4M17

Coursework Assignment 3: Minimisation on the Bird Function

Coursework Candidate Number: eng-188do
January 12, 2015

Summary

This report looks into three optimisation techniques: Evolutionary Strategies (ES), Tabu Search (TS) and Particle Swarm Optimisation (PSO). The first two are looked into in more detail, as they are required for the assignment. The control parameters for these optimisation techniques are varied and the result that this has on their performance is analysed. The code to run these three routines has been written by the author of this report in Java 8 and is included in an Appendix. Overall the TS and ES algorithms worked well on finding one of the two global minima of the Bird Function being tested, although the TS was often better at honing in on a minimum. The PSO algorithm would often get stuck in local minima, but this may be because not as long was spent on tuning its settings.

Contents

Acronyms

TS: Tabu (or Taboo) Search,

ES: Evolutionary Strategy,

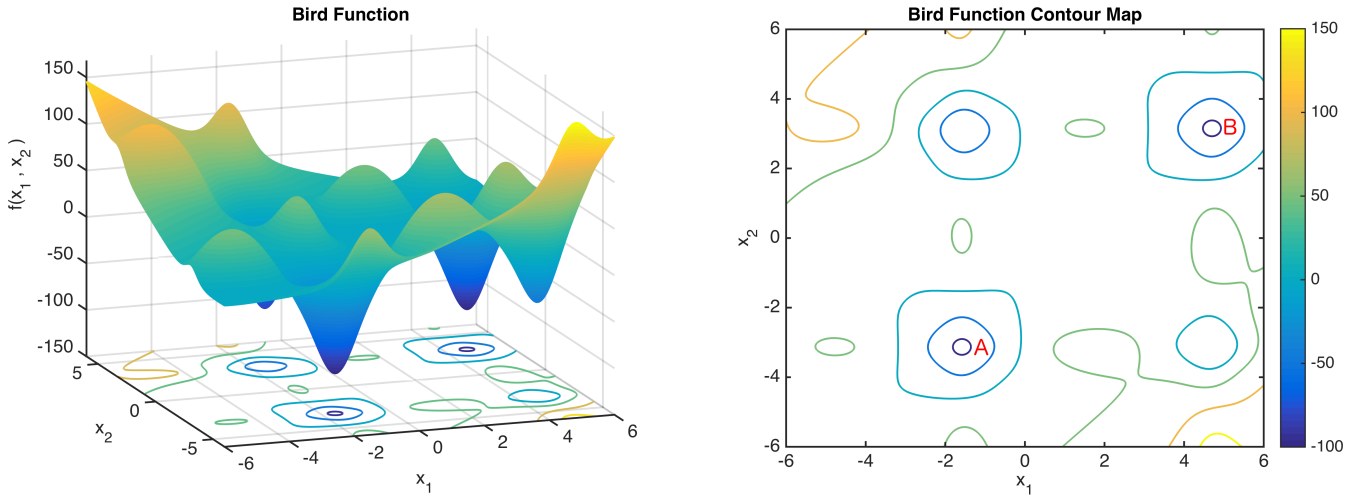
PSO: Particle Swarm Optimisation,

JDK: Java Development Kit.

1 Introduction

This report details an investigation into solving the Bird Function (see Equation 1) [1]. This is plotted over the range of interest in Figure 1. In Figure 1b the two minima have been labelled, with A having coordinates (-1.6, -3.1) and B having coordinates of (4.7, 3.2). These are the two global minima that the algorithms discussed later are trying to find and shall be referred to throughout this text as points A and B. There are also several local minima, apparent from Figure 1a. Note that although defined in the question as x_1 and x_2 the two parameters are labelled in the code as x and y for simplicity.

$$\begin{aligned} &\text{Minimise} && f(\boldsymbol{x}) = \sin(x_1)\exp\Big[(1-\cos(x_2))^2\Big] + \cos(x_2)\exp\Big[(1-\sin(x_1))^2\Big] + (x_1 - x_2)^2 \qquad (1)\\ &\text{subject to} && -6 \leq x_1 \leq 6 \\ & && -6 \leq x_2 \leq 6 \end{aligned}$$



(a) Mesh Image **(b)** Contour Map

Figure 1 – Bird Function over area of interest

The two algorithms used to investigate this problem were the Evolutionary Strategy (ES) and Tabu Search (TS). ES was chosen as it was designed for continuous problems like the Bird Function, in contrast to Genetic Algorithms, which are designed for discrete problems and so almost always perform worse [2, p.GA2]. TS was chosen for the second algorithm, as it was very different. Although not required Particle Swarm Optimisation (PSO) was also analysed. Details of this algorithm are given later but it was chosen, as it offered an interesting mixture of features from the ES and TS algorithms: like ES it consisted of multiple agents working together, but these agents would move in steps through search space and use memory in a similar manner to TS.

This report will describe the code used to analyse the algorithms, then go through the algorithms one by one before bringing them together at the end for a comparison.

2 Code

To analyse the three algorithms, code was written by the author of this report to perform the algorithm steps. It was felt that writing new code was a good idea, as it meant that there was greater understanding of exactly what the program was doing and there could be more explicit variation of the algorithm's control parameters built in.

The code is included in Appendix B. It was written in Java 8 (using Oracle's JDK 1.8), however, the code should be back compatible with Java 7. The program is built in an object oriented way, with more details about the structure to be found in Appendix B.2. As Java was used, some functions

that are built into MATLAB, such as matrix multiplication, had to be implemented from scratch. This means that the code base is quite large. Also, there are interactions between several classes, which means that the code is easier to read on a computer. Therefore, when handing in this report the Eclipse Project (without author name references) will be uploaded to:

`https://github.com/eng-188do/4M17_assignment3`

All the algorithms looked at can include convergence criteria, which can decide when the minimum has been calculated. However, as the number of function evaluations was limited to 1000, a relatively low number, convergence tests were not implemented and instead the algorithm would end when it had used up its quota of function evaluations. The program did not implement an archive, again as the number of function evaluations was so small, but instead stored all the points visited. This meant that archives could be simulated for a run during post processing. The code to do this was written in MATLAB and is also given in the appendix.

3 The Evolutionary Strategy Algorithm

3.1 Algorithm Details

The full details of the Algorithm can be found in [2] and [4, §2], but this section describes some of the finer details.

The initialisation of the population was done randomly throughout the search space, as described on the assignment sheet [1]. However, the strategy parameters were initialised deterministically. For this Bëck suggests a choice of $\sigma_i(0) = \Delta x_i/\sqrt(n)$, where $\sigma_i(0)$ is the initial standard deviation for the i^{th} control parameters, Δx_i is the expected distance between the starting point and the minimum and n is the number of control parameters [4, §2.1.5]. Taking Δx_i as 3 for both control parameters, this suggests using an initial standard deviation of 2.12. Therefore, the initial variances were chosen to be 1, as if they were too small the evolutionary process scales them to the correct range [4, §2.1.5]. The initial rotation angle was chosen to be zero, this meant that the mutation process's probability distribution's contours were initially concentric circles around the current member's point. It was hoped that as the algorithm progressed it would adapt the angles to more appropriate values.

The notation for the mutation is the same as that used in the lecture notes, in particular see equations 3 and 4 in notes [2, p.ES3]. These equations are repeated here as Equations 2 and 3:

$$\sigma_i' = \sigma_i \exp(\tau' \times \mathcal{N}_0 + \tau \times \mathcal{N}_i) \tag{2}$$

$$\alpha_{ij}' = \alpha_{ij} + \beta \times \mathcal{N}_{ij} \tag{3}$$

The $\mathcal{N}$'s are normal distributed variables (see [2, p.ES3] for further details). The τ, τ' and β are algorithm control parameters that are allowed to vary in the code, and are briefly explored in the investigations. The recommended values (using equation 5 from the notes) are 0.0595, 0.500 and 0.0873 respectively.

The recombination strategy used was set the same throughout all the experiments. Discrete recombination was used on the control parameters and intermediate recombination on the strategy parameters, as this was suggested in the notes [2, p.ES5].

