Victoria Hektor
Evolutionary Computing
AI MSc 2024

Evolutionary Computing Final Project

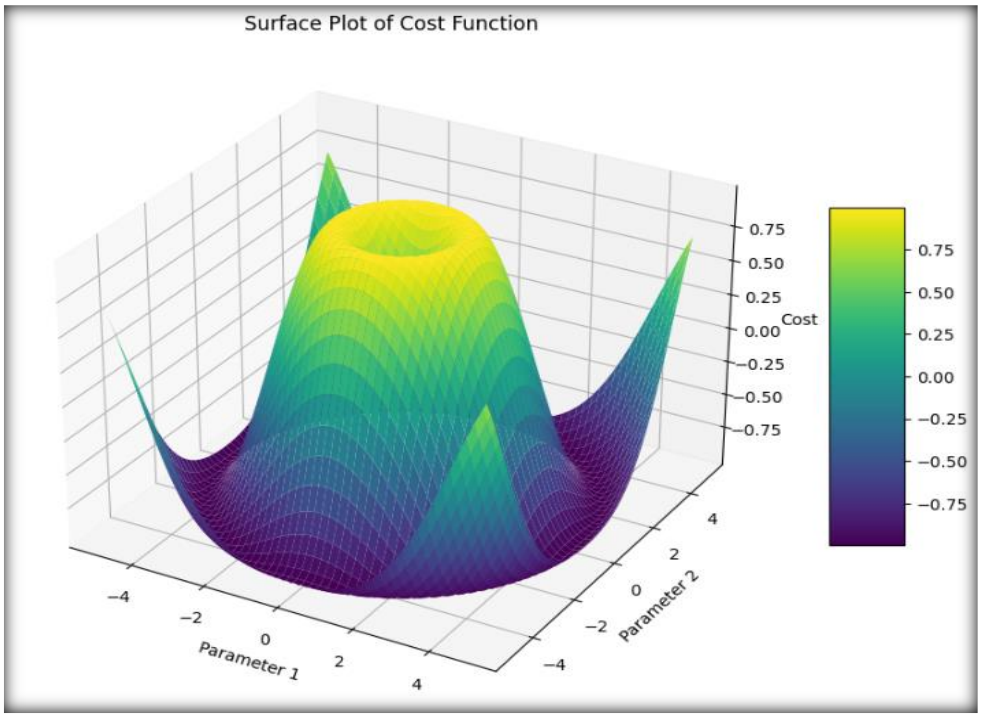


Figure.1. – Surface Plot Of The Fitness Landscape For Differential Evolution

Submission 4 of Evolutionary Computing Unit
Report: Pages 7-26
Appendix: Pages 27-51
References: Pages 52-56
Code: Pages 57-115
Report Word Count: 6528

Table of Contents

Evolutionary Computing Final Project 1
Table of Contents 2
Table of Figures 3
1 - Introduction – Unconstrained Ackley Problem With RGA, PSO, & DE 7
1.2 - RGA & Ackley 7
1. 2.1.- Parameter Tuning 8
1.3 – PSO & Ackley 9
1.3.1 – Individual Parameter Run Analysis 10
1.4 - DE & Ackley 10
1.4.1 – DE/1/Best/Bin 10
1.4.2 – DE/2/Best/Bin 12
1.4.3 - Comparison with de/1/best/bin: 12
1.4.4 – Final Analysis Decision: 13
1.5 – Comparing & Analysing Unconstrained Problems 13
2 - Introduction – Constrained Himmelblau Problem With RGA, PSO, & DE 14
2.1 - Implementation Preparations 14
2.2 - Constrained Himmelblau RGA 14
2.3 - Constrained Himmelblau PSO 15
2.3.1. - Further Parameter Tuning Analysis 17
2.4 - Constrained Himmelblau & DE 20
2.4.1 - DE/1/Best/bin 20
2.4.2 - DE/2/best/bin 22
2.4.3. – DE 1 & 2 Comparison 23
3. - Comparison, Interpretation and Conclusions 23
3.1 – Comparison & Interpretation 23
3.1.1 – Most Efficient Comparison 24
3.1.2 – Changing The Constraints Parameters 25
3.1.3 – Changing the tolerance 25
3.2 - Conclusion 26
Appendix 27
Unconstrained Ackley Function 27
RGA & Ackley 27
PSO & Ackley 32

DE & Ackley 33
Constrained Himmelblau Function With RGA, PSO, & DE 36
Constrained Himmelblau Maths: 37
RGA & Constrained Himmelblau 37
PSO & Constrained Himmelblau 39
DE & Unconstrained Himmelblau 46
Additional Appendices – Constrained g06, g08, Welded Beam 48
Constrained Welded Beam 48
Constrained g06 50
Constrained g08 51
References 52
Code 57
Unconstrained Ackley Function Code 57
Ackley & RGA Code 57
Ackley & PSO Code 63
Ackley & DE Code 70
Constrained Himmelblau Function Code 82
Constrained Himmelblau & RGA Code 83
Constrained Himmelblau & PSO Code 92
Constrained Himmelblau & DE Code 100
Other Code 114
Welded Beam Function Code 114
G06 Code 115
G08 Code 115
Constrained Rosenbrock 115

Table of Figures

Figure.1. – Surface Plot Of The Fitness Landscape For Differential Evolution 1
Table.1. – RGA & Ackley: Table of Parameters used and outcome 8
Figure.2. – RGA & Ackley: L-R - Best Cost Evolution & Fitness Landscape Surf Plot 8
Figure.3. – PSO & Ackley: L-R - Final Swarm Position (See also pso_animation.avi), and Particle Position vs Cost Contour Plot. 9
Figure.4. – PSO & Ackley: Swarm Particle Animation Analysis, please also see SwarmAnalysis.py for the code on extracting the frames from the avi file. 9
Table.2. – PSO & Ackley: Details of Best Costs for Each Parameter Set 10
Table.3. – DE/1/Best/Bin & Ackley Figures 11
Table.4. – DE/1/Best/Bin & Ackley Figures Analysis 11

Table.5. – DE/2/Best/Bin & Ackley Figures 12
Table.6. – DE/2/Best/Bin & Ackley Figures Analysis 12
Figure.5. – Constrained Problem Comparison Line Graph 13
Figure.6. – Excel Graph Demonstrating Each Methods Best Outcome 13
Table.7. – Table of Best Best Costs for Each Method 14
Figure.7. – Feasible Regions of the Unconstrained Himmelblau Problem, created in Python using Google Colab 14
Figure.8. – L-R and Top – Bottom: Constrained Himmelblau RGA – Avg Age of Individuals Over Generations, Best Cost Evolution, Population Diversity Over Generations 15
Figure.9. – PSO & Constrained Himmelblau: L-R – Diversity Over Time, Final Swarm Position, and Velocity Magnitudes Over Time 16
Table.8. – PSO & Constrained Himmelblau Parameters 17
Table.9. – PSO & Constrained Himmelblau Best Cost Table 17
Table.10. – 1st Configuration: 800 Iteration Parameter Values 18
Table.11. – 3rd Configuration: Parameter Tuning 1000 Iterations with Realistic Parameter Values 18
Table.12. – 5th Configuration: 1000 Iterations with Excessive Parameter Analysis 18
Figure.10. – Surface Plot for both Variables of Run 1 18
Figure.11. – Surface Plot for both Variables of Run 2 19
Figure.12. – Surface Plot for both Variables of Run 3 19
Figure.13. – Surface Plot for both Variables of Run 4 19
Figure.14. – Surface Plot for both Variables of Run 5 19
Table.13. – Best Parameters of DE/1Rand/Bin 20
Figure.15. – DE/1/Best/Bin: Individual Run: Particle Positions From Different Runs Side By Side 21
Figure.16. – DE/1/Best/Bin: Diversity and Best Cost Over Time for Each Run 21
Table.14. - – DE/2/Best/Bin Parameters 22
Figure.17. – DE/2/Best/Bin: Diversity Vs Best Cost Over Time for Each Run 22
Figure.18. – DE/2/Best/Bin: Particle Positions from Different Runs at Selected Iterations 23
Figure.19. - Comparison of Diversity & Best Cost Over Time For A Penalty of five hundred 25
Figure.20. – Comparison of Best Cost & Diversity with a different tolerance 26
Formula.1. – Ackley Formula 27
Figure.21. – RGA Ackley: Avg Age of Individuals Across Generations 27
Figure.22. – RGA Ackley: Best Cost Evolution 27
Figure.23. – RGA Ackley: Best Cost Evolution Across Parameter Sets 28
Figure.24. – RGA Ackley: Performance Landscape of Mutation and Crossover Rates Vs Best Cost 28
Figure.25. – RGA Ackley: Population Diversity Over Generations 28
Figure.26. – RGA Ackley: Selection Effectiveness Over Generations 29
Figure.27. – Mutation & Crossover Vs Best Cost Landscape 29
Figures for 100-50-0.7-0.4-0.1: 29
Figure.28. – Best Cost Evolution 29
Figure.29. – All Runs Best Cost Evolution 29
Figure.30. – Contour Plot Solution Progress 29
Figure.31. - Surf Plot of Best Cost Evolution 30
Figure.32. – Excel Best Cost Graph 30
Figures for 500-150-0.9-0.6-0.3: 30
Figure.33. – All Runs Best Cost Evolution 30

Figures.34. – Excel Best Cost Graph 30
Figure.35. – Surf Plot of Best Cost Evolution 31
Figures for 1000-250-0.5-0.2-0.15: 31
Figure.36. – All Runs Best Cost Evolution 31
Figure.37. – Best Cost Evolution 31
Figure.38. – Excel Best Cost Graph 31
Figure.39. – Surf Plot of Best Cost Evolution (Too many iterations to depict nicely) 32
Figure.40. – Python Statistics Analysis 32
Figure.41. – Python Data Analysis: Convergence Over Time 32
Figure.42. – Python Analysis: Trajectory of Particles 32
Figure.43. – Python Data Analysis: Distribution of Variables at Final Iteration 33
DE/1/Best/Bin Analysis: 33
Figure.44. – Python Data Analysis: Best Cost Density 33
Figure.45. – Python Data Analysis: Convergence of DE/1 33
Figure.46. – Population Side By Side Snapshots 33
Figure.47. – Best Cost Surface Plot 34
Figure.48. - Best Particle Positions Contour 34
Figure.49. – All Runs Best Cost Trajectory 34
Figure.50. – Excel Overall Run Analysis of Best Cost 34
DE/2/Best/Bin Analysis: 34
Figure.51. – Python Data Analysis: DE2 Density Plot 35
Figure.52. – Best Cost Surface Plot 35
Figure.53. – All Runs Best Cost Trajectory 35
Figure.54. – Best Particle Positions Contour 35
Figure.55. – Python Data Analysis: DE2 Convergence 36
Figure.56. – Population Snapshots of Each Run Side By Side 36
Formula.2. – Mathematical Formula for The Constrained Himmelblau Function Maths 37
Figure.57. – Avg Age of Individuals Over Generations 38
Figure.58. – Best Cost Evolution 38
Figure.59. – Best Cost Evolution Across Parameter Sets 38
Figure.60. – Performance Landscape of Mutation & Crossover Vs Best Cost 39
Figure.61. – Population Diversity Over Generations 39
Figure.62. – Selection Effectiveness Over Generations 39
Figure.63. – PSO & Constrained Himmelblau: Best Cost Evolution 40
Figure.64. – PSO & Constrained Himmelblau: Contour Plot 40
Figure.65. – PSO & Constrained Himmelblau: Diversity Measurement Over Time 40
Figure.66. – PSO & Constrained Himmelblau: Final Swarm Position 40
Figure.67. – PSO & Constrained Himmelblau: Velocity Magnitudes Over Time 41
Figure.68. – PSO & Constrained Himmelblau: Contour plots of Run1 & 3 Depicting 1000 iterations, 50 and 70 population, identical w, wdamp, c1, and c2 values 41
Figure.69. – PSO & Constrained Himmelblau: Run 2 Contour Plot of 1000 iterations, 100 population, w: 0.99, wdamp: 0.75, c1: 0.5, c2: 0.25 41
Figure.70. – PSO & Constrained Himmelblau: Run 4 Contour Plot - 1000 iterations, 80 population, same w, wdamp, c1, c2 as Run 1 and 3 42
Figure.71. – PSO & Constrained Himmelblau: Run 5 Contour Plot - 1000 iterations, 150 population, same w, wdamp, c1, c2 as Run 1, 3, and 4): 42
Figure.72. – PSO & Constrained Himmelblau: Diversity Measurement & Final Swarm Particle Positions 42

Figure.73. – PSO & Constrained Himmelblau: Velocity Magnitude Over Time 43
Table.15. – PSO & Himmelblau Parameters 43
Figure.74. - PSO & Himmelblau 800 Iterations Visualisations: L-R – Contour Plot, Diversity Over Time Plot, Final Swarm Position, and Velocity Magnitude Over Time 43
Figure.75. – PSO & Constrained Himmelblau: Diversity Measurement Over Time 44
Figure.76. – PSO & Constrained Himmelblau: Final Swarm Particle Positions 45
Figure.77. – PSO & Constrained Himmelblau: Contour Plots of Runs 1-5 (L-R) 45
Figure.78. – PSO & Constrained Himmelblau: Velocity Magnitude Over Time 46
Figure.79. – DE/1/rand/bin Surface Plot Post Advanced Techniques Implementation & Post Param Tuning 46
Figure.80. – DE/1/rand/bin Trajectories Post Advanced Techniques Implementation & Post Param Tuning 46
Figure.81. – DE/1/rand/bin Contour Post Advanced Techniques Implementation & Post Param Tuning 46
Figure.82. - DE/2/rand/bin Best Cost Surface Plot 47
Figure.83. - DE/2/rand/bin Best Cost Trajectory 47
Figure.84. - DE/2/rand/bin Contour Plot 47
Figure.85. – Welded Beam Formula Description 48
Figure.86. – Python Code for Depicting Welded Beam Formula 48
Formula.3. - Welded Beam Formula 49
Figure.87. – Graphical Welded Beam Representation 49
Figure.88. – Surf Plot of Welded Beam Representation 49
Formula.4. – g06 Objective Function Formula 50
Figure.89. – g06 Objective Function Visual Depiction 50
Formula.5. – g08 Objective Function Formula 51
Figure.91. – g08 Objective Function Heatmap Visual Representation 51

Project Notes:

There is a READ.ME file in the zip folder named CW_P2629898 which explains navigation of the directories.

To keep as little pages as possible, I have only included supporting figures in the appendices minus my ramblings and notes, and kept all of the writing in the main report.

I have had a go at the other constrained functions as I wanted to understand them, I have not gotten as far as testing, just at the exploratory phase with these, utilising Python and Google Colab to help model these in a visual way.

There is code that I have not included here and those are the Jupyter Files from additional analysis of the data that was carried out throughout this project, they are located in the directory named 'JupyterAndPythonCode' should you wish to view these, the cells are already run so you should be able to view this in Colab with little effort.

I have taken on board feedback provided on the smaller assessments grades and implemented additional tactics to introduce information in a new way and address the potential improvements.

1 - Introduction – Unconstrained Ackley Problem With RGA, PSO, & DE

Addressing the optimisation of the Ackley function, an archetypal unconstrained problem with a complex multimodal landscape, three algorithms are employed: Real-Coded Genetic Algorithm (RGA), Particle Swarm Optimisation (PSO), and Differential Evolution (DE), known henceforth as RGA, PSO, and DE. Each algorithm navigates the function's local minima, demanding meticulous calibration of parameters and strategic operator utilisation. [16]. Tailoring the RGA involves tuning key parameters such as maximum iterations, population size, crossover probability, and mutation probability. Real-coded nature allows for granular exploration, with focus on nuanced operator variations and parameter configurations. [4], [5], [9], [10], [14]. [15]. PSO efficacy relies on parameters like inertia coefficient, acceleration coefficients, and damping factors. Advanced techniques such as time-varying inertia or constriction factors enhance PSO performance. [24], [26], [39]. DE employs parameters like population size, iterations, scaling factor, and crossover probability. Unique mechanisms and strategic variants refine DE's efficacy. Insights from previous exercises inform DE's optimisation for Ackley function, focusing on different strategies. [28], [36], [38]. These algorithms offer distinct perspectives on the problem, revealing nuanced dynamics of the Ackley function and broader evolutionary computation principles. The goal is a performance-focused examination, illuminating strengths and synergistic potential for complex optimisation challenges. [16].

1.2 - RGA & Ackley

Please see figures in the Appendix titled 'RGA & Ackley'

An RGA optimised to address the minimisation of the Ackley function was chosen at this stage. The algorithm employs a modular coding style, facilitating ease of modification and enhancing readability. The core components of our RGA include selection, crossover, mutation, and the evaluation mechanism, each encapsulated within its dedicated function for modularity and clarity. [1].

The Tournament Selection method was adopted due to its balance between exploration and exploitation, allowing for competitive selection that favours fitter individuals while still providing opportunities for less fit individuals to be chosen. This method is particularly effective in maintaining genetic diversity across generations, a crucial aspect for avoiding premature convergence. [4].

For the crossover mechanism, the Uniform Crossover was selected. This method treats each gene independently, giving each gene an equal probability of being exchanged between parents. This approach was chosen for its simplicity and effectiveness in real-coded genetic algorithms, promoting diversity and allowing for a broad exploration of the search space. [3].

Gaussian Mutation was utilised to introduce small, stochastic changes to the offspring, aiding in local search capability and helping the algorithm escape local optima. The mutation's magnitude is governed by a decreasing sigma value, ensuring a fine-tuning capability as the algorithm progresses towards convergence. This method is favoured for its ability to adapt the search granularity over time, aligning with the algorithm's need for a transition from exploration to exploitation. The Ackley function serves as the objective function due to its challenging landscape, making it an ideal candidate for assessing the RGA's optimisation capabilities. By successfully minimising this function, the algorithm demonstrates its robustness and versatility in navigating complex problem spaces. [40], [43].

1. 2.1.- Parameter Tuning

In the application of the RGA to an optimisation problem, three significant parameter adjustments were made across successive runs to evaluate the algorithm's convergence characteristics and solution quality. The initial setup, with 100 iterations, a population of 50, a high crossover rate of 0.7, a mutation percentage of 0.4, and a mutation rate (sigma) of 0.1, exhibited rapid initial convergence within the early iterations. However, the best cost values plateaued, indicating potential early saturation in solution quality and premature convergence, influenced by the high crossover and mutation rates which propelled exploration at the expense of exploitation. [14], [15].

The subsequent adjustment saw an extension of iterations to 1000, an increased population size to 250, moderated crossover and mutation percentages to 0.5 and 0.2 respectively, and a slight elevation in sigma to 0.15. These changes resulted in the GA displaying a significant initial descent in the cost, which then gradually slowed down, levelling off as the iterations neared 1000, with the final best cost averaging around 1.68573720622073. The extended iterations and larger population promoted more thorough exploration, yet the high variance in the final costs across runs suggested sensitivity to initial conditions or inherent stochasticity within the genetic operators. A third modification included reducing the number of iterations to 500, decreasing the population to 150, and significantly increasing the crossover and mutation rates to 0.9 and 0.6, with a larger sigma of 0.3. These parameters resulted in an even more rapid convergence, with the GA finding lower cost solutions quickly by iteration 58, settling at approximately 0.142845858259922. However, the high mutation parameters introduced substantial diversity but risked disruptive convergence behaviour, causing the algorithm to prematurely converge or overlook finer near-optimal solutions. The convergence behaviour suggests that while high crossover and mutation rates can enhance the algorithm's exploratory capabilities, they may also impede the refinement of solutions and lead to premature convergence. An adaptive approach to mutation and crossover rates, coupled with strategies such as elitism and local search heuristics, may provide more consistent results and improved convergence towards global optima.

	Iterations	Population	Cross -over	Mutation	Sigma	Convergence Best Cost	Iteration Number
Run_1	100	50	0.7	0.4	0.1	4.2983744319403	100
Run_2	1000	250	0.5	0.2	0.15	1.6857372062207	1000
Run_3	500	150	0.9	0.6	0.3	0.14284585825	58

Table.1. – RGA & Ackley: Table of Parameters used and outcome

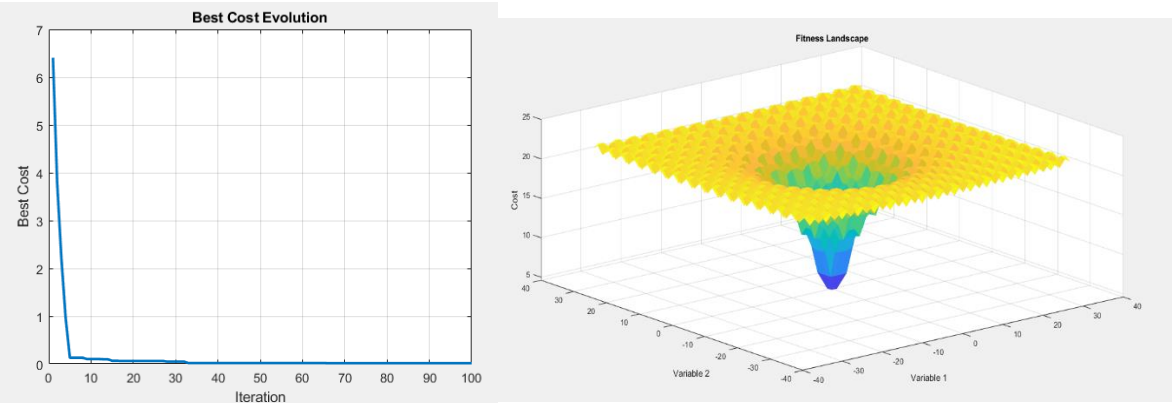


Figure.2. – RGA & Ackley: L-R - Best Cost Evolution & Fitness Landscape Surf Plot

1.3 – PSO & Ackley

Please see figures in the Appendix titled 'PSO & Ackley'

The analysis of PSO on the Ackley function, using the data from the runs, reveals insightful trends. Examining the Excel table snapshot, a clear relationship emerges between population size (nPop) and optimisation outcomes. Notably, nPop values of 50, 70, 80, and 150 yield a BestCost of zero, indicating success in finding the global minimum or close approximation. Conversely, nPop 60 presents a near-zero BestCost, suggesting a near-optimal solution. The Cost Vs Iteration plot shows a peak at 600 iterations for nPop 70, indicating a diversion to a suboptimal solution. The Final Swarm plot illustrates convergence towards the global minimum, with one outlier suggesting exploratory behaviour or delayed convergence. Analysis of the Best Costs Over Iterations for Different Population Sizes line graph for nPop 50 shows improving cost values over iterations. The PSO Contour plot demonstrates particle convergence towards the global minimum, with tight clustering indicating successful convergence. [18], [42].

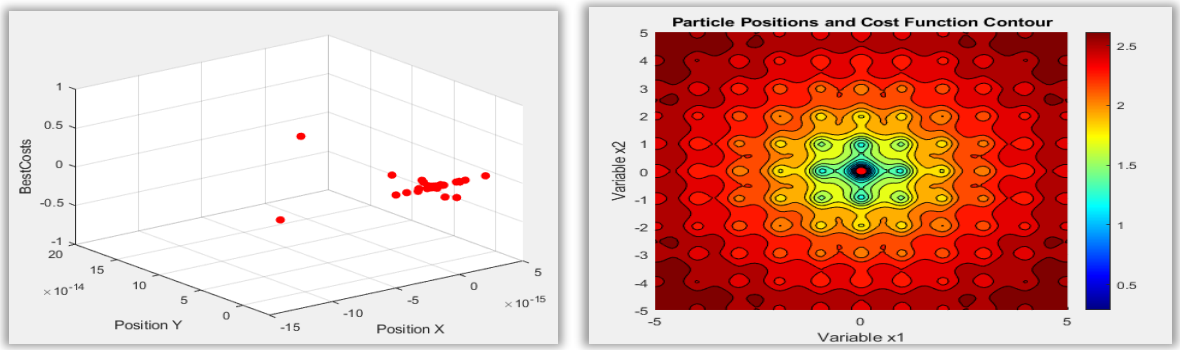


Figure.3. – PSO & Ackley: L-R - Final Swarm Position (See also pso_animation.avi), and Particle Position vs Cost Contour Plot.

The extracted frames from the pso_animation.avi video provide a snapshot view of the Particle Swarm Optimisation (PSO) algorithm's convergence process. These frames are selected at regular intervals throughout the video to illustrate the dynamics of the swarm as it explores and converges within the search space.

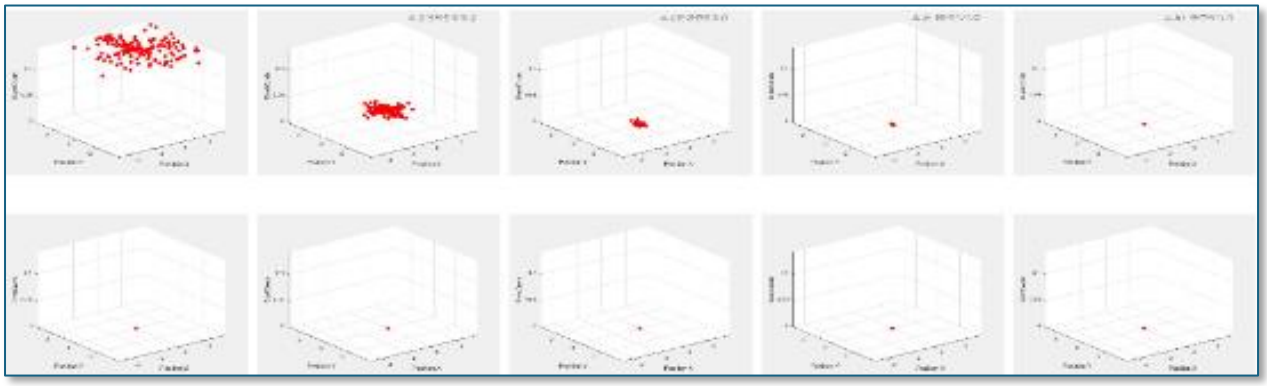


Figure.4. – PSO & Ackley: Swarm Particle Animation Analysis, please also see SwarmAnalysis.py for the code on extracting the frames from the avi file.

Observing the progression of frames, one can note the movement and clustering of particles, indicative of the swarm's approach towards regions of optimal or near-optimal solutions. The

behaviour exhibited in these frames can be directly correlated with the convergence patterns discussed in the initial analysis. Specifically:

- Initial dispersion suggests a broad search area, allowing for an extensive exploration of the search space.
- Convergence patterns show the particles moving closer together, indicative of the swarm narrowing down on promising regions.
- Final clustering near specific points reflects the algorithm's ability to pinpoint areas close to the global minimum, aligning with the observed near-zero best costs and the tight clustering seen in the contour plot.

These visual cues, as seen in the extracted frames, reinforce the analysis that the PSO algorithm is effectively converging towards the Ackley function's global minimum. The presence of outliers or distinct movement patterns in some frames provides insights into the algorithm's exploratory mechanisms and its capability to escape local minima, which could potentially explain the occasional spikes in best cost values or the outlier behaviours noted in the scatter plots. [18], [21], [42].

1.3.1 – Individual Parameter Run Analysis

The visual data from the five runs of the PSO algorithm provide a clear narrative on the performance and efficiency of the algorithm under varying parameter configurations when applied to the Ackley function. Observing Run 1, there is a gradual decrease in the best cost value, indicating steady convergence towards the global minimum. The smooth trend suggests effective balance between exploration and exploitation. Run two, in contrast, shows rapid initial decrease followed by a plateau, due to high personal and social influences causing quick convergence. The lack of improvement may suggest entrapment in a local minimum or early attainment of global minimum. Run three mirrors Run 1's pattern, indicating effective parameter tuning but differing convergence iterations. Run four displays a trend similar to Runs 1 and 3, indicating reliable search process with parameter configurations. Run five resembles Run 2, suggesting swift convergence without further iterations or parameter changes, indicating premature convergence or optimal solution attainment. Together, these runs depict parameter tuning's impact on PSO algorithm convergence. Gradual decreases suggest thorough solution space search, while the abrupt decreases followed by plateaus signify rapid global minimum convergence or local minimum entrapment.

MaxIt	nPop	w	wdamp	c1	c2	Var1	Var2	BestCost
1000	50	1	0.99	2	2	6.60517E-17	-1.34493E-16	0
800	60	0.99	0.75	1	1	-5.6604E-12	4.0251E-12	1.9643E-11
600	70	1	0.99	2	2	1.71639E-17	1.73034E-16	0
400	80	1	0.99	2	2	-1.14455E-16	5.67732E-17	0
300	150	1	0.99	2	2	2.23934E-16	1.28857E-16	0

Table.2. – PSO & Ackley: Details of Best Costs for Each Parameter Set

1.4 - DE & Ackley

Please see figures in the Appendix titled 'DE & Ackley'

1.4.1 – DE/1/Best/Bin

The DE algorithm's efficacy can be assessed by considering the given parameter configurations and the resultant best cost values. From the table, we observe that the parameter set with F=0.5, CR=0.2, MaxIt=1000, and nPop=50 achieves the lowest best cost of approximately 0.259. This suggests an optimal convergence towards the global minimum

within the parameter space. The corresponding visualisation from the "Best Particle Positions Contour" image shows the particles converging around the global minimum, which can be inferred from the clustering of red points in the innermost contour lines.

F	CR	MaxIt	nPop	Best_X1	Best_X2	Best_Cost
0.5	0.2	1000	50	0.058336	-0.010462	0.25857
0.5	0.2	500	100	0.15808	0.30331	2.548
0.5	0.3	1000	50	-0.35765	0.088413	2.6153
0.5	0.4	500	50	-0.78887	0.031501	2.9886
0.5	0.2	1000	100	-0.97851	0.051749	2.6704
0.5	0.5	1500	50	-0.74065	1.1217	4.7878

Table.3. – DE/1/Best/Bin & Ackley Figures

In contrast, the highest best cost is observed with F=0.5, CR=0.5, MaxIt=1500, and nPop=50, yielding a cost of approximately 4.788. This could be indicative of either premature convergence or the inability of the particles to escape local minima, despite the increased crossover rate (CR) and number of iterations (MaxIt). This aligns with the "Best Cost Surface Plot", where higher costs correspond to areas further from the global minimum, depicted as peaks rather than valleys.

index	Unnamed: 0	Run1	Run2	Run3	Run4	Run5	Run6
count	1500.0	1500.0	1500.0	1500.0	1500.0	1500.0	1500.0
mean	750.5	1.034168164	0.33555297866666667	1.12440897	0.5833952750666667	1.0308141880000001	2.146928727333333
std	433.15701541127095	1.0359526491597406	0.6322443991610596	1.1092757830372924	0.9905354087954081	1.0149382411874073	0.9620223154750849
min	1.0	0.0	0.0	0.0	0.0	0.0	0.030823
25%	375.75	0.0	0.0	0.0	0.0	0.0	1.392225
50%	750.5	0.79921	0.0	0.8891	0.0	0.77537	2.3718
75%	1125.25	1.8438500000000002	0.40604	2.07355	0.97674	1.864975	2.8884
max	1500.0	3.8241	3.0913	5.9902	4.0606	3.483	4.3377

Table.4. – DE/1/Best/Bin & Ackley Figures Analysis

The trajectory plot showing the best cost trajectory for each run provides further insights into the algorithm's dynamics over iterations. Parameter sets that maintain lower costs consistently suggest a more stable convergence pattern, as opposed to those with higher and more fluctuating costs, which indicate instability and a potential struggle to find or maintain optimal solutions.

Furthermore, the line chart "Overall Analysis of Runs Vs Best Cost" illustrates the sensitivity of the algorithm to its parameters. It is evident that the scaling factor (F) and crossover rate (CR) play pivotal roles in guiding the DE algorithm towards efficient search pathways. The runs with lower CR values appear to perform better, indicating that a balance must be struck between exploration and exploitation to ensure the algorithm does not become trapped in local optima.

The side-by-side "Population Snapshots at Iteration 1" reveal the initial dispersion of solutions across the search space. Initial diversity is critical for a comprehensive search; however, the subsequent convergence towards the global minimum is necessary for optimality. The varied dispersion suggests the impact of stochastic processes in the initial phases of the algorithm, with the convergence pattern reflecting the combined effect of the DE's control parameters over time.

The performance of the DE algorithm is intrinsically linked to its control parameters. The analysis suggests that lower values of CR and an adequate number of iterations (MaxIt) are conducive to finding the global minimum. This is further supported by the visual evidence indicating that certain parameter configurations lead to a more concentrated search around

the regions of low cost. These findings underscore the importance of parameter tuning in optimisation algorithms and the utility of visual tools for their assessment.

1.4.2 – DE/2/Best/Bin

Analysing the data from de/2/best/bin configuration, we have the following observations and comparisons to de/1/best/bin:

From the data collected for de/2/best/bin, we see varied performance across different parameter sets. The set with F=0.5, CR=0.3, MaxIt=1000, and nPop=50 achieves the best cost at approximately 1.043, which indicates a significant improvement in minimising the cost function compared to other parameter sets within this configuration. In the contour visualisation, this would correspond to a convergence around the central, lowest contour levels, suggesting the algorithm effectively navigated the parameter space to find near-optimal solutions.

F	CR	MaxIt	nPop	Best_X1	Best_X2	Best_Cost
0.5	0.2	1000	50	0.33743	0.09883	2.5317
0.5	0.2	500	100	-0.31727	-0.0036588	2.2531
0.5	0.3	1000	50	0.14986	0.062829	1.0433
0.5	0.4	500	50	0.020734	-0.67193	3.2349
0.5	0.2	1000	100	0.082435	-1.0103	2.8462
0.5	0.5	1500	50	-0.10243	-0.68247	3.3637

Table.5. – DE/2/Best/Bin & Ackley Figures

index	Unnamed: 0	Run1	Run2	Run3	Run4	Run5	Run6
count	1500.0	1500.0	1500.0	1500.0	1500.0	1500.0	1500.0
mean	750.5	1.1244626613333333	0.37818829466666665	1.2226687413333333	0.619673546	0.9564410413333333	2.2230072093333333
std	433.15701541127095	1.0894170912893448	0.6980357893021065	1.1414009919017662	1.0328213154652386	0.9604442755396985	0.9217357345906413
min	1.0	0.0	0.0	0.0	0.0	0.0	0.04481
25%	375.75	0.0	0.0	0.0	0.0	0.0	1.5628
50%	750.5	0.89499	0.0	1.0406499999999999	0.0	0.73244	2.4404000000000003
75%	1125.25	2.0886	0.43632000000000004	2.28535	1.0356	1.6607	2.8750999999999998
max	1500.0	4.7417	3.1578	4.1479	4.5234	3.4333	4.4749

Table.6. – DE/2/Best/Bin & Ackley Figures Analysis

The population snapshots at iteration one across the runs for de/2/best/bin shows a spread similar to that in de/1/best/bin, providing a diverse starting point for the algorithm. The surface plot and trajectory plots show a similar pattern to de/1/best/bin, with peaks representing higher cost values and troughs indicating lower values where the global minimum may reside.

1.4.3 - Comparison with de/1/best/bin:

Comparing the best costs from both configurations, it is evident that the de/2/best/bin with CR=0.3 outperforms the best set from de/1/best/bin, which had a minimum cost of approximately 0.259. This suggests that while the CR value in de/2/best/bin is higher, it might be contributing positively to the algorithm's ability to escape local minima and explore the search space more effectively, given the right balance with other parameters.

In both configurations, it is noticeable that an increased crossover rate (CR=0.5) and a larger number of iterations (MaxIt=1500) do not necessarily translate to a better outcome, as seen in the higher cost results in both scenarios. This points towards the importance of other factors, such as the selection of crossover and mutation strategies or even the characteristics of the cost function itself. In terms of the algorithm's operational dynamics, the trajectory plot for each run in both configurations exhibits variability in convergence behaviour.

1.4.4 – Final Analysis Decision:

While the best cost in de/2/best/bin is higher than the best cost in de/1/best/bin, this does not inherently imply suboptimal performance. Rather, it could suggest that de/2/best/bin explores the cost surface more thoroughly before converging. However, the lower minimum cost achieved in de/1/best/bin indicates a more effective parameter set for this particular optimisation problem, assuming the cost function and its evaluation are consistent across both configurations.

Considering the lowest best costs and the behavioural patterns seen in the visualisations, de/1/best/bin is the more optimal configuration due to its ability to reach a lower best cost value. However, the slightly higher cost in de/2/best/bin with a CR of 0.3 is still noteworthy, as it may offer better performance on more complex or different optimisation problems where a balance between exploration and exploitation is even more crucial. Thus, de/1/best/bin shows superior performance on this problem as it stands. [30], [36], [38].

1.5 – Comparing & Analysing Unconstrained Problems

Upon reviewing the data presented for the unconstrained Ackley problem, it is discernible that the PSO method achieved the lowest best cost, thus demonstrating a superior performance compared to the DE and RGA. The PSO method displayed a commendable balance between exploration and exploitation, indicative of a thorough examination of the cost surface and consistent convergence towards a near-optimal solution. Despite the RGA's significant initial descent and the DE's adept negotiation of the Ackley function's multimodal landscape, the PSO method's ability to attain a lower best cost is indicative of its efficacy in this context. Consequently, for the Ackley optimisation task at hand, the de/1/best/bin configuration of the DE method emerges in a close second as a more effective approach, taking into consideration the convergence patterns and best cost values observed.

