AN OPEN-SOURCE POPULATION INDIFFERENCE ZONE-BASED ALGORITHM FOR SIMULATION OPTIMIZATION

David N. Vuckovich

The Ohio State University
Dept. of Integrated Systems Engineering
1971 Neil Avenue – 210 Baker Systems
Columbus, OH 43210, USA

ABSTRACT

This paper proposes an open-source algorithm for simulation optimization. The intent is to permit many who use a variety of simulation software codes to be able to apply the proposed methods using an MS Excel-Visual Basic interface. First, I will begin by discussing simulation optimization and its usefulness. I will then discuss different methods that are commonly used for simulation optimization. Next, I will present the proposed Population Indifference Zone (PIZ) algorithm and related software code. I will discuss the properties of the proposed method and present the code that runs the Visual Basic program. I will then discuss the functionality of the Population Indifference Zone method with examples of problems to which it might be applied. I conclude with a description of topics for future research.

1 INTRODUCTION

Discrete event simulation continues to grow in popularity presumably because it helps increase profitability through the avoidance of problems in business expansion and manufacturing. Simulation allows for the testing of how system parameters (factors) impact system performance (responses) without costly and time consuming testing on the actual systems. Simulation optimization is the systematic search for factor settings that derive the most desirable expected response values. Interest in simulation optimization also continues to grow perhaps because increasing computer power makes solving large simulation optimization problems computationally feasible (Zhao and Sen 2006).

More generally, after a simulation code has been built and validated, it is then studied to provide decision-support. These phases are often called "output analysis" and generally involve some form of optimization (Fu, Glover, and April 2005). As a motivating example, consider a problem at a major automotive manufacturer that is evaluating alternative policies for running a major production line. Examples of factors in the simulation optimization study include work-in-process management policy (off or on), the conveyor speed at the wash substation, and the transfer time for units at a major substation. By varying these factor settings, a large number of alternative systems are developed and studied using the simulation codes. Because each evaluation is noisy, there is generally a concern about how many replicates of each run should be used to gain useful information and (hopefully) derive the optimal system or combination of factor settings.

Simulation optimization methods can be divided into two types: (1) problem specific and (2) generic or "black box" methods (Zhao and Sen 2006). The problem specific methods usually require less time to find the a solution, but are generally more difficult to apply in simulation optimization because they require the attention of expert operations researchers. Sometimes problem generic methods can be relatively inefficient because they involve comparison of solutions that give very similar average response values. Such comparisons are computationally costly particularly when each simulation takes a long time (Fu, Chen, and Shi 2008). On the other hand, black box methods can be applied to many types of problems and can be used by anyone with limited knowledge of simulation.

Black box methods can further be divided into methods based on constant sample sizes and systemically varying sample sizes. There are multiple types of black box methods including tabu searches, scatter searches, and genetic algorithms. In the context of constant sample size methods, tabu searches and scatter searches have been established as the most effective according to many authors (Fu, Glover, and April 2005). The other simulation optimization methods being employed and studied are ranking and selection, response surface methodology, gradient-based procedures, stochastic approximation, random search, and sample path optimization (Fu, Glover, and April 2005; Fu, Chen, and Shi 2008).

For the variable sample size, black box methods, the computational dominance of black box and scatter search methods is less well established. Here, we focus on hybrid technology that involves a population search method (like a genetic algorith) combined with selection and ranking methods. Therefore, this paper follows up on the essential insights in Boesel, Nelson, and Ishii (2003). However, the methods here have rigorous properties building on mathematical results in Zheng and Allen (2007).

Many commercial codes support optimization using simulation through either full factorial experimental designs or through deterministic heuristics applied to simulation optimization. In either case, these methods historically tended to use constant numbers of samples for each solution being evaluated at each iteration in the method. However, recently most of the main simulation software, such as Arena, AnyLogic, ProModel and its affiliated software, FlexSim, and SIMUL8 use OptQuest as the optimizer for their simulations (Simulation Software Survey; Arena Comparison Statement; Fu, Glover, and April 2005, OptQuest). OptQuest is a "general-purpose optimizer" which runs a complicated metaheuristic using a variable sample size scatter search and tabu search to find optimal solutions (Laguna 1997; Fu, Glover, and April 2005). The method of ranking and selection has become increasingly studied and applied in order to come to conclusions on the goodness of a solution (Fu, Chen, and Shi 2008).

It is likely that OptQuest offers greater computational efficiency than the methods here as well as better integration with existing software packages. However, we are not aware of any rigorous properties related to convergence or other associated with OptQuest. The intent here is to offer open-source code for a method with likely reasonable performance that is associated with rigorous convergence properties. Therefore, the proposed method can be viewed as a springboard for more efficient methods. Also, for cases in which integration with OptQuest or its cost are prohibited, the proposed methods may fill a useful gap.

Section 2 proposes the population indifference zone (PIZ) method. In Section 3, rigorous properties of the PIZ method are established building on results in Zheng and Allen (2007). Section 4 describes details related to implementing the code which can be downloaded from the web. Problems that can be used to compare the proposed method with alternatives are described in Section 5. Section 6 closes with a brief discussion and description of opportunities for future research.

2 THE POPULATION INDIFFERENCE ZONE (PIZ) METHOD

In this section, the population indifference zone (PIZ) method is proposed. The method includes three sub-phases: the formation of the solutions to be evaluate (e.g., creation of a new population in a genetic algorithm), the preliminary evaluation of of those solutions using subset selection, and the follow-up evaluation of solutions (if necessary) using an indifference zone procedure.

The specific problem that this paper is aimed at solving is: minimize $E(y(xi) + \varepsilon)$.

Subset selection (steps 2-5) is described in Goldsman, Nelson, Opicka, and Pritsker (1999). That procedures and those steps achieve a subset having a probability greater than P^* of containing a solution with mean or expected value within $\delta_{(1)}$ of the optimal or best system solution. The indifference parameter, $\delta_{(1)}$, can be set to zero, but generally fewer systems will be eliminated. Let k denote the number of alternative systems being compared. Assume we want to find a subset containing a system with mean within $\delta_{(1)}$ of the smallest, i.e., we are minimizing.

The indifference zone method is then used (steps 6-9) which is a two-stage selection from Sullivan and Wilson(1989). The sub-procedure starts with a set of k alternative systems and terminate with a subset of m, where the method user picks m. The user picks m together with the indifference parameter δ_2 and the lower bound on the quality probability P^*. For example, one might start with k = 100 systems and plan to end with 10 systems with one having a mean within $\delta_{(2)} = 3.0$ of the true best mean from the original 100 with probability $P^* = 0.95$. The procedure is based on pre-tabulated Rinott's constants, denoted here as $h = \text{Rinott}_{k,m,n0,P^*}$. Rinott (1978) was probably the first to tabulate these constants.

Population Indifference Zone (PIZ) Method

Step 1. (Initialization) Create k system alternatives by sampling uniformly from the decision-space.

Step 2. Evaluate all systems using n_0 samples, which are generally batch sample average values. Denote the resulting values $X_{i,j}$, with $i = 1,\ldots,k$ referring to the system and $j = 1,\ldots,n_0$ referring to the sample.

Step 3. Calculate the sample means or Monte Carlo estimates, $X_{\text{bar},i}$, for each system $i = 1,\ldots,k$. Denote the index for the system with the best mean as "b".

