

5R1: Stochastic Optimization Coursework

Vincent Y.F. Tan
Sidney Sussex College

March 12, 2005

1 Introduction

In this coursework exercise, the Biased Monte Carlo Sampling (BMCS) Algorithm [1] and the Genetic Algorithm (GA) [2] were successfully implemented. Their respective performances on the Keane's Bump Problem were thoroughly investigated. In particular, the effect of different parameters on the algorithms were analyzed carefully.

1.1 The problem

The 2-dimensional problem is stated as:

$$\max \quad f(x_1, x_2) = \left\| \frac{[\cos(x_1)]^4 + [\cos(x_2)]^4 - 2[\cos(x_1)]^2[\cos(x_2)]^2}{\sqrt{x_1^2 + 2x_2^2}} \right\| \quad (1)$$

s.t

$$0 \leq x_i \leq 10 \quad i = 1, 2 \quad (2)$$

$$x_1 x_2 > 0.75 \quad (3)$$

$$x_1 + x_2 < 15 \quad (4)$$

The global maximum is located at $\mathbf{x}^* = (1.601 \quad 0.469)$ where the value of the objective function is approximately $f(\mathbf{x}^*) = 0.365$. This lies on the constraint defined by (3). Other local maxima and their corresponding objective function values are tabulated in Table 1. This is a fairly difficult problem with multiple local maxima and a global maximum along a constraint boundary. A contour and a mesh plot of the function is shown in Figure 1.

1.2 Constraints

To deal with constraints, we adopt a simple penalty function [3] formulation, which converts the constrained problem as stated in section 1.1 into an unconstrained problem. This is described in greater detail in the following sections where the implementation and analysis of the BMCS and GA are discussed. For completeness, it was noted that barrier functions [3] may also be used.

Point	A	B	C	D
Coordinates \mathbf{x}	(3.088 1.517)	(1.550 3.075)	(0.470 1.599)	(4.660 0.161)
Objective Function $f(\mathbf{x})$	0.263	0.215	0.273	0.202

Table 1: Local Maxima for Keane's Bump Problem.

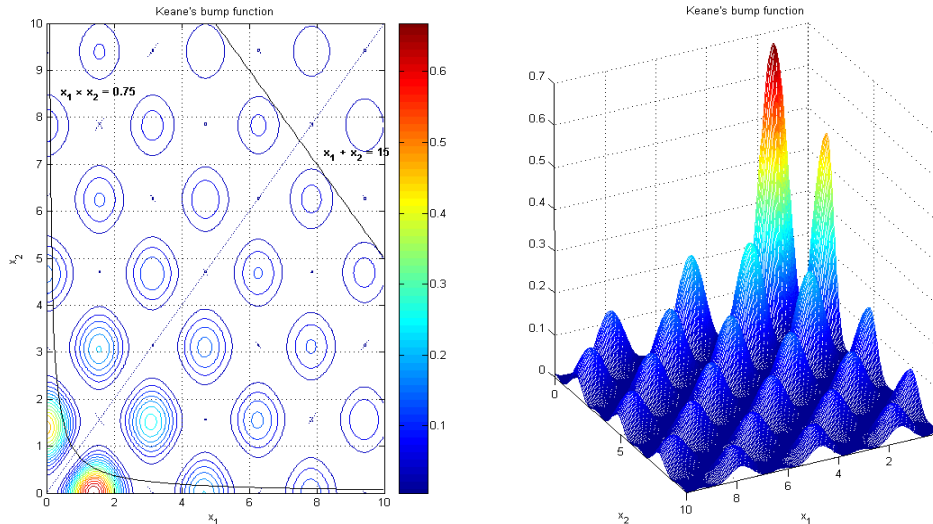


Figure 1: Contour and Mesh plots of Keane's Bump Function.

1.3 Structure of report

In Section 2, the Biased Monte Carlo Sampling Algorithm's performance on the Bump Function will be studied. In Section 3, the Genetic Algorithm's performance on the same problem will be examined. In these two sections, the effect of changing various parameters will be explored. A comparison of the two algorithms will be done in Section 4. Conclusions and perspectives will be presented in Section 5.

2 Biased Monte Carlo Sampling Algorithm

2.1 Program Structure

A implementation of the BMCS algorithm in MATLAB can be found in Appendix A. Firstly, the search space is divided into M regions. The horizontal axis is divided into N_1 strips and the vertical axis N_2 strips. After the value of the objective function at randomly selected point in each of the $M = N_1 N_2$ sub-regions has been calculated surveyNo times, the selection probability, as a function of the selection pressure S is calculated. The probabilities p are linear in S and they are stored in a tensor. The formula for calculating the selection probabilities is given by

$$p_j = \frac{S(M + 1 - 2R_j) + 2(R_j - 1)}{M(M - 1)} \quad (5)$$

where R_j is the rank assigned to the j^{th} region. The regions are then selected using an appropriately weighted roulette wheel and the process is repeated until a total of 5000 function evaluations have been performed. The penalty parameter associated with constraint given by equation (3) is increased linearly from 10 to 50 as the probabilities are updated during the course of the search. The new probabilities are stored in the matrix p_{cur} . Note that only points that lie in feasible space are sampled during the roulette wheel selection process.

Parameter	N_1	N_2	surveyNo	S	\mathbf{P}_{\min}	\mathbf{P}_{\max}	# Prob. Updates
Value	20	20	4	2	$\begin{pmatrix} 10 & 1 \end{pmatrix}$	$\begin{pmatrix} 50 & 1 \end{pmatrix}$	5

Table 2: Initialization of parameters for the BMCS algorithm.

2.2 Objective Function and Constraints

The code for calling the objective function can be found in Appendix B. The function takes in a matrix Chrom¹, which must have 2 columns and a penalty parameter p , which is a vector of length 2 because we have 2 constraints. An important change to the objective function was made. Instead of maximizing the Bump Function, it was sought to minimize the negative of the Bump Function. Hence the overall unconstrained minimization problem can be stated simply as

$$\min \quad f_A(\mathbf{x}) = -f(\mathbf{x}) + \mathbf{p}^T \mathbf{c}_V(\mathbf{x}) \quad (6)$$

$$\min \quad f_A(\mathbf{x}) = -f(\mathbf{x}) + p_1(\max[0, 0.75 - x_1x_2])^2 + p_2(\max[0, x_1 + x_2 - 15])^2 \quad (7)$$

where $f(\mathbf{x})$ is given in Equation (1).

2.3 Archiving Schemes

Two archiving schemes were carried out. Firstly, I implemented an archiving scheme which records (and outputs at the end of each run) the best 10 solutions found. The code can be found in Appendix C. Secondly, I implemented a dissimilarity archiving scheme [1] with $D_{min} = 1$ and $D_{sim} = 0.05$. The code can be found in Appendix D.

2.4 Initialization

The parameters of the optimization problem were initialized to the values as shown in Table 2. These are the default parameters that are used in the numerical experiments conducted. The search space was divided into $M = 400$ regions in which they were ranked after 4 points were randomly selected in each region and based on a selection pressure of $S = 2$. Selection probabilities were updated 5 times during the search. I ran the code with $K = 50$ fixed random seeds and typical plots of the search pattern are shown in Figure 2.

2.5 Discussion

The blue dots show the points evaluated during the initial survey where 4 random points were chosen in each region. The red dots show the points that are evaluated during the roulette wheel selection process based on the selection probabilities computed. The magenta circles show the $L = 10$ archived points. The plot on the left shows the effect of using the first archiving scheme and the plot on the right shows the effect of using the dissimilarity archiving scheme. I observed from Figure 2 that most of the red points are concentrated near the peaks of search terrain, which is expected. There are also no red points in the infeasible regions, indicating that all of the search, after the initial survey, is performed in feasible space. Besides, from the plot at the left, the archived points are mainly at the global optimum and point A (see Table 1). It is noted that there is an insufficient number of function evaluations near the global optimum at this stage. However, the plot on the right shows that most of the local maxima are located fairly efficiently by the BMCS algorithm. This includes all the local maxima as stated in Table 1. More importantly, the global optimum has been found.

¹This stands, quite clearly for Chromosomes, which is consistent with the terminology in the next section on Genetic Algorithms.

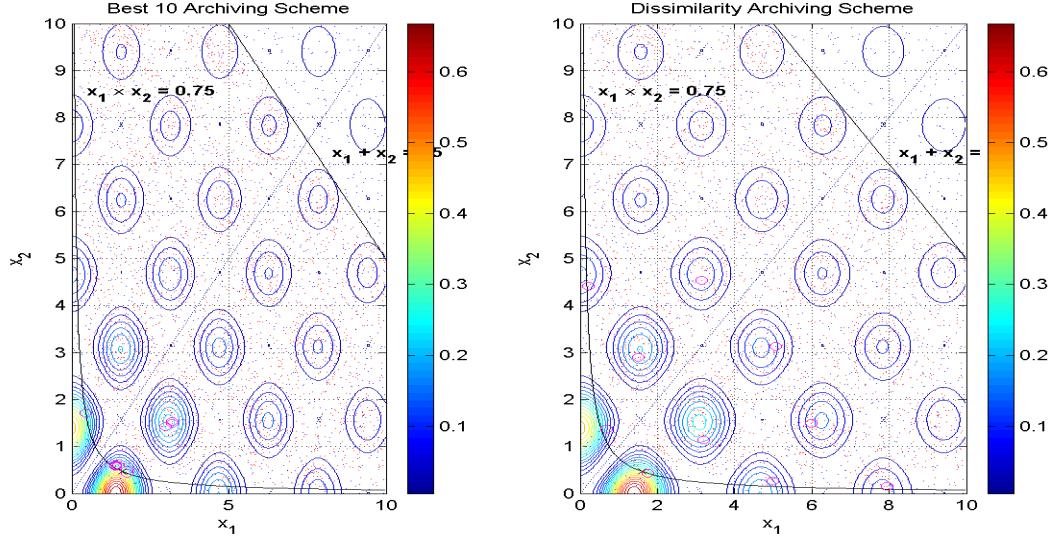


Figure 2: Typical Search Pattern in BMCS. Blue dots indicate solutions that are evaluated during initial survey. Red dots indicate solutions that are evaluated based on selection probabilities p_j . Magenta circles are the archived solutions. The left plot shows the search pattern and archived solutions for the best 10 archiving scheme. The right plot shows the search pattern and archived solutions for the dissimilarity archiving scheme.

2.6 Performance Measures

Before examining the effect of the various parameters on the BMCS algorithm, it is necessary to define a few performance measures. At the end of each run using the dissimilarity archiving scheme, the value of the best objective found is recorded. This is averaged over multiple runs. The standard deviation² in the value of the best objective found over multiple runs is also computed. The simulation was performed $K = 50$ times with fixed random seeds and $b_k = \max f(\mathbf{x}_k)$ where \mathbf{x}_k are the points sampled at the k^{th} iteration. The performance measures can be stated mathematically as

$$\mu = \frac{1}{K} \sum_{k=1}^K b_k \quad (8)$$

$$\sigma = \sqrt{\frac{1}{K-1} \sum_{k=1}^K (b_k - \mu)^2} \quad (9)$$

A good algorithm would produce a mean μ that is close to $f(\mathbf{x}^*) = 0.365$ and a standard deviation σ to be as small as possible. The c.p.u. run-time t_e for 5000 iterations is also recorded.