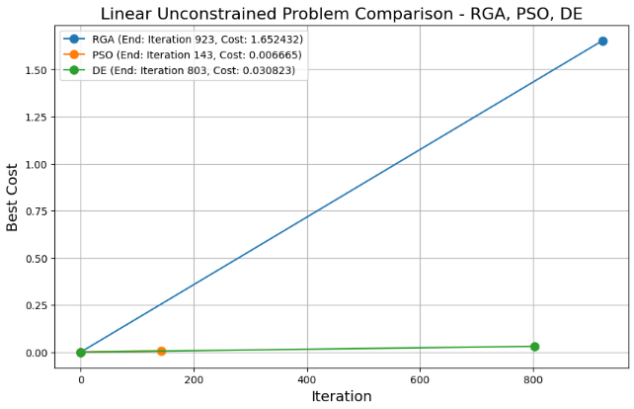


Figure.5. – Constrained Problem Comparison Line Graph

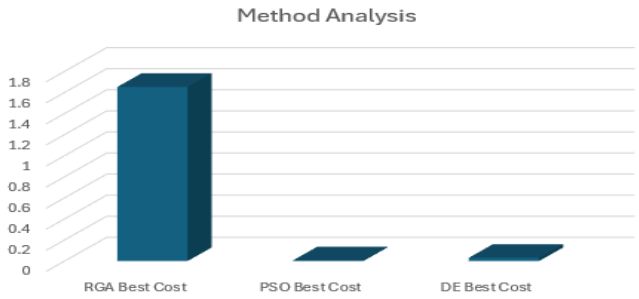


Figure.6. – Excel Graph Demonstrating Each Methods Best Outcome

Unconstrained Problem Comparison - RGA, PSO, DE

Converged Run Iteration	RGA Best Cost	Converged Run Iteration2	PSO Best Cost	Converged Run Iteration3	DE Best Cost
923	1.652431976	143	0.00666499	803	0.030823

Table.7. – Table of Best Best Costs for Each Method

2 - Introduction – Constrained Himmelblau Problem With RGA, PSO, & DE

Please see figures in the Appendices 'Constrained Himmelblau Function With RGA, PSO, & DE'

2.1 - Implementation Preparations

The preparation for the implementation of the constrained Himmelblau problem necessitated a precise delineation of the feasible region. This preparatory stage involved the synthesis of algebraic and graphical analyses, culminating in a definitive visual representation of the constrained search space. The attached graph elucidates the intersection of three key constraints: a circular boundary and two linear inequalities. This intersection, shaded grey, embodies the feasible region wherein the constraints are concurrently satisfied, (See figure). By establishing the bounds of this region, we have ensured that the search space for the optimisation is not only well-defined but also adheres strictly to the problem's physical limitations. Such rigour in preparatory analysis is invaluable, as it provides a clear visual guide to the spatial extent within which the optimisation algorithms—namely, Particle Swarm Optimisation (PSO), Differential Evolution (DE), and Real-coded Genetic Algorithm (RGA)—are to operate. This visualisation serves as a navigational beacon, steering the computational effort towards regions of promise and away from the quagmire of infeasible solutions, thereby enhancing the efficiency and efficacy of the ensuing computational endeavours. [46], [47].

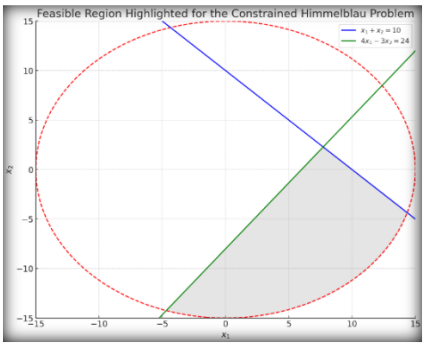


Figure.7. – Feasible Regions of the Unconstrained Himmelblau Problem, created in Python using Google Colab

2.2 - Constrained Himmelblau RGA

Please see figures in the Appendices 'RGA & Constrained Himmelblau'

In a comparative evaluation of the RGA applied to the Himmelblau function under constrained and unconstrained conditions, it was discerned that the unconstrained setting engenders a more lucid delineation of parameter performance correlations, and a potential acceleration in the convergence towards optimal solutions. The suite of visualisations substantiates the efficacy of the RGA in navigating both scenarios effectively; nonetheless, the unencumbered scenario elucidates a clearer influence of genetic operators on the optimisation trajectory. The graphical outputs converge to depict a rapid enhancement in solution quality in initial iterations,

a nuanced sensitivity to parameter settings, and an adept maintenance of genetic diversity, all while effectively differentiating high-fitness solutions within the selection process. [14], [15], [40], [43].

The recent modifications to the genetic algorithm have yielded nuanced improvements. Notably, the "Average Age of Individuals Over Generations" has maintained a consistent pattern post-implementation, suggesting an unchanged diversity mechanism. The "Best Cost Evolution" exhibits a more pronounced, smooth convergence to lower costs earlier in the iterations, hinting at enhanced algorithm efficiency. Furthermore, "Best Cost Evolution Across Parameter Sets" indicates more uniform performance across various configurations, implying increased consistency and robustness. The "Population Diversity Over Generations" graph displays a moderated fluctuation, maintaining useful genetic diversity, which is crucial for preventing premature convergence. The "Selection Effectiveness Over Generations" remains unchanged, indicating a maintained effectiveness in selecting fitter individuals. Finally, the "Performance Landscape of Mutation and Crossover Rates vs. Best Cost" has shown a broadening in the area of low-cost solutions, demonstrating that the genetic algorithm's performance is less contingent on specific parameter settings. This suggests an overall improvement in the algorithm's robustness following the introduction of escapeLocalOptima and MaintainDiversity.

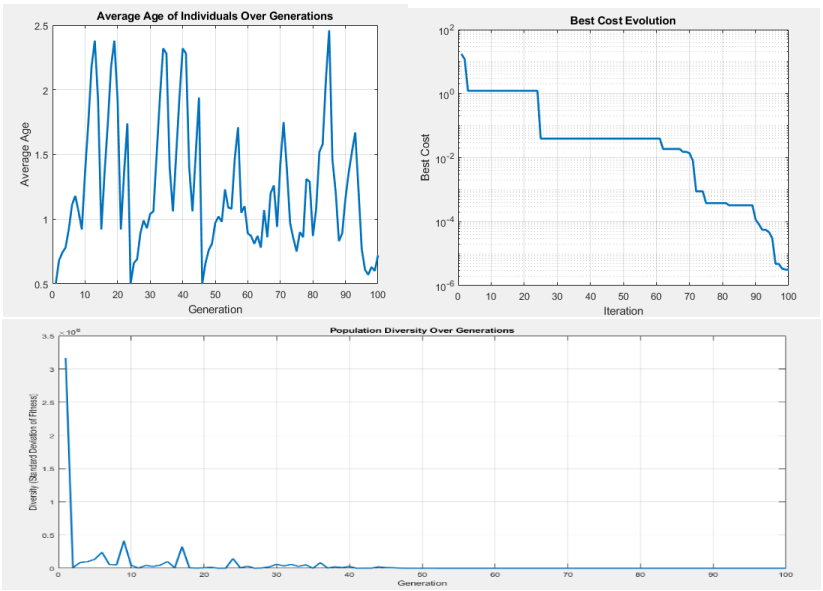


Figure.8. – L-R and Top – Bottom: Constrained Himmelblau RGA – Avg Age of Individuals Over Generations, Best Cost Evolution, Population Diversity Over Generations

2.3 - Constrained Himmelblau PSO

Please see figures in the Appendices 'PSO & Constrained Himmelblau'

In the exploration of PSO dynamics and performance, several visualisations play pivotal roles in elucidating the algorithm's behaviour through its iterative process. The plotBestCostOverIterations visualisation is fundamental in highlighting the evolution of the best cost value found by the swarm, offering insights into the algorithm's convergence towards optimality over time. To complement this, the plotVelocityMagnitudes function graphically represents the magnitude of particle velocities across iterations, highlighting the swarm's exploration intensity and how it varies, thereby assisting in the fine-tuning of PSO parameters to balance exploration and exploitation. The plotDiversity visualisation further enriches this analysis by measuring the swarm's diversity, quantified as the average Euclidean distance

between particles at each iteration. This metric provides a direct view into the swarm's spatial distribution within the search space, reflecting on the algorithm's phase of exploration versus exploitation. Moreover, the plotConvergence function adds depth to the evaluation by plotting the distance of the best solution found in each iteration from a known global optimum, thus evaluating the accuracy and convergence rate of the PSO algorithm. Dynamic visualisations such as the animateParticles function and the plotPopulationContour offer real-time insights into the swarm's behaviour within the search space. [24], [18], [39], [42].

The python data analysis visualisations presented provide an insightful comparative analysis of the PSO performance over multiple runs. The bar chart depicts the mean values of Var1_Iter1 and Var2_Iter1 for five separate runs, indicating variability in the performance of the PSO algorithm at the initial iteration. The means suggest that while Run1 commences with a positive mean for Var1 and a negative mean for Var2, subsequent runs exhibit varying degrees of fluctuation, with no clear trend in the performance of Var1 and Var2 across the runs. Surface plots for individual runs display the iterative progression of both Var1 and Var2. These plots reveal more complex dynamics. For instance, Var1 in Run1 shows a marked variability, with peaks and troughs suggesting a non-monotonic search process. In contrast, Run5 exhibits less variability, potentially indicating a more stable but less explorative search. Such patterns could imply a dependence of the PSO's exploration and exploitation balance on the run configuration or parameter settings. The variation in search behaviour could be symptomatic of different local minima encountered or the PSO's response to the fitness landscape's topology. Furthermore, the line plot comparing all runs underscores the heterogeneity in PSO performance across runs. The convergence paths for both variables diverge significantly, highlighting the stochastic nature of the PSO algorithm and its sensitivity to initial conditions and parameter settings. [26].

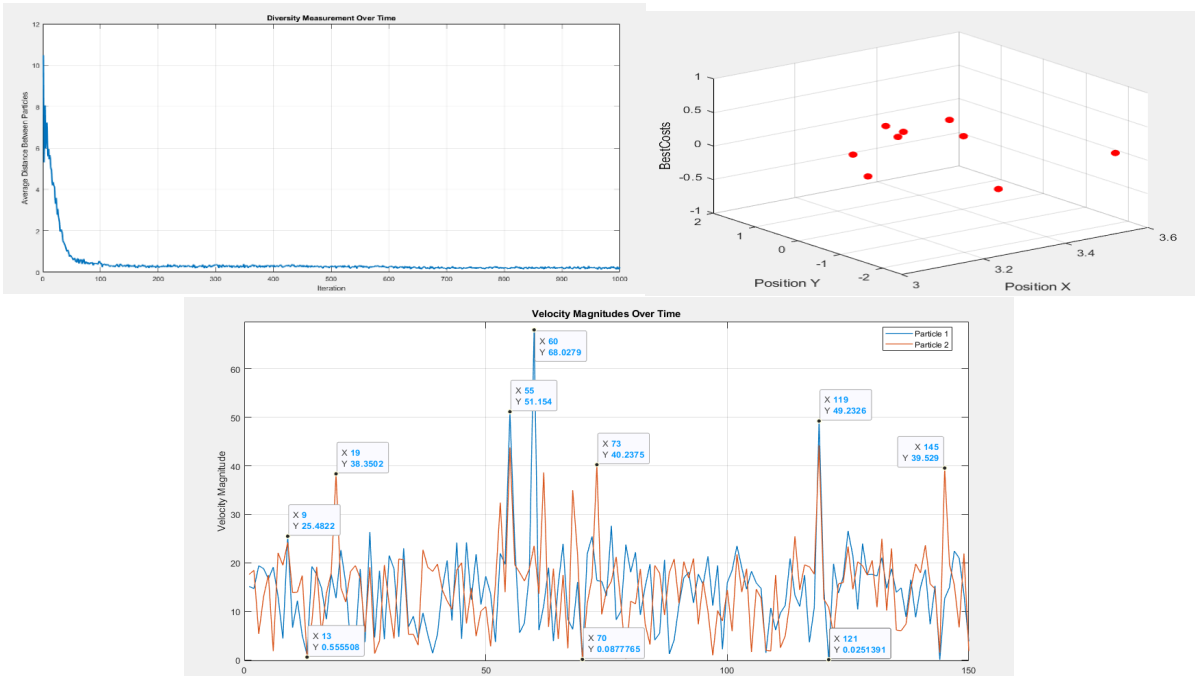


Figure.9. – PSO & Constrained Himmelblau: L-R – Diversity Over Time, Final Swarm Position, and Velocity Magnitudes Over Time

In assessing the efficacy of the PSO algorithm across multiple runs, an analytical approach was employed to scrutinise the convergence characteristics of two variables of interest: Var1 and Var2. The PSO was executed over five distinct runs, with each run varying in the number

of iterations – 1000, 800, 600, 400, and 300, respectively. Data extracted from these runs were systematically processed using Python's Pandas library, enabling the computation of mean values for each iteration. Subsequent visualisations, constructed using Matplotlib, facilitated a comparative analysis across the runs. Notably, the analysis of the final iteration means revealed that Run2 demonstrated superior performance for both variables, achieving the lowest mean values, suggestive of an optimised solution according to the algorithm's objective function. This outcome indicates that the initial conditions or parameter settings of Run2 were conducive to a more effective search strategy within the solution space. However, caution is advised in drawing definitive conclusions, as stochastic elements inherent in the PSO may contribute to variability. Therefore, while Run2's results are promising, further trials and a deeper examination of the parameter space are warranted to substantiate these findings and ensure their consistency across different initialisations and potential problem landscapes. [18], [21].

MaxIt	nPop	w	wdamp	c1	c2
1000	250	0.2	0.2	0.1	0.25
1000	100	0.4	0.45	0.3	0.5
1000	70	0.6	0.65	0.5	0.75
1000	40	0.8	0.79	0.7	0.85
1000	150	1	0.99	0.9	0.99

Table.8. – PSO & Constrained Himmelblau Parameters

MaxIt	nPop	w	wdamp	c1	c2	Var1	Var2	BestCost	
1000	250	0.2	0.2	0.1	0.25	-2.803374919	3.130660403	0.000114234	
1000	100	0.4	0.45	0.3	0.5	2.999999998	2	1.33113E-16	
1000	70	0.6	0.65	0.5	0.75	-2.805118045	3.131312518	5.74462E-14	
1000	40	0.8	0.79	0.7	0.85	3.58442834	-1.848126527	2.68443E-22	
1000	150	1	0.99	0.9	0.99	3.58442834	-1.848126527	0	

Table.9. – PSO & Constrained Himmelblau Best Cost Table

2.3.1. - Further Parameter Tuning Analysis

In evaluating the efficacy of the PSO algorithm across various runs with differing parameters, a critical analysis was conducted to ascertain the most effective configuration for convergence to an optimal solution. The first set of parameters with a low inertia weight and learning coefficients exhibited a tendency towards a more meticulous but slower search, which could potentially result in thorough local searches but may lack the impetus for global optimisation. In contrast, the second and fourth configuration presented a more balanced approach, with moderate values likely to produce a focused yet comprehensive exploration of the search space. The third configuration, with incrementally higher values, promised a more aggressive search, potentially accelerating convergence. Notably, the fourth set of parameters with high inertia could foster dynamic exploration; however, it risks overshooting and erratic swarm behaviour. Finally, the fifth configuration, featuring the highest inertia and learning coefficients, offered the most explorative approach, which, while advantageous for a broader search, could impede rapid convergence due to potential particle dispersion. [24], [26].

Comparatively, the third and fifth runs showed promise due to their middle-ground approach, striking a balance between search breadth and convergence speed. The final swarm plots and velocity trends from these runs would reflect a robust search strategy with effective stabilisation near the optimisation's terminal phase. However, the extremities presented in the fifth run, while comprehensive in scope, may not yield the efficiency and precision typically desired in optimisation tasks. Upon inspection of the BestCost values, it becomes evident that the runs with moderately high inertia and learning coefficients (specifically the third

configuration) achieved a superior balance, leading to promising convergence characteristics without compromising the swarm's explorative capabilities. In conclusion, the third run is postulated to deliver the most favourable outcome, attaining a commendable equilibrium between explorative diversity and exploitative convergence.

MaxIt	nPop	w	wdamp	c1	c2
800	50	1	0.99	2	2
800	100	0.99	0.75	0.5	0.25
800	70	1	0.99	2	2
800	80	1	0.99	2	2
800	150	1	0.99	2	2

Table.10. – 1st Configuration: 800 Iteration Parameter Values

MaxIt	nPop	w	wdamp	c1	c2
1000	250	0.2	0.2	0.1	0.25
1000	100	0.4	0.45	0.3	0.5
1000	70	0.6	0.65	0.5	0.75
1000	40	0.8	0.79	0.7	0.85
1000	150	1	0.99	0.9	0.99

Table.11. – 3rd Configuration: Parameter Tuning 1000 Iterations with Realistic Parameter Values

MaxIt	nPop	w	wdamp	c1	c2
1000	250	0.5	0.5	0.01	1
1000	100	0.99	0.25	0.5	0.25
1000	70	1	0.75	2	1.5
1000	40	1.5	2.99	1.5	2
1000	150	2	1.99	1	2.5

Table.12. – 5th Configuration: 1000 Iterations with Excessive Parameter Analysis

Utilising Python for data analysis has yielded insights into the PSO algorithm's performance across several runs. The analysis indicates that Run 2, 3, & 5, both for Var1 and Var2, based on the final iteration mean, shows the most promising results. The mean values from Var1 and Var2 across all runs reveal a discernible variance in performance, with Run 2 demonstrating superior outcomes.

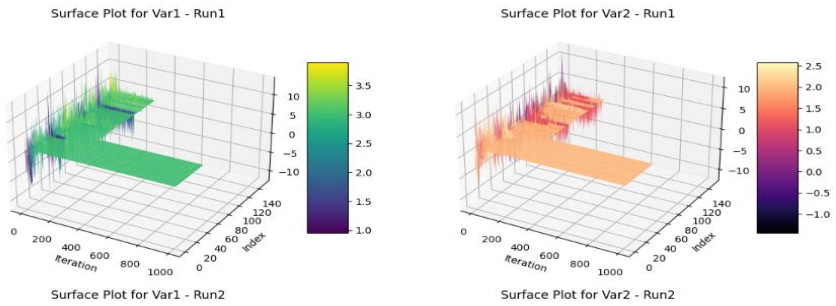


Figure.10. – Surface Plot for both Variables of Run 1

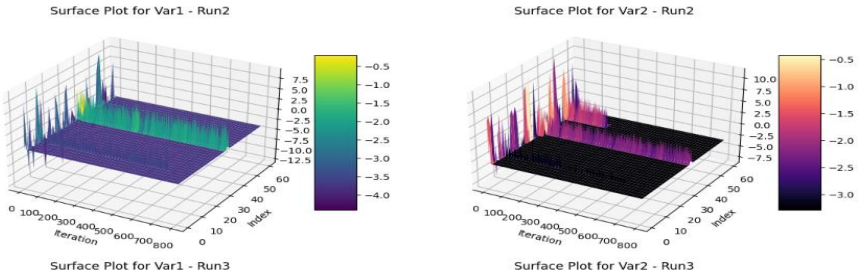


Figure.11. – Surface Plot for both Variables of Run 2

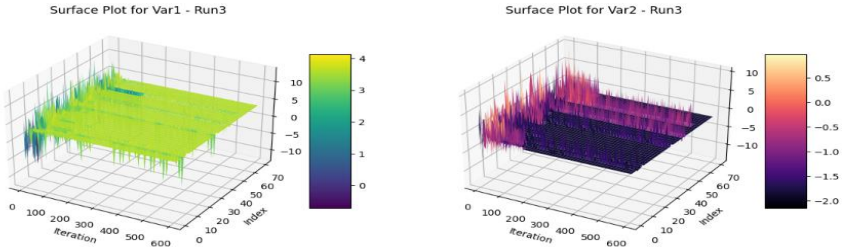


Figure.12. – Surface Plot for both Variables of Run 3

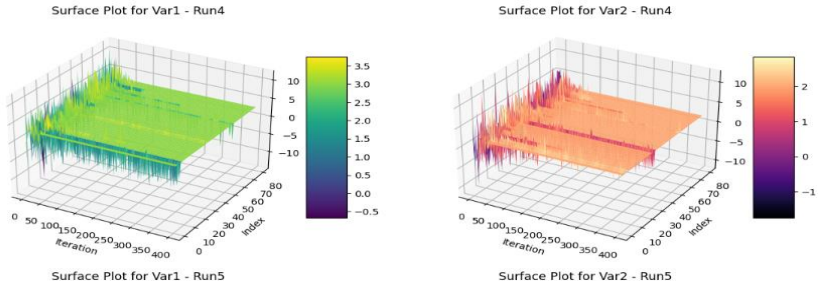


Figure.13. – Surface Plot for both Variables of Run 4

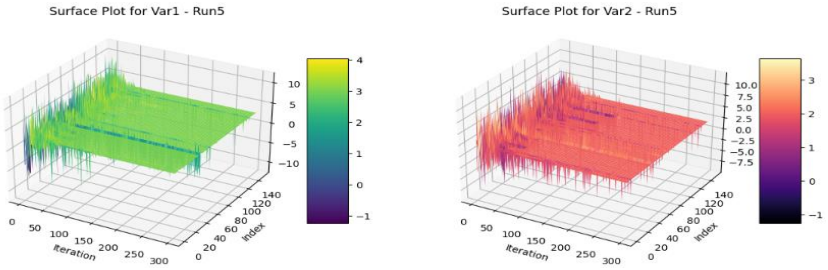


Figure.14. – Surface Plot for both Variables of Run 5

Surface plots further elucidate the optimisation process's dynamics. For Run 1, the surface plot displays significant fluctuations in cost values across iterations, suggesting less stability. In comparison, Run 2, 5's surface plot for Var1 shows a consistent descent towards lower cost values, indicating a stable and effective convergence pattern. The diverse behaviour observed in the surface plots for Runs 3, 4, and 5 suggests varying degrees of exploration and exploitation, with some runs exhibiting more pronounced peaks and troughs, potentially indicative of the swarm's struggle to balance search breadth against precise convergence.

When comparing these runs against each other, a different outcome can be assumed, Run 2 and 5's balanced parameter settings — moderate population size and inertia weight — seem to facilitate an effective exploration and exploitation trade-off, confirmed by the convergence

to lower cost values. This contrasts with the excessive parameter values seen in other runs, which, while ambitious, may not necessarily translate into better optimisation performance due to the risk of erratic search behaviour or premature convergence.

The analysis advocates that Run 2 and 5, with its moderate parameter configuration, yields the most effective optimisation process. It strikes a prudent balance between thorough search and efficient convergence, outperforming other runs with either conservative or excessive parameter values. This aligns with the established understanding that in PSO, extreme parameter values often detract from the algorithm's performance, whereas balanced parameters enhance it.

2.4 - Constrained Himmelblau & DE

Please see figures in the Appendices 'DE & Constrained Himmelblau'

In studying the effectiveness of DE algorithms for constrained optimisation, two rounds of parameter tuning and function enhancements were evaluated visually. The initial phase showed varied convergence behaviour across different parameters, indicated by peaks of varying heights in the best cost surface plot. After incorporating novel functions like adaptive mutation rates and local search, algorithmic stability notably improved. The contour plot displayed focused convergence of solution candidates, suggesting enhanced exploration capabilities while effectively exploiting the solution space. The modified algorithm demonstrated rapid convergence with stable trajectories across iterations, indicating successful avoidance of local minima and consistent identification of the global minimum. Iterative parameter adjustments, including tuning population size for diversity and expanding mutation factor range for broader exploration, were employed. Fine-tuning of crossover probability, periodicity of local search, immigration strategies, and tolerance thresholds further optimized the algorithm's balance between exploration and exploitation, enhancing its optimization proficiency. [45], [30], [36].

2.4.1 - DE/1/Best/bin

With regards to parameter adjustments, the increase in beta_min and pCR values in some parameter sets (such as Set 1 and Set 6) is contributing to a more vigorous exploration phase, which, in conjunction with a higher immigrant fraction, could be introducing beneficial diversity to the population. Moreover, the modulations in rateIncrease, maxBetaMin, and minBetaMax parameters are fine-tuning the algorithm's adaptive capabilities, facilitating a more responsive and dynamic search process. [44], [45], [48].

Set	MaxIt	nPop	beta_min	beta_max	pCR	immigrantFraction	rateIncrease	maxBetaMin	minBetaMax
1	1000	150	0.4	0.8	0.4	0.1	0.01	0.1	0.7
2	500	200	0.2	0.8	0.2	0.05	0.05	0.2	0.8
3	1000	150	0.1	0.9	0.3	0.2	0.09	0.3	0.9
4	500	150	0.5	0.8	0.1	0.15	0.1	0.4	0.7
5	1000	200	0.3	0.7	0.6	0.25	0.15	0.5	0.8
6	1500	150	0.6	0.8	0.5	0.4	0.2	0.6	0.9

Table.13. – Best Parameters of DE/1Rand/Bin

Spread and Overlap of Particles: At iteration one hundred, the particles are widely spread, which implies the algorithm is in an early exploration phase. There is a notable diversity in the search space, as particles are exploring various regions to locate areas of lower cost. By iteration 250, the particles start to converge towards certain regions, indicating the algorithm is beginning to exploit areas that potentially contain the optimum solution. The reduction in spread suggests that the algorithm is narrowing down on promising regions. At iteration five

hundred, the particle convergence becomes more pronounced. Particles are clustering around minima, which the algorithm has identified as potential solutions to the problem.

Convergence Over Time: The progression from iteration 100 to 500 shows how the algorithm transitions from exploration to exploitation. The density and clustering of particles at later iterations suggest that the algorithm has effectively reduced the search space to focus on specific regions.

Comparison Across Runs: The plots indicate that despite starting with different initial conditions or parameters (six different runs), the particles in all runs seem to display a similar behaviour over time, converging towards specific areas in the search space.

Algorithm Performance: The areas of convergence are consistent with the known global minimum of the Himmelblau function, this indicates reliable performance of the algorithm.

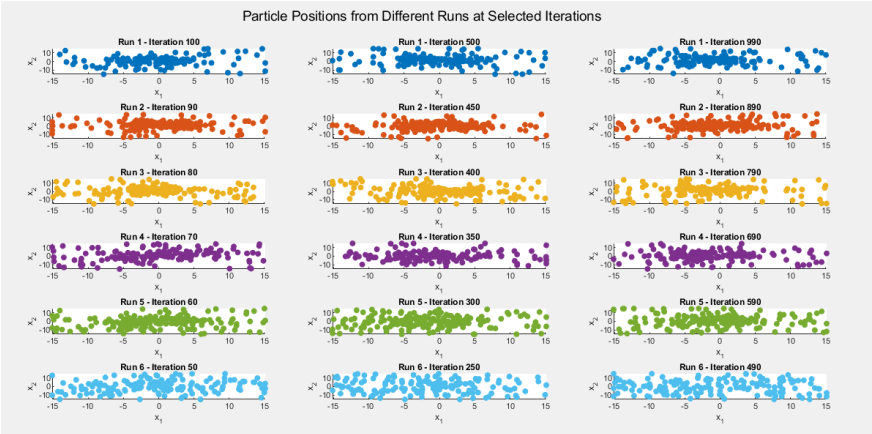


Figure.15. – DE/1/Best/Bin: Individual Run: Particle Positions From Different Runs Side By Side

In the Diversity and Best Cost Over Time for Each Run graph, a trend is observed, where the best cost for each run appears to be decreasing over time, demonstrating the algorithm's capability to improve solutions iteratively. However, this decrease in best cost is not necessarily accompanied by a proportional decrease in diversity. The parallel yet distinct trajectories of diversity and cost reduction accentuate the algorithm's nuanced balance between exploration and exploitation; as it converges on the optima, it retains a degree of population variance to facilitate the escape from potential sub-optimal solutions.

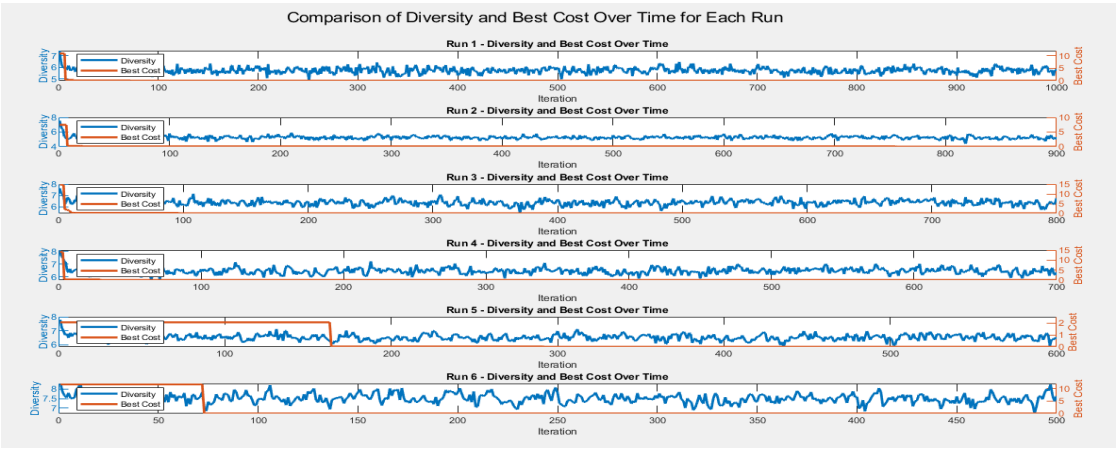


Figure.16. – DE/1/Best/Bin: Diversity and Best Cost Over Time for Each Run

The comparison between the two images elucidates the dynamic interplay within the DE algorithm—the balance between diversification and intensification. The stability in population diversity, coupled with the progressive improvement in best cost, evidences the algorithm's efficacy in maintaining a diverse gene pool while steering towards optimal solutions. This interplay is crucial, especially in multidimensional and rugged search landscapes, where the propensity to converge prematurely or to explore ineffectively could impede finding a global optimum.

2.4.2 - DE/2/best/bin

The visualisations of the differential evolution algorithm reveal its swift convergence and efficient optimisation across various runs. The 'Best Cost Surface' graph indicates that the algorithm quickly minimises costs, as shown by the initial peaks rapidly declining towards the global minimum. The 'Best Cost Trajectory' further underscores this by illustrating a rapid decrease to a plateau, suggesting the global minimum is achieved and maintained. Lastly, the 'Contour' plot shows the optimal solutions' locations within the search space, offering a visual confirmation of the algorithm's effectiveness in finding low-cost areas. (See Appendix for additional figures).

Set	MaxIt	nPop	beta_min	beta_max	pCR	immigrantFraction	rateIncrease	maxBetaMin	minBetaMax
1	1000	150	0.4	0.8	0.4	0.1	0.01	0.1	0.7
2	500	200	0.2	0.8	0.2	0.05	0.05	0.2	0.8
3	1000	150	0.1	0.9	0.3	0.2	0.09	0.3	0.9
4	500	150	0.5	0.8	0.1	0.15	0.1	0.4	0.7
5	1000	200	0.3	0.7	0.6	0.25	0.15	0.5	0.8
6	1500	150	0.6	0.8	0.5	0.4	0.2	0.6	0.9

Table.14. - – DE/2/Best/Bin Parameters

Turning to the "Diversity of the Population Over Time" graph, the varying levels of standard deviation across runs point to the algorithm's exploration capability. The maintained diversity suggests that the algorithm is not converging prematurely and maintains an exploration of the search space throughout its execution, which is crucial for avoiding local optima and ensuring robust optimisation. [49].

Comparing the Diversity and Best Cost Over Time chart, one notices the inverse relationship typically observed between diversity and best cost. Initially high diversity, which reduces over time, often correlates with the convergence towards a global minimum, as depicted by the reducing best cost. This trend highlights the algorithm's balanced approach between exploration and exploitation—maintaining sufficient diversity to explore the search space while converging towards an optimal solution.

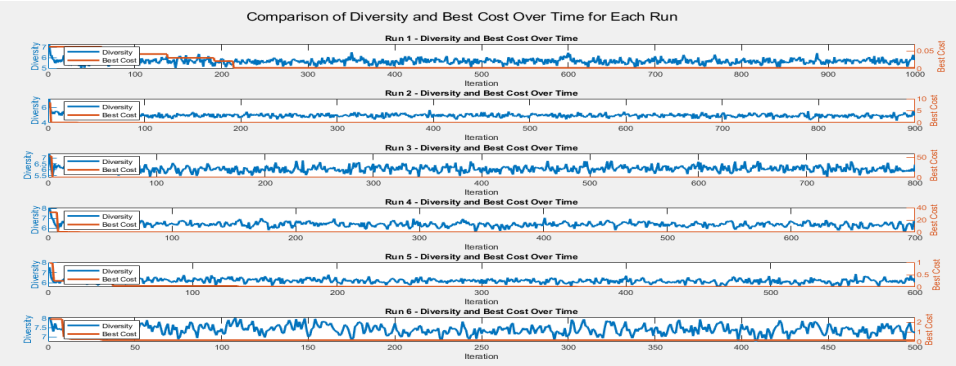


Figure.17. – DE/2/Best/Bin: Diversity Vs Best Cost Over Time for Each Run

Finally, the side-by-side visualisation of Particle Positions from Different Runs at Selected Iterations illustrates the convergence pattern of each run, with particles clustering more tightly around the global optima as iterations increase. This provides a detailed depiction of the algorithm's progress and its effectiveness in homing in on the optimal regions of the search space over time.

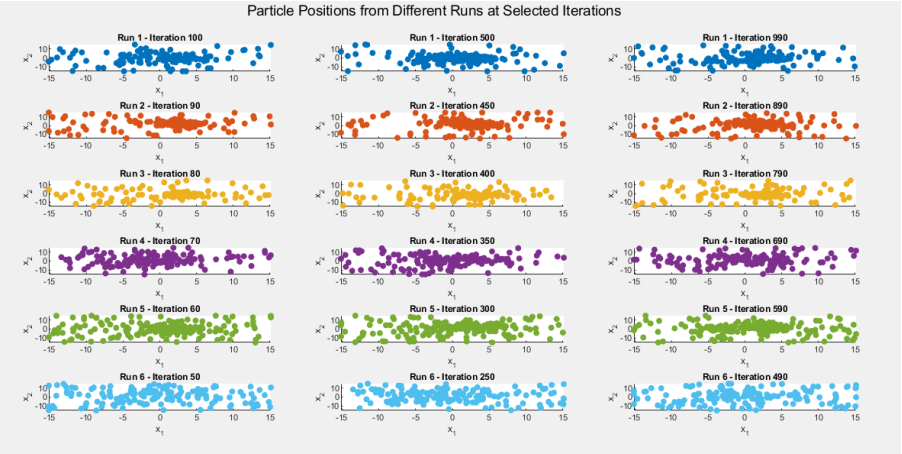


Figure.18. – DE/2/Best/Bin: Particle Positions from Different Runs at Selected Iterations

These visualisations present a compelling narrative of the algorithm's optimisation journey, characterised by efficient convergence, sustained minimisation of the best cost, and a strategic balance between the exploration of the search space and the exploitation of promising areas, culminating in the identification of global optima.

2.4.3. – DE 1 & 2 Comparison

Upon a thorough comparison of the DE1 and DE2 methods, DE2 demonstrates superior performance. This assertion is substantiated by a faster convergence rate towards the global minimum, as evidenced by the steep decline in best cost values at an earlier stage of iterations. Additionally, DE2 exhibits a more pronounced balance between exploration and exploitation—retaining population diversity to a commendable degree while steadily homing in on optimal solutions. The comparative analysis thus decisively favours DE2 for its enhanced efficiency and robust optimisation capabilities.

3. - Comparison, Interpretation and Conclusions

3.1 – Comparison & Interpretation

In an overarching synthesis of the results derived from the implementation of the Differential Evolution (DE), Particle Swarm Optimisation (PSO), and Real-Coded Genetic Algorithm (RGA) methods on both constrained and unconstrained problems, a nuanced perspective emerges. The DE method, particularly the DE2 variant, manifested a compelling proficiency, achieving rapid convergence with a balanced exploitation and exploration, as indicated by the steep and early decline in best cost values. This suggests that DE2 is highly effective for the Himmelblau Constrained problem.

The PSO algorithm demonstrated significant efficacy, particularly in the second run, where it achieved the lowest mean values, denoting a strong alignment with the optimisation objective. The visualisation of particle aggregation suggests that the swarm intelligence of the PSO is

conducive to identifying promising solution regions, further affirming the second run's optimality within the given parameters.

The RGA displayed a clear delineation between parameters and outcomes in the unconstrained scenarios, suggesting an accelerated convergence process when unburdened by constraints. The analysis indicates that the RGA's genetic operators are well-suited to optimising the Himmelblau function in an unconstrained setting, offering an incisive insight into parameter performance correlation.

When considering the Ackley function, DE1 outperformed DE2 by reaching a lower best cost. This result implies that DE1's parameters are well-calibrated for this specific problem. Nevertheless, DE2's performance could be optimised further to potentially surpass DE1 in different contexts. The PSO approach to the Ackley function demonstrated an adept convergence, with particles clustering near the global minimum, endorsing the algorithm's efficiency in solution refinement. In contrast, the RGA exhibited a tendency towards premature convergence in the presence of high crossover and mutation rates. An adaptive strategy might rectify this, enhancing the RGA's capability to consistently locate global optima.

While DE2 is designated as the superior method for the Himmelblau Constrained problem due to its fast convergence and balanced search strategy, for the unconstrained Ackley problem, DE1 is preferred for its lower best cost achievement. The PSO is distinguished for its efficiency in optimising the Ackley function. In parallel, RGA's unconstrained scenario provides valuable insights into the optimisation trajectory. These findings underline the importance of context and problem-specific tuning of algorithms to achieve optimal performance.

3.1.1 – Most Efficient Comparison

For the Himmelblau Constrained problem, DE2 is the standout method with its rapid convergence and ability to maintain diversity, suggesting an advanced capability for navigating through the constraints effectively. The RGA, while more transparent in its parameter performance relationship, shows a pronounced capacity for optimisation in the absence of constraints. Its genetic operators appear finely attuned to the unconstrained variant of the Himmelblau function, leading to fast initial improvements. PSO, with its exemplary performance in the second run, displays an efficient balance of exploration and exploitation, showing its proficiency in finding optimal solutions within the given tolerance levels.

