International Journal of Scientific & Engineering Research, Volume 6, Issue 2, February-2015
ISSN 2229-5518

Comparative Performances of Crossover Functions in Genetic Algorithms

Ezeani M I[*1]; Okonkwo O R; Onyesolu M O; Osita E G
Department of Computer Science, Nnamdi Azikiwe University,
PMB 5025, Awka, Anambra State, Nigeria

Abstract
Genetic Algorithms have been widely applied to various kinds of optimisation problems. In this work, a Genetic Algorithm is designed to solve the three classic numerical optimisation problems – *Rastrigin*, *Schwefel* and *Griewank*. An experiment to observe the comparative performances of five different crossover functions was conducted. Also, the possible effect of *aging* out some of the old individuals from the population was hinted at. A parameter set expected to give the optimal performance and a discussion on the design considerations are presented below.

1.0 Background Introduction
The field of evolutionary computation is a rapidly growing one. Mitchell [Mit96], in discussing the use of evolution as an inspiration for solving computational problems, observed that,

> *..the mechanism of evolution seem well suited for some of the most pressing computational problems in many fields. Many computational problems require searching through a huge number of possibilities for solutions.*

Genetic Algorithm is one of the most popular techniques used in evolutionary computation [Gol89]. However, designing and tuning a Genetic Algorithm to solve such problems have always involved trade-offs. This is often because the nature of the problem and desired quality of the result are often weighed

[1]*Corresponding Author: igezeani@yahoo.com

alongside the available resources for solving it. An overview of Genetic Algorithms is presented in the next section as well as the general discussion on the problem and approaches taken to tackle it.

Genetic Algorithms (GAs)

GAs are population based search algorithms, originally developed by John Holland (1975), based on the principles of the *Darwinian Theory of Evolution and Natural Selection* [Whi05]. They are currently among the most widely used heuristic approaches to multiobjective optimisation. A GA is said to be population based because it works with a population of individual solutions. These individual solutions are data structures encoded in *chromosome*-like forms i.e. a concatenation of *genes*. The gene values could be bit-strings, real values or symbols.

The first step in the canonical algorithm is to randomly generate individuals that form the *initial* population at generation *g:=0*. It then loops through the processes of *selection* of parents for the intermediated population (or *mating pool*); *crossover* and *mutation* that produces new offspring and finally, *replacement* of the parent population with the offspring population. This is done over a specified number of *generations* and stops when a certain target solution quality is reached or when any other termination criterion is met. Issues concerning the choice of the main operations of selection, crossover, mutation and replacement will be discussed in *section 3: Methodology*. Below is a sample structure of a typical GA:

1. *Generate (P(0))*
2. *t := 0*
3. *while not Termination_Criterion(P(t))*
4. *do*
5. *Evaluate(P(t))*
6. *P'(t) := Selection(P(t))*

7. P'(t) := Recombination(P'(t))
8. P'(t) := Mutation(P'(t))
9. P(t+1) := Replace(P(t), P'(t))
10. t := t+1
11. return Best_Solution_Found

2.0 Task Description

The summary of the task in this assignment is to design and tune a single **robust** algorithm that can solve the three classic numerical optimisation test problems – *Rastrigin*, *Schwefel* and *Griewank* – up to given targets. This algorithm should be able to solve **all** the three problems using the **same set of initial configuration parameters**. **Java** is expected to be used for the development. The number of **evaluations** of the objective functions required to solve the three numeric functions should not exceed **30,000** and the **performance** should be improved as much as possible. Any additional feature implemented or experimented on may as well count for the overall assessment.

Overview of the functions (Rastrigin, Schwefel and Griewank)

The above given functions are classics that are often used as global numerical optimisation test problems. A brief overview of each of the functions in the context of our problem is presented below.

Rastrigin

- *Function definition:* $\boldsymbol{f(x) = an + \sum_{i=1}^{n}[x_i^2 - a\cos(bx_i)]}$
- *Number of variables, n:* **n = 20**
- *Search domain for x_i:* $\mathbf{-5.12 \leq x_i \leq 5.12,\ i = 1, 2, \ldots, n.}$
- *Value of a:* **10**
- *Value of b:* **2π**
- *Number of local minima*: **many**
- *Actual global minima*: $\boldsymbol{x^* = (0, \ldots, 0),\ f(x^*) = 0.}$
- *Set target:* **< 0.9**

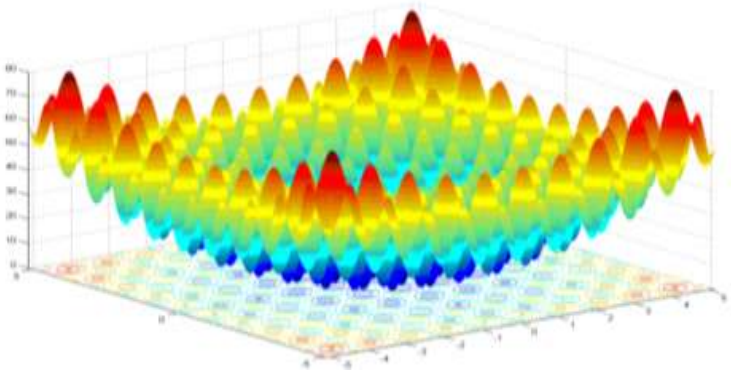


Fig 1: Rastrigin Function Graph for n=2

Schwefel

- *Function definition:* $f(x) = \sum_{i=1}^{n} -x_i \sin(\sqrt{|x_i|})$
- *Number of variables, n:* **n = 10**
- *Search domain for x_i:* **$-500 \leq x_i \leq 500$, i = 1, 2, . . . , n.**
- *Number of local minima*: **many**
- *Actual global minima:* ***x* = (1, …, 1), f(x*) = 0.***
- *Set target value:* **< − 4187.5**

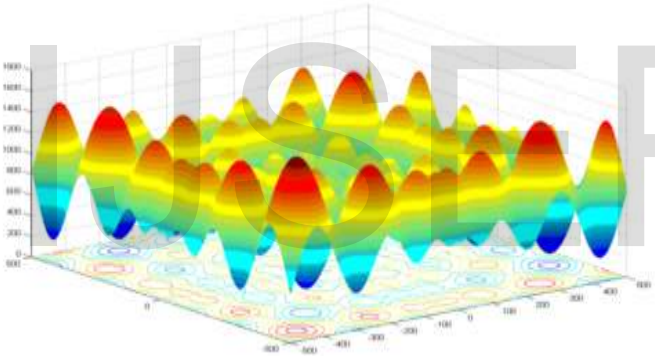


Fig 2: Schwefel Function Graph for n=2

Griewank

- *Function definition:* $f(x) = 1 + \frac{\sum_{i=1}^{n} x_i^2}{4000} - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right)$
- *Number of variables, n:* **n = 10**
- *Search domain for x_i:* **$-600 \leq x_i \leq 600$, i = 1, 2, . . . , n.**
- *Number of local minima*: **many**
- *Actual global minima*: ***x* = (0, …, 0), f(x*) = 0.***
- *Set target value:* **< 0.1**