Step 4. Next, we examine the differences between the samples from each system paired with the samples from system b. Compute the standard deviations of the differences using:

$$S^2_{i,j} = \{\textstyle\sum_{j=1,\ldots,n0} [X_{i,j} - X_{b,j} - (X_{i,\text{bar}} - X_{b,\text{bar}})]^2\}/(n_0 - 1) \qquad (2.1)$$

and

$$W_{i,b} = (t_{\nu,n0-1})(S_{i,b})/[(n_0)^{1/2}] \tag{2.2}$$

with

$$\nu = 1 - (P^*)^{[1/(k-1)]} \tag{2.3}$$

and where "t" is a critical value for the t-distribution with $\alpha = \nu$ and n0 – 1 degrees of freedom .

Step 5. Form the subset by including only the systems with means satisfying:

$$(X_{i,\mathrm{bar}} - X_{b,\mathrm{bar}}) \leq \mathit{maximum}(W_{i,b} - \delta_{(1)}, 0.0) \tag{2.4}$$

where the "maximum" implies taking the larger value which could be 0.0. If the subset contains less than 10% of the generation skip to step 10.

Note that the above method with $\delta = 0$ is essentially creating single sample differences of each subsystem and the apparently best system. The approach essentially creates these difference and t-tests whether the differences are statistically significant. With $\delta = 0$, the previous steps are guaranteed to keep the best solution with probability greater than P^*.

Step 6. (First stage, optional because we can reuse the *Step 2* evaluations) Evaluate all systems with n_0 samples. These are generally batch means with typical initial values equal to $n_0 = 10$ or $n_0 = 20$. Fewer samples are generally needed if the batches are large. Calculate the sample means, $X_{i,\mathrm{bar}}$, and sample standard deviations, S_i, for all system responses.

Step 7. Calculate the total number of samples first stage plus follow-up for each system using:

$$n_i = \mathit{maximum}\{n_0 + 1, \mathit{roundup}[(h)^2 (S_i)^2 / (\delta_{(2)})^2]\}. \tag{2.5}$$

Step 8. (Second stage) Perform the additional n_i runs and then calculate the means of these second stage runs, $X_{\mathrm{bar},i}^{(2)}$. Denote the index for the system with the best mean as "b".

Step 9. Using the first stage standard deviations, S_i, calculate:

$$W_i = \frac{n}{n_i}\left[1 + \sqrt{1 - \frac{n_i}{n}\left(1 - \frac{(n_i - n)\delta_{(2)}^2}{h^2 S_i^2}\right)}\right] \quad \text{for } i = 1,\ldots,k \tag{2.6}$$

and keep m subsystems with among these with the smallest $W_i X_{i,\mathrm{bar}} + (1 - W_i) X_{i,bar}^{(2)}$ values.

The above procedure (Steps 6-9) is essentially the same as the Koenig and Law (1985) procedure.

Step 10. (Termination) Is any of our solutions good enough? If yes, stop, otherwise continue. Or if stopping rule has been reached, stop, otherwise continue.

Step 11. (Form the next population) Copy the 10% highest ranked solutions into the next population. If the subset selection eliminated 90% or more solutions, the ones left are the elitist subset. Otherwise, they are the subset from the indifference zone procedure. Fill in 80% of the total generation by selecting two systems at random, then for each variable in the system select a random number from 0 to 1. If the number is greater than .8 switch the level of the selected variable of the two systems. Repeat this for each variable. Fill the remaining 10%, or slightly more depending on the results of the subset selection, solutions with uniform random selections from the decision-space.

Willcox (1984) presents tables for Rinott's constants including those in Table 2-1. Use the condensed table below to select the appropriate Rinott's constant h. The is for use when population size k is equal to 10 and where l is the number of solutions in the subset. When we use 100 solutions in the population ad n_0 equal to 10 the bound on h is 3.3. We will use this as an approximation in this code because we do not currently have a proper lookup function to find all possible Rinott's constants.

Table 2-1. Rinott's constant values for population of size 10 where "l" is the # of solutions not eliminated in subset selection.

k	10
n_0	5
P^*	.95
l	h
2	3.107
3	3.908
4	4.390

5	4.746
6	5.025
7	5.260
8	5.463
9	5.641
10	5.799

The above method can be used very generally as long as solutions can be sampled uniformly from the solution space. It essentially draws from the huge space of solutions batches and then compares the batches efficiently using selection and ranking methods. In the next section, some of the rigorous properties of the PIZ method are established.

The presented PISA method is only a framework in the sense that virtually any method could be used to fill the population including tabu and scatter search methods. The only step that would be modified would be *Step 11* to the specifications of the method chosen to fill the population.

3 METHOD PROPERTIES

There are multiple desirable properties of the PIZ method. The first is that it is a black box method in the way that it is relatively simple and can be applied to virtually all simulation optimization problems as long as one can sample uniformly from the search space. The second property is that it offers some potential advantage in computational efficiency compared with constant sample size methods. This follows because poor quality solutions are usually removed from consideration in *Step 5*. This way, these solutions do not waste valuable CPU run time during the relatively exhaustive indifference zone-based evaluations in *Step 8*.

In addition, the PIZ method is associated with some rigorous claims about the solution quality as we next establish. This is not true for other methods of similar type including the genetic algorithm-based methods in Aizawa and Wah (1994) and Bernshteyn (2001). The makes it, perhaps a strong platform for additional method development.

The first rigorous claim is described in Theorem 1.

Theorem 1. Assume that there are N normal populations in the search space (N is generally very large). After M iterations, the PIZ method will retain at least one solution with objective value within ($M\times\delta$) of the best solution evaluated with probability greater than $(P^*)^{2M}$.

Proof. In the subset selection steps, we keep the best solution with probability greater than P^*. In the indifference zone steps we keep a solution within δ of the best with probability greater than P^*. Therefore, in each complete iteration we keep a solution within δ of the best with probability greater than $(P^*)^2$. This follows because of independence of the two positive events, i.e., keep the best and then keep a solution within δ of the best. The next iteration, there might an additional loss of δ. Therefore, after M iterations the loss is guaranteed to be less than ($M\times\delta$) with probability greater than $(P^*)^{2M}$.

Zheng and Allen (2007) describes the changing of both the δ and the P* in each population as the algorithm iterates. They do this to provided long run guarantees based on essentially Theorem 1. Examples of schedules offering long run guarantees are in equations (3.1), (3.2), (3.3), and (3.4). If at each iteration one has the probability of greater than P_t, to keep the solution with an objective value within δ_t of the real best solution in generation t, then:

$$P^* = \prod_{t=0}^{\infty} P^t, \text{ and} \tag{3.1}$$

$$\Delta = \sum_{t=0}^{\infty} \delta_t \tag{3.2}$$

Using the above equations one can calculate δ_t and P^t for each generation t using the final Δ and P^t chosen by the user. These equations are presented below where ϵ, o, u, and s are predetermined parameters where $0 < s < 1$, $o < u < 1$ and $0 < \epsilon < 1$.

$$\delta_t = \Delta(1-s)s^t \tag{3.3}$$

$$P^t = 1 - \frac{u}{(t+o)^{1+\epsilon}} \tag{3.4}$$

Applying a schedule to dt and pt can generate long run convergence to within d of global optimum with probability P*.

4 CODE DETAILS

The appendix contains the open-source code for simulation optimization.

4.1 Code explanation

The MS Excel-Visual Basic code uses a user-form to receive information from the user like number of variables, generation size, and total generations. The code itself breaks down into 6 main sections: setup, mapping, subset selection, indifference zone, cross-over, and output. In the setup basic computer programming techniques are applied: variables are named and their data type defined, arrays are dimensioned, and some variable values are set. The mapping section, which could possibly present the most challenging issue when trying to solve some problems, selects random variables from a uniform (0,1) distribution and maps them to the decision-space, which for the A4 function described in section 5.1 and used in this code is (-1.28, 1.28). The function used to do this is x =1.28 − 2.56[N(0,1)]. It is irrelevant that x will always be negative because $x^4$is always positive.