In the context of the unconstrained Ackley problem, DE1 exhibits superior efficiency by achieving the lowest best cost, indicating an optimally tuned set of parameters for this specific scenario. PSO, through its intelligent swarming behaviour, demonstrates effective convergence to the global minimum, particularly in the second run, and RGA indicates a potential for fast convergence given the right conditions, although it may be prone to premature convergence without adaptive strategies.

Comparing efficiencies across all methods for a given tolerance:

DE: DE2 is more efficient for the constrained Himmelblau function, and DE1 for the unconstrained Ackley problem, reflecting DE's versatility in dealing with diverse types of landscapes and constraints. With PSO: Exhibits a consistent and robust performance, with the second run highlighting the algorithm's ability to utilise its swarm intelligence to converge to a solution within the tolerance levels efficiently. Finally, RGA: Shows potential for rapid improvement but requires careful parameter tuning and potentially adaptive strategies to avoid premature convergence and to maintain efficacy within the desired tolerance.

3.1.2 – Changing The Constraints Parameters

Empirical data suggests that a moderation of constraint penalties to a threshold of between 1000 and 500 yields an improved resolution of best costs, indicative of a more refined convergence towards the global minimum. This amelioration, however, plateaus beyond this constraint level, indicating a critical juncture where further reduction ceases to engender additional optimisation benefits. Consequently, it is postulated that the five hundred constraint penalty represents an optimal balance, fostering solution feasibility while averting the risk of constraint violations that might compromise the integrity of the optimisation process.

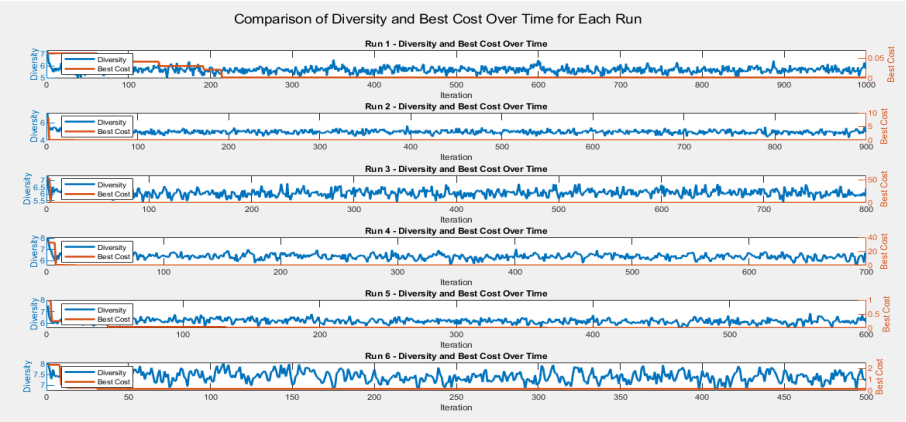


Figure.19. - Comparison of Diversity & Best Cost Over Time For A Penalty of five hundred

In the context of the PSO, applied to the same constrained Himmelblau function, a similar pattern emerges. Diminishing the constraint penalties enhances the propensity of the swarm to navigate the search space more freely, achieving lower best costs and improving the efficacy of the optimisation routine. However, an inflection point is observed around the five hundred constraints mark, beyond which the cost reductions become negligible. This suggests the existence of a penalty equilibrium that delicately negotiates between the rigour of constraint enforcement and the agility of search dynamics within the solution space. The DE method's adaptation to the constrained Himmelblau problem exhibits a corollary trend. With the constraint penalties calibrated to 1000 and 500, the algorithm demonstrates an optimised performance, aligning with lower best cost findings. This calibration represents a sweet spot, where the cost function's sensitivity to constraint violations is balanced against the algorithm's ability to effectively explore and exploit the problem landscape. As such, it is deduced that constraint penalties set around five hundred underpin an efficient DE algorithm.

3.1.3 – Changing the tolerance

Upon analysis of the optimisation visualisations and associated data with the adjusted tolerance parameters, a discernible impact on the performance of the algorithms is observable. The tightening of constraints from 1×10−5 to 1×10−6 has instigated a more rapid convergence and enhanced precision in particle clustering, particularly evident in the contour plots, which display a more concentrated localisation of particles around the optima. The trajectory plots reveal a steeper descent towards minimal best cost values at an accelerated pace, corroborating an increased algorithmic efficiency under more stringent tolerance. This suggests an improved delineation of the solution landscape, enabling the algorithms to bypass suboptimal regions with greater adeptness. The diversity plots, while demonstrating a variable impact on population diversity, indicate a maintenance of solution variability, suggesting that the optimisation process has not been detrimentally affected by over-constriction. The spreadsheet data reinforces these observations, revealing overall lower best cost figures

under the adjusted tolerance parameters. The implemented tolerance of 1e-6 has fostered a higher resolution in the search for global minima, by accentuating the sensitivity of the algorithms to finer gradations within the problem's landscape.

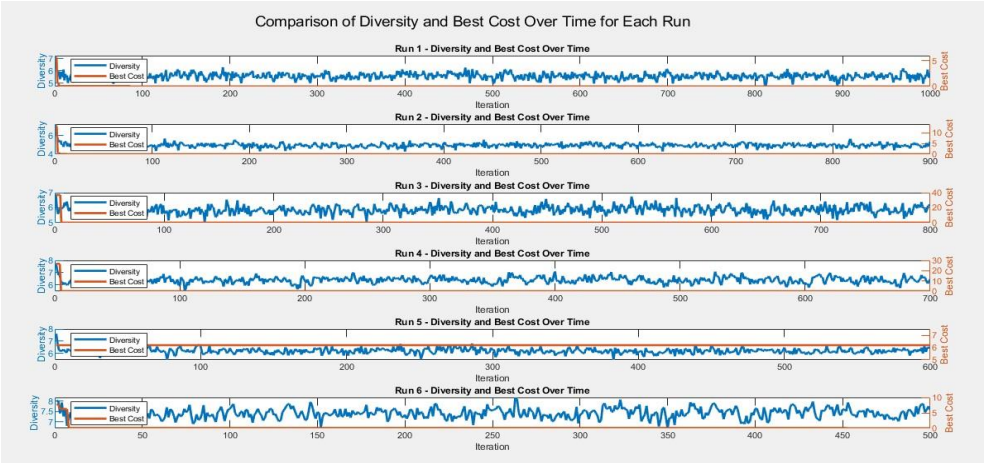


Figure.20. – Comparison of Best Cost & Diversity with a different tolerance

3.2 - Conclusion

Each method's unique characteristics, such as DE's balance of diversity and convergence, PSO's particle dynamics, and RGA's genetic operations, contribute differently to the optimisation process. The choice of the best method thus hinges on the particular problem being addressed, the constraints involved, and the desired balance between exploration and exploitation within the search space. Future investigations might delve into the hybridisation of these methods or the development of adaptive parameter tuning to further refine their optimisation capabilities. The determination of the 'best' method from the studied algorithms hinges on the context of their application and the criteria for 'best'. In the case of the Himmelblau Constrained problem, DE2 excels due to its swift convergence and equilibrium between exploration and exploitation. For the unconstrained Ackley problem, DE1 is preferred, considering its lower best cost value, which suggests effective parameter tuning for the given problem. Meanwhile, PSO highlights its strength in the Ackley function optimisation through intelligent particle movement and clustering, indicating efficient navigation of the search space.

One could argue that PSO's ability to maintain a trajectory towards the global minimum and its demonstrated efficiency might position it as a strong contender for the 'best' overall method. This is particularly evident in the second run, which aligns closely with the optimisation objectives. Additionally, the RGA's effectiveness in the unconstrained scenario with Himmelblau function and its potential for improved convergence through adaptive strategies cannot be discounted. Each method has its unique advantages that could make it the best choice under certain conditions. DE2's robustness in handling the constrained Himmelblau function, DE1's precision in Ackley function optimisation, and PSO's adaptability and intelligent search capabilities are all compelling in their own right. Therefore, a decision on the best method should consider the specific requirements and constraints of the problem at hand, along with the desired balance between explorative diversity and exploitative precision. The overarching conclusion would be to recommend DE2 for constrained problems requiring balanced search strategies and DE1 for unconstrained problems where lower best costs are a priority, with PSO serving as an excellent alternative for problems that benefit from dynamic particle-based search techniques.

Appendix

Unconstrained Ackley Function

As I had already created the Ackley function in week 2 exercises, please see the below figure.

Mathematical Formula:

$$f(x) = -a \cdot \exp\left(-b \cdot \sqrt{\frac{1}{2}(x_1^2 + x_2^2)}\right) - \exp\left(\frac{1}{2}(\cos(c \cdot x_1) + \cos(c \cdot x_2))\right) + a + \exp(1)$$

where $a = 20, b = 0.2,$ and $c = 2\pi.$

Formula.1. – Ackley Formula

RGA & Ackley

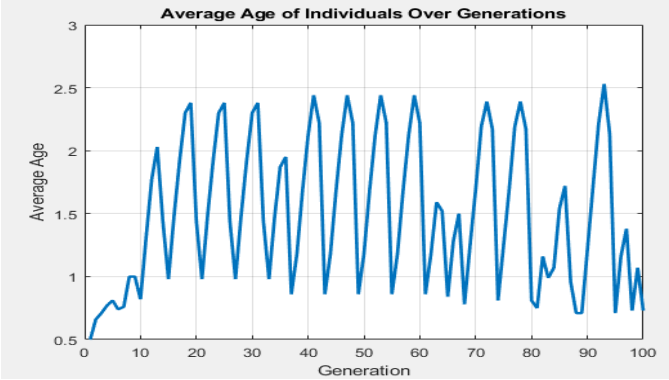


Figure.21. – RGA Ackley: Avg Age of Individuals Across Generations

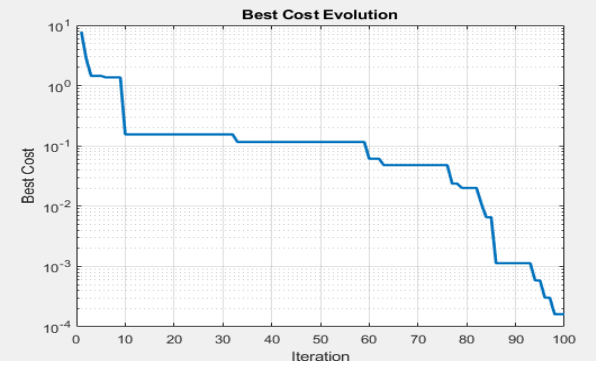


Figure.22. – RGA Ackley: Best Cost Evolution

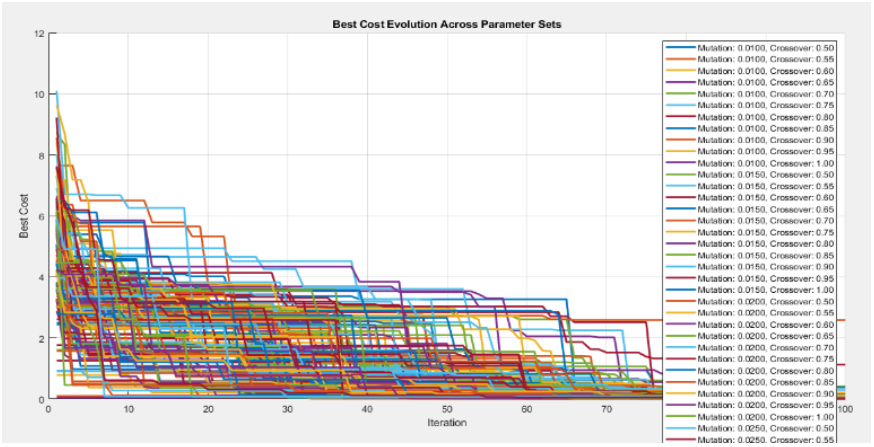


Figure.23. – RGA Ackley: Best Cost Evolution Across Parameter Sets

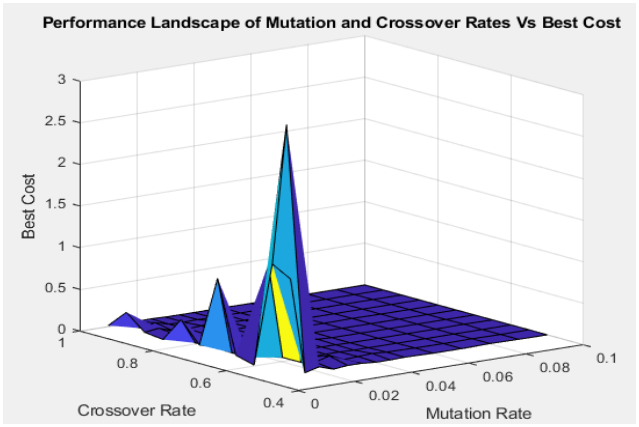


Figure.24. – RGA Ackley: Performance Landscape of Mutation and Crossover Rates Vs Best Cost

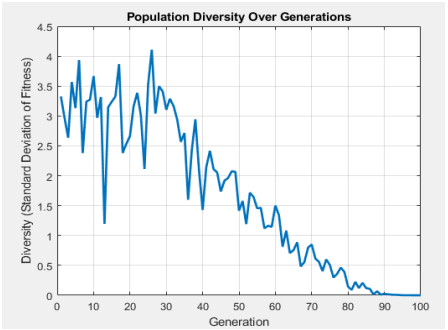


Figure.25. – RGA Ackley: Population Diversity Over Generations

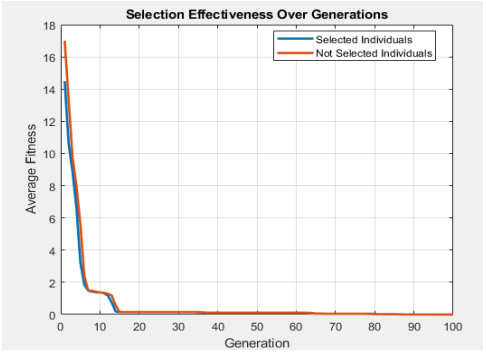


Figure.26. – RGA Ackley: Selection Effectiveness Over Generations

Additional Supporting Visualisations

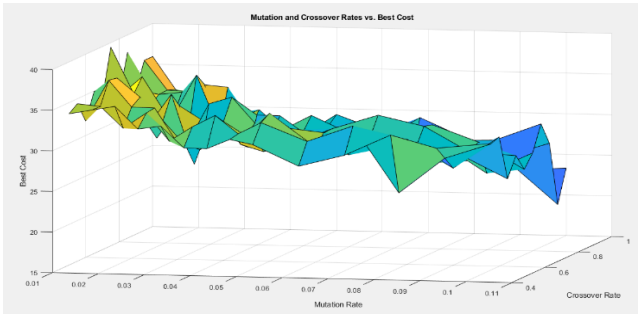


Figure.27. – Mutation & Crossover Vs Best Cost Landscape

Figures for 100-50-0.7-0.4-0.1:

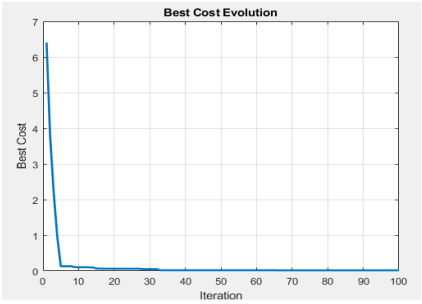


Figure.28. – Best Cost Evolution

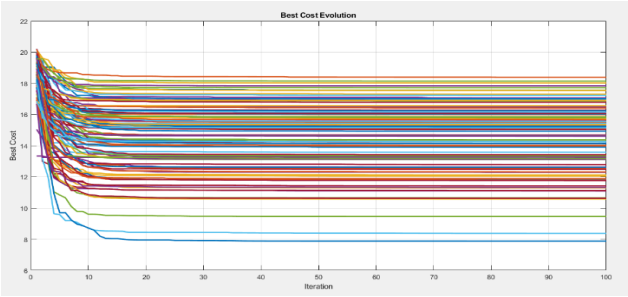


Figure.29. – All Runs Best Cost Evolution

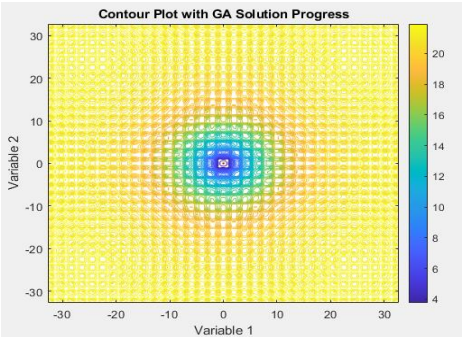


Figure.30. – Contour Plot Solution Progress

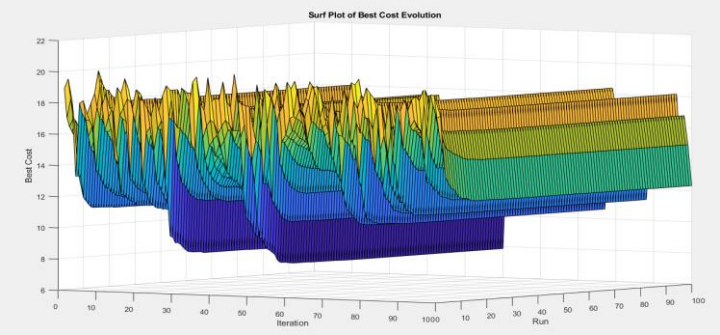


Figure.31. - Surf Plot of Best Cost Evolution

Figure.32. – Excel Best Cost Graph

Figures for 500-150-0.9-0.6-0.3:

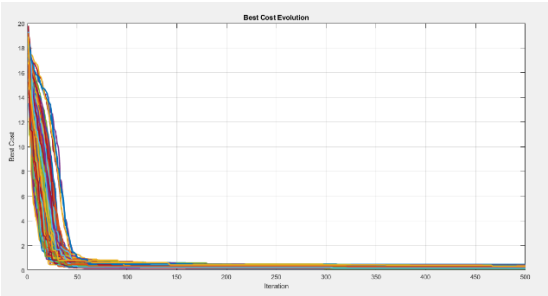
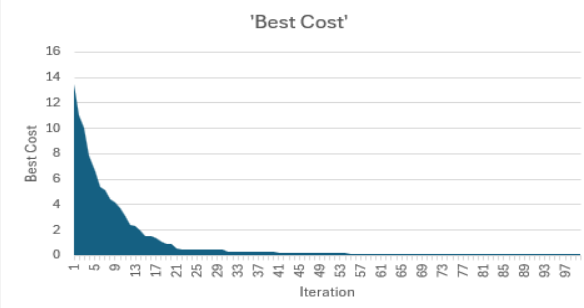


Figure.33. – All Runs Best Cost Evolution



Figures.34. – Excel Best Cost Graph

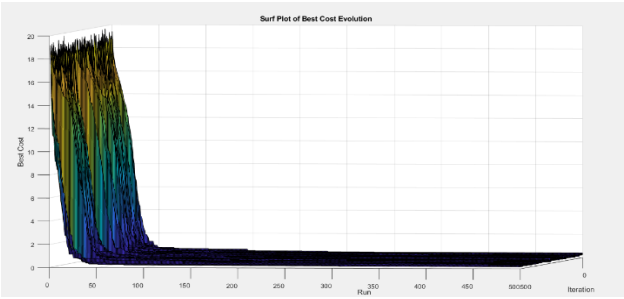


Figure.35. – Surf Plot of Best Cost Evolution

Figures for 1000-250-0.5-0.2-0.15:

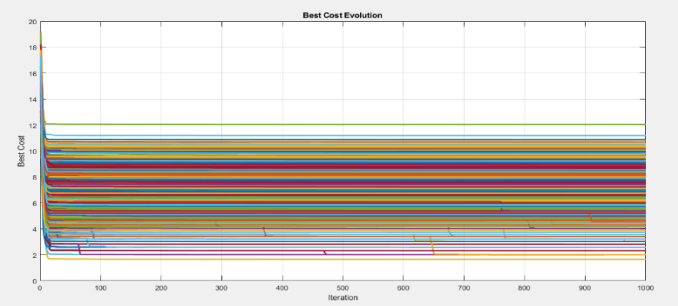


Figure.36. – All Runs Best Cost Evolution

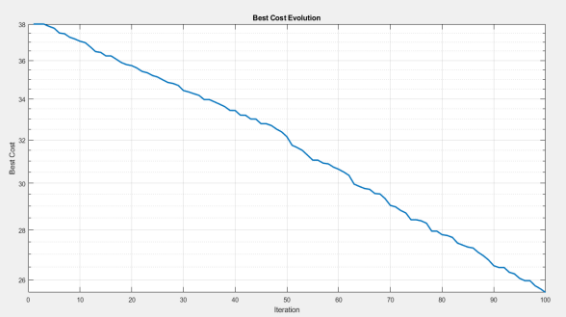


Figure.37. – Best Cost Evolution

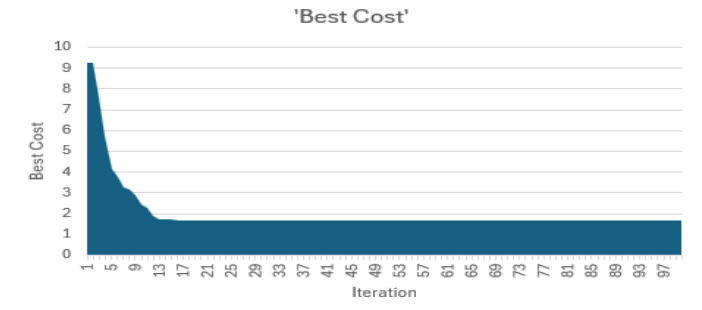


Figure.38. – Excel Best Cost Graph

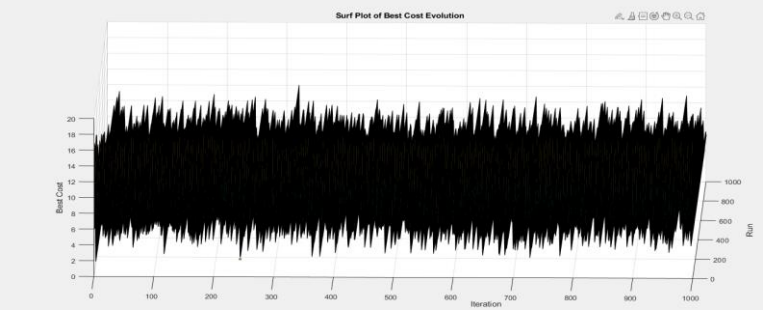


Figure.39. – Surf Plot of Best Cost Evolution (Too many iterations to depict nicely)

PSO & Ackley

```
Descriptive statistics for final positions of all particles:
         Particle  Iteration          Var1          Var2
count  150.000000      150.0  1.500000e+02  1.500000e+02
mean    74.500000      300.0  9.536404e-16  1.736249e-16
std     43.445368        0.0  1.163024e-14  1.982858e-15
min      0.000000      300.0 -4.584668e-15 -6.597692e-15
25%     37.250000      300.0 -1.378616e-16 -1.056919e-16
50%     74.500000      300.0 -1.568283e-17 -3.273903e-17
75%    111.750000      300.0  5.616848e-17  7.260814e-17
max    149.000000      300.0  1.419731e-13  1.997175e-14

Standard deviation of final positions (lower values indicate convergence):
Var1    1.163024e-14
Var2    1.982858e-15
dtype: float64
```

Figure.40. – Python Statistics Analysis

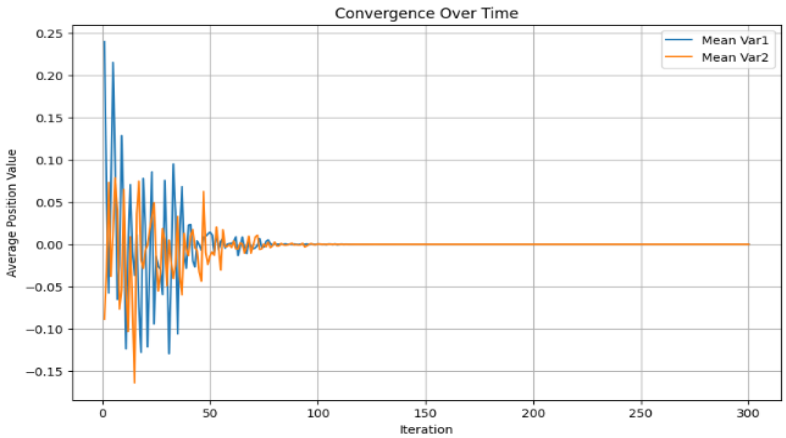


Figure.41. – Python Data Analysis: Convergence Over Time

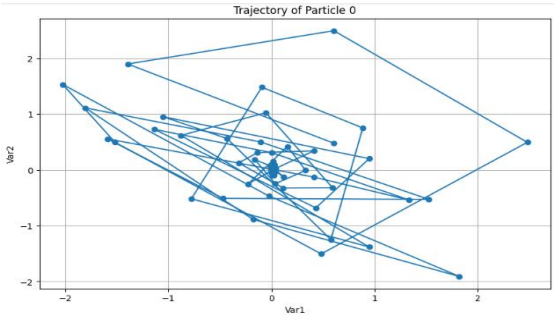


Figure.42. – Python Analysis: Trajectory of Particles

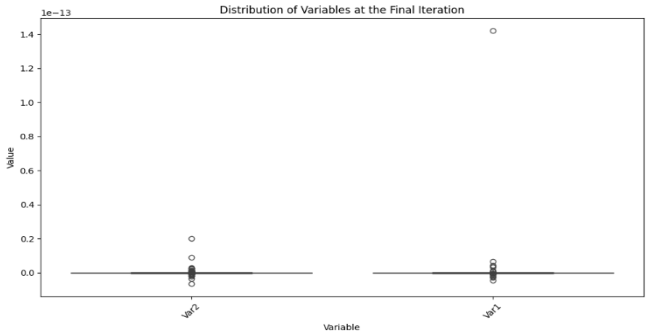


Figure.43. – Python Data Analysis: Distribution of Variables at Final Iteration

DE & Ackley

DE/1/Best/Bin Analysis:

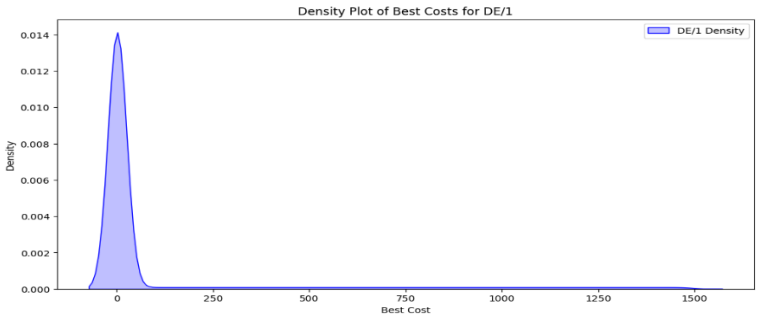


Figure.44. – Python Data Analysis: Best Cost Density

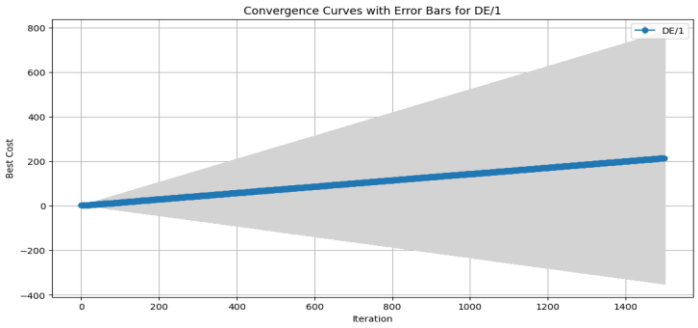


Figure.45. – Python Data Analysis: Convergence of DE/1

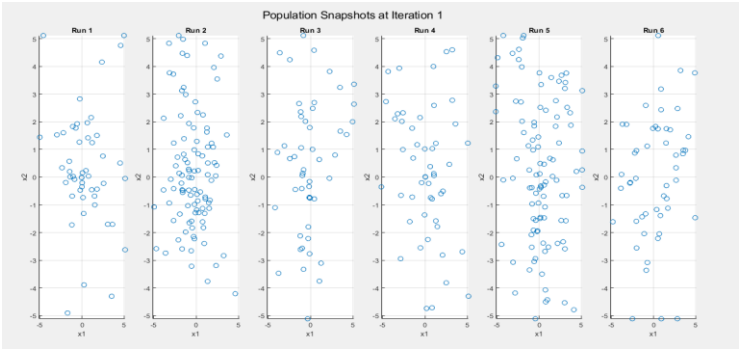


Figure.46. – Population Side By Side Snapshots

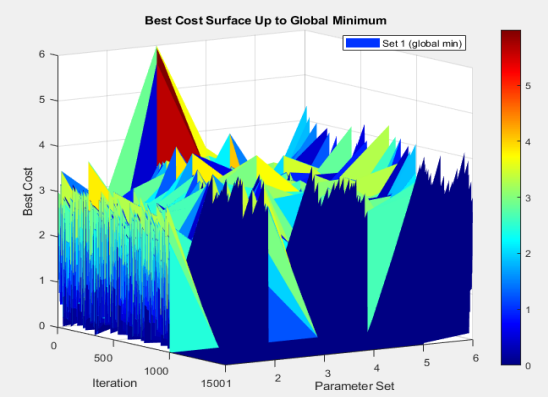


Figure.47. – Best Cost Surface Plot

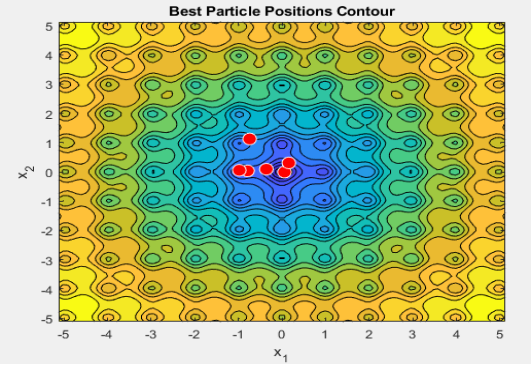


Figure.48. - Best Particle Positions Contour

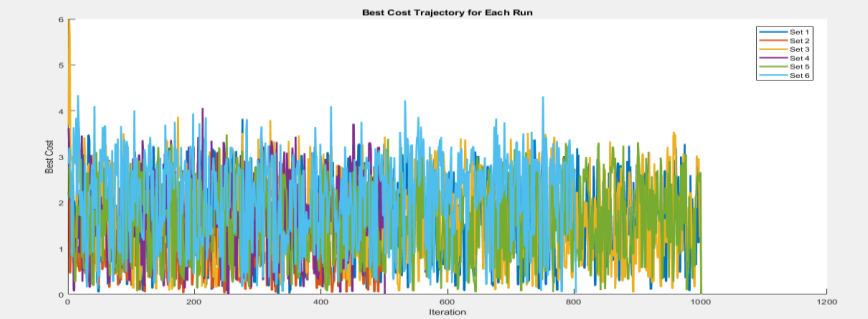


Figure.49. – All Runs Best Cost Trajectory

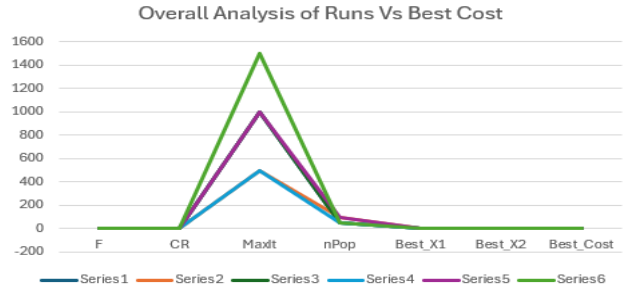


Figure.50. – Excel Overall Run Analysis of Best Cost

DE/2/Best/Bin Analysis:

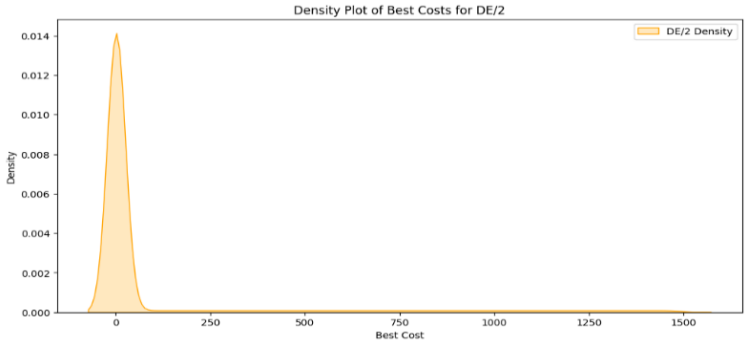


Figure.51. – Python Data Analysis: DE2 Density Plot

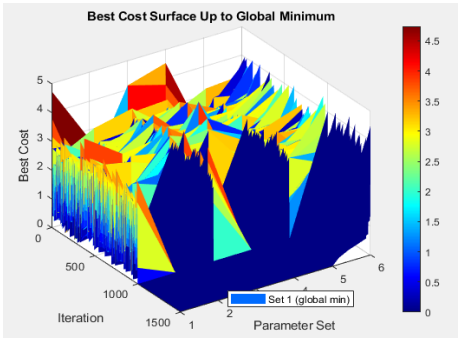


Figure.52. – Best Cost Surface Plot

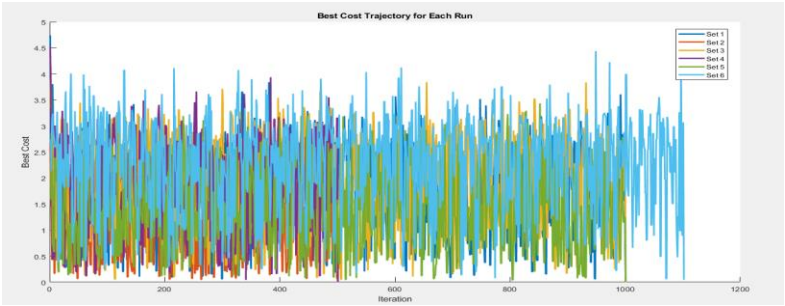


Figure.53. – All Runs Best Cost Trajectory

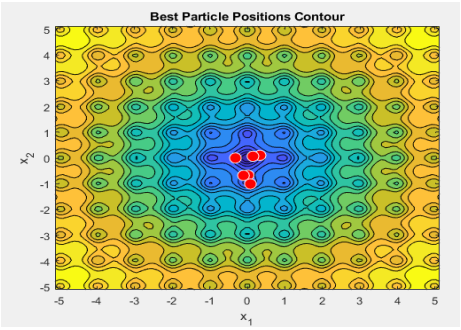


Figure.54. – Best Particle Positions Contour

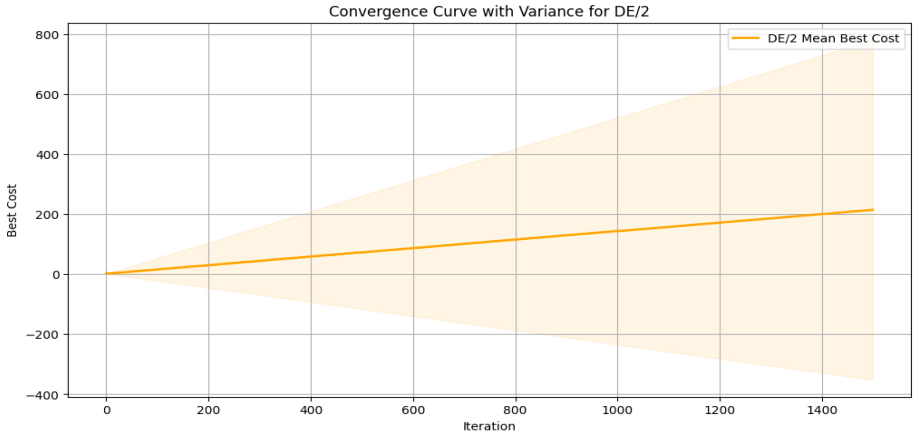


Figure.55. – Python Data Analysis: DE2 Convergence

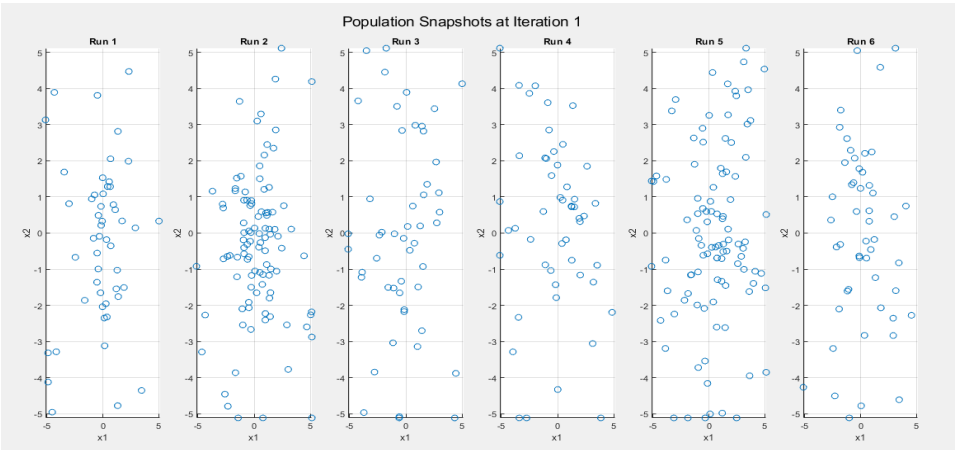


Figure.56. – Population Snapshots of Each Run Side By Side

Constrained Himmelblau Function With RGA, PSO, & DE

In the preparation for implementing the constrained Himmelblau problem, a technical workflow was established using Python within the Google Colab environment. Within this environment, the Python programming language was employed to perform a series of computational tasks to analyse and visualise the constraints of the Himmelblau problem. Specifically, the NumPy library was utilised to create a dense grid of points that spanned the area of interest in the problem's domain. This was facilitated through the linspace and meshgrid functions, which generated a two-dimensional grid of x1 and x2 values within the specified range.