2.7 Analysis of Performance

In this section, the performance of the code will be analyzed. Various parameters will be changed to optimize the efficiency of the algorithm.

2.7.1 Changing $M = N_1 N_2$, the number of regions and \mathbf{p} the penalty parameter

In this numerical experiment, all the parameters were initialized to the values shown in Table 2. However, N_1 which is kept the same as N_2 is varied together with the \mathbf{p} penalty parameter vector \mathbf{p} . The second

²Note that the unbiased estimator of the standard deviation is used.

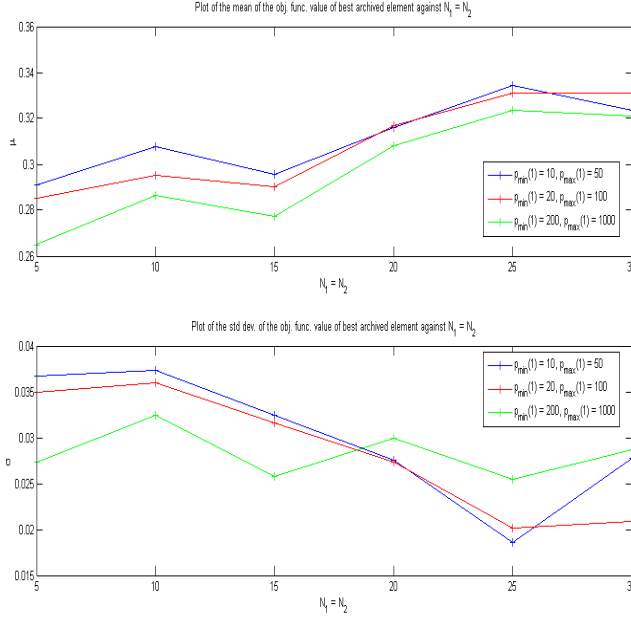


Figure 3: Variation of the statistics μ and σ of best objective value in archive against $N_1 = N_2$ for different values of the penalty parameter.

Parameter	N_1^*	S^*	\mathbf{P}_{\min}^*	\mathbf{P}_{\max}^*
Optimal Value	25	1.4-1.6	$\begin{pmatrix} 10 & 1 \end{pmatrix}$	$\begin{pmatrix} 50 & 1 \end{pmatrix}$

Table 3: Optimal Values for the parameters for the BMCS algorithm.

element of \mathbf{p} , corresponding to the constraint in equation (4) is kept at unity. The first element is changed and μ and σ are calculated. The results are shown in Figure 3. Note that the number of survey points in each region stays constant at 4. Hence, as the number of regions increases, the remaining number of search points following the initial survey decreases.

Clearly, it is observed that as we increase the number of regions $M = N_1 N_2$, the better the BMCS algorithm performs for a fixed set of penalty parameters. This is the case even though the number of objective function evaluations based on the selection probabilities decreases as the number of regions increases. The search space resolution is better when M increases and hence the likelihood of getting close to the optimal value of \mathbf{x} increases. From Figure 3, it is observed that the optimal value for $N_1 = N_2$ is around 25 and the penalty parameters should take the values $\mathbf{p}_{\min} = \begin{pmatrix} 10 & 1 \end{pmatrix}$ and $\mathbf{p}_{\max} = \begin{pmatrix} 50 & 1 \end{pmatrix}$.

2.7.2 Changing S , the selection pressure

The penalty parameters and N_1 were reset to $\mathbf{p}_{\min} = \begin{pmatrix} 10 & 1 \end{pmatrix}$ and $\mathbf{p}_{\max} = \begin{pmatrix} 50 & 1 \end{pmatrix}$ and $N_1 = N_2 = 20$ respectively. Now the selection pressure S was changed from 1 to 2 in steps of 0.2. μ and σ were computed for each value of S and the results are summarized in Figure 4. It was observed that S does not play a terribly significant role in the performance of the algorithm, probably because probabilities are updated periodically, which has a smoothing effect. From Figure 4, it is observed that the optimal value for S lies in the range 1.4 to 1.6.

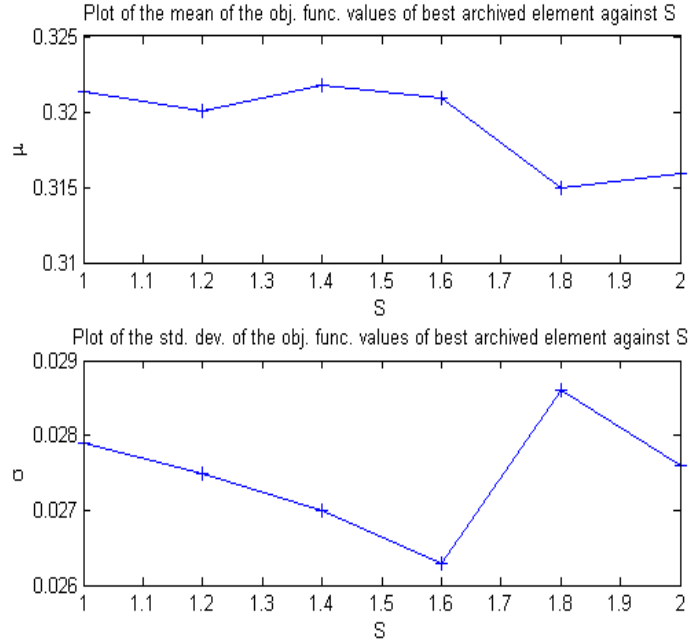


Figure 4: Variation of the statistics μ and σ of best objective value in archive against S .

2.8 Concluding remarks on the BMCS Algorithm

The optimal values for the parameters $N_1 = N_2$, the penalty parameters and the selection pressure are summarized in Table 3. Finally, the average c.p.u run time t_e for each BMCS run is around 1.2820 seconds.

3 Genetic Algorithm

3.1 Program Structure

The MATLAB program was adapted from <http://www.shef.ac.uk/~gaipp/ga-toolbox/>. The main program and other important functions can be found in Appendix E. The first few lines of code set the parameters that the GA uses, such as the number (N_{ind}) and length (PRECI) of chromosomes, the crossover (XOV) and mutation (MUT) rates the number of generations and the binary representation scheme. Next, an initial uniformly distributed, entirely random binary population, Chrom, is created using the function `crtbp`. The objective function `objfun3`, which will be discussed in greater detail in the next subsection, is evaluated to produce the vector of objective values `ObjV`.

The generational loop executes single-point crossover (`xovsp`) and mutation (`mut`) until the objective function is evaluated a total of 5000 times. Finally, new individuals are re-inserted into the population using the function `reins`, and the generational counter, `gen` is incremented. Plots depicting the search pattern and the objective function reduction are also produced.

3.2 Objective Function and Constraints

Similar to the approach adopted in Section 2, a penalty function approach was used. Also since minimizing the negative of the objective function in (1) will result in negative values, a non-linear ranking scheme [2] was employed. To handle the constraints in a GA, the approach discussed in [1], (chapter 3) was used. Hence

Parameter	N_{ind}	GGap	PRECI	XOV	MUT	S	\mathbf{p}
Value	100	1.0	30	0.95	0.01	2.0	(5000 1)

Table 4: Initialization of parameters for GA.

the unconstrained problem can be stated as

$$\min \quad f_A(\mathbf{x}) = -f(\mathbf{x}) + M^k \mathbf{p}^T \mathbf{c}_V(\mathbf{x}) \quad (10)$$

where M is the generation number and $k = 3$ is a constant exponent. The dependence of the penalty on generation number biases the search increasingly heavily towards feasible space as the search progresses. The code for the objective function objfun3, can be found in Appendix F.

3.3 Archiving Schemes

Again, two different archiving schemes were used. Firstly, an archiving scheme that records and outputs the best 10 solutions found was implemented. Following that, the dissimilarity archiving scheme was used to evaluate the performance of the GA. These codes can be found in Appendix C and Appendix D respectively.

3.4 Initialization

The Genetic Algorithm was initialized with the parameters as shown in Table 4. These are the default parameters that are used in the numerical experiments conducted. A plot of the evolution of the algorithm through the various generations can be found in Figure 5. The final search pattern for the two different archiving schemes is shown in Figure 6. Lastly, the graph in Figure 7 shows the reduction of the objective function through the generations for a typical run. The best (highest fitness value) phenotype was identified and its corresponding objective function value was plotted against the generation counter.

N_{ind} is defined as the number of individuals in a population. GGap is the generation gap, i.e. the fraction of new individuals that are created from their parents. PRECI is the number of bits that are used to represent the binary chromosomes. XOv and MUT are the crossover and mutation probabilities respectively and S is the selection pressure used in the nonlinear ranking process. \mathbf{p} is the vector of penalty parameters. Finally, only single point crossover is considered unless otherwise stated.

3.5 Discussion

From Figure 5, we observe that as the search progresses, the population concentrates at the global optimum. Convergence was achieved after about 25 generations. Figure 6 shows the search pattern after 5000 objective function evaluations have been performed. The top plot shows that if the Best 10 archiving scheme is used, all the elements of the archive are at the global optimum. This shows that there are ample searches in and around the global optimum. Clearly, this is an advantage of the GA over the BMCS. From the bottom plot, it is observed that the archived points are the main peaks of the search space. More importantly, the global optimum has been found. Figure 7 substantiates the earlier claim that the global optimum was successfully located after the 25th generation. In fact, there is unnoticeable improvement after the 16th generation.

3.6 Performance Measures

As in the previous section on the BMCS, two performance measures were used. In equation (8), μ is defined as the mean of the best objective function value in the archive over $K = 50$ runs of the algorithm. σ , defined in equation (9), is the corresponding standard deviation. Note that for reproducibility and consistency, the same $K = 50$ random seeds were used throughout the numerical experiments that are detailed in the following sections.

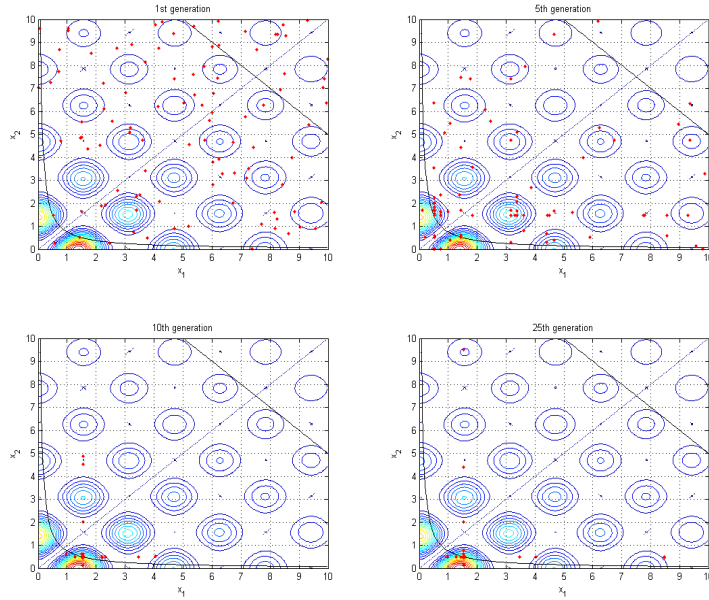


Figure 5: Evolution of the GA.

