall, it can be changed to (Ns – D + 1)/Nall with D being the desired number of rules. This makes sure that chromosomes with a number of rules equal or less than the desired number gets a higher evaluation score and so are more likely to be chosen as parent chromosomes.

II. Compare different optimization techniques on CEC’2005 functions

A. Function 7: Shifted Rotated Griewank’s Function without Bounds

Function 7 is a shifted and rotated version of Griewank’s function, a function used in the testing of optimization created by Griewank in 1981. It is a multi-modal, non-separable and scalable function. The true minimum of this function is -180.





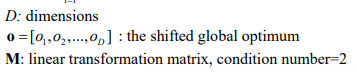
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Function 7 – 2 dimensions | | | | |
|  | Min | Max | Mean | STD |
| GA | 0.0013 | 0.0295 | 0.0118 | 0.0094 |
| PSO | 0 | 0.0373 | 0.0106 | 0.0123 |
| SA | 0.0271 | 0.4241 | 0.1105 | 0.1057 |

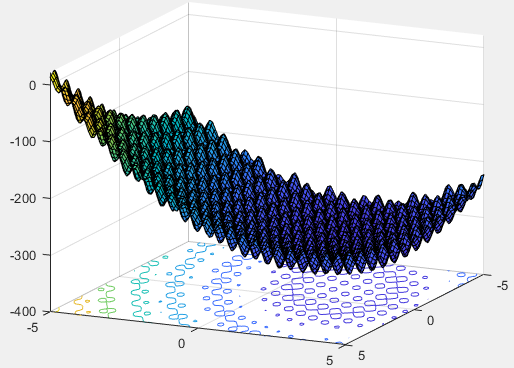
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Function 7 – 10 dimensions | | | | |
|  | Min | Max | Mean | STD |
| GA | 0.0708 | 0.888 | 0.3594 | 0.2361 |
| PSO | 0.0517 | 0.753 | 0.2445 | 0.2135 |
| SA | 0.0959 | 1.277 | 0.3383 | 0.3011 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Function 7 – 100 dimensions | | | | |
|  | Min | Max | Mean | STD |
| GA | 1.1395 | 1.2441 | 1.1946 | 0.0327 |
| PSO | 2.7e-4 | 0.0129 | 0.0045 | 0.0051 |
| SA | 16.9873 | 101.3529 | 58.6796 | 24.9142 |

B. Function 10: Shifted Rotated Rastrigin’s Function

Function 10 is a shifted and rotated version of Rastrigin’s function, a function used in the testing of optimization created by Rastrigin in 1974. It is a multi-modal, non-separable and scalable function. The true minimum of this function is -330





|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Function 10 – 2 dimensions | | | | |
|  | Min | Max | Mean | STD |
| GA | 0.0230 | 1.3747 | 0.8788 | 0.4354 |
| PSO | 0 | 0.995 | 0.2684 | 0.4536 |
| SA | 4e-10 | 1.9899 | 0.796 | 0.5578 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Function 10 – 10 dimensions | | | | |
|  | Min | Max | Mean | STD |
| GA | 17.0481 | 62.8302 | 35.2588 | 13.4606 |
| PSO | 9.9496 | 37.8083 | 18.9705 | 7.9815 |
| SA | 46.7629 | 146.258 | 87.5556 | 28.9195 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Function 10 – 100 dimensions | | | | |
|  | Min | Max | Mean | STD |
| GA | 1.1e3 | 2.2e3 | 1.6e3 | 314 |
| PSO | 1.3e3 | 1.1e4 | 4e3 | 2.87e3 |
| SA | 1.5e3 | 1.9e3 | 1.6e3 | 124 |

C. Conclusion

As seen in the results, the PSO has the best performance for both functions at all dimensions except for Function 10 at 100 dimensions which produced the worst results. Following that, simulated annealing’s performance varied between the different functions, with it being the second best on function 7 at dimensions 2 and 10 but terribly at dimension 100. For function 10, the results get worse for all three algorithms as more dimensions are added to it, the same is for function 7 but not as near of a variation between the dimensions.

III. 1-Dimension Bin Packing

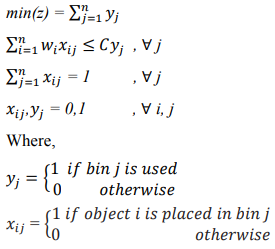
A. Encoding

For the bin packing problem, there are three different ways to represent a solution: bin-based, object-based, and group-based. In bin-based, each gene in the chromosome refers to which bin a brick belongs to. For example, a chromosome of 3765 means object 1 is in bin 3, object 2 is in bin 7 and so on [1]. The advantage is that the fixed length makes it suitable for genetic operators, however it generates redundant solutions.