With the grid in place, the constraint functions were vectorised and evaluated across all grid points. This included the circle constraint, defined by the equation x1/2 +x2/2 ≤225, and two linear constraints, x 1 +x 2 ≤10 and 4x 1 −3x 2 ≥24, corresponding to the problem's specified conditions. The evaluation yielded Boolean masks that indicated whether each point on the grid satisfied the individual constraints.

These masks were then logically combined using an 'AND' operation to ascertain the points that simultaneously satisfied all constraints, effectively demarcating the feasible region. This region was highlighted on a plot using Matplotlib, a comprehensive library for creating static, interactive, and animated visualisations in Python. The contour and contourf functions generated contour lines for the circle constraint and filled the feasible region, with the plot function delineating the linear constraints. A grey overlay indicated the feasible region, providing a clear visual aid for the subsequent optimisation process.

Constrained Himmelblau Maths:

The function Himmelblau_Constrained(x) calculates the value of the objective function with constraints based on the Himmelblau function. Here is the mathematical representation:

Objective function:

f(x)=(x2/1+x2−11)^2+(x1+x2/2−7)^2

Constraints:

Circle constraint:

g1(x)=x1/2+x2/2−225≤0

Linear constraint:

g2(x)=x1+x2−10≤0

Debs constraint:

g3(x)=4x1−3x2−24≤0

Penalty parameters:

k1=1000 (Penalty parameter for the circle constraint)

k2=1000 (Penalty parameter for the linear constraint)

k3=1000 (Penalty parameter for the Debs constraint)

Decision Variables: $x1, x2$

$g1 = x_1^2 + x_2^2 - 225$ (Circle constraint)

$g2 = x_1 + x_2 - 10$ (Linear constraint)

$g3 = 4x_1 - 3x_2 - 24$ (Debs constraint)

$k1, k2, k3 = 1000$ (Penalty parameters)

$f = (x_1 + x_2^2 - 7)^2 + (x_1^2 + x_2 - 11)^2$ (Objective function)

Cost function with penalties:

$$\text{cost} = k_1 \max(0, x_1^2 + x_2^2 - 225)^2 + k_2 \max(0, x_1 + x_2 - 10)^2 + k_3 \max(0, 4x_1 - 3x_2 - 24)^2 + (x_1 + x_2^2 - 7)^2 + (x_1^2 + x_2 - 11)^2$$

Formula.2. – Mathematical Formula for The Constrained Himmelblau Function Maths

RGA & Constrained Himmelblau

Constrained Vs Unconstrained Himmelblau Function

After testing across both the Constrained and Unconstrained Himmelblau Function (my curiosity wanted to see the difference), I noted the following; the unconstrained nature of the problem typically allows for a more straightforward search process. The absence of constraints leads to clearer parameter performance relationships and potentially quicker convergence to optimal solutions. These visualisations suggest that while the RGA is effective in both constrained and unconstrained environments, the absence of constraints provides a more transparent understanding of the influence of genetic operators and parameters on the optimisation process. Further comparative analysis could involve quantitative measures such as the number of generations to convergence and the variability in final solution quality to provide a more nuanced understanding of the differences between the constrained and unconstrained scenarios.

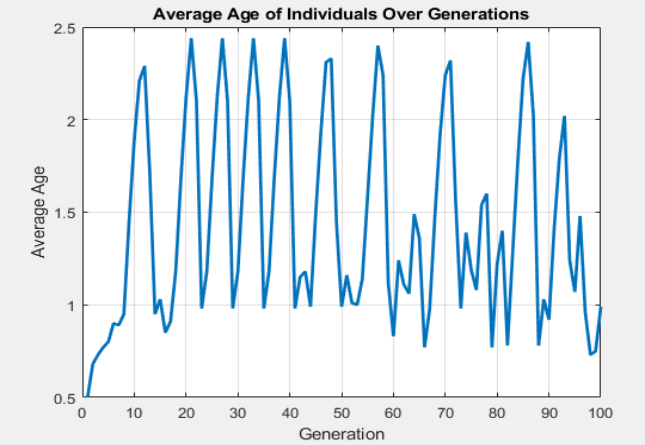


Figure.57. – Avg Age of Individuals Over Generations

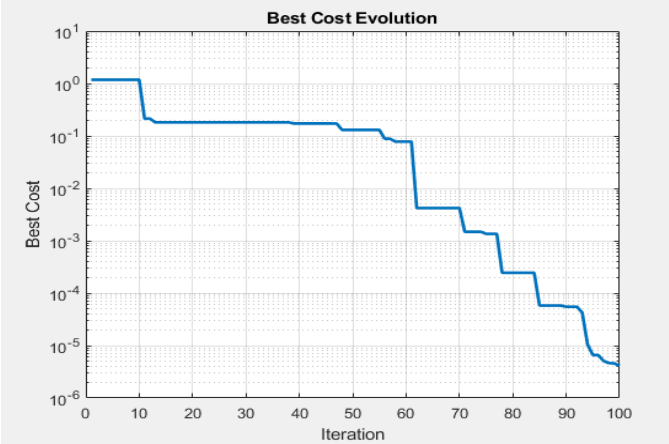


Figure.58. – Best Cost Evolution

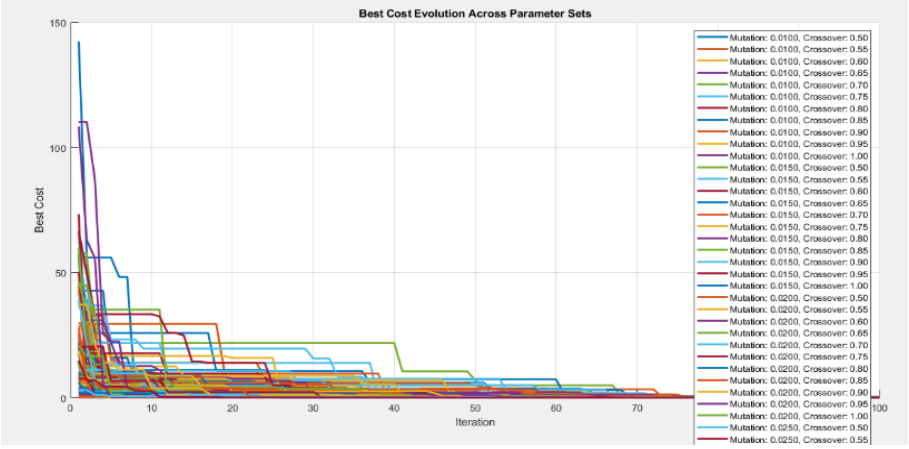


Figure.59. – Best Cost Evolution Across Parameter Sets

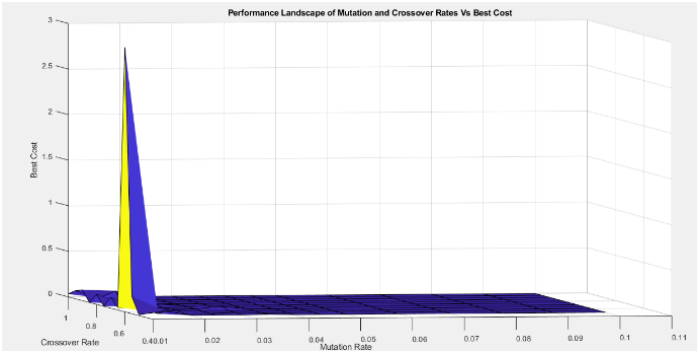


Figure.60. – Performance Landscape of Mutation & Crossover Vs Best Cost

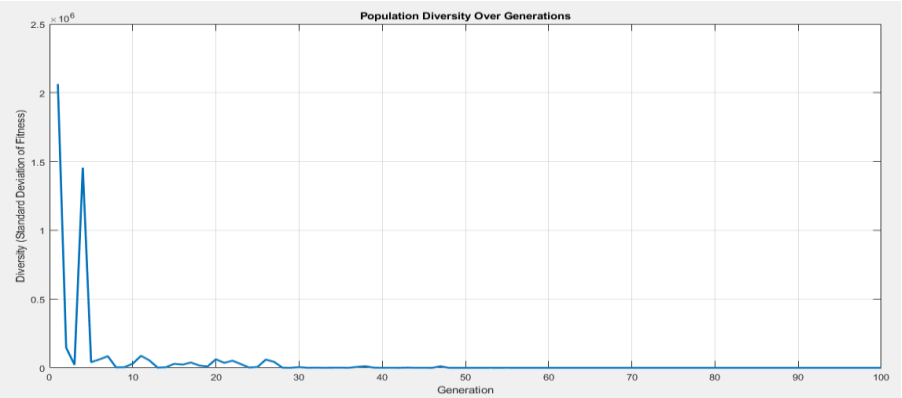


Figure.61. – Population Diversity Over Generations

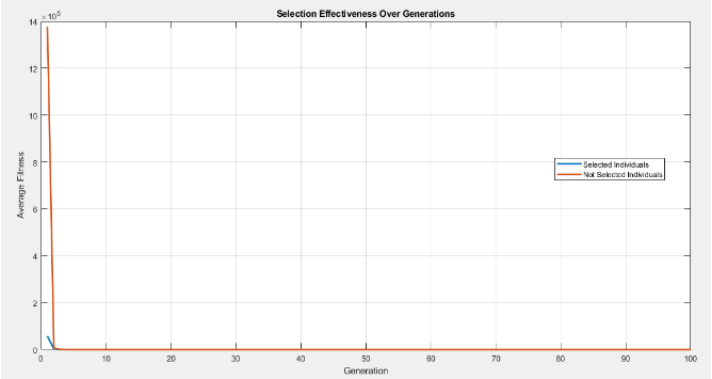


Figure.62. – Selection Effectiveness Over Generations

PSO & Constrained Himmelblau

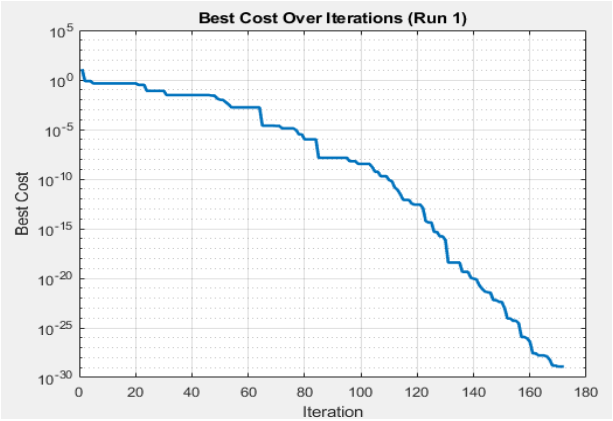


Figure.63. – PSO & Constrained Himmelblau: Best Cost Evolution

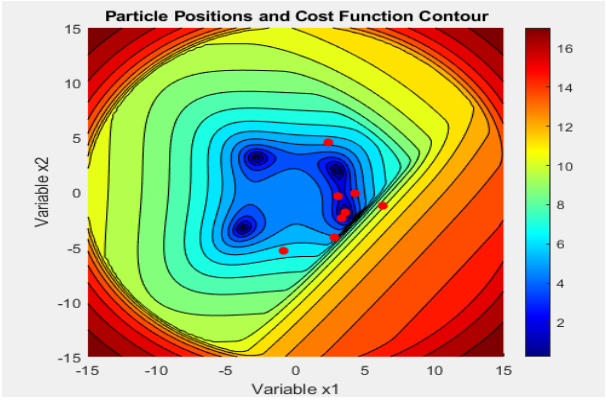


Figure.64. – PSO & Constrained Himmelblau: Contour Plot

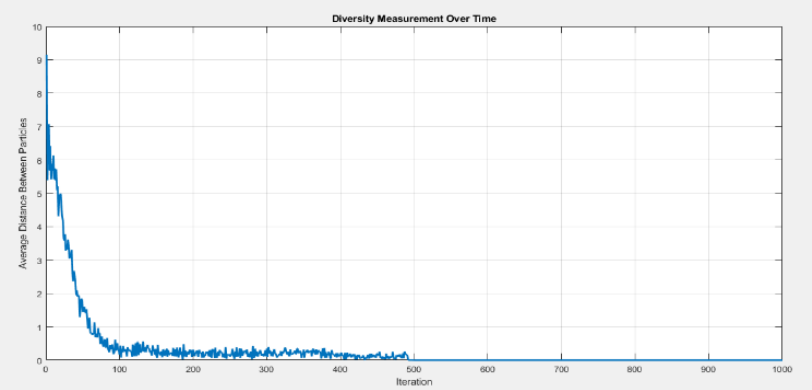


Figure.65. – PSO & Constrained Himmelblau: Diversity Measurement Over Time

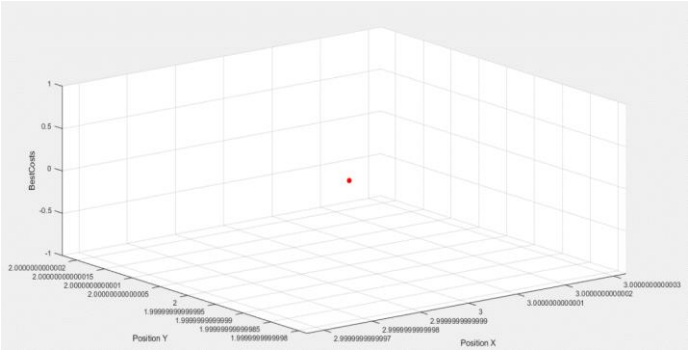


Figure.66. – PSO & Constrained Himmelblau: Final Swarm Position

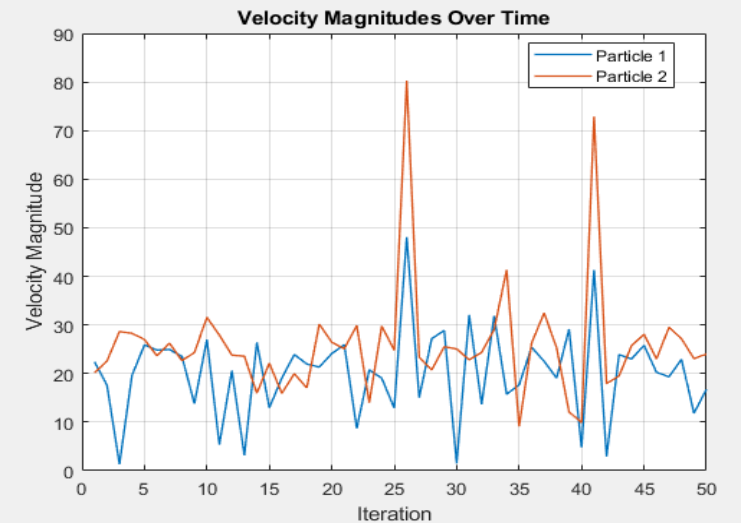


Figure.67. – PSO & Constrained Himmelblau: Velocity Magnitudes Over Time

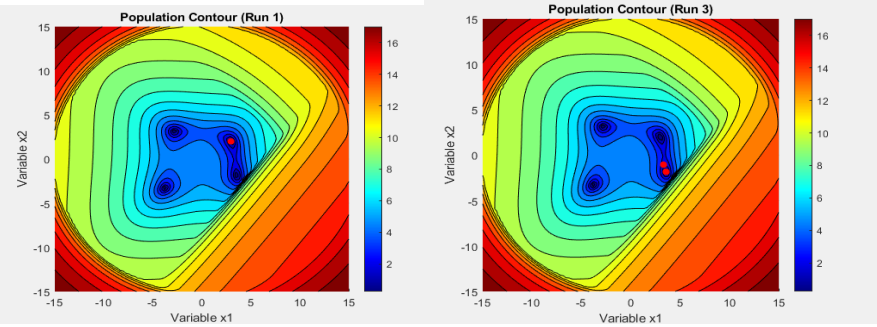


Figure.68. – PSO & Constrained Himmelblau: Contour plots of Run1 & 3 Depicting 1000 iterations, 50 and 70 population, identical w, wdamp, c1, and c2 values

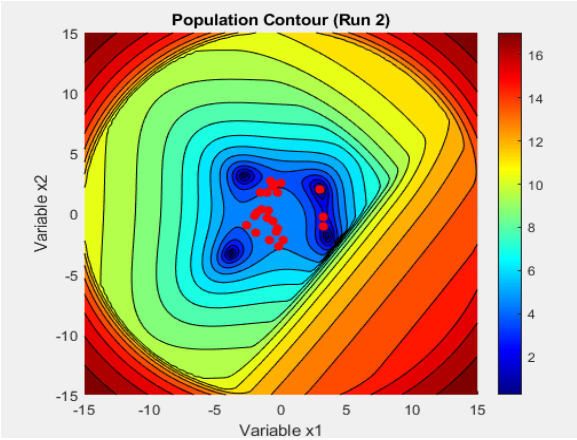


Figure.69. – PSO & Constrained Himmelblau: Run 2 Contour Plot of 1000 iterations, 100 population, w: 0.99, wdamp: 0.75, c1: 0.5, c2: 0.25

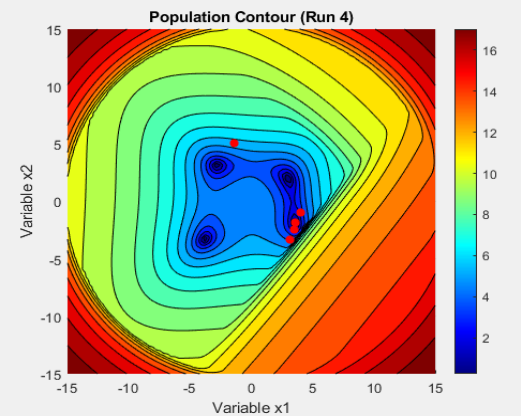


Figure.70. – PSO & Constrained Himmelblau: Run 4 Contour Plot - 1000 iterations, 80 population, same w, wdamp, c1, c2 as Run 1 and 3

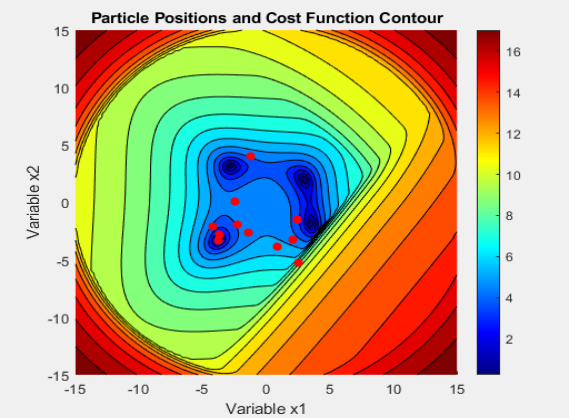


Figure.71. – PSO & Constrained Himmelblau: Run 5 Contour Plot - 1000 iterations, 150 population, same w, wdamp, c1, c2 as Run 1, 3, and 4):

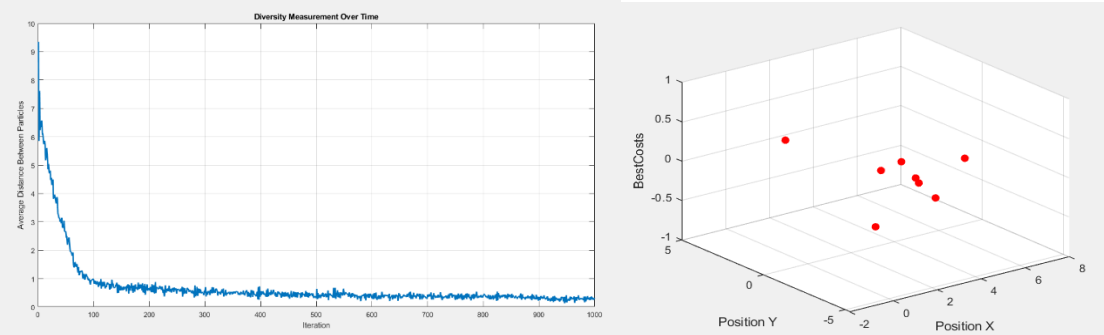


Figure.72. – PSO & Constrained Himmelblau: Diversity Measurement & Final Swarm Particle Positions

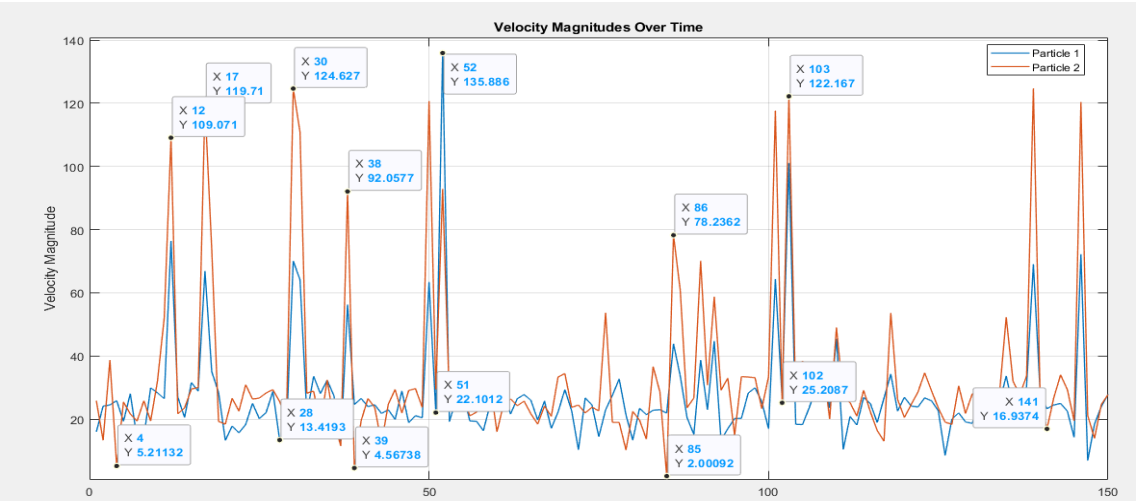


Figure.73. – PSO & Constrained Himmelblau: Velocity Magnitude Over Time

MaxIt	nPop	w	wdamp	c1	c2
800	50	1	0.99	2	2
800	100	0.99	0.75	0.5	0.25
800	70	1	0.99	2	2
800	80	1	0.99	2	2
800	150	1	0.99	2	2

Table.15. – PSO & Himmelblau Parameters

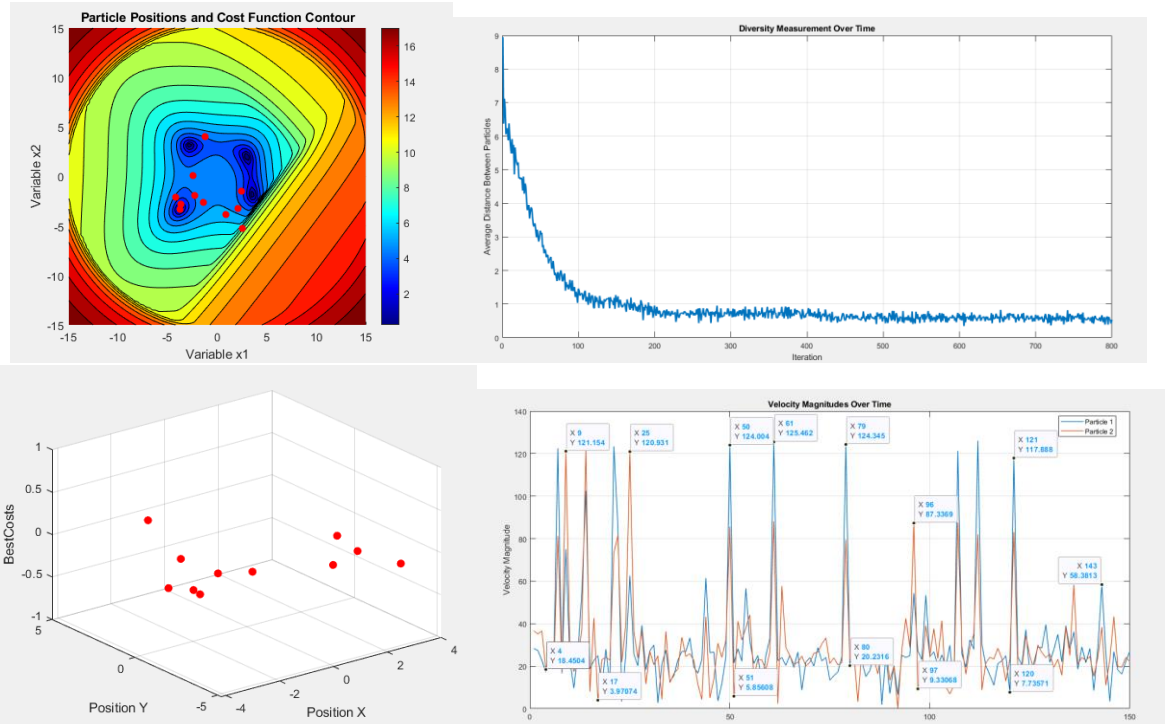


Figure.74. - PSO & Himmelblau 800 Iterations Visualisations: L-R – Contour Plot, Diversity Over Time Plot, Final Swarm Position, and Velocity Magnitude Over Time

Parameter Tuning Analysis (1000 Iterations):

First Configuration:
➢ Population (nPop): 250

➢ Inertia Weight (w): 0.50
➢ Damping Ratio (wdamp): 0.50
➢ Personal Learning Coefficient (c1): 0.01
➢ Social Learning Coefficient (c2): 1.00
Second Configuration:
➢ Population (nPop): 100
➢ Inertia Weight (w): 0.99
➢ Damping Ratio (wdamp): 0.25
➢ Personal Learning Coefficient (c1): 0.5
➢ Social Learning Coefficient (c2): 0.25
Third Configuration:
➢ Population (nPop): 70
➢ Inertia Weight (w): 1.00
➢ Damping Ratio (wdamp): 0.75
➢ Personal Learning Coefficient (c1): 2.00
➢ Social Learning Coefficient (c2): 1.50
Fourth Configuration:
➢ Population (nPop): 40
➢ Inertia Weight (w): 1.50
➢ Damping Ratio (wdamp): 2.99
➢ Personal Learning Coefficient (c1): 1.50
➢ Social Learning Coefficient (c2): 2.00
Fifth Configuration:
➢ Population (nPop): 150
➢ Inertia Weight (w): 2.00
➢ Damping Ratio (wdamp): 1.99
➢ Personal Learning Coefficient (c1): 1.00
➢ Social Learning Coefficient (c2): 2.50

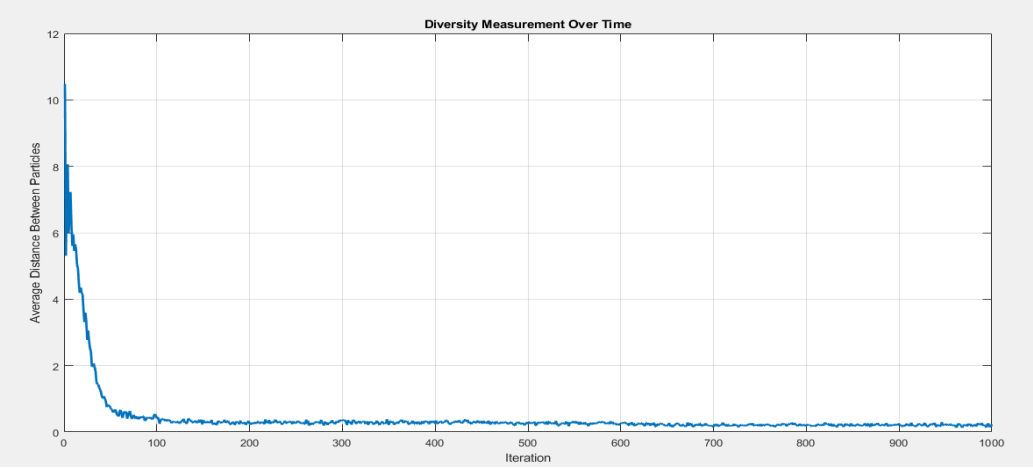


Figure.75. – PSO & Constrained Himmelblau: Diversity Measurement Over Time

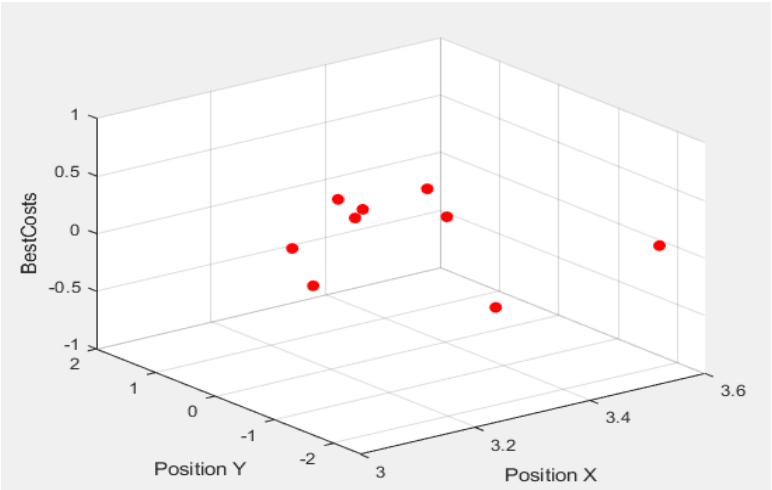


Figure.76. – PSO & Constrained Himmelblau: Final Swarm Particle Positions

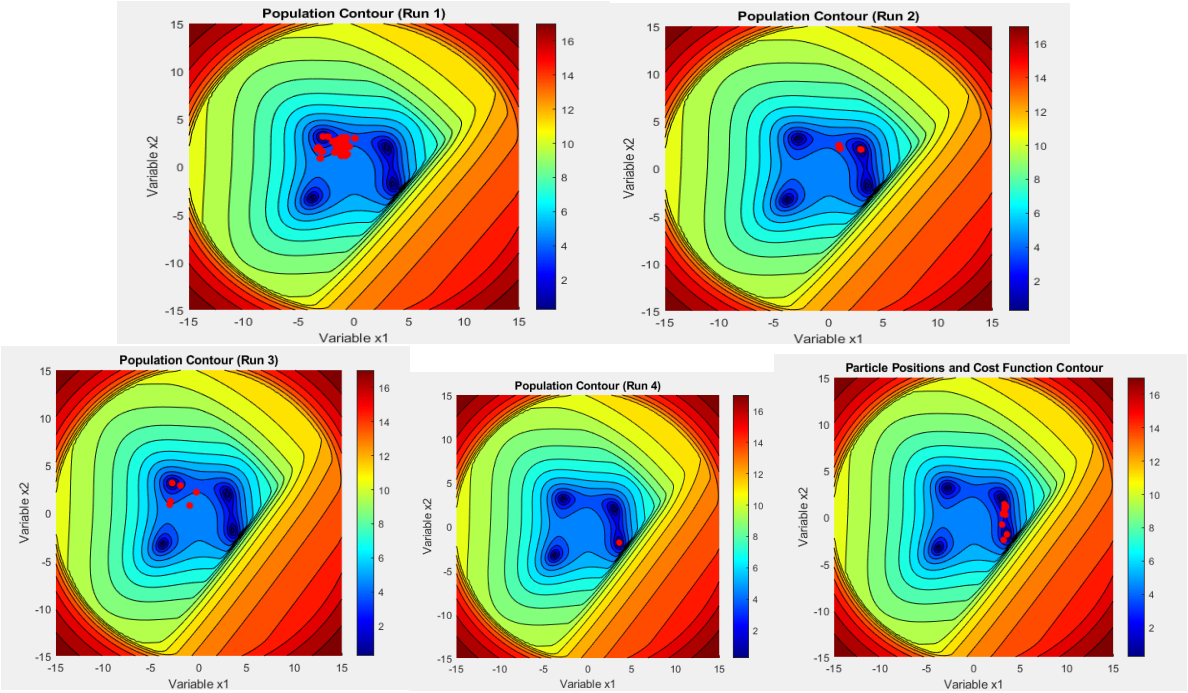


Figure.77. – PSO & Constrained Himmelblau: Contour Plots of Runs 1-5 (L-R)

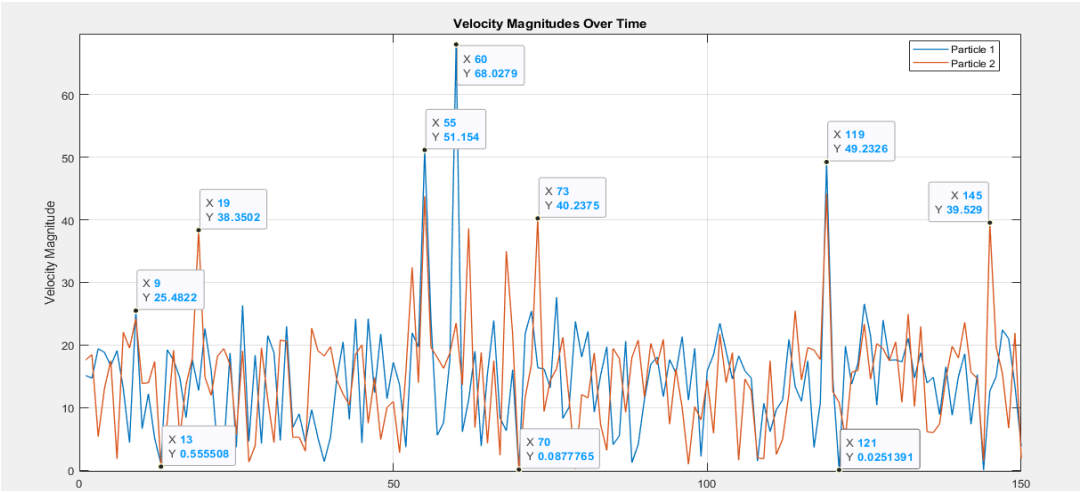


Figure.78. – PSO & Constrained Himmelblau: Velocity Magnitude Over Time

DE & Unconstrained Himmelblau

DE/1/rand/bin

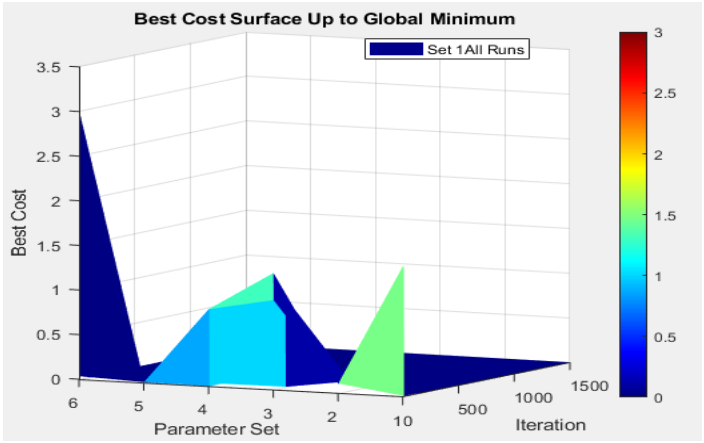


Figure.79. – DE/1/rand/bin Surface Plot Post Advanced Techniques Implementation & Post Param Tuning

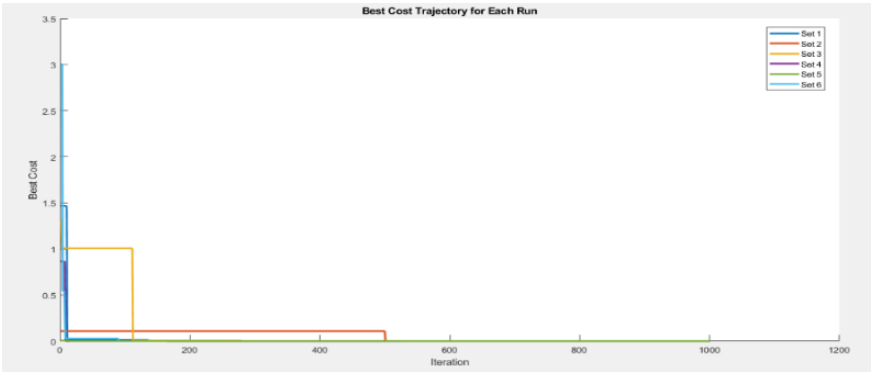


Figure.80. – DE/1/rand/bin Trajectories Post Advanced Techniques Implementation & Post Param Tuning

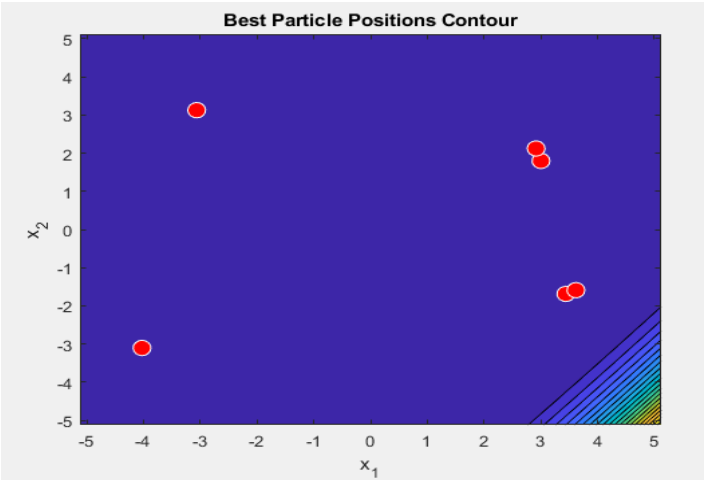


Figure.81. – DE/1/rand/bin Contour Post Advanced Techniques Implementation & Post Param Tuning

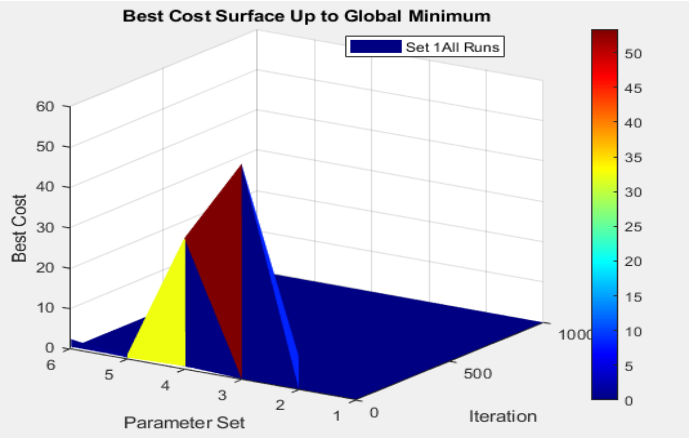


Figure.82. - DE/2/rand/bin Best Cost Surface Plot

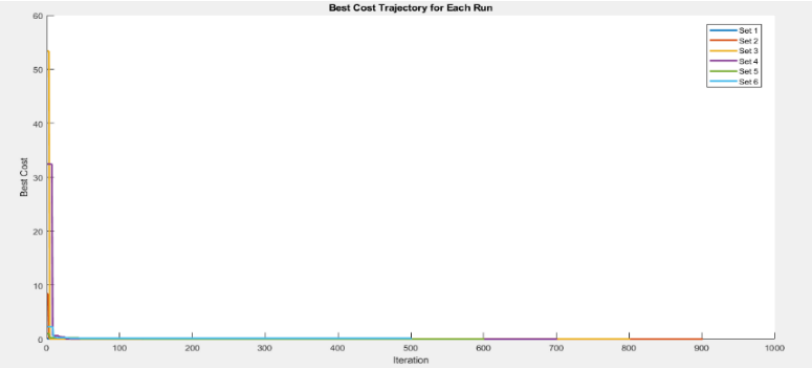


Figure.83. - DE/2/rand/bin Best Cost Trajectory

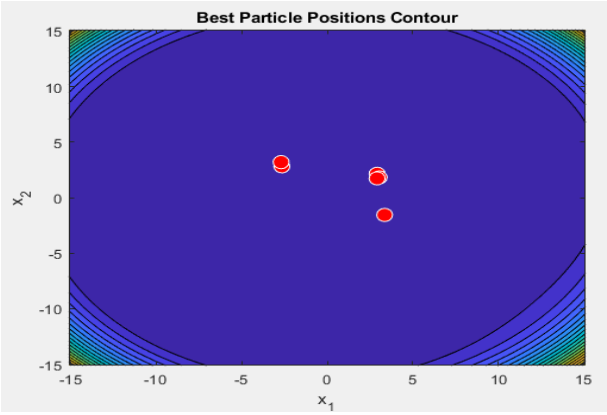


Figure.84. - DE/2/rand/bin Contour Plot

Additional Appendices – Constrained g06, g08, Welded Beam

Constrained Welded Beam

Shear stress (τ) must not exceed a maximum value (τ_{max}): The shear stress is calculated from the applied load and the geometry of the welded joint.