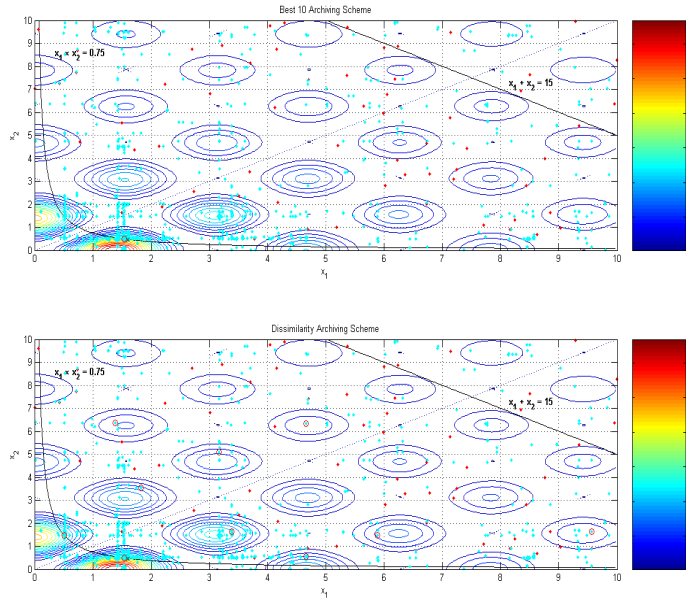


Figure 6: Search pattern of the GA. The plot on the top shows the Best 10 archiving scheme. The plot on the bottom shows the dissimilarity archiving scheme. The initial population is in red and subsequent phenotypes are in cyan. Archived points are the red circles. Note from the top plot that there appears to be only 1 archived point. In fact there are $L = 10$ but all of them are coincident.

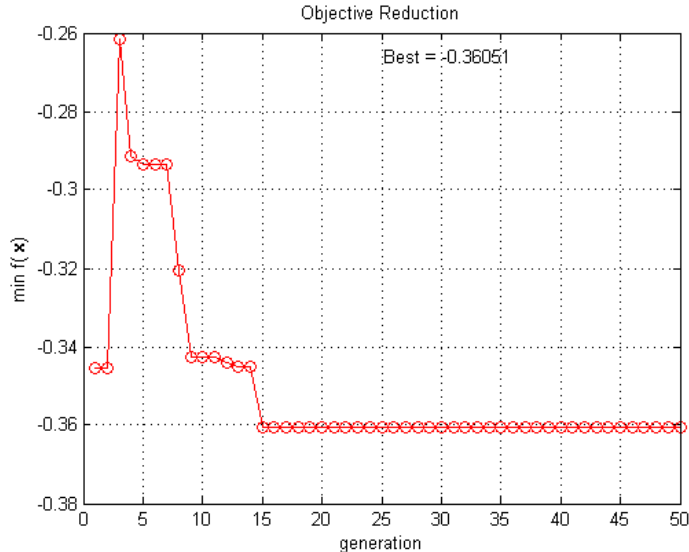


Figure 7: Objective function reduction.

3.7 Analysis of performance

In this section, the performance of the GA will be analyzed. Various parameters, including N_{ind} , XOY and MUT will be changed to optimize the efficiency of the algorithm.

3.7.1 Changing N_{ind} , the number of individuals in a population

The number of individuals N_{ind} was varied. A total of 5 samples ($N_{ind} = [25, 50, 100, 200, 250]$) were used and the same statistical criteria μ and σ as in section 2 were computed for each GA run. The results are summarized in Figure 8. It is clear that the higher the value of N_{ind} , the more efficient the algorithm. μ increases and σ decreases both monotonically as N_{ind} increases. I postulate that this is because the more individuals there are in a population, the more effectively crossover and mutation occur in the population, giving rise to more diverse offspring, making it easier to locate the global optimum in the search space.

3.7.2 Changing XOY, the crossover rate

The crossover rate XOY plays an important role in the GA. In essence, it is the probability of crossover of 2 individuals in a population to form a hopefully fitter offspring. Here the XOY rate is varied from 0.5 to 1 in steps of 0.05 and the results are summarized in Figure 9. We see that the optimal value of XOY, the one that gives the highest value of μ and the lowest value of σ , is a moderate value of 0.70.

3.7.3 Changing the number of crossover points in the binary string

For this experiment, all the parameters were reset to the values given in Table 4. The single (hereby called ‘xovsp’) and multi-point crossover (hereby called ‘xovmp’) schemes were tested. In a xovmp, the crossover operator is applied to consecutive pairs of individuals in several points in the binary representation of the individuals. This happens with crossover probability XOY. This is shown diagrammatically in Figure 10. The results are summarized in Table 5. Unsurprisingly, using a xovmp results in a significantly higher average. This shows that the offspring generated by the parents has to be more diverse to locate the global optimum because this is a difficult, multi-modal optimization problem.

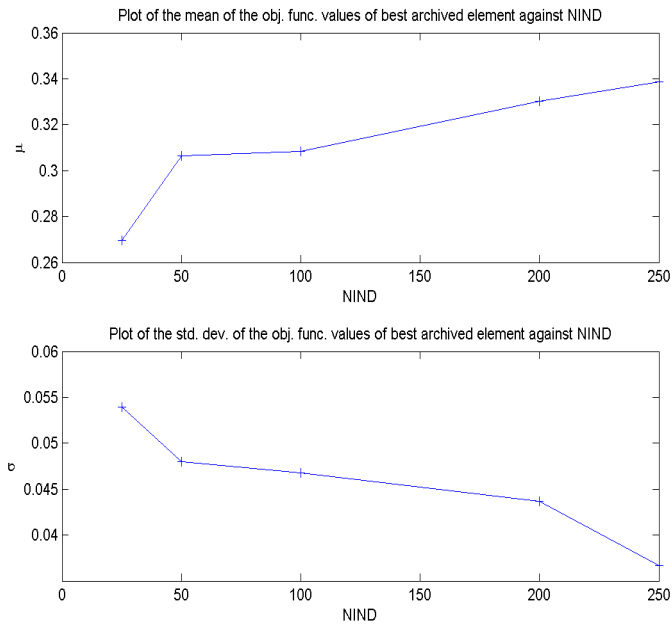


Figure 8: Variation of the statistics μ and σ of best objective value in archive against N_{ind} .

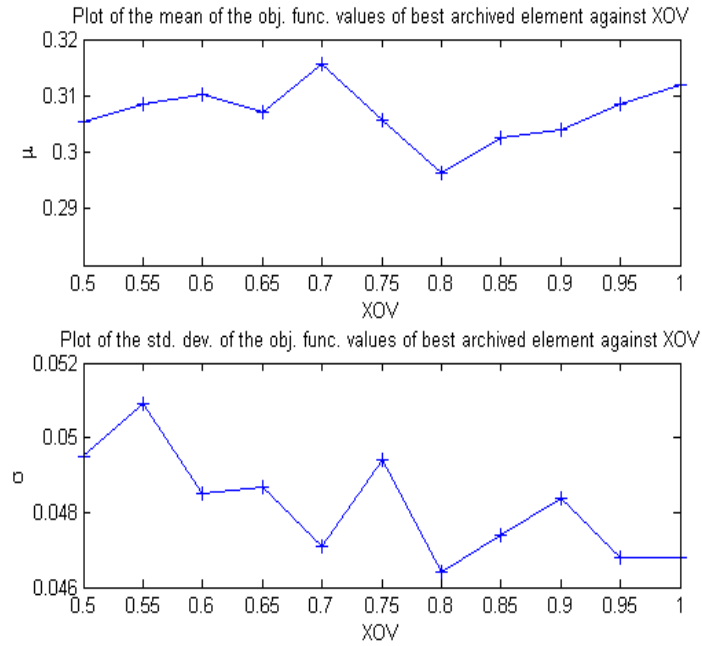


Figure 9: Variation of the statistics μ and σ of best objective value in archive against XOV.

Measure	xovsp	xovmp
μ	0.3084	0.3234
σ	0.0468	0.0461

Table 5: Comparison between single (xovsp) and multi-point (xovmp) crossover in a GA.

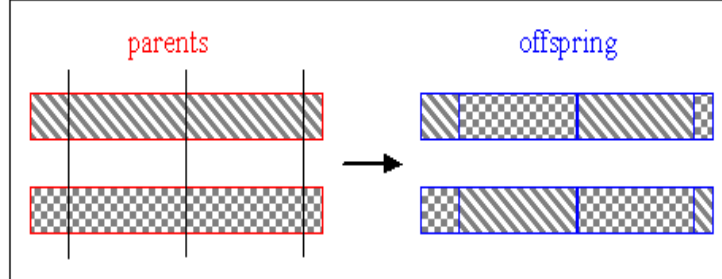


Figure 10: Multiple point crossover (xovmp) in a GA

3.7.4 Changing MUT, the mutation rate

The mutation rate is defined as the probability with which each gene changes value in a child. The mutation rate took on 12 values from 10^{-4} to $10^{-1/2}$, spaced equally on a logarithmic scale. The results are plotted in Figure 11. Note the logarithmic scale on the horizontal axis. It can be observed that in general, the algorithm requires a pretty high mutation rate to perform well. This is hardly surprising since the optimization problem is difficult and multi-modal. Repeated mutations are required to locate fitter individuals if the optimization is trapped in a local maximum. The optimal value for the mutation rate is in the region $10^{-.792} \approx 0.162$. This is a relatively high value. Textbook values for the mutation rate range from 0.001 to 0.01 but clearly, these values do not produce the best results for the GA. Also, it is noted that increasing MUT biases the search towards a more stochastic search, much likened to a Monte Carlo simulation. Hence, a GA with a high mutation rate can be considered to be a hybrid algorithm, possessing the good traits of both a random Monte Carlo search and a canonical GA.

3.7.5 Changing S, the selection pressure

The selection pressure used in the GA determines the degree of bias in the nonlinear ranking of the individuals in a population. The selection pressure typically ranges between 1 to 2 and these are the two boundary values of the samples. A total of 6 samples between 1 to 2 (inclusive) are used in this numerical experiment and the results are shown in Figure 12. Somewhat surprisingly, the smaller the value of the selection pressure, the better the algorithm performs. However, the changes in μ and σ are not significant over the range of S values that were used, indicating that the actual value of S is probably rather insignificant in the overall algorithm.

3.7.6 Changing PRECI, the number of bits used to represent binary chromosomes

PRECI was defined to be the number of bits used to represent the binary chromosomes in a population. It was varied from 20 to 40 in steps of 5. Very surprisingly, even though μ showed no noticeable trend, the standard deviation of the best archived element σ showed a clear increasing trend as PRECI was increased from 20 to 40. I postulate that 20 bits are sufficient to represent the binary string and it is not advantages to use an excessive number of bits. This will result in a larger variation of the results, which is undesired. We want the algorithm to consistently produce good results.