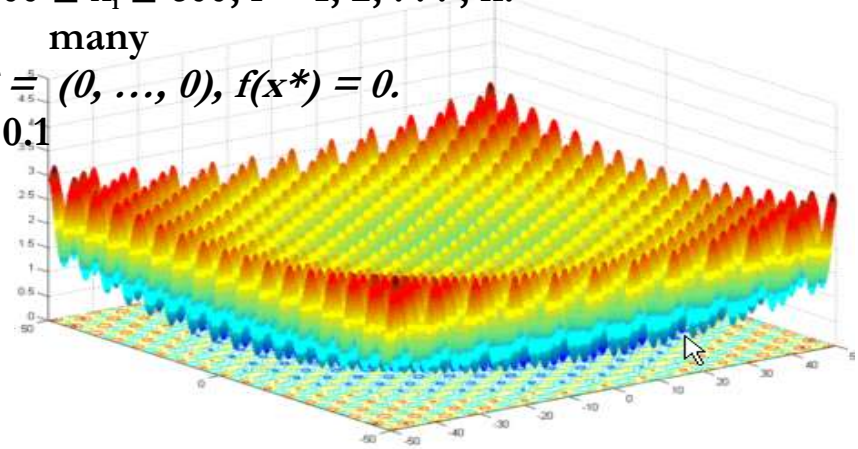


Fig 3: Griewank Function Graph for n=2

3.0 Methodology

This experiment was conducted with an already developed algorithm for the test problems. However, it was modified to introduce additional crossover operators and the *age* operator. The major tasks carried out in this work were tuning the algorithm to obtain a set of configuration parameters that will achieve the desired aim i.e. solving each of the problems to target and within the total maximum number of evaluations and then with the solution parameter set, the tests on the crossover functions and the *age* function were conducted. The structure of the algorithm used including its operators will be briefly presented and then followed by the experiment procedure, results and discussions.

The Algorithm

The GA performance is often determined by the choice of operators and parameters made during the design. The following defines the main aspects of the design of the GA used in this work.

Representation

Although most canonical[2] GAs use binary encoding of chromosomes, other types of coding are possible. Real valued coding was used in this work. A good justification for that may be drawn from the works of [Wri91] and [ScE93].

Selection

This algorithm used tournament selection to pick the parents that will be recombined from the population specifically for the *BLX-0.5*, *Linear* and *Uniform* crossovers. Research has confirmed that this selection scheme can effectively improve performance [GoD91]. However, for the two other crossovers operators – *Multi-Parent (1)* and *Multi-Parent (2)* – normal random

[2] The original structure of GA as proposed by Holland is often referred to as the canonical GA

selection was adopted because this approach seemed more explorative and yet produced good results.

Crossover operators

There were five different crossover operators featured in this work. The first (BLX-0.5 and Linear) two are typically used with real values because they also can introduce new information into the search space by producing new allele values. The others (multi-parents and uniform) can recombine based on the already existing information. They do not introduce any new traits into the population and so a more prominent with binary coding. All the five operators are discussed under Testing the crossover operators in section 3.

Mutation

Probabilistic creep mutation described by [Wri91] which is less disruptive than replacing a gene completely with a new random real value, was used. A perturbation is likely to increase or decrease the gene value but the maximum creep size was varied to obtain results during the tuning.

Replacement

The steady state replacement scheme of the *extinction of the worst* was used. Only one offspring is produced by the crossover and mutation and, if better, the new offspring replaces the worst individual in the population.

The Experiment: Tuning the Algorithm

The problem of selecting an optimum set of parameters for the GA is complex itself [Vin03]. The two main approaches to this are often building an adaptive GA or some kind of heuristic trial and error method. The latter was largely applied in this work.

Testing for Creep Size and Distribution (theta)

A quick evaluation of the performance of the mutation rate on the three functions was made (fig 4.) and the result indicated that Schwefel did not work at all but for some occasional noise at some points. Rastrigin was also shown to be sensitive to the mutation rate. However, mRate of 0.1 was used to examine the effect of maxCreep and mDist (Theta).

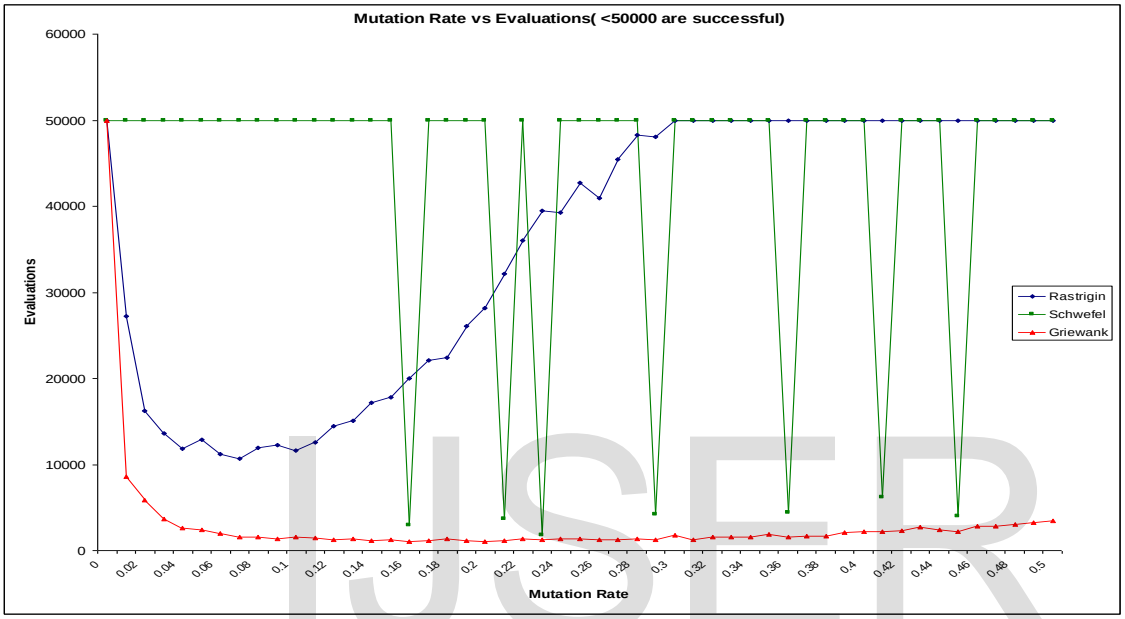


Fig 4 Max Evals: *50,000* Population Size: *20* Over Init: *100* Max Creep: *0.5* Theta: *5*

So for the three functions, the *maxCreep* and the *theta* were varied and the graph of the average number of evaluations that solved the problem with the maximum number of evaluation being 30, 000 was plotted. (*ref. to Appendix I for the data and figs 5,6,7*)

Rastrigin gave good results for higher values of *theta* (8 – 15) and lower values of *maxCreep* (0.10 – 0.7). However, Schwefel seemed to be affected more by *maxCreep* than *theta*. Lower values of *maxCreep* generally produced bad results with occasional cases of exceptionally good performances (noise). Also, higher

values of *maxCreep* (0.7 and above) almost guaranteed above average performance and helps the algorithm to better explore the search space and not get trapped in a local optima.

Griewank is somewhat different from the other two. Although, it gave comparatively good performance, the best results clustered around lower values of *maxCreep* and the mid values of *theta*. (*see Appendix I*)

The experiment, as well as the graphs, shows that there are quite a lot more issues with the nature of the individual functions. The target for Griewank can easily be met but with a combination of the other two functions makes the problem a little more demanding.