Bending stress (σ) must not exceed a maximum value (σ_{max}): The bending stress is calculated based on the load applied at the end of the beam and its cross-sectional geometry.

Buckling load (P_c) must be greater than the applied load (P): This ensures the beam does not buckle under the applied load.

Deflection (δ) must not exceed a maximum value (δ_{max}): The deflection is determined by the applied load, the length of the beam, and its material and cross-sectional properties.

Figure.85. – Welded Beam Formula Description

```python
import numpy as np
import matplotlib.pyplot as plt

def constraint_welded_beam(x):
    # Decompose the decision variables
    h = x[0]  # Thickness of the weld
    l = x[1]  # Length of the attached part of the beam
    t = x[2]  # Thickness of the beam
    b = x[3]  # Width of the beam

    # Constants and Parameters
    P = 6000  # Load in lbs
    L = 14  # Length of the beam in inches
    E = 30e6  # Modulus of elasticity in psi
    G = 12e6  # Shear modulus in psi

    # Derived Calculations
    Pc = (4.013 * E * np.sqrt((t**2 * b**6) / 36)) / (L**2) * (1 - t * L / (2 * np.pi) * np.sqrt(E / (4 * G)))
    M = P * (L + (l / 2))
    R = np.sqrt((l**2 / 4) + ((h + t)**2 / 4))
    J = 2 * (np.sqrt(2) * h * l * ((l**2 / 12) + ((h + t)**2 / 4)))

    # Calculating tau (shear stress), sigma (bending stress), delta (deflection)
    tau = (P / (np.sqrt(2) * h * l)) + ((M * R) / J)
    sigma = (6 * P * L) / (b * t**2)
    delta = (4 * P * L**3) / (E * b * t**3)

    # Constraints
    tau_max = 13600  # Maximum shear stress in psi
    sigma_max = 30000  # Maximum bending stress in psi
    delta_max = 0.25  # Maximum deflection in inches
    c = np.array([tau - tau_max, sigma - sigma_max, Pc - P, delta - delta_max])

    # No equality constraints in this problem
    ceq = []

    return c, ceq

# Example usage
x = [0.2, 100, 10, 10]  # Example decision variables
c, ceq = constraint_welded_beam(x)
print("Constraints (c):", c)
print("Equality constraints (ceq):", ceq)

# Plotting the mathematical formula
eqn = r"$\tau - \tau_{max}$, $\sigma - \sigma_{max}$, $P_c - P$, $\delta - \delta_{max}$"
plt.text(0.5, 0.5, eqn, fontsize=16, ha='center')
plt.axis('off')
plt.show()
```

Figure.86. – Python Code for Depicting Welded Beam Formula

$$\tau - \tau_{max},\ \sigma - \sigma_{max},\ P_c - P,\ \delta - \delta_{max}$$

Formula.3. - Welded Beam Formula

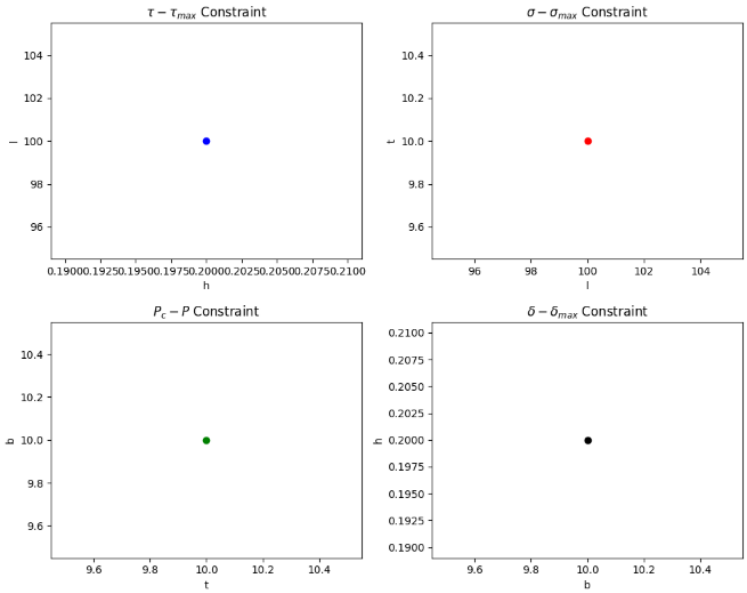


Figure.87. – Graphical Welded Beam Representation

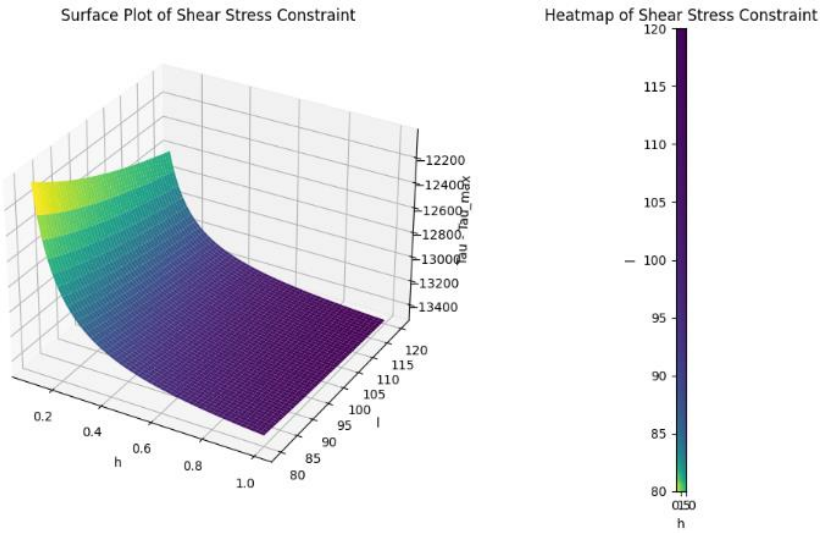


Figure.88. – Surf Plot of Welded Beam Representation

g06 Problem
Objective Function: $f(x) = (x_1 - 10)^3 + (x_2 - 20)^3$
Subject to constraints:
$c_1(x) : -(x_1 - 5)^2 - (x_2 - 5)^2 + 100 \leq 0$
$c_2(x) : (x_1 - 6)^2 + (x_2 - 5)^2 - 82.81 \leq 0$
$13 \leq x_1 \leq 100, 0 \leq x_2 \leq 100$

Formula.4. – g06 Objective Function Formula

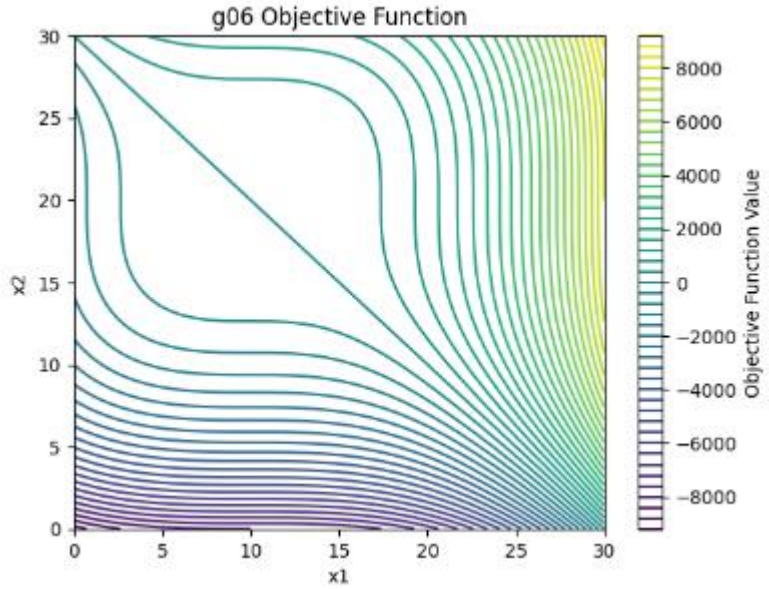


Figure.89. – g06 Objective Function Visual Depiction

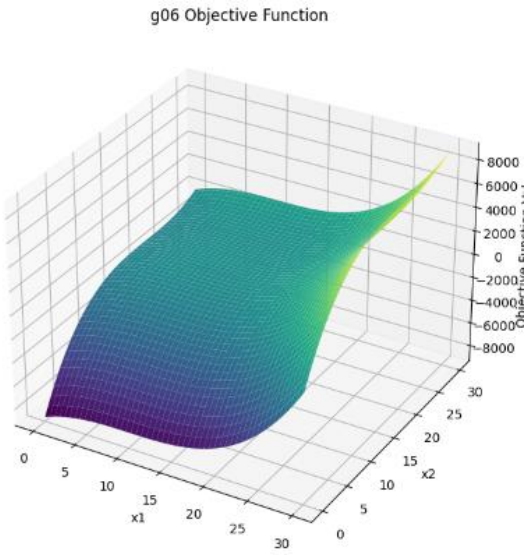


Figure.90. – g06 Objective Function Surf Plot

Constrained g08

g08 Problem

Objective Function: $f(x) = -\frac{\sin^3(2\pi x_1)\cdot\sin(2\pi x_2)}{x_1^3\cdot(x_1+x_2)}$

$0 \le x_1 \le 10, 0 \le x_2 \le 10$

Formula.5. – g08 Objective Function Formula

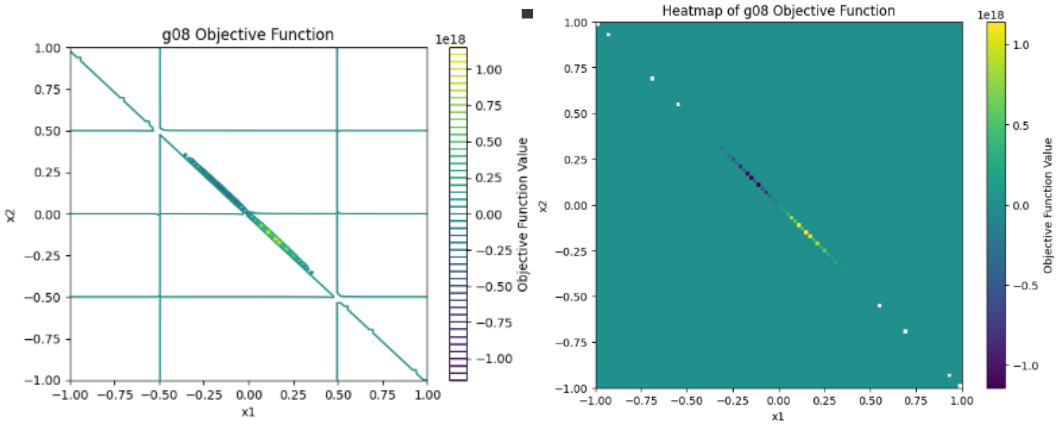


Figure.91. – g08 Objective Function Heatmap Visual Representation

References

[1]J. Yang and C. K. Soh, “Structural Optimization by Genetic Algorithms with Tournament Selection,” *Journal of Computing in Civil Engineering*, vol. 11, no. 3, pp. 195–200, Jul. 1997, doi: https://doi.org/10.1061/(asce)0887-3801(1997)11:3(195).

[2]R. Z. Farahani and M. Elahipanah, “A Genetic Algorithm to Optimize the Total Cost and Service Level for just-in-time Distribution in a Supply Chain,” *International Journal of Production Economics*, vol. 111, no. 2, pp. 229–243, Feb. 2008, doi: https://doi.org/10.1016/j.ijpe.2006.11.028.

[3]Y. Xue, H. Zhu, J. Liang, and A. Słowik, “Adaptive Crossover Operator Based multi-objective Binary Genetic Algorithm for Feature Selection in Classification,” *Knowledge-Based Systems*, vol. 227, no. 1, p. 107218, Sep. 2021, doi: https://doi.org/10.1016/j.knosys.2021.107218.

[4]D. F. Hougen and P. A. Diaz-Gomez, “Optimization of Parameters for Binary Genetic Algorithms,” ., vol. 1, no. 1, Jan. 2007.

[5]O. A. Abdul-Rahman, M. Munetomo, and K. Akama, “An Adaptive Parameter binary-real Coded Genetic Algorithm for Constraint Optimization problems: Performance Analysis and Estimation of Optimal Control Parameters,” *Information Sciences*, vol. 233, no. 1, pp. 54–86, Jun. 2013, doi: https://doi.org/10.1016/j.ins.2013.01.005.

[6]Minh Nghia Le, Yew Soon Ong, and Quang Huy Nguyen, “Optinformatics for Schema Analysis of Binary Genetic Algorithms,” ., vol. 1, no. 1, Jul. 2008, doi: https://doi.org/10.1145/1389095.1389308.

[7]A. K. Shukla, P. Singh, and M. Vardhan, “A New Hybrid Feature Subset Selection Framework Based on Binary Genetic Algorithm and Information Theory,” *International Journal of Computational Intelligence and Applications*, vol. 18, no. 03, p. 1950020, Sep. 2019, doi: https://doi.org/10.1142/s1469026819500202.

[8]H. Kuk-Hyun and K. Jong-Hwan, “Quantum-inspired Evolutionary Algorithm for a Class of Combinatorial Optimization,” *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 6, pp. 580–593, Dec. 2002, doi: https://doi.org/10.1109/tevc.2002.804320.

[9]G. Bharathi, “User’s Guide Genetic Algorithm TOOLBOX for Use with MATLAB □,” *www.academia.edu*, vol. 1, no. 1, Accessed: Feb. 19, 2024. [Online]. Available: https://www.academia.edu/4990605/Users_Guide_Genetic_Algorithm_TOOLBOX_For_Use_with_MATLAB_

[10]MathWorks, “How the Genetic Algorithm Works - MATLAB & Simulink,” *www.mathworks.com*, 2023. https://www.mathworks.com/help/gads/how-the-genetic-algorithm-works.html

[11]M. Kaya, “The Effects of a New Selection Operator on the Performance of a Genetic

Algorithm," *Applied Mathematics and Computation*, vol. 217, no. 19, pp. 7669–7678, Jun. 2011, doi: https://doi.org/10.1016/j.amc.2011.02.070.
[12]M. Kaya, "The Effects of Two New Crossover Operators on Genetic Algorithm Performance," *Applied Soft Computing*, vol. 11, no. 1, pp. 881–890, Jan. 2011, doi: https://doi.org/10.1016/j.asoc.2010.01.008.
[13]A. Sadegheih, "Scheduling Problem Using Genetic algorithm, Simulated Annealing and the Effects of Parameter Values on GA Performance," *Applied Mathematical Modelling*, vol. 30, no. 2, pp. 147–154, Feb. 2006, doi: https://doi.org/10.1016/j.apm.2005.03.017.
[14]D. E. Goldberg, *Genetic Algorithms in search, optimization, and Machine Learning*. Addison-Wesley, 1988.
[15]D. Heiss-Czedik, "An Introduction to Genetic Algorithms.," *Artificial Life*, vol. 3, no. 1, pp. 63–65, Jan. 1997, doi: https://doi.org/10.1162/artl.1997.3.1.63.
[16]C. R. Reeves, "Evolutionary computation: a Unified Approach," *Genetic Programming and Evolvable Machines*, vol. 8, no. 3, pp. 293–295, Jul. 2007, doi: https://doi.org/10.1007/s10710-007-9035-9.
[17]D. SIlva, F. Rador, and C. Moraes, "Particle Swarm Optimization and Quantum Particle Swarm Optimization to Multidimensional Function Approximation," *scholarly.org*, 2018. https://scholarly.org/pdf/display/particle-swarm-optimization-and-quantum-particle-swarm-optimization-to-multidimensional-function-approximation (accessed Feb. 23, 2024).
[18]M. N. Ab Wahab, S. Nefti-Meziani, and A. Atyabi, "A Comprehensive Review of Swarm Optimization Algorithms," *PLOS ONE*, vol. 10, no. 5, p. e0122827, May 2015, doi: https://doi.org/10.1371/journal.pone.0122827.
[19]A. Stacey, M. Jancic, and I. Grundy, "Particle Swarm Optimization with Mutation | IEEE Conference Publication | IEEE Xplore," *ieeexplore.ieee.org*, May 24, 2004. https://ieeexplore.ieee.org/document/1299838/authors#authors (accessed Feb. 23, 2024).
[20]A. Idowu, "Introduction to Global Optimization Algorithms for Continuous Domain Functions":, *Medium*, Mar. 12, 2021. https://towardsdatascience.com/introduction-to-global-optimization-algorithms-for-continuous-domain-functions-7ad9d01db055
[21]S. Sengupta, S. Basak, and R. Peters, "Particle Swarm Optimization: A Survey of Historical and Recent Developments with Hybridization Perspectives," *Machine Learning and Knowledge Extraction*, vol. 1, no. 1, pp. 157–191, Oct. 2018, doi: https://doi.org/10.3390/make1010010.
[22]A. Raß, M. Schmitt, and R. Wanka, "Explanation of Stagnation at Points That Are Not Local Optima in Particle Swarm Optimization by Potential Analysis," *arXiv.org*, Apr. 30, 2015. https://arxiv.org/abs/1504.08241 (accessed Feb. 29, 2024).
[23]Z.-G. Liu, X.-H. Ji, and Y.-X. Liu, "Hybrid non-parametric Particle Swarm Optimization

and Its Stability Analysis," *Expert Systems with Applications*, vol. 92, no. 1, pp. 256–275, Feb. 2018, doi: https://doi.org/10.1016/j.eswa.2017.09.012.
[24]D. Freitas, L. G. Lopes, and F. Morgado-Dias, "Particle Swarm Optimisation: a Historical Review up to the Current Developments," *Entropy*, vol. 22, no. 3, p. 362, Mar. 2020, doi: https://doi.org/10.3390/e22030362.
[25]A. G. Gad, "Particle Swarm Optimization Algorithm and Its Applications: a Systematic Review," *Archives of Computational Methods in Engineering*, vol. 29, no. 5, pp. 2531–2561, Apr. 2022, doi: https://doi.org/10.1007/s11831-021-09694-4.
[26]M. Jain, V. Saihjpal, N. Singh, and S. B. Singh, "An Overview of Variants and Advancements of PSO Algorithm," *Applied Sciences*, vol. 12, no. 17, p. 8392, Aug. 2022, doi: https://doi.org/10.3390/app12178392.
[27]D. G. Mayer, B. P. Kinghorn, and A. A. Archer, "Differential Evolution – an Easy and Efficient Evolutionary Algorithm for Model Optimisation," *Agricultural Systems*, vol. 83, no. 3, pp. 315–328, Mar. 2005, doi: https://doi.org/10.1016/j.agsy.2004.05.002.
[28]T. Eltaeib and A. Mahmood, "Differential Evolution: A Survey and Analysis," *Applied Sciences*, vol. 8, no. 10, p. 1945, Oct. 2018, doi: https://doi.org/10.3390/app8101945.
[29]M. Georgioudakis and V. Plevris, "A Comparative Study of Differential Evolution Variants in Constrained Structural Optimization," *Frontiers in Built Environment*, vol. 6, no. 1, Jul. 2020, doi: https://doi.org/10.3389/fbuil.2020.00102.
[30]J. Yao, Z. Chen, and Z. Liu, "Improved ensemble of differential evolution variants," *PLOS ONE*, vol. 16, no. 8, p. e0256206, Aug. 2021, doi: https://doi.org/10.1371/journal.pone.0256206.
[31]A. K. Qin, V. L. Huang, and P. N. Suganthan, "Differential Evolution Algorithm With Strategy Adaptation for Global Numerical Optimization," *IEEE Transactions on Evolutionary Computation*, vol. 13, no. 2, pp. 398–417, Apr. 2009, doi: https://doi.org/10.1109/tevc.2008.927706.
[32]S. Paterlini and T. Krink, "Differential evolution and particle swarm optimisation in partitional clustering," *Computational Statistics & Data Analysis*, vol. 50, no. 5, pp. 1220–1247, Mar. 2006, doi: https://doi.org/10.1016/j.csda.2004.12.004.
[33]G. C. Onwubolu, "Design of Hybrid Differential Evolution and Group Method of Data Handling Networks for Modeling and Prediction," *Information Sciences*, vol. 178, no. 18, pp. 3616–3634, Sep. 2008, doi: https://doi.org/10.1016/j.ins.2008.05.013.
[34]A. Draa, S. Bouzoubia, and I. Boukhalfa, "A sinusoidal differential evolution algorithm for numerical optimisation," *Applied Soft Computing*, vol. 27, no. 1, pp. 99–126, Feb. 2015, doi: https://doi.org/10.1016/j.asoc.2014.11.003.
[35]W.-Y. Lin, "A GA–DE Hybrid Evolutionary Algorithm for Path Synthesis of four-bar

Linkage," *Mechanism and Machine Theory*, vol. 45, no. 8, pp. 1096–1107, Aug. 2010, doi: https://doi.org/10.1016/j.mechmachtheory.2010.03.011.
[36]Mathworks, "The Implementation of Differential Evolution in Matlab," *uk.mathworks.com*, Mar. 07, 2024. https://uk.mathworks.com/matlabcentral/fileexchange/77617-the-implementation-of-differential-evolution-in-matlab (accessed Mar. 07, 2024).
[37]Bipul Luitel and G.K. Venayagamoorthy, "Differential evolution particle swarm optimization for digital filter design," *Missouri S&T*, vol. 1, no. 1, Jun. 2008, doi: https://doi.org/10.1109/cec.2008.4631335.
[38]M. F. Ahmad, N. A. M. Isa, W. H. Lim, and K. M. Ang, "Differential evolution: A recent review based on state-of-the-art works," *Alexandria Engineering Journal*, vol. 61, no. 5, pp. 3831–3872, May 2022, doi: https://doi.org/10.1016/j.aej.2021.09.013.
[39]J. Kennedy and R. Eberhart, "Particle swarm optimization," *Proceedings of ICNN'95 - International Conference on Neural Networks*, vol. 4, no. 1, pp. 1942–1948, 1995, doi: https://doi.org/10.1109/icnn.1995.488968.
[40]Z. Wang, J. Ma, and L. Zhang, "State-of-Health Estimation for Lithium-Ion Batteries Based on the Multi-Island Genetic Algorithm and the Gaussian Process Regression," *IEEE Access*, vol. 5, no. 1, pp. 21286–21295, 2017, doi: https://doi.org/10.1109/access.2017.2759094.
[41]Sergio Gerardo de-los-Cobos-Silva, Miguel Ángel Gutiérrez-Andrade, Roman Anselmo Mora-Gutiérrez, P. Lara-Velázquez, Eric Alfredo Rincón-García, and Antonin Ponsich, "An Efficient Algorithm for Unconstrained Optimization," *Mathematical problems in engineering*, vol. 2015, no. 1, pp. 1–17, Jan. 2015, doi: https://doi.org/10.1155/2015/178545.
[42]W. Cai, L. Yang, and Y. Yu, "Solution of ackley function based on particle swarm optimization algorithm," *IEEE Xplore*, Aug. 01, 2020. https://ieeexplore.ieee.org/document/9213634 (accessed Jul. 14, 2022).
[43]T. Bäck and H.-P. Schwefel, "An Overview of Evolutionary Algorithms for Parameter Optimization," *Evolutionary Computation*, vol. 1, no. 1, pp. 1–23, Mar. 1993, doi: https://doi.org/10.1162/evco.1993.1.1.1.
[44]G. Wu, X. Shen, H. Li, H. Chen, A. Lin, and Ponnuthurai Nagaratnam Suganthan, "Ensemble of differential evolution variants," ., vol. 423, no. 1, pp. 172–186, Jan. 2018, doi: https://doi.org/10.1016/j.ins.2017.09.053.
[45]E. Mezura-Montes, J. Velázquez-Reyes, and C. A. Coello Coello, "A comparative study of differential evolution variants for global optimization," *Proceedings of the 8th annual conference on Genetic and evolutionary computation - GECCO '06*, vol. 1, no. 1, 2006, doi: https://doi.org/10.1145/1143997.1144086.
[46]A. Homaifar, C. X. Qi, and S. H. Lai, "Constrained Optimization Via Genetic Algorithms,"

SIMULATION, vol. 62, no. 4, pp. 242–253, Apr. 1994, doi: https://doi.org/10.1177/003754979406200405.

[47]Z. Michalewicz and M. Schoenauer, “Evolutionary Algorithms for Constrained Parameter Optimization Problems,” *Evolutionary Computation*, vol. 4, no. 1, pp. 1–32, Mar. 1996, doi: https://doi.org/10.1162/evco.1996.4.1.1.

[48]R. I. Colautti and J. A. Lau, “Contemporary evolution during invasion: evidence for differentiation, natural selection, and local adaptation,” *Molecular Ecology*, vol. 24, no. 9, pp. 1999–2017, Apr. 2015, doi: https://doi.org/10.1111/mec.13162.

[49]M. Yang, C. Li, Z. Cai, and J. Guan, “Differential Evolution With Auto-Enhanced Population Diversity,” ., vol. 45, no. 2, pp. 302–315, Feb. 2015, doi: https://doi.org/10.1109/tcyb.2014.2339495.

Code

Unconstrained Ackley Function Code

Ackley & RGA Code

Ackley.m

```
function z = Ackley(x)
    a = 20;
    b = 0.2;
    c = 2*pi;
    d = numel(x);  % Number of dimensions
    sum1 = sum(x.^2);
    sum2 = sum(cos(c*x));
    z = -a * exp(-b * sqrt(sum1/d)) - exp(sum2/d) + a + exp(1);
end
```

AppAckley.m

```
% Problem Definition
problem.CostFunction = @(x) Ackley(x);  % Cost Function
problem.nVar = 2;  % Number of Decision Variables
problem.VarMin = -32.768;  % Lower Bound of Variables
problem.VarMax = 32.768;   % Upper Bound of Variables

% GA Parameters
params.MaxIt = 50;        % Maximum Number of Iterations
params.nPop = 25;          % Population Size
params.pC = 0.9;           % Crossover Percentage
params.pM = 0.6;           % Mutation Percentage
params.sigma = 0.3;        % Mutation Rate

% Array of parameter sets for different runs
paramSets = [
    struct('MaxIt', 1000, 'nPop', 50, 'pC', 0.9, 'pM', 0.6, 'sigma', 0.3);
    struct('MaxIt', 800, 'nPop', 60, 'pC', 0.8, 'pM', 0.5, 'sigma', 0.3);
    struct('MaxIt', 600, 'nPop', 70, 'pC', 0.7, 'pM', 0.4, 'sigma', 0.3);
    struct('MaxIt', 400, 'nPop', 80, 'pC', 0.6, 'pM', 0.3, 'sigma', 0.3);
    struct('MaxIt', 300, 'nPop', 150, 'pC', 0.5, 'pM', 0.2, 'sigma', 0.3);
    struct('MaxIt', 1000, 'nPop', 250, 'pC', 0.5, 'pM', 0.1, 'sigma', 0.1);
    struct('MaxIt', 800, 'nPop', 160, 'pC', 0.6, 'pM', 0.25, 'sigma', 0.15);
    struct('MaxIt', 600, 'nPop', 170, 'pC', 0.7, 'pM', 0.4, 'sigma', 0.5);
    struct('MaxIt', 400, 'nPop', 180, 'pC', 0.8, 'pM', 0.3, 'sigma', 0.4);
    struct('MaxIt', 300, 'nPop', 150, 'pC', 0.9, 'pM', 0.2, 'sigma', 0.4);
];

% Run the RGA with the parameter sets
tolerance = 1e-2; % Set your desired tolerance
out = RunRGA(problem, params, tolerance);

% Tolerance Check
globalMin = 0; % Min for Ackley function
tolerance = 1e-2; % User-defined tolerance

disp(class(out.bestCosts));
disp(size(out.bestCosts));

% Initialise an array to store the minimum values from each cell
```

```
minValues = zeros(size(out.bestCosts));

% Loop through each cell and find the minimum value
for i = 1:numel(out.bestCosts)
    % Access the numeric or logical data within each cell
    cellData = out.bestCosts{i};

    % Compute the minimum value within the cell
    minValues(i) = min(cellData(:));
end

% Find the overall minimum value across all cells
bestSolutionFound = min(minValues);

% Check the best solution against the global minimum after the GA completes
if abs(bestSolutionFound - globalMin) < tolerance
    disp('Success: Tolerance criteria met.');
else
    disp('The run did not meet the tolerance criteria.');
end

% Process the data
processedData = ProcessData(out.bestCosts);

SaveToExcel(processedData, paramSets, results);
PlotBestCostEvolution(results.bestCosts);
PlotFullEvolution(results.bestCosts);
PlotSurfPlot(results);
```

RunRGA.m

```
function results = RunRGA(problem, paramSets, tolerance)
    % Problem Definition
    CostFunction = problem.CostFunction;
    nVar = problem.nVar;
    VarMin = problem.VarMin;
    VarMax = problem.VarMax;

    % Initialise results structure
    results = struct();

    % Initialise empty arrays to store the best solution and best cost for each
run
    bestSolutions = cell(1, length(paramSets)); % Stores the best solution of each
run
    bestCosts = cell(1, length(paramSets)); % Stores the best cost of each run

    % Perform the runs for each parameter set
    for paramIndex = 1:length(paramSets)
        params = paramSets(paramIndex);

        % Parameters of GA for this run
        MaxIt = params.MaxIt;
        nPop = params.nPop;
        pC = params.pC;
        sigma = params.sigma;
        numIterations = params.MaxIt;
        pM = params.pM;
```

```
        % Initialise empty arrays to store the best solution and best cost for
this run
        bestSol = [];
        bestCost = inf;

        % Run the RGA for multiple iterations
        for iteration = 1:numIterations
            % Initialization
            empty_individual.Position = [];
            empty_individual.Cost = [];
            pop = repmat(empty_individual, nPop, 1);

            % Initialise Population
            for i = 1:nPop
                pop(i).Position = unifrnd(VarMin, VarMax, [1, nVar]);
                pop(i).Cost = CostFunction(pop(i).Position);
            end

            % Main loop of GA
            for it = 1:MaxIt
                % GA Operations
                for k = 1:2:nPop
                    % Selection
                    i1 = TournamentSelection(pop, 3);
                    i2 = TournamentSelection(pop, 3);

                    % Crossover
                    [pop(k).Position, pop(k+1).Position] =
UniformCrossover(pop(i1).Position, pop(i2).Position, pC);

                    % Mutation with Mutation Probability pM
                    if rand() < pM
                        pop(k).Position = GaussianMutation(pop(k).Position,
sigma);
                    end
                    if rand() < pM
                        pop(k+1).Position = GaussianMutation(pop(k+1).Position,
sigma);
                    end

                    % Apply Bounds
                    pop(k).Position = max(pop(k).Position, VarMin);
                    pop(k).Position = min(pop(k).Position, VarMax);
                    pop(k+1).Position = max(pop(k+1).Position, VarMin);
                    pop(k+1).Position = min(pop(k+1).Position, VarMax);

                    % Evaluate Offsprings
                    pop(k).Cost = CostFunction(pop(k).Position);
                    pop(k+1).Cost = CostFunction(pop(k+1).Position);

                    % Update Best Solution
                    if pop(k).Cost < bestCost
                        bestSol = pop(k);
                        bestCost = pop(k).Cost;
                    end
                    if pop(k+1).Cost < bestCost
                        bestSol = pop(k+1);
                        bestCost = pop(k+1).Cost;
                    end
```

```
                end

                % Print iteration information to the console
                disp(['Param Set: ' num2str(paramIndex) ', Iteration ' num2str(it)
': Best Cost = ' num2str(bestCost)]);

                % Check termination criteria
                if bestCost < tolerance
                    disp('Terminating optimization: Tolerance criteria met.');
                    break; % Exit the optimization loop if the threshold is met
                end
            end
        end

        % Store the best solution and best cost for this run
        bestSolutions{paramIndex} = bestSol;
        bestCosts{paramIndex} = bestCost;
    end

    % Store results
    results.bestSolutions = bestSolutions;
    results.bestCosts = bestCosts;
end
```

GaussianMutation.m

```
function y = GaussianMutation(x, sigma)
    nVar = numel(x);
    y = x + sigma * randn(size(x));
end
```

PlotBestCostEvolution.m

```
function PlotBestCostEvolution(bestCosts)
    % Plot best cost evolution
    figure;
    for i = 1:length(bestCosts)
        plot(bestCosts{i}, 'LineWidth', 1.5);
        hold on;
    end
    xlabel('Iteration');
    ylabel('Best Cost');
    title('Best Cost Evolution');
    legend('Run 1', 'Run 2', 'Run 3', 'Run 4', 'Run 5'); % Update legend as per
the number of runs
    hold off;
end
```

PlotFullEvolution.m

```
function PlotFullEvolution(bestCosts)
    % Plot combined evolution of all runs
    figure;
    allBestCosts = cell2mat(bestCosts); % Combine all best costs into a single
array
    plot(allBestCosts, 'LineWidth', 1.5);
    xlabel('Iteration');
    ylabel('Best Cost');
    title('Combined Fitness Landscape of All Runs');
```

```
end
```

PlotGAContour.m

```
function PlotGAContour(problem, out)
    % Extract lower and upper bounds from the problem definition
    VarMin = problem.VarMin; % The minimum value of your design variables
    VarMax = problem.VarMax; % The maximum value of your design variables

    % Generate grid points for the contour plot
    [X, Y] = meshgrid(linspace(VarMin, VarMax, 100), linspace(VarMin, VarMax,
100));

    % Calculate the cost for each grid point
    Z = arrayfun(@(x, y) problem.CostFunction([x y]), X, Y);

    % Generate the contour plot of the cost function landscape
    contour(X, Y, Z, 50); % Adjust the number 50 if need more or fewer contour
lines
    hold on;

    % Iterate over each cell in bestSolutions and plot the best solution for each
iteration
    for i = 1:numel(out.bestSolutions)
        bestSolution = out.bestSolutions{i}; % Access the cell corresponding to
the current iteration
        plot(bestSolution(1), bestSolution(2), 'ro', 'LineWidth', 2);
    end

    % Further customisation
    title('Contour Plot with GA Solution Progress');
    xlabel('Variable 1');
    ylabel('Variable 2');
    colorbar;
    hold off;
end
```

PlotSurfPlot.m

```
function PlotSurfPlot(bestSolutions)
    % Plot the surf plot
    if size(bestSolutions, 1) ~= 2
        error('Surf plot can only be generated for 2D problems.');
    end

    figure;
    % Generate meshgrid for surf plot
    [X, Y] = meshgrid(linspace(problem.VarMin, problem.VarMax, 100));
    Z = zeros(size(X));

    % Compute cost for each point in the meshgrid
    for i = 1:numel(X)
        Z(i) = problem.CostFunction([X(i), Y(i)]);
    end

    surf(X, Y, Z);
    xlabel('Variable 1');
    ylabel('Variable 2');
    zlabel('Cost');
```

```
    title('Surf Plot of Cost Function');
end
```

ProcessData.m

```
function processedData = ProcessData(bestCosts)
    % Get the number of iterations
    MaxIt = size(bestCosts, 1);

    % Create a cell array to hold the processed data
    processedData = cell(MaxIt + 1, 2);

    % Fill in the headers for the processed data
    processedData{1, 1} = 'Iteration';
    processedData{1, 2} = 'Best Cost';

    % Fill in the processed data
    for it = 1:MaxIt
        % Get the current row of best costs
        row = bestCosts(it, :);

        % Filter out non-numeric values
        rowNumeric = row(isnumeric(row));

        % Check if there are numeric values in the row
        if ~isempty(rowNumeric)
            % Find the minimum cost for this iteration
            minCost = min(rowNumeric);
        else
            % If there are no numeric values, set minCost to NaN
            minCost = NaN;
        end

        % Fill in the iteration number and best cost
        processedData{it + 1, 1} = it;
        processedData{it + 1, 2} = minCost;
    end
end
```

SaveToExcel.m

```
function SaveToExcel(processedData, paramSets, results)
    % Create a new Excel workbook
    spreadsheet = 'RGA_Data.xlsx';
    delete(spreadsheet); % Delete existing file if it exists
    xls = actxserver('Excel.Application');
    workbook = xls.Workbooks.Add();
    xls.Visible = 1;

    % Write data to the first sheet with headings
    headings = {'MaxIt', 'nPop', 'pC', 'pM', 'sigma', 'Best Cost'};
    worksheet = workbook.Sheets.Item(1);
    worksheet.Name = 'Summary';
    range = worksheet.Range('A1:F1');
    range.Value = headings;

    % Write data to the first sheet
    data = cell(length(paramSets), 6);
    for i = 1:length(paramSets)
```

```
        params = paramSets(i);
        data{i, 1} = params.MaxIt;
        data{i, 2} = params.nPop;
        data{i, 3} = params.pC;
        data{i, 4} = params.pM;
        data{i, 5} = params.sigma;
        data{i, 6} = results.bestCosts{i};
    end
    range = worksheet.Range('A2:F' + string(length(paramSets) + 1));
    range.Value = data;

    % Add a new sheet for each run and write the results
    for i = 1:length(paramSets)
        worksheet = workbook.Sheets.Add([],
workbook.Sheets.Item(workbook.Sheets.Count));
        worksheet.Name = 'Run ' + string(i);
        range = worksheet.Range('A1:A' + string(params.MaxIt));
        range.Value = results.bestCosts{i};
    end

    % Save the workbook
    workbook.SaveAs(spreadsheet);
    workbook.Close();
    xls.Quit();
end
```

TournamentSelection.m

```
function index = TournamentSelection(pop, tournamentSize)
    % Randomly select individuals for the tournament
    selectedIndices = randperm(length(pop), tournamentSize);
    [~, bestIndex] = min([pop(selectedIndices).Cost]);
    index = selectedIndices(bestIndex);
end
```

UniformCrossover.m

```
function [child1, child2] = UniformCrossover(parent1, parent2, crossoverRate)
    nVar = numel(parent1);
    child1 = parent1;
    child2 = parent2;
    for i = 1:nVar
        if rand() <= crossoverRate
            child1(i) = parent2(i);
            child2(i) = parent1(i);
        end
    end
end
```

Ackley & PSO Code

NB: animateParticles.m & saveHistoricalData.m are listed in the constrained method code section so will not be included here.