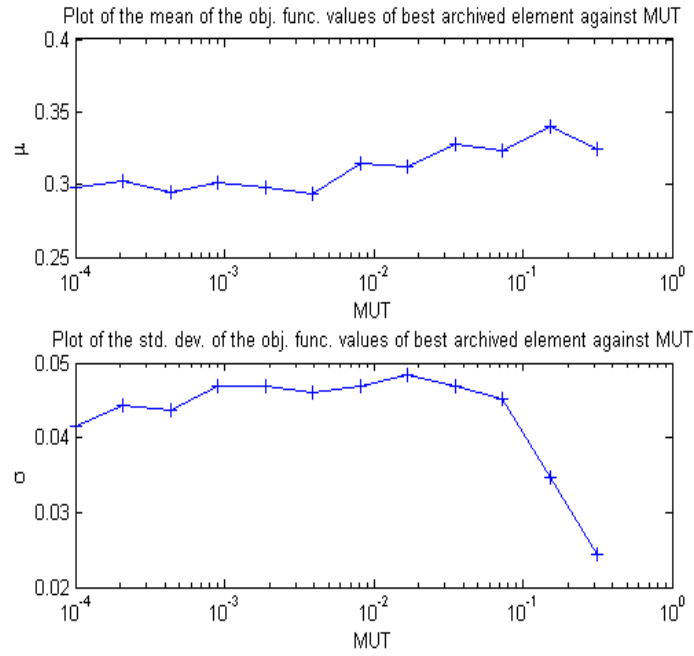


Figure 11: Variation of the statistics μ and σ of best objective value in archive against MUT.

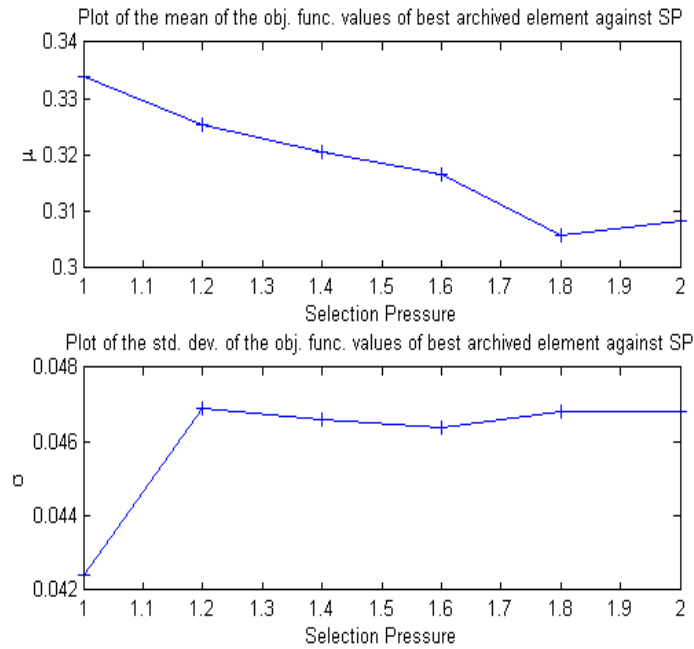


Figure 12: Variation of the statistics μ and σ of best objective value in archive against S .

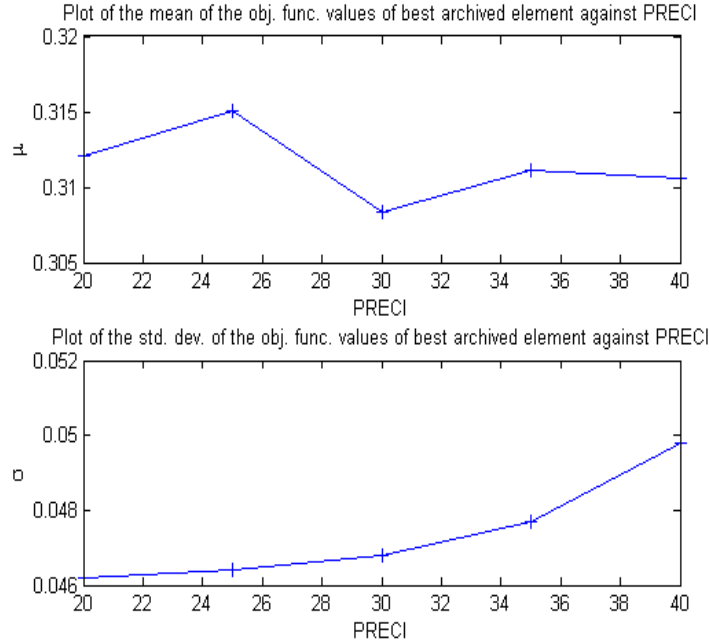


Figure 13: Variation of the statistics μ and σ of best objective value in archive against PRECI.

Parameter	N_{ind}^*	XOV*	XOV method*	MUT*	S^*	PRECI*
Optimal Value	250	0.70	xovmp	0.162	1.0	20

Table 6: Optimal Values for the parameters for the GA.

3.8 Concluding remarks on the GA

The GA was successfully adapted and various experiments were performed. I found the optimal values of the parameters for the Keane’s Bump Problem. These values are tabulated in Table 6. Finally, the average c.p.u run time t_e for each BMCS run is around 2.1356 seconds.

4 Comparison and Comments

In this section, I will qualitatively and quantitatively compare and contrast the performance of the two algorithms that were mentioned in Sections 2 and 3. The efficiency, ability to locate the global optimum and the computational time will be discussed briefly.

4.1 Efficiency

From Figures 2 and 6, it is observed that the BMCS search algorithm performs poorly when compared to the GA. From Figure 6, we observe that the GA is able to converge at the global optimum quickly and efficiently. There are also ample function evaluations around the region surrounding the global optimum as evidenced by the elements in the archive when using the Best 10 archiving scheme. In comparison, the BMCS is truly a random search and it is computationally expensive especially for difficult real life problems where the number of function evaluations is limited by the physical system. There are an insufficient number of function evaluations in and around the global optimum as can be seen from Figure 2. This reduces the quality of the search significantly.

Opt. Method	BMCS	GA
c.p.u run time t_e (secs)	1.2820	2.1356

Table 7: Comparison of t_e between the two different methods.

4.2 Location of Global Optimum

From the comparison graphs (Figures 3 and 4 and 8 to 13), it is observed that both methods can locate the global optimum. μ in both algorithms is close to $f(\mathbf{x}^*) = 0.365$. The BMCS, as mentioned, is rather inefficient in its search and occasionally misses the global optimum especially if the subdivision of the search space is too coarse. The GA is susceptible to convergence at local maxima but the performance vastly improves if the parameters are changed the optimal ones in Table 6. The search not only concentrates around the global optimum, it also locates the the global optimum in a larger percentage of the runs. In my report, I did not comment on the number if maxima located by the dissimilarity archiving scheme. That could well be another performance measure that one could adopt. In this report, I focused mainly on the global optimum and in particular how far the best solution is from the global optimum at $\mathbf{x}^* = (1.601 \quad 0.469)$.

4.3 C.P.U. run time

Each run of the GA takes rather longer than the BMCS as can be seen in in Table 7. This is due to the fact that there are rather more operations in the GA. For example the crossover, mutation and reinsertion operators require a significant amount of computational power. Both algorithms involve a total of 5000 function evaluations.

4.4 Comments on the Biased Monte Carlo Sampling Algorithm

It was observed that as $N_1 = N_2$ increases, the mean μ increases and σ decreases. Thus the performance is improved. This is because as the grid gets finer, the ability to local the global maximum improves. Besides, the penalty parameter vector \mathbf{p} cannot be allowed to be large. A moderate value suggested in section 2 should be used. Besides, a moderate value of the selection pressure should be used to ensure a large μ and small σ .

From the numerical experiments in section 2, we note that the BMCS performs reasonably well. The average of the best objective function value observed $\mu_{BMCS}(\boldsymbol{\lambda}_{BMCS}^*) \approx 0.330$, approximately 10% lower than the value of the objective function at the global optimum $f(\mathbf{x}^*)$. Here $\boldsymbol{\lambda}_{BMCS}^*$ is the vector of optimal parameters for the BMCS.

4.5 Comments on the Genetic Algorithm

The GA uses a more sophisticated method to locate the global optimum. Two important results were observed from the analysis in section 3. They are worth highlighting here.

1. The optimal mutation rate is rather large. This is because the problem is multi-modal and furthermore the global optimum lies on a constraint boundary. If the algorithm is “stuck” in a local maxima, an effective mutation, one that leads to the global optimum, must occur before the results can be improved.
2. The number of bits used in the binary representation of the control variables directly influences the variance of the results. One ought to use around 20 binary bits for this problem. Besides reducing the variability of the results, the c.p.u. run time would be faster as the matrices in the code are smaller.

The GA performs well. The average of the best objective function value observed $\mu_{GA}(\boldsymbol{\lambda}_{GA}^*) \approx 0.345$. Again $\boldsymbol{\lambda}_{GA}^*$ is the vector of optimal parameters for the GA. This is a substantial improvement when compared to the BMCS. This is especially so if the mutation rate MUT is changed to a value between 0.1 and 0.2, a

large value when compared to traditional GAs. In fact $\mu_{GA}(\lambda_{GA}^*)$ is a mere 5% lower than the value of the objective function at the global optimum $f(\mathbf{x}^*)$.

5 Conclusion and Perspectives

Both the BMCS algorithm and GA were successfully implemented in MATLAB. They were tested on a difficult, nonlinear, multi-modal, constrained optimization problem, namely the Keane's Bump Problem. We observed that in general the GA performs better than the BMCS when it comes to locating the global optimum. There are ample objective function evaluations in and around the global optimum. The BMCS algorithm does well to locate other local maxima.

Furthermore, I have done a series of numerical experiments to determine the optimal BMCS and GA parameters for the Keane's Bump Problem. These are summarized in Tables 3 and 6. Finally, I would like to highlight the optimal and unconventionally high value of the mutation rate in the GA. This indicates that the search for the global optimum requires a combination of both a random search and a more systematic, traditional GA. In general, if the optimal values for the various parameters are used, the GA performs better than the BMCS.

References

- [1] G.T. Parks. 5R1 Lecture Notes. 2005.
- [2] D.E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison Wesley Publishing Company, Reading, MA, 1989.
- [3] Luenberger D.G. *Introduction to linear and nonlinear programming*. Addison Wesley Publishing Company, Reading, MA, 1973.