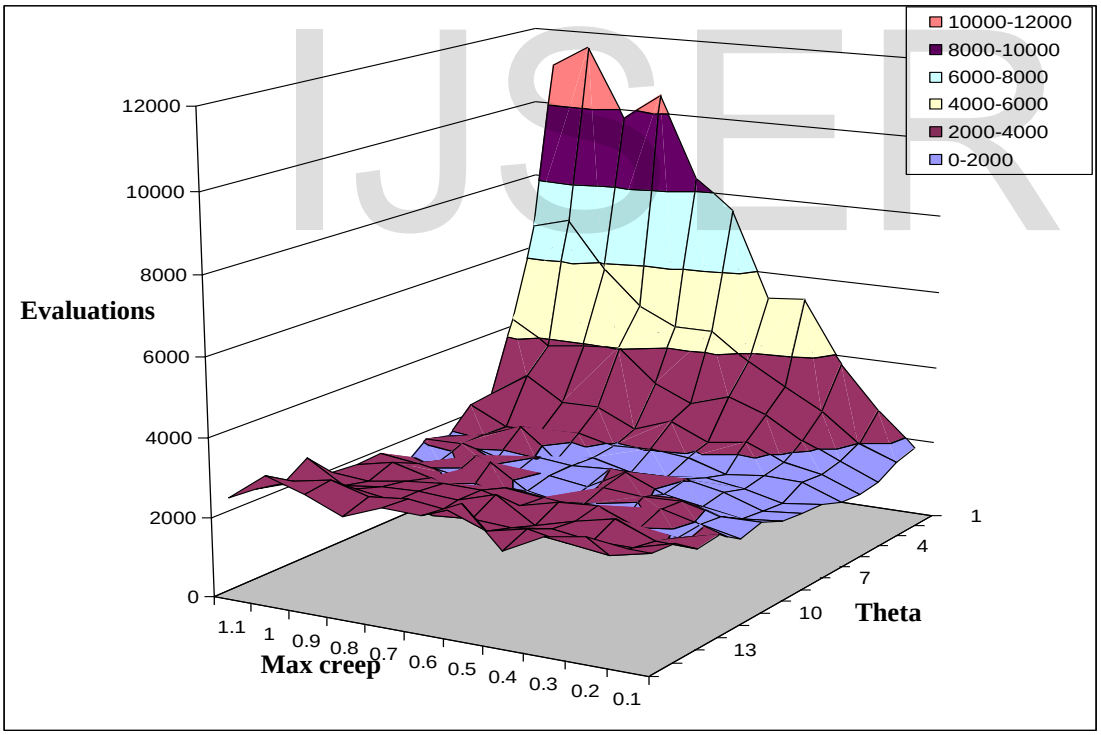


Fig 5: **Griewank: Max-Creep + Theta:** mRate: 0.1; popSize: 15; tSize: 2

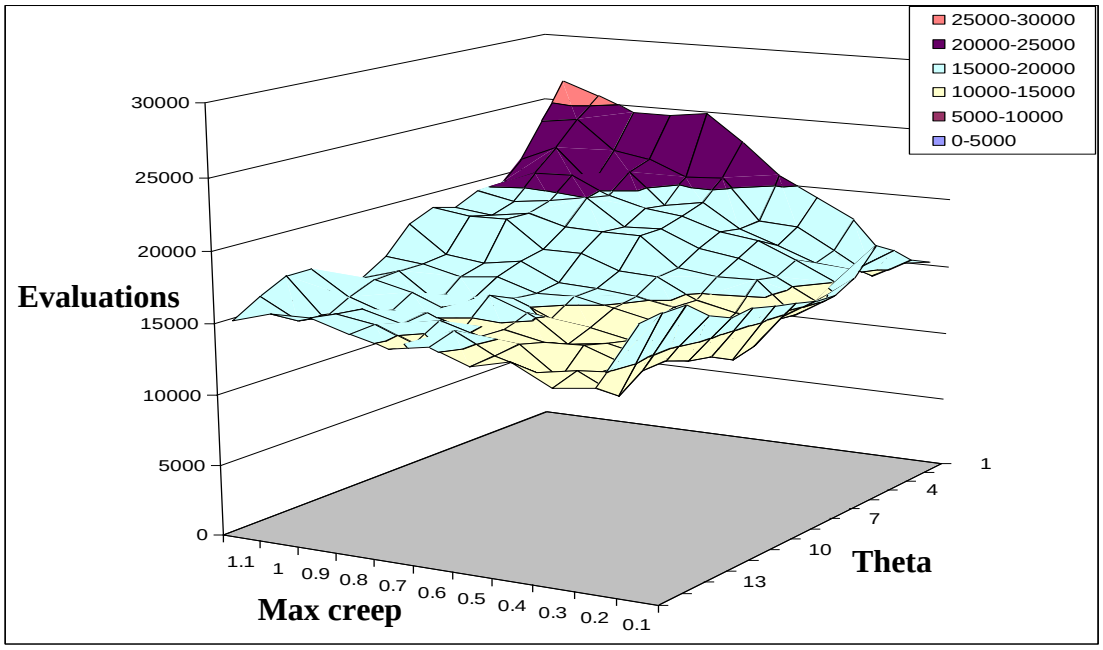


Fig 6: **Rastrigin: Max-Creep + Theta:** mRate: 0.1; popSize: 15; tSize: 2

However, tracing the data table Appendix I, the values identified were:

- theta = 9 or **10**
- maxCreep = between 0.8 and **0.9**

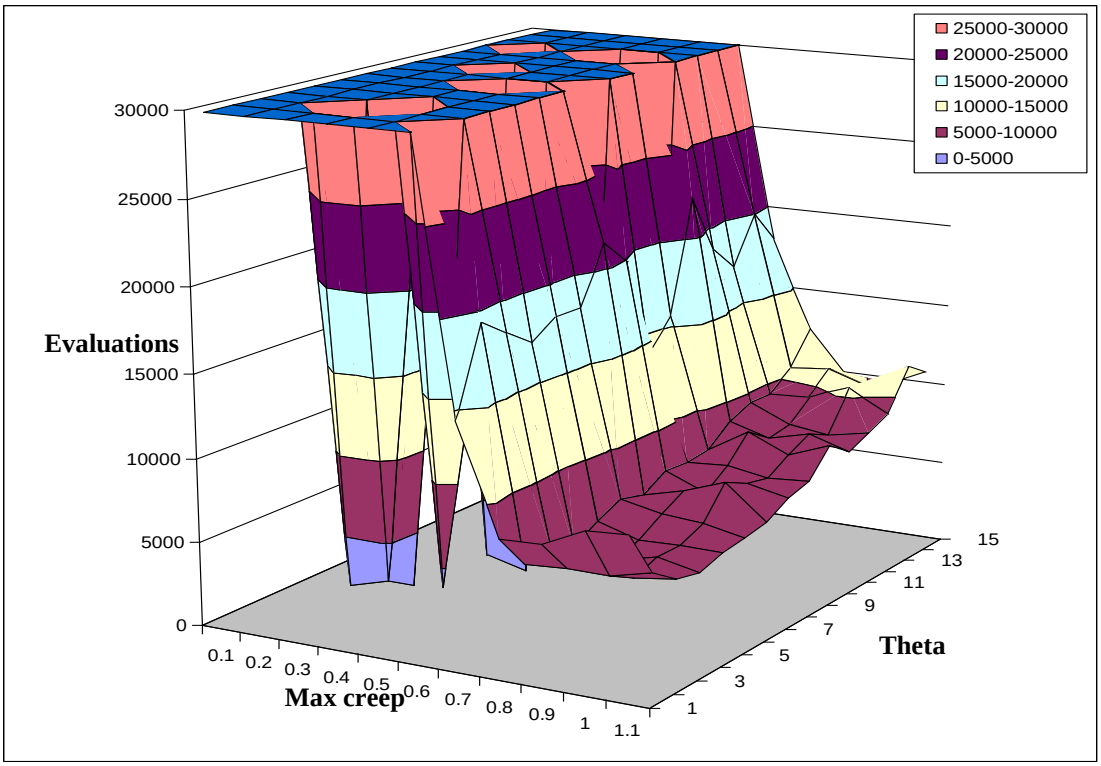


Fig 7: **Schwefel: Max-Creep + Theta:** mRate: 0.1; popSize: 15; tSize: 2

Fine-tuning the initial values (*popSize*, *tSize* and *mRate*)

