Evolutionary Computing: A Comprehensive Analysis

Name: **Babu Pallam** Pnumber**: P2849288**
Block 3: **CSIP5304 - Fuzzy Logic & Evolutionary Computing** Submitted to: **Dr Zacharias Anastassi**

1. Introduction

Evolutionary Computing (EC) is one of the extensive fields in Artificial Intelligence. This comprises of different algorithms or paradigms for optimizing complex problems which are impossible to conclude using traditional approach. These algorithms include Genetic Algorithm(GA), Particle Swarm Optimization(PSO), Differential Evolution(DE), Ant Colony Optimization(ACO), and others... This report presents an approach of applying the three evolutionary algorithms such as Real GA, PSO, and DE, for solving unconstrained and constrained optimization problems, using mathematical performance testing functions such as RosenBrock and Himmelblau functions.

Next section provides a literature review, including the how this branch of Computer Science has been developed, and what are the main algorithms which involve in these domains for solving problems of different types. In the next section, the description of the problem and implementation details has been discussed prior to the performance testing. Followed by this, section 4 presents the investigation done through several performance testing that by changing the parameter values, operators, and implementing different variants of same technique. Demonstrating the best approach to get the optimum solution has been done under this section. In section 5, “Interpretations and Analysis”, various Observations Drawn made has been discussed. Followed by, as section 6, the areas where further research is needed have been presented which may need deep research to extend the performance of algorithms. Finally, concluded in section 7.

2. Literature Review

. This section starts with an introduction to evolutionary computing. Followed by that the mathematical functions for optimization has been discussed. Then the tools which are available nowadays to handle evolutionary computing algorithms has been described.

2.1. Evolutionary Computing

Evolution process in biology has been always an inspiration and the researchers have always shown the curiosity to comprise that process into a structure. It started from 1950’s with the discoveries of different evolutionary algorithms and then those has been collectively introduced under Evolutonary Computation in 1994 at Proceedings of the First IEEE Conference on Evolutionary Computation (Electronics, 1994). Evolutionary Computing (EC) is the term which is being used popular which comes under Computational Intelligence (Duch, 2007). Computational Intelligence is the study of development of computational strategies to solve the real-world problems. Artificial Neural Network and Fuzzy Logic Systems are some of those. EC includes various algorithms such as Genetic Algorithms,

Differential Evolution, Particle Swarm Optimization, Ant Colony Optimization (Dorigo, 2006) and so on. This Among those three algorithms which have been considered in this report for a comprehensive analysis are presented in the following subsections.

2.1.1 Genetic Algorithm

Genetic Algorithm (GA) has been introduced by John Holland in his paper named “Outline for a logical theory of adaptive systems” (Holland, 1962) in 1962. He introduced a framework in this paper which imitate the evolution process for solving complex optimization and search problems under diverse of domains.

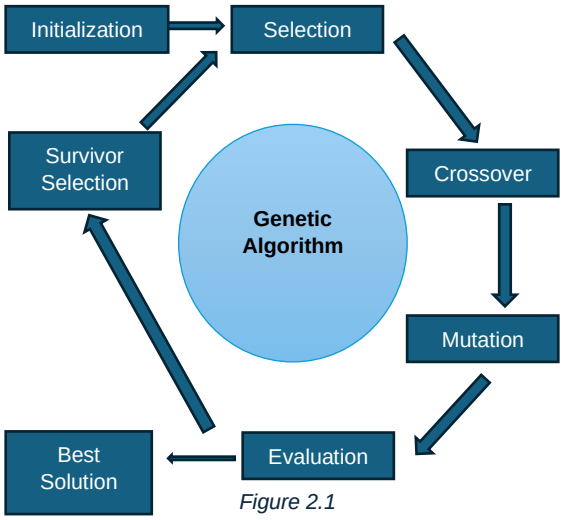


Figure 2.1

Figure 2.1 depicts the steps involved in genetic algorithm. It begins with initialization, then selection process; followed by crossover, and finally mutation before evaluation. After the evaluation, either best solution can be presented or select the survivors to continue next generation process.

2.1.2 Particle Swarm Optimization

Particle Swarm Optimization (PSO) was introduced by James Kennedy and Russell Eberhart in 1995 (Eberhart, 1995). This algorithm takes inspiration from how the flock of swarms move in the space in search of food source. Here, each swarm (particle) is considered as a potential solution to the optimization problem which needed to be solved.

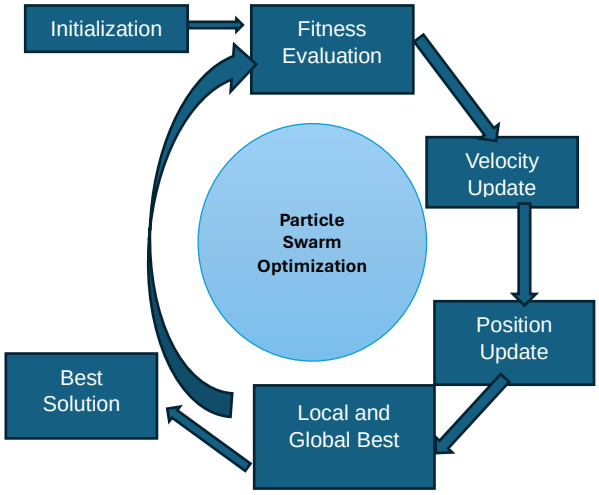


Figure 2.2

Figure 2.2 depicts how the PSO works. Initialization phase involves the initialization of population of particle. Followed by that find the solution each particle in “Fitness Evaluation”. Then at “Velocity Update”, update the velocity of each particle using momentum part, cognitive part, and social part. After updating velocity of each particle, find new position of each particle by adding the updated velocity with the current position vector. Update local best and global best of the solution. If the convergence to the optimum solution happened, take the “Best Solution”, otherwise repeat the generation until it gets its convergence point.

2.1.3 Differential Evolution

Differential Evolution (DE) was introduced by Rainer Stone and others in 1997 (Storn, 997). It is like GA, but mutation would be performed before crossover and the solutions are considered as vectors of real numbers. The evolution process using DE has been described in Figure 2.3.

It begins with Initialization of the random population space. Followed by that, mutation operation performed. In which, for each solution, referred as target vector, a mutation vector is calculated using base vector and vector differences depends upon the variant of DE. Then using the selected solution and mutation vector, trail vector is being created, which is called crossover operation, with the help of target vector and mutation vector. Finally, selection phase, in which compare trail and target vectors, and choose the best which replaces the target vector. Based on criteria, best solution can be presented, otherwise perform next generation until reaches the convergence to the best solution.

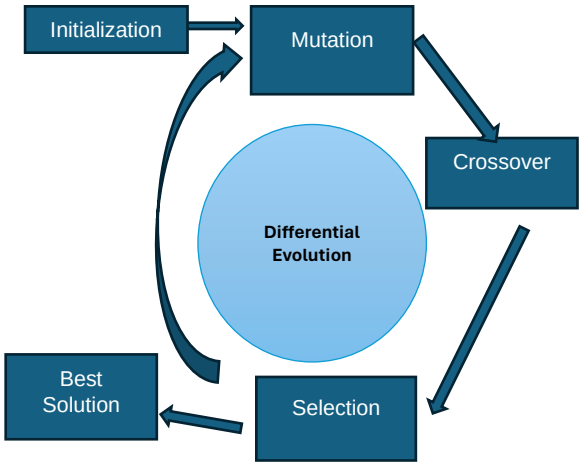


Figure 2.3

This report presents how these three algorithms workNext section discusses the importance of mathematical functions for optimization of evolutionary algorithms.

2.2. Mathematical Optimization Functions

Mathematical optimization functions are used in many fields of computational Intelligence. The importance of these functions are in implementing the algorithms effectively with almost efficiency and optimization, using computational tools; in which these functions can be replaced by real world problem scenarios while implementing it. The importance domains where these functions are commonly used in problem solving, efficiency checking, process improvement, risk management in specific domains and others. Depends upon the on problems, the functions type would vary, including linear functions, quadratic functions, convex functions, and others.

Among several test functions available, two renown functions have been taken into consideration for the performance analysis of the three algorithms chosen are Rosenbrock function (Rosenbrock, 1960) and Himmelblau function (Himmelblau, 2018). The function description are provided in section 3 under section 3.1 – Problem descriptions.

2.3. Tools for EC

There are several frameworks on different programming languages are available today which are widely used in academic research and Industries. Python based PyGMO (Biscani., 2015), and Optunity (Marc Claesen, 2014), Java based MOEA (others, 2009) framework are some of those well-known available today.

This report approaches the optimization of algorithms using MATLAB, which is a framework and a language developed by MathWorks (MATWORKS, Since 1994). Using MATLAB, the implementation of algorithms and performance testing with the help of different graphs and plot features of MATLAB have been carried out. The codes and graphs resulted have been provided in the Appendices.

3. Problem Descriptions and Implementation Details

3.1 Problem Descriptions

The problem definitions of unconstrained and constrained problems are provided in the first two sections, and the tolerance value which should be set has been discussed in the third section.

3.1.1 Problem Description for Unconstrained Problem

For testing of the selected optimization algorithms, **Rosenbrock function** has been taken into consideration. This objective function is considered as the problem which should be solved using the selected algorithms where number of variable equals to 2. The function implemented is defined below.

Minimise f (x_1, x_2) = 100*(x_2-x_1^2)^2 + (1-x_1)^2,

Bounds: -10 <= x_1 <= 10 and -10 <= x_2 <= 10

The optimum solution of this function is zero, which is defined as f (1,1) = 0.

The values associated with Problem, here Rosenbrock function are same with all the algorithms. Those are as given below.

Problem associated values

- Number of variables (nVar) = 2
- Variable bound:
- Minimum bound(VarMin) = -5
- Maximum bound(VarMax) = 5
- ToleranceValue = 10^-2

3.1.2 Problem Description for Constrained Problem

The problem which has taken into consideration here to optimize the selected three algorithms is minimization of the **Himmelblau Function** with two variables, which is defined as follows,

$$f(x,y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2$$

Under minimization, four local minima are available for this problem, which are as listed below.

1. $f(3,2) = 0$
2. $f(-3.779310, -3.283186) = 0$
3. $f(-2.805118, 3.283186) = 0$
4. $f(3.584458, -1.848126)) = 0$

Adding Constraints

These above solutions are associated with unconstrained evaluation of the Himmelblau function. To make it a constraint problem**,** the following conditions has been added to this problem. The conditions are as given below.

$Constraint\ 1: (x_1 - 5)^2 + x_2^2 \geq 26$

$Constraint\ 2: (4x_1 + x_2) \geq 2$

$Constraint\ 3: x_1, x_2 \geq 0$

The optimum solution to the Himmelblau function subjected to the above three constraints is as given below.

$x^* = (3.763, 4.947) \implies f(x^*) = \mathbf{517.063}$

Adding Static Penalty

To enforce the conditions/ constraints to the objective function, static penalty has been implemented to penalize the solutions so that the solutions moves close into the feasible region when it violates the subjected conditions of the objective function. In this research, a penalty parameter has been considered, referred as R which used with the constraints to make it reflected into the result of objective function.

The Total cost or Result (referred as FitnessValue) of the objective function is as given below.

$F(x, y) = f(x, y) + R * (P_A(x, y) + P_B(x, y))$

In which,

f(x, y) : the Himmelblau function

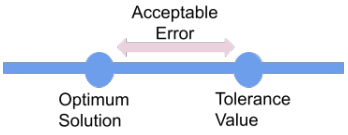
R : Penalty parameter

$P_{c1}(x, y)$, $P_{c2}(x, y)$: the penalty terms for constraints, C1 and C2.

3.1.3 Terminologies used in this Report

3.1.3.1 Tolerance Value

The problems mentioned in the above two sections are hard to converge into its optimized solution within a certain number of generations. Thereby, a value has been set for each problem, referred to as Tolerance Value. **This value is considered as the minimum acceptable solution to a problem during the evolution process with maximum possible error** (See img 1)**.** In minimization problem, a tolerance value must be set for which is greater than the actual output since the possibility of convergence is high if tolerance value is higher. The given table provides the tolerance values set over actual values during the optimization of algorithms with the acceptable error chosen.



Img 1

Minimization Problem	**Actual Value (Cost)**	**Tolerance Value**	**Acceptable Error**
Unconstrained Rosenbrock Function	**0**	**10^{-2}**	10^{-2} - 0 **= 10^{-2}**
Constrained HimmelBlau Function	**517.063**	**520**	520-517.063 **= 2.937**

3.1.3.2 Median Value

The term “Median” is defined as the value from a list that is sorted from smallest to highest, which is the middle value from the list if the number of items in the list is odd, and average of the two middle elements from the list if the number of items in the list is even. The “median” term had been used in this report more often. Which indicates the theoretical optimum of the minimum number of iterations needed to reach this tolerance value given, from the results obtained after several successful execution of the code.

3.2 Implementation Details

This section provides the values which has been set to different parameters, which is called as parameter specification, of algorithms for performance testing. Though, the problem definition is different for both, except the tolerance value, rest of the parameter are common and same in algorithms while handling both constrained and unconstrained problem.

3.2.1 Implementation Details of RGA

Unconstrained Real-Coded Genetic Algorithm (UnConRGA) and Constrained Real-Coded Genetic Algorithm (ConRGA) uses same parameter specification.

Specification For Benchmark A:

Algorithm associated values

- Maximum iteration(MaxIt) = 500;
- Number of population(nPop) = 40;
- Crossover probability(pC) = 1;
- Spread factor of SBX crossover(gamma) = 15;
- Mutation probability(mu) = 0.5;
- Distribution index for Polynomial mutation(sigma) = 20

The operators implemented in each phase of the RGA

- Selection Operation: **Binary Tournament Selection**
- Crossover Operation: **SBX Crossover**
- Mutation Operation: **Polynomial Mutation**
- Survivor Selection Operation: **Mu + Lambda Survivor Selection**

Specification for Benchmark B

Algorithm associated values

- Maximum iteration(MaxIt) = 500;
- Number of population(nPop) = 40;
- Crossover probability(pC) = 1;
- Probability coefficient for Selection operator(beta) = 1 (used in Benchmark UnConRGA-B)
- Mutation probability(mu) = 0.5;

The operators implemented in each phase of the RGA

- Selection Operation: **Stochastic Universal Sampling Selection**
- Crossover Operation: **Linear Crossover**
- Mutation Operation: **Random Mutation**
- Survivor Selection Operation: (**Mu, Lambda) Survivor Selection**

Note that Benchmark A for UnConRGA is known as **UnConRGA-A** and for ConRGA is known as **ConRGA-A**.

3.2.2 Implementation Details of PSO

Unconstrained Particle Swam Optimization Algorithm (UnConPSO) and Constrained Particle Swam Optimization Algorithm (ConPSO) uses same parameter specification.

Specification for Benchmark A

Algorithm associated values

- Maximum iteration(MaxIt) = 200;
- Number of population(nPop) = 40;
- Inertia Coefficient (w) = 1
- Damping Ratio of Inertia Coefficient(wdamp) = 0.99
- Personal Acceleration Coefficient(c1) = 1.5
- Social Acceleration Coefficient(c2) = 1.5

- Note that c1 and c2 values range has been considered as in between 0 to 2.

Specification for Benchmark B

This comprises the extension of the algorithm by giving dynamicity to w and wdamp by implementing with the introduction of contraction coefficients (see Appendix 8 8.A).

Note that Benchmark A for UnConPSO is known as **UnConPSO-A** and for ConPSO is known as **ConPSO-A**.

3.2.3 Implementation Details of DE

Unconstrained Differential Evolution (UnConDE) and Constrained Differential Evolution (ConDE) uses same parameter specification.

Specification for Benchmark A

Implemented Variant of DE : DE /rand/1/bin

In which,

- rand - random base vector
- 1 - one vector differences in production of mutate vector
- bin - DE uses binomial distribution, which uses uniform crossover for crossover operation.

Algorithm associated values

- Maximum iteration(MaxIt) = 1000;
- Number of population(nPop) = 40;
- Lower Bound of Scaling Factor(beta_min) = 0.2
- Upper Bound of Scaling Factor(beta_max) = 0.8
- Crossover Probability(pCR) = 0.5

Specification for Benchmark B

Implemented Variants of DE :

- **DE /best/1/bin**
  In which,
  - best – best solution in the previous generation
  - 1 - one vector differences in production of mutate vector
  - bin - DE uses binomial distribution
- **DE /best/2/bin**
  In which,
  - best – best solution in the previous generation
  - 2 - two vector differences in production of mutate vector
  - bin - DE uses binomial distribution

Note that Benchmark A for UnConDE is known as **UnConDE-A** and for ConDE is known as **ConDE-A**.

4. Investigations and Observations

The results obtained during the performance testing has been discussed here. How each algorithm tried to reach the optimum solution on different parameter combinations have been provided based on the code implemented with different benchmarks mentioned in the section 3. For each implementation, the outcomes obtained in performance analysis has been given in different subsections below. The result of each test is the number of iterations needed to reach the acceptable tolerance value.

4.1 Unconstrained RGA (UnConRGA)

The result obtained while checking the effect of parameters and effect of operators has in finding the optimum solution for Rosenbrock function are presented in this section

4.1.1. Benchmark UnConRGA-A

To see the progress of convergence of the solution towards the best solution, contour plots have been used. (see Appendix 7 7.A.1). In first generation, the solutions are populated over entire population space. As generations go, the convergence has been seen. At gen 200, the best solution of that generation reached the tolerance value where almost all solutions were converged close to the optimum solution.

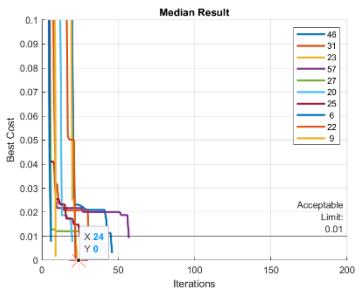


Figure 4.1.1.1

The Median (See section 3.1.3.2) was found to be 24 (see Figure 4.1.1.1). The performance result is obtained based on the specification given in section 3.2.1, has been provided in the table (*see Appendix 8 8.B.1*). The observations found has been concluded below.

- For larger tolerance values the result gives better, i.e. lesser number of generations are required to reach the tolerance value given.

- Variable bound influence the result, as the bound of variable get narrowed, lesser number of generations are required to get the desired output.
- mu = 0, means no mutation at all, since this doesn't make sense, we take mu = 0.3 value as a tuned value of mu from the testing performed.

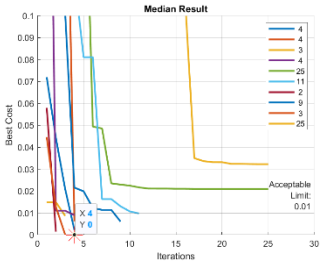


Figure 4.1.1.3

The median has been calculated based on the tuned parameter values, *(see Appendix 8 8.B.1*), which resulted to 4 (see Figure 4.1.1.3).

4.1.2. Benchmark UnConRGA-B

The contour plots has been made with this new implementation after replacing the operators (See in Appendix 7 7.A.2). The median found with this implementation was 118. (see Figure 4.1.2.1);

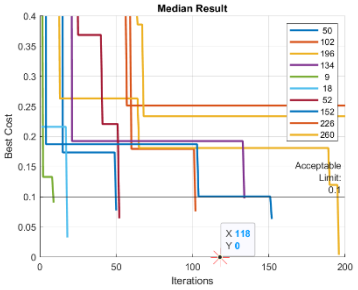


Figure 4.1.2.1

The performance result is obtained with the specification given in section 3.2.1 Benchmark B is given as a table (see Appendix 8 8.B.2). The main observations drawn have been noted below.

- The effects of parameters are significant.
- All the associated parameters, such as MaxIt, nPop, pC, and mu, shows same behaviour pattern.
- Beta parameter used in selection operator doesn't have any direct influence on the result.

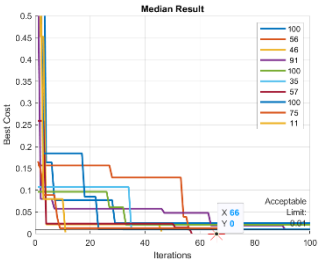


Figure 4.1.2.2

The median has been calculated based on the tuned parameter values, which resulted to 66 (see Figure 4.1.2.2)

4.2 Constrained RGA (ConRGA)

4.2.1 Benchmark ConRGA-A

The contour plots has been plotted to see the convergence pattern with the constraints. (See Appendix 7 7.B.1). The shaded region int the plots are infeasible regions. At gen 1, the population has been widely populated in both regions. And as generations go, the convergence to a single point has been noticed. At 123rd generation, the convergence to the acceptable value(which is considered as tolerance value in this report) has been seen when x =3.7646 and y=4.9472. Median value has been calculated, which was 46 (see Figure 4.2.1.1). The performance analysis done to see the effect of parameters has been given as a table with associated outputs.

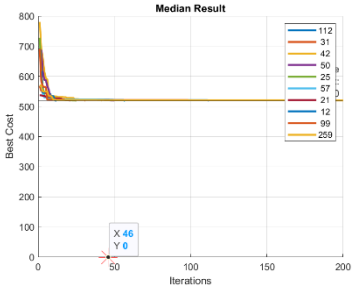


Figure 4.2.1.1

- When tolerance value is high then convergence would happen so fast.
- For least value of mu it gives a better result. But for higher number of nPop value, even with mu =1, gives a better

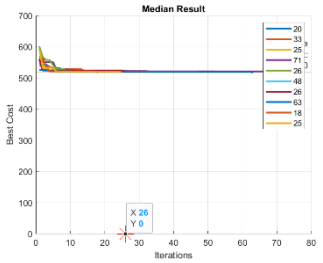


Figure 4.2.1.2

result, for example, nPop=400, and mu=1, with rest of the parameter according to base specification, resulted 5.

- Higher pC gives good result.

The median has been calculated based on the tuned parameter values, which resulted to 26 (see Figure 4.1.2.2).

4.2.2 Benchmark ConRGA-B

The contour plots has been plotted to see the convergence pattern with the constraints. (See Appendix 7 7.B.2). The convergence found at 1255th generation. The median found was 2805.5, approximated to 2806 generations, (see Figure 4.2.2.1);

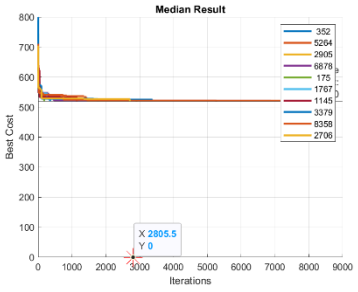


Figure 4.2.2.1

Like done in the above section, performance analysis has been performed and provided as a table (*see Appendix 8 8.B.4*). The main observations are given below.

- Effect of operators are like the unconstraint RGA.
- It was worth to note that how mu value effects the results. If mu is around 0.5, the result found to be minimum and when it's moving away from 0.5, the results increases.

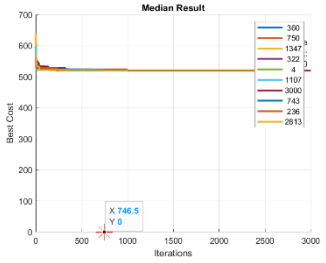


Figure 4.2.2.2

The median has been calculated based on the tuned parameter values, which resulted to 746.5, approximated to 755 (see Figure 4.2.2.2)

4.3 Unconstrained PSO (UnConPSO)

4.3.1. Benchmark UnConPSO-A

To see the progress of convergence of the solution towards the best solution, contour plots have been used (see appendix 7 7.C.1). Convergence to the tolerance value has been occurred at 28th generation. The median value found was 17 (see Figure 4.3.1.1).

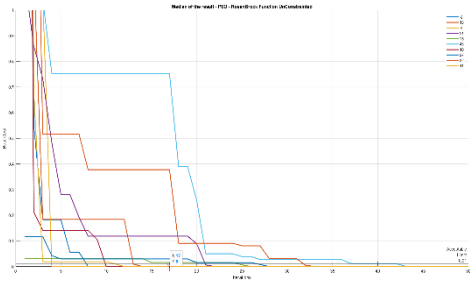


Figure 4.3.1.1

The performance result obtained has been provided as a table (*see Appendix 8 8.B.5*). The main observations found are noted below.

- For positive value of c1 and negative values of c2, the result found to be significantly large.

The median has been calculated based on the tuned parameter values, which resulted to 4 (see Figure 4.1.1.3)

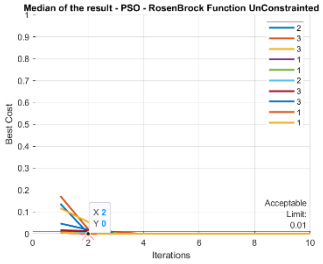


Figure 4.3.1.2

4.3.2. Benchmark UnConPSO-B

With the implementation of constriction coefficient which in turn gives dynamicity to w and wdamp variables, the contour plots (Appendix 7 7.C.2) have been plotted and it shows the convergence at 22th generation to the tolerance value.

The median has been calculated and which found to be 31 (See in Figure 4.3.2.1).

The performance result is obtained with the specification given in section 3.2.2 Benchmark B is given as a table (*see Appendix 8 8.B.6*).

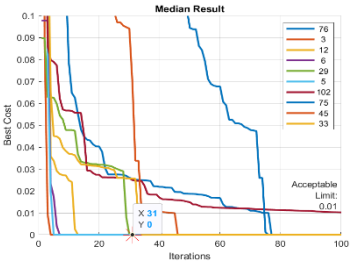


Figure 4.3.2.1

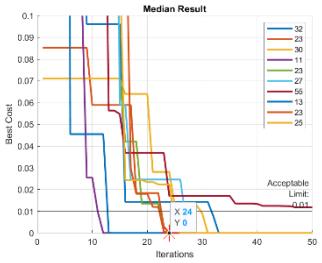


Figure 4.3.2.2

The median has been calculated based on the tuned parameter values, which resulted to 24 (see Figure 4.3.2.2)

4.4 Constrained PSO (ConPSO)

4.4.1. Benchmark ConPSO-A

To see the progress of convergence of the solution towards the best solution, contour plots have been used (appendix 7 7.D.1). The median result found after several tests done which was 26 (see Figure 4.4.1.1). The performance result is obtained with the specification given in section 3.2.1 Benchmark A is given as a table (*see Appendix 8 8.B.7*). Main conclusions drawn are given below.

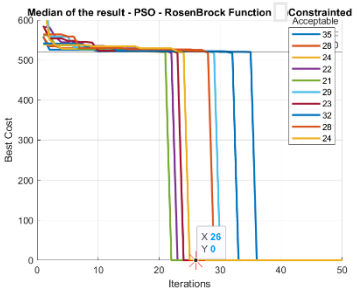


Figure 4.4.1.1

- mu = 0, means no mutation at all which gives a better result, but in this case, the nature of algorithm would change, since this doesn't make sense, mu = 0.3 has been considered as tuned value of mu.

The median result has been calculated based on the tuned parameter values, which resulted to 4 (see Figure 4.1.1.2)

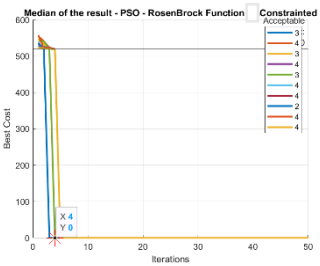


Figure 4.4.1.2

4.4.2. Benchmark ConPSO-B

The objective of this benchmark was to extend section 4.4.1, by implementing dynamicity to inertia coefficient using constriction coefficient, and adding dynamic nature to wdamp which changes the w in every generation. The contour plots associated with this implementation has been provided in Appendix 7 7.D.2.

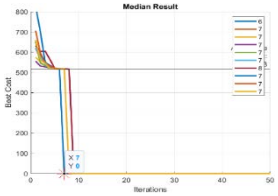


Figure 4.4.2.1

The median result has been calculated and which was found to be 118 (see Figure 4.1.2.1). The performance result is obtained with the specification given in section 3.2.2 is given as a table (*see Appendix 8 8.B.8*). Further analysis done has been done in the following section.