APPENDICES

A BMCS code

```
function [l,obj,gopt,elapsed_time] = MCKeane(stateNo,S)

% Biased Monte Carlo Simulation on Keane's Bump Function
% Inputs - surveyNo, S, N1, N2, L
% Outputs - archive l
% Written by:
% Vincent Tan
% 02 February 2005

rand('state',stateNo);

stime = clock;           % Start time

%%%%%% Boundary values
surveyNo = 4;           % Number of times to eval obj. function in each square.
%S = 2;                 % Selection Pressure
edges = [0 10; 10 0];  % Boundaries

% Tile x1-x2 plane
N1 = 20;                % Number of discretizations in each dimension: Equal in each dimension
N2 = N1;                % Number of discretizations in each dimension: Equal in each dimension
side = 10/N1;           % Length of each side of square
M = N1*N2;              % Total number of squares

% D_min high => identify lower valleys but less dissimilarity
% D_min low => identify higher valleys but more dissimilarity
D_min = 1; D_sim = .05; % Thresholds
l = [0.1 0.1];          % Random initialization.
L = 10;                 % Number of elements in archive
counter = zeros(N1,N2); % Count the number of times objective function is evaluated in a particular subspace
objCounter = 0;         % Count number of times objective function is evaluated
switch_archive = 0;     % 1 - Best, Default - Dissimilarity archiving scheme
switch_plot = 0;        % Plot: On = 1, Off = 0;
gopt = [1.6009 0.4685]; % Global optimum

%%%%%% Survey and calculation of probabilities
j = 0; pen = [10 1]; func1 = objKeane(l,pen);

for l1 = edges(1,1):side:edges(1,2)-side
    j = j+1;
    i = 0;
    for l2 = edges(2,1):-side:edges(2,2)+side
        i = i+1;
        u1 = l1+side; u2 = l2-side;
        for k = 1:surveyNo

            % Evaluate coords of square
            x1 = l1+(u1-l1)*rand(1,1);
            x2 = u2+(l2-u2)*rand(1,1);

            if (switch_plot == 1)
                plot(x1,x2,'b'); hold on; %drawnow;
            end

            % Evaluate Keane's function
            x = [x1 x2];
            f(k) = objKeane(x,pen); objCounter = objCounter+1;
        end
    end
end
```

```

    % Archive solutions
    if (switch_archive == 1)
        % Only if constraints are satisfied
        if ((x(1)*x(2)>.75) & (x(1)+x(2)<15))
            [l,funcl] = archive_best(x,f(k),l,funcl,L);
        end
    else
        if ((x(1)*x(2)>.75) & (x(1)+x(2)<15))
            [l,funcl] = archive(x,f(k),l,funcl,D_min,D_sim,L,pen);
        end
    end
    counter(i,j) = counter(i,j)+1;
end

% Find the average of the objective function for this square and
% store in a matrix.
fave(i,j) = sum(f)/surveyNo;
clear f;
end
end

%%%%%% Assign rank numbers to each square.

% Arrange into a vector.
faveVec = fave(:); % Column

% Sorted vector
faveVecSort = sort(faveVec);

for i = 1:N1
    for j = 1:N2
        k = 1;
        while(fave(i,j) ~= faveVecSort(k))
            k = k+1;
        end
        sorted(i,j,1) = k; % k is the counter
    end
end

% Assign probabilities
p(:, :, 1) = (S.*(M+1-2.*squeeze(sorted(:, :, 1)))+2.*(squeeze(sorted(:, :, 1))-1))./(M*(M-1)); p_cur = p(:, :, 1);

% Some other constants
values = 1:M; % For ranking purposes
N = 5e3 - M*surveyNo; % Number of times obj. function is evaluated after initial probs calculated
noUpdateTimes = 5; % Number of times p is updated using current values of obj. function
f1 = max(max(fave)).*ones(N1,N2); % Random initialization
f1 = zeros(N1,N2); % Random initialization
maxpen = [50 pen(2)]; minpen = [10 pen(2)]; % Penalty parameters to be increased linearly

%%%%%% Until convergence do
for j = 1:noUpdateTimes

    % Increase the penalty parameter as we go along
    m = (maxpen(1)-minpen(1))/(5e3-1);
    c = minpen(1)-m;
    pen_cur(j, :) = [m*j+c pen(2)];

    % Used if noUpdateTimes == 1
    if noUpdateTimes == 1
        pen_cur(j, :) = [20 pen(2)];
    end

    innerCounter = 0;

```

```

%for i = 1:N/noUpdateTimes
while (innerCounter < N/noUpdateTimes)
    % Sample points according to probability distribution p_{ij}
    pVec = p_cur(:);
    x0 = discrete_sample(1,values,pVec);

    % Associate value of x0 to a bin.
    x_N = ceil(x0/N1); % Column of the p matrix i.e. 'j'
    r = mod(x0,N1); % Row of the p matrix i.e. 'i'
    if (r == 0)
        r = N1; % Slight adjustment
    end
    counter(r,x_N) = counter(r,x_N)+1;

    % Identify the box
    l1 = edges(1,1)+(x_N-1)*side; u1 = l1+side;
    l2 = edges(2,1)-(r-1)*side; u2 = l2-side;

    % Generate points
    x1 = l1+(u1-l1)*rand(1,1);
    x2 = u2+(l2-u2)*rand(1,1);
    x = [x1 x2];

    % Reject solutions that violate constraints
    if ((x(1)*x(2))>.75) & (x(1)+x(2)<15)

        % Evaluate objective function
        f11 = objKeane([x1 x2], pen_cur(j,:)); objCounter = objCounter+1;
        innerCounter = innerCounter+1;

        % Archive solutions
        if (switch_archive == 1)
            [l,func1] = archive_best(x,f11,l,func1,L);
        else
            [l,func1] = archive(x,f11,l,func1,D_min,D_sim,L,pen_cur(j,:));
        end

        % Plot results
        if (switch_plot == 1)
            plot(x1,x2,'r'); hold on; %drawnow;
        end

        % Book-keeping
        %f1(r,x_N) = f1(r,x_N) + objKeane([x1 x2],pen_cur(j,:)); % Compute
        %objective function here and sum.
        f1(r,x_N) = f1(r,x_N)+f11;
    end
end % End inner loop

%% Periodically update probabilities using current values of obj. function
[p(:,:,j+1), sorted(:,:,j+1)] = update(fave,f1,counter,N1,N2,surveyNo,S);
p_cur = p(:,:,j+1);
end
% End loop

% Total run-time of simulation in seconds
elapsed_time = etime(clock,stime);

if (switch_plot == 1)
    % Sketch contours of the Keane's bump function.
    plotKeane;
    title('Biased Monte Carlo Simulation on Keane's bump function');

    % Plot points in archive.
    plot(squeeze(l(:,1)),squeeze(l(:,2)), 'mo');
end

```

```

    % Plot global optimum
    plot(gopt(1), gopt(2), 'kx');
end

%%%%%%%%%%%% Output results
l = l'; obj = -funcl';
% disp('Results of Biased Monte Carlo Simulation');
% disp('-----');
% disp('The 10 best points are located at');
% disp('which have objective function values of');
% disp('respectively.')
```

B Objective Function code

```

% OBJKEANE.M      (OBJective function for Keane's Bump FUNction 3)
%
% This function implements the Keane's BUMP function.
%
% Syntax:  ObjVal = objKeane(Chrom,p)
%
% Input parameters:
%   Chrom   - Matrix containing the chromosomes of the current
%             population. Each row corresponds to one individual's
%             string representation.
%             if Chrom == [], then speziell values will be returned
%   p       - Penalty Parameter (to be increased as we iterate along)
%
% Output parameters:
%   ObjVal  - Column vector containing the objective values of the
%             individuals in the current population.
%
% The optimal value of Keane's BUMP function is 0.36497974409160
% which occurs at [1.6009000000000000  0.46848647635705]
%
% Author:      Vincent Tan

function ObjVal = objKeane(Chrom,p);

x1 = Chrom(:,1); x2 = Chrom(:,2);
ObjVal = -abs(((cos(x1)).^4+(cos(x2)).^4-2.*((cos(x1)).*(cos(x2))).^2)./sqrt(x1.^2+2.*x2.^2));

for i = 1:length(x1)
    Penalty(i) = p(1)*(max([0 0.75-x1(i)*x2(i)]))^2+p(2)*(max([0 x1(i)+x2(i)-15]))^2;
end

ObjVal = ObjVal + Penalty';

% End of function
```

C Best 10 solutions archiving code

```

function [l_new,obj_l_new] = archive_best(J,funcJ,l_old,obj_l_old,L)
% BEST L POINTS ARCHIVE SYSTEM
% Syntax
% [l_new,obj_l_old] = archive_best(J,funcJ,l_old,obj_l_old,L)
% Archive new point
% Inputs
% L - the number of 'best' solutions.
```

```

% l_old - old matrix of size m by 2
% obj_l_old - objective functions of the points in the archive
% J - new point to be considered
% funcJ = func(J)
%
% Output
% l_new - new vector of size m by 2
% Written by:
% Vincent Tan
% 02 February 2005

[m,n] = size(l_old);
% n = 2 i.e. we should have 2 columns.
% m = no. of points we currently have in our archive

%for i = 1:m
    %f = objKeane(l_old,pen); % Evaluation of objective function
%end

[fmax, imax] = max(obj_l_old); % Worst point in archive and its location
[fmin, imin] = min(obj_l_old); % Best point in archive and its location

if (m < L)
    % Just append our archive
    l_new = [l_old; J]; obj_l_new = [obj_l_old; funcJ];
else
    % Compare worst point to new point
    if (funcJ < fmax)
        % Replace
        l_new = replace(l_old,J,imax);
        obj_l_new = replace(obj_l_old,funcJ,imax);
    else
        % Do not replace. This new point is worse.
        l_new = l_old;
        obj_l_new = obj_l_old;
    end
end
end

```

D Dissimilarity archiving code

```

function [l_new,obj_l_new] = archive(J,funcJ,l_old,obj_l_old,D_min,D_sim,L,pen)

% DISSIMILARITY ARCHIVE SYSTEM
% Archive new point
% Syntax
% [l_new,obj_l_new] = archive(J,funcJ,l_old,obj_l_old,D_min,D_sim,L)
% Inputs
% L - the number of 'best' solutions.
% l_old - old matrix of size m by 2
% obj_l_old - objective functions of the points in l_old
% J - new point to be considered
% funcJ = objKeane(J)
% D_min and D_sum - thresholds
%
% Output
% l_new - new vector of size m by 2
% Written by:
% Vincent Tan
% 02 February 2005

% if (l_old == []) % To cope with the null set

```

```

%      l_new = [l_old J];
% else

[m,n] = size(l_old);

for i = 1:m
    d(i) = norm(J-l_old(i,:)); % Euclidean distances
end

% f(i) = objKeane(l_old(i,:),pen); % Evaluation of objective function

[d_min,d_min_loc] = min(d); % Minimum Euclidean distances
[d_max,d_max_loc] = max(d); % Minimum Euclidean distances
[f_min,f_min_loc] = min(obj_l_old); % Best point in archive
[f_max,f_max_loc] = max(obj_l_old); % Worst point in archive

if (m < L) % Fewer than L solutions
    % Append if x_J is sufficiently dissimilar to all solns archived.
    if (d_min > D_min)

        l_new = [l_old; J]; obj_l_new = [obj_l_old; funcJ];
    else
        % Don't archive
        l_new = l_old; obj_l_new = obj_l_old;
    end

elseif (m==L) % Archive is full
    % Replace if x_J is sufficiently dissimilar to all solns archived.
    % AND better than the worst of these.
    if ((d_min > D_min) & (funcJ < f_max))
        % Replace worst solution x_G at location f_max_loc

        l_new = replace(l_old,J,f_max_loc); obj_l_new =replace(obj_l_old,funcJ,f_max_loc);
    elseif ((d_min< D_min) & (funcJ < f_min))
        % Replace solution that it most closely resembles x_E at location
        % d_min_loc

        l_new = replace(l_old,J,d_min_loc); obj_l_new =replace(obj_l_old,funcJ,d_min_loc);
    elseif ((d_min < D_min) & (funcJ < objKeane(l_old(d_min_loc,:),pen)) & (d_min < D_sim))
        % Replace solution that it most closely resembles x_E at location
        % d_min_loc

        l_new = replace(l_old,J,d_min_loc); obj_l_new =replace(obj_l_old,funcJ,d_min_loc);
    else

        l_new = l_old; obj_l_new = obj_l_old;
    end
end
else

end
end