popSize

With the parameters for *theta* and *maxCreep* as shown above, values for *popSize* were tested. For the results, Schewefel and Griewank showed relative stability across varying *popSize* values but Rastrigin was obviously more sensitive to varying population sizes (*Fig 9*). However, for their combined number of evaluations, relatively good performance was observed for *popSize* values between 5 and 25 (*Fig 9*).

It is noteworthy though that in all these tests, the *overInit* remained constant i.e. 100, but it is believed to have some effects on the starting average quality of the population. Also, considering that the experiment is being run to target, accurate measurement of the quality of the solution was not part of the work.

tSize

With a population size of 10, the overall performance of the three functions in terms the number of evaluations was best at *tSize* = 7 (*see fig 10*). Fig 11 shows their individual performances in terms of the number of evaluations. Considering the configuration set being used, the likely value for *tSize* could be anything between 6 and 8.

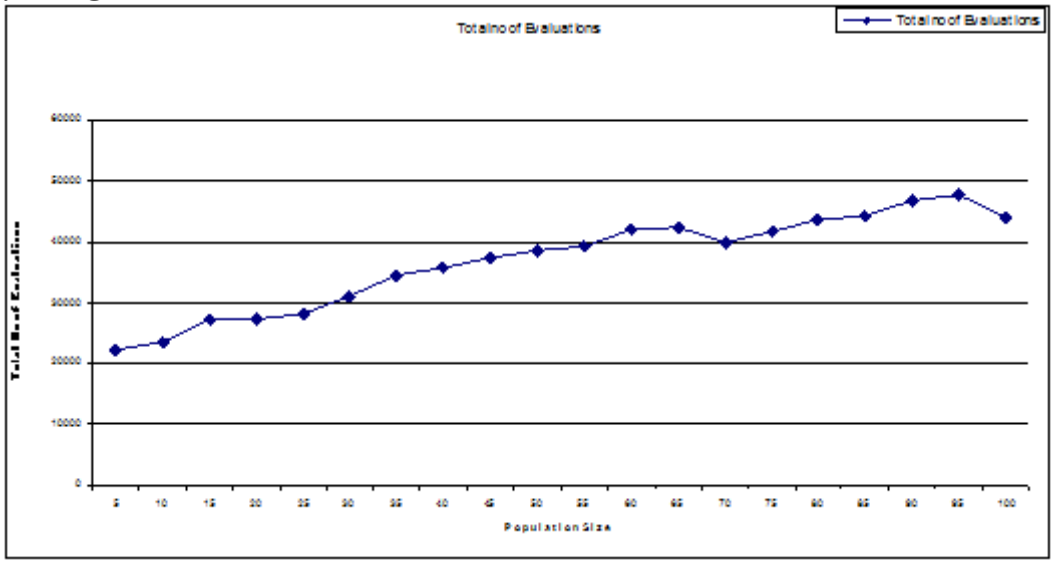


Fig 8: **Total Evaluation vs Population size:** mRate: 0.1; tSize: 2

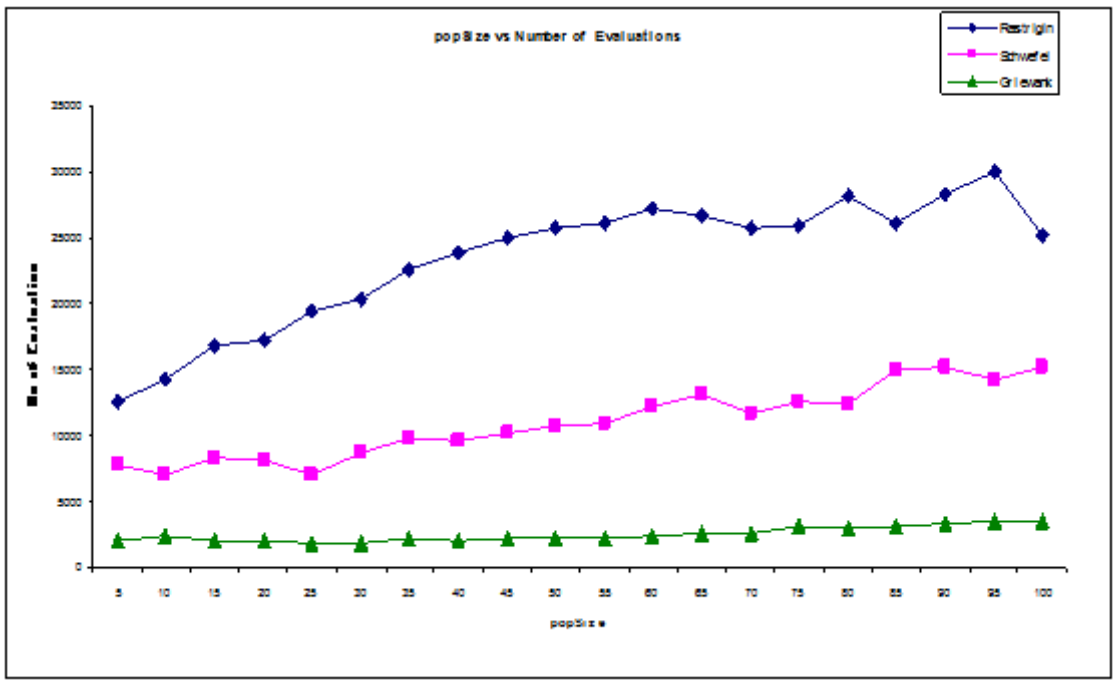


Fig 9: **Evaluation vs Population size:** mRate= 0.1; tSize= 2

mRate

The actual effect of the mutation rate was the last to be tested with the configuration parameter values derived from the above experiments. *popSize*, *tSize*, *theta* and *maxCreep* were kept at 10, 7, 10, 0.9 respectively. The combined performance of the three functions in terms of evaluations seemed best at points between 0.8 and 0.16. However, the least number of evaluations was best at 0.12 and 0.16.