AppPSO.m

```
clc; clear;
```

```
%% Problem Definition
problem.CostFunction = @(x) Ackley(x);  % Problem function
problem.nVar = 2;
problem.VarMin = -5;  % Lower Bound
problem.VarMax = 5; % Upper Bound

%% Parameters of PSO

% Constriction Coefficients
kappa = 1;
phi1 = 2.05;
phi2 = 2.05;
phi = phi1 + phi2;
chi = 2*kappa/abs(2-phi-sqrt(phi^2-4*phi));

params.w = chi;             % Intertia Coefficient
params.wdamp = 1;           % Damping Ratio of Inertia Coefficient
params.c1 = chi*phi1;       % Personal Acceleration Coefficient
params.c2 = chi*phi2;       % Social Acceleration Coefficient
params.ShowIterInfo = true; % Flag for Showing Iteration Information

% Tolerance and global minimum
tolerance = 1e-2;
globalMin = [0, 0];  % Theoretical global minimum

% Array of parameter sets for different runs
paramSets = [
    struct('MaxIt', 1000, 'nPop', 50, 'w', 1.00, 'wdamp', 0.99, 'c1', 2.00, 'c2',
2.00, 'ShowIterInfo', true);
    struct('MaxIt', 800, 'nPop', 60, 'w', 0.99, 'wdamp', 0.75, 'c1', 1.00, 'c2',
1.00, 'ShowIterInfo', true);
    struct('MaxIt', 600, 'nPop', 70, 'w', 1.00, 'wdamp', 0.99, 'c1', 2.00, 'c2',
2.00, 'ShowIterInfo', true);
    struct('MaxIt', 400, 'nPop', 80, 'w', 1.00, 'wdamp', 0.99, 'c1', 2.00, 'c2',
2.00, 'ShowIterInfo', true);
    struct('MaxIt', 300, 'nPop', 150, 'w', 1.00, 'wdamp', 0.99, 'c1', 2.00, 'c2',
2.00, 'ShowIterInfo', true);
];

% Initialise arrays to store results
allBestSolutions = zeros(length(paramSets), problem.nVar);
allBestCosts = zeros(length(paramSets), 1);

% Loop through each parameter set
for i = 1:length(paramSets)
    params = paramSets(i);

    % PSO Main Body
    out = PSO(problem, params);  % Use the PSO function with the current parameter
set

    % Store Results
    allBestSolutions(i, :) = out.BestSol.Position;
    allBestCosts(i) = out.BestSol.Cost;
    BestSol = out.BestSol;
    BestCosts = out.BestCosts;
    particle_history = out.particle_history;
    particleHistory = out.particleHistory;
```

```
    % Tolerance Check
    error = max(abs(out.BestSol.Position - globalMin));
    if error < tolerance
        fprintf('Run %d: Solution found within the acceptable tolerance.\n', i);
    else
        fprintf('Run %d: Best Cost = %.4f, not within the acceptable tolerance
after %d iterations.\n', i, allBestCosts(i), params.MaxIt);
    end

    % Save Results to Spreadsheet for each parameter set
    saveHistoricalData(out.particleHistory, sprintf('historicalData_Run%d.xlsx',
i));

    % Visualisation for each parameter set
    figure; % Create a new figure for each set of results
    plotBestCostOverIterations(out.BestCosts);
    title(sprintf('Best Cost Over Iterations (Run %d)', i));
    drawnow; % Ensure plots are updated before the next iteration

    figure; % Create a new figure for contour plot
    plotPopulationContour(problem.CostFunction, out.pop, problem.VarMin,
problem.VarMax, params.nPop);
    title(sprintf('Population Contour (Run %d)', i));
    drawnow; % Ensure plots are updated before the next iteration

    % Optional: animateParticles function can be called here if needed
    animateParticles(out.particleHistory, out.BestCosts, params.MaxIt,
params.nPop);
end

% Compile all results into a table after all runs are complete
allEntries = cell(length(paramSets), 9); % 9 for each of the parameters and
results

for i = 1:length(paramSets)
    allEntries(i,:) = {paramSets(i).MaxIt, paramSets(i).nPop, paramSets(i).w, ...
        paramSets(i).wdamp, paramSets(i).c1, paramSets(i).c2, ...
        allBestSolutions(i, 1), allBestSolutions(i, 2), allBestCosts(i)};
end

resultsTable = cell2table(allEntries, 'VariableNames', ...
    {'MaxIt', 'nPop', 'w', 'wdamp', 'c1', 'c2', 'Var1', 'Var2', 'BestCost'});

% Write the compiled results to the spreadsheet
writetable(resultsTable, 'PSO_Results.xlsx');

% Plot comparison of Best Costs for all parameter sets
figure; % Create a new figure for comparison plot
PlotBestCostsComparison(allBestCosts, paramSets);
title('Comparison of Best Costs for Different Parameter Sets');

%% Visualisations
% Call the visualisation functions
plotBestCostOverIterations(BestCost);
plotPopulationContour(problem.CostFunction, out.pop, problem.VarMin,
problem.VarMax, params.nPop);
animateParticles(particleHistory, BestCosts, params.MaxIt, params.nPop);
PlotBestCostsComparison(allBestCosts, paramSets);
```

PSO.m

```
function out = PSO(problem, params)
    %% Problem Definiton
    CostFunction = problem.CostFunction;  % Cost Function
    nVar = problem.nVar;        % Number of Unknown (Decision) Variables
    VarSize = [1 nVar];         % Matrix Size of Decision Variables
    VarMin = problem.VarMin;     % Lower Bound of Decision Variables
    VarMax = problem.VarMax;    % Upper Bound of Decision Variables


    %% Parameters of PSO
    MaxIt = params.MaxIt;   % Maximum Number of Iterations
    nPop = params.nPop;     % Population Size (Swarm Size)
    w = params.w;           % Intertia Coefficient
    wdamp = params.wdamp;   % Damping Ratio of Inertia Coefficient
    c1 = params.c1;         % Personal Acceleration Coefficient
    c2 = params.c2;         % Social Acceleration Coefficient

    % The Flag for Showing Iteration Information
    ShowIterInfo = params.ShowIterInfo;

    MaxVelocity = 0.2*(VarMax-VarMin);
    MinVelocity = -MaxVelocity;

    particle_history = zeros(MaxIt, nPop, nVar);
    % Initialise a matrix to store particle history (needed for animation)
    particleHistory = zeros(nPop, nVar, MaxIt);

    %% Initialisation
    % The Particle Template
    empty_particle.Position = [];
    empty_particle.Velocity = [];
    empty_particle.Cost = [];
    empty_particle.Best.Position = [];
    empty_particle.Best.Cost = [];

    % Create Population Array
    particle = repmat(empty_particle, nPop, 1);

    % Initialise Global Best
    GlobalBest.Cost = inf;

    % Initialise Population Members
    for i=1:nPop

        % Generate Random Solution
        particle(i).Position = unifrnd(VarMin, VarMax, VarSize);

        % Initialise Velocity
        particle(i).Velocity = zeros(VarSize);

        % Evaluation
        particle(i).Cost = CostFunction(particle(i).Position);

        % Update the Personal Best
        particle(i).Best.Position = particle(i).Position;
        particle(i).Best.Cost = particle(i).Cost;
```

```
        % Update Global Best
        if particle(i).Best.Cost < GlobalBest.Cost
            GlobalBest = particle(i).Best;
        end

    end

    % Array to Hold Best Cost Value on Each Iteration
    BestCosts = zeros(MaxIt, 1);

    %% Main Loop of PSO

    for it=1:MaxIt

        for i=1:nPop

            % Update Velocity
            particle(i).Velocity = w*particle(i).Velocity ...
                + c1*rand(VarSize).*(particle(i).Best.Position -
particle(i).Position) ...
                + c2*rand(VarSize).*(GlobalBest.Position - particle(i).Position);

            % Apply Velocity Limits
            particle(i).Velocity = max(particle(i).Velocity, MinVelocity);
            particle(i).Velocity = min(particle(i).Velocity, MaxVelocity);

            % Update Position
            particle(i).Position = particle(i).Position + particle(i).Velocity;

            % Apply Lower and Upper Bound Limits
            particle(i).Position = max(particle(i).Position, VarMin);
            particle(i).Position = min(particle(i).Position, VarMax);

            % Evaluation
            particle(i).Cost = CostFunction(particle(i).Position);

            % Store the history
            particle_history(it, i, :) = particle(i).Position;

            % Update Personal Best
            if particle(i).Cost < particle(i).Best.Cost

                particle(i).Best.Position = particle(i).Position;
                particle(i).Best.Cost = particle(i).Cost;

                % Update Global Best
                if particle(i).Best.Cost < GlobalBest.Cost
                    GlobalBest = particle(i).Best;
                end

            end

        end

        % Store positions for animation
        for i = 1:nPop
            particleHistory(i, :, it) = particle(i).Position;
        end
```

```
        % Store the Best Cost Value
        BestCosts(it) = GlobalBest.Cost;

            %% Enhanced Display Iteration Information
            if ShowIterInfo
                disp(['Iteration ' num2str(it) ': Best Cost = '
num2str(BestCosts(it))]);
                fprintf('Current Best Position: [%f, %f]\n',
GlobalBest.Position); % Prints the current best position
            end

        % Damping Inertia Coefficient
        w = w * wdamp;

    end

    out.pop = particle;
    out.BestSol = GlobalBest;
    out.BestCosts = BestCosts;
    out.particle_history = particle_history;
    out.particleHistory = particleHistory;
    out.particle_final_positions = [particle.Position];

end
```

PlotAckleySurface.m

```
function PlotAckleySurface()
% PlotAckleySurface generates a surface plot of the Ackley function.

% Define the domain for the plot
x = linspace(-5, 5, 400);
y = linspace(-5, 5, 400);

[X, Y] = meshgrid(x, y);

% Preallocate Z for speed
Z = zeros(size(X));

% Evaluate the Ackley function at each point (x,y)
for i = 1:size(X, 1)
    for j = 1:size(X, 2)
        Z(i, j) = Ackley([X(i, j), Y(i, j)]);
    end
end

% Generate the surface plot
figure;
surf(X, Y, Z, 'EdgeColor', 'none');
title('Surface Plot of the Ackley Function');
xlabel('x');
ylabel('y');
zlabel('Ackley(x, y)');
colorbar; % Adds a color bar to indicate the scale
view(-45, 45); % Adjusts the view angle for better visualisation
end
```

plotBestCostComparison.m

```
function PlotBestCostsComparison(allBestCosts, paramSets)
    figure;
    hold on;
    for i = 1:size(allBestCosts, 2) % each column represents a run's best costs
over iterations
        plot(allBestCosts(:, i), 'LineWidth', 2, 'DisplayName', sprintf('Pop Size
= %d', paramSets(i).nPop));
    end
    hold off;
    title('Best Costs Over Iterations for Different Population Sizes');
    xlabel('Iteration');
    ylabel('Best Cost');
    legend show;
    grid on;
end
```

PlotBestCostOverIterations.m

```
function plotBestCostOverIterations(BestCosts)
    figure;
    semilogy(BestCosts, 'LineWidth', 2);
    xlabel('Iteration');
    ylabel('Best Cost');
    title('Best Cost Over Iterations');
    grid on;
end
```

plotPopulationContour.m

```
function plotPopulationContour(CostFunction, particle, VarMin, VarMax, nPop)
    % Prepare Mesh for Contour Plots
    x1 = linspace(VarMin, VarMax, 100); % Adjust the granularity as needed
    x2 = linspace(VarMin, VarMax, 100);
    [X1, X2] = meshgrid(x1, x2);

    % Evaluate the Cost Function on the grid
    Z = arrayfun(@(x, y) CostFunction([x, y]), X1, X2);

    % Clear and hold the figure
    clf; % Clears current figure window and resets hold state
    hold on;

    % Contour Plot of Cost Function
    contourf(X1, X2, log(Z+1), 20); % Filled contour plot with more levels
    colormap(jet); % Change to a more colorful colormap
    colorbar; % Optionally add a color bar to indicate the scale

    % Plot whole Population as red stars
    for i = 1:nPop
        plot(particle(i).Position(1), particle(i).Position(2), 'r*', 'LineWidth',
2);
    end

    % Enhancements
    title('Particle Positions and Cost Function Contour');
    xlabel('Variable x1');
    ylabel('Variable x2');
```

```
    axis equal;
    grid on; % Optionally add a grid
    hold off;
end
```

Ackley & DE Code

NB: As the same code other than the de.m file is used across the constrained and unconstrained DE landscape, I will include just the two DE files. Please see Constrained Himmelblau PSO Code Section.

de.m (DE/1)

```
%% DE /best/1/bin with Dynamic Parameter Testing incorporating a halt 10
iterations after achieving the global minimum within specified tolerance
clc; clear;

%% Problem Definition
CostFunction = @(x) Ackley(x);  % Cost Function
nVar = 2;                       % Number of Decision Variables
VarSize = [1 nVar];             % Decision Variables Matrix Size
VarMin = -5.12;                 % Lower Bound of Decision Variables
VarMax = 5.12;                  % Upper Bound of Decision Variables
tolerance = 1e-5;               % Tolerance for considering the global minimum
achieved

% DE parameters
F = 0.5; % value for scaling factor F
params_F = F;
immigrantFraction = 0.1;        % adjust as needed
prevBestCost = inf;             % Initialise with a high value assuming
minimisation
restartFraction = 0.2;          % Restart 20% of the population

% Initialisation
BestSol = struct('Cost', inf, 'Position', [], 'Index', 0);

%% Parameter Sets as Struct Array
parameterSets = [
    struct('MaxIt', 1000, 'nPop', 50, 'beta_min', 0.2, 'beta_max', 0.8, 'pCR',
0.2, 'immigrantFraction', 0.1, 'rateIncrease', 0.01, 'maxBetaMin', 0.5,
'minBetaMax', 0.7);  % Set 1
    struct('MaxIt', 500, 'nPop', 100, 'beta_min', 0.2, 'beta_max', 0.8, 'pCR',
0.2, 'immigrantFraction', 0.05, 'rateIncrease', 0.05, 'maxBetaMin', 0.5,
'minBetaMax', 0.7); % Set 2
    struct('MaxIt', 1000, 'nPop', 50, 'beta_min', 0.1, 'beta_max', 0.9, 'pCR',
0.3, 'immigrantFraction', 0.2, 'rateIncrease', 0.09, 'maxBetaMin', 0.5,
'minBetaMax', 0.7); % Set 3
    struct('MaxIt', 500, 'nPop', 50, 'beta_min', 0.2, 'beta_max', 0.8, 'pCR', 0.4,
'immigrantFraction', 0.15, 'rateIncrease', 0.1, 'maxBetaMin', 0.5, 'minBetaMax',
0.7);  % Set 4
    struct('MaxIt', 1000, 'nPop', 100, 'beta_min', 0.3, 'beta_max', 0.7, 'pCR',
0.2, 'immigrantFraction', 0.25, 'rateIncrease', 0.15, 'maxBetaMin', 0.5,
'minBetaMax', 0.7);% Set 5
    struct('MaxIt', 1500, 'nPop', 50, 'beta_min', 0.2, 'beta_max', 0.6, 'pCR',
0.5, 'immigrantFraction', 0.4, 'rateIncrease', 0.2, 'maxBetaMin', 0.5,
'minBetaMax', 0.7)   % Set 6
];
```

```
%% Initialise Results Storage
allBestCosts = zeros(max([parameterSets.MaxIt]), length(parameterSets) + 1); % +1
for iteration numbers
allBestCosts(:, 1) = 1:max([parameterSets.MaxIt]); % Column 1 is iteration numbers
globalMinIter = zeros(1, length(parameterSets)); % Initialise globalMinIter
%maxIterations = max([parameterSets.MaxIt]);
particlePositions = cell(length(parameterSets), 1);

%% Snapshot Variable Initialisations
% Determine the maximum number of iterations from all parameter sets
maxIterations = max([parameterSets.MaxIt]);

% Specify the iteration points for taking snapshots dynamically
snapshotIterations = [100, 350, maxIterations];

% Initialise a structure to hold the snapshots for each parameter set
snapshots = repmat(struct('Iteration', num2cell(snapshotIterations), 'Population',
[]), length(parameterSets), 1);

%% Initialise Summary table before the loop starts
Summary = table();

%% Main Loop Over Parameter Sets
for run = 1:length(parameterSets)
    params = parameterSets(run);
    adaptive_beta_min = params.beta_min;
    adaptive_beta_max = params.beta_max;
    previousBestCost = inf; % For tracking improvement
    particlePositions{run} = zeros(parameterSets(run).nPop, nVar,
parameterSets(run).MaxIt);
    CR = params.pCR; % Define the crossover rate for the current run

    % Initialise Population
    pop = struct('Position', [], 'Cost', inf(params.nPop, 1));
    for i = 1:params.nPop
        pop(i).Position = unifrnd(VarMin, VarMax, VarSize);
        pop(i).Cost = CostFunction(pop(i).Position);
    end
    [BestCost, index] = min([pop.Cost]);
    BestSol = pop(index);

    %% DE Main Loop
    for it = 1:params.MaxIt
        for i = 1:params.nPop
            pop(i).Cost = CostFunction(pop(i).Position);
            % Compare individual cost to the best cost found so far
            if pop(i).Cost < BestSol.Cost
                BestSol.Cost = pop(i).Cost;
                BestSol.Position = pop(i).Position;
                BestSol.Index = i;  % Store the index of the best solution
            end

            % Mutation
                    A = randperm(params.nPop, 3);
                    a = A(1);
                    b = A(2);
                    c = A(3);
```

```
                    y = pop(a).Position + params.beta_min*(pop(b).Position -
pop(c).Position);
                    y = max(y, VarMin);
                    y = min(y, VarMax);

            % Crossover (binomial)
            z = pop(i).Position;
            jRand = randi([1 numel(z)]);
            for j = 1:numel(z)
                if rand <= params.pCR || j == jRand
                    z(j) = y(j);
                end
            end

            % Selection
            NewSol.Position = z;
            NewSol.Cost = CostFunction(NewSol.Position);
            if NewSol.Cost < pop(i).Cost
                pop(i) = NewSol;
                if NewSol.Cost < BestSol.Cost
                    BestSol = NewSol;
                end
            end
        end

        % Introduce Random Immigrants
        pop = introduceRandomImmigrants(pop, CostFunction, VarMin, VarMax,
VarSize, params);

        % Update BestSol and BestCost logic...
        [currentBestCost, index] = min([pop.Cost]);  % Find the current best cost
and index
        if currentBestCost < BestSol.Cost  % If the current best is better than
the overall best
            BestSol = pop(index);  % Update BestSol
            BestCost = currentBestCost;  % Update BestCost with the current
iteration best
        end

        % Use the updated BestCost for adaptMutationRates
        [adaptive_beta_min, adaptive_beta_max] =
adaptMutationRates(adaptive_beta_min, adaptive_beta_max, params, BestCost,
previousBestCost);

             pop = restartPopulation(pop, VarMin, VarMax, VarSize,
restartFraction, CostFunction);

             % Apply local search if required
        if mod(it, 50) == 0
            pop = applyLocalSearch(pop, @localSearchFunction, CostFunction,
VarMin, VarMax);
        end

        improvementRate = abs(prevBestCost - BestCost) / prevBestCost;
        if improvementRate < 0.01
            pop = applyLocalSearch(pop, @localSearchFunction, CostFunction,
VarMin, VarMax);
        end
```

```
        % Update Best Cost Record
        allBestCosts(it, run + 1) = BestCost;

        % Store positions for this iteration
        for i = 1:parameterSets(run).nPop
            particlePositions{run}(i, :, it) = pop(i).Position;
        end

        if abs(BestSol.Cost) <= tolerance
            globalMinReached = true;
            globalMinIter(run) = it;
            if postGlobalMinIterations > 10
                break;
            end
        end

        % Update prevBestCost for the next iteration
        prevBestCost = BestCost;

        % After finding the best solution, update the BestSol structure
        [BestCost, BestIndex] = min([pop.Cost]);
        BestSol.Cost = BestCost;
        BestSol.Position = pop(BestIndex).Position;
        BestSol.Index = BestIndex; % Store the index of the best solution

        disp(['Iteration ' num2str(it) ' in Run ' num2str(run) ': Best Cost = '
num2str(BestCost)]);

        % Take snapshots at specified iterations
        if ismember(it, snapshotIterations)
            % Store population positions for the snapshot
            popPositions = reshape([pop.Position], nVar, [])';
            snapshots(run).Population = popPositions;  % Store snapshot for the
current parameter set 'run'
            disp(['Snapshot taken at Iteration ' num2str(it) ' for Parameter Set '
num2str(run)]);
            disp(['Population size: ' num2str(size(popPositions))]);  % Print out
population size for debugging
        end
    end

    % At the end of each run, create a new row in the summary table
    newSummary = table(F, CR, params.MaxIt, params.nPop, BestSol.Position(1),
BestSol.Position(2), BestSol.Cost, ...
        'VariableNames', {'F', 'CR', 'MaxIt', 'nPop', 'Best_X1', 'Best_X2',
'Best_Cost'});

    Summary = [Summary; newSummary];  % Append the new row to the summary table
end

%% Save summary Data:
disp(Summary);

% Save the table to a CSV file
writetable(Summary, 'DE_Summary.csv');

%% Save Iteration Data
csvwrite('DE_AllRunsBestCosts.csv', allBestCosts);
```

```
%% Save Medians in Table:
% Display a table of results
disp('Final Solution:');
disp(['Position = [' num2str(BestSol.Position) ']']);
disp(['Cost = ' num2str(BestSol.Cost)]);

%% Visualisation: Trajectory of Each Run's Journey up to Global Minimum
% New figure for the line graph
figure;
hold on;
colors = lines(length(parameterSets));

% Variable to store the minimum iteration for each parameter set
minIterations = zeros(length(parameterSets), 1);

for run = 1:length(parameterSets)
    params = parameterSets(run);
    itData = allBestCosts(:, run + 1);
    minIt = find(itData == min(itData), 1, 'first');
    minIterations(run) = minIt; % Store the minimum iteration
    plot(1:minIt, itData(1:minIt), 'LineWidth', 2, 'Color', colors(run, :));
end

legend(arrayfun(@(x) ['Set ' num2str(x)], 1:length(parameterSets),
'UniformOutput', false), 'Location', 'best');
xlabel('Iteration');
ylabel('Best Cost');
title('Best Cost Trajectory for Each Run');

hold off;

%% Create a 3D Surface Plot for Best Costs Over Iterations and Parameter Sets
% Determine the maximum number of iterations to display based on when global
minima were reached
maxDisplayIterations = max(globalMinIter);

% If no global minima were reached, use the maximum number of iterations for any
run
if maxDisplayIterations == 0
    maxDisplayIterations = maxIterations;
end

% Initialise the TruncatedBestCostsGrid with NaN values
TruncatedBestCostsGrid = NaN(length(parameterSets), maxDisplayIterations);

% Loop through each parameter set to fill in the best costs
for run = 1:length(parameterSets)
    if globalMinIter(run) > 0
        % Use only up to the iteration when the global minimum was reached
        TruncatedBestCostsGrid(run, 1:globalMinIter(run)) =
allBestCosts(1:globalMinIter(run), run + 1)';
    else
        % Use all iterations if the global minimum was not reached
        TruncatedBestCostsGrid(run, :) = allBestCosts(:, run + 1)';
    end
end

% Truncate the IterationGrid and ParameterSetGrid for display using the
maxDisplayIterations
```

```
[TruncatedIterationGrid, TruncatedParameterSetGrid] =
meshgrid(1:maxDisplayIterations, 1:length(parameterSets));

% Create the truncated surface plot
figure;
surf(TruncatedIterationGrid, TruncatedParameterSetGrid, TruncatedBestCostsGrid,
'EdgeColor', 'none');
xlabel('Iteration');
ylabel('Parameter Set');
zlabel('Best Cost');
title('Best Cost Surface Up to Global Minimum');
colorbar;
view(-45, 45);
colormap(jet);
legend(arrayfun(@(x) ['Set ' num2str(x) ' (global min)'], 1:length(parameterSets),
'UniformOutput', false), 'Location', 'best');

%% Contour Plot of Particle Journeys for a Selected Parameter Set
% Find the best solution from the final population of each run
bestPositions = zeros(length(parameterSets), nVar);
for run = 1:length(parameterSets)
    params = parameterSets(run);  % Use params specific for the current run
    finalPopCosts = arrayfun(@(i) CostFunction(particlePositions{run}(i, :, end)),
1:params.nPop);
    [~, idx] = min(finalPopCosts);  % Find the index of the best solution in the
final population
    bestPositions(run, :) = particlePositions{run}(idx, :, end);
end

% Create a meshgrid for the contour plot of the Rastrigin function
x = linspace(VarMin, VarMax, 100);
y = linspace(VarMin, VarMax, 100);
[X, Y] = meshgrid(x, y);
Z = arrayfun(@(x, y) CostFunction([x y]), X, Y);

% Plot the contour of the objective function
figure;
% Reduce the number of contour lines to make the plot less busy
contourf(X, Y, Z, 20); % Adjust the number to get the desired granularity
hold on;
colormap('parula'); % Set a colormap that provides a clear background

% Plot the best solutions as red balls
% Adjust the MarkerSize if needed
for i = 1:size(bestPositions, 1)
    plot(bestPositions(i, 1), bestPositions(i, 2), 'ro', 'MarkerSize', 10, ...
        'MarkerFaceColor', 'r', 'MarkerEdgeColor', 'w'); % White edge for better
visibility
end

% Label the plot
xlabel('x_1');
ylabel('x_2');
title('Best Particle Positions Contour');
hold off;

%% Plot side-by-side x-y plots of the snapshots for comparison
numRuns = length(snapshots);
```

```
numIterations = size(snapshots(1).Population, 3); % snapshots have the same number
of iterations

for j = 1:numIterations
    figure; % Create a new figure for each iteration
    for i = 1:numRuns
        subplot(1, numRuns, i);
        scatter(snapshots(i).Population(:, 1, j), snapshots(i).Population(:, 2,
j));
        title(sprintf('Run %d', i));
        xlabel('x1');
        ylabel('x2');
        axis([VarMin VarMax VarMin VarMax]);
        grid on;
    end
    sgtitle(sprintf('Population Snapshots at Iteration %d', j));
end
```

de.m (DE/2)

```
%% DE /best/2/bin with Dynamic Parameter Testing incorporating a halt 10
iterations after achieving the global minimum within specified tolerance
clc; clear;

%% Problem Definition
CostFunction = @(x) Himmelblau_Constrained(x);  % Cost Function
nVar = 2;                           % Number of Decision Variables
VarSize = [1 nVar];                 % Decision Variables Matrix Size
VarMin = -5.12;                     % Lower Bound of Decision Variables
VarMax = 5.12;                      % Upper Bound of Decision Variables
tolerance = 1e-5;                   % Tolerance for considering the global minimum
achieved

% DE parameters
F = 0.5; % value for scaling factor F
params_F = F;
immigrantFraction = 0.1;            % adjust as needed
prevBestCost = inf;                 % Initialise with a high value assuming
minimisation
restartFraction = 0.2;              % Restart 20% of the population

% Initialisation
BestSol = struct('Cost', inf, 'Position', [], 'Index', 0);

%% Parameter Sets as Struct Array
parameterSets = [
    struct('MaxIt', 1000, 'nPop', 50, 'beta_min', 0.2, 'beta_max', 0.8, 'pCR',
0.2, 'immigrantFraction', 0.1, 'rateIncrease', 0.01, 'maxBetaMin', 0.5,
'minBetaMax', 0.7);  % Set 1
    struct('MaxIt', 500, 'nPop', 100, 'beta_min', 0.2, 'beta_max', 0.8, 'pCR',
0.2, 'immigrantFraction', 0.05, 'rateIncrease', 0.05, 'maxBetaMin', 0.5,
'minBetaMax', 0.7); % Set 2
    struct('MaxIt', 1000, 'nPop', 50, 'beta_min', 0.1, 'beta_max', 0.9, 'pCR',
0.3, 'immigrantFraction', 0.2, 'rateIncrease', 0.09, 'maxBetaMin', 0.5,
'minBetaMax', 0.7); % Set 3
    struct('MaxIt', 500, 'nPop', 50, 'beta_min', 0.2, 'beta_max', 0.8, 'pCR', 0.4,
'immigrantFraction', 0.15, 'rateIncrease', 0.1, 'maxBetaMin', 0.5, 'minBetaMax',
0.7);  % Set 4
```

```
    struct('MaxIt', 1000, 'nPop', 100, 'beta_min', 0.3, 'beta_max', 0.7, 'pCR',
0.2, 'immigrantFraction', 0.25, 'rateIncrease', 0.15, 'maxBetaMin', 0.5,
'minBetaMax', 0.7);% Set 5
    struct('MaxIt', 1500, 'nPop', 50, 'beta_min', 0.2, 'beta_max', 0.6, 'pCR',
0.5, 'immigrantFraction', 0.4, 'rateIncrease', 0.2, 'maxBetaMin', 0.5,
'minBetaMax', 0.7)   % Set 6
];

%% Initialise Results Storage
allBestCosts = zeros(max([parameterSets.MaxIt]), length(parameterSets) + 1); % +1
for iteration numbers
allBestCosts(:, 1) = 1:max([parameterSets.MaxIt]); % Column 1 is iteration numbers
globalMinIter = zeros(1, length(parameterSets)); % Initialise globalMinIter
%maxIterations = max([parameterSets.MaxIt]);
particlePositions = cell(length(parameterSets), 1);

%% Snapshot Variable Initialisations
% Determine the maximum number of iterations from all parameter sets
maxIterations = max([parameterSets.MaxIt]);

% Set the iteration points at which you want to take snapshots
% Specify the iteration points for taking snapshots dynamically
snapshotIterations = [100, 350, maxIterations];  % Adjust as needed
%snapshotIterations = [round(maxIterations * 0.25), round(maxIterations * 0.5),
round(maxIterations * 0.75)];

% Initialise a structure to hold the snapshots for each parameter set
snapshots = repmat(struct('Iteration', num2cell(snapshotIterations), 'Population',
[]), length(parameterSets), 1);

%% Initialise Summary table before the loop starts
Summary = table();

%% Main Loop Over Parameter Sets
for run = 1:length(parameterSets)
    params = parameterSets(run);
    adaptive_beta_min = params.beta_min;
    adaptive_beta_max = params.beta_max;
    previousBestCost = inf; % For tracking improvement
    particlePositions{run} = zeros(parameterSets(run).nPop, nVar,
parameterSets(run).MaxIt);
    CR = params.pCR; % Define the crossover rate for the current run

    % Initialise Population
    pop = struct('Position', [], 'Cost', inf(params.nPop, 1));
    for i = 1:params.nPop
        pop(i).Position = unifrnd(VarMin, VarMax, VarSize);
        pop(i).Cost = CostFunction(pop(i).Position);
    end
    [BestCost, index] = min([pop.Cost]);
    BestSol = pop(index);

    %% DE Main Loop
    for it = 1:params.MaxIt
        for i = 1:params.nPop
            pop(i).Cost = CostFunction(pop(i).Position);
            % Compare individual cost to the best cost found so far
            if pop(i).Cost < BestSol.Cost
                BestSol.Cost = pop(i).Cost;
```

```
                BestSol.Position = pop(i).Position;
                BestSol.Index = i;  % Store the index of the best solution
            end
            % Choose two distinct random vector indices from the population (not
including the best individual)
            % Ensure unique indices for a, b, c, and d that are not equal to
BestSol.Cost
            indices = randperm(params.nPop);
            indices(indices == BestSol.Cost) = []; % Exclude BestSol.Cost
            a = indices(1);
            b = indices(2);
            c = indices(3);
            d = indices(4);

            % Mutation using best individual and two difference vectors
            y = BestSol.Position + params_F * (pop(a).Position - pop(b).Position)
...
                + params_F * (pop(c).Position - pop(d).Position);
            y = max(y, VarMin);
            y = min(y, VarMax);

            % Crossover (binomial)
            z = pop(i).Position;
            jRand = randi([1 numel(z)]);
            for j = 1:numel(z)
                if rand <= params.pCR || j == jRand
                    z(j) = y(j);
                end
            end

            % Selection
            NewSol.Position = z;
            NewSol.Cost = CostFunction(NewSol.Position);
            if NewSol.Cost < pop(i).Cost
                pop(i) = NewSol;
                if NewSol.Cost < BestSol.Cost
                    BestSol = NewSol;
                end
            end
        end

        % Introduce Random Immigrants
        pop = introduceRandomImmigrants(pop, CostFunction, VarMin, VarMax,
VarSize, params);

        % Update BestSol and BestCost logic...
        [currentBestCost, index] = min([pop.Cost]);  % Find the current best cost
and index
        if currentBestCost < BestSol.Cost  % If the current best is better than
the overall best
            BestSol = pop(index);  % Update BestSol
            BestCost = currentBestCost;  % Update BestCost with the current
iteration best
        end

        % Use the updated BestCost for adaptMutationRates
        [adaptive_beta_min, adaptive_beta_max] =
adaptMutationRates(adaptive_beta_min, adaptive_beta_max, params, BestCost,
previousBestCost);
```

```
            pop = restartPopulation(pop, VarMin, VarMax, VarSize,
restartFraction, CostFunction);

            % Apply local search if required
        if mod(it, 50) == 0
            pop = applyLocalSearch(pop, @localSearchFunction, CostFunction,
VarMin, VarMax);
        end

        improvementRate = abs(prevBestCost - BestCost) / prevBestCost;
        if improvementRate < 0.01
            pop = applyLocalSearch(pop, @localSearchFunction, CostFunction,
VarMin, VarMax);
        end

        % Update Best Cost Record
        allBestCosts(it, run + 1) = BestCost;

        % Store positions for this iteration
        for i = 1:parameterSets(run).nPop
            particlePositions{run}(i, :, it) = pop(i).Position;
        end

        if abs(BestSol.Cost) <= tolerance
            globalMinReached = true;
            globalMinIter(run) = it;
            if postGlobalMinIterations > 10
                break;
            end
        end

        % Update prevBestCost for the next iteration
        prevBestCost = BestCost;

        % After finding the best solution, update the BestSol structure
        [BestCost, BestIndex] = min([pop.Cost]);
        BestSol.Cost = BestCost;
        BestSol.Position = pop(BestIndex).Position;
        BestSol.Index = BestIndex; % Store the index of the best solution

        disp(['Iteration ' num2str(it) ' in Run ' num2str(run) ': Best Cost = '
num2str(BestCost)]);

        % Take snapshots at specified iterations
        if ismember(it, snapshotIterations)
            % Store population positions for the snapshot
            popPositions = reshape([pop.Position], nVar, [])';
            snapshots(run).Population = popPositions;  % Store snapshot for the
current parameter set 'run'
            disp(['Snapshot taken at Iteration ' num2str(it) ' for Parameter Set '
num2str(run)]);
            disp(['Population size: ' num2str(size(popPositions))]);  % Print out
population size for debugging
        end
    end

    % At the end of each run, create a new row in the summary table
```

```
    newSummary = table(F, CR, params.MaxIt, params.nPop, BestSol.Position(1),
BestSol.Position(2), BestSol.Cost, ...
        'VariableNames', {'F', 'CR', 'MaxIt', 'nPop', 'Best_X1', 'Best_X2',
'Best_Cost'});

    Summary = [Summary; newSummary];  % Append the new row to the summary table
end

%% Save summary Data:
disp(Summary);

% Save the table to a CSV file
writetable(Summary, 'DE_Summary.csv');

%% Save Iteration Data
csvwrite('DE_AllRunsBestCosts.csv', allBestCosts);

%% Save Medians in Table:
% Display a table of results
disp('Final Solution:');
disp(['Position = [' num2str(BestSol.Position) ']']);
disp(['Cost = ' num2str(BestSol.Cost)]);

%% Visualisation: Trajectory of Each Run's Journey up to Global Minimum
% New figure for the line graph
figure;
hold on;
colors = lines(length(parameterSets));

% Variable to store the minimum iteration for each parameter set
minIterations = zeros(length(parameterSets), 1);

for run = 1:length(parameterSets)
    params = parameterSets(run);
    itData = allBestCosts(:, run + 1);
    minIt = find(itData == min(itData), 1, 'first');
    minIterations(run) = minIt; % Store the minimum iteration
    plot(1:minIt, itData(1:minIt), 'LineWidth', 2, 'Color', colors(run, :));
end

legend(arrayfun(@(x) ['Set ' num2str(x)], 1:length(parameterSets),
'UniformOutput', false), 'Location', 'best');
xlabel('Iteration');
ylabel('Best Cost');
title('Best Cost Trajectory for Each Run');

hold off;

%% Create a 3D Surface Plot for Best Costs Over Iterations and Parameter Sets
% Determine the maximum number of iterations to display based on when global
minima were reached
maxDisplayIterations = max(globalMinIter);

% If no global minima were reached, use the maximum number of iterations for any
run
if maxDisplayIterations == 0
    maxDisplayIterations = maxIterations;
end
```

```
% Initialise the TruncatedBestCostsGrid with NaN values
TruncatedBestCostsGrid = NaN(length(parameterSets), maxDisplayIterations);

% Loop through each parameter set to fill in the best costs
for run = 1:length(parameterSets)
    if globalMinIter(run) > 0
        % Use only up to the iteration when the global minimum was reached
        TruncatedBestCostsGrid(run, 1:globalMinIter(run)) =
allBestCosts(1:globalMinIter(run), run + 1)';
    else
        % Use all iterations if the global minimum was not reached
        TruncatedBestCostsGrid(run, :) = allBestCosts(:, run + 1)';
    end
end

% Truncate the IterationGrid and ParameterSetGrid for display using the
maxDisplayIterations
[TruncatedIterationGrid, TruncatedParameterSetGrid] =
meshgrid(1:maxDisplayIterations, 1:length(parameterSets));

% Create the truncated surface plot
figure;
surf(TruncatedIterationGrid, TruncatedParameterSetGrid, TruncatedBestCostsGrid,
'EdgeColor', 'none');
xlabel('Iteration');
ylabel('Parameter Set');
zlabel('Best Cost');
title('Best Cost Surface Up to Global Minimum');
colorbar;
view(-45, 45);
colormap(jet);
legend(arrayfun(@(x) ['Set ' num2str(x) ' (global min)'], 1:length(parameterSets),
'UniformOutput', false), 'Location', 'best');

%% Contour Plot of Particle Journeys for a Selected Parameter Set
% Find the best solution from the final population of each run
bestPositions = zeros(length(parameterSets), nVar);
for run = 1:length(parameterSets)
    params = parameterSets(run);  % Use params specific for the current run
    finalPopCosts = arrayfun(@(i) CostFunction(particlePositions{run}(i, :, end)),
1:params.nPop);
    [~, idx] = min(finalPopCosts);  % Find the index of the best solution in the
final population
    bestPositions(run, :) = particlePositions{run}(idx, :, end);
end

% Create a meshgrid for the contour plot of the Rastrigin function
x = linspace(VarMin, VarMax, 100);
y = linspace(VarMin, VarMax, 100);
[X, Y] = meshgrid(x, y);
Z = arrayfun(@(x, y) CostFunction([x y]), X, Y);

% Plot the contour of the objective function
figure;
% Reduce the number of contour lines to make the plot less busy
contourf(X, Y, Z, 20); % Adjust the number to get the desired granularity
hold on;
colormap('parula'); % Set a colormap that provides a clear background
```

```
% Plot the best solutions as red balls
% Adjust the MarkerSize if needed
for i = 1:size(bestPositions, 1)
    plot(bestPositions(i, 1), bestPositions(i, 2), 'ro', 'MarkerSize', 10, ...
        'MarkerFaceColor', 'r', 'MarkerEdgeColor', 'w'); % White edge for better
visibility
end

% Label the plot
xlabel('x_1');
ylabel('x_2');
title('Best Particle Positions Contour');
hold off;

numRuns = length(snapshots);
numIterations = size(snapshots(1).Population, 3); % snapshots have the same number
of iterations

for j = 1:numIterations
    figure; % Create a new figure for each iteration
    for i = 1:numRuns
        subplot(1, numRuns, i);
        scatter(snapshots(i).Population(:, 1, j), snapshots(i).Population(:, 2,
j));
        title(sprintf('Run %d', i));
        xlabel('x1');
        ylabel('x2');
        axis([VarMin VarMax VarMin VarMax]);
        grid on;
    end
    sgtitle(sprintf('Population Snapshots at Iteration %d', j));
end
```