In object-based, the objects placed together with the bins they belong to, with the chromosome being partitioned by the bins. For example, a chromosome of 12|34|5 means the objects 1 and 2 are in bin 1, objects 3 and 4 are in bin 2 and object 5 is in bin 3 [1]. The problem with this format is that the same solution can be represented in different ways, such as 34|5|12.

In group-based (proposed by Falkenauer [3]), the chromosome is split into objects which represent one bin, with each object having all the bricks belonging to that bin. For example, chromosome 1 3 5 2 3 means that object 1 is in bin 1, objects 2 and 5 are in bin 3, object 3 is in bin 5 and object 4 is in bin 2 [1]. An alternative way to represent it is to replace bin numbers with letters, so the chromosome would be ACEBC instead. The advantage to this is that the gene represents both objects and groups which allows the group part to be used in the genetic operations while the object part is used to identify the bricks in a bin. However, the problem with this format is that the chromosomes have a variable length, depending on how many bins there are. This is still, however, the best of the three if the issues of variable chromosome lengths can be overcome.

Below is the formula to the bin packing problem, the second line makes sure the size of the objects in a bin doesn’t exceed the limit of said bin. Line 3 makes sure that an object is placed in only one bin, so there are no duplicates.



B. Crossover Operator

Two chromosomes are chosen for the crossover, a gene is selected from each chromosome to be swapped with each other. The bins which contain the bricks in the new gene are deleted, and the bricks within those bins are moved to a free brick list. Then the new gene is added into the chromosome and the bricks in the free brick list and added to the chromosome [2].

C. Mutation Operator

If a chromosome is chosen to be mutated, a mutation size is randomized which determines how many bins from the chromosome is mutated. For each iteration of mutation size, a bin is randomly chosen and deleted with its bricks being added to a free list. After each iteration of mutation size, all the free bricks are added back to the chromosome.

D. Inversion Operator

The inversion operator is used to invert the chromosome, this is needed because the chromosome has undergone gene separation and rearrangement. This is used to place the gene in an order that helps its survival and efficiency, in the case of bin packing, the bins that are filled the most.

E. Fitness Function

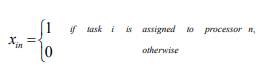
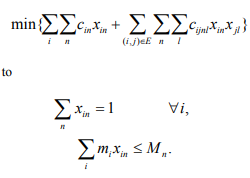
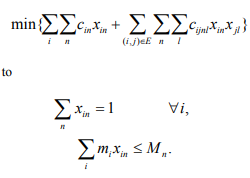
Falkenauer with Delchambre [4] also introduced a fitness evaluation function for the bin packing, with m being the number of bins used, Ii is the sum of the bricks in the bin i, and c is the height of the bin, with k being a heuristic exponential factor.



The above fitness function calculates the efficiency of each bin and takes a squared average of those efficiencies.

F. Limited Number of Bins

Now while the above solution is designed to minimise the number of bins, a simple solution is to restrict the number of bins to 3 or 4 and enforce that a solution cannot exceed this. With that, the algorithm would have to be changed so that it fits as many bricks in the limited number of bins, assesses that configuration like before, with the main difference being that some bricks are not included in the configuration. However, by limiting the number of bins to a fixed number it changes the problem from the normal bin packing problem to one that is more similar to the multiprocessor scheduling problem, and by using a fitness function designed for that, a better solution can be achieved, with the fitness function below being an example [5].



G. 2-Dimension Bin Packing

With the introduction of the extra dimension, the bricks need to be represented differently as some bricks can be placed next to each other. With that, the group-based representation is adapting so while each brick has the bin that it belongs to it also have the x and y position of the brick within the bin [6]. A possible extension is to include the angle of the brick, this would also allow for more bricks to be within a bin.

This 2-dimension bin packing design will share the same crossover and mutator operators as the 1-dimension version, with a similar fitness function. As the previous fitness function calculated the efficiency of the bin by taking the sum of the bricks and dividing it by the height of the bin, a similar thing can do done but instead with the volumes of the bricks and the volume of the bin.