3.2 Search Patterns Followed By Evolutionary Strategies

Figure 2 shows the population progression when starting with a random seed of -561489423. For this run: the algorithm control parameters were set as recommended in the previous subsection, the initial variance was set to one and a (15, 75)-selection was used. The function evaluation quota allowed the surviving population for 13 generations to be calculated (taking $15 + 15 \times 5 \times 13 = 990$ function calls). As one can see from Figure 2a the members are initially distributed all over the

space. Quickly, by the first generation, they are already clustered around local minimum (Figure 2b). From generations 2-6 the population has a presence around the three best minima in the space (including the two global minima), and in generation 4 (Figure 2d) they are mostly gathered around the worst of the 3 (i.e. the local minima). By Generation 7 the presence around minima B has disappeared even though the population at the local minima to the left of point B has survived. The thirteenth population, shown in Figure 2h, is firmly clustered around minimum A and the algorithm has converged on one of the two global minima, so is a success.

Figure 3 shows the average and lowest value of all the population members for each given generation. As one would expect the average objective function drops as the generation number increases. The best value (shown by the orange line) is always less than the average (note the different scales). As the scheme being used is not elitist, the minimum can go up, for example from generation 4 to generation 7. This corroborates with Figure 2, where one can see in generation 4 there is a member very close to minimum B but in generation 7 the points are all further away from either of the global minima.

Figure 4 shows the contents of a Best 5 Dissimilar Solutions Archive when run with D_{sim}=0.05 and D_{min}=0.4. D_{min} is perhaps too small, as 4 of the archive points are around minimum A. The importance of using an archive is shown by this Figure, as the record of minimum B is kept in the archive, which would be lost if only the best point was extracted. Different archive settings are not explored in this report, as for a 1000 function evaluations one could just store the whole lot. However, the code allows easy modification of the archive parameters if needed.

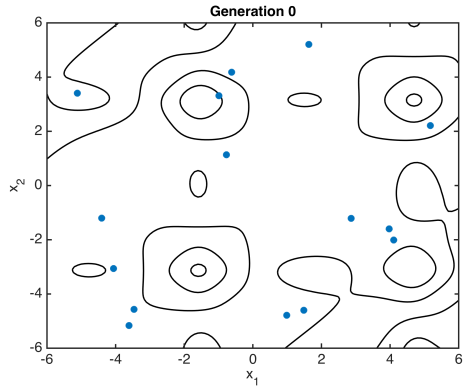
It would be interesting to try expanding ES by having two different tribes in the population to try to find the two different global minima. These could be penalised for being too close to each other through a modification to the objective function. This was not tried due to time and also its scalability - it may be useful for problems where you know you have two equal valued minima, but you would not know this to start off with when approaching a new problem.

Figure 5 shows the number of times each of the minima (or a point nearby) were found when the ES algorithm was run 1000 times with the same settings but with different random seeds. The average minimum value found was -106.2587. As one can see from the Figure, the algorithm does well generally finding points around the minimum, and doesn't favour a particular minimum over another - finding a point at or very near minimum A 497 times and points around minimum B 496 times, with the remaining 7 runs finding point near a local minimum. One can conclude from this that when using ESs it is important to run them multiple times, as they can find other minima and each time they do not necessarily find the minimum exactly (or at least to machine precision), due to their inherently random nature.

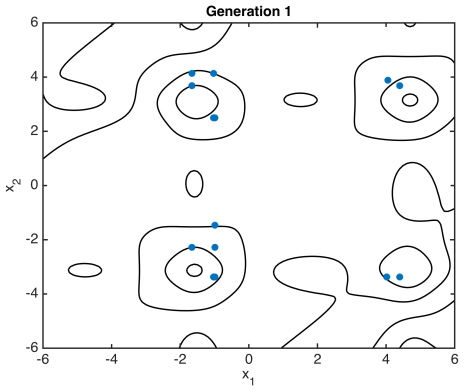
3.3 Experiments on using an elitist scheme

Figure 6 shows the difference between different selection schemes. Figure 6a shows the same thing as Figure 3 except the data has been averaged over 1000 runs using different random seeds. This means that while you would expect to see the minimum going up in a non-elitist scheme it does not as the bumps have been smoothed out by the averaging. Compared to Figure 6b little difference is seen, however, the elitist scheme's initial curve is steeper and flattens out at an earlier generation - suggesting that it is therefore a faster and better algorithm to use for this problem. However, for sake of comparison with the previous section the non-elitist algorithm will be used in the next sections. Also the elitist scheme was avoided as using elitist schemes can cause the algorithm to get stuck in local valleys or minima [3, p.164].

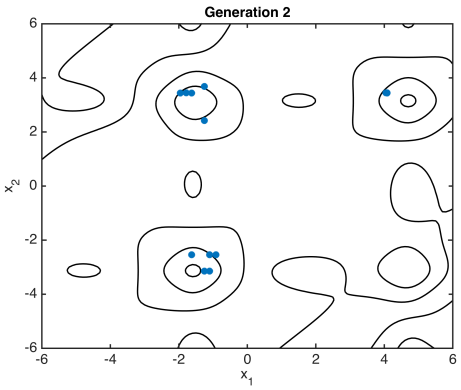
The elitist scheme's behaviour is also shown when analysing individual one off cases. Using the same random seed, of -561489423, as for the previous section, Figure 7 shows how the algorithm progresses differently. Initially Figure 7a shows how the four main minima are represented for longer by the population, however, by generation 4 (Figure 7b) the population is all grouped around minimum A, whereas it took till generation 9 in the non-elitist scheme.



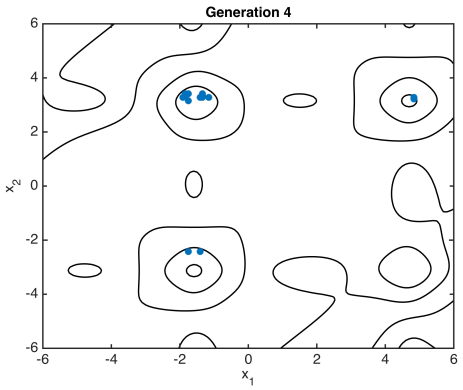
(a) Generation 0



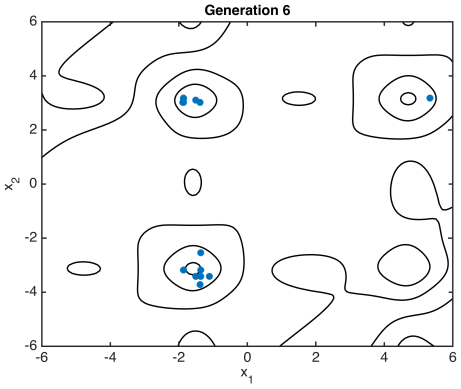
(b) Generation 1



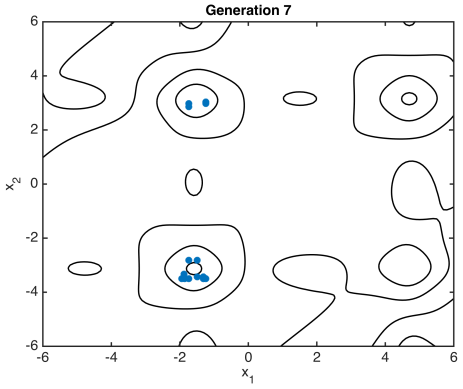
(c) Generation 2



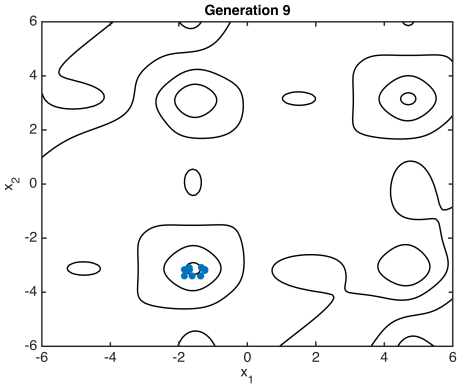
(d) Generation 4



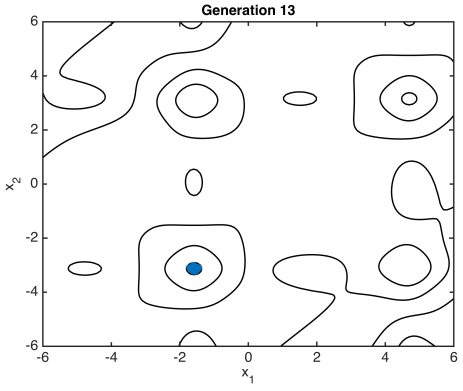
(e) Generation 6



(f) Generation 7



(g) Generation 9



(h) Generation 13

Figure 2 – How the ES algorithm progresses (Blue dots show surviving members of population).