```

E GA codes

```

function [l,obj,fmax,gopt,elapsed_time] = sga(stateNo)

% sga
%
% This script implements the Simple Genetic Algorithm described
% in the examples section of the GA Toolbox manual.
%

% Author:      Andrew Chipperfield
% History:    23-Mar-94      file created

```

```

% Adapted by Vincent Tan Feb 2005

% clear; clc;
rand('state',stateNo);

% Timer
stime = clock;          % Start time

% GA Parameters
NIND = 100;             % Number of individuals per subpopulations
MAXGEN = 5e3/NIND-1;   % maximum Number of generations
GGAP = 1.0;            % Generation gap, how many new individuals are created
NVAR = 2;              % Dimensionality of problem / No. of decision variables
PRECI = 30;            % Precision of binary representation
XOV = .95;            % Crossover probability
MUTR = .01;           % Mutation Probability
SP = 2.0;              % Selection Pressure
RFun = [SP 1];        % Ranking method: SP = 2, NL ranking

% Archiving Parameters
switch_archive = 0;    % Archiving scheme: 1 - Best, Default - Dissimilarity archiving scheme
L = 10;               % Number of points in archive
D_min = 1;            %
D_sim = .05;          % Thresholds

% Other parameters
gopt = [1.6009 0.4685]; % Global Optimum
switch_plot = 0;      % Plot: On = 1, Off = 0;

% Build field descriptor
FieldD = [rep([PRECI],[1, NVAR]); 0 0; 10 10; 1 1; 0 0; 1 1; 1 1];

% Initialise population
Chrom = crtbp(NIND, NVAR*PRECI);

% Reset counters
Best = NaN*ones(MAXGEN,1); % best in current population
Ave = NaN*ones(MAXGEN,1); % mean of current population
gen = 0;                   % generational counter

% Evaluate initial population
Phen = bs2rv(Chrom,FieldD); ObjV = objfun3(Phen,gen);

% Start archiving
l = [0.1 0.1]; funcl = objfun3(l,gen); if (switch_archive == 1)
    for i = 1:NIND
        P = Phen(i,:);
        if ((P(1)*P(2)>.75) & (P(1)+P(2)<15))
            [l,funcl] = archive_best_ga(Phen(i,:),ObjV(i),l,funcl,L);
        end
    end
else
    for i = 1:NIND
        P = Phen(i,:);
        if ((P(1)*P(2)>.75) & (P(1)+P(2)<15))
            [l,funcl] = archive_ga(Phen(i,:),ObjV(i),l,funcl,D_min,D_sim,L,gen);
        end
    end
end

if (switch_plot == 1)
    % Plot the contours of the objective function including the constraint boundaries
    subplot(211);
    plotKeane;
end

```

```

title('A Genetic Algorithm Search on Keane''s bump function'); hold on;
x1 = Phen(:,1); x2 = Phen(:,2);
plot(x1,x2,'c.');
```

```

% Track best individual and display convergence
subplot(212);
Best(gen+1) = min(ObjV);
plot(Best,'ro');xlabel('generation'); ylabel('f(x)');
text(0.5,0.95,['Best = ', num2str(Best(gen+1))],'Units','normalized');
```

```

end
```

```

% Start GA
% Generational loop
while gen < MAXGEN,

    % Assign fitness-value to entire population
    FitnV = ranking(ObjV,RFun);

    % Select individuals for breeding
    SelCh = select('sus', Chrom, FitnV, GGAP);

    % Recombine selected individuals (crossover)
    SelCh = recomb('xovmp',SelCh,XOV);

    % Perform mutation on offspring
    SelCh = mut(SelCh,MUTR);

    % Evaluate offspring, call objective function
    ObjVSel = objfun3(bs2rv(SelCh,FieldD),gen);

    % Reinsert offspring into current population
    [Chrom ObjV]=reins(Chrom,SelCh,1,1,ObjV,ObjVSel);
    Phen = bs2rv(Chrom,FieldD);
    ObjVal = objfun3(Phen,gen);

    % Increment generational counter
    gen = gen+1;

    % Plot Results if (switch_plot == 1)
    if (switch_plot == 1)
        subplot(211);
        x1 = Phen(:,1); x2 = Phen(:,2);
        plot(x1,x2,'c.');
```

```

        hold on;
        drawnow

        % Plot objective function reduction
        Best(gen+1) = min(ObjVal);
        subplot(212);
        plot(Best,'ro-'); xlabel('generation'); ylabel('min f(\bf x)');
        text(0.5,0.95,['Best = ', num2str(Best(gen+1))],'Units','normalized');
```

```

        drawnow
    end

    % Update archives
    if (switch_archive == 1)
        for i = 1:NIND
            P = Phen(i,:);
            if ((P(1)*P(2)>.75) & (P(1)+P(2)<15))
                [l,func1] = archive_best_ga(Phen(i,:),ObjVal(i),l,func1,L);
            end
        end
    else
        for i = 1:NIND
```

```

        P = Phen(i,:);
        if ((P(1)*P(2)>.75) & (P(1)+P(2)<15))
            [l,funcl] = archive_ga(Phen(i,:),ObjVal(i),l,funcl,D_min,D_sim,L,gen);
        end
    end
end
end
% End of GA

% Total run-time of simulation in seconds
elapsed_time = etime(clock,time);

% Complete plots
if (switch_plot == 1)
    subplot(212); title('Objective reduction'); drawnow;
    grid on;
    subplot(211);
    plot(squeeze(1(:,1)),squeeze(1(:,2)),'ro');
    plot(gopt(1), gopt(2),'kx'); drawnow;
end

% Output Optimal value of x
% disp('Results of GA');
% disp('-----');
% Phen = bs2rv(Chrom,FieldD);
[Y,I] = min(ObjVal);
% disp('Maximum value of f(x) is =');
fmax = -Y;

% l = 1';
% obj = -objfun3(l',gen+1)';

% Output results
% disp('The 10 best points are located at');
l = 1';
% disp('which have objective function values of');
obj = -funcl';
% disp('respectively.')
```

```

%-----
% BS2RV.m - Binary string to real vector
%
% This function decodes binary chromosomes into vectors of reals. The
% chromosomes are seen as the concatenation of binary strings of given
% length, and decoded into real numbers in a specified interval using
% either standard binary or Gray decoding.
%
% Syntax:      Phen = bs2rv(Chrom,FieldD)
%
% Input parameters:
%
%      Chrom    - Matrix containing the chromosomes of the current
%                 population. Each line corresponds to one
%                 individual's concatenated binary string
%                 representation. Leftmost bits are MSb and
%                 rightmost are LSb.
%
%      FieldD   - Matrix describing the length and how to decode
%                 each substring in the chromosome. It has the
%                 following structure:
%
%      [len;      (num)
%       lb;       (num)
%       ub;       (num)
%       code;     (0=binary   | 1=gray)
```

```

%           scale;      (0=arithmetic | 1=logarithmic)
%           lbin;      (0=excluded   | 1=included)
%           ubin;      (0=excluded   | 1=included)
%
%       where
%       len  - row vector containing the length of
%             each substring in Chrom. sum(len)
%             should equal the individual length.
%       lb,
%       ub   - Lower and upper bounds for each
%             variable.
%       code - binary row vector indicating how each
%             substring is to be decoded.
%       scale - binary row vector indicating where to
%             use arithmetic and/or logarithmic
%             scaling.
%       lbin,
%       ubin - binary row vectors indicating whether
%             or not to include each bound in the
%             representation range
%
% Output parameter:
%
%           Phen      - Real matrix containing the population phenotypes.
%
% Author: Carlos Fonseca, Updated: Andrew Chipperfield
% Date: 08/06/93,      Date: 26-Jan-94

function Phen = bs2rv(Chrom,FieldD)

% Identify the population size (Nind)
% and the chromosome length (Lind)
[Nind,Lind] = size(Chrom);

% Identify the number of decision variables (Nvar)
[seven,Nvar] = size(FieldD);

if seven ~= 7
    error('FieldD must have 7 rows.');
```

end

```

% Get substring properties
len = FieldD(1,:);
lb = FieldD(2,:);
ub = FieldD(3,:);
code = ~(~FieldD(4,:));
scale = ~(~FieldD(5,:));
lin = ~(~FieldD(6,:));
uin = ~(~FieldD(7,:));

% Check substring properties for consistency
if sum(len) ~= Lind,
    error('Data in FieldD must agree with chromosome length');
```

end

```

if ~all(lb(scale).*ub(scale)>0)
    error('Log-scaled variables must not include 0 in their range');
```

end

```

% Decode chromosomes
Phen = zeros(Nind,Nvar);

lf = cumsum(len);
li = cumsum([1 len]);
```

```

Prec = .5 .^ len;

logsgn = sign(lb(scale));
lb(scale) = log( abs(lb(scale)) );
ub(scale) = log( abs(ub(scale)) );
delta = ub - lb;

Prec = .5 .^ len;
num = (~lin) .* Prec;
den = (lin + uin - 1) .* Prec;

for i = 1:Nvar,
    idx = li(i):lf(i);
    if code(i) % Gray decoding
        Chrom(:,idx)=rem(cumsum(Chrom(:,idx)'),'2);
    end
    Phn(:,i) = Chrom(:,idx) * [ (.5).^(1:len(i))' ];
    Phn(:,i) = lb(i) + delta(i) * (Phn(:,i) + num(i)) ./ (1 - den(i));
end

expand = ones(Nind,1);
if any(scale)
    Phn(:,scale) = logsgn(expand,:) .* exp(Phn(:,scale));
end
%-----

% XOVMP.m                Multi-point crossover
%
%      Syntax: NewChrom = xovmp(OldChrom, Px, Npt, Rs)
%
%      This function takes a matrix OldChrom containing the binary
%      representation of the individuals in the current population,
%      applies crossover to consecutive pairs of individuals with
%      probability Px and returns the resulting population.
%
%      Npt indicates how many crossover points to use (1 or 2, zero
%      indicates shuffle crossover).
%      Rs indicates whether or not to force the production of
%      offspring different from their parents.
%
% Author: Carlos Fonseca,   Updated: Andrew Chipperfield
% Date: 28/09/93,          Date: 27-Jan-94

function NewChrom = xovmp(OldChrom, Px, Npt, Rs);

% Identify the population size (Nind) and the chromosome length (Lind)
[Nind,Lind] = size(OldChrom);

if Lind < 2, NewChrom = OldChrom; return; end

if nargin < 4, Rs = 0; end if nargin < 3, Npt = 0; Rs = 0; end
if nargin < 2, Px = 0.7; Npt = 0; Rs = 0; end if isnan(Px), Px = 0.7; end if
isnan(Npt), Npt = 0; end
if isnan(Rs), Rs = 0; end
if isempty(Px), Px = 0.7; end
if isempty(Npt), Npt = 0; end
if isempty(Rs), Rs = 0; end

Xops = floor(Nind/2); DoCross = rand(Xops,1) < Px; odd = 1:2:Nind-1; even = 2:2:Nind;

% Compute the effective length of each chromosome pair
Mask = ~Rs | (OldChrom(odd, :) ~= OldChrom(even, :)); Mask = cumsum(Mask')';

% Compute cross sites for each pair of individuals, according to their