In general, the code uses many for loops and arrays to store the information needed. In the subset selection section the objective value of each iteration of each system is calculated to then find X_{bar}. Application.Worksheetfunction.Small is used to find the smallest objective value X_{bar}. Before the systems are selected the code finds the number that will be selected so that an array Copyover() can be dimensioned properly. Once the subset is selected it moves on to the indifference zone. First it checks to see if Copyover() has more than 10% of the original population in it. If it does it proceeds. The indifference zone is rather similar to the subset selection section, but with slightly different computations. The major difference is that in the indifference zone section some systems require more samples than others. To account for this we found the largest new sample size required. The array was then dimensioned with that n, and each system requires a reference to its new sample size.

The cross-over section begins by selecting the systems at random with uniform probabilities of being chosen by using Application.Worksheetfunction.Ceiling[GenerationSize * Rnd() 1]. This selects a system 1 through the total number in the generation to be one of the parents. We use four new arrays called FirstParent(), SecondParent(), FirstChild(), and SecondChild(). Once the parents are selected a random number (0,1) is picked. If the number is greater than .8 the value for the variable in FirstParent() is assigned to the value for the variable in SecondChild() and vice versa. If the number is less than .8 the value for the variable in FirstParent() is assigned to the value of the variable in FirstChild(). The code uses the basic Cells(R,C) command to output the value of the variables from the selected system.

4.2 Adapting the code

If one were interested in adapting the code to another function, all that would be necessary is to change the mapping section of the code. This is where the A4 function is located in the code. The comment in the code says, "Runs the A4 problem as a simulation." Filling the population could be changed as well, but would take more effort. All that needs to be done is change out the cross-over section and write code for tabu search or scatter search or your favorite search method.

5 NUMERICAL EXAMPLES

In this section, problems are described that can be used to evaluate the PIZ method and compare its computational performance with alternatives. These include the A4 function and the noisy Rastrigin function.

5.1 A4 function

Aizawa and Wah (1994) and Bernshteyn (2001) use a test function called the A4 function. This function is built into the code described in the appendix. This test function involves the random variable ε which is N(0,1) and can be written:

$$y(x) = \left[\sum_{i=1}^{k} ix_i^4\right] + A\epsilon \tag{5.1}$$

where typically k = 30 dimensions and -1.28 ≤ x ≤ 1.28. This equation can be studied for different values of A from 1 to 64, do 10 replicates of the whole procedure.

The PIZ method was tested with the A4 function. The results are shown in Table 5-1 and graphical representations are shown in Figures 5.1 – 5.5. The solution strength was measured by evaluating the A4 function without noise. Y-Bar is the average of four of these replicates. After seeing these results one can be confident in saying that the PIZ method is superior to constant sample size, especially for more variables (10) in a smaller population (10). It is also clear everything, but sigma had a major impact on the average objective value; however, if sigma were varied more than just 1 there would probably have been a larger impact. We can say this just from experience while debugging the code. This would be another issue to address in the future. The total number of simulations run had a smaller effect than the other factors.

Table 5-1. Results of PIZ Method Testing on A4 Function.

A	Num of Var	n0	Generation Size	# of Function evals.	Y_Bar	S
2	3	5	10	1000	0.1789	0.3024
2	3	5	10	10000	0.6371	0.4811
2	10	5	10	1000	5.3653	3.6542
2	10	5	10	10000	3.3782	1.8258
2	3	10	100	1000	0.0094	0.0071
2	3	10	100	10000	0.0141	0.0154
2	10	10	100	1000	4.2584	1.0010
2	10	10	100	10000	1.4863	1.0503
1	3	5	10	1000	0.1424	0.1677
1	3	5	10	10000	0.1919	0.1978
1	10	5	10	1000	6.3125	1.5192
1	10	5	10	10000	1.5610	0.7929
1	3	10	100	1000	0.0171	0.0181
1	3	10	100	10000	0.0054	0.0055
1	10	10	100	1000	3.8757	1.2707
1	10	10	100	10000	1.4220	0.6787
2	3	20	10	1000	1.2510	1.1176
2	3	20	10	10000	0.1972	0.2159
2	10	20	10	1000	16.0049	8.0635
2	10	20	10	10000	14.6057	4.2168
2	3	20	100	1000	0.0108	0.0097
2	3	20	100	10000	0.0245	0.0153
2	10	20	100	1000	3.5343	2.4640
2	10	20	100	10000	3.1535	0.8398
1	3	20	10	1000	0.5005	0.5932
1	3	20	10	10000	0.8665	0.9407
1	10	20	10	1000	17.2263	6.2762
1	10	20	10	10000	13.3052	6.0228
1	3	20	100	1000	0.0476	0.0396
1	3	20	100	10000	0.0201	0.0352
1	10	20	100	1000	5.0307	2.4679
1	10	20	100	10000	3.4815	1.8323

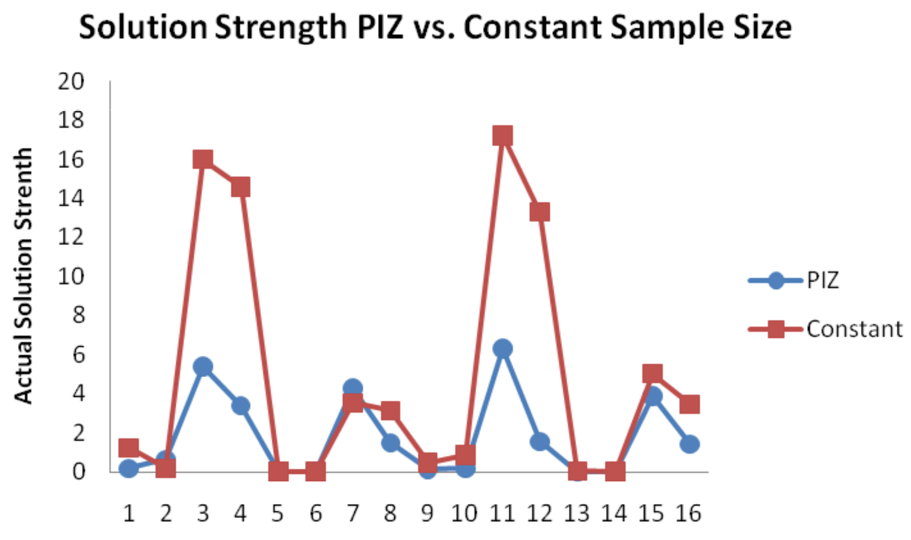

Figure 5.1. A graph of solution strength comparing the PIZ method and constant sample size.

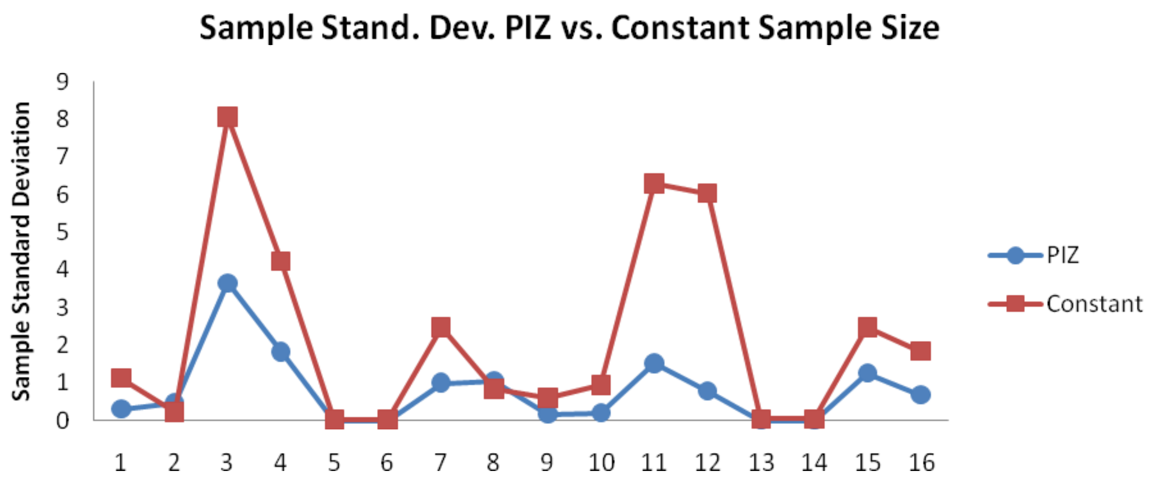

Figure 5.2. A graph of sample standard deviation comparing the PIZ method and constant sample size.