Also Schwefel and Griewank were tending toward reduced number of evaluations as the mutation rate increased but the performance of Rastrigin was not guaranteed (a shot at it actually showed worse performance). Generally, a mutation rate of 0.12 will be ideal considering the data collected. *See Table 1 and Figs 12, 13.*

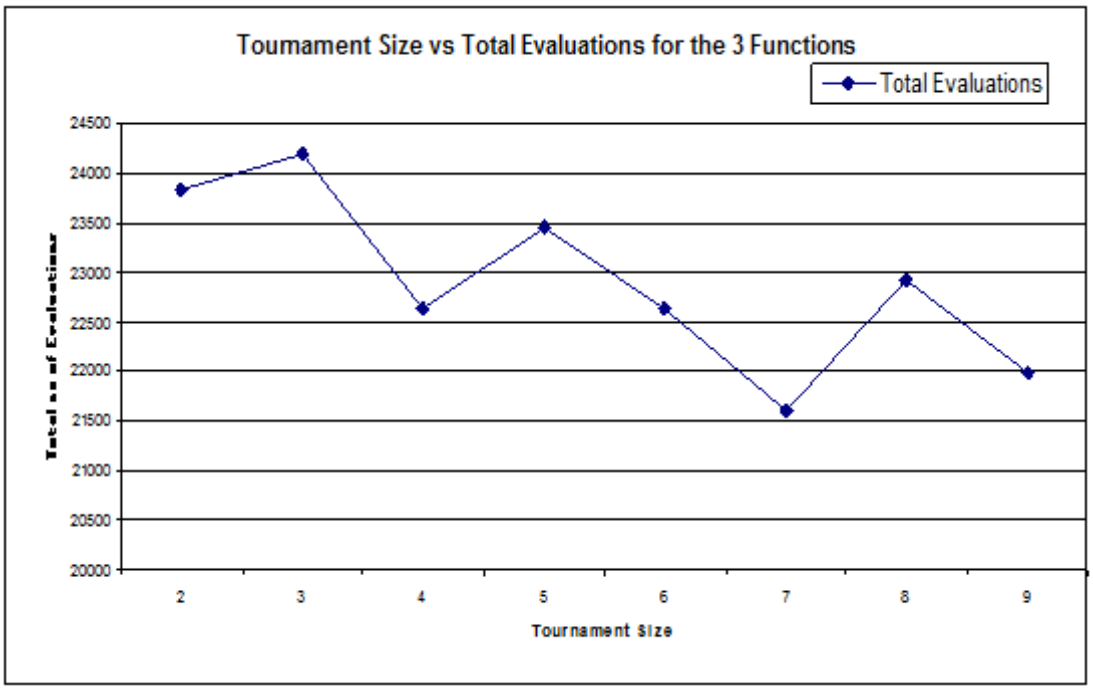


Fig 10: **Total Evaluation vs Tournament size:** mRate: 0.1; tSize: 2, popSize = 10

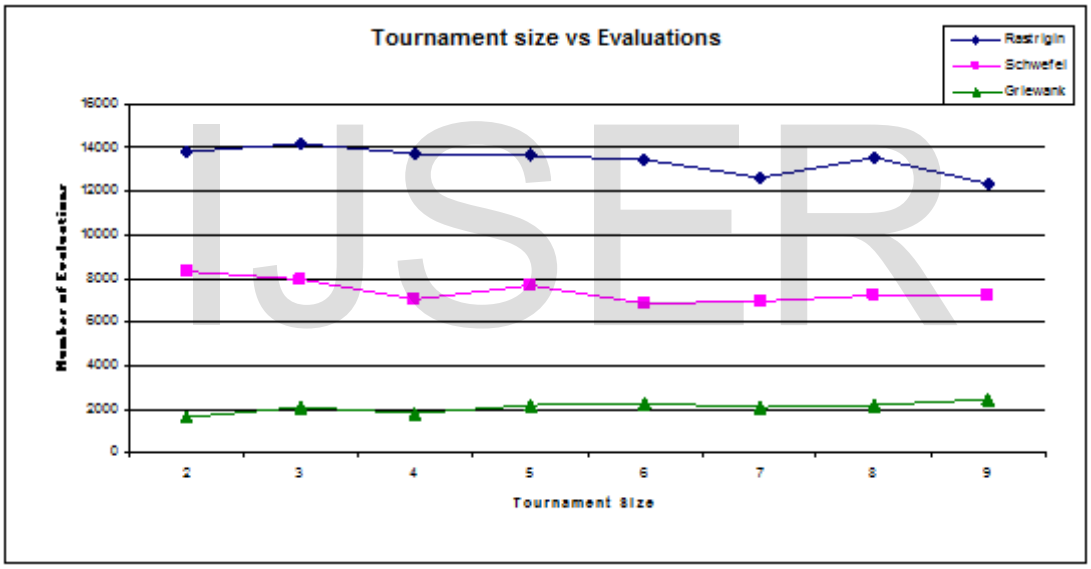


Fig 11: **Evaluations vs tournament size:** mRate: 0.1; popSize = 10

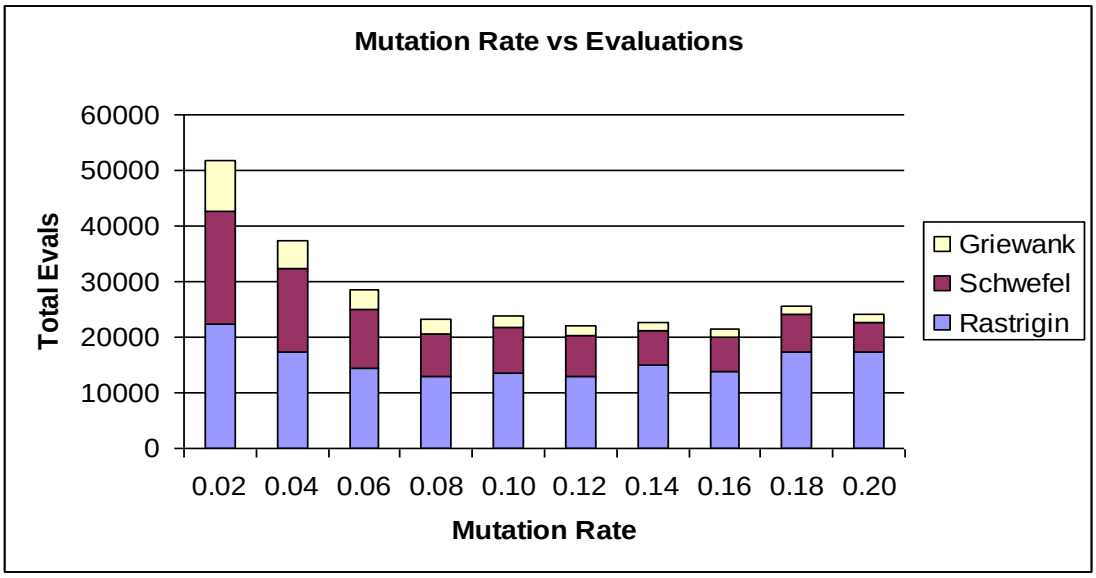


Fig 12: **Total Evaluations vs Mutation Rate:** tSize: 7, popSize = 10

Table 1: mRate vs Evaluation

mRate	Rastrigin	Schwefel	Griewank	Total Evals
0.02	22387	20314	9024	***51725***
0.04	17291	15112	5011	***37414***
0.06	14534	10413	3467	***28414***
0.08	12943	7648	2764	***23355***
0.10	13623	8003	2125	***23751***
0.12	12930	7443	1602	***21975***
0.14	15142	6125	1468	***22735***
0.16	13824	6186	1418	***21428***
0.18	17388	6721	1611	***25720***
0.20	17329	5399	1413	***24141***