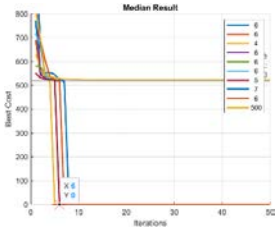


Figure 4.4.2.2

The median has been calculated based on the tuned parameter values, which resulted to 66 (see Figure 4.1.2.3)

4.5 Unconstrained DE (UnConDE)

4.5.1. Benchmark UnConDE-A

The contour plot has been given (See Appendix 7 7.E.1). To find a theoretical optimum of the minimum number of iterations needed to reach this tolerance value given, the code has been tested several times. The median found was 27 (26.5 approximated to 27) (see Figure 4.5.1.1).

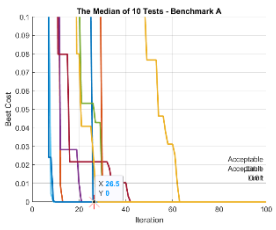


Figure 4.5.1.1

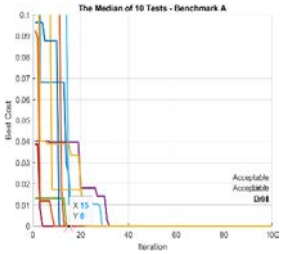


Figure 4.5.1.2

The performance result is obtained with the specification given in section 3.2.3 is given as a table (*see Appendix 8 8.B.9*). The further analysis done has been provide in the following section.

The median has been calculated based on the tuned parameter values, which resulted to 15 (see Figure 4.5.1.2).

4.5.2. Benchmark UnConDE-B

4.5.2.1 Variant 1: DE /best/1/bin

To see the progress of convergence of the solution towards the best solution, contour plots have been used (See appendix 7 7.E.2.1). The median of results found was 17 (see Figure 4.1.1.2). The performance result obtained is given as a table (*see Appendix 8 8.B.10.1*).

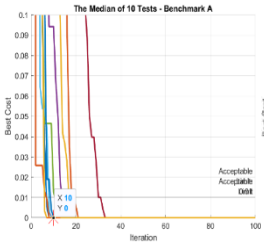


Figure 4.5.2.1.1

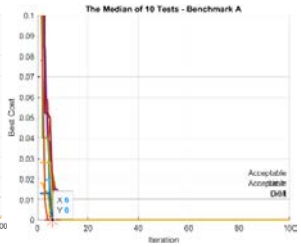


Figure 4.5.2.1.2

Based on the optimized parameter values, the median has been calculated, which was found to be 6 (see Figure 4.1.1.3).

4.5.2.2 Variant 2: DE /best/2/bin

To see the progress of convergence of the solution towards the best solution, contour plots have been used. (see appendix 7 7.E.2.1).

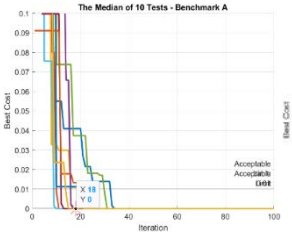


Figure 4.5.2.2.1

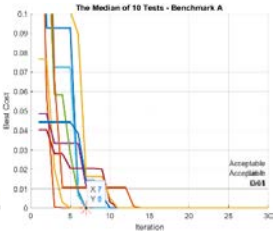


Figure 4.5.2.2.2

The median result has been calculated and which was found to be 18 (see Figure 4.5.2.2.1). To find the effect of parameters a performance analysis has been done (*see Appendix 8 8.B.10.2*). The analysis has been included in the following section.

The median has been calculated based on the tuned parameter values, which resulted to 7 (see Figure 4.5.2.2.2)

4.6 Constrained DE (ConDE)

4.6.1. Benchmark ConDE-A

The contour plots showing the convergence has been implemented. (appendix 7 7.F.1). The convergence to the acceptable result is reached at 17th generation. The median result found was 26 (see Figure 4.6.1.1).

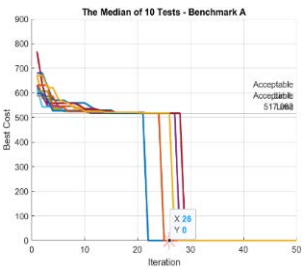


Figure 4.6.1.1

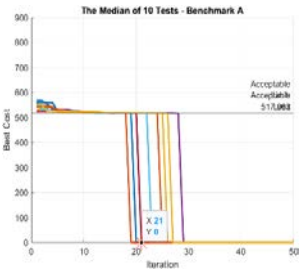


Figure 4.6.1.2

The result of performance analysis is given as a table (*see Appendix 8 8.B.11*). A detailed analysis has been presented in the following sections.

The median has been calculated based on the tuned parameter values, which resulted to 21 (see Figure 4.6.1.2)

4.6.2. Benchmark ConDE-B

4.6.2.1 Variant 1: DE /best/1/bin

To see the progress of convergence of the solution towards the best solution, contour plots have been used (see Appendix 7 7.F.2.1). The median result found was 10 (see Figure 4.6.2.1.1). The performance result is obtained with the specification given in section 3.2.3 is given as a table (*see Appendix 8 8.B.12.1*). The further analysis has been given in the following section.

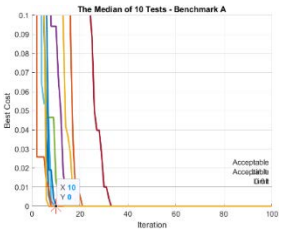


Figure 4.6.2.1.1

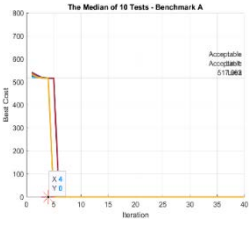


Figure 4.6.2.1.2

The median has been calculated based on the tuned parameter values, which resulted to 4 (see Figure 4.6.2.1.2)

4.6.2.2 Variant 2: DE /best/2/bin

To see the progress of convergence of the solution towards the best solution, contour plots have been used (See Appendix 7 7.F.2.2). The median found was 17 (see Figure 4.6.2.2.1). The performance result is obtained with the specification given in section 3.2.1 Benchmark A is given as a table (*see Appendix 8 8.B.12.2*).

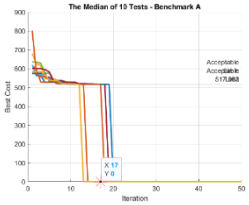


Figure 4.6.2.2.1

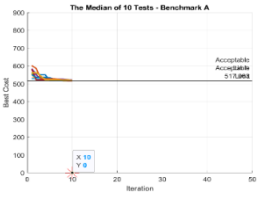


Figure 4.6.2.2.2

The median has been calculated based on the tuned parameter values, which resulted to 10 (see Figure 4.6.2.2.2).

5. Analysis and Interpretations

5.1 Comparison of Median Results

TABLE 1: Median Result								
	Unconstrained (Rosenbrock)				Constrained (Himmelblau)			
	Benchmark A		Benchmark B		Benchmark A		Benchmark B	
	Median (Before Tuning)	Median (After Tuning)	Median (Before Tuning)	Median (After Tuning)	Median (Before Tuning)	Median (After Tuning)	Median (Before Tuning)	Median (After Tuning)
RGA	24	4	118	66	46	26	2806	747
PSO	17	2	31	24	26	4	7	6
DE	27	15	Variant 1: DE /best/1/bin		28	21	Variant 1: DE /best/1/bin	
			10	6			10	4
			Variant 1: DE /best/2/bin				Variant 1: DE /best/2/bin	
			18	7			17	10

Median result is the value which indicates the generations required for each algorithm to solve the objective function (problem) by reaching the solution which can be accepted as a best solution, which probably close to the optimum solution or optimum solution itself. The above table provides the median found, before tuning and after tuning, in each benchmark of every algorithm in unconstrained and constrained problems. There is a significant improvement observed in both. In RGA constraint problem, there is a significant change of median can be observed in Benchmark B compared to Benchmark A because of the effect of operators. In PSO unconstraint problem, the same behaviour can be seen since the median has been hiked after the implementation of constriction coefficients to inertia coefficient. Further discussion about the constriction coefficients has been done in subsection 5.4.

5.2 Comparison of Constrained vs Unconstraint Problem

A detailed comparison has been made with respect to the problems that have been dealt with the selected three algorithms in EC. Four factors which might influence the performance are considered.

First, **size of search space**, which has a significant effect in the algorithm. As variable bound increases, i.e. search space increases, the generation numbers required would also be increased since, the possibility of randomness in the solution space selected, for the algorithm is high, that in turn leads into higher result of generation count. For instance, in consider constrained RGA, (see Figure in Appendix 2 2.A.1.6), and in unconstrainted DE (see Figure in Appendix 6 6.B.1.4), the similar observations can be seen.

Second, **The time of Convergence**, the time required for convergence has been calculated in terms of number of generations (referred with the term ‘result’). It can be easily seen that from the TABLE 1 in section 5.1, among constrained problem implementations, the generation count can be found as significantly high compared to unconstrained implementations of algorithms. In case of constraint problem, the search space is limited to a certain boundary depends on the constraints associated with, compared to same problem with no constraints. So reaching the acceptable solution in the feasible region could take more generations for constrained problems.

Third, **Nature of Objective Function**, this could also have an effect in the algorithm, depends upon the number of local minima exist for the problem. For instance, while solving a real-world problem, if there are multiple ways we can find or reach to the best solution need, then it would be less time consuming and easy to draw to that solution. The same analogy can be applied here too. For unconstrained problems, since the whole search space can be considered as a feasible region, the solution can be reached with lesser number of generations. But for unconstrainted it would be hard and results opposite.

Fourth, **the algorithm chosen**, this would be a noticeable factor, while compare the Median results given in TABLE 1 under subsection 5.1. Depends upon the algorithms chosen the median value varies widely. The features implemented and the extension made which would eventually optimize the performance of the algorithm also has significant impact in the number of generations to reach the required result.

5.3 Impact of Tolerance value

Based on the definition of “tolerance value” given in the section 3.1.3.1, this value is considered as the value which can be accepted as the solution that is close to the optimum value, but with an acceptable error difference. Depends upon the acceptable error bound the result will get improved.

The **quality of the solution** is important, and the error must be minimum as possible from the best solution we have. In unconstrained problem, the chosen maximum error possible was 10^{-2}. But the test results for different tolerance value has shown the variation in the result. For example, see figure given in Appendix 3, 3.B.1.5, which is of unconstraint problem that illustrates the effect of tolerance value. For constraint problems the acceptable error chosen was 2.937. for other possible error values, tests has been evaluated for constraint problems (see Appendix 2 – Figure 2.A.1.7).

It can be easily concluded that, for a problem which needs to be solved with limited time and resources, and accuracy is not a major criterion, then for wide tolerance value, this could be achieved. As the acceptable error reduces, the more work is needed for the algorithm to reach optimum solution.

5.3 Notes on RGA: Effect of Operators

It is worth noting that the effect of operators (benchmark B of RGA algorithm implementation) in RGA. Each operator created a randomness in the solutions or the creation of offsprings solutions. A brief review found during the analysis has been given below.

Selection operator chosen over Binary Tournament Selection was Stochastic Universal Sampling Selection, this would give more efficiency to the code, since with the help of the probability distribution factor, the selection of solutions to be closer and helps to avoid a greater number of clustering that may happens among the solution space selected.

Crossover operator chosen over SBX crossover operator was Linear crossover operator. The comparison of the results forms two observations. Use of SBX crossover gives more wide exploration of solution space over Linear crossover. In case of linear crossover, the offsprings created are closer to its parents, but it’s a simple algorithm in terms of time consumption compared to SBX Crossover.

Mutation operator chosen over Polynomial mutation is Random mutation. In case of polynomial mutation, we add a small value with the parent solution to make more randomness in the offspring solution. So, the solutions would be more close to the parent

solution, its more towards the exploitation rather exploration. But random mutation creates simple offspring randomly, so the crossover operation might not have any such effect in the generation process, if mutation probability is very high, it tends to be more exploration over the whole solution space. For example, see TABLE 1 in section 5.1, while comparing the median of Benchmark B with Benchmark A in RGA, for both in constrained and unconstrained problems, this different of median value can be noted.

Mu, Lambda survivor selection has been used in Benchmark B over mu + lambda survivor selection. Mu, Lambda survivor selection chose the best from the offsprings neglecting the whole parent population in the previous generation. Mu +Lambda does choose best from both parent and offsprings solutions. Mu, Lambda Operator can give more randomness in the problem since it has been used along with random mutation operator. Mu, lambda does exploration and mu + lambda provides exploitation.

Summarize operator's effect described above, in benchmark B, the use of linear crossover, random mutation and mu, lambda survivor selection would tends to do exploration rather exploitation which would make the algorithm to run more number of generation in order to reach the required solution.

5.4 Notes on PSO: Effect of Implementation of Dynamicity

By implementing dynamicity, the aim was too creative a control over the exploration and exploitation of population space. While comparing the Benchmark B of PSO for constraint and unconstraint against Benchmark A of both, several observations can be made. Two observations found are as given below.

Exploration and Exploitation

When constriction coefficient value, kappa is high, the algorithm does more exploitation since the generation count decreases significantly to reach the required solution. The contour plots of unconstrainted PSO Benchmark A (See Appendix 7 7.C.1) and Benchmark B (See Appendix 7 7.C.2) shows this phenomenon graphically. When, in 7.C.1, at 20^{th} generation the solutions getting convergence more, in 7.C.2, the same result found in 10^{th} generation itself. The same can be witnessed in constrained PSO too. (See Appendix 7.D). The optimum value of these parameters can be found only through tuning since the problem and the solution space both has got impact on these parameters.

Nature of the Problem

For example, from TABLE 1, in benchmark B's of both problems of PSO, it can be easily seen that, in case of solving unconstraint Rosenbrock function, after implementing constriction operator the median has been increased compared to benchmark A. So, the effect of the dynamicity has not improved the algorithm in terms of number of generations, but in terms of exploration, it has been improved greatly. Whereas, for constrained Himmelblau function, benchmark B is more efficient than benchmark A; this shows that the dynamicity has improved the algorithm greatly for constraint problems.

5.5 Notes on DE: Effect of Variants

After the analysis of the effects of variants of DE, such as DE rand/1/bin (Benchmark A), DE best/1/bin (Benchmark B), and DE best/2/bin (Benchmark B) for solving the constrained and unconstrained problem, some observations have been drawn which are noted below.

- In case of DE rand/1/bin, which has a tendency of doing more exploration since the base vector chosen is a random vector from the whole population while creating trail vector. So, it does more extensive exploration over the entire population.
- In case of DE best/1/bin, the trail vector is mainly influenced by best found in the previous generation, so this will in turn prioritize exploitation, because as generation goes the solutions would more inclined towards the parents rather going away from the parent vector.
- In case of DE best/2/bin, which also does like DE best/1/bin, but since the number of differences is 2, which would let the trial vector move away more from the parent or particle, which eventually causes exploration but not of great extend like what happens in DE rand/1/bin.
- The median of all these three shows that DE best/1/bin gives best result (See TABLE 1 in section 5.1) compared to the rest of the variants. It is worth to note that the graphical representation of how convergence takes place. In DE rand/1/bin (See Appendix 7 7.E.1), because of exploration, the randomness created in generations, through at 40th generation the solutions started to get into a cluster, at 533rd generation it reached the optimum value (here we considered optimum result rather tolerance value). Whereas in DE best/1/bin (See Appendix 7 7.E.2.1), at 40th generation, every solution has been close into the solution. And finally, 146th generation provides the optimum solution.

5.6 Impact of Static Penalty in Constrained problems

The impact of static penalty in solving constrained problems were found to be significant throughout the optimization process and performance testing. The observations made are drawn below.

Observation 1: Preservation of Feasibility Region

For different values of R, the solutions spread across the population space has been found to be different. That is, for lower values of R, the penalty is less, which make the solutions/ offsprings fall into infeasible region more than feasible region.

For example, in unconstrained RGA (See Appendix 2 2.A.1.8), when R=2, solutions are in infeasible region, so the solutions produced are far away from the optimum solution though each generation produces result which is far below the optimum solution required. When R=46, the best solution found reaches near optimum solution in 1st generation, but not all no convergence. When R=100, the penalty is high, so solutions are more strictly in the feasible region. As R goes high, like 500 and 1000, the penalty of the solution which comes under the infeasible region is high, which preserves the feasibility region.

Observation 2: Effect of Algorithm

With RGA, for different values of R values, the algorithm doesn't have any impact on the exploration over solution space (see Appendix 2 2.A.1.8). Whereas, in the case PSO (See Appendix 4 4.A.1.7), for different values of R, the algorithm tents to explore the solution space, which would indirectly improve the performance of algorithm. In case of DE (See Appendix 6 6.A.1.5), this exploration can be seen more for higher values of R and it reaches the solution in lesser number of generations.

6. Comments for Further Research

This section provides some observations found which needed to be investigated further to optimize the algorithm in terms of time, cost, and efficiency, with a deep analysis.

6.1. Effect of Number of Population (nPop)

How nPop values influence the algorithms have been observed and a common pattern observed throughout tests. For higher values of nPop, the number of generations are lesser to get optimized solution since the possibility of having optimum result in a single generation is very high. But as a side effect, the time consumption is also high when nPop is higher since resource utilization would also be high. Based on No Free Lunch Theorem, this can be tolerated, but further investigation is mandatory to make more improvement.

6.2. Effect of Penalty Techniques

This report uses Static penalty as the penalty technique with constrained problem for optimization of algorithm. Since the penalty technique can influence domains such as, preservation of feasibility region, quality of solution, efficiency of computation in terms of resources and time, exploration capacity of the algorithm and others, further research is necessary with alternative penalty techniques to conclude the optimized specification for algorithms.

6.3. Effect of Different Objective functions

This report discussed about optimization of constrain and unconstrained algorithms with one function for each. For an accurate prediction and improve the efficiency of algorithm, further research is important in terms of testing the algorithm with different mathematical problems with different number of constraints, types, and variables.

7. Conclusion

Evolutionary computing has been considered as one of the most robust domains for problem solving where solutions are hard to determine with maximum efficiency. This report discussed about three algorithms, such as Genetic algorithm, Particle Swarm Optimization, and Differential Evolution. The objective functions that have been taken as and for unconstrainted problem solving was Rosenbrock function, and for constrained problem it was Himmelblau function. The problem definitions has been included in this report; followed by that, specification of each algorithm has been described. Based on the specification described, performance analysis has been done for all algorithms with two types, such as constrained and unconstrained. The observation found has been noted along with analysis of different aspects of the implementation have been discussed. Finally, the improvements and provisions for further improvement have been recommended, along with a conclusion.

References

Biscani., D. I. a. F., 2015. *PYGMO.* [Online]
Available at: https://esa.github.io/pygmo/

Dorigo, M. a. B. M. a. S. T., 2006. Ant colony optimizatio. *IEEE computational intelligence magazine,* 1-4(IEEE), pp. 28-39.

Duch, W., 2007. What is Computational Intelligence and where is it going?. In: *Challenges for computational intelligence.* s.l.:Springer, pp. 1-13.

Eberhart, R. a. K. J., 1995. Particle swarm optimization. In: 4, ed. *Proceedings of the IEEE international conference on neural networks.* Citeseer: IEEE, pp. 1942--1948.

Electronics, I. N. N. C. a. I. o. E. a., 1994. *Proceedings of the First IEEE Conference on Evolutionary Computation: IEEE World Congress on Computational Intelligence, June 27-June 29, 1994, Walt Disney World Dolphin Hotel, Orlando, Florida.* number={v. 1},isbn={9780780318991}, ed. s.l.:IEEE Neural Networks Council}.

Himmelblau, D. M. a. o., 2018. *Applied nonlinear programming.* McGraw-Hill ed. s.l.:s.n.

Holland, J. H., 1962. Outline for a logical theory of adaptive systems. *Journal of the ACM (JACM)},* 9(ACM New York, NY, USA), pp. 297-314.

Marc Claesen, J. S. a. D. P., 2014. *OPTUNITY.* [Online]
Available at: https://optunity.readthedocs.io/en/latest/

MATWORKS, Since 1994. *MathWorks.* [Online]
Available at: https://www.mathworks.com/products/matlab.html
[Accessed 2024].

others, D. H. a., 2009. *MOEA.* [Online]
Available at: http://moeaframework.org/

Rosenbrock, H. H., 1960. An Automatic Method for Finding the Greatest or Least Value of a Function. *The Computer Journal,* 3(10.1093/comjnl/3.3.175), pp. 175-184.

Storn, R. a. P. K., 997. Differential evolution--a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization,* 11(Springer), pp. 341--359.

7. Appendices

Appendix 1

1.A: Benchmark UnConRGA-A

1.A.1 Outcomes of Tests

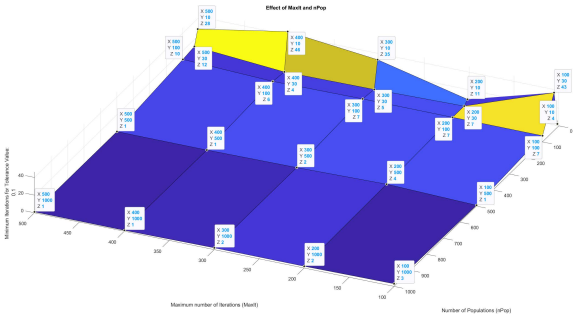


Figure 1.A.1.1

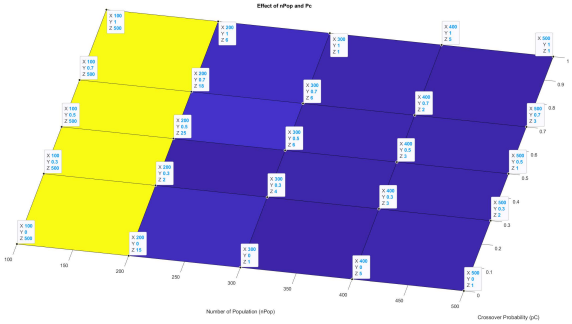


Figure 1.A.1.2

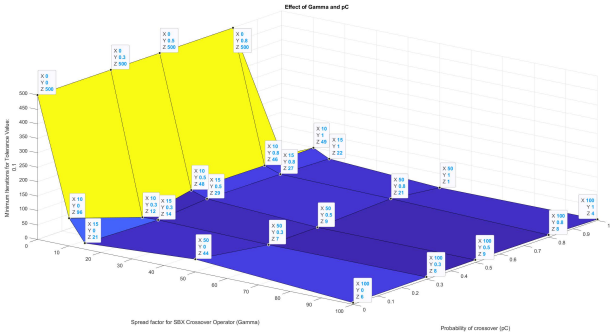


Figure 1.A.1.3

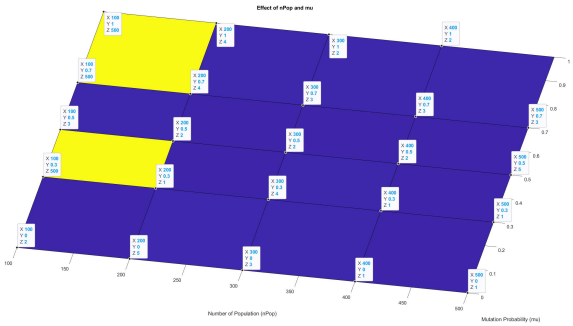


Figure 1.A.1.4

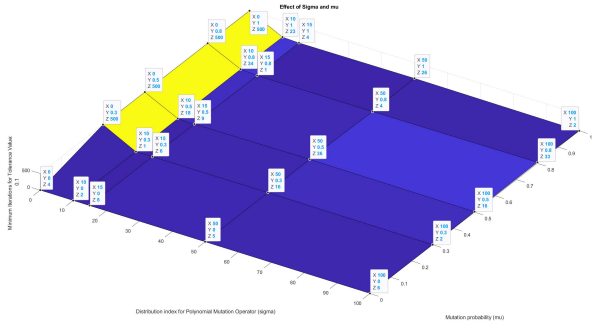


Figure 1.A.1.5

1.A.2 MATLAB Code Associated with the tests

Unconstraint RGA Implementation:

%RGA Algorithm:

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

% Problem
CostFunction = problem.CostFunction;
nVar = problem.nVar;
VarSize = [1, nVar];
VarMin = problem.VarMin;
VarMax = problem.VarMax;
toleranceValue = problem.toleranceValue;
% Params
MaxIt = params.MaxIt;
nPop = params.nPop;
% beta = params.beta;
pC = params.pC;
nC = round(pC*nPop/2)*2;
gamma = params.gamma;
mu = params.mu;
sigma = params.sigma;

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

% Best Solution Ever Found
bestsol.Cost = inf;

% Initialization
pop = repmat(empty_individual, nPop, 1);
for i = 1:nPop

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

    % Evaluate Solution
    pop(i).Cost = CostFunction(pop(i).Position(1),
pop(i).Position(2));

    % Compare Solution to Best Solution Ever Found
    if pop(i).Cost < bestsol.Cost
        bestsol = pop(i);
    end

end

% Best Cost of Iterations
```

```
bestcost = nan(MaxIt, 1);

% Main Loop
for it = 1:MaxIt

    % Initialize Population for Evolution Stategy
    popc = repmat(empty_individual, nC, 1);

    % Applying Binary Tournament Selection
    popc  = applyBinaryTournamentSelection(pop,nPop,popc,nC);

    %Effect of Operators
    % Applying Stochastic Universal Sambling Selection Opearation
    % popc  = applyStochasticUniversalSamplingSelection(pop, nPop,
popc, nC, beta);

    % Convert popc to nC/2x2 Matrix for Crossover Operation
    popc = reshape(popc,nC/2,2);

    % Applying Crossover Operation
    for k = 1:nC/2
        % Perform SBX Crossover Operation
        [popc(k, 1).Position, popc(k, 2).Position] =
applySBXCrossover(popc(k,1), popc(k,2), pC, gamma);
        %Effect of Operators
        %Perform Linear Crossover
        % [popc(k, 1).Position, popc(k, 2).Position] =
applyLinearCrossover(popc(k,1).Position, popc(k,2).Position, pC,
CostFunction,empty_individual );
        % Check for Variable Bounds
        popc(k,1).Position = max(popc(k,1).Position, VarMin);
        popc(k,1).Position = min(popc(k,1).Position, VarMax);
        popc(k,2).Position = max(popc(k,2).Position, VarMin);
        popc(k,2).Position = min(popc(k,2).Position, VarMax);
    end
    % Convert popc to Single-Column Matrix
    popc = popc(:);
    % Apply Mutation Operation
    for l = 1:nC

        % Perform Polynomial Mutation
        popc(l).Position = applyPolynomialMutation(popc(l).Position,
mu, sigma, VarMin, VarMax);
        %Effect of Operators
        %perform Random Mutation Operator
        % popc(l).Position = applyRandomMutation(popc(l).Position,
mu, VarMin, VarMax);

        % Check for Variable Bounds
```

```
        popc(l).Position = max(popc(l).Position, VarMin);
        popc(l).Position = min(popc(l).Position, VarMax);

        % Evaluation
        popc(l).Cost =
CostFunction(popc(l).Position(1),popc(l).Position(2));

        % Compare Solution to Best Solution Ever Found
        if popc(l).Cost < bestsol.Cost
            bestsol = popc(l);
        end

    end

    %applying Mu + Lambda Survivor Selection survivor selection
operaton
    pop =applyMuPlusLambdaSurvivorSelection([pop; popc], nPop);

    %Effect of Operators
    %apply Mu, Lambda Survivor Selection Operator
    % pop = applyMuLambdaSurvivorSelection(popc, nPop);

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

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

    if bestcost(it) <= toleranceValue
        disp(['Tolerance value has been reached at generation: '
num2str(it)]);
        % Results
        out.bestsol = bestsol;
        out.bestcost = bestcost;
        out.minIterationToReachToleranceValue = it;
        return;
    end


end

% Results
out.pop = pop;
out.bestsol = bestsol;
out.bestcost = bestcost;
%the below line set is by maing an assumption that MaxIt is atleast
needed
```