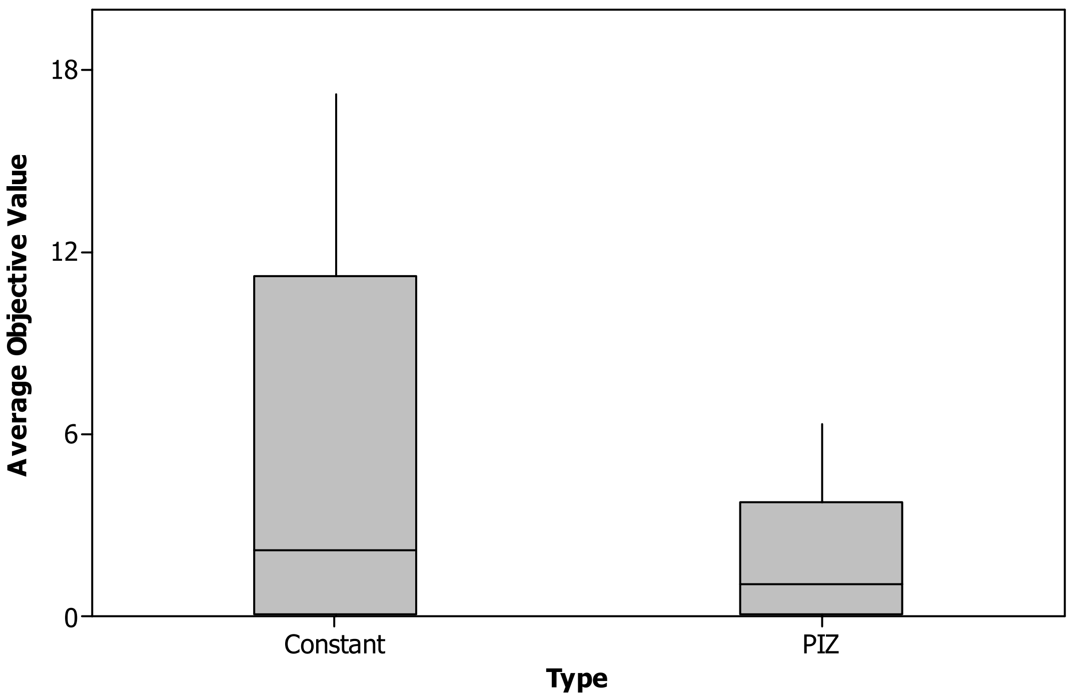

Figure 5.3. Box plot of the average objective value (solution strength) of constant sample size and PIZ.

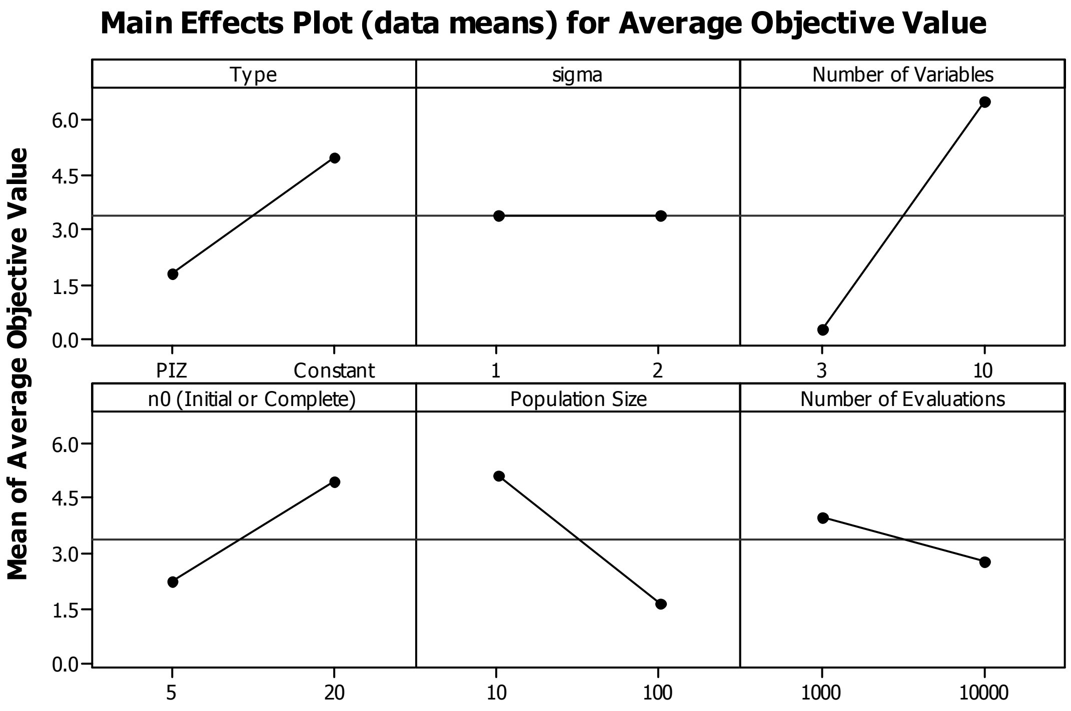

Figure 5.4. Main effects plot for the average objective value (solution strength).

5.2 Rastrigin function

Aizawa and Wah (1994) present another test function. The Rastrigin function involves the same random variable ε as described above, but now $-5.12 \leq x \leq 5.12$. This function will soon be used to test the robustness of the PIZ method.

$$y(x) = 20 + \left[\sum_{i=1}^{K} x_i^2 - \cos(2\pi x_i)\right] + A\epsilon \tag{5.2}$$

6 DISCUSSION

This thesis has proposed an algorithm for simulation optimization called the population indifference zone (PIZ) method. It is called an algorithm because it is associated with rigorous convergence guarantees as described in section 3. Perhaps more importantly, it is available with open-source in a Visual Basic implementation. There are, however, a number of opportunities for future research.

First, the PIZ method can be compared computationally with constant sample size and other alternatives using the functions in Section 5. Such results can establish the benefits of using the method in practice. As for now it is clear that that PIZ method is superior to the constant sample size method. The robustness of the PIZ method is still in question and will soon be tested with the Rastrigin function. Second, a download site for the code can be established to facilitate others benefiting from the code. Third, an example of using SHELL of other visual basic commands to control an actual simulation code executable can be developed. This will help engineers at major companies benefit from simulation optimization even in cases in which connection with OptQuest is difficult or impossible.

Finally, the application of PIZ to a real problem could illustrate the method and its practical benefits. This could be done in an integrated method together with other visual basic capabilities at a major manufacturer.

ACKNOWLEDGEMENTS

I would like to thank Theodore T. Allen who provided much of the work and is my thesis advisor, for his passion and continued guidance from beginning to end of this project. This work was partially supported by a grant from Honda of America. John Gillard and Timothy Leopold provided insights into what matters in real-world applications. Also, I would like to thank Cathy Xia for agreeing to be on the committee. I would also like to thank my parents for their unending love, care and support along with my ultimate teammates and friends.