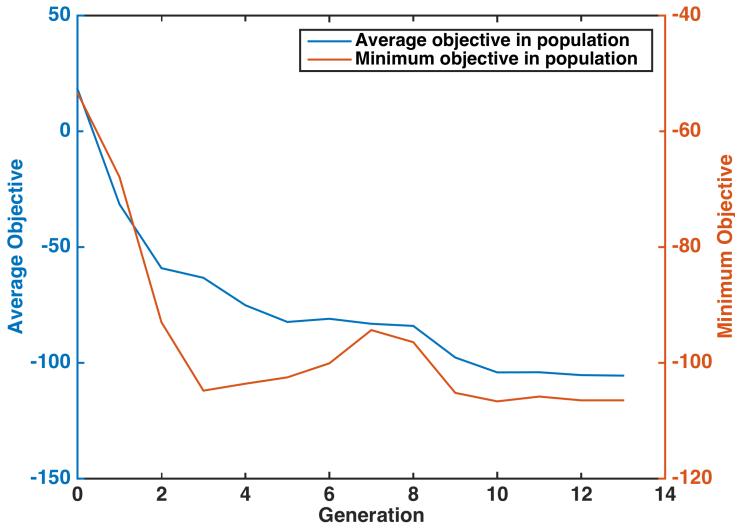


Figure 3 – Objective reduction (first ES run - default parameters)

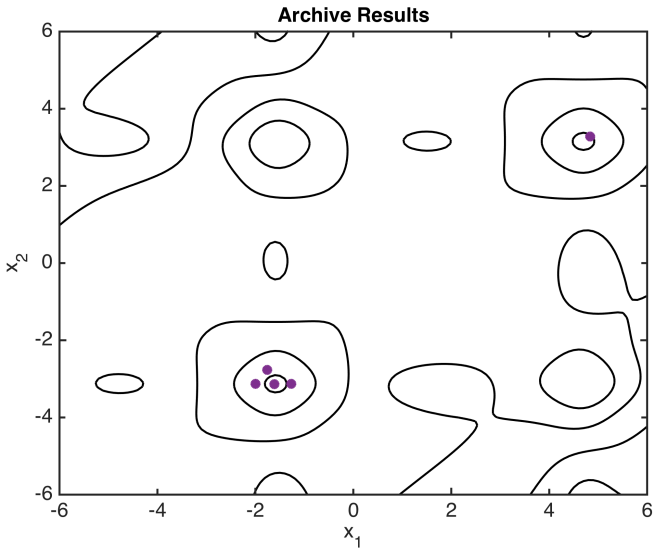


Figure 4 – Best 5 Dissimilar Solutions Archive (D_{min}=0.4, D_{sim}=0.05)

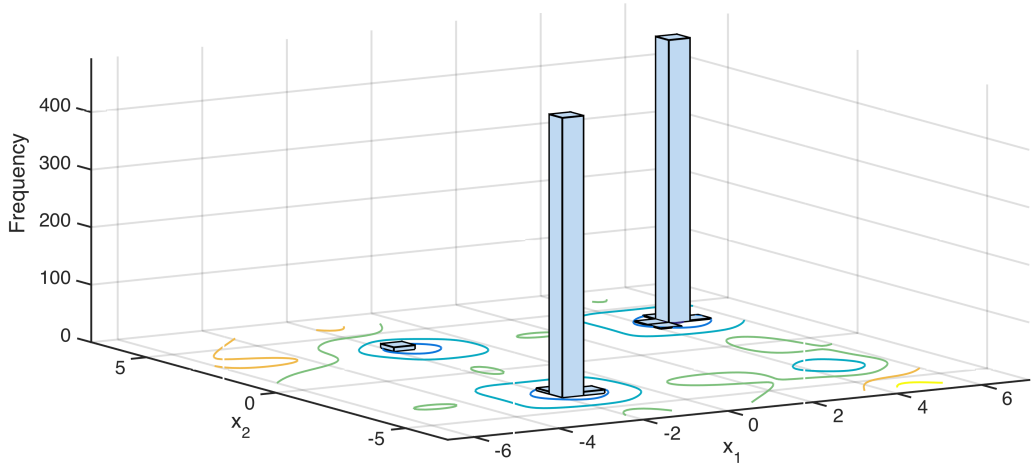
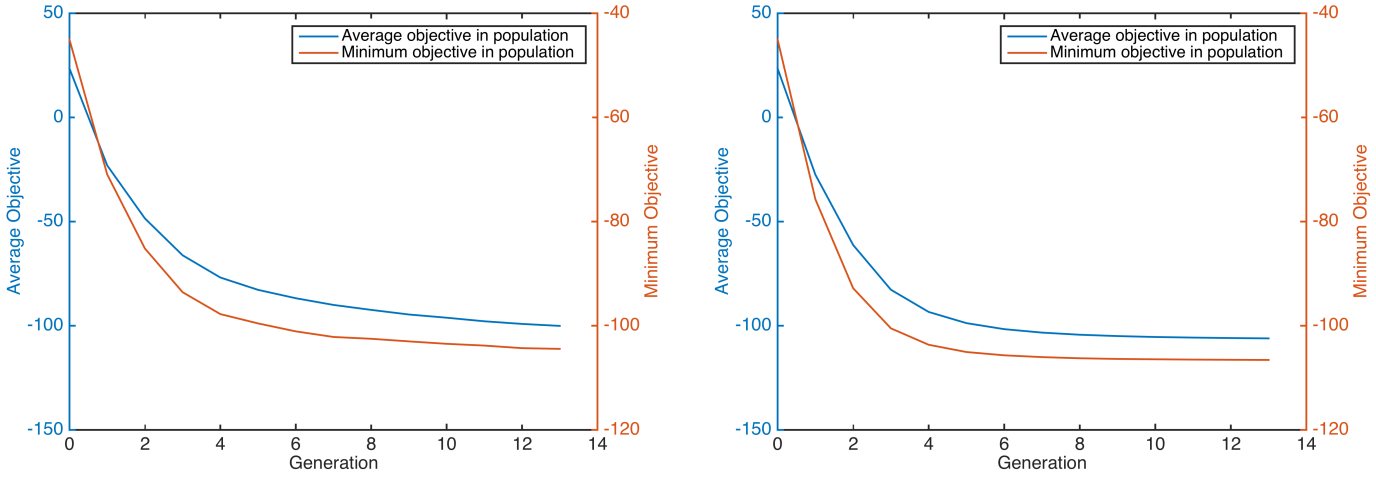


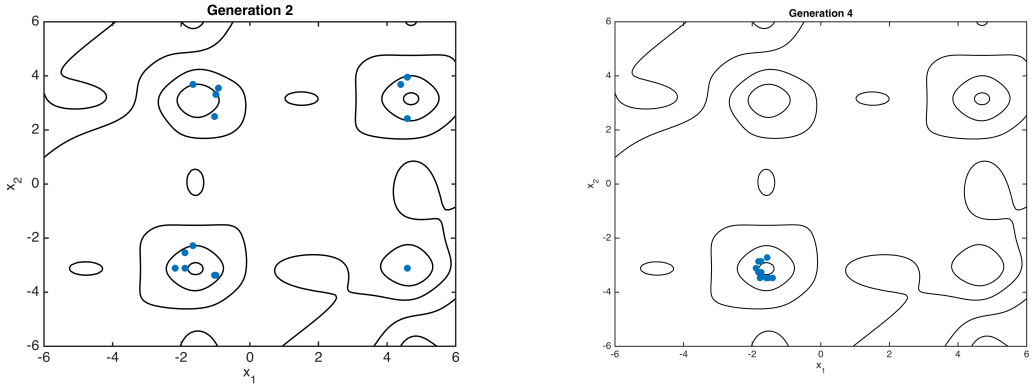
Figure 5 – Histogram showing how often the two minimums were found over 1000 differently random seeded runs (overplayed on contour map) - using default settings.