```

```

% effective length and Px (two equal cross sites mean no crossover)
xsites(:, 1) = Mask(:, Lind); if Npt >= 2,
    xsites(:, 1) = ceil(xsites(:, 1) .* rand(Xops, 1));
end xsites(:,2) = rem(xsites + ceil((Mask(:, Lind)-1) .* rand(Xops, 1)) ...
    .* DoCross - 1 , Mask(:, Lind) )+1;

% Express cross sites in terms of a 0-1 mask
Mask = (xsites(:,ones(1,Lind)) < Mask) == ...
    (xsites(:,2*ones(1,Lind)) < Mask);

if ~Npt,
    shuff = rand(Lind,Xops);
    [ans,shuff] = sort(shuff);
    for i=1:Xops
        OldChrom(odd(i),:)=OldChrom(odd(i),shuff(:,i));
        OldChrom(even(i),:)=OldChrom(even(i),shuff(:,i));
    end
end

% Perform crossover
NewChrom(odd,:) = (OldChrom(odd,:).* Mask) + (OldChrom(even,:).*(~Mask));
NewChrom(even,:) = (OldChrom(odd,:).*(~Mask)) + (OldChrom(even,:).*Mask);

% If the number of individuals is odd, the last individual cannot be mated
% but must be included in the new population
if rem(Nind,2),
    NewChrom(Nind,:)=OldChrom(Nind,:);
end

if ~Npt,
    [ans,unshuff] = sort(shuff);
    for i=1:Xops
        NewChrom(odd(i),:)=NewChrom(odd(i),unshuff(:,i));
        NewChrom(even(i),:)=NewChrom(even(i),unshuff(:,i));
    end
end

%-----

% MUT.m
%
% This function takes the representation of the current population,
% mutates each element with given probability and returns the resulting
% population.
%
% Syntax:   NewChrom = mut(OldChrom,Pm,BaseV)
%
% Input parameters:
%
%   OldChrom - A matrix containing the chromosomes of the
%               current population. Each row corresponds to
%               an individuals string representation.
%
%   Pm       - Mutation probability (scalar). Default value
%               of Pm = 0.7/Lind, where Lind is the chromosome
%               length is assumed if omitted.
%
%   BaseV    - Optional row vector of the same length as the
%               chromosome structure defining the base of the
%               individual elements of the chromosome. Binary
%               representation is assumed if omitted.
%
% Output parameter:
%
%   NewChrom - A Matrix containing a mutated version of

```

```

%           OldChrom.
%

% Author: Andrew Chipperfield
% Date: 25-Jan-94

function NewChrom = mut(OldChrom,Pm,BaseV)

% get population size (Nind) and chromosome length (Lind)
[Nind, Lind] = size(OldChrom) ;

% check input parameters
if nargin < 2, Pm = 0.7/Lind ; end if isnan(Pm), Pm = 0.7/Lind; end

if (nargin < 3), BaseV = crtbase(Lind); end
if (isnan(BaseV)), BaseV = crtbase(Lind); end if (isempty(BaseV)), BaseV = crtbase(Lind); end

if (nargin == 3) & (Lind ~= length(BaseV))
    error('OldChrom and BaseV are incompatible'), end

% create mutation mask matrix
BaseM = BaseV(ones(Nind,1),:) ;

% perform mutation on chromosome structure
NewChrom = rem(OldChrom+(rand(Nind,Lind)<Pm).*ceil(rand(Nind,Lind).*(BaseM-1)),BaseM);

%-----

% REINS.M      (RE-INsersion of offspring in population replacing parents)
%
% This function reinserts offspring in the population.
%
% Syntax: [Chrom, ObjVCh] = reins(Chrom, SelCh, SUBPOP, InsOpt, ObjVCh, ObjVSEL)
%
% Input parameters:
%   Chrom      - Matrix containing the individuals (parents) of the current
%                population. Each row corresponds to one individual.
%   SelCh      - Matrix containing the offspring of the current
%                population. Each row corresponds to one individual.
%   SUBPOP     - (optional) Number of subpopulations
%                if omitted or NaN, 1 subpopulation is assumed
%   InsOpt     - (optional) Vector containing the insertion method parameters
%                ExOpt(1): Select - number indicating kind of insertion
%                    0 - uniform insertion
%                    1 - fitness-based insertion
%                if omitted or NaN, 0 is assumed
%                ExOpt(2): INSR - Rate of offspring to be inserted per
%                subpopulation (% of subpopulation)
%                if omitted or NaN, 1.0 (100%) is assumed
%   ObjVCh     - (optional) Column vector containing the objective values
%                of the individuals (parents - Chrom) in the current
%                population, needed for fitness-based insertion
%                saves recalculation of objective values for population
%   ObjVSEL    - (optional) Column vector containing the objective values
%                of the offspring (SelCh) in the current population, needed for
%                partial insertion of offspring,
%                saves recalculation of objective values for population
%
% Output parameters:
%   Chrom      - Matrix containing the individuals of the current
%                population after reinsertion.
%   ObjVCh     - if ObjVCh and ObjVSEL are input parameter, than column
%                vector containing the objective values of the individuals
%                of the current generation after reinsertion.

```

```

% Author:      Hartmut Pohlheim
% History:    10.03.94    file created
%            19.03.94    parameter checking improved

function [Chrom, ObjVCh] = reins(Chrom, SelCh, SUBPOP, InsOpt, ObjVCh, ObjVSel);

% Check parameter consistency
if nargin < 2, error('Not enough input parameter'); end
if (nargout == 2 & nargin < 6), error('Input parameter missing: ObjVCh and/or ObjVSel'); end

[NindP, NvarP] = size(Chrom);
[NindO, NvarO] = size(SelCh);

if nargin == 2, SUBPOP = 1; end
if nargin > 2,
    if isempty(SUBPOP), SUBPOP = 1;
    elseif isnan(SUBPOP), SUBPOP = 1;
    elseif length(SUBPOP) ~= 1, error('SUBPOP must be a scalar'); end
end

if (NindP/SUBPOP) ~= fix(NindP/SUBPOP), error('Chrom and SUBPOP disagree'); end
if (NindO/SUBPOP) ~= fix(NindO/SUBPOP), error('SelCh and SUBPOP disagree'); end
NIND = NindP/SUBPOP; % Compute number of individuals per subpopulation
NSEL = NindO/SUBPOP; % Compute number of offspring per subpopulation

IsObjVCh = 0; IsObjVSel = 0;
if nargin > 4,
    [m0, n0] = size(ObjVCh);
    if n0 ~= 1, error('ObjVCh must be a column vector'); end
    if NindP ~= m0, error('Chrom and ObjVCh disagree'); end
    IsObjVCh = 1;
end
if nargin > 5,
    [m0, n0] = size(ObjVSel);
    if n0 ~= 1, error('ObjVSel must be a column vector'); end
    if NindO ~= m0, error('SelCh and ObjVSel disagree'); end
    IsObjVSel = 1;
end

if nargin < 4, INSR = 1.0; Select = 0; end
if nargin >= 4,
    if isempty(InsOpt), INSR = 1.0; Select = 0;
    elseif isnan(InsOpt), INSR = 1.0; Select = 0;
    else
        INSR = NaN; Select = NaN;
        if (length(InsOpt) > 2), error('Parameter InsOpt too long'); end
        if (length(InsOpt) >= 1), Select = InsOpt(1); end
        if (length(InsOpt) >= 2), INSR = InsOpt(2); end
        if isnan(Select), Select = 0; end
        if isnan(INSR), INSR = 1.0; end
    end
end

if (INSR < 0 | INSR > 1), error('Parameter for insertion rate must be a scalar in [0, 1]'); end
if (INSR < 1 & IsObjVSel ~= 1), error('For selection of offspring ObjVSel is needed'); end
if (Select ~= 0 & Select ~= 1), error('Parameter for selection method must be 0 or 1'); end
if (Select == 1 & IsObjVCh == 0), error('ObjVCh for fitness-based exchange needed'); end

if INSR == 0, return; end
NIns = min(max(floor(INSR*NSEL+.5),1),NIND); % Number of offspring to insert

% perform insertion for each subpopulation
for irun = 1:SUBPOP,
    % Calculate positions in old subpopulation, where offspring are inserted

```

```

    if Select == 1, % fitness-based reinsertion
        [Dummy, ChIx] = sort(-ObjVCh((irun-1)*NIND+1:irun*NIND));
    else % uniform reinsertion
        [Dummy, ChIx] = sort(rand(NIND,1));
    end
    PopIx = ChIx((1:NIns)')+ (irun-1)*NIND;
% Calculate position of Nins-% best offspring
    if (NIns < NSEL), % select best offspring
        [Dummy,OffIx] = sort(ObjVSel((irun-1)*NSEL+1:irun*NSEL));
    else
        OffIx = (1:NIns)';
    end
    SelIx = OffIx((1:NIns)'+(irun-1)*NSEL);
% Insert offspring in subpopulation -> new subpopulation
    Chrom(PopIx,:) = SelCh(SelIx,:);
    if (IsObjVCh == 1 & IsObjVSel == 1), ObjVCh(PopIx) = ObjVSel(SelIx); end
end

% End of function
%-----

% RANKING.M (RANK-based fitness assignment)
%
% This function performs ranking of individuals.
%
% Syntax: FitnV = ranking(ObjV, RFun, SUBPOP)
%
% This function ranks individuals represented by their associated
% cost, to be *minimized*, and returns a column vector FitnV
% containing the corresponding individual fitnesses. For multiple
% subpopulations the ranking is performed separately for each
% subpopulation.
%
% Input parameters:
% ObjV - Column vector containing the objective values of the
% individuals in the current population (cost values).
% RFun - (optional) If RFun is a scalar in [1, 2] linear ranking is
% assumed and the scalar indicates the selective pressure.
% If RFun is a 2 element vector:
% RFun(1): SP - scalar indicating the selective pressure
% RFun(2): RM - ranking method
% RM = 0: linear ranking
% RM = 1: non-linear ranking
% If RFun is a vector with length(Rfun) > 2 it contains
% the fitness to be assigned to each rank. It should have
% the same length as ObjV. Usually RFun is monotonously
% increasing.
% If RFun is omitted or NaN, linear ranking
% and a selective pressure of 2 are assumed.
% SUBPOP - (optional) Number of subpopulations
% if omitted or NaN, 1 subpopulation is assumed
%
% Output parameters:
% FitnV - Column vector containing the fitness values of the
% individuals in the current population.
%
% Author: Hartmut Pohlheim (Carlos Fonseca)
% History: 01.03.94 non-linear ranking
% 10.03.94 multiple populations

function FitnV = ranking(ObjV, RFun, SUBPOP);
%NonLin=1;
% Identify the vector size (Nind)
[Nind,ans] = size(ObjV);