APPENDIX

```
    Option Explicit
    Option Base 1
________________________________________________________________________
Private Sub DisplayMatrix_Click()
    DisplayMatrix = True
End Sub
________________________________________________________________________


Private Sub GeneticAlgorithm_Click()                    ' Runs Program When Userform is clicked

Application.ScreenUpdating = False                     """""""""Turns off screen updating

Dim i As Integer, j As Integer, k As Integer, m As Integer, n As Integer, q As Integer, p As Integer, z As Integer
Dim NumOfVariables As Long, TwoTimesNumOfVariables As Long, ThreeTimesNumOfVariables As Long
Dim arrMyArray() As Double, ObjectiveArray() As Double, NewBigArray() As Double, OptimumArray() As Double,
SwitchingValueArray() As Double
Dim GenerationSize As Double, Generation As Long, SwitchingValue As Double
Dim TenthLarge As Double, TenthSmall As Double, TenthLargest As Double, TenthSmallest As Double, Small As
Double
Dim FirstParentRand As Double, SecondParentRand As Double, FirstParent() As Double, SecondParent() As Double,
Child() As Double, SecondChild() As Double
Dim TotalGenerations As Long, SimulationRunCounter As Long
Dim Percent As Double, Indif As Integer, SolutionQuality() As Double
Dim new_n() As Long, n0 As Integer, t As Long, b As Long, delta2 As Double, delta1 As Double
Dim XBar As Double, StandardDeviations2 As Double, StandardDeviations As Double
Dim SubsetWeight As Double, Nu As Double, TValue As Double, BestSubsetWeight As Double
Dim CopyOver() As Double, w As Long, s As Long, largestn As Long, u As Long
Dim IndifSubsetWeight() As Double, CopyOver2() As Double, ConstantCopyOver() As Double
Dim IndifSubsetWeightDiff() As Double, CopyOverCounter() As Boolean
Dim IndifSubsetWeight1() As Double, r As Long, ExitLoop As Double, ExitLoop2 As Double
Dim CopyOverCounter2() As Boolean, OptimumSetingsArray() As Double, NewOptimumSetingsArray() As Double
Dim Rinott As Double, epsilon As Double, Best As Long, FinalVariablesArray() As Double



NumOfVariables = CDbl(NumberOfVariables)                         'Reads in Variables from userform
TwoTimesNumOfVariables = NumOfVariables * 2
ThreeTimesNumOfVariables = NumOfVariables * 3
GenerationSize = CDbl(GenSize)
TotalGenerations = CDbl(TotGen)
Percent = CDbl(Perc)
Randomize

ReDim arrMyArray(GenerationSize, TwoTimesNumOfVariables + 1)              'Dimensions Arrays with information
from userform
ReDim ObjectiveArray(GenerationSize)
ReDim NewBigArray(GenerationSize, TwoTimesNumOfVariables + 1)
ReDim OptimumArray(TotalGenerations)
ReDim SwitchingValueArray(NumOfVariables)
ReDim FirstParent(NumOfVariables)
ReDim SecondParent(NumOfVariables)
ReDim FirstChild(TwoTimesNumOfVariables + 1)
ReDim SecondChild(TwoTimesNumOfVariables + 1)
```

```
    ReDim SolutionQuality(GenerationSize)
    ReDim arrMyArray(GenerationSize, (NumOfVariables * (n0 + 1) + n0 + 3))
    ReDim new_n(GenerationSize)
    ReDim NewBigArray(GenerationSize, (NumOfVariables * (n0 + 1) + n0 + 3))
    ReDim CopyOverCounter(GenerationSize)
    ReDim CopyOverCounter2(GenerationSize)
    ReDim OptimumSettingsArray(TotalGenerations, NumOfVariables)
    ReDim FinalVariablesArray(GenerationSize, NumOfVariables)
    ReDim ConstantCopyOver(GenerationSize, NumOfVariables)

    SimulationRunCounter = 0                    ' Counts the total number of simulation that were run would that would be
run
    'DisplayMatrix = True                   """""" Sets value as true to display for when debugging
    UserForm1.Hide                         'closes the userform
    If RunWithErrors = False Then


    Else


    'DisplayMatrix = True                   """""" Sets value as true to display for when debugging

    n0 = 10                                     ' Sets the values of other variables
    Rinott = 3.3                                 ' May add to userform
    delta1 = 0
    delta2 = 1                                  ' Need function or table for rinott's constant
    epsilon = 2

    Indif = 22
    Generation = 1

    Do Until Generation > TotalGenerations 'Or SimulationRunCounter > 100000                    ' Will run program un-
til the total number of generations is completed
      ReDim ObjectiveArray(GenerationSize)

      For k = 1 To GenerationSize                          ' Fills in the entire first generation with uniform random num-
bers
          If Generation = 1 Then                         ' from the sample space. In this case (0,1)
            For i = 1 To NumOfVariables
              arrMyArray(k, i) = Rnd()
              If DisplayMatrix = True Then
                Cells(k, i) = arrMyArray(k, i)           """" Will display the array if necessary
              End If
            Next i
          Else
            If k > GenerationSize - Percent * GenerationSize Or (k > z And k <= Percent * GenerationSize) Then
              For i = 1 To NumOfVariables                     ''' Same as above but only for the mutants of the GA
                arrMyArray(k, i) = Rnd()
                If DisplayMatrix = True Then
                  Cells(k, i) = arrMyArray(k, i)
                End If
              Next i
            End If
          End If
    ''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
    '''''''''''''''''''''''''''''''''''''''''''''''''''''Begin Subset Selection''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
    ''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
        XBar = 0
        For t = 1 To n0
          SimulationRunCounter = SimulationRunCounter + 1            '''' counts the number of simulation runs over the
whole algorithm
          If SimulationRunCounter > 1000 Then
            For ExitLoop = 1 To GenerationSize
              For ExitLoop2 = 1 To NumOfVariables
                FinalVariablesArray(ExitLoop, ExitLoop2) = arrMyArray(ExitLoop, ExitLoop2)
              Next ExitLoop2
            Next ExitLoop
            Exit Do
          End If

          For i = NumOfVariables + 1 To TwoTimesNumOfVariables          '''' Runs the A4 problem as a simulation
            arrMyArray(k, i + (t - 1) * NumOfVariables) = ((i - NumOfVariables) * ((1.28 - 2.56 * arrMyArray(k, i - Nu-
mOfVariables)) ^ 4))

            If DisplayMatrix = True Then
              Cells(k, i + (t - 1) * NumOfVariables) = arrMyArray(k, i + (t - 1) * NumOfVariables)         '''' Will display
the array if necessary
            End If
          Next i

          arrMyArray(k, ((n0 + 1) * NumOfVariables) + t) = 0
          For j = 1 To NumOfVariables                                  ' calculates the objective vlue of each
            arrMyArray(k, ((n0 + 1) * NumOfVariables) + t) = arrMyArray(k, ((n0 + 1) * _
            NumOfVariables) + t) + arrMyArray(k, j + (t * NumOfVariables))              ' and begins calculating
X-bar for each solution
          Next j

          arrMyArray(k, ((n0 + 1) * NumOfVariables) + t) = arrMyArray(k, ((n0 + 1) * NumOfVariables) + t) + epsilon *
WorksheetFunction.NormInv(Rnd(), 0, 1)
          XBar = XBar + arrMyArray(k, ((n0 + 1) * NumOfVariables) + t)

          If DisplayMatrix = True Then
            Cells(k, ((n0 + 1) * NumOfVariables) + t) = arrMyArray(k, ((n0 + 1) * NumOfVariables) + t)          '''' Will
display the array if necessary
          End If
        Next t

        XBar = XBar / n0                                               'calculates true value of X-bar
        arrMyArray(k, (NumOfVariables * (n0 + 1) + n0 + 1)) = XBar

        If DisplayMatrix = True Then
          Cells(k, (NumOfVariables * (n0 + 1) + n0 + 1)) = arrMyArray(k, (NumOfVariables * (n0 + 1) + n0 + 1))
        End If

      ObjectiveArray(k) = arrMyArray(k, (NumOfVariables * (n0 + 1) + n0 + 1))                    ''' Puts the objective
values (x-bar) in an array
      Next k

      TenthLarge = Application.WorksheetFunction.Large(ObjectiveArray, Percent * GenerationSize)             '''' Finds
the tenth largest objective value
```

```
        TenthSmall = Application.WorksheetFunction.Small(ObjectiveArray, Percent * GenerationSize)              "" Finds
the tenth smallest objective value
        Small = Application.WorksheetFunction.Small(ObjectiveArray, 1)                                 "" Finds lowest objec-
tive value

        For k = 1 To GenerationSize
          If arrMyArray(k, (NumOfVariables * (n0 + 1) + n0 + 1)) = Small Then                  "' finds the solution with the
losest objective value
            Best = k
            k = k + GenerationSize
          End If
        Next k



        StandardDeviations2 = 0