Constrained Himmelblau Function Code

```
function [cost] = Himmelblau_Constrained(x)
    % Himmelblau_Constrained function definition with constraints
      % In this function, Himmelblau_Constrained, the input x is a vector
containing the values of the decision variables.
      % The function calculates the value of the objective function with
constraints based on the Himmelblau function.
      % Constraints are implemented using penalty functions. Three constraints
are defined:
      % Circle constraint: x1/2 + x2/2 ≤225
      % Linear constraint: x1+x2 ≤10
    % Debs Constraint: 4*x1 - 3*x2 - 24
      % If a constraint is violated (i.e., if its value is greater than zero), a
penalty is added to the objective function.
      % The penalty parameter values k1 and k2 control the severity of the
penalty for each constraint violation.

    % Decision Variables
    x1 = x(1);
    x2 = x(2);

    % Constraints
    g1 = x1^2 + x2^2 - 225; % Circle constraint
    g2 = x1 + x2 - 10;      % Linear constraint
```

```
        g3 = 4*x1 - 3*x2 - 24;  % Debs constraint

    % Penalty Parameters
    k1 = 1000; % Penalty parameter for g1 (circle constraint)
    k2 = 1000; % Penalty parameter for g2 (linear constraint)
       k3 = 1000; % Penalty parameter for g3 (Debs constraint)

    % Objective Function
    f = (x1^2 + x2 - 11)^2 + (x1 + x2^2 - 7)^2;

    % Penalty Function
    if g1 <= 0 && g2 <= 0 && g3 <= 0
        cost = f;
    else
        cost = f + k1 * max(0, g1)^2 + k2 * max(0, g2)^2 + k3 * max(0, g3)^2;
    end
end
```

Constrained Himmelblau & RGA Code

App1.m:

```
%% Problem Definition
problem.CostFunction = @(x) Himmelblau_Constrained(x); % Himmelblau_Constrained
problem.nVar = 2; % Number of variables
problem.VarMin = [-15, -15];  % Lower Bound for each variable
problem.VarMax = [15, 15];    % Upper Bound for each variable

%% GA Parameters
params.MaxIt = 100; % Maximum number of iterations
params.nPop = 250; % Population size
pop = zeros(numIndividuals, 4); % Initialise matrix for population for Welded Beam
Only, comment out when change
params.pC = 0.7; % Crossover probability
params.mu = 0.02; % Mutation rate
params.maxAge = 5; % Maximum age for AgeBasedSurvivor
params.tournamentSize = 3; % Tournament size for selection

% Number of runs
numRuns = 6;

% Perform the runs for each combination of parameters
runCounter = 0; % Initialise run counter

% Initialise matrix to hold all runs
BestCostsMatrix = zeros(params.MaxIt, numRuns);

% Define a broader and more varied range of parameters for mutation and crossover
mutationRates = [0.01, 0.015, 0.02, 0.025, 0.03, 0.035, 0.04, 0.045, 0.05, 0.06,
0.07, 0.08, 0.09, 0.1];
crossoverRates = [0.5, 0.55, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 1.0];

% Prepare a structure to store the results
results = struct();

% Perform the runs for each combination of parameters
for mu = mutationRates
    for pc = crossoverRates
        if runCounter >= numRuns
```

```
            break; % Exit the inner loop if we have reached the desired number of
runs
        end
        params.mu = mu; % Set the mutation rate
        params.pC = pc; % Set the crossover probability

        % Run the GA with the current set of parameters
        out = RunGA(problem, params);

        % Generate a valid field name by formatting numbers to avoid scientific
notation
        mu_str = strrep(num2str(mu, '%.6f'), '.', 'p'); % Replace period with 'p'
        pc_str = strrep(num2str(pc, '%.6f'), '.', 'p'); % Replace period with 'p'

        runID = sprintf('run_mu%s_pc%s', mu_str, pc_str);

        % Store the results with the valid field name
        results.(runID).bestcost = out.bestcost;
        results.(runID).diversity = out.diversity;
        results.(runID).avgAge = out.avgAge;
        results.(runID).selectionEffectiveness = out.selectionEffectiveness;

        % Store the best solution
        results.(runID).bestSolution = out.bestsol.Position;

        runCounter = runCounter + 1; % Increment run counter after each GA run
    end
    if runCounter >= numRuns
        break; % Exit the outer loop if we have reached the desired number of runs
    end
end

%% Save to spreadsheet
% Prepare the matrix to hold the best costs from each run
numRuns = numel(fieldnames(results));
numIters = params.MaxIt;
BestCostsMatrix = zeros(numIters, numRuns);

% Fill the matrix with the best cost data from each run
runCounter = 1;
for runID = fieldnames(results)'
    BestCostsMatrix(:, runCounter) = results.(runID{1}).bestcost;
    runCounter = runCounter + 1;
end

% Define filename for the Excel file
filename = 'BestCostsMultipleRuns.xlsx';

% Save the matrix to Excel
writematrix(BestCostsMatrix, filename);

%% Run GA
out = RunGA(problem, params);

%% Results Visualisations
runIDs = fieldnames(results);
numRuns = length(runIDs);

% Initialise an array to hold the final best cost of each run
```

```
finalBestCosts = zeros(1, numRuns);

% Populate the array with the final best cost of each run
for i = 1:numRuns
    finalBestCosts(i) = results.(runIDs{i}).bestcost(end);
end

% Sort the runs by their final best cost in ascending order
[sortedCosts, sortIndex] = sort(finalBestCosts);

% Select the indices of the 10 best runs, ensuring not to exceed the total number
of runs
topRunsIndex = sortIndex(1:min(10, numel(sortedCosts)));

% Initialise the figure for plotting
figure;

% Adjust the loop to iterate over indices of the selected runs
for idx = 1:length(topRunsIndex)
    selectedRunID = runIDs{topRunsIndex(idx)};
    subplot(2, 5, idx);  % Arrange the subplots in 2 rows and 5 columns
    plot(results.(selectedRunID).bestSolution, 'LineWidth', 2);

    % Use regular expression to extract mutation and crossover values
    tokens = regexp(selectedRunID, 'run_mu(\d+p\d+)_pc(\d+p\d+)', 'tokens');
    if isempty(tokens)
        error('The runID format does not match the expected pattern.');
    end

    % Convert the mutation and crossover values back to numeric format
    muValue = str2double(strrep(tokens{1}{1}, 'p', '.'));  % Replace 'p' back with
period
    pcValue = str2double(strrep(tokens{1}{2}, 'p', '.'));  % Replace 'p' back with
period
    title(sprintf('Mutation: %.4f, Crossover: %.2f', muValue, pcValue));

    xlabel('Gene Position');
    ylabel('Gene Expression');
    grid on;
end
annotation('textbox', [0.5, 0.9, 0.1, 0.1], 'String', 'Mutation vs Crossover Run
Comparison', ...
        'EdgeColor', 'none', 'HorizontalAlignment', 'center', 'FontSize', 14);

% Best Cost Evolution
figure;
semilogy(out.bestcost, 'LineWidth', 2);
title('Best Cost Evolution');
xlabel('Iteration');
ylabel('Best Cost');
grid on;

% Diversity Measurement
figure;
plot(out.diversity, 'LineWidth', 2);
title('Population Diversity Over Generations');
xlabel('Generation');
ylabel('Diversity (Standard Deviation of Fitness)');
grid on;
```

```
% Average Age Distribution
figure;
plot(out.avgAge, 'LineWidth', 2);
title('Average Age of Individuals Over Generations');
xlabel('Generation');
ylabel('Average Age');
grid on;

% Selection Effectiveness
figure;
plot(out.selectionEffectiveness, 'LineWidth', 2);
legend({'Selected Individuals', 'Not Selected Individuals'}, 'Location', 'Best');
title('Selection Effectiveness Over Generations');
xlabel('Generation');
ylabel('Average Fitness');
grid on;

%% New visualisations:
% Plot the best cost evolution for each parameter set on the same graph
figure;
hold on;
for runID = fieldnames(results)'
    % Extract the actual mutation and crossover values from runID for the legend
    tokens = regexp(runID{1}, 'run_mu(\d+p\d+)_pc(\d+p\d+)', 'tokens');
    if isempty(tokens)
        error('The runID format does not match the expected pattern.');
    end

    muValue = str2double(strrep(tokens{1}{1}, 'p', '.'));  % Convert 'p' back to
period
    pcValue = str2double(strrep(tokens{1}{2}, 'p', '.'));  % Convert 'p' back to
period
    descriptiveName = sprintf('Mutation: %.4f, Crossover: %.2f', muValue,
pcValue);

    % Plot the best cost evolution with a semilogarithmic scale
    semilogy(results.(runID{1}).bestcost, 'LineWidth', 2, 'DisplayName',
descriptiveName);
end

title('Best Cost Evolution Across Parameter Sets');
xlabel('Iteration');
ylabel('Best Cost');
legend('show');
grid on;
hold off;

%% Surf Visualisation:
% Initialise the matrix for storing best costs
bestCostsMatrix = zeros(length(mutationRates), length(crossoverRates));

% Iterating over mutation and crossover rates to run the GA
for i = 1:length(mutationRates)
    for j = 1:length(crossoverRates)
        params.mu = mutationRates(i); % Set the mutation rate
        params.pC = crossoverRates(j); % Set the crossover probability

        % Run the GA with the current set of parameters
```

```
        out = RunGA(problem, params);

        % Store the best cost in the matrix
        bestCostsMatrix(i, j) = out.bestcost(end); % Assuming 'out.bestcost' gives
the final best cost of the run
    end
end

% Create meshgrid for mutation and crossover rates
[X, Y] = meshgrid(mutationRates, crossoverRates);

% Generating the surf plot
figure;
surf(X, Y, bestCostsMatrix'); % Transpose bestCostsMatrix to match the orientation
of X and Y in the plot
xlabel('Mutation Rate');
ylabel('Crossover Rate');
zlabel('Best Cost');
title('Performance Landscape of Mutation and Crossover Rates Vs Best Cost');
```

RunGA.m

```
function out = RunGA(problem, params)

    % Problem
    CostFunction = problem.CostFunction;
    nVar = problem.nVar;

    % Params
    MaxIt = params.MaxIt;
    nPop = params.nPop;
    pC = params.pC;
    nC = round(pC*nPop/2)*2;
    mu = params.mu;
    maxAge = params.maxAge; % New parameter for AgeBasedSurvivor

    % Data Collection Initialisation
    diversity = zeros(MaxIt, 1);
    avgAge = zeros(MaxIt, 1);
    selectionEffectiveness = zeros(MaxIt, 2); % [average fitness of selected,
average fitness of not selected]

    % Template for Empty Individuals
    empty_individual.Position = [];
    empty_individual.Cost = [];
    empty_individual.Age = 0;

    noImproveCounter = 0;
    bestsol.Cost = inf;

    % Population Initialisation
    pop = repmat(empty_individual, nPop, 1);
    for i = 1:nPop
        pop(i).Position = problem.VarMin + rand(1, problem.nVar) .*
(problem.VarMax - problem.VarMin);
        pop(i).Cost = CostFunction(pop(i).Position);
        pop(i).Age = 0; % Initial age
        if pop(i).Cost < bestsol.Cost
            bestsol = pop(i);
```

```
            bestCost = pop(i).Cost; % Initialise the bestCost with the cost of the
best initial individual
        end
    end

    % Best Cost of Iterations
    bestcost = nan(MaxIt, 1);

    % Main Loop
    for it = 1:MaxIt

        % Increment age of all individuals
        for i = 1:nPop
            pop(i).Age = pop(i).Age + 1;
        end

        % Initialise Offsprings Population
        popc = repmat(empty_individual, nC/2, 2);

        % Pre-selection for analysis purposes
        preSelectedFitness = [];
        notSelectedFitness = [];

        % Crossover (HUXCrossover Update)
        for k = 1:nC/2
            % Select Parents (TournamentSelection)
            p1 = TournamentSelection(pop, params.tournamentSize);
            p2 = TournamentSelection(pop, params.tournamentSize);

            % Collecting fitness data for selection analysis
            preSelectedFitness = [preSelectedFitness, pop(p1).Cost, pop(p2).Cost];

            % Perform HUXCrossover
            [popc(k, 1).Position, popc(k, 2).Position] =
HUXCrossover(pop(p1).Position, pop(p2).Position);
        end

        % Post-selection analysis
        allIndices = 1:nPop;
        notSelectedIndices = setdiff(allIndices, [p1, p2]);
        for idx = notSelectedIndices
            notSelectedFitness = [notSelectedFitness, pop(idx).Cost];
        end

        selectionEffectiveness(it, :) = [mean(preSelectedFitness),
mean(notSelectedFitness)];

        % Convert popc to Single-Column Matrix
        popc = popc(:);

        % Mutation
        for l = 1:nC
            popc(l).Position = NonUniformMutation(popc(l).Position, mu, it,
params.MaxIt, problem.VarMin, problem.VarMax);
            popc(l).Cost = CostFunction(popc(l).Position);
            popc(l).Age = 0; % Reset age for new offspring
            if popc(l).Cost < bestsol.Cost
                bestsol = popc(l);
            end
```

```
        end

        % Merge Populations
        pop = [pop; popc];

        % Calculate diversity and average age
        currentFitness = [pop.Cost];
        diversity(it) = std(currentFitness);
        avgAge(it) = mean([pop.Age]);

             % Maintain diversity in the population
             pop = maintainDiversity(pop, 0.1, problem.VarMin, problem.VarMax,
CostFunction);

             % Check for improvement and possibly escape local optima
             currentBestCost = min([pop.Cost]);
             if currentBestCost < bestCost
                   bestCost = currentBestCost;
                   noImproveCounter = 0;
             else
                   noImproveCounter = noImproveCounter + 1;
             end
             [pop, noImproveCounter] = escapeLocalOptima(pop, bestCost,
noImproveCounter, 20, params.mu, problem.VarMin, problem.VarMax, CostFunction);

             % Adapt mutation and crossover rates
             params.mu = adaptiveRate(it, params.MaxIt, 0.1, 0.01);
             params.pC = adaptiveRate(it, params.MaxIt, 0.9, 0.6);

        % Age-Based Survivor Selection
        pop = AgeBasedSurvivor(pop, maxAge, nPop);

        % Update Best Cost of Iteration
        bestcost(it) = bestsol.Cost;

        % Display Iteration Information
        disp(['Iteration ' num2str(it) ': Best Cost = ' num2str(bestcost(it))]);
    end

    % Include collected data in the output
    out.diversity = diversity;
    out.avgAge = avgAge;
    out.selectionEffectiveness = selectionEffectiveness;

    % Results
    out.pop = pop;
    out.bestsol = bestsol;
    out.bestcost = bestcost;

end
```

adaptiveRate.m

```
function rate = adaptiveRate(iteration, maxIter, initialRate, finalRate)
    % Linearly adapt the rate from initial to final over the iterations
    rate = initialRate - (iteration / maxIter) * (initialRate - finalRate);
    rate = max(rate, finalRate); % Ensure rate does not go below finalRate
end
```

AgeBasedSurvivor.m:

```
function pop = AgeBasedSurvivor(pop, maxAge, nPop)
    % Eliminate individuals exceeding maxAge by setting their cost to infinity
    for i = 1:length(pop)
        if pop(i).Age > maxAge
            pop(i).Cost = inf;  % Mark for elimination
        end
    end

    % Sort the population based on cost, which moves individuals marked for
elimination to the end
    pop = SortPopulation(pop);

    % Ensure the population size does not drop below nPop
    if length(pop) > nPop
        pop = pop(1:nPop); % Keep only the top nPop individuals
    end
end
```

BiasedUniformCrossover.m

```
function [y1, y2] = BiasedUniformCrossover(x1, x2, bias)
    % BiasedUniformCrossover performs uniform crossover with a bias towards one
parent
    % Inputs: x1, x2 are parent binary strings (vectors), bias is the probability
of choosing a gene from x1
    % Outputs: y1, y2 are offspring binary strings (vectors)

    % Initialise offspring
    y1 = zeros(1, length(x1));
    y2 = zeros(1, length(x2));

    for i = 1:length(x1)
        if rand <= bias
            y1(i) = x1(i);
            y2(i) = x2(i);
        else
            y1(i) = x2(i);
            y2(i) = x1(i);
        end
    end
end
```

escapeLocalOptima.m

```
function [pop, noImproveCounter] = escapeLocalOptima(pop, bestCost,
noImproveCounter, threshold, mutationRate, VarMin, VarMax, CostFunction)
    if noImproveCounter >= threshold
        for i = 1:numel(pop)
            % Here, create a new individual and assign its fields separately
            newIndividual.Position = VarMin + (VarMax - VarMin) .* rand(1,
length(VarMin));
            newIndividual.Cost = CostFunction(newIndividual.Position);
            newIndividual.Age = 0;

            % Now assign the new individual to the population
            pop(i) = newIndividual;
        end
        % Reset the no improvement counter after the escape
        noImproveCounter = 0;
    end
```

```
end
```

HUXCrossover.m

```
function [y1, y2] = HUXCrossover(x1, x2)
    % HUXCrossover performs half uniform crossover on binary strings
    % Inputs: x1, x2 are parent binary strings (vectors)
    % Outputs: y1, y2 are offspring binary strings (vectors)

    % Identifying indices where the parents differ
    diffIndices = find(x1 ~= x2);

    % Randomly selecting half of the differing indices to swap
    indicesToSwap = randsample(diffIndices, floor(length(diffIndices)/2));

    % Creating offspring by copying parents
    y1 = x1;
    y2 = x2;

    % Swapping the selected bits
    y1(indicesToSwap) = x2(indicesToSwap);
    y2(indicesToSwap) = x1(indicesToSwap);
end
```

MaintainDiversity.m

```
function pop = maintainDiversity(pop, diversityThreshold, VarMin, VarMax,
CostFunction)
    % Extract costs from the population structures for diversity calculation
    costs = [pop.Cost];

    % Calculate diversity as the standard deviation of the costs
    diversity = std(costs);

    if diversity < diversityThreshold
        % Determine the number of new individuals to introduce
        numNewIndividuals = ceil(0.1 * numel(pop));

        for i = 1:numNewIndividuals
            % Generate a new individual within the variable bounds
            pop(i).Position = VarMin + rand(1, size(VarMin, 2)) .* (VarMax -
VarMin);
            % Calculate the Cost for the new individual
            pop(i).Cost = CostFunction(pop(i).Position);
            % Reset the Age for the new individual
            pop(i).Age = 0;
        end
    end
end
```

NonUniformMutation.m

```
function [NewPosition] = NonUniformMutation(OldPosition, mu, CurrentIteration,
MaxIterations, VarMin, VarMax)
    % NonUniformMutation Applies non-uniform mutation to a candidate solution
    %   OldPosition: the current position of the solution
    %   mu: mutation rate
    %   CurrentIteration: the current iteration number
    %   MaxIterations: the maximum number of iterations
    %   VarMin: the minimum bounds for the decision variables
    %   VarMax: the maximum bounds for the decision variables

    % Initialise the new position with the old position
```

```
    NewPosition = OldPosition;

    % Number of decision variables
    nVar = numel(OldPosition);

    % Mutation parameter 'b' determines the degree of non-uniformity
    % Typically set to a value between 2 and 5
    b = 3;

    % Apply non-uniform mutation to each decision variable
    for j = 1:nVar
        % Only mutate with a probability of 'mu'
        if rand() <= mu
            % Choose the direction of mutation randomly
            if rand() < 0.5
                % Mutate towards upper bound
                delta = VarMax(j) - OldPosition(j);
                NewPosition(j) = OldPosition(j) + delta * (1 - rand()^((1 -
CurrentIteration / MaxIterations)^b));
            else
                % Mutate towards lower bound
                delta = OldPosition(j) - VarMin(j);
                NewPosition(j) = OldPosition(j) - delta * (1 - rand()^((1 -
CurrentIteration / MaxIterations)^b));
            end

            % Ensure the new position respects the variable bounds
            NewPosition(j) = max(NewPosition(j), VarMin(j));
            NewPosition(j) = min(NewPosition(j), VarMax(j));
        end
    end
end
```

SortPopulation.m

```
function pop = SortPopulation(pop)

    [~, so] = sort([pop.Cost]);
    pop = pop(so);

end
```

TournamentSelection.m

```
function i = TournamentSelection(pop, tournamentSize)
    % tournamentSize is the number of individuals to be selected for the
tournament

    % Randomly select 'tournamentSize' individuals
    selectedIndices = randperm(length(pop), tournamentSize);
    tournamentIndividuals = pop(selectedIndices);

    % Find the best individual in the tournament
    [~, bestIndex] = min([tournamentIndividuals.Cost]);
    i = selectedIndices(bestIndex);
end
```

Constrained Himmelblau & PSO Code

AppPSO.m

```
clc; clear;
```

```
%% Problem Definition
problem.CostFunction = @(x) Himmelblau_Constrained(x);  % Problem function
problem.nVar = 2;
problem.VarMin = [-15, -15];  % Lower Bound for each variable
problem.VarMax = [15, 15];    % Upper Bound for each variable
globalOptimumCost = 0;

%% Parameters of PSO

% Constriction Coefficients
kappa = 1;
phi1 = 2.05;
phi2 = 2.05;
phi = phi1 + phi2;
chi = 2*kappa/abs(2-phi-sqrt(phi^2-4*phi));

params.w = chi;             % Intertia Coefficient
params.wdamp = 1;           % Damping Ratio of Inertia Coefficient
params.c1 = chi*phi1;       % Personal Acceleration Coefficient
params.c2 = chi*phi2;       % Social Acceleration Coefficient
params.ShowIterInfo = true; % Flag for Showing Iteration Information

% Tolerance and global minimum
tolerance = 1e-2;
globalMin = [0, 0];  % Theoretical global minimum

% Initialise the `results` structure
results = struct();

% Define the number of runs you want to perform
numberOfRuns = 1;

% Array of parameter sets for different runs
paramSets = [
     struct('MaxIt', 1000, 'nPop', 250, 'w', 0.20, 'wdamp', 0.20, 'c1', 0.10,
'c2', 0.25, 'ShowIterInfo', true);
     struct('MaxIt', 1000, 'nPop', 100, 'w', 0.40, 'wdamp', 0.45, 'c1', 0.30,
'c2', 0.50, 'ShowIterInfo', true);
     struct('MaxIt', 1000, 'nPop', 70, 'w', 0.60, 'wdamp', 0.65, 'c1', 0.50, 'c2',
0.75, 'ShowIterInfo', true);
     struct('MaxIt', 1000, 'nPop', 40, 'w', 0.80, 'wdamp', 0.79, 'c1', 0.70, 'c2',
0.85, 'ShowIterInfo', true);
     struct('MaxIt', 1000, 'nPop', 150, 'w', 1.00, 'wdamp', 0.99, 'c1', 0.90,
'c2', 0.99, 'ShowIterInfo', true);
];

% Initialise arrays to store results
allBestSolutions = zeros(length(paramSets), problem.nVar);
allBestCosts = zeros(length(paramSets), numberOfRuns);

% Loop through each parameter set
for runIndex = 1:numberOfRuns
    for i = 1:length(paramSets)
        params = paramSets(i);

        % PSO Main Body
        out = PSO(problem, params);  % Use the PSO function with the current
parameter set
```

```
        % Store Results
        allBestSolutions(i, :) = out.BestSol.Position;
        allBestCosts(i, runIndex) = out.BestSol.Cost;
        BestSol = out.BestSol;
        BestCosts = out.BestCosts;
        particle_history = out.particle_history;
        particleHistory = out.particleHistory;
        allVelocityHistory = out.velocityHistory;  % Store the velocity history
for this run

        % Tolerance Check
        error = max(abs(out.BestSol.Position - globalMin));
        if error < tolerance
            fprintf('Run %d: Solution found within the acceptable tolerance.\n',
i);
        else
            fprintf('Run %d: Best Cost = %.4f, not within the acceptable tolerance
after %d iterations.\n', i, allBestCosts(i), params.MaxIt);
        end

        % Save Results to Spreadsheet for each parameter set
        saveHistoricalData(out.particleHistory,
sprintf('historicalData_Run%d.xlsx', i));

        figure; % Create a new figure for contour plot
        plotPopulationContour(problem.CostFunction, out.pop, problem.VarMin,
problem.VarMax, params.nPop);
        title(sprintf('Population Contour (Run %d)', i));
        drawnow; % Ensure plots are updated before the next iteration

        runID = sprintf('run%d', runIndex);
        results.(runID) = out;

    end
end

% Compile all results into a table after all runs are complete
allEntries = cell(length(paramSets), 9); % 9 for each of the parameters and
results

for i = 1:length(paramSets)
    allEntries(i,:) = {paramSets(i).MaxIt, paramSets(i).nPop, paramSets(i).w, ...
        paramSets(i).wdamp, paramSets(i).c1, paramSets(i).c2, ...
        allBestSolutions(i, 1), allBestSolutions(i, 2), allBestCosts(i)};
end

resultsTable = cell2table(allEntries, 'VariableNames', ...
    {'MaxIt', 'nPop', 'w', 'wdamp', 'c1', 'c2', 'Var1', 'Var2', 'BestCost'});

% Write the compiled results to the spreadsheet
writetable(resultsTable, 'PSO_Results.xlsx');

% Debug:
disp(size(allBestCosts));
disp(allBestCosts);

%% Visualisations
plotVelocityMagnitudes(out.velocityHistory);
```

```
plotDiversity(particle_history);
plotConvergence(BestCosts, globalOptimumCost);
plotPopulationContour(problem.CostFunction, out.pop, problem.VarMin,
problem.VarMax, params.nPop);
% Call the animateParticles function
animateParticles(particleHistory, BestCosts, params.MaxIt, params.nPop);
```

PSO.m

```
function out = PSO(problem, params)
    %% Problem Definiton
    CostFunction = problem.CostFunction;  % Cost Function
    nVar = problem.nVar;        % Number of Unknown (Decision) Variables
    VarSize = [1 nVar];         % Matrix Size of Decision Variables
    VarMin = problem.VarMin;     % Lower Bound of Decision Variables
    VarMax = problem.VarMax;    % Upper Bound of Decision Variables


    %% Parameters of PSO
    MaxIt = params.MaxIt;   % Maximum Number of Iterations
    nPop = params.nPop;     % Population Size (Swarm Size)
    w = params.w;           % Intertia Coefficient
    wdamp = params.wdamp;   % Damping Ratio of Inertia Coefficient
    c1 = params.c1;         % Personal Acceleration Coefficient
    c2 = params.c2;         % Social Acceleration Coefficient

    % The Flag for Showing Iteration Information
    ShowIterInfo = params.ShowIterInfo;

    MaxVelocity = 0.2*(VarMax-VarMin);
    MinVelocity = -MaxVelocity;

    particle_history = zeros(MaxIt, nPop, nVar);

    % Initialise a matrix to store particle history (needed for animation)
    particleHistory = zeros(nPop, nVar, MaxIt);

    %% Initialisation
    % The Particle Template
    empty_particle.Position = [];
    empty_particle.Velocity = [];
    empty_particle.Cost = [];
    empty_particle.Best.Position = [];
    empty_particle.Best.Cost = [];

    % Initialise velocity history
    velocityHistory = zeros(nPop, nVar, MaxIt);

    % Create Population Array
    particle = repmat(empty_particle, nPop, 1);

    % Initialise Global Best
    GlobalBest.Cost = inf;

    % Initialise Population Members
    for i=1:nPop

        % Generate Random Solution
        particle(i).Position = unifrnd(VarMin, VarMax, VarSize);
```

```
        % Initialise Velocity
        particle(i).Velocity = zeros(VarSize);

        % Evaluation
        particle(i).Cost = CostFunction(particle(i).Position);

        % Update the Personal Best
        particle(i).Best.Position = particle(i).Position;
        particle(i).Best.Cost = particle(i).Cost;

        % Update Global Best
        if particle(i).Best.Cost < GlobalBest.Cost
            GlobalBest = particle(i).Best;
        end

    end

    % Array to Hold Best Cost Value on Each Iteration
    BestCosts = zeros(MaxIt, 1);

    %% Main Loop of PSO

    for it=1:MaxIt

        for i=1:nPop

            % Update Velocity
            particle(i).Velocity = w*particle(i).Velocity ...
                + c1*rand(VarSize).*(particle(i).Best.Position -
particle(i).Position) ...
                + c2*rand(VarSize).*(GlobalBest.Position - particle(i).Position);

            % Apply Velocity Limits
            particle(i).Velocity = max(particle(i).Velocity, MinVelocity);
            particle(i).Velocity = min(particle(i).Velocity, MaxVelocity);

            % Update Position
            particle(i).Position = particle(i).Position + particle(i).Velocity;

            % Apply Lower and Upper Bound Limits
            particle(i).Position = max(particle(i).Position, VarMin);
            particle(i).Position = min(particle(i).Position, VarMax);

            % Evaluation
            particle(i).Cost = CostFunction(particle(i).Position);

            % Store the history
            particle_history(it, i, :) = particle(i).Position;
            velocityHistory(i, :, it) = particle(i).Velocity; % To store velocity

            % Update Personal Best
            if particle(i).Cost < particle(i).Best.Cost

                particle(i).Best.Position = particle(i).Position;
                particle(i).Best.Cost = particle(i).Cost;

                % Update Global Best
                if particle(i).Best.Cost < GlobalBest.Cost
                    GlobalBest = particle(i).Best;
```

```
                end

            end

        end

        % Store positions for animation
        for i = 1:nPop
            particleHistory(i, :, it) = particle(i).Position;
        end

        % Store the Best Cost Value
        BestCosts(it) = GlobalBest.Cost;

             %% Enhanced Display Iteration Information
             if ShowIterInfo
                   disp(['Iteration ' num2str(it) ': Best Cost = '
num2str(BestCosts(it))]);
                   fprintf('Current Best Position: [%f, %f]\n',
GlobalBest.Position); % Prints the current best position
             end

        % Damping Inertia Coefficient
        w = w * wdamp;

    end

    out.pop = particle;
    out.BestSol = GlobalBest;
    out.BestCosts = BestCosts;
    out.particle_history = particle_history;
    out.particleHistory = particleHistory;
    out.particle_final_positions = [particle.Position];
    out.velocityHistory = velocityHistory;

end
```

animateParticles.m

```
function animateParticles(particleHistory, BestCosts, MaxIt, nPop)
    figure(5);
    plotHandle = scatter3(NaN, NaN, NaN, 'o', 'MarkerEdgeColor', 'r',
'MarkerFaceColor', 'r');
    xlabel('Position X');
    ylabel('Position Y');
    zlabel('BestCosts');

    % Create a VideoWriter object to save the animation
    writerObj = VideoWriter('particle_animation.avi');
    writerObj.FrameRate = 10; % Set the frame rate (frames per second)
    open(writerObj); % Open the VideoWriter object

    for it = 1:MaxIt
        xPos = particleHistory(:, 1, it);
        yPos = particleHistory(:, 2, it);
        zPos = repmat(BestCosts(it), 1, nPop);
        set(plotHandle, 'XData', xPos, 'YData', yPos, 'ZData', zPos);
        drawnow;

        % Get the current frame
```

```
        frame = getframe(gcf);

        % Write the current frame to the video
        writeVideo(writerObj, frame);

        pause(0.1);
    end

    % Close the VideoWriter object
    close(writerObj);
end
```

plotConvergence.m

```
function plotConvergence(bestCosts, globalOptimumCost)
    figure;
    % Calculate the absolute difference between the best cost of each iteration
and the global optimum cost
    convergenceRate = abs(bestCosts - globalOptimumCost);

    plot(convergenceRate, 'LineWidth', 2);
    title('Convergence Plot');
    xlabel('Iteration');
    ylabel('Absolute Error from Global Optimum');
    grid on;
end
```

plotDiversity.m

```
function plotDiversity(particle_history)
    figure;
    numIterations = size(particle_history, 1); % to match the first dimension
    diversity = zeros(1, numIterations);

    for i = 1:numIterations
        % The dimensions are reordered to match particle_history structure
        currentPositions = squeeze(particle_history(i, :, :));
        % Calculate pairwise distances between all particles
        distances = pdist(currentPositions, 'euclidean');
        % Average distance as a measure of diversity
        diversity(i) = mean(distances);
    end

    plot(diversity, 'LineWidth', 2);
    title('Diversity Measurement Over Time');
    xlabel('Iteration');
    ylabel('Average Distance Between Particles');
    grid on;
end
```

plotPopulationContour.m

```
function plotPopulationContour(CostFunction, particle, VarMin, VarMax, nPop)
    % Prepare Mesh for Contour Plots
    x1 = linspace(VarMin(1), VarMax(1), 100);
    x2 = linspace(VarMin(2), VarMax(2), 100);
    [X1, X2] = meshgrid(x1, x2);

    % Evaluate the Cost Function on the grid
    Z = arrayfun(@(x, y) CostFunction([x, y]), X1, X2);

    % Clear and hold the figure
    clf; % Clears current figure window and resets hold state
```

```
    hold on;

    % Contour Plot of Cost Function
    contourf(X1, X2, log(Z+1), 20); % Filled contour plot with more levels
    colormap(jet); % Change to a more colorful colormap
    colorbar; % Optionally add a color bar to indicate the scale

    % Plot whole Population as red stars
    for i = 1:nPop
        plot(particle(i).Position(1), particle(i).Position(2), 'r*', 'LineWidth',
2);
    end

    % Enhancements
    title('Particle Positions and Cost Function Contour');
    xlabel('Variable x1');
    ylabel('Variable x2');
    axis equal;
    grid on; % Optionally add a grid
    hold off;
end
```

plotVelocityMagnitudes.m

```
function plotVelocityMagnitudes(particleVelocityHistory)
    figure;
    % Calculate the magnitude of velocity for each particle at each iteration
    velocityMagnitudes = squeeze(sqrt(sum(particleVelocityHistory.^2, 3)));

    plot(velocityMagnitudes, 'LineWidth', 1);
    title('Velocity Magnitudes Over Time');
    xlabel('Iteration');
    ylabel('Velocity Magnitude');
    legend('Particle 1', 'Particle 2', 'Location', 'Best'); % Adjust the legend
accordingly
    grid on;
end
```

saveHistoricalData.m

```
function saveHistoricalData(particleHistory, filename)
    % Determine the size of particleHistory
    [numParticles, numDimensions, numIterations] = size(particleHistory);

    % Flatten the 3D matrix to a 2D cell array for saving
    flatData = reshape(particleHistory, numParticles, numDimensions *
numIterations);

    % Create a header for the spreadsheet
    header = cell(1, numDimensions * numIterations);
    for it = 1:numIterations
        for dim = 1:numDimensions
            header{(it - 1) * numDimensions + dim} = sprintf('Var%d_Iter%d', dim,
it);
        end
    end

    % Combine header and data
    outputCell = [header; num2cell(flatData)];
```

```
    % Write header and data to file
    writecell(outputCell, filename);
end
```