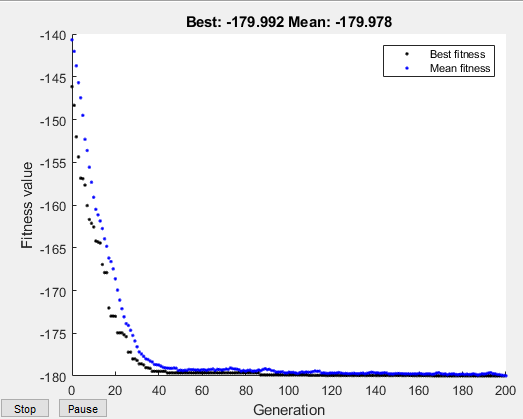
IV. References

1. Kaaouache, Mohamed & Bouamama, Sadok. (2015). Solving bin Packing Problem with a Hybrid Genetic Algorithm for VM Placement in Cloud. Procedia Computer Science.
2. F. Luo, I. D. Scherson and J. Fuentes, "A Novel Genetic Algorithm for Bin Packing Problem in jMetal," 2017 IEEE International Conference on Cognitive Computing (ICCC), Honolulu, HI, 2017, pp. 17-23.
3. Falkenauer, E. (1996). A hybrid grouping genetic algorithm for bin packing. Journal of Heuristics, 2(1), 5–30
4. Falkenauer E, Delchambre A. A genetic algorithm for bin packing and line balancing. Proceedings of the IEEE 1992 international conference on robotics and automation (RA92), May 10–15, Nice, France, 1992
5. Mehrabi, Abbas & Saeed, Mehrabi & Mehrabi, Ali. (2009). An Adaptive Genetic Algorithm for Multiprocessor Task Assignment Problem with Limited Memory. Lecture Notes in Engineering and Computer Science. 2179.
6. Ponce-Perez, A & Perez-Garcia, A & Ayala, Victor. (2005). Bin-Packing Using Genetic Algorithms. 311 - 314.
7. Suganthan, P. N. & Hansen N. & Liang J. J. & Deb K. & Chen Y. P. & Auger A. & Tiwari S. (2005). Problem Definitions and Evaluation Criteria for the CEC 2005 Special Session on Real-Parameter Optimization.
8. Negnevitsky, M., 2002. Artificial Intelligence: A Guide to Intelligent Systems. 1st ed. Harlow: Pearson Education Limited.
9. Algabri, Mohammed & Mathkour, Hassan & Hedjar, Ramdane & Alsulaiman, Mansour. (2015). Comparative study of soft computing techniques for mobile robot navigation in an unknown environment. Computers in Human Behavior