```
%to reach the tolerance value with the given specification. its for
avoid
%the error of indefinit looing while finding the result
out.minIterationToReachToleranceValue = MaxIt;
end
```

%Selection Operator: Tournament Selection

```
function popc = applyBinaryTournamentSelection(pop,nPop,popc,nC)
    % applyBinaryTournamentSelection => Tournament operator with
k=2, pick
    % 2 random from the population and take the best
    %Steps involved
    % 1. choose two random values from 1 to nPop(number of
population)
    % 2. compare the cost
    % 3. chose minimum cost valued index, save it into popc
    % 4. iterate from step 2 nC times        % p1 and p2 consists
the parents corresponding to the random

    k=1;
    while k<=nC
        %Generate two random indices
        %with the following code the random generated number often
happens
        %to be same, so I add code to ensure both random numbers are
same

        %Method2
        r = randperm(nPop);
        [index1, index2] = deal(r(1),r(2));

        % indexes from Population (pop)
        p1 = pop(index1);
        p2 = pop(index2);
        if p1.Cost <= p2.Cost
            %p1 wins in the selection process since the cost is less
            %compared to the competent
            %Assign the value into popc, for further process
            popc(k,1) = p1;
            k=k+1;
        end
    end
end
```

%Crossover Operator: SBX Crossover

```
function [o1, o2] = applySBXCrossover(parent1, parent2, pC, gamma)
```

```
%p1 and p2 are parents, parent 1 and parent 2 , and both are of same
size
%gamma is spread factor for crossover

%compare with ramdom probability with the crossover probability
if rand() < pC

    % Compare cost of parent1 and parent2 and ensure that
parent1<parent2
    % by swapping both
    if parent1.Cost > parent2.Cost
        temp = parent2;
        parent2 = parent1;
        parent1 = temp;
    end

    %perform crossover
    %initialize offprings of parent size
    o1 = zeros(size(parent1.Position));
    o2 = zeros(size(parent1.Position));
    % for each index/ variable in parent we do perform the following
    % operations and generate the curresponding two variables for
each offspring o1 and o2
    for l=1:length(parent1.Position)

        %step1: Generate random probabilities which is in between 0
and 1
        u = rand();
        %step2: Determine beta value corresponding to u value
        %check the condition
        if(u <= 0.5)
            beta = (2*u)^(1/(gamma+1));
        else
            beta = (1/(2*(1-u)))^(1/(gamma+1));
        end

        %perform crossover and generate the variables of o1 and o2
        %curresponding to p1 and p2
        o1(l) = 0.5 * ((parent1.Position(l) + parent2.Position(l)) -
beta * (parent2.Position(l)-parent1.Position(l)));
        o2(l) = 0.5 * ((parent1.Position(l) + parent2.Position(l)) +
beta * (parent2.Position(l)-parent1.Position(l)));


    end
else
    %no crossover
    o1 = parent1.Position;
    o2 = parent2.Position;
```

```
end
end
```
%Mutation Operator: Polynomial Mutation

```
function o = applyPolynomialMutation(p, mu, sigma, VarMin, VarMax)
%p - parent
%mu - mutation probability
%sigma - distribution index used to calculate delta


%initialize offprings
o = zeros(size(p));
% compared randomly generated probability with mutation probability
if rand() < mu
    %perform mutation
    % Iterate over each gene (parameter) in the individual
    for i = 1:length(p)
        % generate random number for mutation
        u = rand();
        % calculate delta value for mutation, delta is the variation
        if u < 0.5
            delta = (2*u)^(1/(sigma+1))-1;
        else
            delta = 1 - (2*(1-u))^(1/(sigma+1));
        end

        % apply mutation using this delta and sigma values
        o(i) = p(i) + delta * (VarMax - VarMin);
    end
else
    %no mutation performed
    o=p;
end
end
```
%survivor selection Operator: mu+Lambda

```
function pop = applyMuPlusLambdaSurvivorSelection(pop, nPop)

    [~, so] = sort([pop.Cost]);
    pop = pop(so);
    % Remove Extra Individuals
    pop = pop(1:nPop);
end
```

Function Call for Figure 4.1.1.2:

```
%% Calling RGA - FOR MEDIAN CALCULATION
% Run 10 experiments and find the median of those results
numberOfIterations =10;
experimentsResults = zeros(1,numberOfIterations);
```

```
for i=1:numberOfIterations
    out = RunRGA(problem, params);
    experimentsResults(i) = out.minIterationToReachToleranceValue;
    %plot the value in the graph if exists otherwise plot it in a
new graph
    hold on
    semilogy(out.bestcost,"LineWidth",2)
    grid on;
    hold off
end
disp(["Results found: " num2str(experimentsResults)])
medianOfResults = median(experimentsResults);
disp(["Median of the Results (Min iterations needed to reach the
tolerance value): " num2str(medianOfResults)])

%plot median value in the graph
hold on
semilogy(medianOfResults,0,"r*","MarkerSize",20)
hold off

%graph parameters
title("Median Result");
xlabel('Iterations');
ylabel('Best Cost');
xlim([0,200]);
ylim([0, 10^-1])
%draw a line parallel to x axis to find on which iteration the
output
%reaches to tolerance value
yline(problem.toleranceValue,'-',{'Acceptable','Limit:',
num2str(problem.toleranceValue) });
legend(num2str(experimentsResults'), 'location', 'northeast');
grid on;
```

Function Call for Figure 1.A.1.1:

```
%% Effect of MaxIt and nPop

%save different values of MaxIt and nPop inorder to do the
experiment
maxItVariation = [100, 200, 300, 400, 500];
nPopVariation = [10, 30, 100, 500, 1000];

% use two for loops to do the experiment with various combinations
of
% parameters taken into consideration
for i=1:numel(maxItVariation)
     % Intertia Coefficient (w)
    params.MaxIt = maxItVariation(i);
    for j=1:numel(nPopVariation)
```

```
        % Damping Ratio of Inertia Coefficient (wdamp)
        params.nPop = nPopVariation(j);
        out = RunRGA(problem, params);
        Z(i,j)= out.minIterationToReachToleranceValue;
    end
end
```

Function Call for Figure 1.A.1.2:

```
%% Effect of Population Size (nPop) and Probability of Crossover
(cP)


%save different values of nPop and pC inorder to do the experiment
nPopVariation = [100, 200, 300, 400, 500];
pCVariation = [0 0.3 0.5 0.7 1];

% use two for loops to do the experiment with various combinations
of
% parameters taken into consideration
for i=1:numel(nPopVariation)
    params.nPop = nPopVariation(i); %number of populations
    for j=1:numel(pCVariation)
        params.pC = pCVariation(i);  % Probability of crossover
        out = RunRGA(problem, params);
        Z(i,j)= out.minIterationToReachToleranceValue;
    end
end
```

Function Call for Figure 1.A.1.3:

```
%% Effect of Gamma and pC

%save different values of Gamma and pC inorder to do the experiment
gammaVariation = [0, 10, 15, 50, 100];
pCVariation = [0 0.3 0.5 0.8 1]; %in between 0 and 1

% use two for loops to do the experiment with various combinations
of
% parameters taken into consideration
for i=1:numel(gammaVariation)
        params.gamma = gammaVariation(i); % Spread factor for SBX
Crossover Operator
    for j=1:numel(pCVariation)
        params.pC = pCVariation(i);  % Probability of crossover
        out = RunRGA(problem, params);
        Z(i,j)= out.minIterationToReachToleranceValue;
    end
end
```

Function Call for Figure 1.A.1.4:

```
%% Effect of Population Size (nPop) and Probability of Mutation (mu)


%save different values of nPop and pC inorder to do the experiment
nPopVariation = [100, 200, 300, 400, 500];
muVariation = [0 0.3 0.5 0.7 1];

% use two for loops to do the experiment with various combinations
of
% parameters taken into consideration
for i=1:numel(nPopVariation)
    params.nPop = nPopVariation(i); %number of populations
    for j=1:numel(muVariation)
        params.mu = muVariation(i);  % Probability of Mutation
        out = RunRGA(problem, params);
        Z(i,j)= out.minIterationToReachToleranceValue;
    end
end
```

Function Call for Figure 1.A.1.5:

```
%% Effect of Sigma and mu

%save different values of Sigma and mu inorder to do the experiment
sigmaVariation = [0, 10, 15, 50, 100];
muVariation = [0 0.3 0.5 0.8 1]; % in between 0 and 1

% use two for loops to do the experiment with various combinations
of
% parameters taken into consideration
for i=1:numel(sigmaVariation)
    params.sigma = sigmaVariation(i);   % Distribution index for
Polynomial Mutation Operator
    for j=1:numel(muVariation)
        params.mu = muVariation(i);% 1/nVar  % Mutation probability
        out = RunRGA(problem, params);
        Z(i,j)= out.minIterationToReachToleranceValue;
    end
end
```

1.B: Benchmark UnConRGA-$\mathcal{B}$

1.B.1 Outcomes of Tests

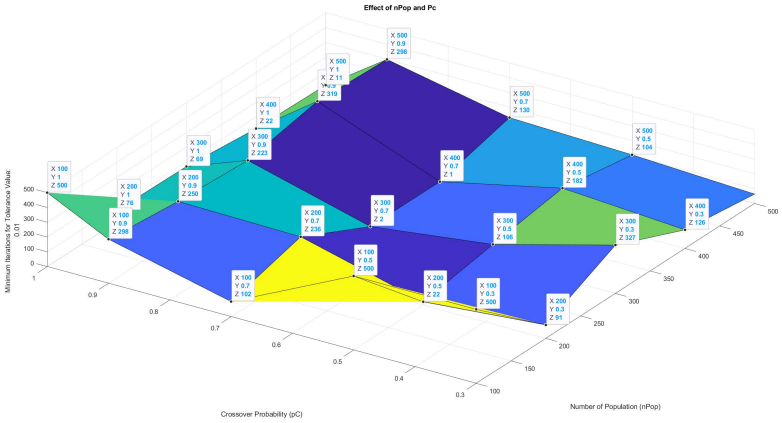


Figure 1.B.1.1

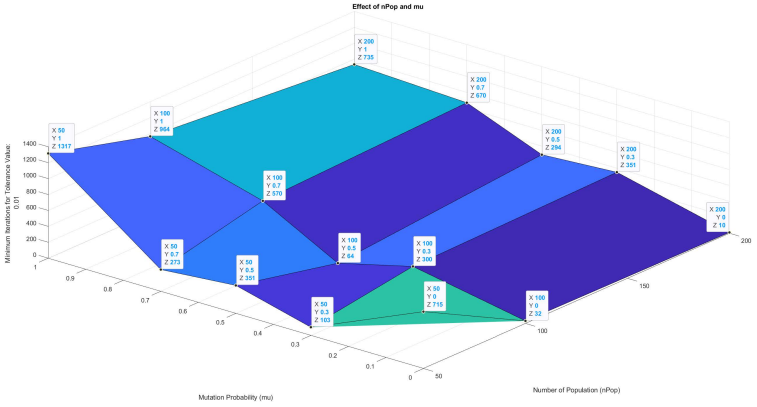


Figure 1.B.1.2

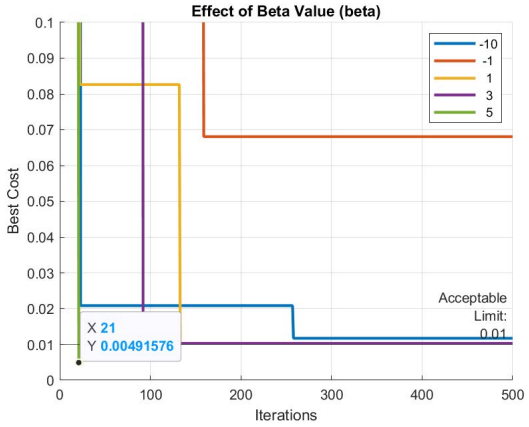


Figure 1.B.1.3

1.B.2 MATLAB Code of Newly Implemented Operators

%Selection Operator: Stochastic Universal Sampling Selection

```
function popc  = applyStochasticUniversalSamplingSelection(pop,
nPop, popc, nC, beta)

    % Selection of Probabilities
    c =[pop.Cost];
    cavg = mean(c);
    if cavg ~= 0
        c =c/cavg;
    end
    %distribution of probabilities based on cost of each solution
    probs = exp(-beta*c);

    %finding cumulative sum
    cumulativeSum = cumsum(probs);

    %create a random variable in between 0 and 1
    r = rand(1);

    % create crossover population from parent population by
iterating steps
    % nC times to find each solution at a time.
    for k=1:nC
        % generate a random number by adding one of nPop divisions
with the
        % random number and take the fraction part (which is less
than zero)
        r = mod(r+(k-1)/nPop,1);
        %find the first index by traversing from left to right in
the 'cumulative sum row matrix' subjected to the
        %condition that r > 'the value at the index'
        index = find(r <= cumulativeSum, 1, 'first');
        % add the solution from parent population into crossover
population
        popc(k,1) = pop(index);
    end
end
```

%Crossover Operator: Linear Crossover

```
function [o1, o2] = applyLinearCrossover(p1, p2,pC,
CostFunction,empty_individual)
%compare with a random probability with crossover probability
if rand() < pC
```

```
    %initialize array for 3 offprings
    offSprings = repmat(empty_individual, 3, 1);
    %generate three crossovers and calculate costs
    offSprings(1).Position =  (1/2).*p1 + (1/2).*p2;
    %calculate the cost
    offSprings(1).Cost = CostFunction( offSprings(1).Position(1),
offSprings(1).Position(2));

    offSprings(2).Position= (3/2).*p1 - (1/2).*p2;
    %calculate the cost
    offSprings(2).Cost = CostFunction( offSprings(2).Position(1),
offSprings(2).Position(2));

    offSprings(3).Position = (3/2).*p2 - (1/2).*p1;
    %calculate the cost
    offSprings(3).Cost = CostFunction( offSprings(3).Position(1),
offSprings(3).Position(2));

    %extract the two best solutions
    [~, so] = sort([offSprings.Cost], "descend");
    offSprings = offSprings(so);
    % Remove Extra Individuals
    o1 = offSprings(1).Position;
    o2 = offSprings(2).Position;
else
    o1 = p1;
    o2 = p2;
end
end
```

%Mutation Operator: Random Mutation

```
function o = applyRandomMutation(p, mu, VarMin, VarMax)

    % Compare with a random probability with mutation probability
    if rand() > mu
        %perform mutation
        for i=1:length(p)
            %create offspring randomly with the bounds of variable
            o(i) = rand()*(VarMax -VarMin);
        end
    else
        %no mutation performed, parent itself refered as offspring
        o=p;
    end

end
```

%Survivor Selection Operator: Mu-Lambda Survivor Selection

```
function pop = applyMuLambdaSurvivorSelection(popc, nPop)
    %choosing best nPop from popc since nPop < nC ==> number of
offspring
    %population is higher than number of parent population
    [~, so] = sort([popc.Cost]);
    popc =popc(so);
    %select the first 'nPop' solutions from updated popc, which will
be the
    %parent population for the next iteration
    pop = popc(1:nPop);
end
```

Appendix 2

2.A: Benchmark ConRGA-A

2.A.1 Outcomes of Tests

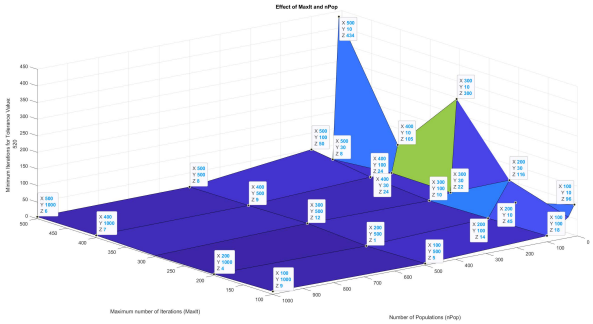


Figure 2.A.1.1

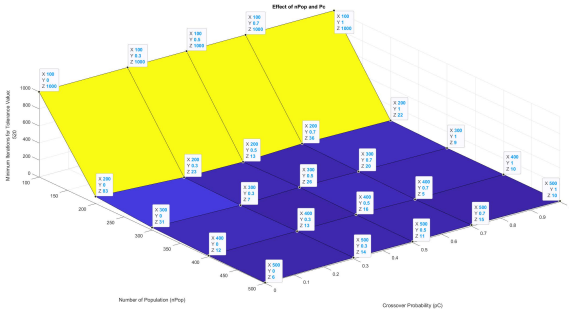


Figure 2.A.1.2

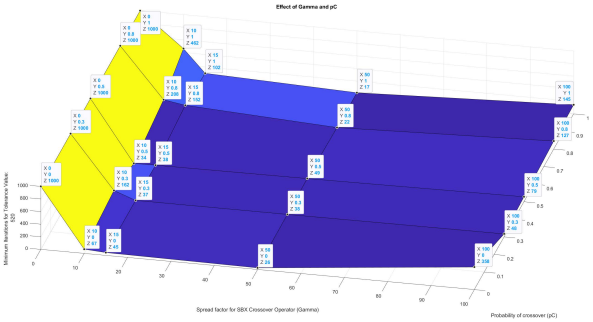


Figure 2.A.1.3

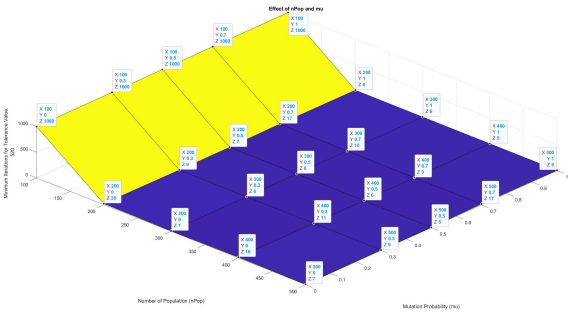


Figure 2.A.1.4

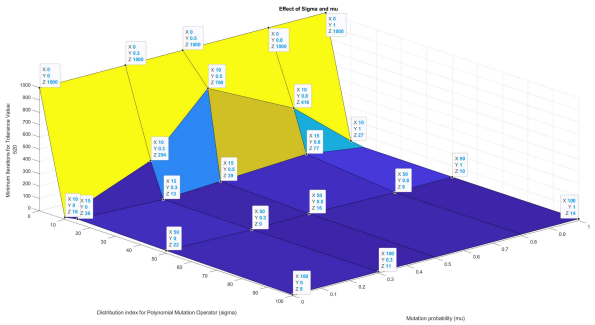


Figure 2.A.1.5

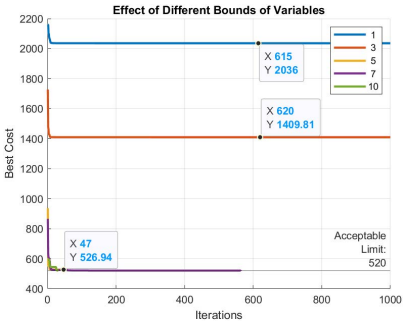


Figure 2.A.1.6

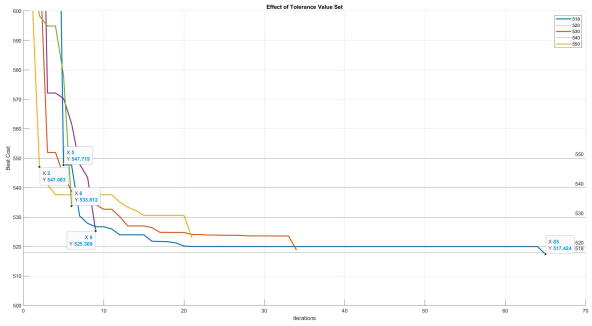


Figure 2.A.1.7

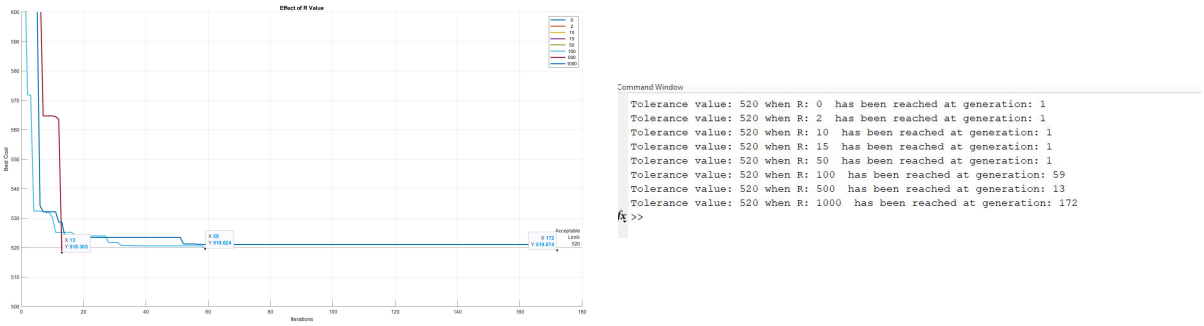


Figure 2.A.1.8

```
if bestcost(it) <= toleranceValue
    disp(['Tolerance value: ' num2str(toleranceValue) ' when R: ' num2str(R) '  has been reached at generation: ' num2str(it)]);
    % Results
    out.bestsol = bestsol;
    out.bestcost = bestcost;
    out.minIterationToReachToleranceValue = it;
    % % disp(bestsol)
    % disp(['Constraints: ', num2str(Constraints(bestsol.Position(1), bestsol.Position(2), R))] );
    return;
end
```

Figure A9

2.A.2 MATLAB Code Associated with the tests

Constraint RGA Implementation:

%Himmelblau function

```
% Himmelblau function
function z = HimmelblauFunction(x,y)

    z = (x.^2 + y - 11).^2 + (x + y.^2 - 7).^2;
end
```

%Implementation of constraints

```
function g = InequalityConstraints(x1, x2, R)

% Constraints g(x1, x2) >= 0
g(:,:,1) = (x1-5).^2 + x2.^2 - 26;
g(:,:,2) = 4*x1+x2-20;
g(:,:,3) = -x1 < 0; %for x1> 0
g(:,:,4) = -x2 < 0; % for x2> 0

% If Constraint satisfied, then make it 0
g(g > 0) = 0;


% Penalty = Σ R*|{g(u1,u2)}|
g = sum(R.*abs(g),3);

% Remove extra dimensions
g = squeeze(g);



%% RGA - Constraints Himmelblue Function and Static Penalty

function out = RunRGA(problem, params)

%% Problem
Constraints = problem.Constraints; % Constraint function
FitnessValue = problem.FitnessValue; %Fitness function
nVar = problem.nVar;
VarSize = [1, nVar];
VarMin = problem.VarMin;
VarMax = problem.VarMax;
toleranceValue = problem.toleranceValue;
%% Params
MaxIt = params.MaxIt;
nPop = params.nPop;
% beta = params.beta;
pC = params.pC;
nC = round(pC*nPop/2)*2;
gamma = params.gamma;
mu = params.mu;
sigma = params.sigma;
R = params.R;
% Template for Empty Individuals
empty_individual.Position = [];
empty_individual.Cost = [];

% Best Solution Ever Found
bestsol.Cost = inf;
```

```
% Initialization
pop = repmat(empty_individual, nPop, 1);
for i = 1:nPop

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

    % Evaluate Solution
      pop(i).Cost = FitnessValue(pop(i).Position(1),
pop(i).Position(2), R);

    % Compare Solution to Best Solution Ever Found
    if pop(i).Cost < bestsol.Cost
        bestsol = pop(i);
    end

end

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

% Main Loop
for it = 1:MaxIt

    % Initialize Population for Evolution Stategy
    popc = repmat(empty_individual, nC, 1);

    % Applying Binary Tournament Selection
    popc  = applyBinaryTournamentSelection(pop,nPop,popc,nC);

    %Effect of Operators
    % Applying Stochastic Universal Sambling Selection Opearation
    % popc  = applyStochasticUniversalSamplingSelection(pop, nPop,
popc, nC, beta);


    % Convert popc to nC/2x2 Matrix for Crossover Operation
    popc = reshape(popc,nC/2,2);


    % Applying Crossover Operation
    for k = 1:nC/2
        % Perform SBX Crossover Operation
        [popc(k, 1).Position, popc(k, 2).Position] =
applySBXCrossover(popc(k,1), popc(k,2), pC, gamma);


        %Effect of Operators
        %Perform Linear Crossover
```

```
        % [popc(k, 1).Position, popc(k, 2).Position] =
applyLinearCrossover(popc(k,1).Position, popc(k,2).Position, pC,
CostFunction,empty_individual );
        % Check for Variable Bounds
        popc(k,1).Position = max(popc(k,1).Position, VarMin);
        popc(k,1).Position = min(popc(k,1).Position, VarMax);
        popc(k,2).Position = max(popc(k,2).Position, VarMin);
        popc(k,2).Position = min(popc(k,2).Position, VarMax);
    end

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

    % Apply Mutation Operation
    for l = 1:nC

        % Perform Polynomial Mutation
        popc(l).Position = applyPolynomialMutation(popc(l).Position,
mu, sigma, VarMin, VarMax);


        %Effect of Operators
        %perform Random Mutation Operator
        % popc(l).Position = applyRandomMutation(popc(l).Position,
mu, VarMin, VarMax);

        % Check for Variable Bounds
        popc(l).Position = max(popc(l).Position, VarMin);
        popc(l).Position = min(popc(l).Position, VarMax);

        % Evaluation
        popc(l).Cost =
FitnessValue(popc(l).Position(1),popc(l).Position(2), R);

        % Compare Solution to Best Solution Ever Found
        if popc(l).Cost < bestsol.Cost
            bestsol = popc(l);
        end

    end

    %applying Mu + Lambda Survivor Selection survivor selection
operaton
    pop =applyMuPlusLambdaSurvivorSelection([pop; popc], nPop);


    %Effect of Operators
    %apply Mu, Lambda Survivor Selection Operator
    % pop = applyMuLambdaSurvivorSelection(popc, nPop);
```

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

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


    if bestcost(it) <= toleranceValue
        disp(['Tolerance value has been reached at generation: '
num2str(it)]);
        % Results
        out.bestsol = bestsol;
        out.bestcost = bestcost;
        out.minIterationToReachToleranceValue = it;
        disp(bestsol)
        disp(['Constraints: ',
num2str(Constraints(bestsol.Position(1), bestsol.Position(2), R))]
);
        return;
    end


end

% Results
out.pop = pop;
out.bestsol = bestsol;
out.bestcost = bestcost;
%the below line set is by maing an assumption that MaxIt is atleast
needed
%to reach the tolerance value with the given specification. its for
avoid
%the error of indefinit looing while finding the result
out.minIterationToReachToleranceValue = MaxIt;
% Display Best Solution in Command Window
disp(bestsol)
disp(['Constraints: ', num2str(Constraints(bestsol.Position(1),
bestsol.Position(2), R))] );

end
```

Function Call for Figure 4.2.1.2:

```
%% RGA - Constraints Himmelblue Function and Static Penalty
clc; clear;

%% Problem Definition
```

```
problem.CostFunction = @(x1,x2) HimmelblauFunction(x1,x2); % Cost
Function
problem.Constraints = @(x1,x2,R) InequalityConstraints(x1,x2,R); %
Constraint
problem.FitnessValue = @(x1,x2,R) problem.CostFunction(x1,x2) +
problem.Constraints(x1,x2,R); % Fitness Value

problem.nVar = 2;              % Number of Decision Variables
problem.VarMin = 0;            % Lower Bound of Decision Variables
x1 and x2 are greater than 0
problem.VarMax = 6;             % Upper Bound of Decision Variables
problem.toleranceValue = 520; % this is the optimum solution when x1
= 3.763 and x2 = 4.947

%% RGA Parameters
params.MaxIt = 1000;
params.nPop = 40;
params.pC = 1;  % Probability of crossover
params.gamma = 15; % Spread factor for SBX Crossover Operator
params.mu = 0.5; % = 1/nVar  % Mutation probability
params.sigma = 20;   % Distribution index for Polynomial Mutation
Operator

params.R = 100;  % Static Penalty Parameter
%% Calling RGA - FOR MEDIAN CALCULATION
% Run 10 experiments and find the median of those results
numberOfIterations =10;
experimentsResults = zeros(1,numberOfIterations);
for i=1:numberOfIterations
    out = RunRGA(problem, params);
    experimentsResults(i) = out.minIterationToReachToleranceValue;
    %plot the value in the graph if exists otherwise plot it in a
new graph
    hold on
    semilogy(out.bestcost,"LineWidth",2)
    grid on;
    hold off
end
disp(["Results found: " num2str(experimentsResults)])
medianOfResults = median(experimentsResults);
disp(["Median of the Results (Min iterations needed to reach the
tolerance value): " num2str(medianOfResults)])

%plot median value in the graph
hold on
semilogy(medianOfResults,0,"r*","MarkerSize",20)
hold off

%graph parameters
```

```
title("Median Result");
xlabel('Iterations');
ylabel('Best Cost');
xlim([0,200]);
ylim([0, 800])
%draw a line parallel to x axis to find on which iteration the
output
%reaches to tolerance value
yline(problem.toleranceValue,'-',{'Acceptable','Limit:',
num2str(problem.toleranceValue) });
legend(num2str(experimentsResults'), 'location', 'northeast');
grid on;
```