The selection process for the both the (μ, λ) and $(\mu + \lambda)$ selection schemes was deterministic. This meant that the best λ of the new generation survived. Other selection mechanisms, which can be probabilistic [4, §5], exist but they were not studied in this report.



(a) (15 , 75)-selection **(b)** (15 , 75)-selection i.e. elitist

Figure 6 – Comparison of different section schemes. These plots have been produced by taking the average values over a 1000 different runs. Other control settings were left at their defaults.



(a) Generation 2 **(b)** Generation 4

Figure 7 – How the elitist ES algorithm progresses

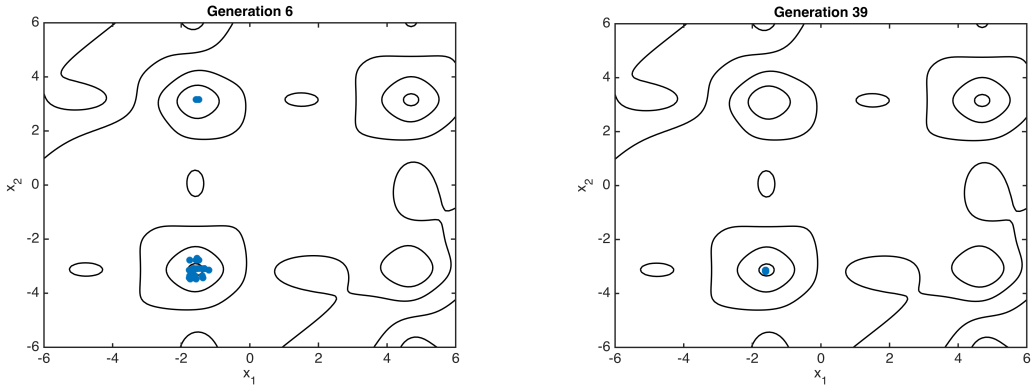
3.4 Changing the Population Parameters

By varying the size of the population, the number of generations evaluated is changed to keep the number of function evaluations fixed at 1000. Figure 9 shows the objective value reduction for each generation when the number in the population was increased to 30 and decreased to 5, whilst the parent child ratio was kept constant at 5. These schemes allowed the evaluation of 6 and 39 generations respectively.

Figure 8 shows characteristic final populations for these two schemes, using the same random seed of -561489423 as before. As one can see from Figure 8a, for the population size of 30 the optimisation has been forced to finish early, with points also around a local minimum. This means there is large difference between the average value of the population and the minimum value at the end of the process, as shown in Figure 9a. Another negative point of this run is that although a local minimum point is still being considered, there are no members around the global minimum of B.

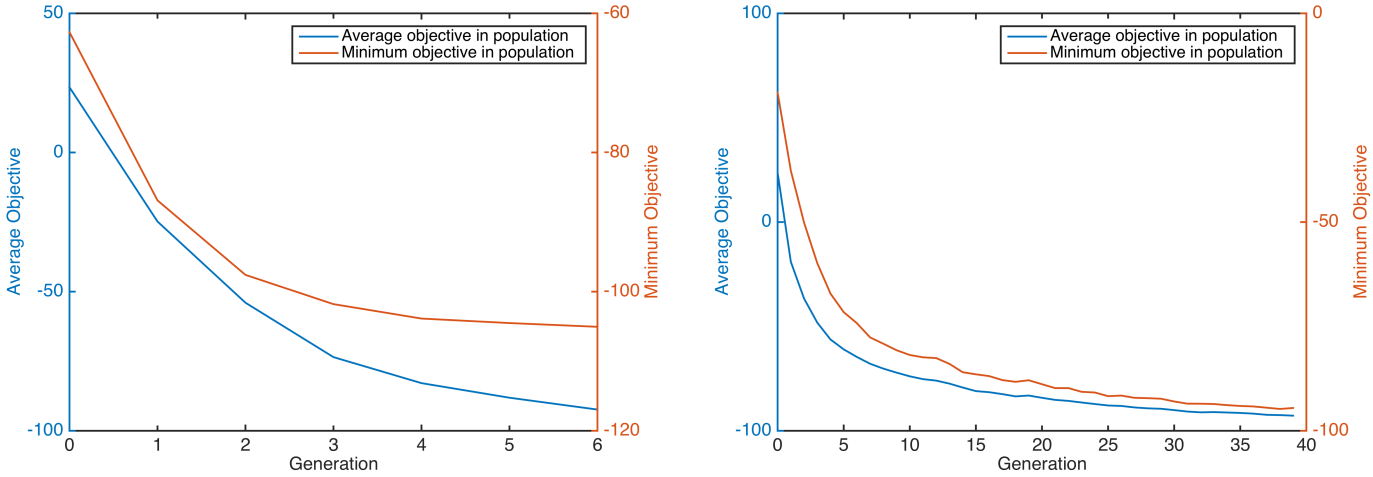
Figure 8b, suggests that the smaller population does better. However, by analysing the generations this population size performs worse that the size of 15, as it takes many generations to settle down around one of the global optima. And by looking at Figure 9b one can see that the average population's minimum value in the last generation is higher than when using larger populations.

Next the parent to child ratio was varied (with the population set at 15). Lowering the ratio to



(a) Generation 6 (population size 30) **(b)** Generation 39 (population size 5)

Figure 8 – The final generation as the population size changes

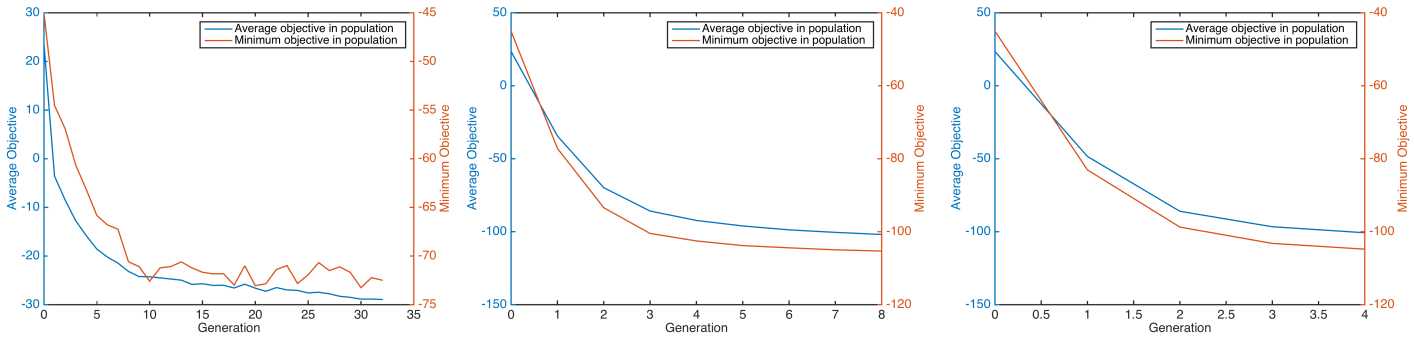


(a) Population size 30 **(b)** Population size 5

Figure 9 – Objective reduction for different population sizes. These plots have been produced by taking the average values over a 1000 different runs.

2 meant that the algorithm had less function evaluations per generation ($15 \times 2 = 30$ compared to 75). So the number of generations evaluated increased. However, at the same time the surviving population for each generation was not as fit as it would have been had more children been assessed. This meant the algorithm took more generations to converge, as seen by Figure 10a. Note also that the average minimum values found (right hand orange scale) were a lot higher than before - indicating worse overall algorithm performance.