        For k = 1 To GenerationSize
          For t = 1 To n0                                               "" calculates the standard deviation of each system
            StandardDeviations2 = StandardDeviations2 + _
            (arrMyArray(k, NumOfVariables * (n0 + 1) + t) - arrMyArray(Best, NumOfVariables * (n0 + 1) + t) - _
            (arrMyArray(k, NumOfVariables * (n0 + 1) + n0 + 1) - arrMyArray(Best, NumOfVariables * (n0 + 1) + n0 + 1)))
^ 2
          Next t
          StandardDeviations2 = StandardDeviations2 / (n0 - 1)
          StandardDeviations = Sqr(StandardDeviations2)
          arrMyArray(k, NumOfVariables * (n0 + 1) + n0 + 2) = StandardDeviations
          If DisplayMatrix = True Then
            Cells(k, NumOfVariables * (n0 + 1) + n0 + 2) = StandardDeviations
          End If
        Next k
        w = 1
        If PIZButton = True Then
          For k = 1 To GenerationSize
            If ObjectiveArray(k) <= TenthSmall Then
              For i = 1 To NumOfVariables
                ConstantCopyOver(w, i) = arrMyArray(k, i)
              Next i
              w = w + 1
            End If
          Next k
        Else
    """""""Weighting""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
        Nu = 1 - (0.95 ^ (1 / (GenerationSize - 1)))                               """" Calculated nu for finding t-value
        TValue = Application.WorksheetFunction.TInv(Nu, n0 - 1)                           """" Finds t-value for confidence in-
terval
        BestSubsetWeight = (TValue) * (arrMyArray(Best, NumOfVariables * (n0 + 1) + n0 + 2)) / (n0 ^ 0.5)  " Finds the
weight of the best solution
        w = 1

        For k = 1 To GenerationSize
          SubsetWeight = (TValue) * (arrMyArray(k, NumOfVariables * (n0 + 1) + n0 + 2)) / (n0 ^ 0.5)
          arrMyArray(k, NumOfVariables * (n0 + 1) + n0 + 3) = SubsetWeight

          CopyOverCounter(k) = False                                         "' Finds the systems to copy over
        If arrMyArray(k, NumOfVariables * (n0 + 1) + n0 + 1) - arrMyArray(Best, NumOfVariables * (n0 + 1) + n0 + 1)
<= _
            Application.WorksheetFunction.Max(BestSubsetWeight - delta1, 0) Then
            CopyOverCounter(k) = True
            For s = 1 To NumOfVariables * (n0 + 1) + n0 + 3
                If DisplayMatrix = True Then
                    Cells(GenerationSize + 2 + w, s) = arrMyArray(k, s)
                End If
            Next s
            w = w + 1
        End If
    Next k

    z = w - 1
    w = 1                                                   ""Copies over the systems
    ReDim CopyOver(z, NumOfVariables * (n0 + 1) + n0 + 3)
    ReDim IndifSubsetWeight(z)

    For k = 1 To GenerationSize
        If CopyOverCounter(k) = True Then
            For s = 1 To NumOfVariables * (n0 + 1) + n0 + 3
                CopyOver(w, s) = arrMyArray(k, s)
            Next s
            w = w + 1
        End If
    Next k

    """""""""""""""""""""""""""Indeference Zone """"""""""""""""""""""""""""""""""""""""""
    """"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""


    If z <= Percent * GenerationSize Then
        Small = Application.WorksheetFunction.Small(ObjectiveArray, 1)          ' If the number copied over is less than
10% skip to crossovers
        OptimumArray(Generation) = Small
    End If

    If z > Percent * GenerationSize Then                        ' If the number is greater than 10% do Indeference Zone
        If GenerationSize = 10 Then
            If z = 2 Then Rinott = 3.107
            If z = 3 Then Rinott = 3.908
            If z = 4 Then Rinott = 4.39
            If z = 5 Then Rinott = 4.746
            If z = 6 Then Rinott = 5.025
            If z = 7 Then Rinott = 5.26
            If z = 8 Then Rinott = 5.463
            If z = 9 Then Rinott = 5.641
            If z = 10 Then Rinott = 5.799
        End If
        For k = 1 To z                                     ' Calculate the number of new samples per systems which =
new_n(k)-n0
            new_n(k) = Application.WorksheetFunction.Max((n0 + 1), _
                (Application.WorksheetFunction.RoundUp(Rinott ^ 2 * (CopyOver(k, NumOfVariables * (n0 + 1) + n0 + 2) ^
2 / delta2), 1)))
```

```
        Next k

        largestn = Application.WorksheetFunction.Large(new_n, 1)                ' Find the largest new n for dimensioning of
array
        ReDim Preserve CopyOver(z, NumOfVariables * (n0 + 1) + n0 + 3 + NumOfVariables * (largestn - n0) + (largestn
- n0) + 3)
        ReDim ObjectiveArray(z)

        For k = 1 To z                                        ' Runs simulations for each system selected for the new_n(k)-
n0 amount of times
            XBar = CopyOver(k, NumOfVariables * (n0 + 1) + n0 + 1) * n0
          For t = 1 To (new_n(k) - n0)
            SimulationRunCounter = SimulationRunCounter + 1                ' Adds to the total count of simulations run
            If SimulationRunCounter > 1000 Then
              For ExitLoop = 1 To GenerationSize
                For ExitLoop2 = 1 To NumOfVariables
                  FinalVariablesArray(ExitLoop, ExitLoop2) = arrMyArray(ExitLoop, ExitLoop2)
                Next ExitLoop2
              Next ExitLoop
                Exit Do
              End If
            For i = 1 To NumOfVariables

              CopyOver(k, i + (n0 + 1) * NumOfVariables + n0 + 3 + (t - 1) * NumOfVariables) = _
              (i * ((1.28 - 2.56 * CopyOver(k, i)) ^ 4))
              If DisplayMatrix = True Then
                Cells(GenerationSize + 2 + k, i + (n0 + 1) * NumOfVariables + n0 + 3 + (t - 1) * NumOfVariables) = _
                CopyOver(k, i + (n0 + 1) * NumOfVariables + n0 + 3 + (t - 1) * NumOfVariables)          "" Will display
the array if necessary
              End If
            Next i

            CopyOver(k, ((n0 + 1) * NumOfVariables) + n0 + 3 + ((new_n(k) - n0) * NumOfVariables) + t) = 0
            j = 1                                                   ' Begins calculating X-Bar
            For j = 1 To NumOfVariables
              CopyOver(k, ((n0 + 1) * NumOfVariables) + n0 + 3 + ((new_n(k) - n0) * NumOfVariables) + t) = _
                CopyOver(k, ((n0 + 1) * NumOfVariables) + n0 + 3 + ((new_n(k) - n0) * NumOfVariables) + t) + _
                 CopyOver(k, ((n0 + 1) * NumOfVariables) + n0 + 3 + j + (t - 1) * NumOfVariables)
            Next j
            CopyOver(k, ((n0 + 1) * NumOfVariables) + n0 + 3 + ((new_n(k) - n0) * NumOfVariables) + t) = _
              CopyOver(k, ((n0 + 1) * NumOfVariables) + n0 + 3 + ((new_n(k) - n0) * NumOfVariables) + t) + _
                epsilon * WorksheetFunction.NormInv(Rnd(), 0, 1)
            XBar = XBar + CopyOver(k, ((n0 + 1) * NumOfVariables) + n0 + 3 + ((new_n(k) - n0) * NumOfVariables) +
t)

            If DisplayMatrix = True Then
              Cells(GenerationSize + 2 + k, ((n0 + 1) * NumOfVariables) + n0 + 3 + ((new_n(k) - n0) * NumOfVariables)
+ t) = _
                CopyOver(k, ((n0 + 1) * NumOfVariables) + n0 + 3 + ((new_n(k) - n0) * NumOfVariables) + t)    "" Will
display the array if necessary
            End If
          Next t

          XBar = XBar / new_n(k)                                              ' Calculates final X-bar for each
system
```

```
          CopyOver(k, (NumOfVariables * (n0 + 1)) + n0 + 3 + ((new_n(k) - n0) * NumOfVariables) + (new_n(k) - n0) +
1) = XBar
          ObjectiveArray(k) = XBar
          If DisplayMatrix = True Then
            Cells(k + 2 + GenerationSize, (NumOfVariables * (n0 + 1)) + n0 + 3 + ((new_n(k) - n0) * NumOfVariables) +
(new_n(k) - n0) + 1) = XBar
          End If
        Next k

        Small = Application.WorksheetFunction.Small(ObjectiveArray, 1)                              ' Finds system
with smallest objective function
        OptimumArray(Generation) = Small

        For k = 1 To z
          If CopyOver(k, (NumOfVariables * (n0 + 1)) + n0 + 3 + ((new_n(k) - n0) * NumOfVariables) + (new_n(k) - n0)
+ 1) = Small Then
            Best = k                                                   'Finds system with lowest objective func-
tion
            k = k + GenerationSize
          End If
        Next k


        '''''''''''IZ Weighting''''''''''''''

        For k = 1 To z
          IndifSubsetWeight(k) = (n0 / new_n(k)) * (1 + Sqr(1 - ((n0 / new_n(k)) * (1 - ((new_n(k) - n0) * delta2 ^ 2) / _
            (Rinott ^ 2 * CopyOver(k, NumOfVariables * (n0 + 1) + n0 + 2) ^ 2)))))                ' calculates weight
            CopyOver(k, (NumOfVariables * (n0 + 1)) + n0 + 3 + ((new_n(k) - n0) * NumOfVariables) + (new_n(k) - n0)