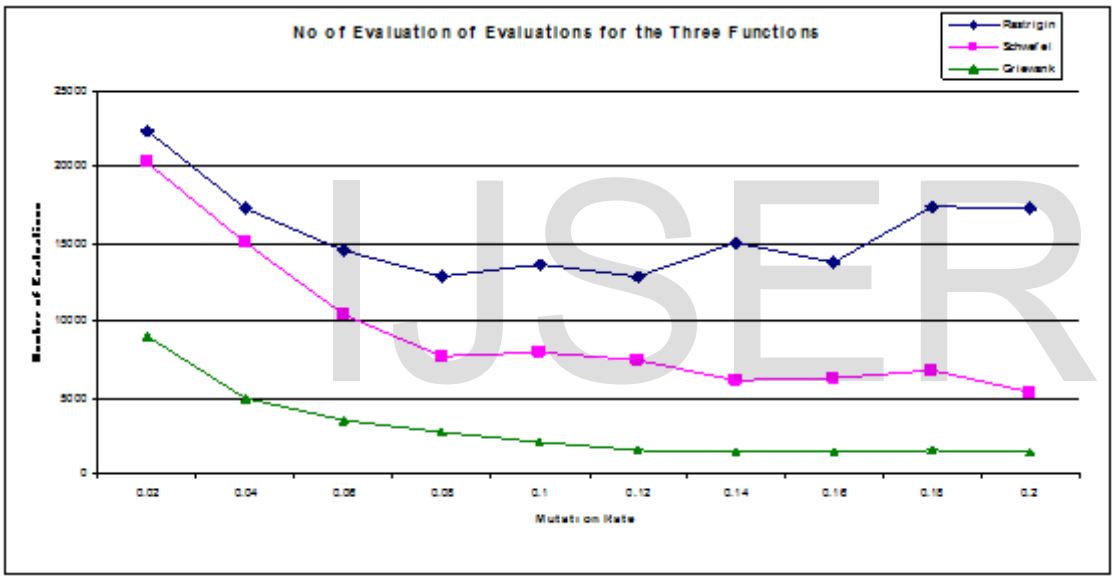


Fig 13: **Evaluations vs Mutation Rate:** tSize: 7, popSize = 10

Testing the crossover operators

The extended blend crossover – BLX-0.5

This is one of the commonly used crossover techniques for real number coding and it was used in this algorithm for the initial tuning phase. Point and uniform crossovers often used with bit-string representations do not introduce new information into the search [Vin03]. For this operator, the new offspring, **z**, is defined from the parents **x** and **y** as: -

$$\boldsymbol{z = x + (2R - 0.5) * (y - x)}$$ *where* R = *random real number*

A sample code is shown below

```
...
if (xoverType == 1){
/* blx-0.5 crossover*/
for (int i = 0; i < nPars; i++)
    offspr.gene[i]=parent1.gene[i]+(2*rand.nextDouble()-0.5)*(parent2.gene[i]- parent1.gene[i]);
}
...
```

Linear crossover

Another operator used with real number coding – Linear crossover – was also introduced. Although this promised good results, it was not efficient because it requires additional evaluations. It works by producing three offspring from two parents and selection the best among them and this causes extra two evaluations for any one replacement. For instance, the three offspring **z1, z2, z3** produced by the parents **x** and **y** are defined by

$$\boldsymbol{z1 = \left(\frac{x}{2}\right) + \left(\frac{y}{2}\right); z2 = 3\left(\frac{x}{2}\right) - \left(\frac{y}{2}\right); z3 = 3\left(\frac{y}{2}\right) - \left(\frac{x}{2}\right)}$$

A sample code is as shown below

```
...
else if(xoverType == 2){
        /* linear crossover*/
        for(int i = 0; i < nPars; i++){
                offspr. gene [i] = parent1. gene [i]/2 + parent2. gene [i]/2;
                off2. gene [i] = 3*parent1. gene [i]/2 - parent2. gene [i]/2;
                off3. gene [i] = 3*parent2. gene [i]/2 - parent1. gene [i]/2;
        }
        // select one of the three: 2 extra evaluations
        if (offspring.evaluate() > off2.evaluate()) offspring = off2;
        if (offspring.eval>off3.evaluate()) offspring = off3;
} // end linear crossover
...
```

Multi parent crossover (1)

In the first version of the multi – parents crossover used, the number of parents used varies and so does the length of the genes each contributes. Parents are selected at random for as long as there is an 'unfilled' gene position

still remaining on the new offspring. Crossover points are determined by adding up the lengths of genes contributed by different parents. The sample code below illustrates the concept.

```
else if(xoverType == 3){
   // multi parent crossover(1): Some parents are used.
   int selLength;
   for(int x_point=0;x_point < nPars;){        /* start from the 1st gene*/
     // make new offsprings with some parents <= gene length
     parent1 = population[rand.nextInt(popSize)]; //get a parent
     selLength = rand.nextInt(nPars-1);           // get a point on the parent
     if (x_point + selLength > nPars-1)
         selLength = (nPars) - x_point ;
     System.arraycopy(parent1. gene,x_point, offspring. gene, x_point,selLength);
     x_point += selLength;                       //get the new crossover point
   } //for
}
```

Multi parent crossover (2)

This is *multi-parent complete*. In this version of the multi–parent crossover, the number of parents used is more determined. In fact, as many parents as the gene length are used such that each gene value is contributed by a newly selected parent. Selection is also basically random not biased to better parents. See the code segment.

```
...
else if(xoverType == 4){
   // multi parent crossover(2): No of parents = gene length.
   int selGene;
   for(int x_point=0;x_point < nPars;){        /* start from the 1st gene*/
     // make new offsprings with some parents = gene length
     parent = population[rand.nextInt(popSize)]; //get a parent
     selGene = x_point;                       // get the gene this parent donates
     System.arraycopy(parent.gene, x_point, offspring. gene, x_point,selLength);
     x_point += selGene;                         //get the new crossover point
   } //for
}
...
```

Uniform crossover

In uniform crossover, two parents were used and they were picked with tournament selection. A mask Boolean pattern is then randomly generated to

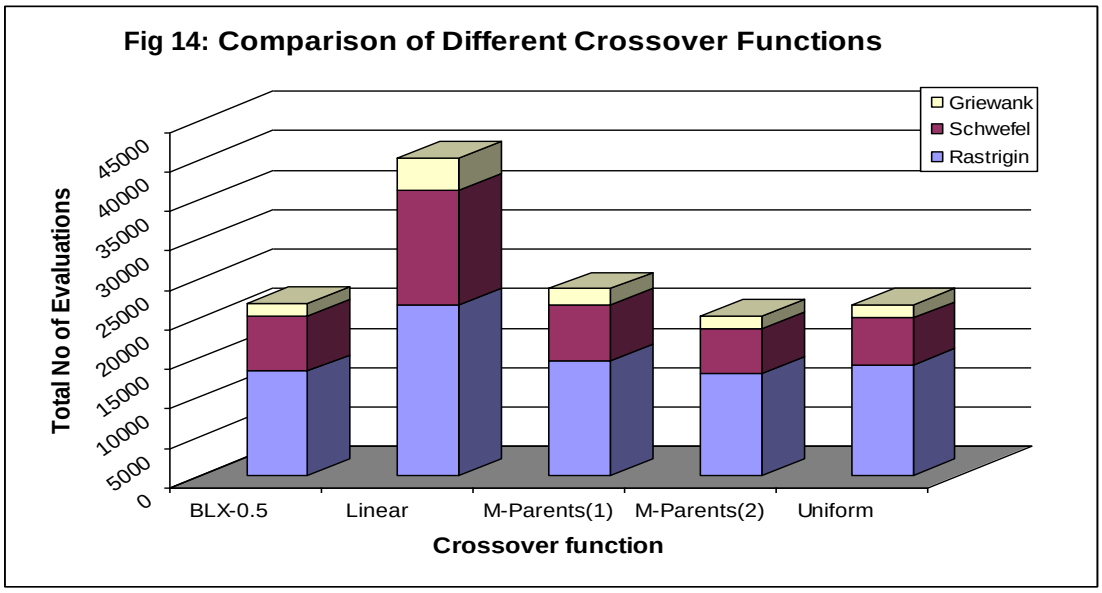
aid in deciding which parent donates which gene in producing a single[3] offspring. An illustration and a sample code segment of two parents, **x** and **y,** producing an offspring, **z**, are given below.

```
...
else if(xoverType == 5){
   //for uniform cross over
   boolean [] mask = new boolean [nPars];
   for (int i=0; i<nPars; i++) mask[i] = rand.nextBoolean(); //generate the Boolean values
   for(int x_point=0;x_point<nPars;x_point++){
      //decide which parent contributes the gene for the new offspring
     if (mask[x_point])
       System.arraycopy(parent1.gene,x_point, offspring.gene,x_point,1);
     else
       System.arraycopy(parent2.gene,x_point, offspring.gene,x_point,1);
    }//end of for
}//uniform crossover
...
```