Function Call for Figure 2.A.1.2:

```
%% Effect of Population Size (nPop) and Probability of Crossover
(cP)


%save different values of nPop and pC inorder to do the experiment
nPopVariation = [100, 200, 300, 400, 500];
pCVariation = [0 0.3 0.5 0.7 1];

% use two for loops to do the experiment with various combinations
of
% parameters taken into consideration
for i=1:numel(nPopVariation)
    params.nPop = nPopVariation(i); %number of populations
    for j=1:numel(pCVariation)
        params.pC = pCVariation(i);  % Probability of crossover
        out = RunRGA(problem, params);
        Z(i,j)= out.minIterationToReachToleranceValue;
    end
end
```

Function Call for Figure 2.A.1.3:

```
%% Effect of Gamma and pC

%save different values of Gamma and pC inorder to do the experiment
gammaVariation = [0, 10, 15, 50, 100];
pCVariation = [0 0.3 0.5 0.8 1]; %in between 0 and 1

% use two for loops to do the experiment with various combinations
of
```

```
% parameters taken into consideration
for i=1:numel(gammaVariation)
        params.gamma = gammaVariation(i); % Spread factor for SBX
Crossover Operator
    for j=1:numel(pCVariation)
        params.pC = pCVariation(i);  % Probability of crossover
        out = RunRGA(problem, params);
        Z(i,j)= out.minIterationToReachToleranceValue;
    end
end


% Describing the attributes for the graph
[X,Y] = meshgrid(gammaVariation,pCVariation);
Z= transpose(Z);
surf(X,Y,Z);
title('Effect of Gamma and pC ');
xlabel('Spread factor for SBX Crossover Operator (Gamma)');
ylabel('Probability of crossover (pC)');
zlabel(["Minimum Iterations for Tolerance Value: "
num2str(problem.toleranceValue)]);
```

Function Call for Figure 2.A.1.4:

```
%% Effect of Population Size (nPop) and Probability of Mutation (mu)


%save different values of nPop and pC inorder to do the experiment
nPopVariation = [100, 200, 300, 400, 500];
muVariation = [0 0.3 0.5 0.7 1];

% use two for loops to do the experiment with various combinations
of
% parameters taken into consideration
for i=1:numel(nPopVariation)
    params.nPop = nPopVariation(i); %number of populations
    for j=1:numel(muVariation)
        params.mu = muVariation(i);  % Probability of Mutation
        out = RunRGA(problem, params);
        Z(i,j)= out.minIterationToReachToleranceValue;
    end
end
```

Function Call for Figure 2.A.1.5:

```
%% Effect of Sigma and mu

%save different values of Sigma and mu inorder to do the experiment
```

```
sigmaVariation = [0, 10, 15, 50, 100];
muVariation = [0 0.3 0.5 0.8 1]; % in between 0 and 1

% use two for loops to do the experiment with various combinations
of
% parameters taken into consideration
for i=1:numel(sigmaVariation)
    params.sigma = sigmaVariation(i);   % Distribution index for
Polynomial Mutation Operator
    for j=1:numel(muVariation)
        params.mu = muVariation(i);% 1/nVar  % Mutation probability
        out = RunRGA(problem, params);
        Z(i,j)= out.minIterationToReachToleranceValue;
    end
end
```

Function Call for Figure 2.A.1.8:

```
%% Effect of Static Penality Parameter R

%save different values of beta values for the experiment
RVariation = [0,2,10,15,50,100,500,1000];

%Do the experiment for number of Iteration
for i=1:numel(RVariation)
    params.R = RVariation(i);
    out= RunRGA(problem, params);
    % plot the result into the existing graph
    hold on
    plot(out.bestcost,"LineWidth",2)
    hold off
end
% Describing the attributes for the graph
title("Effect of R Value");
xlabel('Iterations');
ylabel('Best Cost');
% xlim([0,25]);
ylim([500, 600])
%draw a line parallel to x axis to find on which iteration the
output
%reaches to tolerance value
yline(problem.toleranceValue,'-',{'Acceptable','Limit:',
num2str(problem.toleranceValue) });
legend(num2str(RVariation'), 'location', 'northeast');
grid on;
```

2.B: Benchmark ConRGA-B

2.B.1 Outcomes of Tests

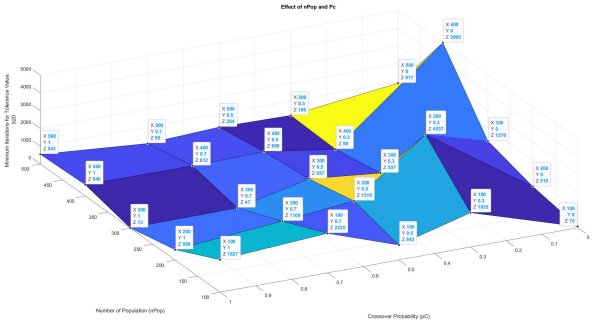


Figure 2.B.1.1

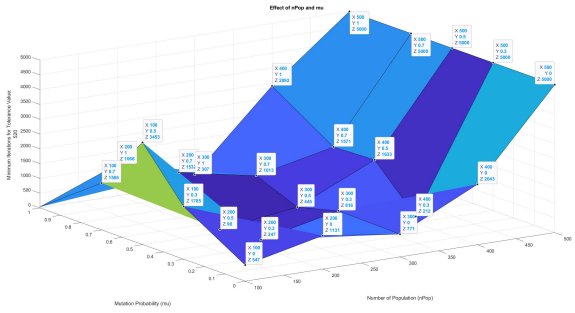


Figure 2.B.1.2

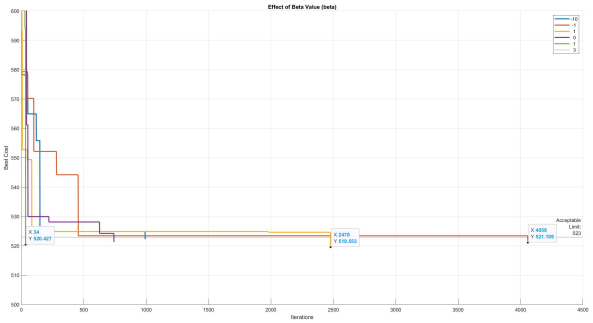


Figure 2.B.1.3

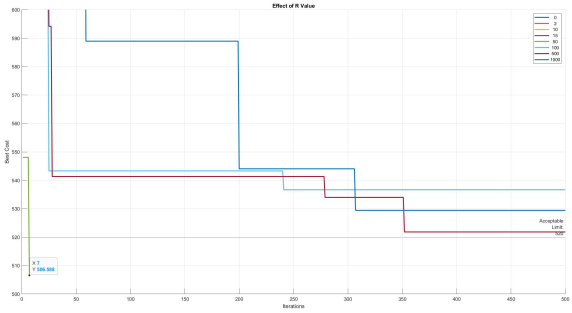


Figure 2.B.1.4

2.B.2 MATLAB Code Associated with the tests

Function Call for Figure 4.2.2.2:

```
%% RGA - Constraints Himmelblue Function and Static Penalty
clc; clear;

%% Problem Definition
problem.CostFunction = @(x1,x2) HimmelblauFunction(x1,x2); % Cost
Function
problem.Constraints = @(x1,x2,R) InequalityConstraints(x1,x2,R); %
Constraint
problem.FitnessValue = @(x1,x2,R) problem.CostFunction(x1,x2) +
problem.Constraints(x1,x2,R); % Fitness Value

problem.nVar = 2;                   % Number of Decision Variables
problem.VarMin = 0;                 % Lower Bound of Decision Variables
x1 and x2 are greater than 0
```

```
problem.VarMax = 6;              % Upper Bound of Decision Variables
problem.toleranceValue = 520; % this is the optimum solution when x1
= 3.763 and x2 = 4.947

%% RGA Parameters
params.MaxIt = 10000;
params.nPop = 40;
params.pC = 1;  % Probability of crossover
params.mu = 0.5; % = 1/nVar  % Mutation probability
params.beta = 1;
params.R = 100;  % Static Penalty Parameter
%% Calling RGA - FOR MEDIAN CALCULATION
% Run 10 experiments and find the median of those results
numberOfIterations =10;
experimentsResults = zeros(1,numberOfIterations);
for i=1:numberOfIterations
    out = RunRGA(problem, params);
    experimentsResults(i) = out.minIterationToReachToleranceValue;
    %plot the value in the graph if exists otherwise plot it in a
new graph
    hold on
    semilogy(out.bestcost,"LineWidth",2)
    grid on;
    hold off
end
disp(["Results found: " num2str(experimentsResults)])
medianOfResults = median(experimentsResults);
disp(["Median of the Results (Min iterations needed to reach the
tolerance value): " num2str(medianOfResults)])

%plot median value in the graph
hold on
semilogy(medianOfResults,0,"r*","MarkerSize",20)
hold off

%graph parameters
title("Median Result");
xlabel('Iterations');
ylabel('Best Cost');
ylim([0, 800])
%draw a line parallel to x axis to find on which iteration the
output
%reaches to tolerance value
yline(problem.toleranceValue,'-',{'Acceptable','Limit:',
num2str(problem.toleranceValue) });
legend(num2str(experimentsResults'), 'location', 'northeast');
grid on;
```

Appendix 3

3.A: Benchmark UnConPSO-A

3.A.1 Outcomes of Tests

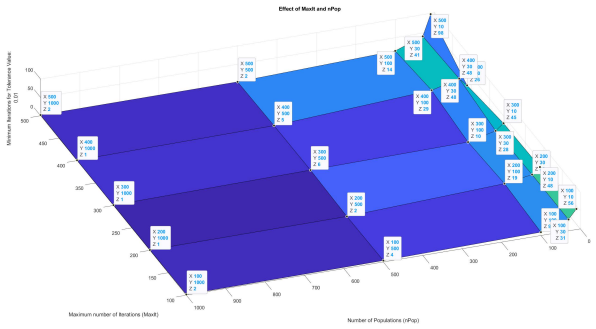


Figure 3.A.1.1

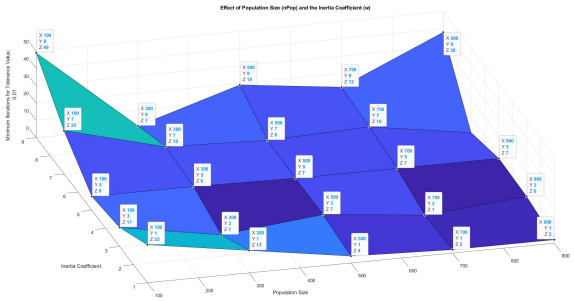


Figure 3.A.1.2

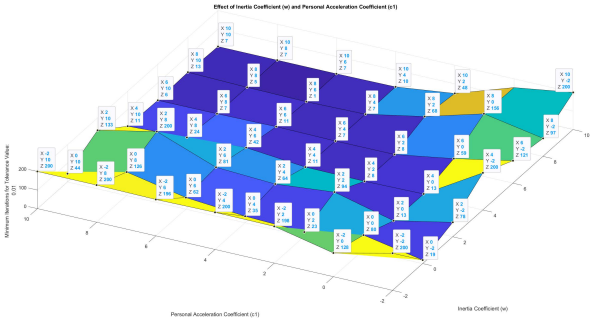


Figure 3.A.1.3

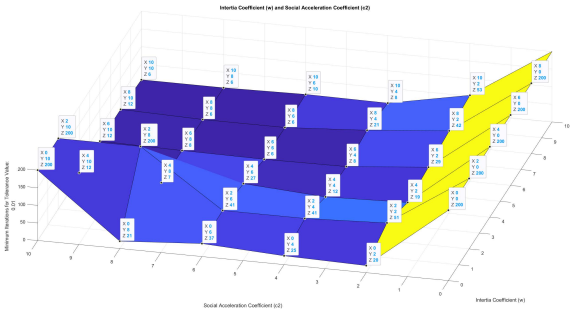


Figure 3.A.1.4

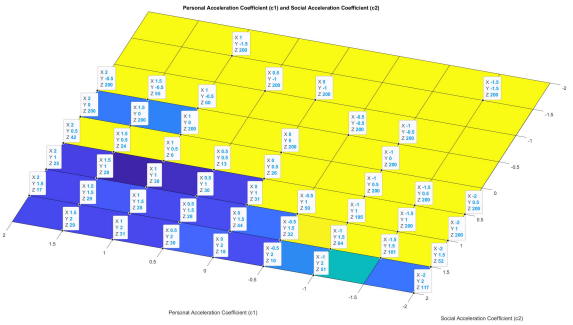


Figure 3.A.1.5

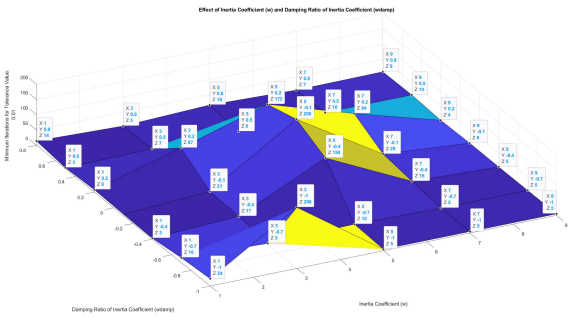


Figure 3.A.1.6

3.A.2 MATLAB Code Associated with the tests

Unconstraint PSO Implementation:

```
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;           % Inertia Coefficient
wdamp = params.wdamp;   % Damping Ratio of Inertia Coefficient
c1 = params.c1;         % Personal Acceleration Coefficient
c2 = params.c2;         % Social Acceleration Coefficient

% Define the velocity bounds
MaxVelocity = 0.2*(VarMax-VarMin);
MinVelocity = -MaxVelocity;

%Defining the paramters for Tolerance Value for Acceptace of the
Result
%(Best Cost for the objective Function)
toleranceValue = params.toleranceValue;
minIterationForToleranceValue = Inf;
ToleranceFlag = false;% for finding the minimum iteration to reach
the tolerance value

%% Initialization

% 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);

% Initialize Global Best
GlobalBest.Cost = inf;

% Initialize Population Members
for i=1:nPop
```

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

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

    % Evaluation
    particle(i).Cost =
CostFunction(particle(i).Position(1),particle(i).Position(2));

    % 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(1),particle(i).Position(2));

        % 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

    % Display Itertion Information
    % disp(['Iteration ' num2str(it) ': Best Cost = '
num2str(GlobalBest.Cost)]);

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

    %Compare the bestcost found in this iteration with the tolerance
value
    if abs(BestCosts(it)) < toleranceValue
        disp(["Tolerance value reached in " num2str(it) "th
iteration"]);
        out.pop = particle;
        out.BestSol = GlobalBest;
        out.BestCosts = BestCosts;
        out.minIterationToReachToleranceValue = it;
        return;
    end

    % Damping Inertia Coefficient
    w = w * wdamp;

end

out.pop = particle;
out.BestSol = GlobalBest;
out.BestCosts = BestCosts;
%the below line set is by maing an assumption that MaxIt is atleast
needed
%to reach the tolerance value with the given specification. its for
avoid
```

```
%the error of indefinit looing while finding the result
out.minIterationToReachToleranceValue = MaxIt;
end
```

Function Call for Figure 4.3.1.2:

```
%% PSO - Unconstraint RosenBrock Function

clc; clear;
%% Problem Definiton

problem.CostFunction = @(x,y) RosenBrockFunction(x,y);  % Cost
Function
problem.nVar = 2;       % Number of Unknown (Decision) Variables
problem.VarMin =  -5;  % Lower Bound of Decision Variables
problem.VarMax =  5;   % Upper Bound of Decision Variables
%% Parameters of PSO

    params.MaxIt = 200;        % Maximum Number of Iterations
params.nPop = 40;           % Population Size (Swarm Size)
params.w = 1;               % Intertia Coefficient
params.wdamp = 0.99;        % Damping Ratio of Inertia Coefficient
params.c1 = 1.5;              % Personal Acceleration Coefficient
params.c2 = 1.5;              % Social Acceleration Coefficient
params.toleranceValue = 10^-2; % which is the optimum solution for
this problme when x=1 and y=1

%% Calling PSO - FOR MEDIAN CALCULATION
% Run 10 experiments and find the median of those results
numberOfIterations =10;
experimentsResults = zeros(1,numberOfIterations);
for i=1:numberOfIterations
    %perform PSO
    out = PSO(problem, params);
    experimentsResults(i) = out.minIterationToReachToleranceValue;
    %plot the value in the graph if exists otherwise plot it in a
new graph
    hold on
    semilogy(out.BestCosts,"LineWidth",2)
    grid on;
    hold off
end
%display the result
disp(["Results found: " num2str(experimentsResults)])
%find the median
medianOfResults = median(experimentsResults);
disp(["Median of the Results (Min iterations needed to reach the
tolerance value): " num2str(medianOfResults)])
```

```
%plot median value in the graph
hold on
semilogy(medianOfResults,0,"r*","MarkerSize",20)
hold off
%add graph attributes
xlabel('Iterations');
ylabel('Best Cost');
title("Median of the result - PSO - RosenBrock Function
UnConstrainted")
xlim([0,50]);
ylim([0, 1])
yline(params.toleranceValue,'-',{'Acceptable','Limit:',
num2str(params.toleranceValue) });
legend(num2str(experimentsResults'), 'location', 'northeast');
```

Function Call for Figure 3.A.1.1:

```
%% Effect of MaxIt and nPop

%save different values of MaxIt and nPop inorder to do the
experiment
maxItVariation = [100, 200, 300, 400, 500];
nPopVariation = [10, 30, 100, 500, 1000];

% use two for loops to do the experiment with various combinations
of
% parameters taken into consideration
for i=1:numel(maxItVariation)
     % Intertia Coefficient (w)
    params.MaxIt = maxItVariation(i);
    for j=1:numel(nPopVariation)
        % Damping Ratio of Inertia Coefficient (wdamp)
        params.nPop = nPopVariation(j);
        out = PSO(problem, params);
        Z(i,j)= out.minIterationToReachToleranceValue;
    end
end
```

Function Call for Figure 3.A.1.2:

```
%% Effect of Population Size (nPop) and the Inertia Coefficient (w)

%save different values of nPop and w inorder to do the experiment
nPopVariation = [100:200:1000];
wVariation = [1:2:10];


% size of variation arrays
```

```
sizeOfnPopVariation = numel(nPopVariation);
sizeOfwVariation = numel(wVariation);

% use two for loops to do the experiment with various combinations
of nPop
% and w
for i=1:sizeOfnPopVariation
    % Population Size (Swarm Size)
    params.nPop = nPopVariation(i);
    for j=1:sizeOfwVariation
         % Intertia Coefficient
        params.w = wVariation(j);
        out = PSO(problem, params);
        Z(i,j)= out.minIterationToReachToleranceValue;
    end
end
```

Function Call for Figure 3.A.1.3:

```
%% Effect of Inertia Coefficient (w) and Personal Acceleration
Coefficient (c1)

%save different values of w and c1 inorder to do the experiment
wVariation = [-2:2:10];
c1Variation = [-2:2:10];

% use two for loops to do the experiment with various combinations
of w
% and c1
for i=1:numel(wVariation)
     % Intertia Coefficient
    params.w = wVariation(i);
    for j=1:numel(c1Variation)
        % Personnel Acceleration Coefficient
        params.c1 = c1Variation(j);
        out= PSO(problem, params);
        Z(i,j)= out.minIterationToReachToleranceValue;
    end
end
```

Function Call for Figure 3.A.1.5:

```
%% Personal Acceleration Coefficient (c1) and Social Acceleration
Coefficient (c2)

%save different values of c1 and c2 inorder to do the experiment
c1Variation = [-2:0.5:2];
c2Variation = [-2:0.5:2];
```

```
% use two for loops to do the experiment with various combinations
of c1
% and c2
for i=1:numel(c1Variation)
     % Personal Acceleration Coefficient (c1)
    params.c1 = c1Variation(i);
    for j=1:numel(c2Variation)
        % Social Acceleration Coefficient (c2)
        params.c2 = c2Variation(j);
        out = PSO(problem, params);
        Z(i,j)= out.minIterationToReachToleranceValue;
    end
end
```

3.B: Benchmark UnConPSO-B

3.B.1 Outcomes of Tests

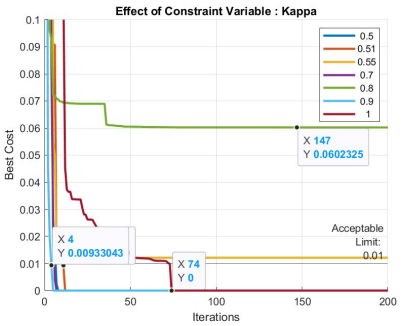


Figure 3.B.1.1

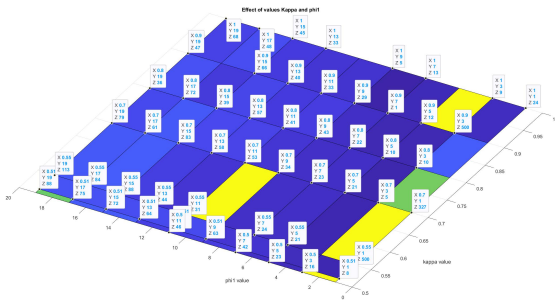


Figure 3.B.1.2

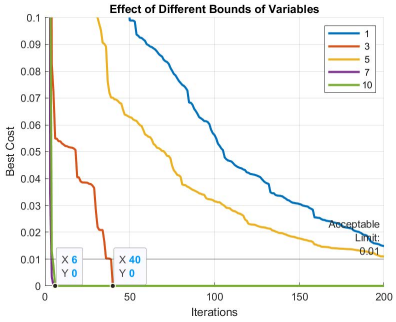


Figure 3.B.1.3

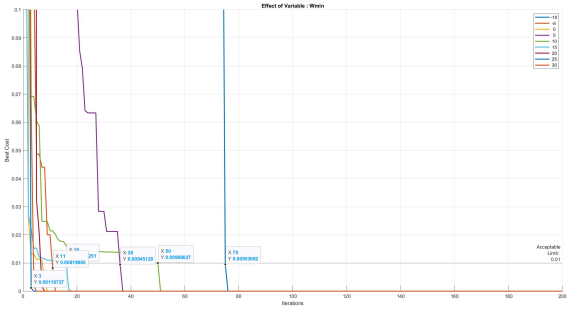


Figure 3.B.1.4

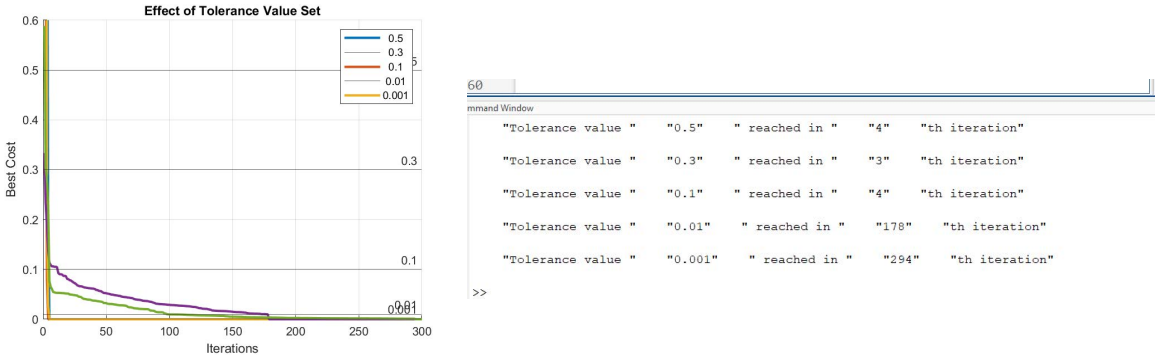


Figure 3.B.1.5

3.B.2 MATLAB Code Associated with the tests

PSO Implementation:

```
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)
wmax= params.wmax;
wmin = params.wmin;
w = params.w;           % Inertia Coefficient
c1 = params.c1;         % Personal Acceleration Coefficient
c2 = params.c2;         % Social Acceleration Coefficient

% Define the velocity bounds
MaxVelocity = 0.2*(VarMax-VarMin);
MinVelocity = -MaxVelocity;

%Defining the paramters for Tolerance Value for Acceptace of the
Result
%(Best Cost for the objective Function)
toleranceValue = params.toleranceValue;
minIterationForToleranceValue = Inf;
ToleranceFlag = false;% for finding the minimum iteration to reach
the tolerance value

%% Initialization

% 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);

% Initialize Global Best
GlobalBest.Cost = inf;

% Initialize Population Members
for i=1:nPop

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

    % Initialize Velocity
```

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

    % Evaluation
    particle(i).Cost =
CostFunction(particle(i).Position(1),particle(i).Position(2));

    % 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
    %assigning dynamic wdamp
    wdamp = (it/MaxIt)*(wmax-wmin);
    w = w * wdamp;
    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(1),particle(i).Position(2));

        % 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

    % Display Itertion Information
    % disp(['Iteration ' num2str(it) ': Best Cost = '
num2str(GlobalBest.Cost)]);

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

    %Compare the bestcost found in this iteration with the tolerance
value
    if BestCosts(it) < toleranceValue
        disp(["Tolerance value " num2str(toleranceValue) " reached
in " num2str(it) "th iteration"]);
        out.pop = particle;
        out.BestSol = GlobalBest;
        out.BestCosts = BestCosts;
        out.minIterationToReachToleranceValue = it;
        return;
    end
end

out.pop = particle;
out.BestSol = GlobalBest;
out.BestCosts = BestCosts;
out.minIterationToReachToleranceValue = MaxIt;
end
```

Function Call for Figure 4.3.2.2:

```
%% PSO - Unconstraint RosenBrock Function

clc; clear;
%% Problem Definiton
```

```
problem.CostFunction = @(x,y) RosenBrockFunction(x,y);  % Cost
Function
problem.nVar = 2;       % Number of Unknown (Decision) Variables
problem.VarMin =  -5;  % Lower Bound of Decision Variables
problem.VarMax =  5;   % Upper Bound of Decision Variables

%% 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.MaxIt = 1000;        % Maximum Number of Iterations
params.nPop = 40;           % Population Size (Swarm Size)
%adding dynamicity to w (Inertia Coefficient)
params.w = chi;             % Intertia Coefficient
%adding dynamicity to wdamp (Damping Ratio of Inertia Coefficient)
params.wmax = chi;          % Maximum value of Intertia Coefficient
params.wmin = 0.1;            % Minimum value of Intertia
Coefficient
params.c1 = chi*phi1;       % Personal Acceleration Coefficient
params.c2 = chi*phi2;       % Social Acceleration Coefficient

params.toleranceValue = 10^(-2); %tolerance value which can be
accepted as solution with the maximum error allowed

%% Calling PSO - FOR MEDIAN CALCULATION
% Run 10 experiments and find the median of those results
numberOfIterations =10;
experimentsResults = zeros(1,numberOfIterations);
for i=1:numberOfIterations
    out = PSO(problem, params);
    experimentsResults(i) = out.minIterationToReachToleranceValue;
    %plot the value in the graph if exists otherwise plot it in a
new graph
    hold on
    semilogy(out.BestCosts,"LineWidth",2)
    grid on;
    hold off
end
disp(["Results found: " num2str(experimentsResults)])
medianOfResults = median(experimentsResults);
disp(["Median of the Results (Min iterations needed to reach the
tolerance value): " num2str(medianOfResults)])

%plot median value in the graph
```

```
hold on
semilogy(medianOfResults,0,"r*","MarkerSize",20)
hold off

title("Median Result");
xlabel('Iterations');
ylabel('Best Cost');
xlim([0,50]);
ylim([0, 0.1])
yline(params.toleranceValue,'-',{'Acceptable','Limit:',
num2str(params.toleranceValue) });
legend(num2str(experimentsResults'), 'location', 'northeast');
```