+ 2) = IndifSubsetWeight(k)
        Next k

        ReDim IndifSubsetWeightDiff(z)
        For k = 1 To z                                                 ' calculates difference in weights
          CopyOver(k, (NumOfVariables * (n0 + 1)) + n0 + 3 + ((new_n(k) - n0) * NumOfVariables) + (new_n(k) - n0) +
3) = _
            CopyOver(k, NumOfVariables * (n0 + 1) + n0 + 3) * CopyOver(Best, (NumOfVariables * (n0 + 1)) + n0 + 3 +
((new_n(Best) - n0) * NumOfVariables) + (new_n(Best) - n0) + 1) + _
            (1 - IndifSubsetWeight(k)) * CopyOver(Best, ((NumOfVariables * (n0 + 1)) + n0 + 3 + ((new_n(Best) - n0) *
NumOfVariables) + (new_n(Best) - n0) + 1))
          IndifSubsetWeightDiff(k) = CopyOver(k, (NumOfVariables * (n0 + 1)) + n0 + 3 + ((new_n(k) - n0) * Nu-
mOfVariables) + (new_n(k) - n0) + 3)
        Next k

        TenthSmall = Application.WorksheetFunction.Small(IndifSubsetWeightDiff, Percent * GenerationSize)          '
finds tenth smallest weight
        u = 1

        For k = 1 To z
          CopyOverCounter2(k) = False
          If IndifSubsetWeightDiff(k) <= TenthSmall Then                                     ' selects ten with lowest
weight
            CopyOverCounter2(k) = True
            For s = 1 To (NumOfVariables * (n0 + 1)) + n0 + 3 + ((new_n(k) - n0) * NumOfVariables) + (new_n(k) - n0)
+ 3
```

```
                If DisplayMatrix = True Then
                    Cells(GenerationSize + 3 + w + u, s) = CopyOver(k, s)
                End If
            Next s
            u = u + 1
        End If
    Next k

    r = u - 1
    ReDim CopyOver2(r, (NumOfVariables * (n0 + 1)) + n0 + 3 + ((largestn - n0) * NumOfVariables) + (largestn - n0)
+ 3)
    u = 1

    For k = 1 To z                                                          ' carries over 10 with losest weights
        If IndifSubsetWeightDiff(k) <= TenthSmall Then
            For s = 1 To (NumOfVariables * (n0 + 1)) + n0 + 3 + ((new_n(k) - n0) * NumOfVariables) + (new_n(k) - n0)
+ 3
                CopyOver2(u, s) = CopyOver(k, s)
            Next s
            u = u + 1
        End If
    Next k

    If DisplayMatrix = True Then
        For k = 1 To GenerationSize + 3 + w + u                                         ' Erases all cells
            For m = 1 To (NumOfVariables * (n0 + 1)) + n0 + 3 + ((largestn - n0) * NumOfVariables) + (largestn - n0) + 3
                Cells(k, m) = ""
            Next m
        Next k
    End If

    For k = 1 To r
        For m = 1 To NumOfVariables                                                 ' Stores the selected system in new
array
            NewBigArray(k, m) = CopyOver2(k, m)
            If DisplayMatrix = True Then
                Cells(k, m) = NewBigArray(k, m)
            End If
        Next m
    Next k
    Indif = 1
Else
    If DisplayMatrix = True Then                                                ' Erases all cells
        For k = 1 To GenerationSize + 3 + w + u
            For m = 1 To (NumOfVariables * (n0 + 1)) + n0 + 3
                Cells(k, m) = ""
            Next m
        Next k
    End If



    For k = 1 To z
        For m = 1 To NumOfVariables                                                 ' Stores the selected system in new
array
```

```
                NewBigArray(k, m) = CopyOver(k, m)
                If DisplayMatrix = True Then
                    Cells(k, m) = CopyOver(k, m)
                End If
            Next m
        Next k
    End If


    End If
    '''''''''''''''''''''''''''''''''''''''''''''''''' cross overs''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''

    For k = 1 To ((1 - (2 * Percent)) * GenerationSize) / 2
        FirstParentRand = Application.WorksheetFunction.Ceiling(GenerationSize * Rnd(), 1)                    ' selects random
system
        SecondParentRand = Application.WorksheetFunction.Ceiling(GenerationSize * Rnd(), 1)                    ' selects ran-
dom system
        For p = 1 To NumOfVariables
            SwitchingValue = Rnd()                                                     ' generates the switching value
            FirstParent(p) = arrMyArray(FirstParentRand, p)                                     ' assigns appropriate setting to
each variable
            SecondParent(p) = arrMyArray(SecondParentRand, p)
            If SwitchingValue > 0.8 Then                                              ' will switch the variable setting if
value is > .8
                FirstChild(p) = SecondParent(p)
                SecondChild(p) = FirstParent(p)
            Else
                FirstChild(p) = FirstParent(p)                                         ' if not keeps them the same
                SecondChild(p) = SecondParent(p)
            End If
        Next p
        For p = 1 To NumOfVariables                                                  ' copies new systems (children) into
new array
            NewBigArray(k + (Percent * GenerationSize), p) = FirstChild(p)
            NewBigArray(k + (Percent * GenerationSize) + ((1 - (2 * Percent)) * GenerationSize) / 2, p) = SecondChild(p)
            If DisplayMatrix = True Then
                Cells(k + (Percent * GenerationSize), p) = FirstChild(p)
                Cells(k + (Percent * GenerationSize) + ((1 - (2 * Percent)) * GenerationSize) / 2, p) = SecondChild(p)
            End If
        Next p
    Next k

    For k = 1 To GenerationSize - GenerationSize * Percent                               ' Copies over new array into
array used from beginng
        If Indif = 1 Then                                                      ' If it did not go through indiference zone
            If k <= r Then
                For m = 1 To NumOfVariables
                    arrMyArray(k, m) = CopyOver2(k, m)
                Next m
            Else
                For m = 1 To NumOfVariables
                    arrMyArray(k, m) = NewBigArray(k, m)
                Next m
            End If
        ElseIf PIZButton = True Then
```

```
            If k <= Percent * GenerationSize Then
                For m = 1 To NumOfVariables
                    arrMyArray(k, m) = ConstantCopyOver(k, m)
                Next m
            Else
                For m = 1 To NumOfVariables
                    arrMyArray(k + (Percent * GenerationSize), m) = NewBigArray(k + (Percent * GenerationSize), m)
                Next m
            End If
        Else
            If k <= z Then                                                                                                      '
                For m = 1 To NumOfVariables
                    arrMyArray(k, m) = CopyOver(k, m)
                Next m
            Else
                For m = 1 To NumOfVariables
                    arrMyArray(k + (Percent * GenerationSize), m) = NewBigArray(k + (Percent * GenerationSize), m)
                Next m
            End If
        End If
    Next k
    Generation = Generation + 1
Loop

''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''out put''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
k = 1
i = 1
j = 1

For k = 1 To GenerationSize

    For i = 1 To NumOfVariables
        Cells(k, i) = arrMyArray(k, i)          '''' Will display the array if necessary
    Next i

    Cells(k, i) = arrMyArray(k, (NumOfVariables * (n0 + 1) + n0 + 1))          '''' Will display the array if necessary

Next k
For Generation = 1 To TotalGenerations                                               ''' Generates out put
    Cells(Generation + GenerationSize + 2, 1) = OptimumArray(Generation)
Next Generation

End If

'For k = 1 To z
'    If ObjectiveArray(k) = Small Then
'            'Cells(2 * GenerationSize + 6 + k, 2) = ObjectiveArray(k)                        ' generates output
'            Best = k
'    End If
'Next k



    For k = 1 To GenerationSize
        SolutionQuality(k) = 0
        For i = 1 To NumOfVariables
            SolutionQuality(k) = SolutionQuality(k) + (i * ((1.28 - 2.56 * FinalVariablesArray(k, i)) ^ 4))
            Cells(k, i) = FinalVariablesArray(k, i)
'       Cells(1, i + 1) = arrMyArray(Best, i)           ' generates output
        Next i
    Next k
    Small = Application.WorksheetFunction.Small(SolutionQuality(), 1)
    For k = 1 To GenerationSize
        If SolutionQuality(k) = Small Then
            Best = k
        End If
    Next k
    Cells(1, i + 1) = SolutionQuality(Best)

        For i = 1 To NumOfVariables
            Cells(2, NumOfVariables + 2 + i) = FinalVariablesArray(Best, i)
        Next i
    Cells(1, i + 2) = SimulationRunCounter - 1

Application.ScreenUpdating = True
End Sub
```