With the parent to child ratio increased to 8, the algorithm exhibited similar behaviour to using a factor of 5 but only 8 generations were assessed, as seen in Figure 10b. When trying it on the random seed of -561489423 the algorithm converged to minimum B instead but exhibited similar behaviour as to when using the ratio of 5. And indeed Figure 10b has similar curves to Figure 7a albeit a bit more steep (note the different scaled axis) - hence showing that it took less generations to converge. This is expected as the child population being assessed each time is larger and hence the survivors should be fitter. Figure 10c shows the objective reduction with a ratio of 14, this as expected shows even quicker convergence. However, increasing the ratio much above this would not be desired as the numbers of generations that can be assessed would drop very low.



(a) Parent-child ratio of 2 **(b)** Parent-child ratio of 8 **(c)** Parent-child ratio of 14

Figure 10 – Objective reduction for different parent child ratios. These plots have been produced by taking the average values over a 1000 different runs. Note x axis scales are different.

4 Tabu Search Algorithm

4.1 Implementation Details

The Tabu Search (TS) algorithm is described in [2, §TS]. The code implements this method. The default search diversification strategy (see Section 4.6) follows that described in [2, p.TS3]. A residence measure based approach [5, p.94] is used with the search space divided up into a grid of 36 subsets - represented by equal area squares. The diversification will choose a random point within the least visited square (or a random least visited square if there are multiple ones) as the new point.

The basic local search routine described in the lecture notes [2] was used in the implementation, as the number of control variables was small (only 2). It was therefore not computationally expensive to evaluate all four potential points around a base point.

The increment was adjusted to fall to half its value when the counter hit the reduction limit (set at 25 as recommended in [2, p.TS4]). It then remained at this value until it next hit the reduction counter again. Some local search variants allow the increment to go up again [3, §9.4], but this was not coded into this implementation.

4.2 Search Patterns Followed by Tabu Search

The search pattern followed by TS is shown in Figures 11 and 12. To produce this figure the algorithm control parameters were set at their recommended values of 10 for the intensification counter, 15 for the diversification counter and 25 for the reduction counter. The initial increment was set at 0.4 and the medium term memory and the short term memory were set at the recommended sizes of 4 and 7 respectively [2]. These settings will be described as the default ones for the experiments that follow. The random seed used was -561489423.

Initially, Figure 11a shows the algorithm immediately finds one of the local minima, through a series of pattern moves. It searches around this local minima and starts to ascend out of it when the intensification counter is hit. By analysing the points explored before hitting this counter, one can see that the search revisits points it has already evaluated (after they have left the short term memory). This suggests that the short term memory is too small for this increment size for this particular problem. However, a smaller increment size at this stage would increase the time spent finding the local minimum in the first place. So like the other control parameters a compromise is needed.

Figure 11c shows the initial diversification, which allows the algorithm to find and explore the area around minimum B. The subsequent reduction and intensification (Figures 11d and 11e) then allows the algorithm to explore the area in more detail. The second diversification (Figure 11f) allows the algorithm to find minimum A. At this point the medium term memory contains points around both minimum A and minimum B, highlighted by the fact that the intensification procedure picks

a point halfway between them (see Figure 11h). This highlights a problem with this intensification strategy and suggests that perhaps a different intensification strategy, such as picking randomly one of the elite solutions [5, p.97], would be more appropriate (see Section 4.6 for a further discussion of this point).

Figure 12 shows the later stages in the tabu search. Here all the intensification is based around minimum A. The diversification forces the search into new areas but now the increment size is so small that little progress is made. This suggests that the method should be modified to either remove diversifications at this stage in the search or increase the increment size when they occur to allow them to be more effective. The former method is briefly investigated in Section 4.6. Figure 12 shows how the algorithm was ultimately successful, finding minimum A.

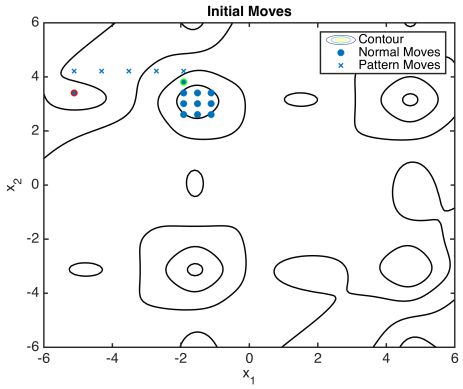
Figure 13 shows how the Tabu Search evolves. One can see that the best base point found so far's value does not change significantly after iteration 40. This is due to the linear scale and by analysing the values it does drop further until about iteration 70 (also see Figure 36 in Appendix A). The diversifications after this point all push the algorithm away from the minimum and waste function evaluations. This can be seen as the rectangular shaped pulses in the figure. Initially the pulse tops are sloped but later on they are flat, as the increment size is small and so the base points are all close to each other and so similar in value.

Figure 14 shows the result of the archive for this search. The same archive settings were used as those for producing Figure 4 for ES. The archive has again recorded points around the two local minima. It should be noted, however, that this is dependent on the random seed and so how the search progresses, and will not always be the case. Again D_{min} should probably be increased to try to get the archive to record points further apart. Due to the way that TS picks the best solution during the local search and hones in on it before it diversifies, the archive results are closer to the minimum for minimum B than the archive for ES.

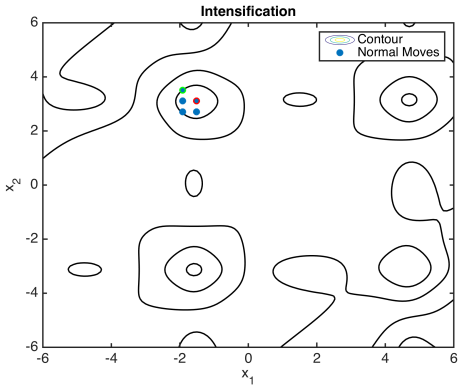
The same settings were then maintained over a run of 1000 tabu searches with different initial random seeds. Nine of these searches failed when getting stuck by not being able to find a new potential point that was not tabu or in bounds. The best result found on each run (i.e. the best item in the archive) is plotted in a histogram form in Figure 15. Comparing this Figure with the equivalent for ES (Figure 5), one can see that the tabu search performs worse, finding a local minimum more times than the ES algorithm. However when finding one of the global minimum it finds a point closer to it more often - seen by the smaller number of bars around the global minima.

The average minimum value over the 1000 runs was -105.56. The histogram shows 559 of the results are around minimum A and 385 around minimum B. The larger amount around minimum A may be because it is closer to the centre of the search space. This means that the algorithm, which is based upon a greedy local search, can approach it from both sides. Minimum B on the other hand cannot be approached easily on its right hand side, as it is close to the boundary and the search diversification routine cannot pick points outside the boundary.

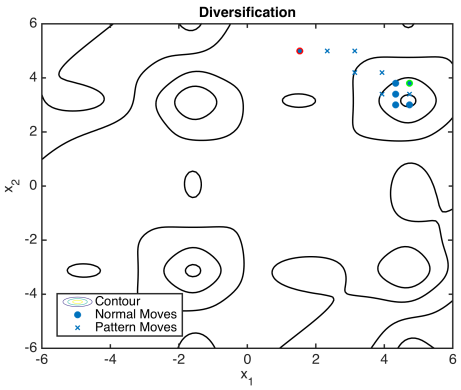
In the next several subsections we will look at comparing this TS, with the parameters set at what we will call their default settings, with other TSs where the parameters or strategies have been changed.