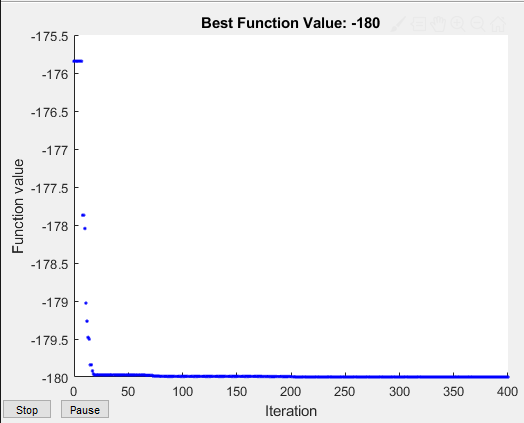
V. Appendix

A. Function 7, 2-D

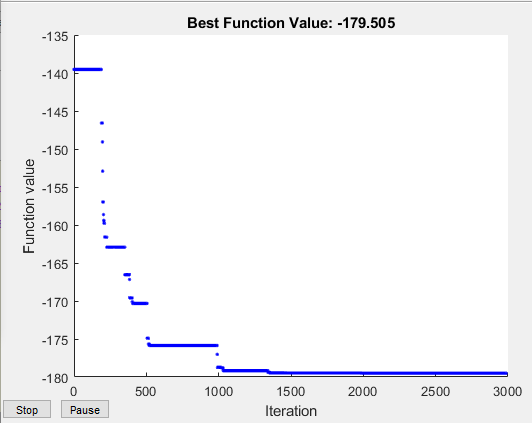
Genetic Algorithm



Particle Swarm

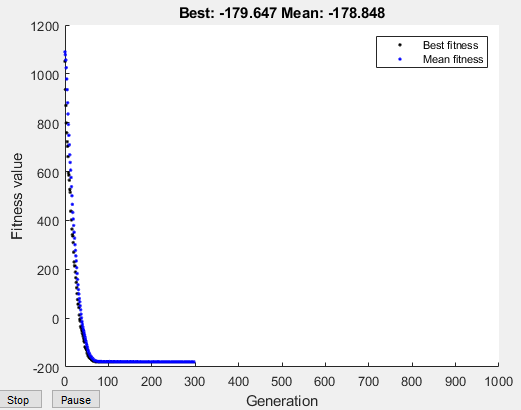


Simulated Annealing

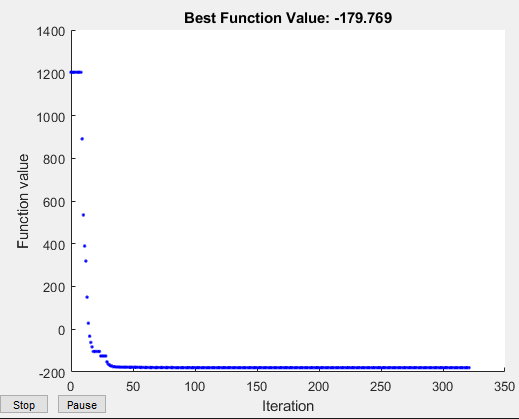


B. Function 7, 10-D

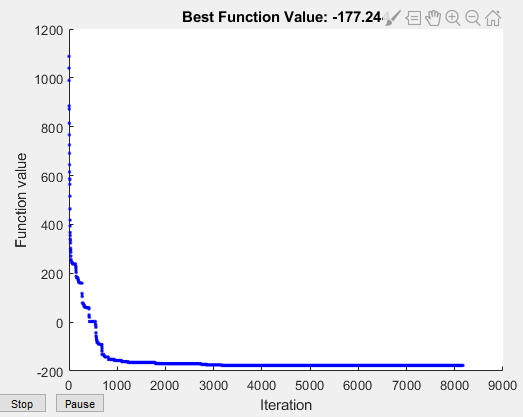
Genetic Algorithm



Particle Swarm

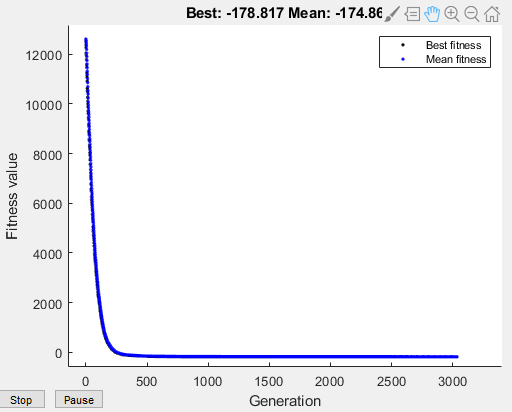


Simulated Annealing

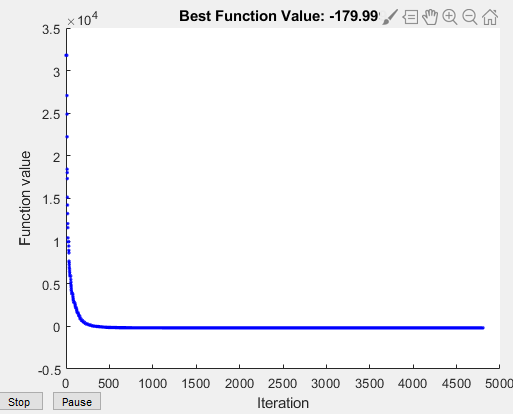


C. Function 7, 100-D

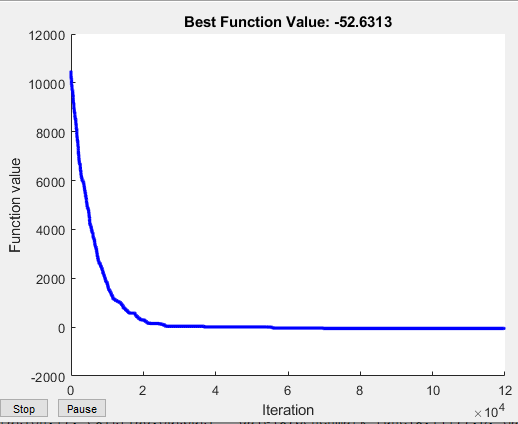
Genetic Algorithm



Particle Swarm

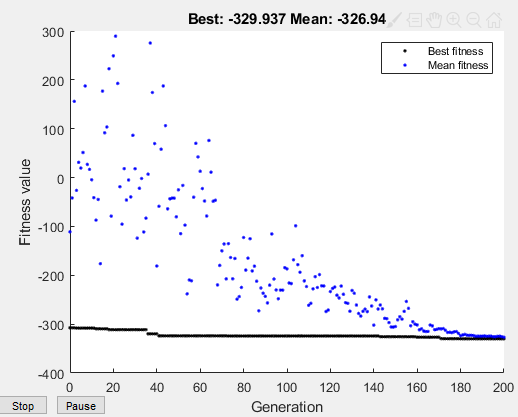


Simulated Annealing

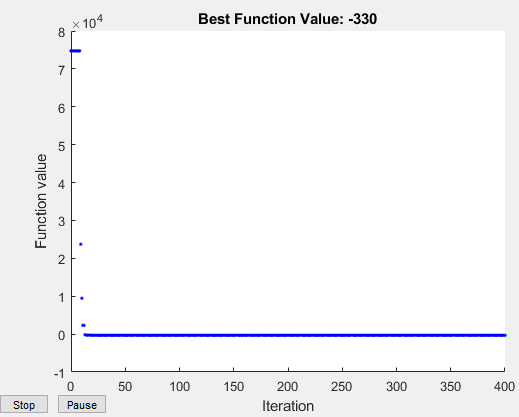


D. Function 10, 2-D

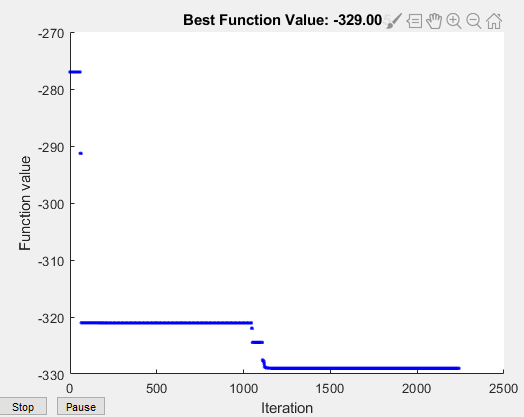
Genetic Algorithm



Particle Swarm

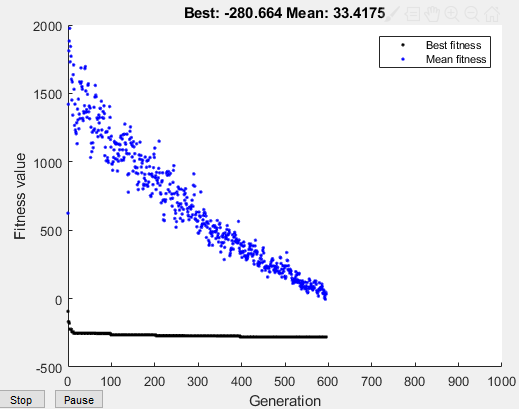


Simulated Annealing