Constrained Himmelblau & DE Code

NB: As there are 2 versions, that use the same functions, I'll include both main files and add the functions that was used in both.

de.m (DE/1)

```
%% DE /best/1/bin with Dynamic Parameter Testing incorporating a halt 10
iterations after achieving the global minimum within specified tolerance
clc; clear;

%% Problem Definition
CostFunction = @(x) Himmelblau_Constrained(x);  % Cost Function
nVar = 2;                           % Number of Decision Variables
VarSize = [1 nVar];                 % Decision Variables Matrix Size
VarMin = -15.12;                     % Lower Bound of Decision Variables
VarMax = 15.12;                      % Upper Bound of Decision Variables
tolerance = 1e-5;                   % Tolerance for considering the global minimum
achieved

immigrantFraction = 0.1;            % adjust as needed
prevBestCost = inf;                 % Initialise with a high value
restartFraction = 0.2;              % Restart 20% of the population

%% Parameter Sets as Struct Array
parameterSets = [
    struct('MaxIt', 1000, 'nPop', 150, 'beta_min', 0.4, 'beta_max', 0.8, 'pCR',
0.4, 'immigrantFraction', 0.1, 'rateIncrease', 0.01, 'maxBetaMin', 0.1,
'minBetaMax', 0.7);  % Set 1
    struct('MaxIt', 900, 'nPop', 200, 'beta_min', 0.2, 'beta_max', 0.8, 'pCR',
0.2, 'immigrantFraction', 0.05, 'rateIncrease', 0.05, 'maxBetaMin', 0.2,
'minBetaMax', 0.8); % Set 2
    struct('MaxIt', 800, 'nPop', 150, 'beta_min', 0.1, 'beta_max', 0.9, 'pCR',
0.3, 'immigrantFraction', 0.2, 'rateIncrease', 0.09, 'maxBetaMin', 0.3,
'minBetaMax', 0.9); % Set 3
    struct('MaxIt', 700, 'nPop', 150, 'beta_min', 0.5, 'beta_max', 0.8, 'pCR',
0.1, 'immigrantFraction', 0.15, 'rateIncrease', 0.1, 'maxBetaMin', 0.4,
'minBetaMax', 0.7);  % Set 4
    struct('MaxIt', 600, 'nPop', 200, 'beta_min', 0.3, 'beta_max', 0.7, 'pCR',
0.6, 'immigrantFraction', 0.25, 'rateIncrease', 0.15, 'maxBetaMin', 0.5,
'minBetaMax', 0.8);% Set 5
    struct('MaxIt', 500, 'nPop', 150, 'beta_min', 0.6, 'beta_max', 0.8, 'pCR',
0.5, 'immigrantFraction', 0.4, 'rateIncrease', 0.2, 'maxBetaMin', 0.6,
'minBetaMax', 0.9)   % Set 6
];

%% Initialise Results Storage
allBestCosts = zeros(max([parameterSets.MaxIt]), length(parameterSets) + 1); % +1
for iteration numbers
allBestCosts(:, 1) = 1:max([parameterSets.MaxIt]); % Column 1 is iteration numbers
globalMinIter = zeros(1, length(parameterSets)); % Initialise globalMinIter
maxIterations = max([parameterSets.MaxIt]);
particlePositions = cell(length(parameterSets), 1);

%% Main Loop Over Parameter Sets
for run = 1:length(parameterSets)
```

```
    params = parameterSets(run);
    adaptive_beta_min = params.beta_min;
    adaptive_beta_max = params.beta_max;
    previousBestCost = inf; % For tracking improvement
    particlePositions{run} = zeros(parameterSets(run).nPop, nVar,
parameterSets(run).MaxIt);

    % Initialise Population
    pop = struct('Position', [], 'Cost', inf(params.nPop, 1));
    for i = 1:params.nPop
        pop(i).Position = unifrnd(VarMin, VarMax, VarSize);
        pop(i).Cost = CostFunction(pop(i).Position);
    end
    [BestCost, index] = min([pop.Cost]);
    BestSol = pop(index);

    %% DE Main Loop
    for it = 1:params.MaxIt
             for i = 1:params.nPop
                   % Mutation
                   A = randperm(params.nPop, 3);
                   a = A(1);
                   b = A(2);
                   c = A(3);
                   y = pop(a).Position + params.beta_min*(pop(b).Position -
pop(c).Position);
                   y = max(y, VarMin);
                   y = min(y, VarMax);

                   % Crossover
                   z = pop(i).Position;
                   jRand = randi([1 numel(z)]);
                   for j = 1:numel(z)
                         if rand <= params.pCR || j == jRand
                               z(j) = y(j);
                         end
                   end

                   % Selection
                   NewSol.Position = z;
                   NewSol.Cost = CostFunction(NewSol.Position);
                   if NewSol.Cost < pop(i).Cost
                         pop(i) = NewSol;
                         if NewSol.Cost < BestSol.Cost
                               BestSol = NewSol;
                         end
                   end
             end

        % Introduce Random Immigrants
        pop = introduceRandomImmigrants(pop, CostFunction, VarMin, VarMax,
VarSize, params);

        % Update BestSol and BestCost logic...
        [currentBestCost, index] = min([pop.Cost]);  % Find the current best cost
and index
        if currentBestCost < BestSol.Cost  % If the current best is better than
the overall best
            BestSol = pop(index);  % Update BestSol
```

```
            BestCost = currentBestCost;  % Update BestCost with the current
iteration best
        end

        % Use the updated BestCost for adaptMutationRates
        [adaptive_beta_min, adaptive_beta_max] =
adaptMutationRates(adaptive_beta_min, adaptive_beta_max, params, BestCost,
previousBestCost);

             pop = restartPopulation(pop, VarMin, VarMax, VarSize,
restartFraction, CostFunction);

             % Apply local search if required
        if mod(it, 50) == 0
            pop = applyLocalSearch(pop, @localSearchFunction, CostFunction,
VarMin, VarMax);
        end

        improvementRate = abs(prevBestCost - BestCost) / prevBestCost;
        if improvementRate < 0.01
            pop = applyLocalSearch(pop, @localSearchFunction, CostFunction,
VarMin, VarMax);
        end

        % Update Best Cost Record
        allBestCosts(it, run + 1) = BestCost;

        % Store positions for this iteration
        for i = 1:parameterSets(run).nPop
            particlePositions{run}(i, :, it) = pop(i).Position;
        end

             % Check for Global Minimum Achievement
           if abs(BestSol.Cost) <= tolerance
                    globalMinReached = true;
                    globalMinIter(run) = it;
                    if postGlobalMinIterations > 10
                           break; % Exit after 10 iterations post global minimum
            end
        end

        % Update prevBestCost for the next iteration
        prevBestCost = BestCost;

        disp(['Iteration ' num2str(it) ' in Run ' num2str(run) ': Best Cost = '
num2str(BestCost)]);
    end
end

%% Save Iteration Data
csvwrite('DE_AllRunsBestCosts.csv', allBestCosts);

%% Visualisation: Trajectory of Each Run's Journey up to Global Minimum
% New figure for the line graph
figure;
hold on;
colors = lines(length(parameterSets));

% Variable to store the minimum iteration for each parameter set
```

```
minIterations = zeros(length(parameterSets), 1);

for run = 1:length(parameterSets)
    params = parameterSets(run);
    itData = allBestCosts(:, run + 1);
    minIt = find(itData == min(itData), 1, 'first');
    minIterations(run) = minIt; % Store the minimum iteration
    plot(1:minIt, itData(1:minIt), 'LineWidth', 2, 'Color', colors(run, :));
end

legend(arrayfun(@(x) ['Set ' num2str(x)], 1:length(parameterSets),
'UniformOutput', false), 'Location', 'best');
xlabel('Iteration');
ylabel('Best Cost');
title('Best Cost Trajectory for Each Run');

hold off;

%% Create a 3D Surface Plot for Best Costs Over Iterations and Parameter Sets
% Determine the maximum number of iterations to display based on when global
minima were reached
maxDisplayIterations = max(globalMinIter);

% If no global minima were reached, use the maximum number of iterations for any
run
if maxDisplayIterations == 0
    maxDisplayIterations = maxIterations;
end

% Initialize the TruncatedBestCostsGrid with NaN values
TruncatedBestCostsGrid = NaN(length(parameterSets), maxDisplayIterations);

% Loop through each parameter set to fill in the best costs
for run = 1:length(parameterSets)
    if globalMinIter(run) > 0
        % Use only up to the iteration when the global minimum was reached
        TruncatedBestCostsGrid(run, 1:globalMinIter(run)) =
allBestCosts(1:globalMinIter(run), run + 1)';
    else
        % Use all iterations if the global minimum was not reached
        TruncatedBestCostsGrid(run, :) = allBestCosts(:, run + 1)';
    end
end

% Truncate the IterationGrid and ParameterSetGrid for display using the
maxDisplayIterations
[TruncatedIterationGrid, TruncatedParameterSetGrid] =
meshgrid(1:maxDisplayIterations, 1:length(parameterSets));

% Create the truncated surface plot
figure;
surf(TruncatedIterationGrid, TruncatedParameterSetGrid, TruncatedBestCostsGrid,
'EdgeColor', 'none');
xlabel('Iteration');
ylabel('Parameter Set');
zlabel('Best Cost');
title('Best Cost Surface Up to Global Minimum');
colorbar;
view(-45, 45);
```

```
colormap(jet);
legend(arrayfun(@(x) ['Set ' num2str(x) 'All Runs'], 1:length(parameterSets),
'UniformOutput', false), 'Location', 'best');

%% Contour Plot of Particle Journeys for a Selected Parameter Set
% Find the best solution from the final population of each run
bestPositions = zeros(length(parameterSets), nVar);
for run = 1:length(parameterSets)
    params = parameterSets(run);  % Use params specific for the current run
    finalPopCosts = arrayfun(@(i) CostFunction(particlePositions{run}(i, :, end)),
1:params.nPop);
    [~, idx] = min(finalPopCosts);  % Find the index of the best solution in the
final population
    bestPositions(run, :) = particlePositions{run}(idx, :, end);
end

% Create a meshgrid for the contour plot of the Rastrigin function
x = linspace(VarMin, VarMax, 100);
y = linspace(VarMin, VarMax, 100);
[X, Y] = meshgrid(x, y);
Z = arrayfun(@(x, y) CostFunction([x y]), X, Y);

%% Contour Plot:
figure;
% Reduce the number of contour lines to make the plot less busy
contourf(X, Y, Z, 20); % Adjust the number to get the desired granularity
hold on;
colormap('parula'); % Set a colormap that provides a clear background

% Plot the best solutions as red balls
% Adjust the MarkerSize if needed
for i = 1:size(bestPositions, 1)
    plot(bestPositions(i, 1), bestPositions(i, 2), 'ro', 'MarkerSize', 10, ...
        'MarkerFaceColor', 'r', 'MarkerEdgeColor', 'w'); % White edge for better
visibility
end

% Label the plot
xlabel('x_1');
ylabel('x_2');
title('Best Particle Positions Contour');
hold off;

%% Side By Side Iterations isualisation:
% Select iterations to display: early, mid, and late convergence stages
selectedIterations = [100, 250, 900]; % For example

% Create a new figure
figure;

% Define colors for each run
colors = lines(length(parameterSets));

% Plot for each run at different stages of convergence
for run = 1:length(parameterSets)
    % Extract MaxIt for the current run
    MaxIt = parameterSets(run).MaxIt;

    % Define stages based on the number of iterations of the current run
```

```
    earlyIter = max(10, round(MaxIt * 0.1)); % 10% into the run
    midIter = round(MaxIt / 2); % 50% into the run
    lateIter = MaxIt - 10; % Near the end of the run

    % Create subplots for early, mid, and late iterations
    for k = 1:3
        subplot(length(parameterSets), 3, (run-1)*3 + k);
        hold on;

        if k == 1
            iter = earlyIter;
        elseif k == 2
            iter = midIter;
        else
            iter = lateIter;
        end

        % Check if the iteration exists for the current run
        if iter <= MaxIt
            % Extract the positions for the current iteration of this run
            positions = reshape(particlePositions{run}(:,:,iter), [], nVar);
            scatter(positions(:,1), positions(:,2), 36, colors(run, :), 'filled');
        end

        % Formatting the plot
        title(sprintf('Run %d - Iteration %d', run, iter));
        xlabel('x_1');
        ylabel('x_2');
        xlim([VarMin, VarMax]);
        ylim([VarMin, VarMax]);
        hold off;
    end
end

% Add a super title to the entire figure
sgtitle('Particle Positions from Different Runs at Selected Iterations');

%% Diversity Over Time Visualisation:
% Preallocate an array to hold the diversity measure for each run and iteration
diversityOverTime = zeros(maxIterations, length(parameterSets));

% Loop over each parameter set
for run = 1:length(parameterSets)
    % Calculate the diversity for each iteration
    for it = 1:parameterSets(run).MaxIt
        % Get the particle positions for the current iteration and run
        positions = particlePositions{run}(:, :, it);

        % Calculate the standard deviation of the positions
        diversityOverTime(it, run) = std(positions(:));
    end
end

% Now, let's plot the diversity over time
figure;
hold on;
colors = lines(length(parameterSets));
for run = 1:length(parameterSets)
```

```
    plot(1:parameterSets(run).MaxIt, diversityOverTime(1:parameterSets(run).MaxIt,
run), 'Color', colors(run, :), 'LineWidth', 2);
end
hold off;

xlabel('Iteration');
ylabel('Diversity (Standard Deviation)');
title('Diversity of the Population Over Time');
legend(arrayfun(@(x) ['Run ' num2str(x)], 1:length(parameterSets),
'UniformOutput', false), 'Location', 'best');

%% Compare Diversity Vs BestCosts:
% Create a new figure
figure;

% Loop over each parameter set
for run = 1:length(parameterSets)
    subplot(length(parameterSets), 1, run);

    % Plot diversity on the left y-axis
    yyaxis left;
    plot(diversityOverTime(1:parameterSets(run).MaxIt, run), 'LineWidth', 2);
    ylabel('Diversity');

    % Plot best cost on the right y-axis
    yyaxis right;
    plot(allBestCosts(1:parameterSets(run).MaxIt, run + 1), 'LineWidth', 2); % +1
to skip the iteration count column
    ylabel('Best Cost');

    % Add labels and title
    xlabel('Iteration');
    title(sprintf('Run %d - Diversity and Best Cost Over Time', run));

    % Add legend
    legend('Diversity', 'Best Cost', 'Location', 'best');
end

% Add an overall title
sgtitle('Comparison of Diversity and Best Cost Over Time for Each Run');
```

de.m (DE/2)

```
%% DE /best/2/bin with Dynamic Parameter Testing incorporating a halt 10
iterations after achieving the global minimum within specified tolerance
clc; clear;

%% Problem Definition
CostFunction = @(x) Himmelblau_Constrained(x);  % Cost Function
nVar = 2;                           % Number of Decision Variables
VarSize = [1 nVar];                 % Decision Variables Matrix Size
VarMin = -15.12;                     % Lower Bound of Decision Variables
VarMax = 15.12;                      % Upper Bound of Decision Variables
tolerance = 1e-6;                   % Tolerance for considering the global minimum
achieved

immigrantFraction = 0.1;            % adjust as needed
prevBestCost = inf;                 % Initialise with a high value
restartFraction = 0.2;              % Restart 20% of the population
```

```
%% Parameter Sets as Struct Array
parameterSets = [
    struct('MaxIt', 1000, 'nPop', 150, 'beta_min', 0.4, 'beta_max', 0.8, 'pCR',
0.4, 'immigrantFraction', 0.1, 'rateIncrease', 0.01, 'maxBetaMin', 0.1,
'minBetaMax', 0.7);  % Set 1
    struct('MaxIt', 900, 'nPop', 200, 'beta_min', 0.2, 'beta_max', 0.8, 'pCR',
0.2, 'immigrantFraction', 0.05, 'rateIncrease', 0.05, 'maxBetaMin', 0.2,
'minBetaMax', 0.8); % Set 2
    struct('MaxIt', 800, 'nPop', 150, 'beta_min', 0.1, 'beta_max', 0.9, 'pCR',
0.3, 'immigrantFraction', 0.2, 'rateIncrease', 0.09, 'maxBetaMin', 0.3,
'minBetaMax', 0.9); % Set 3
    struct('MaxIt', 700, 'nPop', 150, 'beta_min', 0.5, 'beta_max', 0.8, 'pCR',
0.1, 'immigrantFraction', 0.15, 'rateIncrease', 0.1, 'maxBetaMin', 0.4,
'minBetaMax', 0.7);  % Set 4
    struct('MaxIt', 600, 'nPop', 200, 'beta_min', 0.3, 'beta_max', 0.7, 'pCR',
0.6, 'immigrantFraction', 0.25, 'rateIncrease', 0.15, 'maxBetaMin', 0.5,
'minBetaMax', 0.8);% Set 5
    struct('MaxIt', 500, 'nPop', 150, 'beta_min', 0.6, 'beta_max', 0.8, 'pCR',
0.5, 'immigrantFraction', 0.4, 'rateIncrease', 0.2, 'maxBetaMin', 0.6,
'minBetaMax', 0.9)   % Set 6
];

%% Initialise Results Storage
allBestCosts = zeros(max([parameterSets.MaxIt]), length(parameterSets) + 1); % +1
for iteration numbers
allBestCosts(:, 1) = 1:max([parameterSets.MaxIt]); % Column 1 is iteration numbers
globalMinIter = zeros(1, length(parameterSets)); % Initialise globalMinIter
maxIterations = max([parameterSets.MaxIt]);
particlePositions = cell(length(parameterSets), 1);

%% Main Loop Over Parameter Sets
for run = 1:length(parameterSets)
    params = parameterSets(run);
    adaptive_beta_min = params.beta_min;
    adaptive_beta_max = params.beta_max;
    previousBestCost = inf; % For tracking improvement
    particlePositions{run} = zeros(parameterSets(run).nPop, nVar,
parameterSets(run).MaxIt);

    % Initialise Population
    pop = struct('Position', [], 'Cost', inf(params.nPop, 1));
    for i = 1:params.nPop
        pop(i).Position = unifrnd(VarMin, VarMax, VarSize);
        pop(i).Cost = CostFunction(pop(i).Position);
    end
    [BestCost, index] = min([pop.Cost]);
    BestSol = pop(index);

    %% DE Main Loop
    for it = 1:params.MaxIt
             for i = 1:params.nPop
            % Choose five random individuals from the population, distinct from i
            candidates = randperm(params.nPop);
            candidates(candidates == i) = [];  % Exclude the target individual
            r = candidates(1:4);  % Select the first four for mutation

            % Mutation: DE/best/2/bin strategy
            % y = BestIndividual + F * (individual_r1 - individual_r2) + F *
(individual_r3 - individual_r4)
```

```
            y = BestSol.Position + params.beta_min * (pop(r(1)).Position -
pop(r(2)).Position) + ...
                params.beta_min * (pop(r(3)).Position - pop(r(4)).Position);
            y = max(y, VarMin); % Make sure solutions stay within bounds
            y = min(y, VarMax);

                % Crossover
                z = pop(i).Position;
                jRand = randi([1 numel(z)]);
                for j = 1:numel(z)
                    if rand <= params.pCR || j == jRand
                        z(j) = y(j);
                    end
                end

                % Selection
                NewSol.Position = z;
                NewSol.Cost = CostFunction(NewSol.Position);
                if NewSol.Cost < pop(i).Cost
                    pop(i) = NewSol;
                    if NewSol.Cost < BestSol.Cost
                        BestSol = NewSol;
                    end
                end
            end

        % Introduce Random Immigrants
        pop = introduceRandomImmigrants(pop, CostFunction, VarMin, VarMax,
VarSize, params);

        % Update BestSol and BestCost logic...
        [currentBestCost, index] = min([pop.Cost]);  % Find the current best cost
and index
        if currentBestCost < BestSol.Cost  % If the current best is better than
the overall best
            BestSol = pop(index);  % Update BestSol
            BestCost = currentBestCost;  % Update BestCost with the current
iteration best
        end

        % Use the updated BestCost for adaptMutationRates
        [adaptive_beta_min, adaptive_beta_max] =
adaptMutationRates(adaptive_beta_min, adaptive_beta_max, params, BestCost,
previousBestCost);

            pop = restartPopulation(pop, VarMin, VarMax, VarSize,
restartFraction, CostFunction);

            % Apply local search if required
        if mod(it, 50) == 0
            pop = applyLocalSearch(pop, @localSearchFunction, CostFunction,
VarMin, VarMax);
        end

        improvementRate = abs(prevBestCost - BestCost) / prevBestCost;
        if improvementRate < 0.01
            pop = applyLocalSearch(pop, @localSearchFunction, CostFunction,
VarMin, VarMax);
        end
```

```
        % Update Best Cost Record
        allBestCosts(it, run + 1) = BestCost;

        % Store positions for this iteration
        for i = 1:parameterSets(run).nPop
            particlePositions{run}(i, :, it) = pop(i).Position;
        end

             % Check for Global Minimum Achievement
           if abs(BestSol.Cost) <= tolerance
                    globalMinReached = true;
                    globalMinIter(run) = it;
                    if postGlobalMinIterations > 10
                           break; % Exit after 10 iterations post global minimum
             end
        end

        % Update prevBestCost for the next iteration
        prevBestCost = BestCost;

        disp(['Iteration ' num2str(it) ' in Run ' num2str(run) ': Best Cost = '
num2str(BestCost)]);
    end
end

%% Save Iteration Data
csvwrite('DE_AllRunsBestCosts.csv', allBestCosts);

%% Visualisation: Trajectory of Each Run's Journey up to Global Minimum
% New figure for the line graph
figure;
hold on;
colors = lines(length(parameterSets));

% Variable to store the minimum iteration for each parameter set
minIterations = zeros(length(parameterSets), 1);

for run = 1:length(parameterSets)
    params = parameterSets(run);
    itData = allBestCosts(:, run + 1);
    minIt = find(itData == min(itData), 1, 'first');
    minIterations(run) = minIt; % Store the minimum iteration
    plot(1:minIt, itData(1:minIt), 'LineWidth', 2, 'Color', colors(run, :));
end

legend(arrayfun(@(x) ['Set ' num2str(x)], 1:length(parameterSets),
'UniformOutput', false), 'Location', 'best');
xlabel('Iteration');
ylabel('Best Cost');
title('Best Cost Trajectory for Each Run');

hold off;

%% Create a 3D Surface Plot for Best Costs Over Iterations and Parameter Sets
% Determine the maximum number of iterations to display based on when global
minima were reached
maxDisplayIterations = max(globalMinIter);
```

```
% If no global minima were reached, use the maximum number of iterations for any
run
if maxDisplayIterations == 0
    maxDisplayIterations = maxIterations;
end

% Initialise the TruncatedBestCostsGrid with NaN values
TruncatedBestCostsGrid = NaN(length(parameterSets), maxDisplayIterations);

% Loop through each parameter set to fill in the best costs
for run = 1:length(parameterSets)
    if globalMinIter(run) > 0
        % Use only up to the iteration when the global minimum was reached
        TruncatedBestCostsGrid(run, 1:globalMinIter(run)) =
allBestCosts(1:globalMinIter(run), run + 1)';
    else
        % Use all iterations if the global minimum was not reached
        TruncatedBestCostsGrid(run, :) = allBestCosts(:, run + 1)';
    end
end

% Truncate the IterationGrid and ParameterSetGrid for display using the
maxDisplayIterations
[TruncatedIterationGrid, TruncatedParameterSetGrid] =
meshgrid(1:maxDisplayIterations, 1:length(parameterSets));

% Create the truncated surface plot
figure;
surf(TruncatedIterationGrid, TruncatedParameterSetGrid, TruncatedBestCostsGrid,
'EdgeColor', 'none');
xlabel('Iteration');
ylabel('Parameter Set');
zlabel('Best Cost');
title('Best Cost Surface Up to Global Minimum');
colorbar;
view(-45, 45);
colormap(jet);
legend(arrayfun(@(x) ['Set ' num2str(x) 'All Runs'], 1:length(parameterSets),
'UniformOutput', false), 'Location', 'best');

%% Contour Plot of Particle Journeys for a Selected Parameter Set
% Find the best solution from the final population of each run
bestPositions = zeros(length(parameterSets), nVar);
for run = 1:length(parameterSets)
    params = parameterSets(run);  % Use params specific for the current run
    finalPopCosts = arrayfun(@(i) CostFunction(particlePositions{run}(i, :, end)),
1:params.nPop);
    [~, idx] = min(finalPopCosts);  % Find the index of the best solution in the
final population
    bestPositions(run, :) = particlePositions{run}(idx, :, end);
end

% Create a meshgrid for the contour plot of the Rastrigin function
x = linspace(VarMin, VarMax, 100);
y = linspace(VarMin, VarMax, 100);
[X, Y] = meshgrid(x, y);
Z = arrayfun(@(x, y) CostFunction([x y]), X, Y);

%% Contour Plot:
```

```
figure;
% Reduce the number of contour lines to make the plot less busy
contourf(X, Y, Z, 20); % Adjust the number to get the desired granularity
hold on;
colormap('parula'); % Set a colormap that provides a clear background

% Plot the best solutions as red balls
% Adjust the MarkerSize if needed
for i = 1:size(bestPositions, 1)
    plot(bestPositions(i, 1), bestPositions(i, 2), 'ro', 'MarkerSize', 10, ...
        'MarkerFaceColor', 'r', 'MarkerEdgeColor', 'w'); % White edge for better
visibility
end

% Label the plot
xlabel('x_1');
ylabel('x_2');
title('Best Particle Positions Contour');
hold off;

%% Side By Side Iterations isualisation:
% Select iterations to display: early, mid, and late convergence stages
selectedIterations = [100, 250, 900]; % For example

% Create a new figure
figure;

% Define colors for each run
colors = lines(length(parameterSets));

% Plot for each run at different stages of convergence
for run = 1:length(parameterSets)
    % Extract MaxIt for the current run
    MaxIt = parameterSets(run).MaxIt;

    % Define stages based on the number of iterations of the current run
    earlyIter = max(10, round(MaxIt * 0.1)); % 10% into the run
    midIter = round(MaxIt / 2); % 50% into the run
    lateIter = MaxIt - 10; % Near the end of the run

    % Create subplots for early, mid, and late iterations
    for k = 1:3
        subplot(length(parameterSets), 3, (run-1)*3 + k);
        hold on;

        if k == 1
            iter = earlyIter;
        elseif k == 2
            iter = midIter;
        else
            iter = lateIter;
        end

        % Check if the iteration exists for the current run
        if iter <= MaxIt
            % Extract the positions for the current iteration of this run
            positions = reshape(particlePositions{run}(:,:,iter), [], nVar);
            scatter(positions(:,1), positions(:,2), 36, colors(run, :), 'filled');
        end
```

```
        % Formatting the plot
        title(sprintf('Run %d - Iteration %d', run, iter));
        xlabel('x_1');
        ylabel('x_2');
        xlim([VarMin, VarMax]);
        ylim([VarMin, VarMax]);
        hold off;
    end
end

% Add a super title to the entire figure
sgtitle('Particle Positions from Different Runs at Selected Iterations');

%% Diversity Over Time Visualisation:
% Preallocate an array to hold the diversity measure for each run and iteration
diversityOverTime = zeros(maxIterations, length(parameterSets));

% Loop over each parameter set
for run = 1:length(parameterSets)
    % Calculate the diversity for each iteration
    for it = 1:parameterSets(run).MaxIt
        % Get the particle positions for the current iteration and run
        positions = particlePositions{run}(:, :, it);

        % Calculate the standard deviation of the positions
        diversityOverTime(it, run) = std(positions(:));
    end
end

% Now, let's plot the diversity over time
figure;
hold on;
colors = lines(length(parameterSets));
for run = 1:length(parameterSets)
    plot(1:parameterSets(run).MaxIt, diversityOverTime(1:parameterSets(run).MaxIt,
run), 'Color', colors(run, :), 'LineWidth', 2);
end
hold off;

xlabel('Iteration');
ylabel('Diversity (Standard Deviation)');
title('Diversity of the Population Over Time');
legend(arrayfun(@(x) ['Run ' num2str(x)], 1:length(parameterSets),
'UniformOutput', false), 'Location', 'best');

%% Compare Diversity Vs BestCosts:
% Create a new figure
figure;

% Loop over each parameter set
for run = 1:length(parameterSets)
    subplot(length(parameterSets), 1, run);

    % Plot diversity on the left y-axis
    yyaxis left;
    plot(diversityOverTime(1:parameterSets(run).MaxIt, run), 'LineWidth', 2);
    ylabel('Diversity');
```

```
    % Plot best cost on the right y-axis
    yyaxis right;
    plot(allBestCosts(1:parameterSets(run).MaxIt, run + 1), 'LineWidth', 2); % +1
to skip the iteration count column
    ylabel('Best Cost');

    % Add labels and title
    xlabel('Iteration');
    title(sprintf('Run %d - Diversity and Best Cost Over Time', run));

    % Add legend
    legend('Diversity', 'Best Cost', 'Location', 'best');
end

% Add an overall title
sgtitle('Comparison of Diversity and Best Cost Over Time for Each Run');
```

adaptMutationRates.m

```
function [new_beta_min, new_beta_max] = adaptMutationRates(beta_min, beta_max,
params, currentBestCost, previousBestCost)
    if currentBestCost < previousBestCost % improvement is seen as a decrease in
cost
        % Increase beta_min within bounds
        new_beta_min = min(beta_min + params.rateIncrease, params.maxBetaMin);
        % Decrease beta_max but ensure it doesn't go below its minimum allowed
value
        new_beta_max = max(beta_max - params.rateIncrease, params.minBetaMax);
    else
        % If no improvement, keep the rates the same
        new_beta_min = beta_min;
        new_beta_max = beta_max;
    end
end
```

applyLocalSearch.m

```
function pop = applyLocalSearch(pop, localSearchFunction, CostFunction, VarMin,
VarMax)
    % Iterate over the population and apply local search
    for i = 1:length(pop)
        % Apply the local search function to find a new position within bounds
        new_position = localSearchFunction(pop(i).Position, VarMin, VarMax,
CostFunction);

        % Update the individual's position and cost with the new values
        pop(i).Position = new_position;
        pop(i).Cost = CostFunction(new_position);
    end
end
```

introduceRandomImmigrants.m

```
function pop = introduceRandomImmigrants(pop, CostFunction, VarMin, VarMax,
VarSize, params)
    nImmigrants = round(params.immigrantFraction * length(pop)); % Adjusted to use
length(pop) for generality
    for i = 1:nImmigrants
        idx = randi([1, length(pop)]);
        newImmigrant.Position = unifrnd(VarMin, VarMax, VarSize);
        newImmigrant.Cost = CostFunction(newImmigrant.Position);
        pop(idx) = newImmigrant;
    end
```

```
end
```

localSearchFunction.m

```
function newPosition = localSearchFunction(currentPosition, VarMin, VarMax,
CostFunction)
    % Generate random perturbation (Gaussian noise)
    perturbation = randn(size(currentPosition)); % Gaussian noise

    % Add perturbation to the current position
    newPosition = currentPosition + perturbation;

    % Ensure newPosition stays within bounds
    newPosition = max(newPosition, VarMin);
    newPosition = min(newPosition, VarMax);
end
```

restartPopulation.m

```
function pop = restartPopulation(pop, VarMin, VarMax, VarSize, restartFraction,
CostFunction)
    numRestart = round(length(pop) * restartFraction);
    for i = 1:numRestart
        pop(end+1-i).Position = unifrnd(VarMin, VarMax, VarSize); % Correct usage
of VarSize
        pop(end+1-i).Cost = CostFunction(pop(end+1-i).Position);
    end
end
```

Other Code

Welded Beam Function Code

```
function [c, ceq] = constraint_welded_beam(x)
    % Decompose the decision variables
    h = x(1); % Thickness of the weld
    l = x(2); % Length of the attached part of the beam
    t = x(3); % Thickness of the beam
    b = x(4); % Width of the beam

    % Constants and Parameters
    P = 6000; % Load in lbs
    L = 14; % Length of the beam in inches
    E = 30e6; % Modulus of elasticity in psi
    G = 12e6; % Shear modulus in psi

    % Derived Calculations
    Pc = (4.013 * E * sqrt((t^2 * b^6) / 36)) / (L^2) * (1 - t * L / (2 * pi) *
sqrt(E / (4 * G)));
    M = P * (L + (l / 2));
    R = sqrt((l^2 / 4) + ((h + t)^2 / 4));
    J = 2 * (sqrt(2) * h * l * ((l^2 / 12) + ((h + t)^2 / 4)));

    % Calculating tau (shear stress), sigma (bending stress), delta (deflection)
    tau = (P / (sqrt(2) * h * l)) + ((M * R) / J);
    sigma = (6 * P * L) / (b * t^2);
    delta = (4 * P * L^3) / (E * b * t^3);

    % Constraints
    tau_max = 13600; % Maximum shear stress in psi
```

```
    sigma_max = 30000; % Maximum bending stress in psi
    delta_max = 0.25; % Maximum deflection in inches
    c = [tau - tau_max; sigma - sigma_max; Pc - P; delta - delta_max];

    % No equality constraints in this problem
    ceq = [];
end
```

G06 Code

```
function [f, c, ceq] = g06(x)
    % g06 Function for Constrained Optimisation
    % This function represents a nonlinear constrained optimisation problem.
    % It has two decision variables, two inequality constraints, and no equality
constraints.

    % Objective function
    % The function to be minimised.
    f = (x(1) - 10)^3 + (x(2) - 20)^3;

    % Inequality constraints (c <= 0)
    % These are the constraints that the solution must satisfy.
    c = [(x(1) - 5)^2 + (x(2) - 5)^2 - 100, ...
         -((x(1) - 6)^2 + (x(2) - 5)^2 - 82.81)];

    % Equality constraints (ceq = 0)
    % There are no equality constraints for this problem.
    ceq = [];
end
```

G08 Code

```
function f = g08(x)
    % g08 Function for Unconstrained Optimisation
    % This function represents an unconstrained optimisation problem.
    % It has two decision variables and is to be maximised. However, MATLAB
    % optimisers typically minimise functions, so we return the negative.

    % Constants
    % The constants used in the function's formula.
    p = 10;
    q = 15;
    r = 0.1;

    % Objective function
    % The function to be maximised. Negate it for minimisation in MATLAB.
    f = -((sin(2*pi*x(1)))^(3) * sin(2*pi*x(2))) / ...
        ((x(1)^3)*(x(1) + x(2)));

    % Optimisation problems are set up to minimise the objective function.
    % To maximise, you can minimise the negative of the objective function.
end
```

Constrained Rosenbrock

```
function f = rosenbrock_with_constraints(x)
    % Define box constraints
    % Set constraints such that:
    % -2 <= x(1) <= 2
    % -3 <= x(2) <= 3
```

```
    % Rosenbrock function
    f = 100 * (x(2) - x(1)^2)^2 + (1 - x(1))^2;

    % Define box constraints
    % Set constraints such that:
    % -2 <= x(1) <= 2
    % -3 <= x(2) <= 3
    constraint_x0 = [-2, 2];
    constraint_x1 = [-3, 3];

    % Penalise violations of box constraints
    penalty = 0;
    if x(1) < constraint_x0(1) || x(1) > constraint_x0(2)
        penalty = penalty + 1e6 * (max(0, constraint_x0(1) - x(1))^2 + max(0, x(1)
- constraint_x0(2))^2);
    end
    if x(2) < constraint_x1(1) || x(2) > constraint_x1(2)
        penalty = penalty + 1e6 * (max(0, constraint_x1(1) - x(2))^2 + max(0, x(2)
- constraint_x1(2))^2);
    end

    % Add penalty to the objective function
    f = f + penalty;
end
```