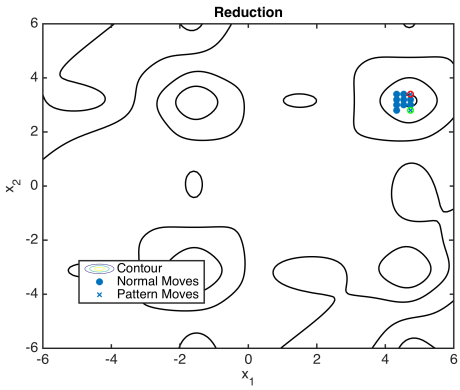
(a) Initial Moves



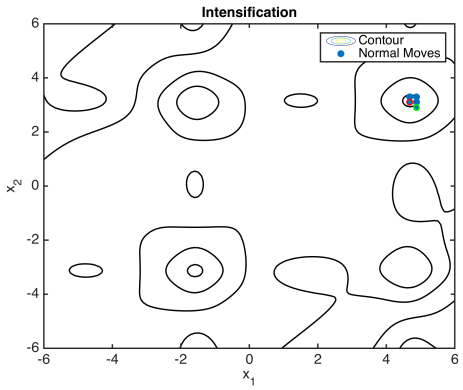
(b) Intensification 1



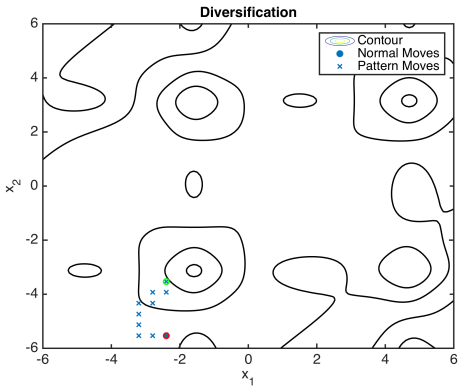
(c) Diversification 1



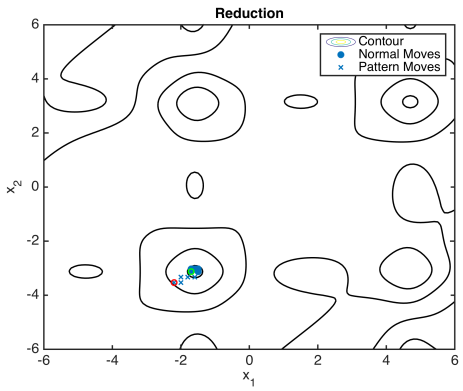
(d) Reduction 1



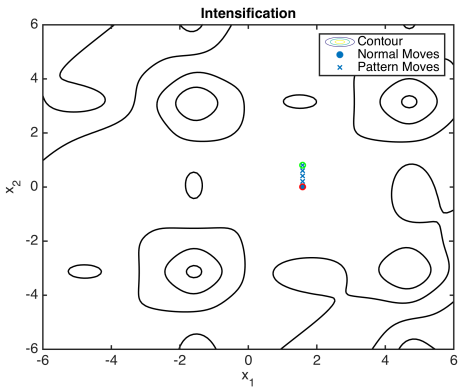
(e) Intensification 2



(f) Diversification 2

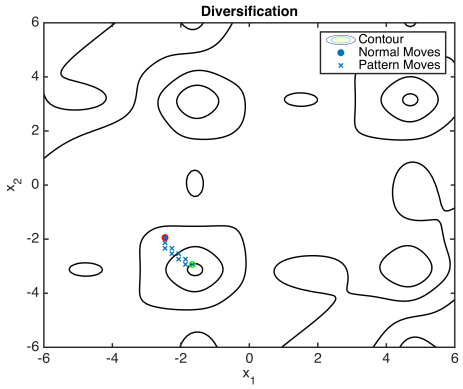


(g) Reduction 2

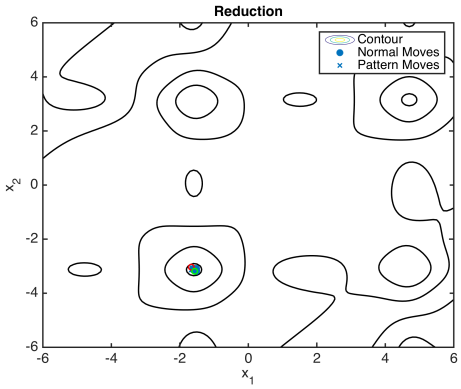


(h) Intensification 3

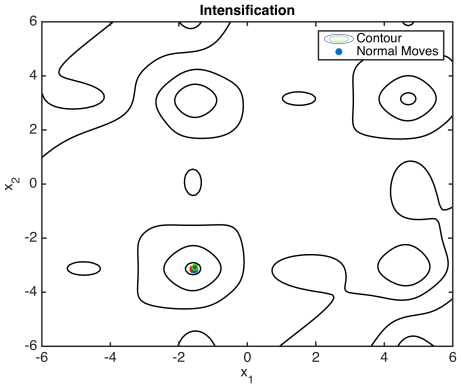
Figure 11 – The first set of base points in the tabu search. The first point for each set of points is circled in red and the last in green. Points that have been added as the result of pattern moves are marked as crosses



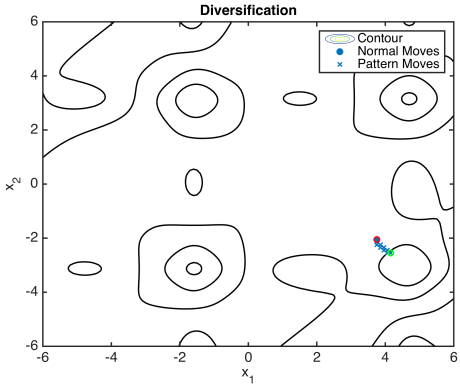
(a) Diversification 3



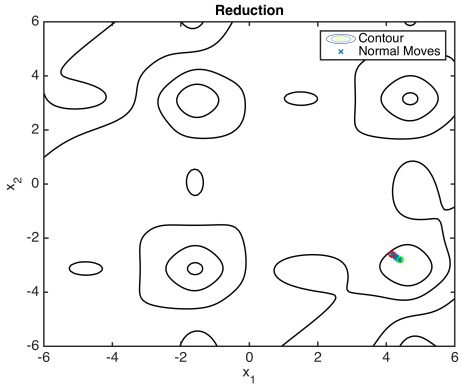
(b) Reduction 3



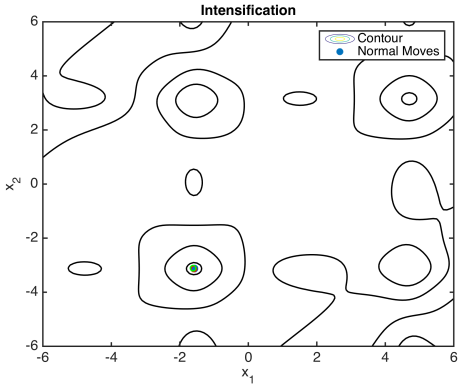
(c) Intensification 4



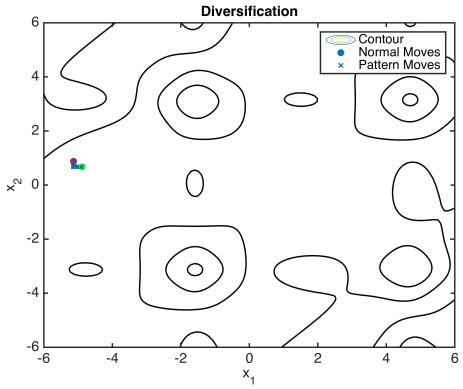
(d) Diversification 4



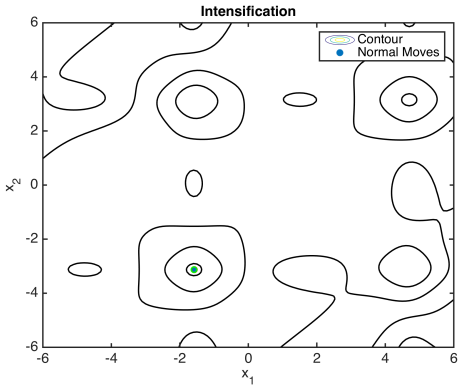
(e) Reduction 4



(f) Intensification 5



(g) Diversification 5



(h) Intensification 9 (last intensification)

Figure 12 – The second set of base points in the tabu search. The first point for each set of points is circled in red and the last in green. Points that have been added as the result of pattern moves are marked as crosses