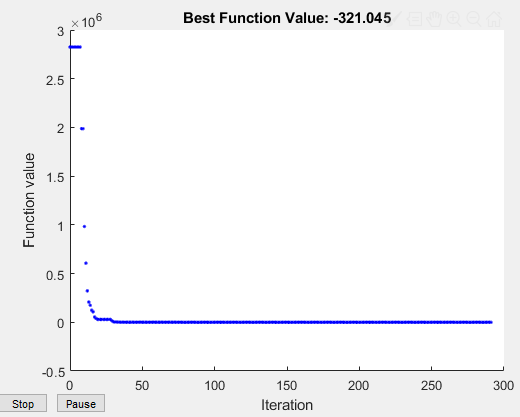


E. Function 10, 10-D

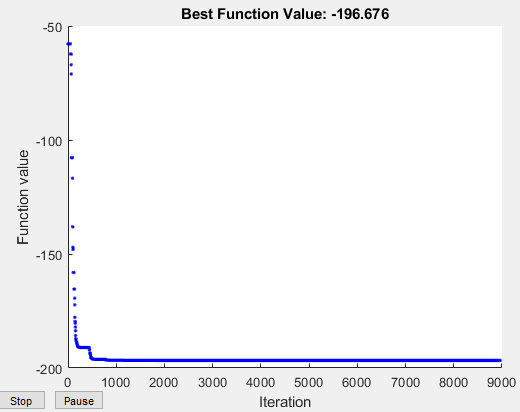
Genetic Algorithm



Particle Swarm

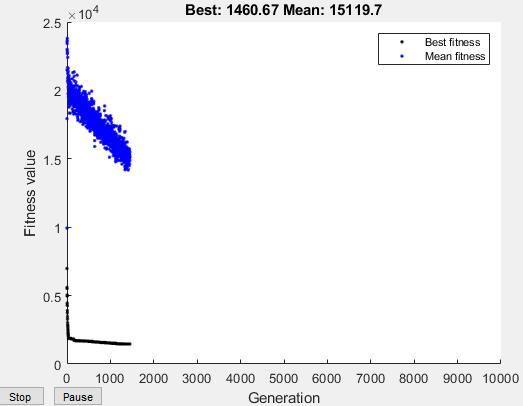


Simulated Annealing

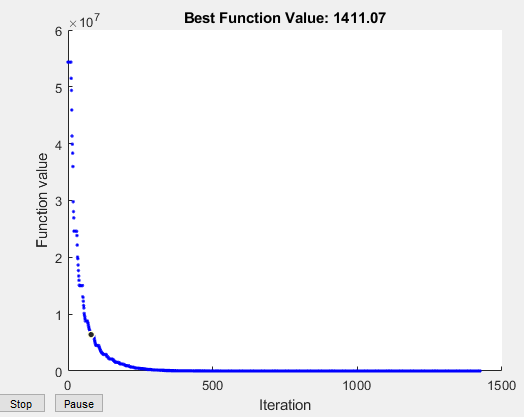


F. Function 10, 100-D

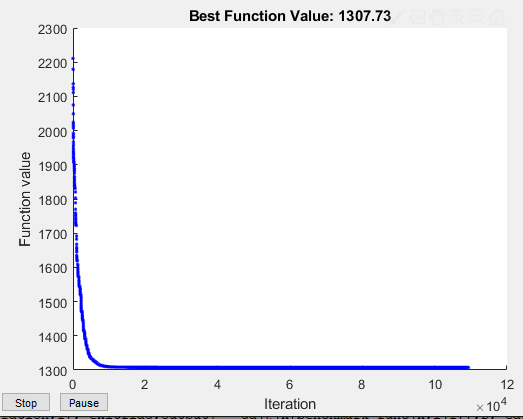
Genetic Algorithm



Particle Swarm

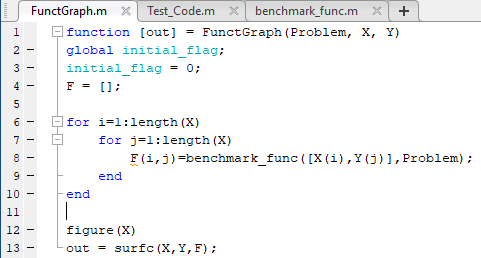


Simulated Annealing

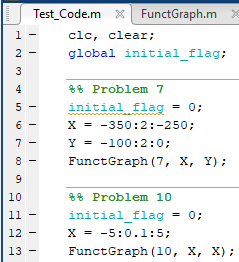


G. MatLab Code

FunctGraph.m plots the functions



Test\_Code.m is the main piece of code, which takes the two functions, runs them through three different algorithms for dimensions of 2, 10, and 100. This section calls the FunctGraph function to plot the functions.



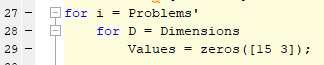