The analysis of the data from the comparison is as shown below.

Table 2 Crossover vs Evaluations

Crossover	Rastrigin	Schwefel	Griewank	Total
BLX-0.5	13215	6927	1621	**21763**
Linear	21543	14610	3991	**40144**
M-Parents(1)	14567	7069	2003	**23639**
M-Parents(2)	12917	5707	1652	**20276**
Uniform	14052	5874	1746	**21672**



[3] Actually, two offspring can be generated but this work considered only one.

Expectedly, linear crossover, which has an additional two evaluations, was the worst in performance. However, the others were within the same range of performance with the BLX-0.5 but it is interesting to note that without any bias in the selection of parents (e.g tournament), the multi-parent functions comparatively well.

The *age* function

One other function that seemed interesting to look at was the *age* function. The whole concept is about elimination individuals that have lasted in the population beyond a certain age limit (number of evaluations). The expectation is that replacing them with new offspring might improve the explorative aspect of the search. However, this experiment did not cover this in detail and therefore cannot give very sound report on its effect. The data (below) generated did not reveal much.


```
...
public void age(Individual [] individuals, int maxAge){
   for (int i = 0; i < popSize; i++) {// go through the entire population
               individuals[i].age++;       // increment age by 1
        if (individuals[i].age > maxAge){   // check if the age limit is exceeded
               Individual parent;
               for(int x_point=0;x_point<nPars;x_point++){ // create a new individual with
                 parent = population[rand.nextInt(popSize)];// multiple parents and make its
                                                            //age = 0
                 System.arraycopy(parent.gene,x_point, individuals[i].gene, x_point,1);
               }
                 individuals[i].age = 0;
        }//if
   }//for
}//age...
```

Table 3 Age Limit vs Evaluations

Age Limit	Rastrigin	Schwefel	Griewank	Total
5000	13501	7266	2119	***22886***
10000	11803	7597	2102	***21502***
15000	13415	7419	1939	***22773***
20000	13152	6437	2208	***21797***

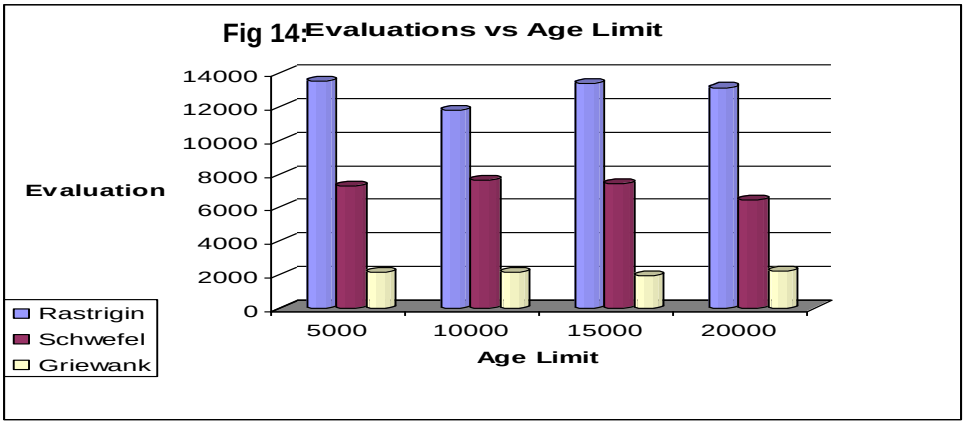


Fig 14: Evaluations vs Age Limit

4.0 Conclusion

Generally, Rastrigin required far more time than the other two to reach the target. In summary, the optimum set of configuration parameters that was derived from this experiment are:

Population size: **10**
Tournament size: **7**
Mutation rate: **0.12**
Max Creep: **0.9**
Max Distribution (Theta) **10**
Crossover: **BLX-0.5**, the "**Multi-Parent"**s, **Uniform**

The above set of configuration parameter values consistently guarantees a total number of evaluations less than 23,000 in meeting the required targets on all the three functions. The linear crossover naturally would not do well compared to the others if the key assessment factor is *number of evaluations*. But others performed well enough.

However, one of the weaknesses of this experiment is not considering the effect of other parameters on the performance of the crossover operator. Also BLX-0.5 remains the best choice for its ability to explore the search space better by producing new allele values. This goes also in considering the effect of the *age* function. Although, the age function did not show any interesting

performance pattern, it may be worth probing further into it. Future works on this need to explore those areas.

5.0 References

[GoD91] Goldberg D, Deb K, 1991. ***A Comparative Analysis Of Selection Schemes Used In Genetic Algorithms***, *Foundations of Genetic Algorithms,* ed. Morgan Kaufmann (ISBN 15558601708)

[Gol89] Goldberg D. 1989. ***Genetic Algorithm in Search, Optimisation and Machine Learning***. Canada, Addison Wesley.

[Mit96] Mitchell M, 1996. ***An Introduction to Genetic Algorithms.*** London, The MIT Press

[Pin05] Pintér, JD, 2005 ***Applied Nonlinear Optimization in Modeling Environments: Using Integrated Modeling and Solver Environments***. Boca Raton, FL: CRC Press, 2005

[ScE93] Schaffer JD, Eshelman L. 1993 ***Real-Coded Genetic Algorithms and Interval Schemata*** *Foundations of Genetic Algorithms*. D. Whitley ed. Morgan Kaufmann.

[Vin03] Vincent, J. 2003**, *Numerical Optimisation Using Genetic Algorithms*** [online] School of DEC, Bournemouth University. Available from: *http://dec.bournemouth.ac.uk/staff/ jvincent/teaching /ec/numerical_optimisation_using_ genetic_algorithms.pdf* [Accessed 08 April 2006].

[Whi05] Whitley D. 1994, ***A Genetic Algorithm Tutorial*** – Statistics and Computing 4, 65–85

[Wri91] Wright A 1991 ***Genetic Algorithms for Real Parameter Optimisation***. *Foundations of Genetic Algorithms*. G. Rawlins ed. Morgan Kaufmann.