Function Call for Figure 3.B.1.1:

```
%% Effect of Constraint Variable : Kappa

%different number of 'kappa' values
% kappaVariations = [1:0.05:1.5]; %values in between 1 and 1.5 -- B
kappaVariations = [0.50, 0.51, 0.55, 0.7, 0.8,0.9,1]; %values in
between
% 0 and 1 C


%Variable to store the number of Iterations take to reach the
minimum
%value (In Min One Problem, it is zero)
numberOfIterations = numel(kappaVariations);
%run the experiment for numberOfIterations times
for i=1:numberOfIterations
    % Parameters of PSO
    % Constriction Coefficients
    kappa = kappaVariations(i);
    phi1 = 2.05;
    phi2 = 2.05;
    phi = phi1 + phi2;
    chi = 2*kappa/abs(2-phi-sqrt(phi^2-4*phi));

    params.MaxIt = 500;        % Maximum Number of Iterations
    params.w = chi;             % Intertia Coefficient
    %adding dynamicity to wdamp (Damping Ratio of Inertia
Coefficient)
    params.wmax = chi;          % Maximum value of Intertia
Coefficient
    params.wmin = 0.1;            % Minimum value of Intertia
Coefficient
    params.c1 = chi*phi1;       % Personal Acceleration Coefficient
    params.c2 = chi*phi2;       % Social Acceleration Coefficient
    params.nPop = 40;           % Population Size (Swarm Size)
```

```
    out = PSO(problem, params);
    % plot the result into the existing graph
    hold on

semilogy(out.BestCosts,"LineWidth",2,'DisplayName',num2str(kappaVari
ations(i)));
    hold off
end
```

Appendix 4

4.A: Benchmark ConPSO-A

4.A.1 Outcomes of Tests

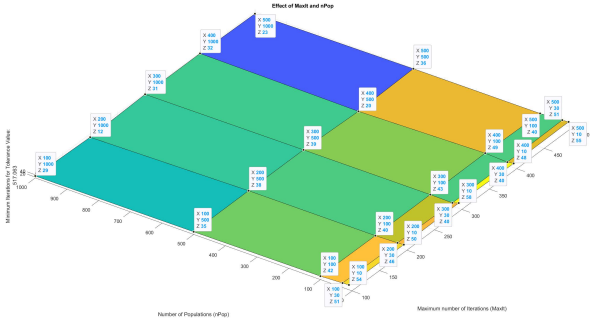


Figure 4.A.1.1

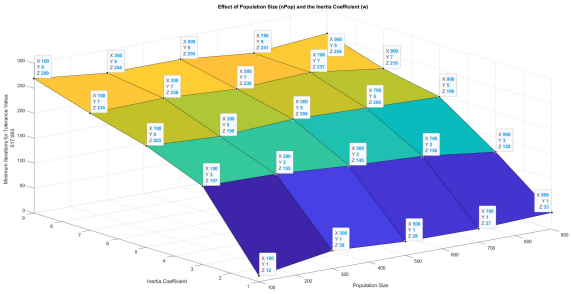


Figure 4.A.1.2

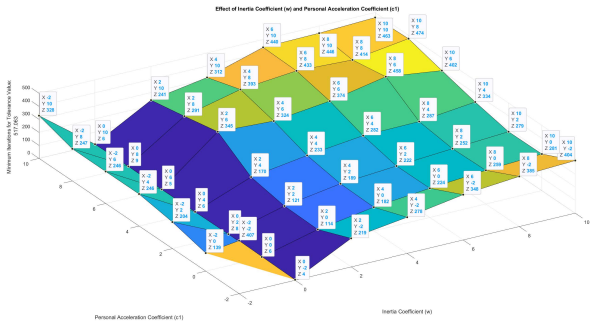


Figure 4.A.1.3

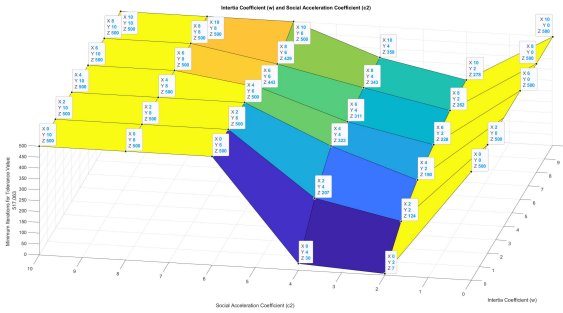


Figure 4.A.1.4

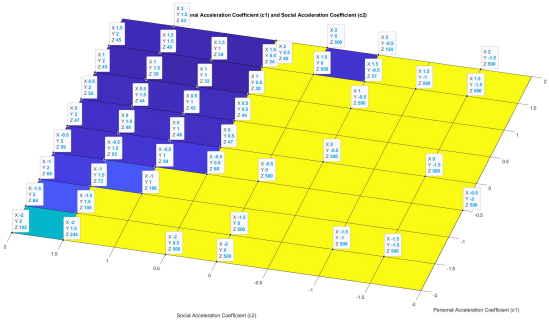


Figure 4.A.1.5

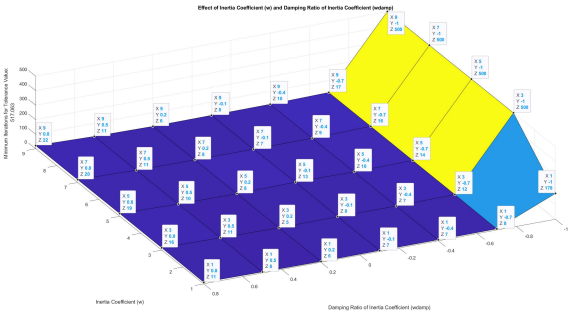


Figure 4.A.1.6

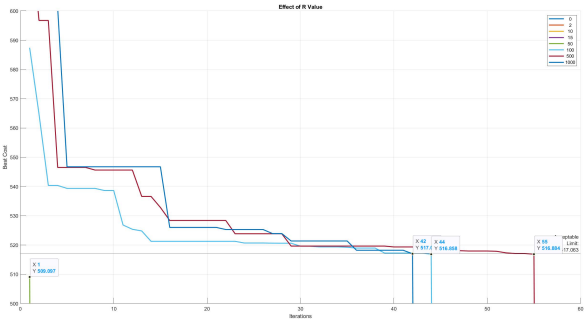


Figure 4.A.1.7

4.A.2 MATLAB Code Associated with the tests

Constraint PSO Implementation:

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


%% Problem Definiton

CostFunction = problem.CostFunction; % Cost Function
Constraints = problem.Constraints; % Constraint function
FitnessValue = problem.FitnessValue; %Fitness 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;            % Inertia Coefficient
wdamp = params.wdamp;
c1 = params.c1;          % Personal Acceleration Coefficient
c2 = params.c2;          % Social Acceleration Coefficient
toleranceValue = params.toleranceValue;
MaxVelocity = 0.2*(VarMax-VarMin);
MinVelocity = -MaxVelocity;

R = params.R;

% Define the velocity bounds
MaxVelocity = 0.2*(VarMax-VarMin);
MinVelocity = -MaxVelocity;

%Defining the paramters for Tolerance Value for Acceptace of the
Result
%(Best Cost for the objective Function)
toleranceValue = params.toleranceValue;
minIterationForToleranceValue = Inf;
ToleranceFlag = false;% for finding the minimum iteration to reach
the tolerance value

%% Initialization

% 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);

% Initialize Global Best
GlobalBest.Cost = inf;

% Initialize Population Members
for i=1:nPop

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

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

    % Evaluation
    particle(i).Cost =
FitnessValue(particle(i).Position(1),particle(i).Position(2),R);

    % 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 =
FitnessValue(particle(i).Position(1),particle(i).Position(2),R);

        % 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

    % Display Itertion Information
    % disp(['Iteration ' num2str(it) ': Best Cost = '
num2str(GlobalBest.Cost)]);

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

    %Compare the bestcost found in this iteration with the tolerance
value
    if abs(BestCosts(it)) < toleranceValue
        disp(["Tolerance value reached in " num2str(it) "th
iteration"]);
        out.pop = particle;
        out.BestSol = GlobalBest;
        out.BestCosts = BestCosts;
        out.minIterationToReachToleranceValue = it;
        return;
    end
```

```
    % Damping Inertia Coefficient
    w = w * wdamp;

end

out.pop = particle;
out.BestSol = GlobalBest;
out.BestCosts = BestCosts;
%the below line set is by maing an assumption that MaxIt is atleast
needed
%to reach the tolerance value with the given specification. its for
avoid
%the error of indefinit looing while finding the result
out.minIterationToReachToleranceValue = MaxIt;
end
```

Function Call for Figure 4.4.1.2:

```
%% PSO - Constraint Himmelblu Funciton with Static Penalty
clc; clear;

%% Problem Definition
problem.CostFunction = @(x1,x2) HimmelblauFunction(x1,x2); % Cost
Function
problem.Constraints = @(x1,x2,R) InequalityConstraints(x1,x2,R); %
Constraint
problem.FitnessValue = @(x1,x2,R) problem.CostFunction(x1,x2) +
problem.Constraints(x1,x2,R); % Fitness Value
problem.nVar = 2;               % Number of Decision Variables
problem.VarMin = 0;             % Lower Bound of Decision Variables
x1 and x2 are greater than 0
problem.VarMax = 6;              % Upper Bound of Decision Variables

%% Parameters of PSO

params.MaxIt = 500;         % Maximum Number of Iterations
params.nPop = 50;            % Population Size (Swarm Size)
params.w = 1;                % Intertia Coefficient
params.wdamp = 0.99;         % Damping Ratio of Inertia Coefficient
params.c1 = 1.5;               % Personal Acceleration Coefficient
params.c2 = 1.5;               % Social Acceleration Coefficient

params.R = 100;              % Static Penalty Parameter

params.toleranceValue = 520 ; % this is the optimum solution, but
used to demonstrate the convergence for this problem

%% Calling PSO - FOR MEDIAN CALCULATION
% Run 10 experiments and find the median of those results
```

```
numberOfIterations =10;
experimentsResults = zeros(1,numberOfIterations);
for i=1:numberOfIterations
    %perform PSO
    out = PSO(problem, params);
    experimentsResults(i) = out.minIterationToReachToleranceValue;
    %plot the value in the graph if exists otherwise plot it in a
new graph
    hold on
    semilogy(out.BestCosts,"LineWidth",2)
    grid on;
    hold off
end
%display the result
disp(["Results found: " num2str(experimentsResults)])
%find the median
medianOfResults = median(experimentsResults);
disp(["Median of the Results (Min iterations needed to reach the
tolerance value): " num2str(medianOfResults)])

%plot median value in the graph
hold on
semilogy(medianOfResults,0,"r*","MarkerSize",20)
hold off
%add graph attributes
xlabel('Iterations');
ylabel('Best Cost');
title("Median of the result - PSO - RosenBrock Function
UnConstrainted")
xlim([0,50]);
ylim([0, 600])
yline(params.toleranceValue,'-',{'Acceptable','Limit:',
num2str(params.toleranceValue) });
legend(num2str(experimentsResults'), 'location', 'northeast');
```

Function Call for Figure 4.A.1.2:

```
%% PSO - Constraint Himmelblu Funciton with Static Penalty
clc; clear;

%% Problem Definition
problem.CostFunction = @(x1,x2) HimmelblauFunction(x1,x2); % Cost
Function
problem.Constraints = @(x1,x2,R) InequalityConstraints(x1,x2,R); %
Constraint
problem.FitnessValue = @(x1,x2,R) problem.CostFunction(x1,x2) +
problem.Constraints(x1,x2,R); % Fitness Value
problem.nVar = 2;              % Number of Decision Variables
```

```
problem.VarMin = 0;            % Lower Bound of Decision Variables
x1 and x2 are greater than 0
problem.VarMax = 6;             % Upper Bound of Decision Variables

%% Parameters of PSO

params.MaxIt = 500;        % Maximum Number of Iterations
params.nPop = 50;           % Population Size (Swarm Size)
params.w = 1;               % Intertia Coefficient
params.wdamp = 0.99;        % Damping Ratio of Inertia Coefficient
params.c1 = 1.5;              % Personal Acceleration Coefficient
params.c2 = 1.5;              % Social Acceleration Coefficient

params.R = 100;             % Static Penalty Parameter
%Generation numbers needed to be plot as a graph
params.genNumber = [1,30,100];
params.toleranceValue = 517.063 ; % this is the optimum solution,
but used to demonstrate the convergence for this problem
%% Effect of Population Size (nPop) and the Inertia Coefficient (w)

%save different values of nPop and w inorder to do the experiment
nPopVariation = [100:200:1000];
wVariation = [1:2:10];


% size of variation arrays
sizeOfnPopVariation = numel(nPopVariation);
sizeOfwVariation = numel(wVariation);

% use two for loops to do the experiment with various combinations
of nPop
% and w
for i=1:sizeOfnPopVariation
    % Population Size (Swarm Size)
    params.nPop = nPopVariation(i);
    for j=1:sizeOfwVariation
         % Intertia Coefficient
        params.w = wVariation(j);
        out = PSO(problem, params);
        Z(i,j)= out.minIterationToReachToleranceValue;
    end
end

% Describing the attributes for the graph
[X,Y] = meshgrid(nPopVariation,wVariation);
Z= transpose(Z)
surf(X,Y,Z)
title('Effect of Population Size (nPop) and the Inertia Coefficient
(w)')
```

```
xlabel('Population Size');
ylabel('Inertia Coefficient');
zlabel(["Minimum Iterations for Tolerance Value: "
num2str(params.toleranceValue)]);
```

4.B: Benchmark ConPSO-B

4.B.1 Outcomes of Tests

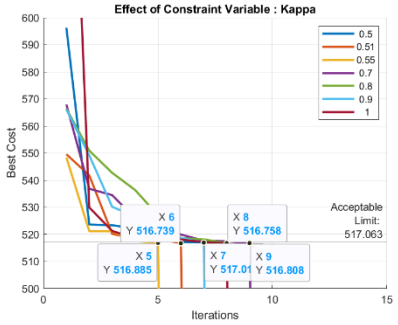


Figure 4.B.1.1

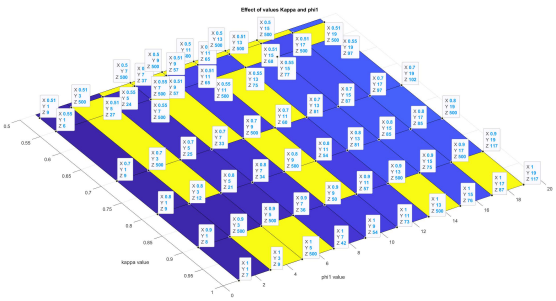


Figure 4.B.1.2

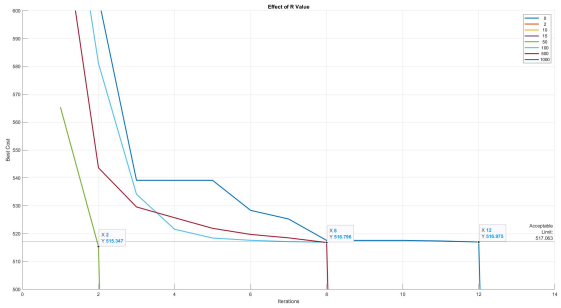


Figure 4.B.1.3

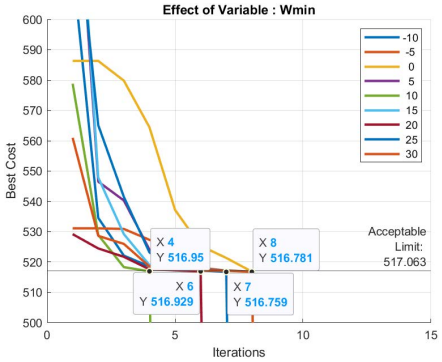


Figure 4.B.1.4

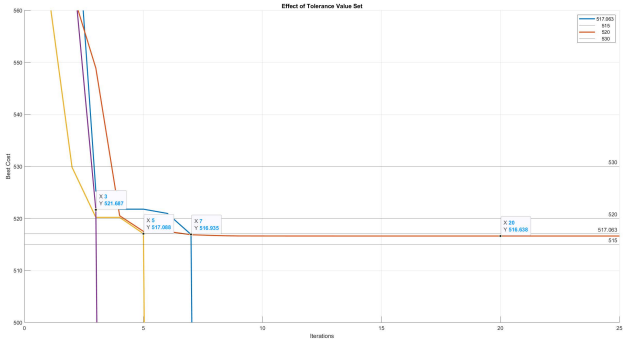


Figure 4.B.1.5

4.B.2 MATLAB Code Associated with the tests

Function Call for Figure 4.4.2.2:

```
%% PSO - Constraint Himmelblu Funciton with Static Penalty
clc; clear;
```

```
%% Problem Definition
problem.CostFunction = @(x1,x2) HimmelblauFunction(x1,x2); % Cost
Function
problem.Constraints = @(x1,x2,R) InequalityConstraints(x1,x2,R); %
Constraint
problem.FitnessValue = @(x1,x2,R) problem.CostFunction(x1,x2) +
problem.Constraints(x1,x2,R); % Fitness Value
problem.nVar = 2;               % Number of Decision Variables
problem.VarMin = 0;             % Lower Bound of Decision Variables
x1 and x2 are greater than 0
problem.VarMax = 6;              % Upper Bound of Decision Variables

%% 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.MaxIt = 500;        % Maximum Number of Iterations
params.nPop = 40;           % Population Size (Swarm Size)
%adding dynamicity to w (Inertia Coefficient)
params.w = chi;             % Intertia Coefficient
%adding dynamicity to wdamp (Damping Ratio of Inertia Coefficient)
params.wmax = chi;          % Maximum value of Intertia Coefficient
params.wmin = 0.1;            % Minimum value of Intertia
Coefficient
params.c1 = chi*phi1;       % Personal Acceleration Coefficient
params.c2 = chi*phi2;       % Social Acceleration Coefficient

params.R = 100;             % Static Penalty Parameter
params.toleranceValue = 517.063 ; % this is the optimum solution,
but used to demonstrate the convergence for this problem

%% Calling PSO - FOR MEDIAN CALCULATION
% Run 10 experiments and find the median of those results
numberOfIterations =10;
experimentsResults = zeros(1,numberOfIterations);
for i=1:numberOfIterations
    out = PSO(problem, params);
    experimentsResults(i) = out.minIterationToReachToleranceValue;
    %plot the value in the graph if exists otherwise plot it in a
new graph
    hold on
    semilogy(out.BestCosts,"LineWidth",2)
    grid on;
    hold off
```

```
end
disp(["Results found: " num2str(experimentsResults)])
medianOfResults = median(experimentsResults);
disp(["Median of the Results (Min iterations needed to reach the
tolerance value): " num2str(medianOfResults)])

%plot median value in the graph
hold on
semilogy(medianOfResults,0,"r*","MarkerSize",20)
hold off

title("Median Result");
xlabel('Iterations');
ylabel('Best Cost');
xlim([0,50]);
ylim([0,800])
yline(params.toleranceValue,'-',{'Acceptable','Limit:',
num2str(params.toleranceValue) });
legend(num2str(experimentsResults'), 'location', 'northeast');
```

Function Call for Figure 4.B.1.1:

```
%% Effect of Constraint Variable : Kappa

%different number of 'kappa' values
% kappaVariations = [1:0.05:1.5]; %values in between 1 and 1.5 -- B
kappaVariations = [0.50, 0.51, 0.55, 0.7, 0.8,0.9,1]; %values in
between
% 0 and 1 C

%Variable to store the number of Iterations take to reach the
minimum
%value (In Min One Problem, it is zero)
numberOfIterations = numel(kappaVariations);
%run the experiment for numberOfIterations times
for i=1:numberOfIterations
    % Parameters of PSO
    % Constriction Coefficients
    kappa = kappaVariations(i);
    % 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.MaxIt = 500;        % Maximum Number of Iterations
    params.nPop = 40;           % Population Size (Swarm Size)
```

```
    %adding dynamicity to w (Inertia Coefficient)
    params.w = chi;             % Intertia Coefficient
    %adding dynamicity to wdamp (Damping Ratio of Inertia
Coefficient)
    params.wmax = chi;          % Maximum value of Intertia
Coefficient
    params.wmin = 0.1;            % Minimum value of Intertia
Coefficient
    params.c1 = chi*phi1;       % Personal Acceleration Coefficient
    params.c2 = chi*phi2;       % Social Acceleration Coefficient

    out = PSO(problem, params);
    % plot the result into the existing graph
    hold on

semilogy(out.BestCosts,"LineWidth",2,'DisplayName',num2str(kappaVari
ations(i)));
    hold off
end
```

Function Call for Figure 4.B.1.3:

```
%% Effect of Static Penality Parameter R

%save different values of beta values for the experiment
RVariation = [0,2,10,15,50,100,500,1000];

%Do the experiment for number of Iteration
for i=1:numel(RVariation)
    params.R = RVariation(i);
    out= PSO(problem, params);
    % plot the result into the existing graph
    hold on
    plot(out.BestCosts,"LineWidth",2)
    hold off
end
```

Appendix 5

5.A: Benchmark UnConDE-A (DE rand/1/bin)

5.A.1 Outcomes of Tests

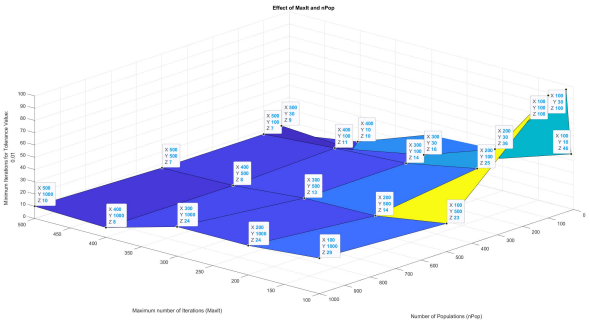


Figure 5.A.1.1

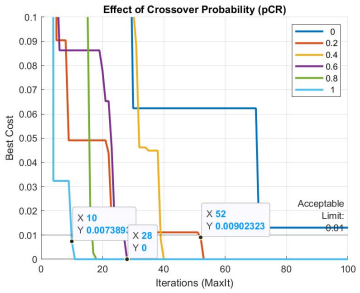


Figure 5.A.1.2

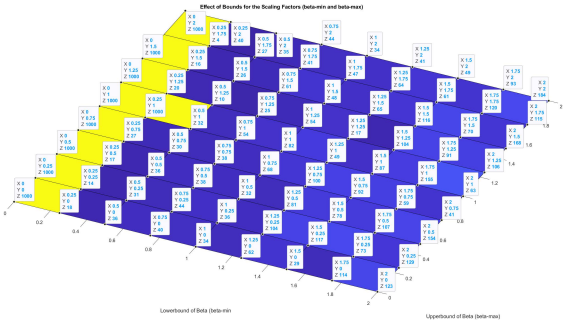


Figure 5.A.1.3

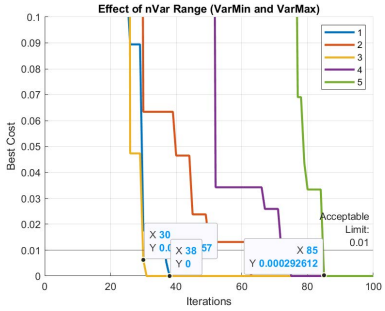


Figure 5.A.1.4

5.A.2 MATLAB Code Associated with the tests

Unconstraint DE Implementation:

```
%% DE /rand/1/bin
function out = DE(problem, params)
CostFunction = problem.CostFunction; % Cost Function
VarSize = [1 problem.nVar];     % Decision Variables Matrix Size
VarMin = problem.VarMin;            % Lower Bound of Decision
Variables
VarMax = problem.VarMax;             % Upper Bound of Decision
Variables
toleranceValue = problem.toleranceValue; % tolerance value at which
the solution is acceptable with the maximum error possible

%% DE Parameters

MaxIt = params.MaxIt;       % Maximum Number of Iterations
```

```
nPop = params.nPop;           % Population Size
beta_min = params.beta_min;     % Lower Bound of Scaling Factor (0)
beta_max = params.beta_max;     % Upper Bound of Scaling Factor (2)
pCR = params.pCR;          % Crossover Probability


%% Initialization

empty_individual.Position = [];
empty_individual.Cost = [];

BestSol.Cost = inf;

pop = repmat(empty_individual, nPop, 1);
%sample solution space
for i = 1:nPop
    %sample creation
    pop(i).Position = unifrnd(VarMin, VarMax, VarSize);
    %sample solution creation
    pop(i).Cost =
CostFunction(pop(i).Position(1),pop(i).Position(2));

    if pop(i).Cost<BestSol.Cost
        BestSol = pop(i);
    end

end
%inilialize the BestCost of all generations into zeros
BestCost = zeros(MaxIt, 1);
%% DE Main Loop

for it = 1:MaxIt

    for i = 1:nPop
        %ith individual
        x = pop(i).Position;
        %create a random arrangement (reorder the population
numbers)
        A = randperm(nPop);
        %Note: we can do this with rand, but we want all numbers
different

        %remove the i index from this since we dont want that to
        %be target vector
        A(A == i) = [];

        a = A(1);   % Target Vector index
        b = A(2);   % Random Vector 1 index
        c = A(3);   % Random Vector 2 index
```

```
        % Mutant Vector
        beta = unifrnd(beta_min, beta_max, VarSize);

        %y is the mutant vector
        y = pop(a).Position + beta.*(pop(b).Position -
pop(c).Position);
        y = max(y, VarMin);
        y = min(y, VarMax);

        % Trial Vector (Crossover between Target Vector x and Mutant
Vector y)
        z = zeros(size(x));
        j0 = randi([1 numel(x)]);
        for j = 1:numel(x)
            if j == j0 || rand <= pCR
                z(j) = y(j);%from the mutant vector
            else
                z(j) = x(j);%from the target vector
            end
        end

        NewSol.Position = z;
        NewSol.Cost =
CostFunction(NewSol.Position(1),NewSol.Position(2));

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

    end

    % Update Best Cost
    BestCost(it) = BestSol.Cost;

    % Show Iteration Information
    disp(['Iteration ' num2str(it) ': Best Cost = '
num2str(BestCost(it))]);
    % Comparing the best cost with the tolerance value
    % This code has been added for improving efficiency of the code
    if BestCost(it) <= toleranceValue
        disp(['Tolerance value has been reached at generation: '
num2str(it)]);
        %return from the function
        out.BestCost = BestCost;
```

```
        out.BestSol =BestSol;
        out.minIterationToReachToleranceValue = it;
        return;
    end

end

out.BestCost = BestCost;
out.BestSol =BestSol;
out.minIterationToReachToleranceValue =MaxIt;
end
```

Function Call for Figure 4.5.1.2:

```
%% DE /rand/1/bin RosenBrock Function Unconstrainted
clc; clear;
%% Problem Definition

problem.CostFunction = @(x,y) RosenBrockFunction(x,y);    % Cost
Function
problem.nVar = 2;               % Number of Decision Variables
problem.VarMin = -5;             % Lower Bound of Decision Variables
problem.VarMax = 5;              % Upper Bound of Decision Variables
problem.toleranceValue = 10^-2; % tolerance value at which the
solution is acceptable with the maximum error possible