REFERENCES

Aizawa, A.N. and Wah, B.W. (1994). “Scheduling of Genetic Algorithms in a Noisy Environment”. *Evolutionary Computation*, 2(2), 97-122.

“Arena Comparison Statement,” Rockwell Automation, 12 Feb. 2009, <http://www.arenasimulation.com/news/docs/Arena%20comparison%20statement%202007.pdf>.

Bernshteyn, Mikhail (2001), “Simulation Optimization Methods That Combine Multiple Comparisons and Genetic Algorithms with Applications in Design for Computer and Supersaturated Experiments,” The Ohio State University, Industrial, Welding & Systems Engineering.

Boesel, J., B. L. Nelson, and N. Ishii (2003), “A Framework for Simulation-Optimization Software,” IIE Transactions, 35:3, 221-229.

Fu, M. C., C. Chun-Hung, and L. Shi (2008), “Some Topics for Simulation Optimization,” *Proceedings of the 2008 Winter Simulation Conference*, S. J. Mason, R. R. Hill, L. Mönch, O. Rose, T. Jefferson, J. W. Fowler, eds.

Fu, M. C., F. W. Glover, and J. April (2005), “Simulation optimization: a review, new developments, and applications,” *Proceedings of the 2005 Winter Simulation Conference*, M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, eds.

Goldsman D., B. L. Nelson, T. Opicka, and A. A. B. Pritsker (1999), “A Ranking and selection Project: Experiences from A University-Industry Collaboration,” In *Proceedings of the 1999 Winter Simulation Conference*, P. A. Farrington, H. B. Nembhard, D. T. Sturrock, and G. W. Evans, eds. 83-92. Piscataway, New Jersey: Institute of Electrical Engineers.

Koenig, L. W. and Law, A. M. (1985), “A procedure for Selecting a Subset of Size m Containing the I Best of k Independent Normal Populations, with Applications to Simulation,” *Communications in Statistics: Simulation and Computation*, 14, 719-734.

Laguna, Manuel (1997), “Optimization of Complex Systems with OptQuest,” University of Colorado, Graduated School of Business Administration.

“OptQuest,” ProModel, 12 Feb. 2009, <http://www.promodel.com/products/optquest/>.

Rinott, Y. (1978), “On Two-stage Selection Procedures and Related Probability Inequalities,” *Communications in Statistics*, 7: 799-811.

“Simulation Software Survey,” *ORMS Today*, Page 4, October 2007.

Sullivan, D. A. and Wilson, J. R. (1989), “Restricted Subset Selection Procedures for Simulation,” *Operations Research*, 37, 1, 52- 71.

Wilcox, Rand R. Apr 1984. “A Table for Rinott’s Selection Procedure.” *Journal of Quality Technology*, 16, 97-100.

Zhao, L. and S. Sen (2006), “A Comparison of Sample-Path Based Simulation-Optimization and Stochastic Decomposition For Multi-Location Transshipment Problems,” *Proceedings of the 2006 Winter Simulation Conference*, L. F. Perrone, F. P. Wieland, J. Liu, B. G. Lawson, D. M. Nicol, and R. M. Fujimoto, eds.

Zheng, N., and T. T. Allen (2007), “Subset Selection and Optimization for Selecting Binomial Systems Applied to Supersaturated Design Generation,” *Proceedings of the 2007 Winter Simulation Conference*, S. G. Henderson, B. Biller, M. – H. Hsieh, J. Shortle, J. D. Tew, and R. R. Barton, eds.