```

```

if nargin < 2, RFun = []; end
if nargin > 1, if isnan(RFun), RFun = []; end, end
if prod(size(RFun)) == 2,
    if RFun(2) == 1, NonLin = 1;
    elseif RFun(2) == 0, NonLin = 0;
    else error('Parameter for ranking method must be 0 or 1'); end
    RFun = RFun(1);
    if isnan(RFun), RFun = 2; end
elseif prod(size(RFun)) > 2,
    if prod(size(RFun)) ~= Nind, error('ObjV and RFun disagree'); end
end

if nargin < 3, SUBPOP = 1; end
if nargin > 2,
    if isempty(SUBPOP), SUBPOP = 1;
    elseif isnan(SUBPOP), SUBPOP = 1;
    elseif length(SUBPOP) ~= 1, error('SUBPOP must be a scalar'); end
end

if (Nind/SUBPOP) ~= fix(Nind/SUBPOP), error('ObjV and SUBPOP disagree'); end
Nind = Nind/SUBPOP; % Compute number of individuals per subpopulation

% Check ranking function and use default values if necessary
if isempty(RFun),
    % linear ranking with selective pressure 2
    RFun = 2*[0:Nind-1]/(Nind-1);
elseif prod(size(RFun)) == 1
    if NonLin == 1,
        % non-linear ranking
        if RFun(1) < 1, error('Selective pressure must be greater than 1');
        elseif RFun(1) > Nind-2, error('Selective pressure too big'); end
        Root1 = roots([RFun(1)-Nind [RFun(1)*ones(1,Nind-1)]]);
        RFun = (abs(Root1(1)) * ones(Nind,1)) .^ [(0:Nind-1)'];
        RFun = RFun / sum(RFun) * Nind;
    else
        % linear ranking with SP between 1 and 2
        if (RFun(1) < 1 | RFun(1) > 2),
            error('Selective pressure for linear ranking must be between 1 and 2');
        end
        RFun = 2-RFun + 2*(RFun-1)*[0:Nind-1]/(Nind-1);
    end
end;

FitnV = [];

% loop over all subpopulations
for irun = 1:SUBPOP,
    % Copy objective values of actual subpopulation
    ObjVSub = ObjV((irun-1)*Nind+1:irun*Nind);
    % Sort does not handle NaN values as required. So, find those...
    NaNix = isnan(ObjVSub);
    Validix = find(~NaNix);
    % ... and sort only numeric values (smaller is better).
    [ans,ix] = sort(-ObjVSub(Validix));

    % Now build indexing vector assuming NaN are worse than numbers,
    % (including Inf!)...
    ix = [find(NaNix) ; Validix(ix)];
    % ... and obtain a sorted version of ObjV
    Sorted = ObjVSub(ix);

    % Assign fitness according to RFun.
    i = 1;
    FitnVSub = zeros(Nind,1);

```

```

        for j = [find(Sorted(1:Nind-1) ~= Sorted(2:Nind)); Nind]',
            FitnVSub(i:j) = sum(RFun(i:j)) * ones(j-i+1,1) / (j-i+1);
            i =j+1;
        end

% Finally, return unsorted vector.
[ans,uix] = sort(ix);
FitnVSub = FitnVSub(uix);

% Add FitnVSub to FitnV
FitnV = [FitnV; FitnVSub];
end

% End of function
%-----
% SELECT.M          (universal SELECTION)
%
% This function performs universal selection. The function handles
% multiple populations and calls the low level selection function
% for the actual selection process.

%
% Syntax: SelCh = select(SEL_F, Chrom, FitnV, GGAP, SUBPOP)
%
% Input parameters:
%   SEL_F      - Name of the selection function
%   Chrom      - Matrix containing the individuals (parents) of the current
%               population. Each row corresponds to one individual.
%   FitnV      - Column vector containing the fitness values of the
%               individuals in the population.
%   GGAP       - (optional) Rate of individuals to be selected
%               if omitted 1.0 is assumed
%   SUBPOP     - (optional) Number of subpopulations
%               if omitted 1 subpopulation is assumed
%
% Output parameters:
%   SelCh      - Matrix containing the selected individuals.

% Author:      Hartmut Pohlheim
% History:     10.03.94      file created

function SelCh = select(SEL_F, Chrom, FitnV, GGAP, SUBPOP);

% Check parameter consistency
if nargin < 3, error('Not enough input parameter'); end

% Identify the population size (Nind)
[NindCh,Nvar] = size(Chrom);
[NindF,VarF] = size(FitnV);
if NindCh ~= NindF, error('Chrom and FitnV disagree'); end
if VarF ~= 1, error('FitnV must be a column vector'); end

if nargin < 5, SUBPOP = 1; end
if nargin > 4,
    if isempty(SUBPOP), SUBPOP = 1;
    elseif isnan(SUBPOP), SUBPOP = 1;
    elseif length(SUBPOP) ~= 1, error('SUBPOP must be a scalar'); end
end

if (NindCh/SUBPOP) ~= fix(NindCh/SUBPOP), error('Chrom and SUBPOP disagree'); end
Nind = NindCh/SUBPOP; % Compute number of individuals per subpopulation

if nargin < 4, GGAP = 1; end
if nargin > 3,

```

```

        if isempty(GGAP), GGAP = 1;
        elseif isnan(GGAP), GGAP = 1;
        elseif length(GGAP) ~= 1, error('GGAP must be a scalar');
        elseif (GGAP < 0), error('GGAP must be a scalar bigger than 0'); end
    end

% Compute number of new individuals (to select)
    NSel=max(floor(Nind*GGAP+.5),2);

% Select individuals from population
    SelCh = [];
    for irun = 1:SUBPOP,
        FitnVSub = FitnV((irun-1)*Nind+1:irun*Nind);
        ChrIx=feval(SEL_F, FitnVSub, NSel)+(irun-1)*Nind;
        SelCh=[SelCh; Chrom(ChrIx,:)];
    end

% End of function
%-----
% RECOMBIN.M      (RECOMBINation high-level function)
%
% This function performs recombination between pairs of individuals
% and returns the new individuals after mating. The function handles
% multiple populations and calls the low-level recombination function
% for the actual recombination process.
%
% Syntax: NewChrom = recomb(REC_F, OldChrom, RecOpt, SUBPOP)
%
% Input parameters:
%   REC_F      - String containing the name of the recombination or
%               crossover function
%   Chrom      - Matrix containing the chromosomes of the old
%               population. Each line corresponds to one individual
%   RecOpt     - (optional) Scalar containing the probability of
%               recombination/crossover occurring between pairs
%               of individuals.
%               if omitted or NaN, 1 is assumed
%   SUBPOP     - (optional) Number of subpopulations
%               if omitted or NaN, 1 subpopulation is assumed
%
% Output parameter:
%   NewChrom   - Matrix containing the chromosomes of the population
%               after recombination in the same format as OldChrom.
%
% Author:      Hartmut Pohlheim
% History:     18.03.94      file created

function NewChrom = recomb(REC_F, Chrom, RecOpt, SUBPOP);

% Check parameter consistency
    if nargin < 2, error('Not enough input parameter'); end

% Identify the population size (Nind)
    [Nind,Nvar] = size(Chrom);

    if nargin < 4, SUBPOP = 1; end
    if nargin > 3,
        if isempty(SUBPOP), SUBPOP = 1;
        elseif isnan(SUBPOP), SUBPOP = 1;
        elseif length(SUBPOP) ~= 1, error('SUBPOP must be a scalar'); end
    end
end

```

```

if (Nind/SUBPOP) ~= fix(Nind/SUBPOP), error('Chrom and SUBPOP disagree'); end
Nind = Nind/SUBPOP; % Compute number of individuals per subpopulation

if nargin < 3, RecOpt = 0.7; end
if nargin > 2,
    if isempty(RecOpt), RecOpt = 0.7;
    elseif isnan(RecOpt), RecOpt = 0.7;
    elseif length(RecOpt) ~= 1, error('RecOpt must be a scalar');
    elseif (RecOpt < 0 | RecOpt > 1), error('RecOpt must be a scalar in [0, 1]'); end
end

% Select individuals of one subpopulation and call low level function
NewChrom = [];
for irun = 1:SUBPOP,
    ChromSub = Chrom((irun-1)*Nind+1:irun*Nind,:);
    NewChromSub = feval(REC_F, ChromSub, RecOpt);
    NewChrom=[NewChrom; NewChromSub];
end

% End of function
%-----

```

F Objective Function code

```

% OBJFUN3.M      (OBJECTive function for Keane's Bump FUNCTION 3)
%
% This function implements the Keane's BUMP function 3.
%
% Syntax:  ObjVal = objfun3(Chrom,switch)
%
% Input parameters:
%   Chrom    - Matrix containing the chromosomes of the current
%              population. Each row corresponds to one individual's
%              string representation.
%              if Chrom == [], then speziell values will be returned
%   switch    - if Chrom == [] and
%              switch == 1 (or []) return boundaries
%              switch == 2 return title
%              switch == 3 return value of global minimum
%   M        - Generation Number
%
% Output parameters:
%   ObjVal    - Column vector containing the objective values of the
%              individuals in the current population.
%              if called with Chrom == [], then ObjVal contains
%              switch == 1, matrix with the boundaries of the function
%              switch == 2, text for the title of the graphic output
%              switch == 3, value of global minimum
%
% The optimal value of Keane's BUMP function is 0.36497974409160
% which occurs at [1.6009000000000000  0.46848647635705]

% Author:      Vincent Tan

function ObjVal = objfun3(Chrom,M,switc);

% Dimension of objective function
%Dim = 20;
Dim = 2;

```

```

% Compute population parameters
[Nind,Nvar] = size(Chrom);

% Check size of Chrom and do the appropriate thing
% if Chrom is [], then define size of boundary-matrix and values
if Nind == 0
    % return text of title for graphic output
    if switc == 2
        ObjVal = ['KEANES Bump function 3-' int2str(Dim)];
    % return value of global minimum
    elseif switc == 3
        ObjVal = -.36;
    % define size of boundary-matrix and values
    else
        % lower and upper bound, identical for all n variables
        ObjVal = [0; 10];
        ObjVal = ObjVal(1:2,ones(Dim,1));
    end
% if Dim variables, compute values of function
elseif Nvar == Dim
    % function 3,
    % n = Dim, 0 <= xi <= 10
    % global minimum at
    x1 = Chrom(:,1);
    x2 = Chrom(:,2);
    ObjVal = -abs(((cos(x1)).^4+(cos(x2)).^4-2.*((cos(x1)).*(cos(x2))).^2)./sqrt(x1.^2+2.*x2.^2));
    %ObjVal = 1./((abs(((cos(x1)).^4+(cos(x2)).^4-2.*((cos(x1)).*(cos(x2))).^2)./sqrt(x1.^2+2.*x2.^2)));
    k = 3; w = 5e3;
    for i = 1:length(x1)
        Penalty(i) = (M^k)*(w*(max([0 0.75-x1(i)*x2(i)]))^2+(max([0 x1(i)+x2(i)-15]))^2);
    end
    ObjVal = ObjVal + Penalty';
    % ObjVal = diag(Chrom * Chrom'); % both lines produce the same
% otherwise error, wrong format of Chrom
else
    error('size of matrix Chrom is not correct for function evaluation');
end

% End of function

```