%% DE Parameters
params.MaxIt = 1000;       % Maximum Number of Iterations
params.nPop = 40;          % Population Size
params.beta_min = 0.2;     % Lower Bound of Scaling Factor (0)
params.beta_max = 0.8;     % Upper Bound of Scaling Factor (2)
params.pCR = 0.5;          % Crossover Probability

%% Calling DE
% number of iterations required
noOfTests = 10;
% array to store the result of each test, initially all assign to
infinity
minIterationsFromEachTest = inf(1,noOfTests);
%test the experiment for noOfTests times
for i=1:noOfTests
    out = DE(problem, params);

minIterationsFromEachTest(i)=out.minIterationToReachToleranceValue;
    %plot the graph for this test result
    hold on
    plot(out.BestCost,"LineWidth",2)
    hold off
end
```

```
%% Show Results
%Calculate Median value
medianOfAllSolutions = median(minIterationsFromEachTest);

%plot median value in the graph
hold on
plot(medianOfAllSolutions,0,"r*","MarkerSize",20)
hold off
title("The Median of 10 Tests - Benchmark A")
%attributes to the graph
xlabel('Iteration');
ylabel('Best Cost');
yline(problem.toleranceValue,'-',{'Acceptable','Limit'});
xlim([0,100]);
ylim([0,0.1]);
%draw a line parallel to x axis to find on which iteration the
output
%reaches to tolerance value
yline(problem.toleranceValue,'-',{'Acceptable','Limit:',
num2str(problem.toleranceValue) });

grid on;
```

Function Call for Figure 5.A.1.1:

```
%% Effect of MaxIt and nPop

%save different values of MaxIt and nPop inorder to do the
experiment
maxItVariation = [100, 200, 300, 400, 500];
nPopVariation = [10, 30, 100, 500, 1000];

% use two for loops to do the experiment with various combinations
of
% parameters taken into consideration
for i=1:numel(maxItVariation)
        params.MaxIt = maxItVariation(i);       % Maximum Number of
Iterations
    for j=1:numel(nPopVariation)
        params.nPop = nPopVariation(i);          % Population Size
        out = DE(problem, params);
        Z(i,j)= out.minIterationToReachToleranceValue;
    end
end


% Describing the attributes for the graph
[X,Y] = meshgrid(maxItVariation,nPopVariation);
```

```
Z= transpose(Z)
surf(X,Y,Z)
title('Effect of MaxIt and nPop ');
xlabel('Maximum number of Iterations (MaxIt)');
ylabel('Number of Populations (nPop)');
zlabel(["Minimum Iterations for Tolerance Value: "
num2str(problem.toleranceValue)]);
```

Function Call for Figure 5.A.1.2:

```
%% Calling DE for different pCR values to see the effect of it in
the Result
%different number of 'pCR' values
pCRVariation = [0:0.2:1];

%Variable to store the number of Iterations take to reach the
tolerance
%value
numberOfIterations = numel(pCRVariation);
%repeat the test for numberOfIterations times
for i=1:numberOfIterations
    %assign value to pCR
    params.pCR = pCRVariation(i);
    out= DE(problem, params);
    % plot the result into the existing graph else create a new
graph
    hold on
    semilogy(out.BestCost,"LineWidth",2);
    hold off
end

% Describing the attributes for the graph
title("Effect of Crossover Probability (pCR)")
xlabel('Iterations (MaxIt)');
ylabel('Best Cost');
%for the purpous of seeing the change in each experiment, xlim is
used to
%get a more close view
ylim([0 0.1])
xlim([0 100]);
%draw a line parallel to x axis to find on which iteration the
output
%reaches to tolerance value
yline(problem.toleranceValue,'-',{'Acceptable','Limit:',
num2str(problem.toleranceValue) });
grid on;
legend(num2str(pCRVariation.'), 'location', 'northeast');
```

Function Call for Figure 5.A.1.3:

```
%% Effect of beta(beta_min and beta_max) in the result

%save different values of beta_min and beta_max
boundVariations = [0:0.25:2];% for figure-A
% boundVariations = [-2,-1,3,4];% for figure-C

% size of variation arrays for the loop generation
sizeOfbetaMinVariation = numel(boundVariations);
sizeOfbetaMaxVariation = numel(boundVariations);

% use two for loops to do the experiment with various combinations
for i=1:sizeOfbetaMaxVariation
    % Lowerbound for scaling factor-beta (bata_min)
    params.beta_min = boundVariations(i);
    for j=1:sizeOfbetaMaxVariation
        % Upperbound for scaling factor-beta (bata_max)
        params.beta_max = boundVariations(i);
        out = DE(problem, params);
        Z(i,j)= out.minIterationToReachToleranceValue;
    end
end

% Describing the attributes for the graph
[X,Y] = meshgrid(boundVariations,boundVariations);
Z= transpose(Z)
surf(X,Y,Z)
title('Effect of Bounds for the Scaling Factors (beta-min and beta-
max)')
xlabel('Lowerbound of Beta (beta-min');
ylabel('Upperbound of Beta (beta-max)');
zlabel(["Minimum Iterations for Tolerance Value: "
num2str(problem.toleranceValue)]);
```

5.B: Benchmark UnConDE-B – Variant 1 (DE best/1/bin)

5.B.1 Outcomes of Tests

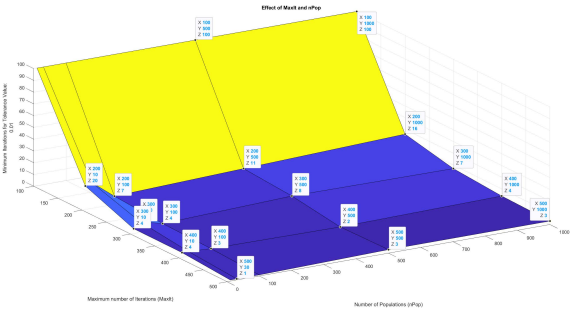


Figure 5.B.1.1

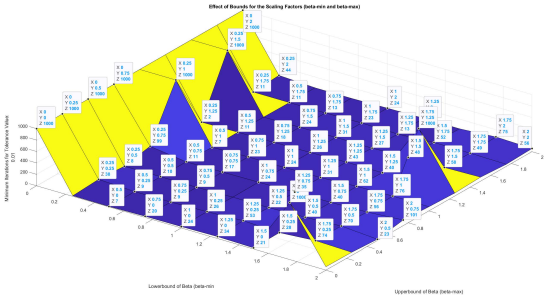


Figure 5.B.1.2

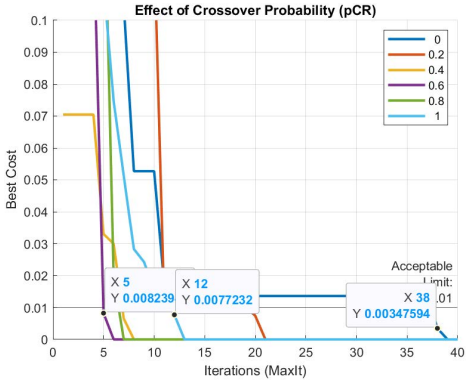


Figure 5.B.1.3

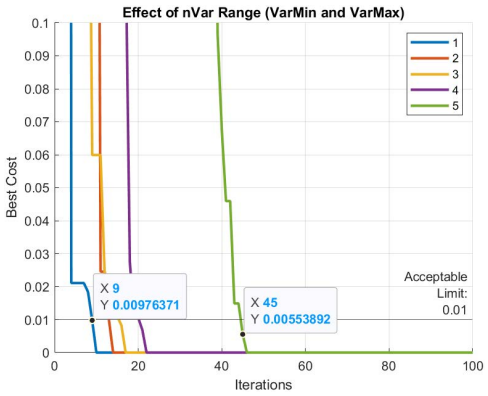


Figure 5.B.1.4

5.B.2 MATLAB Code Associated with the tests

Unconstraint DE Implementation:

```
%% DE /best/1/bin
function out = DE_V1(problem, params)
CostFunction = problem.CostFunction; % Cost Function
VarSize = [1 problem.nVar];     % Decision Variables Matrix Size
VarMin = problem.VarMin;            % Lower Bound of Decision
Variables
VarMax = problem.VarMax;             % Upper Bound of Decision
Variables
toleranceValue = problem.toleranceValue; % tolerance value at which
the solution is acceptable with the maximum error possible

%% DE Parameters

MaxIt = params.MaxIt;       % Maximum Number of Iterations
nPop = params.nPop;          % Population Size
beta_min = params.beta_min;     % Lower Bound of Scaling Factor (0)
beta_max = params.beta_max;     % Upper Bound of Scaling Factor (2)
```

```
pCR = params.pCR;          % Crossover Probability


%% Initialization

empty_individual.Position = [];
empty_individual.Cost = [];

BestSol.Cost = inf;

pop = repmat(empty_individual, nPop, 1);
%sample solution space
for i = 1:nPop
    %sample creation
    pop(i).Position = unifrnd(VarMin, VarMax, VarSize);
    %sample solution creation
    pop(i).Cost =
CostFunction(pop(i).Position(1),pop(i).Position(2));

    if pop(i).Cost<BestSol.Cost
        BestSol = pop(i);
    end

end
%inilialize the BestCost of all generations into zeros
BestCost = zeros(MaxIt, 1);

%% DE Main Loop

for it = 1:MaxIt
    %assign the best solution as best solution in the current
generation
    %which is used as base vector for mutate vector formation
    bestSolutionInPreviousGen = BestSol;
    %process
    for i = 1:nPop
        %ith individual
        x = pop(i).Position;
        %create a random arrangement (reorder the population
numbers)
        A = randperm(nPop);
        %Note: we can do this with rand, but we want all numbers
different

        %remove the i index from this since we dont want that to
        %be target vector
        A(A == i) = [];

        a = A(1);   % Random Vector 1 index
```

```
        b = A(2);   % Random Vector 2 index

        % Mutant Vector (Mutation)
        beta = unifrnd(beta_min, beta_max, VarSize);

        %y is the mutant vector - best is used as base vector
        y = bestSolutionInPreviousGen.Position +
beta.*(pop(a).Position - pop(b).Position);
        y = max(y, VarMin);
        y = min(y, VarMax);

        % Trial Vector (Crossover between Target Vector x and Mutant
Vector y)
        z = zeros(size(x));
        j0 = randi([1 numel(x)]);
        for j = 1:numel(x)
            if j == j0 || rand <= pCR
                z(j) = y(j);%from the mutant vector
            else
                z(j) = x(j);%from the target vector
            end
        end

        NewSol.Position = z;
        NewSol.Cost =
CostFunction(NewSol.Position(1),NewSol.Position(2));

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

    end

    % Update Best Cost
    BestCost(it) = BestSol.Cost;

    % Show Iteration Information
    disp(['Iteration ' num2str(it) ': Best Cost = '
num2str(BestCost(it))]);

    if BestCost(it) <= toleranceValue
        disp(['Tolerance value has been reached at generation: '
num2str(it)]);
        out.BestCost = BestCost;
        out.BestSol =BestSol;
```

```
        out.minIterationToReachToleranceValue = it;
        return;
    end

end
out.BestCost = BestCost;
out.BestSol =BestSol;
%assing number of iteration to reach minimum tolerance value as
MaxIt since the generations of MaxIt has
%not reached to the required result, this is done as part of
plotting the
%graph without any error
out.minIterationToReachToleranceValue =MaxIt;
end
```

Function Call for Figure 4.5.2.1.2:

```
%% Calling DE
% number of iterations required
noOfTests = 10;
% array to store the result of each test, initially all assign to
infinity
minIterationsFromEachTest = inf(1,noOfTests);
%test the experiment for noOfTests times
for i=1:noOfTests
    out = DE_V1(problem, params);

minIterationsFromEachTest(i)=out.minIterationToReachToleranceValue;
    %plot the graph for this test result
    hold on
    plot(out.BestCost,"LineWidth",2)
    hold off
end
```

5.C: Benchmark UnConDE-B – Variant 2 (DE best/2/bin)

5.C.1 Outcomes of Tests

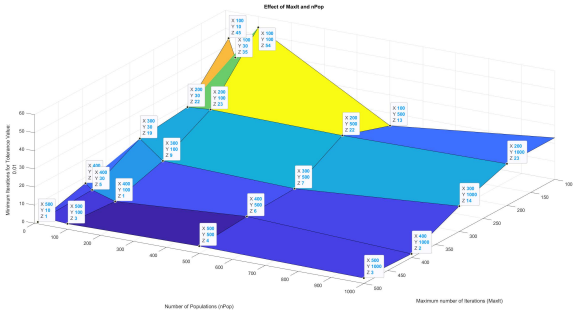


Figure 5.C.1.1

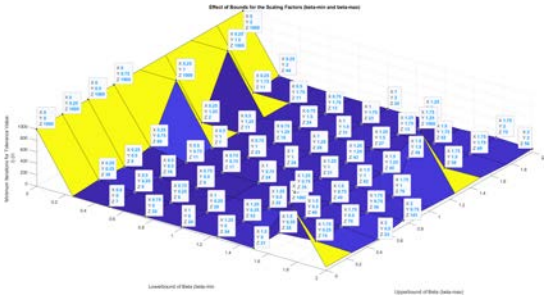


Figure 5.C.1.2

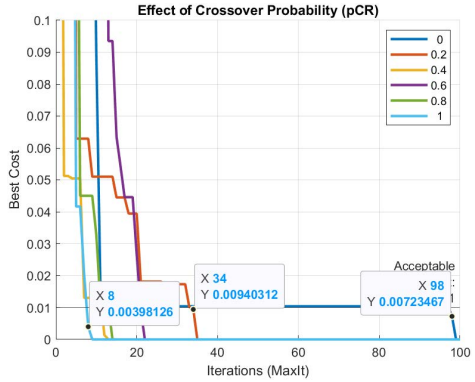


Figure 5.C.1.3

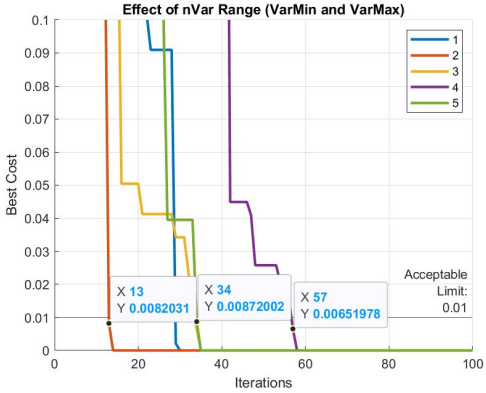


Figure 5.C.1.4

5.C.2 MATLAB Code Associated with the tests

Unconstraint DE Implementation:

```
%% DE /best/2/bin
function out = DE_V2(problem, params)
CostFunction = problem.CostFunction; % Cost Function
VarSize = [1 problem.nVar];     % Decision Variables Matrix Size
VarMin = problem.VarMin;            % Lower Bound of Decision
Variables
VarMax = problem.VarMax;             % Upper Bound of Decision
Variables
toleranceValue = problem.toleranceValue; % tolerance value at which
the solution is acceptable with the maximum error possible

%% DE Parameters

MaxIt = params.MaxIt;       % Maximum Number of Iterations
nPop = params.nPop;          % Population Size
beta_min = params.beta_min;     % Lower Bound of Scaling Factor (0)
beta_max = params.beta_max;     % Upper Bound of Scaling Factor (2)
```

```
pCR = params.pCR;          % Crossover Probability


%% Initialization

empty_individual.Position = [];
empty_individual.Cost = [];

BestSol.Cost = inf;

pop = repmat(empty_individual, nPop, 1);
%sample solution space
for i = 1:nPop
    %sample creation
    pop(i).Position = unifrnd(VarMin, VarMax, VarSize);
    %sample solution creation
    pop(i).Cost =
CostFunction(pop(i).Position(1),pop(i).Position(2));

    if pop(i).Cost<BestSol.Cost
        BestSol = pop(i);
    end

end
%inilialize the BestCost of all generations into zeros
BestCost = zeros(MaxIt, 1);

%% DE Main Loop

for it = 1:MaxIt
    %assign the best solution as best solution in the current
generation
    %which is used as base vector for mutate vector formation
    bestSolutionInPreviousGen = BestSol;
    %process
    for i = 1:nPop
        %ith individual
        x = pop(i).Position;
        %create a random arrangement (reorder the population
numbers)
        A = randperm(nPop);
        %Note: we can do this with rand, but we want all numbers
different

        %remove the i index from this since we dont want that to
        %be target vector
        A(A == i) = [];

        a = A(1);   % Random Vector 1 index
```

```
        b = A(2);   % Random Vector 2 index
        c = A(3);  % Random Vector 3 index
        d = A(4);   %random Vector 4 index
        % Mutant Vector (Mutation)
        beta = unifrnd(beta_min, beta_max, VarSize);

        %y is the mutant vector - best is used as base vector
        y = bestSolutionInPreviousGen.Position +
beta.*(pop(a).Position + pop(b).Position - pop(c).Position -
pop(d).Position);
        y = max(y, VarMin);
        y = min(y, VarMax);

        % Trial Vector (Crossover between Target Vector x and Mutant
Vector y)
        z = zeros(size(x));
        j0 = randi([1 numel(x)]);
        for j = 1:numel(x)
            if j == j0 || rand <= pCR
                z(j) = y(j);%from the mutant vector
            else
                z(j) = x(j);%from the target vector
            end
        end

        NewSol.Position = z;
        NewSol.Cost =
CostFunction(NewSol.Position(1),NewSol.Position(2));

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

    end

    % Update Best Cost
    BestCost(it) = BestSol.Cost;

    % Show Iteration Information
    disp(['Iteration ' num2str(it) ': Best Cost = '
num2str(BestCost(it))]);

    if BestCost(it) <= toleranceValue
        disp(['Tolerance value has been reached at generation: '
num2str(it)]);
```

```
        out.BestCost = BestCost;
        out.BestSol =BestSol;
        out.minIterationToReachToleranceValue = it;
        return;
    end

end
out.BestCost = BestCost;
out.BestSol =BestSol;
%assing number of iteration to reach minimum tolerance value as
MaxIt since the generations of MaxIt has
%not reached to the required result, this is done as part of
plotting the
%graph without any error
out.minIterationToReachToleranceValue =MaxIt;
end
```

Function Call for Figure 4.5.2.2.2:

```
%% DE /best/2/bin RosenBrock Function Unconstrainted
clc; clear;
%% Problem Definition

problem.CostFunction = @(x,y) RosenBrockFunction(x,y);    % Cost
Function
problem.nVar = 2;              % Number of Decision Variables
problem.VarMin = -5;            % Lower Bound of Decision Variables
problem.VarMax = 5;             % Upper Bound of Decision Variables
problem.toleranceValue = 10^-2; % tolerance value at which the
solution is acceptable with the maximum error possible

%% DE Parameters
params.MaxIt = 1000;       % Maximum Number of Iterations
params.nPop = 40;          % Population Size
params.beta_min = 0.2;     % Lower Bound of Scaling Factor (0)
params.beta_max = 0.8;     % Upper Bound of Scaling Factor (2)
params.pCR = 0.5;          % Crossover Probability

%% Calling DE
% number of iterations required
noOfTests = 10;
% array to store the result of each test, initially all assign to
infinity
minIterationsFromEachTest = inf(1,noOfTests);
%test the experiment for noOfTests times
for i=1:noOfTests
    out = DE_V2(problem, params);
```

```
minIterationsFromEachTest(i)=out.minIterationToReachToleranceValue;
    %plot the graph for this test result
    hold on
    plot(out.BestCost,"LineWidth",2)
    hold off
end

%% Show Results
%Calculate Median value
medianOfAllSolutions = median(minIterationsFromEachTest);

%plot median value in the graph
hold on
plot(medianOfAllSolutions,0,"r*","MarkerSize",20)
hold off
title("The Median of 10 Tests - Benchmark A")
%attributes to the graph
xlabel('Iteration');
ylabel('Best Cost');
yline(problem.toleranceValue,'-',{'Acceptable','Limit'});
xlim([0,100]);
ylim([0,0.1]);
%draw a line parallel to x axis to find on which iteration the
output
%reaches to tolerance value
yline(problem.toleranceValue,'-',{'Acceptable','Limit:',
num2str(problem.toleranceValue) });

grid on;
```

Appendix 6

6.A: Benchmark ConDE-A (DE rand/1/bin)

6.A.1 Outcomes of Tests

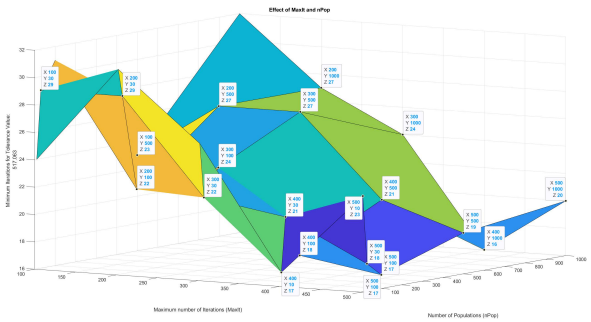


Figure 6.A.1.1

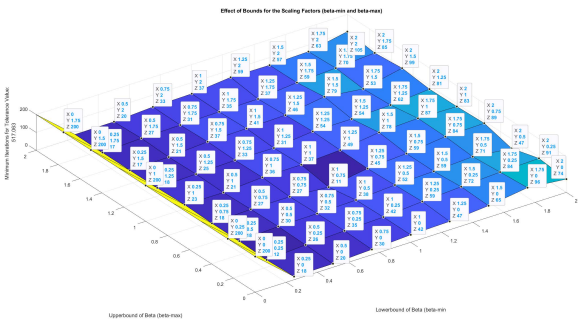


Figure 6.A.1.2

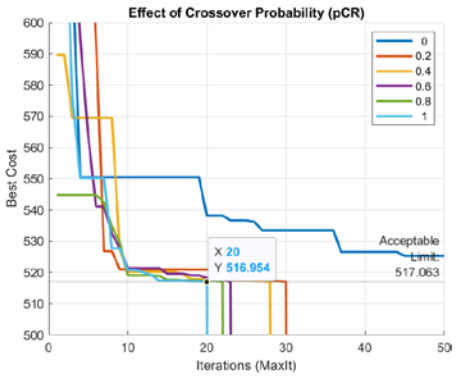


Figure 6.A.1.3

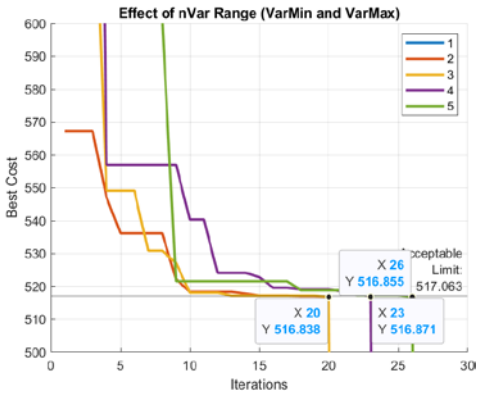


Figure 6.A.1.4

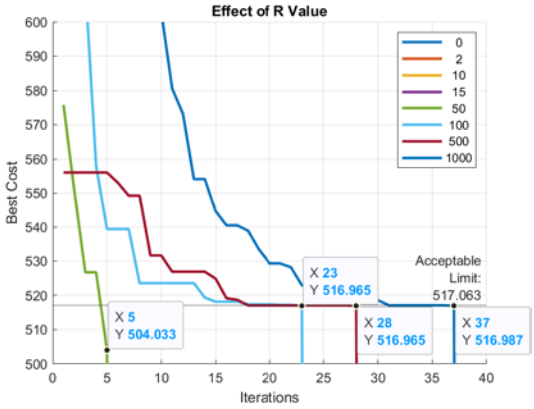


Figure 6.A.1.5

6.A.2 MATLAB Code Associated with the tests

Constraint DE Implementation:

```
%% DE /rand/1/bin
function out = DE(problem, params)
CostFunction = problem.CostFunction; % Cost Function
Constraints = problem.Constraints; % Constraint function
FitnessValue = problem.FitnessValue; %Fitness function

VarSize = [1 problem.nVar];     % Decision Variables Matrix Size
VarMin = problem.VarMin;            % Lower Bound of Decision
Variables
VarMax = problem.VarMax;             % Upper Bound of Decision
Variables
toleranceValue = problem.toleranceValue; % tolerance value at which
the solution is acceptable with the maximum error possible

%% DE Parameters

MaxIt = params.MaxIt;       % Maximum Number of Iterations
nPop = params.nPop;          % Population Size
beta_min = params.beta_min;     % Lower Bound of Scaling Factor (0)
beta_max = params.beta_max;     % Upper Bound of Scaling Factor (2)
pCR = params.pCR;          % Crossover Probability

R = params.R;             % Static Penalty Parameter

%% Initialization

empty_individual.Position = [];
empty_individual.Cost = [];

BestSol.Cost = inf;

pop = repmat(empty_individual, nPop, 1);
%sample solution space
for i = 1:nPop
    %sample creation
    pop(i).Position = unifrnd(VarMin, VarMax, VarSize);
    %sample solution creation
    pop(i).Cost =
FitnessValue(pop(i).Position(1),pop(i).Position(2), R);

    if pop(i).Cost<BestSol.Cost
        BestSol = pop(i);
    end

end
%inilialize the BestCost of all generations into zeros
BestCost = zeros(MaxIt, 1);
%% DE Main Loop
```

```
for it = 1:MaxIt

    for i = 1:nPop
        %ith individual
        x = pop(i).Position;
        %create a random arrangement (reorder the population
numbers)
        A = randperm(nPop);
        %Note: we can do this with rand, but we want all numbers
different

        %remove the i index from this since we dont want that to
        %be target vector
        A(A == i) = [];

        a = A(1);   % Target Vector index
        b = A(2);   % Random Vector 1 index
        c = A(3);   % Random Vector 2 index

        % Mutant Vector
        beta = unifrnd(beta_min, beta_max, VarSize);

        %y is the mutant vector
        y = pop(a).Position + beta.*(pop(b).Position -
pop(c).Position);
        y = max(y, VarMin);
        y = min(y, VarMax);

        % Trial Vector (Crossover between Target Vector x and Mutant
Vector y)
        z = zeros(size(x));
        j0 = randi([1 numel(x)]);
        for j = 1:numel(x)
            if j == j0 || rand <= pCR
                z(j) = y(j);%from the mutant vector
            else
                z(j) = x(j);%from the target vector
            end
        end

        NewSol.Position = z;
        NewSol.Cost =
FitnessValue(NewSol.Position(1),NewSol.Position(2), R); %Function
evaluation

        % Selection
        if NewSol.Cost<pop(i).Cost
            pop(i) = NewSol;
            if pop(i).Cost<BestSol.Cost
```

```
                BestSol = pop(i);
            end
        end

    end

    % Update Best Cost
    BestCost(it) = BestSol.Cost;

    % Show Iteration Information
    disp(['Iteration ' num2str(it) ': Best Cost = '
num2str(BestCost(it))]);
    % Comparing the best cost with the tolerance value
    % This code has been added for improving efficiency of the code
    if BestCost(it) <= toleranceValue
        disp(['Tolerance value has been reached at generation: '
num2str(it)]);
        %return from the function
        out.BestCost = BestCost;
        out.BestSol =BestSol;
        out.minIterationToReachToleranceValue = it;
        disp(['Constraints: ', num2str(Constraints(x(1), x(2), R))]
);
        return;
    end

end

out.BestCost = BestCost;
out.BestSol =BestSol;
%assing number of iteration to reach minimum tolerance value as
MaxIt since the generations of MaxIt has
%not reached to the required result, this is done as part of
plotting the
%graph without any error
out.minIterationToReachToleranceValue =MaxIt;
% Display Best Solution in Command Window
disp(BestSol)
x = BestSol.Position;
disp(['Constraints: ', num2str(Constraints(x(1), x(2), R))] );
end
```

Function Call for Figure 4.6.1.2:

```
%% DE /rand/1/bin with Contraint HImmelblue function
clc; clear;

%% Problem Definition
```

```
problem.CostFunction = @(x1,x2) HimmelblauFunction(x1,x2); % Cost
Function
problem.Constraints = @(x1,x2,R) InequalityConstraints(x1,x2,R); %
Constraint
problem.FitnessValue = @(x1,x2,R) problem.CostFunction(x1,x2) +
problem.Constraints(x1,x2,R); % Fitness Value

problem.nVar = 2;              % Number of Decision Variables
problem.VarMin = 0;            % Lower Bound of Decision Variables
x1 and x2 are greater than 0
problem.VarMax = 6;             % Upper Bound of Decision Variables
problem.toleranceValue = 517.063; % this is not the optimum
solution, but to demonstrate the convergence for this problem

%% DE Parameters

params.MaxIt = 200;       % Maximum Number of Iterations
params.nPop = 50;          % Population Size
params.beta_min = 0.3;     % Lower Bound of Scaling Factor (0)
params.beta_max = 0.8;     % Upper Bound of Scaling Factor (2)
params.pCR = 0.5;          % Crossover Probability
params.R = 100;             % Static Penalty Parameter


%% Calling DE
% number of iterations required
noOfTests = 10;
% array to store the result of each test, initially all assign to
infinity
minIterationsFromEachTest = inf(1,noOfTests);
%test the experiment for noOfTests times
for i=1:noOfTests
    out = DE(problem, params);

minIterationsFromEachTest(i)=out.minIterationToReachToleranceValue;
    %plot the graph for this test result
    hold on
    plot(out.BestCost,"LineWidth",2)
    hold off
end

%% Show Results
%Calculate Median value
medianOfAllSolutions = median(minIterationsFromEachTest);

%plot median value in the graph
hold on
plot(medianOfAllSolutions,0,"r*","MarkerSize",20)
hold off
```

```
title("The Median of 10 Tests - Benchmark A")
%attributes to the graph
xlabel('Iteration');
ylabel('Best Cost');
yline(problem.toleranceValue,'-',{'Acceptable','Limit'});
xlim([0,50]);
ylim([0, 900]);
%draw a line parallel to x axis to find on which iteration the
output
%reaches to tolerance value
yline(problem.toleranceValue,'-',{'Acceptable','Limit:',
num2str(problem.toleranceValue) });

grid on;
```

Function Call for Figure 6.A.1.1:

```
%% Effect of MaxIt and nPop

%save different values of MaxIt and nPop inorder to do the
experiment
maxItVariation = [100, 200, 300, 400, 500];
nPopVariation = [10, 30, 100, 500, 1000];

% use two for loops to do the experiment with various combinations
of
% parameters taken into consideration
for i=1:numel(maxItVariation)
        params.MaxIt = maxItVariation(i);       % Maximum Number of
Iterations
    for j=1:numel(nPopVariation)
        params.nPop = nPopVariation(i);          % Population Size
        out = DE(problem, params);
        Z(i,j)= out.minIterationToReachToleranceValue;
    end
end


% Describing the attributes for the graph
[X,Y] = meshgrid(maxItVariation,nPopVariation);
Z= transpose(Z)
surf(X,Y,Z)
title('Effect of MaxIt and nPop ');
xlabel('Maximum number of Iterations (MaxIt)');
ylabel('Number of Populations (nPop)');
zlabel(["Minimum Iterations for Tolerance Value: "
num2str(problem.toleranceValue)]);
```

Function Call for Figure 6.A.1.2:

```
%% Effect of beta(beta_min and beta_max) in the result

%save different values of beta_min and beta_max
boundVariations = [0:0.25:2];% for figure-A
% boundVariations = [-2,-1,3,4];% for figure-C

% size of variation arrays for the loop generation
sizeOfbetaMinVariation = numel(boundVariations);
sizeOfbetaMaxVariation = numel(boundVariations);

% use two for loops to do the experiment with various combinations
for i=1:sizeOfbetaMaxVariation
    % Lowerbound for scaling factor-beta (bata_min)
    params.beta_min = boundVariations(i);
    for j=1:sizeOfbetaMaxVariation
        % Upperbound for scaling factor-beta (bata_max)
        params.beta_max = boundVariations(i);
        out = DE(problem, params);
        Z(i,j)= out.minIterationToReachToleranceValue;
    end
end
```

Function Call for Figure 6.A.1.3:

```
%% Calling DE for different pCR values to see the effect of it in
the Result
%different number of 'pCR' values
pCRVariation = [0:0.2:1];

%Variable to store the number of Iterations take to reach the
tolerance
%value
numberOfIterations = numel(pCRVariation);
%repeat the test for numberOfIterations times
for i=1:numberOfIterations
    %assign value to pCR
    params.pCR = pCRVariation(i);
    out= DE(problem, params);
    % plot the result into the existing graph else create a new
graph
    hold on
    semilogy(out.BestCost,"LineWidth",2);
    hold off
end
```

Function Call for Figure 6.A.1.5:

```
%% Effect of Static Penality Parameter R

%save different values of beta values for the experiment
RVariation = [0,2,10,15,50,100,500,1000];
```

```
%Do the experiment for number of Iteration
for i=1:numel(RVariation)
    params.R = RVariation(i);
    out= DE(problem, params);
    % plot the result into the existing graph
    hold on
    plot(out.BestCost,"LineWidth",2)
    hold off
end
```

6.B: Benchmark UnConDE-B – Variant 1 (DE best/1/bin)

6.B.1 Outcomes of Tests

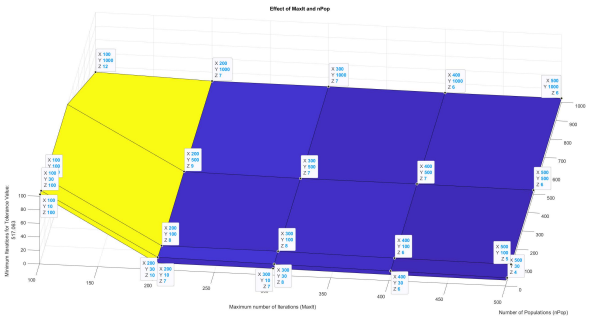


Figure 6.B.1.1

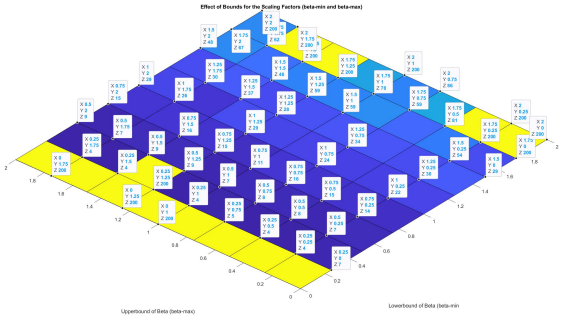


Figure 6.B.1.2

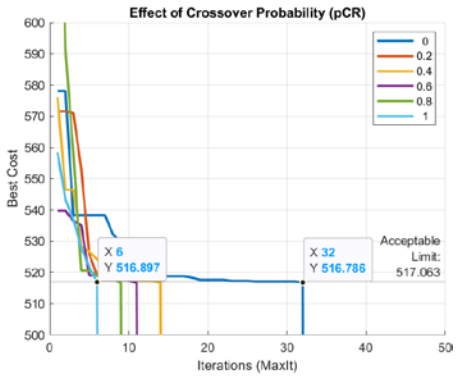


Figure 6.B.1.3

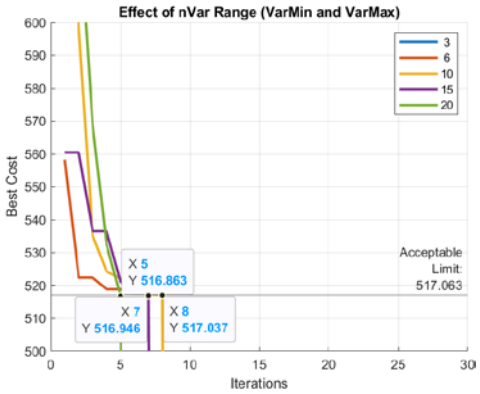


Figure 6.B.1.4

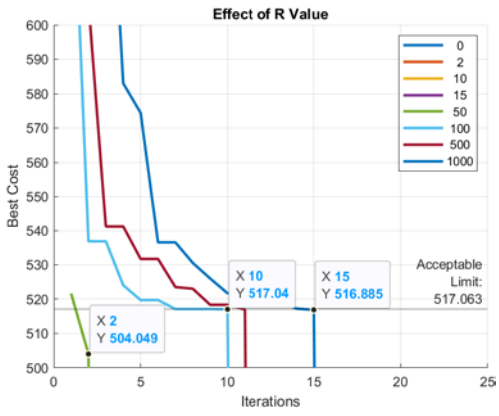


Figure 6.B.1.5

6.B.2 MATLAB Code Associated with the tests

Constraint DE Implementation:

```
%% DE /best/1/bin with Contraint HImmelblue function with Statoc
Penality
function out = DE(problem, params)
CostFunction = problem.CostFunction; % Cost Function
Constraints = problem.Constraints; % Constraint function
FitnessValue = problem.FitnessValue; %Fitness function

VarSize = [1 problem.nVar];     % Decision Variables Matrix Size
VarMin = problem.VarMin;            % Lower Bound of Decision
Variables
VarMax = problem.VarMax;             % Upper Bound of Decision
Variables
toleranceValue = problem.toleranceValue; % tolerance value at which
the solution is acceptable with the maximum error possible

%% DE Parameters

MaxIt = params.MaxIt;       % Maximum Number of Iterations
nPop = params.nPop;          % Population Size
beta_min = params.beta_min;     % Lower Bound of Scaling Factor (0)
beta_max = params.beta_max;     % Upper Bound of Scaling Factor (2)
pCR = params.pCR;          % Crossover Probability

R = params.R;             % Static Penalty Parameter

%% Initialization

empty_individual.Position = [];
empty_individual.Cost = [];

BestSol.Cost = inf;

pop = repmat(empty_individual, nPop, 1);
%sample solution space
for i = 1:nPop
    %sample creation
    pop(i).Position = unifrnd(VarMin, VarMax, VarSize);
    %sample solution creation
    pop(i).Cost =
FitnessValue(pop(i).Position(1),pop(i).Position(2), R);

    if pop(i).Cost<BestSol.Cost
        BestSol = pop(i);
    end

end
%inilialize the BestCost of all generations into zeros
BestCost = zeros(MaxIt, 1);
%% DE Main Loop
```

```
for it = 1:MaxIt
%assign the best solution as best solution in the current generation
    %which is used as base vector for mutate vector formation
    bestSolutionInPreviousGen = BestSol;
    %process
    for i = 1:nPop
        %ith individual
        x = pop(i).Position;
        %create a random arrangement (reorder the population
numbers)
        A = randperm(nPop);
        %Note: we can do this with rand, but we want all numbers
different

        %remove the i index from this since we dont want that to
        %be target vector
        A(A == i) = [];

        a = A(1);   % Random Vector 1 index
        b = A(2);   % Random Vector 2 index

        % Mutant Vector (Mutation)
        beta = unifrnd(beta_min, beta_max, VarSize);

        %y is the mutant vector - best is used as base vector
        y = bestSolutionInPreviousGen.Position +
beta.*(pop(a).Position - pop(b).Position);
        y = max(y, VarMin);
        y = min(y, VarMax);

        % Trial Vector (Crossover between Target Vector x and Mutant
Vector y)
        z = zeros(size(x));
        j0 = randi([1 numel(x)]);
        for j = 1:numel(x)
            if j == j0 || rand <= pCR
                z(j) = y(j);%from the mutant vector
            else
                z(j) = x(j);%from the target vector
            end
        end

        NewSol.Position = z;
        NewSol.Cost =
FitnessValue(NewSol.Position(1),NewSol.Position(2), R); %Function
evaluation

        % Selection
```

```
        if NewSol.Cost<pop(i).Cost
            pop(i) = NewSol;
            if pop(i).Cost<BestSol.Cost
                BestSol = pop(i);
            end
        end

    end

    % Update Best Cost
    BestCost(it) = BestSol.Cost;

    % Show Iteration Information
    disp(['Iteration ' num2str(it) ': Best Cost = '
num2str(BestCost(it))]);
    % Comparing the best cost with the tolerance value
    % This code has been added for improving efficiency of the code
    if BestCost(it) <= toleranceValue
        disp(['Tolerance value has been reached at generation: '
num2str(it)]);
        %return from the function
        out.BestCost = BestCost;
        out.BestSol =BestSol;
        out.minIterationToReachToleranceValue = it;
        disp(['Constraints: ', num2str(Constraints(x(1), x(2), R))]
);
        return;
    end

end

out.BestCost = BestCost;
out.BestSol =BestSol;
%assing number of iteration to reach minimum tolerance value as
MaxIt since the generations of MaxIt has
%not reached to the required result, this is done as part of
plotting the
%graph without any error
out.minIterationToReachToleranceValue =MaxIt;
% Display Best Solution in Command Window
disp(BestSol)
x = BestSol.Position;
disp(['Constraints: ', num2str(Constraints(x(1), x(2), R))] );
end
```

Function Call for Figure 4.6.2.1.2:

```
%% DE /rand/1/bin with Contraint HImmelblue function
```

```
clc; clear;

%% Problem Definition
problem.CostFunction = @(x1,x2) HimmelblauFunction(x1,x2); % Cost
Function
problem.Constraints = @(x1,x2,R) InequalityConstraints(x1,x2,R); %
Constraint
problem.FitnessValue = @(x1,x2,R) problem.CostFunction(x1,x2) +
problem.Constraints(x1,x2,R); % Fitness Value

problem.nVar = 2;              % Number of Decision Variables
problem.VarMin = 0;            % Lower Bound of Decision Variables
x1 and x2 are greater than 0
problem.VarMax = 6;             % Upper Bound of Decision Variables
problem.toleranceValue = 517.063; % this is not the optimum
solution, but to demonstrate the convergence for this problem

%% DE Parameters

params.MaxIt = 200;       % Maximum Number of Iterations
params.nPop = 50;          % Population Size
params.beta_min = 0.3;     % Lower Bound of Scaling Factor (0)
params.beta_max = 0.8;     % Upper Bound of Scaling Factor (2)
params.pCR = 0.5;          % Crossover Probability
params.R = 100;             % Static Penalty Parameter


%% Calling DE
% number of iterations required
noOfTests = 10;
% array to store the result of each test, initially all assign to
infinity
minIterationsFromEachTest = inf(1,noOfTests);
%test the experiment for noOfTests times
for i=1:noOfTests
    out = DE(problem, params);

minIterationsFromEachTest(i)=out.minIterationToReachToleranceValue;
    %plot the graph for this test result
    hold on
    plot(out.BestCost,"LineWidth",2)
    hold off
end

%% Show Results
%Calculate Median value
medianOfAllSolutions = median(minIterationsFromEachTest);

%plot median value in the graph
```

```
hold on
plot(medianOfAllSolutions,0,"r*","MarkerSize",20)
hold off
title("The Median of 10 Tests - Benchmark A")
%attributes to the graph
xlabel('Iteration');
ylabel('Best Cost');
yline(problem.toleranceValue,'-',{'Acceptable','Limit'});
xlim([0,50]);
ylim([0, 900]);
%draw a line parallel to x axis to find on which iteration the
output
%reaches to tolerance value
yline(problem.toleranceValue,'-',{'Acceptable','Limit:',
num2str(problem.toleranceValue) });

grid on;
```

6.C: Benchmark ConDE-B – Variant 2 (DE best/2/bin)

6.C.1 Outcomes of Tests

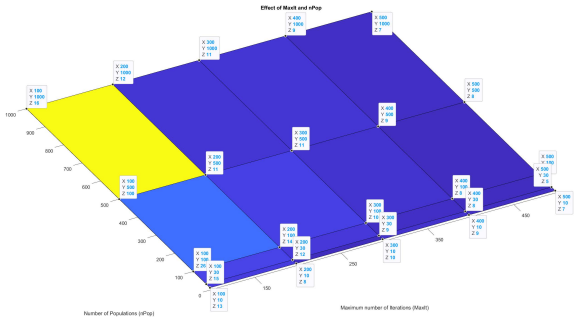


Figure 6.C.1.1

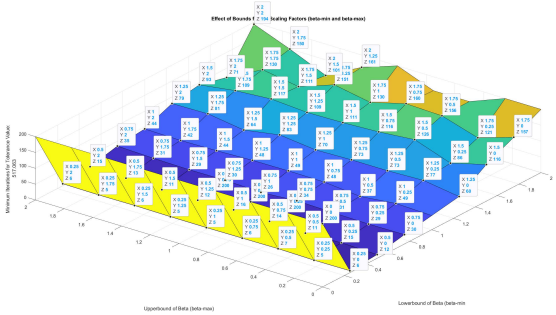


Figure 6.C.1.2

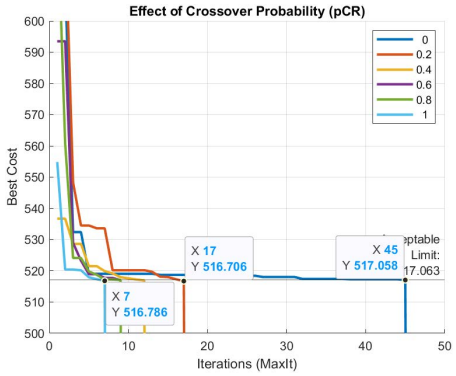


Figure 6.C.1.3

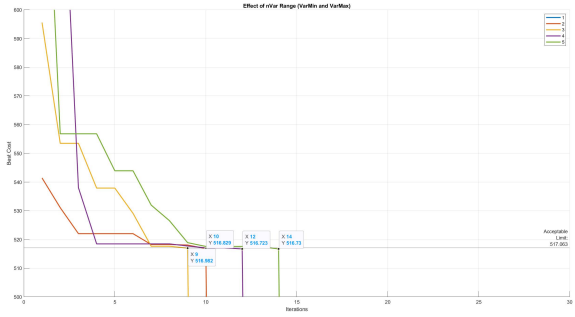


Figure 6.C.1.4

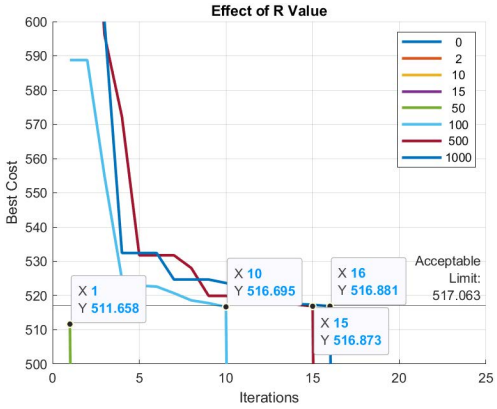


Figure 6.C.1.5

6.C.2 MATLAB Code Associated with the tests

Constraint DE Implementation:

```
%% DE /best/2/bin with Contraint HImmelblue function
function out = DE(problem, params)
CostFunction = problem.CostFunction; % Cost Function
Constraints = problem.Constraints; % Constraint function
FitnessValue = problem.FitnessValue; %Fitness function
```

```
VarSize = [1 problem.nVar];     % Decision Variables Matrix Size
VarMin = problem.VarMin;            % Lower Bound of Decision
Variables
VarMax = problem.VarMax;             % Upper Bound of Decision
Variables
toleranceValue = problem.toleranceValue; % tolerance value at which
the solution is acceptable with the maximum error possible

%% DE Parameters

MaxIt = params.MaxIt;       % Maximum Number of Iterations
nPop = params.nPop;          % Population Size
beta_min = params.beta_min;     % Lower Bound of Scaling Factor (0)
beta_max = params.beta_max;     % Upper Bound of Scaling Factor (2)
pCR = params.pCR;          % Crossover Probability

R = params.R;             % Static Penalty Parameter

%% Initialization

empty_individual.Position = [];
empty_individual.Cost = [];

BestSol.Cost = inf;

pop = repmat(empty_individual, nPop, 1);
%sample solution space
for i = 1:nPop
    %sample creation
    pop(i).Position = unifrnd(VarMin, VarMax, VarSize);
    %sample solution creation
    pop(i).Cost =
FitnessValue(pop(i).Position(1),pop(i).Position(2), R);

    if pop(i).Cost<BestSol.Cost
        BestSol = pop(i);
    end

end
%inilialize the BestCost of all generations into zeros
BestCost = zeros(MaxIt, 1);
%% DE Main Loop

for it = 1:MaxIt
    %assign the best solution as best solution in the current
generation
    %which is used as base vector for mutate vector formation
    bestSolutionInPreviousGen = BestSol;
```

```
    %process
    for i = 1:nPop
        %ith individual
        x = pop(i).Position;
        %create a random arrangement (reorder the population
numbers)
        A = randperm(nPop);
        %Note: we can do this with rand, but we want all numbers
different

        %remove the i index from this since we dont want that to
        %be target vector
        A(A == i) = [];

        a = A(1);   % Random Vector 1 index
        b = A(2);   % Random Vector 2 index
        c = A(3);  % Random Vector 3 index
        d = A(4);   %random Vector 4 index
        % Mutant Vector (Mutation)
        beta = unifrnd(beta_min, beta_max, VarSize);

        %y is the mutant vector - best is used as base vector
        y = bestSolutionInPreviousGen.Position +
beta.*(pop(a).Position + pop(b).Position - pop(c).Position -
pop(d).Position);
        y = max(y, VarMin);
        y = min(y, VarMax);


        % Trial Vector (Crossover between Target Vector x and Mutant
Vector y)
        z = zeros(size(x));
        j0 = randi([1 numel(x)]);
        for j = 1:numel(x)
            if j == j0 || rand <= pCR
                z(j) = y(j);%from the mutant vector
            else
                z(j) = x(j);%from the target vector
            end
        end

        NewSol.Position = z;
        NewSol.Cost =
FitnessValue(NewSol.Position(1),NewSol.Position(2), R); %Function
evaluation

        % Selection
        if NewSol.Cost<pop(i).Cost
            pop(i) = NewSol;
```

```
            if pop(i).Cost<BestSol.Cost
                BestSol = pop(i);
            end
        end

    end

    % Update Best Cost
    BestCost(it) = BestSol.Cost;

    % Show Iteration Information
    disp(['Iteration ' num2str(it) ': Best Cost = '
num2str(BestCost(it))]);
    % Comparing the best cost with the tolerance value
    % This code has been added for improving efficiency of the code
    if BestCost(it) <= toleranceValue
        disp(['Tolerance value has been reached at generation: '
num2str(it)]);
        %return from the function
        out.BestCost = BestCost;
        out.BestSol =BestSol;
        out.minIterationToReachToleranceValue = it;
        disp(['Constraints: ', num2str(Constraints(x(1), x(2), R))]
);
        return;
    end

end

out.BestCost = BestCost;
out.BestSol =BestSol;
%assing number of iteration to reach minimum tolerance value as
MaxIt since the generations of MaxIt has
%not reached to the required result, this is done as part of
plotting the
%graph without any error
out.minIterationToReachToleranceValue =MaxIt;
% Display Best Solution in Command Window
disp(BestSol)
x = BestSol.Position;
disp(['Constraints: ', num2str(Constraints(x(1), x(2), R))] );
end
```

Function Call for Figure 4.6.2.2.2:

```
%% DE /rand/1/bin with Contraint HImmelblue function
clc; clear;

%% Problem Definition
```

```
problem.CostFunction = @(x1,x2) HimmelblauFunction(x1,x2); % Cost
Function
problem.Constraints = @(x1,x2,R) InequalityConstraints(x1,x2,R); %
Constraint
problem.FitnessValue = @(x1,x2,R) problem.CostFunction(x1,x2) +
problem.Constraints(x1,x2,R); % Fitness Value

problem.nVar = 2;              % Number of Decision Variables
problem.VarMin = 0;            % Lower Bound of Decision Variables
x1 and x2 are greater than 0
problem.VarMax = 6;             % Upper Bound of Decision Variables
problem.toleranceValue = 517.063; % this is not the optimum
solution, but to demonstrate the convergence for this problem

%% DE Parameters

params.MaxIt = 200;       % Maximum Number of Iterations
params.nPop = 50;          % Population Size
params.beta_min = 0.3;     % Lower Bound of Scaling Factor (0)
params.beta_max = 0.8;     % Upper Bound of Scaling Factor (2)
params.pCR = 0.5;          % Crossover Probability
params.R = 100;             % Static Penalty Parameter


%% Calling DE
% number of iterations required
noOfTests = 10;
% array to store the result of each test, initially all assign to
infinity
minIterationsFromEachTest = inf(1,noOfTests);
%test the experiment for noOfTests times
for i=1:noOfTests
    out = DE(problem, params);

minIterationsFromEachTest(i)=out.minIterationToReachToleranceValue;
    %plot the graph for this test result
    hold on
    plot(out.BestCost,"LineWidth",2)
    hold off
end

%% Show Results
%Calculate Median value
medianOfAllSolutions = median(minIterationsFromEachTest);

%plot median value in the graph
hold on
plot(medianOfAllSolutions,0,"r*","MarkerSize",20)
hold off
```