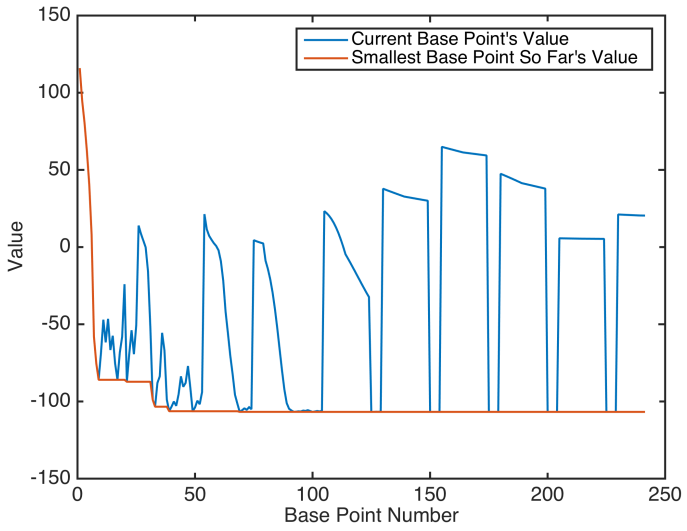


Figure 13 – Base Point's Values

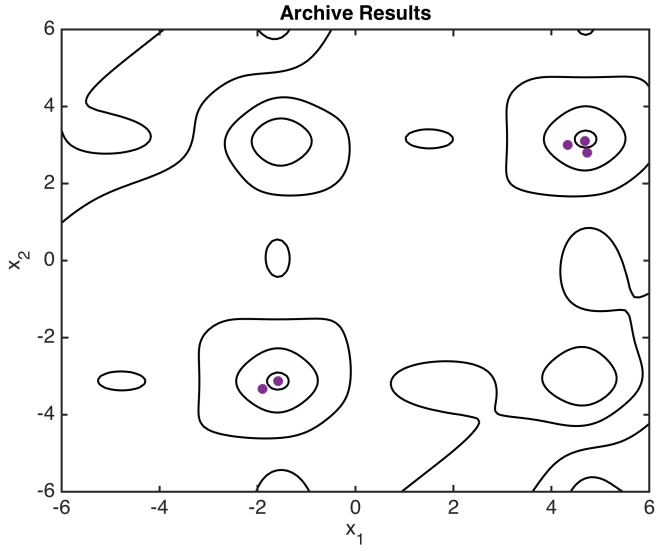


Figure 14 – Best 5 Dissimilar Solutions Archive (D_{min}=0.4, D_{sim}=0.05)

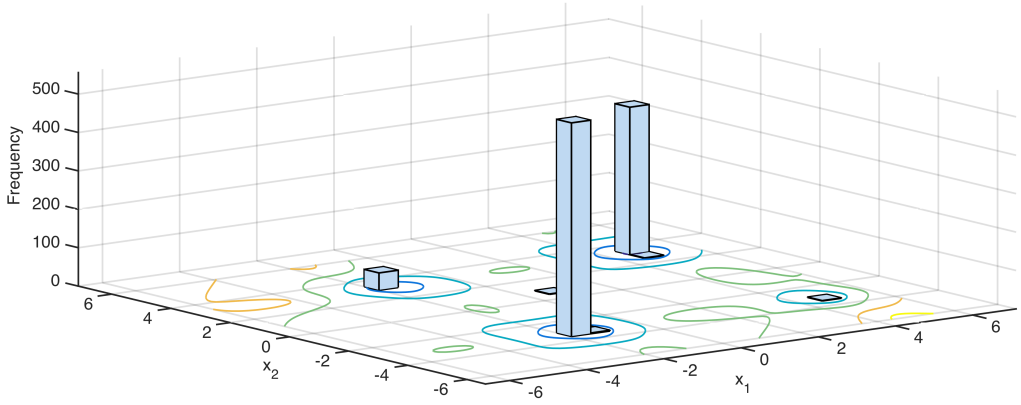


Figure 15 – Histogram showing how often the two minimums were found over 1000 differently random seeded runs (overplayed on contour map). Initial Increment 0.4.

4.3 Increasing the Initial Increment Size

By increasing the initial increment size, the algorithm was able to initially perform a much greater overall search of the space, albeit at a lower resolution. Increasing the initial increment to 0.8 whilst keeping all the other parameters the same meant that over the 1000 runs the algorithm would fail 30 times (i.e. 3%)- due to not being able to make a move that kept within the bounds or was not taboo. This increase is to be expected with the higher initial increments, as the number of all possible moves across the search space is reduced and you are more likely to come into contact with an edge and hence get stuck.

Overall however, when the algorithm worked it was less likely to find a local minimum - as shown by Figure 16. 600 of the results are around minimum A and 363 around minimum B with 7 in a local minimum. This improved performance for finding one of the global minima may be because of its wider initial search of the space. With the usual random seed of -561489423, the initial search and search after the first diversification is shown by Figure 17. From this one can see that the larger steps allows it to find points around both of the local minimum before the first reduction.

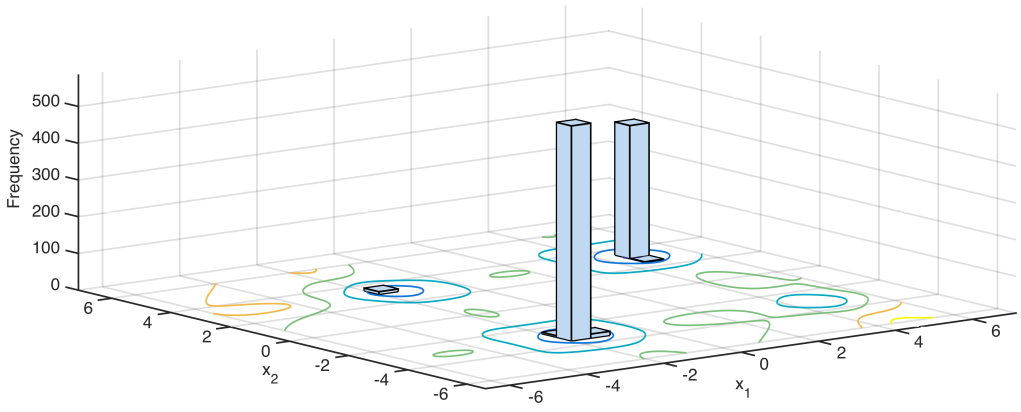
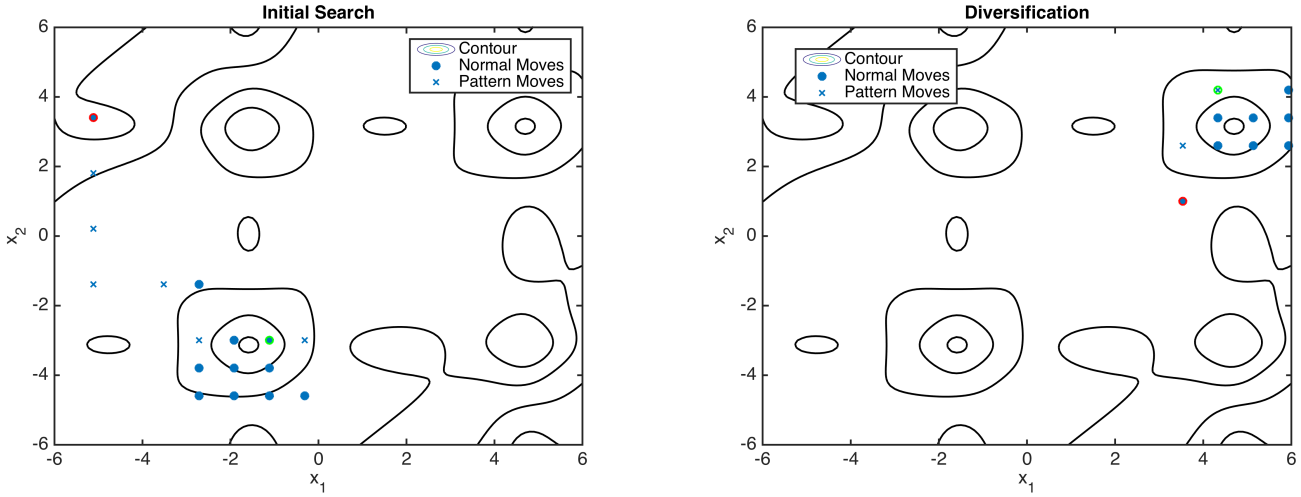


Figure 16 – Histogram showing how often the two minimums were found over 1000 differently random seeded runs (overplayed on contour map). Initial Increment 0.8.