6.0 **Appendix I:** showing data collected on *Theta*, *mCreep* and *No of Evaluations* for three functions

	PSize		TournamSize		No of Trials	Mutation Rate =
RASTRIGIN	=	15	=	2	=50	00.1000

Theta

mCreep |	1	2	3	4	5	6	7	8	9	10	11	12	13
0.1	**15207**	**15884**	*17102*	*17928*	*16122*	**15386**	**15538**	**15923**	*16276*	*16670*	*16558*	*17287*	*18514*
0.2	**14876**	**14429**	**15396**	**14964**	**14108**	**13698**	**13860**	**12576**	**12190**	**12958**	**13285**	**13923**	**13838**
0.3	*17904*	*16037*	**15619**	*16277*	**15623**	**15935**	**14050**	**13295**	**14069**	**13834**	**13233**	**13289**	**13443**
0.4	*19260*	*17533*	*16616*	*17464*	*16653*	**15841**	**14652**	**14678**	**14310**	**13491**	**13849**	**14306**	**12608**
0.5	*20736*	*17671*	*17566*	*17844*	*17080*	*16187*	*16018*	**15180**	**14861**	**14021**	**13438**	**14129**	**14586**
0.6	*23002*	*20700*	*19311*	*18700*	*18264*	*18255*	*16157*	**15445**	**15379**	**14371**	**15340**	**14691**	**14808**
0.7	*24892*	*20497*	*18809*	*19039*	*18109*	*18273*	*17138*	**15548**	**15286**	**15505**	*16208*	**14475**	**15560**
0.8	*24517*	*21563*	*21043*	*19693*	*20004*	*18298*	*17160*	*16831*	*16200*	**15231**	**14762**	**15218**	**15001**
0.9	*24519*	*22025*	*19794*	*20989*	*19546*	*19031*	*19083*	*16399*	**15853**	**14606**	**15803**	**14843**	**15723**
1	*25563*	*24177*	*22378*	*20899*	*19187*	*18643*	*19028*	*16697*	*16515*	*16502*	*16400*	**15687**	**14839**
1.1	*26448*	*23787*	*21259*	*19788*	*19766*	*18940*	*19412*	*18674*	*16759*	**15753**	**14559**	*17427*	*17466*
Average	**21539**	**19482**	**18627**	**18508**	**17678**	**17135**	**16554**	**15568**	**15245**	**14813**	**14858**	**15025**	**15126**

Minimum **12190** Maximum **26448** Median 15946

pSize = 15 tSize = 2 No of Trials =50 Mutation Rate = 00.1000

GRIEWANK

Theta mCreep |	1	2	3	4	5	6	7	8	9	10	11	12	13
0.1	**1782**	**1643**	**1361**	**1298**	**1347**	**1548**	**1584**	**1657**	**1912**	**1768**	**2152**	**2094**	*2505*
0.2	*2669*	**1987**	**1775**	**1668**	**1723**	**1430**	**1503**	**1575**	**1901**	**1968**	**1918**	**2118**	*2249*
0.3	*3794*	*2723*	**2149**	**1853**	**1748**	**1644**	**1845**	**1732**	**2135**	**2197**	**1987**	*2514*	2240
0.4	*5421*	*3258*	*2735*	*2241*	**1641**	**1813**	**2017**	**1885**	**2089**	**1618**	*2411*	*2280*	*2370*
0.5	*5334*	*3811*	*3094*	**2098**	**1932**	**1691**	**1949**	*2325*	**2172**	**1972**	*2345*	**2024**	*2241*
0.6	*7596*	*4493*	*2774*	*2519*	**2020**	**1655**	**1983**	**1774**	**1631**	**2029**	**2091**	*2640*	*2310*
0.7	*8360*	*4508*	*3161*	**2156**	**1959**	**1873**	**1931**	**1931**	**1769**	**2125**	*2345*	*2345*	**2231**
0.8	*10503*	*4960*	*4028*	*2646*	**2148**	**1679**	**2010**	**1806**	*2377*	**1759**	*2367*	*2436*	*2434*
0.9	*9769*	*5842*	*3951*	*2611*	**2096**	*2431*	**1845**	**2156**	**1914**	**2096**	*2253*	*2375*	*2599*
1	*11607*	*7057*	*3832*	*3223*	**2037**	**2067**	**2094**	**1876**	**2064**	*2336*	**2107**	*2265*	*2524*
1.1	*11037*	*6811*	*4439*	*2593*	*2531*	**1893**	**2059**	**1604**	**2166**	**2135**	**2202**	*2830*	*2400*
Average	**7079**	**4281**	**3027**	**2264**	**1926**	**1793**	**1893**	**1847**	**2012**	**2000**	**2198**	**2356**	**2373**
Minimum	**1298**		Maximum	11607	Median	2240							

Note: In *red italics* are the values above the median of all the values while **bolded blue** are values below the median

SCHWEFEL	pSize =	15	tSize =	2	No of Trials =50		Mutation Rate = 00.1000					
Theta mCreep|	1	2	3	4	5	6	7	8	9	10	11	12	13
0.1	*30000*	*30000*	*30000*	*30000*	*30000*	*30000*	*30000*	*30000*	*30000*	*30000*	*30000*	*30000*	*30000*
0.2	*30000*	*30000*	*30000*	*30000*	*30000*	*30000*	*30000*	*30000*	*30000*	*30000*	*30000*	**3056**	*30000*
0.3	*30000*	*30000*	**2057**	*30000*	*30000*	*30000*	*30000*	*30000*	**1219**	*30000*	*30000*	*30000*	*30000*
0.4	*30000*	*30000*	**2661**	**1719**	*30000*	*30000*	**1505**	*30000*	*30000*	*30000*	*30000*	**2247**	*30000*
0.5	*30000*	*30000*	*30000*	*30000*	*30000*	*30000*	**903**	*30000*	*30000*	*30000*	*30000*	**1199**	*30000*
0.6	*30000*	**3874**	*30000*	*30000*	*30000*	*30000*	*30000*	**1728**	*30000*	*30000*	**1492**	*30000*	*30000*
0.7	**14519**	**19388**	**18261**	**17229**	**18177**	**18158**	**21487**	**19883**	**13796**	**15659**	**22363**	**18775**	**17129**
0.8	**8179**	**6029**	**6461**	**5692**	**6523**	**7136**	**6765**	**7578**	**7508**	**7647**	**8829**	**9423**	**9368**
0.9	**8234**	**6161**	**5666**	**5289**	**5759**	**6659**	**7395**	**6625**	**7647**	**9151**	**8093**	**8357**	**9536**
1	**9436**	**6171**	**5345**	**5855**	**5860**	**6450**	**8251**	**7578**	**7239**	**8056**	**8754**	**8086**	**10469**
1.1	**9560**	**6754**	**5666**	**5326**	**5788**	**6008**	**6235**	**7066**	**7405**	**8965**	**7969**	**8532**	**9159**
Average	**20903**	**18034**	**15102**	**17374**	**20192**	**20401**	**15686**	**18223**	**17710**	**20862**	**18864**	**13607**	**21424**
Minimum	**903**		Maximum	**30000**	Median	25200.5							

Note: In *red italics* are the values above the median of all the values while **bolded blue** are values below the median

7.0 Appendix II: A version of the GA code used for the experiment

Evolution.Java runs the different crossover function for any of the functions selected with the pre-predefined parameters. Evolution_Age.Java is the same only that it runs the with different *age limits*. The MersenneTwisterFast.Java is the random number generator class and Individual.Java defines the structure of each individual in the population.

There are also two output file that are generated after each run: result.txt which has the details of the run and data.txt which keeps the summary.