This section of code sets up the options for the algorithms. For the genetic algorithm, the maxStallGenerations is increased from 50 to 150 and CrossoverFraction is increased from 0.8 to 0.9. For the particle swarm, the number of maxStallIterations is increased from 20 to 150. For the Simulated Annealing, its combined with pattern search, which runs during the simulated annealing to help increase its performance.



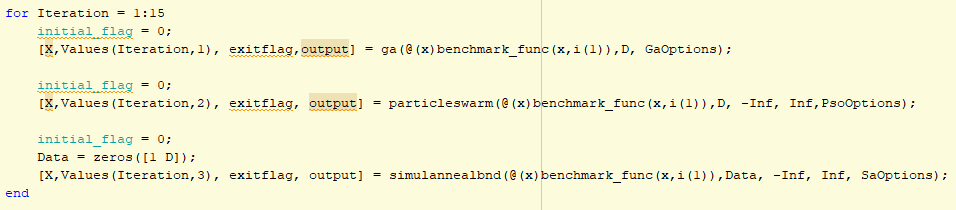
This section defines which functions are to be used and what dimensions to be used.



For loops are set up to automatically run through the different function at the different dimensions



Runs all three algorithms 15 times for each function and for each dimension, whilst recording the value of the algorithm



Takes the values from the algorithms and compares them against the true minimum for that function. It takes the min, max, the mean and standard deviation of those values.