```
title("The Median of 10 Tests - Benchmark A")
%attributes to the graph
xlabel('Iteration');
ylabel('Best Cost');
yline(problem.toleranceValue,'-',{'Acceptable','Limit'});
xlim([0,50]);
ylim([0, 900]);
%draw a line parallel to x axis to find on which iteration the
output
%reaches to tolerance value
yline(problem.toleranceValue,'-',{'Acceptable','Limit:',
num2str(problem.toleranceValue) });

grid on;
```

Appendix 7

7.A: Contour Plot of UnConRGA

7.A.1: Benchmark UnConRGA-A

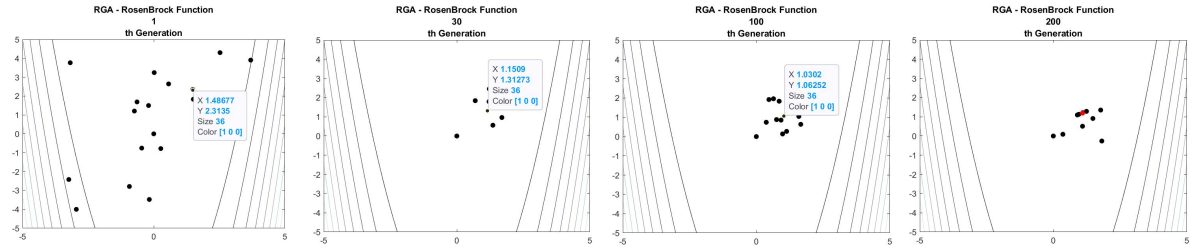


Figure 7.A.1

7.A.2: Benchmark UnConRGA-B

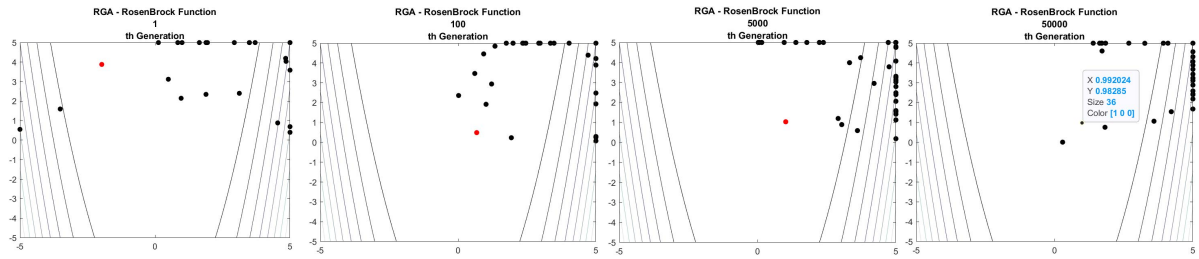


Figure 7.A.2

7.B: Contour Plot of ConRGA

7.B.1: Benchmark ConRGA-A

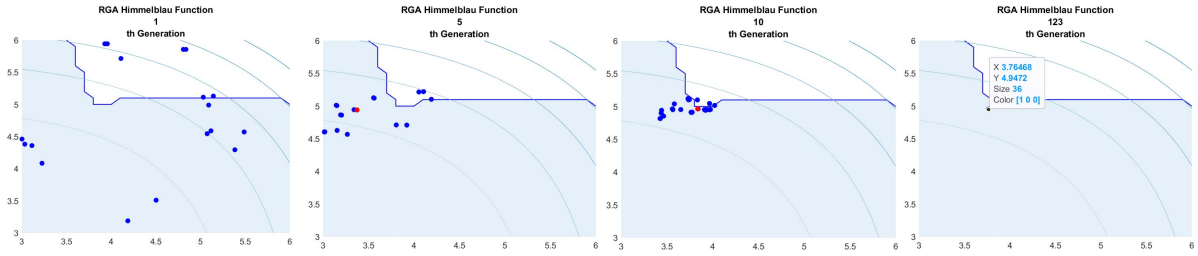


Figure 7.B.1

7.B.2: Benchmark ConRGA-B

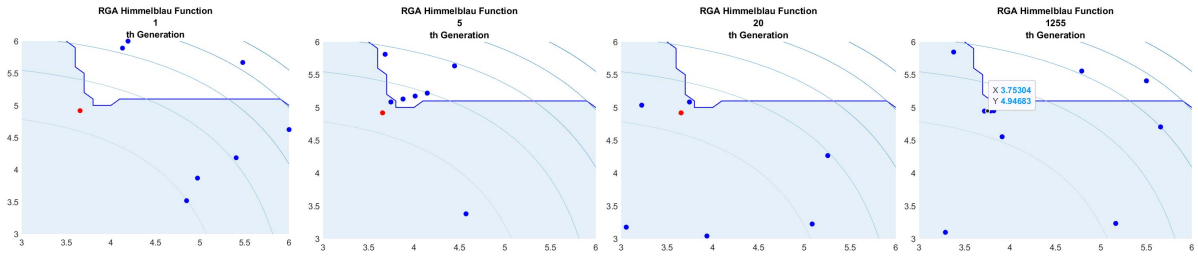


Figure7.B.2

7.C: Contour Plot of UnConPSO

7.C.1: Benchmark UnConPSO-A

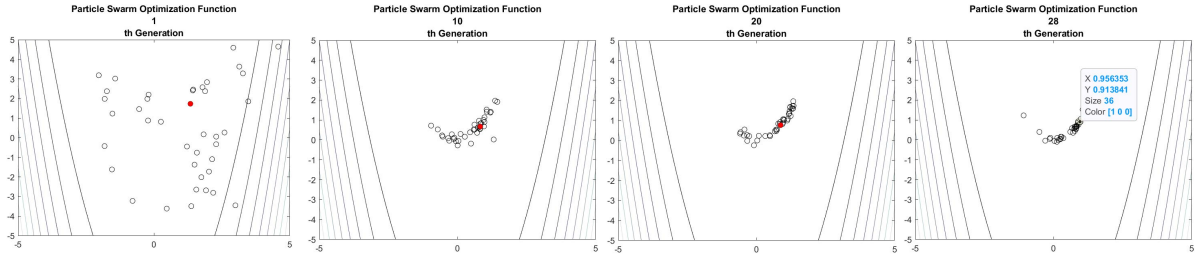


Figure 7.C.1

7.C.2: Benchmark UnConPSO-B

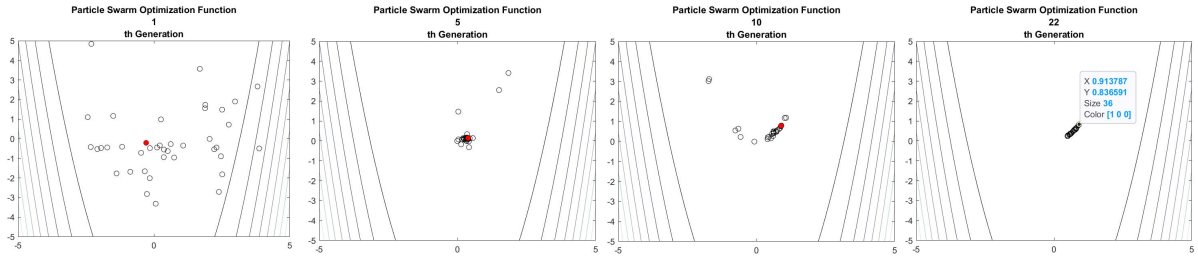


Figure 7.C.2

7.D: Contour Plot of ConPSO

7.D.1: Benchmark ConPSO-A

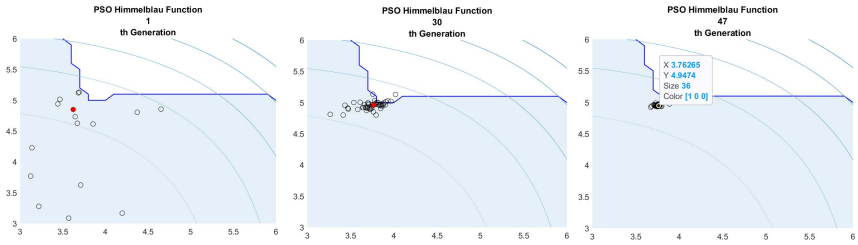


Figure 7.D.1

7.D.2: Benchmark ConPSO-B

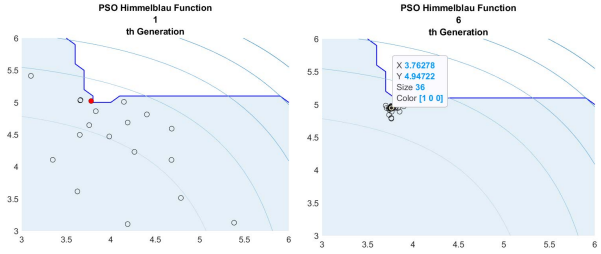


Figure 7.D.2

7.E: Contour Plot of UnConDE

7.E.1: Benchmark UnConDE -A

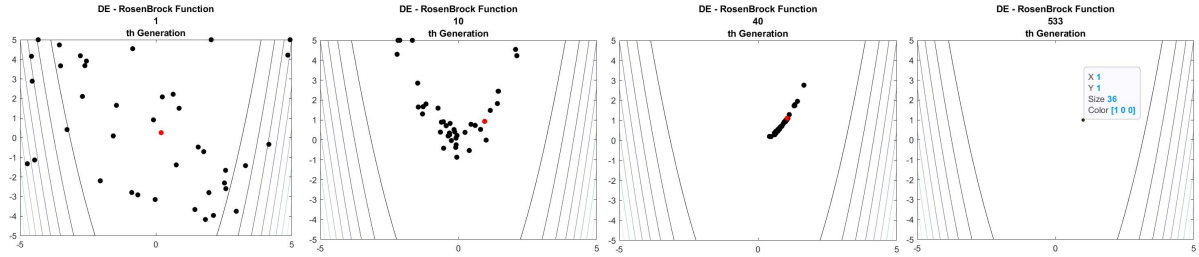


Figure 7.E.1

7.E.2: Benchmark UnConDE *-B*

7.E.2.1. Variant 1: DE /best/1/bin

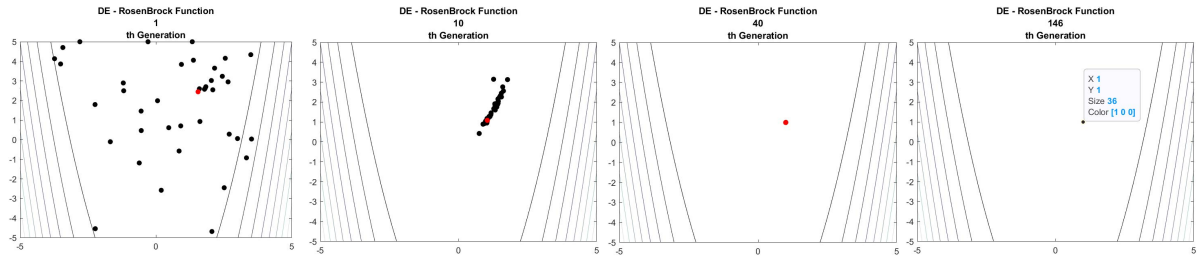


Figure 7.E.2.1

7.E.2.2 Variant 2: DE /best/2/bin

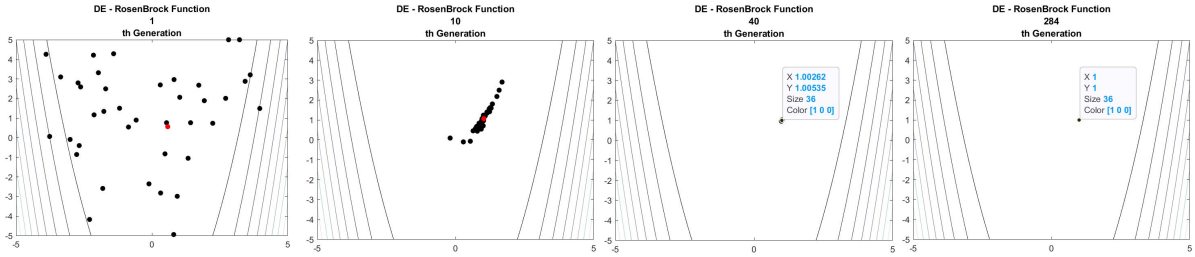


Figure 7.E.2.2

7.F: Contour Plot of ConDE

7.F.1: Benchmark ConDE *-A*

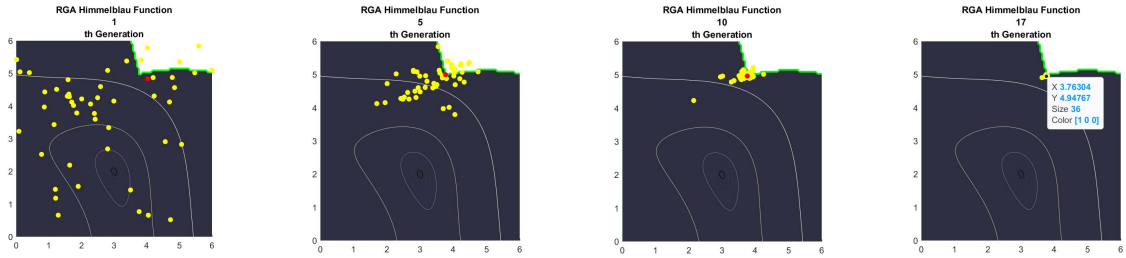


Figure 7.F.1

7.F.2: Benchmark ConDE *-B*

7.F.2.1 Variant 1: DE /best/1/bin

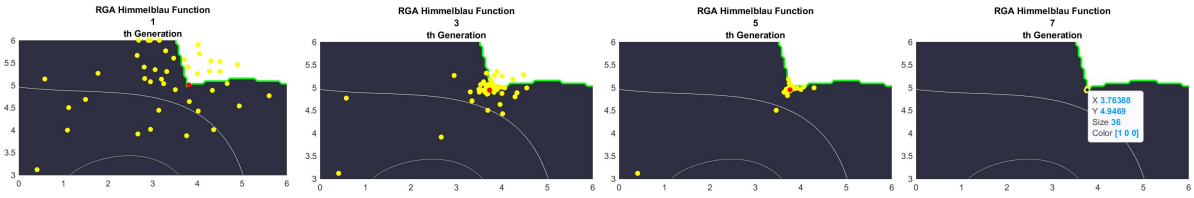


Figure 7.F.2.1

7.F.2.2 Variant 2: DE /best/2/bin

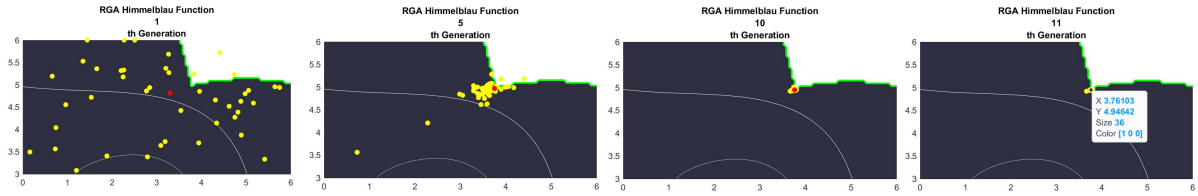


Figure 7.F.2.2

7.G: Contour Plot of RGA

% appRGA.m

```
% RGA with Unconstrainted RosenBrock Function (Optimum value when
f(1,1)=0)
clc; clear;
%% Problem Definition

problem.CostFunction = @(x,y) RosenBrockFunction(x,y);
problem.nVar = 2; %as per given in the lecture slide
problem.VarMin = -5; %as per given in the lecture slide
problem.VarMax = 5; %as per given in the lecture slide
problem.toleranceValue = 0; % tolerance value at which the solution
is acceptable with the maximum error possible

%% GA Parameters

params.MaxIt = 50000;
params.nPop = 40;
params.beta = 1;% For random probability in selection Operator
params.pC = 1;  % Probability of crossover
params.mu = 0.5; % = 1/nVar  % Mutation probability

%Generation numbers needed to be plot as a graph
params.genNumber = [1,100,5000,50000];
%% Run Real GA
out = RunRGAContour(problem, params);
```

%RunRGAContour.m

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

% Problem
```

```
CostFunction = problem.CostFunction;
nVar = problem.nVar;
VarSize = [1, nVar];
VarMin = problem.VarMin;
VarMax = problem.VarMax;
toleranceValue = problem.toleranceValue;

% Params
MaxIt = params.MaxIt;
nPop = params.nPop;
beta = params.beta;
pC = params.pC;
nC = round(pC*nPop/2)*2;
mu = params.mu;
genNumber =params.genNumber;%generation index of which graph has to
be ploted

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

% Best Solution Ever Found
bestsol.Cost = inf;

% Initialization
pop = repmat(empty_individual, nPop, 1);
for i = 1:nPop
    % Generate Random Solution
    pop(i).Position = unifrnd(VarMin, VarMax, VarSize);
    % Evaluate Solution
    pop(i).Cost = CostFunction(pop(i).Position(1),
pop(i).Position(2));
    % Compare Solution to Best Solution Ever Found
    if pop(i).Cost < bestsol.Cost
        bestsol = pop(i);
    end
end

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

%plot Control Variables
plotX = zeros(MaxIt,nPop);
plotY = zeros(MaxIt,nPop);

% Main Loop
for it = 1:MaxIt

    % Initialize Population for Evolution Stategy
```

```
    popc = repmat(empty_individual, nC, 1);

    % Applying Binary Tournament Selection
    popc  =
applyStochasticUniversalSamplingSelection(pop,nPop,popc,nC,beta);

    % Convert popc to nC/2x2 Matrix for Crossover Operation
    popc = reshape(popc,nC/2,2);


    % Applying Crossover Operation
    for k = 1:nC/2
        % Perform SBX Crossover Operation
        [popc(k, 1).Position, popc(k, 2).Position] =
applyLinearCrossover(popc(k,1).Position, popc(k,2).Position, pC,
CostFunction,empty_individual );
        % Check for Variable Bounds
        popc(k,1).Position = max(popc(k,1).Position, VarMin);
        popc(k,1).Position = min(popc(k,1).Position, VarMax);
        popc(k,2).Position = max(popc(k,2).Position, VarMin);
        popc(k,2).Position = min(popc(k,2).Position, VarMax);
    end
    % Convert popc to Single-Column Matrix
    popc = popc(:);

    % Apply Mutation Operation
    for l = 1:nC

        % Perform Polynomial Mutation
        popc(l).Position = applyRandomMutation(popc(l).Position, mu,
VarMin, VarMax);

        % Check for Variable Bounds
        popc(l).Position = max(popc(l).Position, VarMin);
        popc(l).Position = min(popc(l).Position, VarMax);

        % Evaluation
        popc(l).Cost =
CostFunction(popc(l).Position(1),popc(l).Position(2));

        % Compare Solution to Best Solution Ever Found
        if popc(l).Cost < bestsol.Cost
            bestsol = popc(l);
        end

        %save the best soliution found in the iteration it of
particle i
        %into plotX and plotY inorder to draw the graph
        plotX(it,l) = popc(l).Position(1);
```

```
        plotY(it,l) = popc(l).Position(2);

    end

    %applying Mu + Lambda Survivor Selection survivor selection
operaton
    pop =applyMuLambdaSurvivorSelection([pop; popc], nPop);

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

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


    %plot the graph with the newly updated solutions and best
received in the above
    %generation process( the drawing code has been seperated from
this into
    %another file named DrawTheContourPlot.m)
    %check with the toleranc value to make sure that the graph has
been
    %ploted for the optimum solution (0) here tolerance value =0 has
been
    %set
    if bestcost(it) <= toleranceValue
        disp(['Tolerance value has been reached at generation: '
num2str(it)]);
        genNumber =[it];
        DrawTheContourPlot
        %Once optimum has been reached no need to run the code again
so
        %we terminate the exicution at here
        out.bestsol = bestsol;
        out.bestcost = bestcost;
        out.minIterationToReachToleranceValue = it;
        return;
    else
        DrawTheContourPlot
    end

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

end
```

%DrawTheContourPlot.m

```
if any(it == genNumber(:))
    %this line of code below is to  make sure that all graphs come
into
    %seperate
    figure(it);
    % plot the contour graph
    %create mesh
    [X,Y] = meshgrid(VarMin:0.1:VarMax, VarMin:0.1:VarMax);
    %create the possible solution space with X and Y
    Z = 100*(Y-X.^2).^2+(1-X).^2;
    %draw contour
    colormap(bone)
    contour(X,Y,Z);

    %mark the points in the contour graph for the iteration 'it'
    for loop = 1:nPop
        hold on;
        scatter(plotX(it,loop),plotY(it,loop),"filled","k" );
        title(['RGA - RosenBrock Function' num2str(it) "th
Generation"]);
        hold on
        scatter(bestsol.Position(1),
bestsol.Position(2),"filled","o","r");
        hold off
    end
end
```

Appendix 8

8.A: Equations used for PSO Benchmark B

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

%calculation of w
w = w * wdamp
```

Equation 1

```
% adding dynamicity to wdamp
params.wmax = chi;% Maximum value of
Inertia Coefficient
params.wmin = 0.1; % Minimum value of
Inertia Coefficient

%calculation of w
wdamp = (it/MaxIt)*(wmax-wmin);
w = w * wdamp
```

Equation 2

8.B: Tables of Investigations and Observations

8.B.1. Benchmark UnConRGA-A

#	Parameter/ Combination of Parameters Taken for Tuning	Best Result	Tuned Parameter Values	Observations	Reference to the Figure
1	nPop and MaxIt	1	nPop = 500 and MaxIt=400	Higher value of nPop gives best result.	Appendix 1 Fig 1.A.1.1
2	nPop and pC	1	nPop = 300 and pC =1	Higher pC would better the result.	Appendix 1 Fig 1.A.1.2
3	nPop and mu	1	nPop=400 and mu=0	Less mu value gives better result.	Appendix 1 Fig 1.A.1.4
4	sigma and mu	1	sigma= 10 and mu= 0.3		Appendix 1 Fig 1.A.1.5
5	gamma and pC	1	gamma = 50 and pC = 1		Appendix 1 Fig 1.A.1.3
6	Tolerance Value	0.1	ToleranceValue = 0.1		Appendix 1 Fig 1.A.1.6
7	Variable bound (VarMin and VarMax)	1	VarMin = -1 and VarMax =1	Low variable bound convergence to optimum solution easily	Appendix 1 Fig 1.A.1.7

8.B.2. Benchmark UnConRGA-B

#	Parameter/ Combination of Parameters Taken for Tuning	Best Result	Tuned Parameter Values	Observations	Reference to the Figure
1	nPop and pC	1	nPop = 300 and pC =0.7		Appendix 1 Fig 1.B.1.1
2	nPop and mu	1	nPop=100 and mu=0		Appendix 1 Fig 1.B.1.2
5	beta	21	Beta =5		Appendix 1 Fig 1.B.1.3

8.B.3. Benchmark ConRGA-A

#	Parameter/ Combination of Parameters Taken for Tuning	Best Result	Tuned Parameter Values	Observations	Reference to the Figure`
1	nPop and MaxIt	4	nPop = 1000 and MaxIt=200	Large nPop gives better result	Appendix 2 Fig 2.A.1.1
2	nPop and pC	5	nPop = 400 and pC =0.7		Appendix 2 Fig 2.A.1.2
3	nPop and mu	5	nPop=400 and mu=1		Appendix 2 Fig 2.A.1.4
4	sigma and mu	1	sigma= 10 and mu= 0.3		Appendix 2 Fig 2.A.1.5
5	gamma and pC	17	gamma = 50 and pC = 1	Higher pC gives better result	Appendix 2 Fig 2.A.1.3
6	Tolerance Value	2	ToleranceValue = 550		Appendix 2 Fig 2.A.1.7
7	R	13	R=500		Appendix 2 Fig 2.A.1.8

8.B.4. Benchmark ConRGA-B

#	Parameter/ Combination of	Best Result	Tuned Parameter Values	Observations	Reference to the Figure

	Parameters Taken for Tuning				
1	nPop and pC	13	nPop = 300 and pC =1	higher nPop and pC will give better result	Appendix 2 Fig 2.B.1.1
2	nPop and mu	98	nPop=200 and mu=0.5		Appendix 2 Fig 2.B.1.2
3	Beta	34	Beta =1	Beta doesn't have much effect	Appendix 2 Fig 2.B.1.3
4	R	7	50		Appendix 2 Fig 2.B.1.4

8.B.5. Benchmark UnConPSO-A

#	Parameter/ Combination of Parameters Taken for Tuning	Best Result	Tuned Parameter Values	Observations	Reference to the Figure
1	nPop and MaxIt	1	nPop = 1000 and MaxIt=200		Appendix 3 Fig 3.A.1.1
2	nPop and w	1	nPop = 300 and w =3		Appendix 3 Fig 3.A.1.2
3	c1 and c2	18	c1 =-0.5 and c2 =2		Appendix 3 Fig 3.A.1.5
4	w and c1	1	w=6 and c1=8		Appendix 3 Fig 3.A.1.3
5	w and c2	6	w=6 and c2=8		Appendix 3 Fig 3.A.1.4
6	w and wdamp	3	w=1 and wdamp=-0.4		Appendix 3 Fig 3.A.1.6

8.B.6. Benchmark UnConPSO-B

#	Parameter/ Combination of Parameters Taken for Tuning	Best Result	Tuned Parameter Values	Observations	Reference to the Figure
1	Kappa	4	Kappa =0.9		Appendix 3 Fig 3.B.1.1
2	Variable Bounds	6	VarMin =-10 and VarMax =10		Appendix 3 Fig 3.B.1.3
3	kappa and phi1	5	Kappa=1 and phi1=9		Appendix 3 Fig 3.B.1.2
4	Wmin	3	wmin=-10		Appendix 3 Fig 3.B.1.4

8.B.7. Benchmark ConPSO-A

#	Parameter/ Combination of Parameters Taken for Tuning	Best Result	Tuned Parameter Values	Observations	Reference to the Figure
1	nPop and MaxIt	36	nPop = 500 and MaxIt=500		Appendix 4 Fig 4.A.1.1
2	nPop and w	12	nPop = 100 and w =1		Appendix 4 Fig 4.A.1.2
3	c1 and c2	30	c1 =1 and c2 =0.5		Appendix 4 Fig 4.A.1.5
4	w and c1	4	w=0 and c1=-2		Appendix 4 Fig 4.A.1.3
5	w and c2	7	w=0 and c2=2		Appendix 4 Fig 4.A.1.4
6	w and wdamp	6	w=1 and wdamp=0.2		Appendix 4 Fig 4.A.1.6
7	R	42	R=1000	Through R=50, gives the solution, R=1000 makes all solutions close to the best solution.	Appendix 4 Fig 4.A.1.7

8.B.8. Benchmark ConPSO-B

#	Parameter/ Combination of Parameters Taken for Tuning	Best Result	Tuned Parameter Values	Observations	Reference to the Figure
1	kappa	5	kappa =0.55		Appendix 4 Fig 4.B.1.1
2	wmin	4	wmin = 10		Appendix 4 Fig 4.B.1.3
3	kappa and phi1	6	kappa=0.55 and phi1=1		Appendix 4 Fig 4.B.1.2
4	R	8	R=500		Appendix 4 Fig 4.B.1.4

8.B.9. Benchmark UnConDE-A

#	Parameter/ Combination of Parameters Taken for Tuning	Best Result	Tuned Parameter Values	Observations	Reference to the Figure
1	nPop and MaxIt	5	nPop = 500 and MaxIt=500		Appendix 5 Fig 5.A.1.1
2	pCR	10	pCR =1		Appendix 5 Fig 5.A.1.2
3	Variable Range	30	VarMin = -3 and VarMax =3		Appendix 5 Fig 5.A.1.4
4	Beta Range	4	beta_min = 0.25 and beta_max = 1.75		Appendix 5 Fig 5.A.1.3

8.B.10. Benchmark UnConDE-B

8.B.10.1 Variant 1: DE /best/1/bin

#	Parameter/ Combination of Parameters Taken for Tuning	Best Result	Tuned Parameter Values	Observations	Reference to the Figure
1	nPop and MaxIt	1	nPop = 300 and MaxIt=500		Appendix 5 Fig 5.A.1.1
2	pCR	5	pCR =0.6		Appendix 5 Fig 5.A.1.2
3	Variable Range	9	VarMin = -1 and VarMax =1		Appendix 5 Fig 5.A.1.4
4	Beta Range	2	beta_min = 0.25 and beta_max = 1.25		Appendix 5 Fig 5.A.1.3

8.B.10.2 Variant 2: DE /best/2/bin

#	Parameter/ Combination of Parameters Taken for Tuning	Best Result	Tuned Parameter Values	Observations	Reference to the Figure
1	nPop and MaxIt	1	nPop = 500 and MaxIt=10		Appendix 5 Fig 5.A.21
2	pCR	8	pCR =1		Appendix 5 Fig 5.A.2.2
3	Variable Range	13	VarMin = -2 and VarMax =2		Appendix 5 Fig 5.A.2.4
4	Beta Range	4	beta_min = 0.25 and beta_max = 1.5		Appendix 5 Fig 5.A.2.3

8.B.11. Benchmark ConDE-A

#	Parameter/ Combination of Parameters Taken for Tuning	Best Result	Tuned Parameter Values	Observations	Reference to the Figure
1	nPop and MaxIt	16	nPop = 1000 and MaxIt=500		Appendix 6 Fig 6.B.1.1
2	pCR	20	pCR =1		Appendix 6 Fig 6.B.1.3
3	R	23	R=100		Appendix 6 Fig 6.B.1.5
4	Beta Range	11	beta_min = 0.75 and beta_max = 1		Appendix 6 Fig 6.B.1.2

8.B.12. Benchmark ConDE-B

8.B.12.1 Variant 1: DE /best/1/bin

#	Parameter/ Combination of Parameters Taken for Tuning	Best Result	Tuned Parameter Values	Observations	Reference to the Figure
1	nPop and MaxIt	4	nPop = 30 and MaxIt=500		Appendix 6 Fig 6.B.2.1
2	pCR	6	pCR =1		Appendix 6 Fig 6.B.2.3
3	R	10	R=100		Appendix 6 Fig 6.B.2.5
4	Beta Range	4	beta_min = 0.25 and beta_max = 0.5		Appendix 6 Fig 6.B.2.2

8.B.12.2 Variant 2: DE /best/2/bin

#	Parameter/ Combination of Parameters Taken for Tuning	Best Result	Tuned Parameter Values	Observations	Reference to the Figure
1	nPop and MaxIt	8	nPop = 200 and MaxIt=10		Appendix 6 Fig 6.B.3.1
2	pCR	7	pCR =1		Appendix 6 Fig 6.B.3.3
3	R	10	R = 100		Appendix 6 Fig 6.B.3.5
4	Beta Range	5	beta_min = 0.25 and beta_max = 1		Appendix 6 Fig 6.B.3.2