(a) Initial Moves **(b)** After Diversification 1

Figure 17 – Initial search patterns with initial increment of 0.8

4.4 Increasing The Counters

Figure 13 shows that a lot of the diversification steps towards the end of the program waste time, as by then the increment step is too small to explore properly. It may therefore make more sense to increase the counters to spend more time searching before the reducing the step size. Figure 18 shows the results found over 1000 runs with the counters set at 20, 30 and 40 for the diversify, intensify and reduction counters respectively. It finds minimum A 576 times and minimum B 382 times (and fails 5 times). So 958 times the algorithm finds one of the two global minima, a slight improvement on the 944 times when using the default settings.

Figure 19a shows how the base point's value changes over one run . The number of diversifications has changed, as seen by the fewer number of pulses.

Increasing the counters further was explored next. Figure 20 shows the results found over 1000 runs when the counters were increased to 20, 40 and 50 for intensify, diversify and reduction counters respectively. For this run of 1000, 11 failed, 549 found minimum A and 412 minimum B. That means 961 attempts found one of the two global minima, compared to 944 before with the default settings, suggesting these counter settings are better. Also, although the search still finds minimum A more times than minimum B the difference is much less.

Figure 19b shows the base point evolution for one run. One can see from this figure that the optimum base point value was found later in the process, after evaluating about 125 base points. This

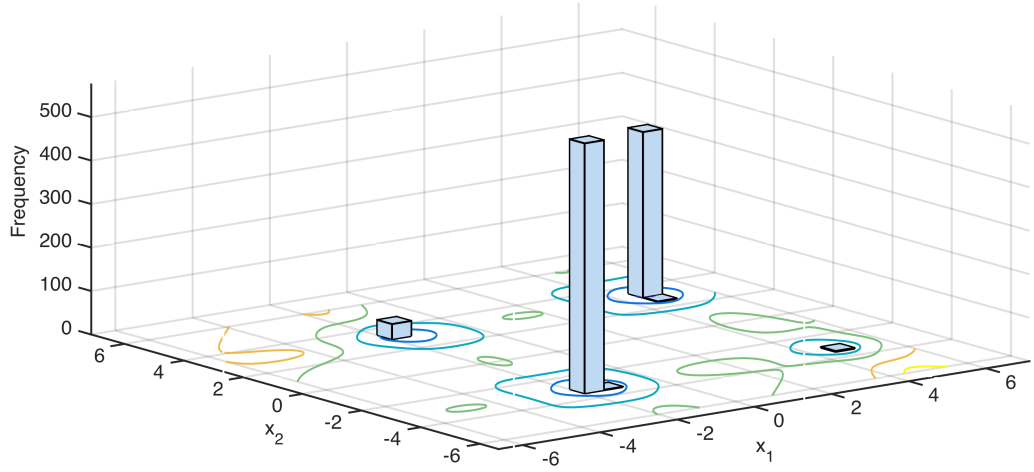
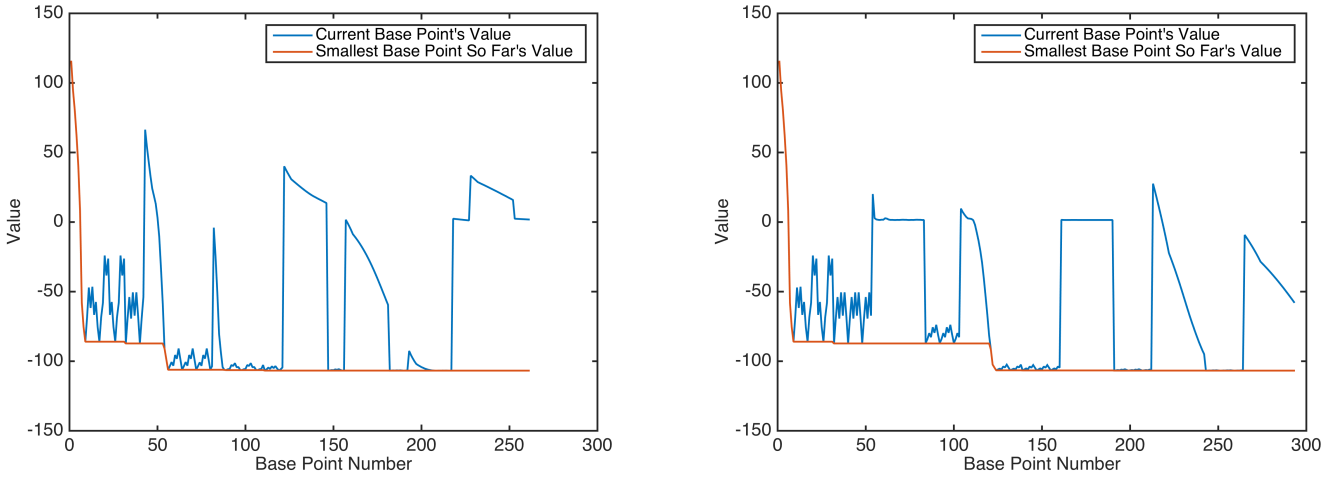


Figure 18 – Histogram showing how often the two minimums were found over 1000 differently random seeded runs (overplayed on contour map). Initial Increment 0.4. But counters have been increased to 20,30 and 35 for intensify, diversify and reduction counters respectively.



(a) 20, 30 and 35 for intensify, diversify and reduction counters

(b) 20, 40 and 50 for intensify, diversify and reduction counters respectively.

Figure 19 – Base Point's Values With New Counter Settings (random seed was -561489423)

is because the algorithm spends more time searching before intensification occurs. The increased counters have made the algorithm more effective but they should not be increased too much further or not enough intensification occurs.

The other problem with large counters is that when combined with small short term memories the algorithm can come back and re-explore points it has already checked (an inefficient use of function evaluations) as the counters allow more steps per stage. Therefore, with the same settings the short term memory was increased to 15. This algorithm spent more time exploring an area and was forced out of local minima more effectively - see Figure 21 for the initial step on one run. Over a 1000 runs, the algorithm failed 394 times, as it could not make a step that was in bounds and not taboo. This highlights the problem with larger short term memories. The algorithm found minimum A 371 times and minimum B 219 times, so because of the failures the algorithm was worse than before.

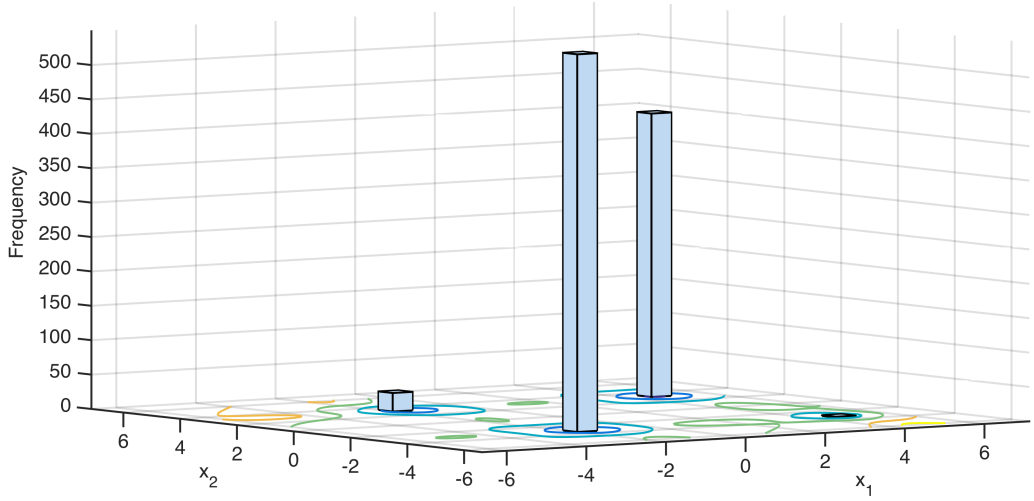


Figure 20 – Histogram showing how often the two minimums were found over 1000 differently random seeded runs (overplayed on contour map). Initial Increment 0.4. But counters have been increased to 20, 40 and 50 for intensify, diversify and reduction counters respectively.

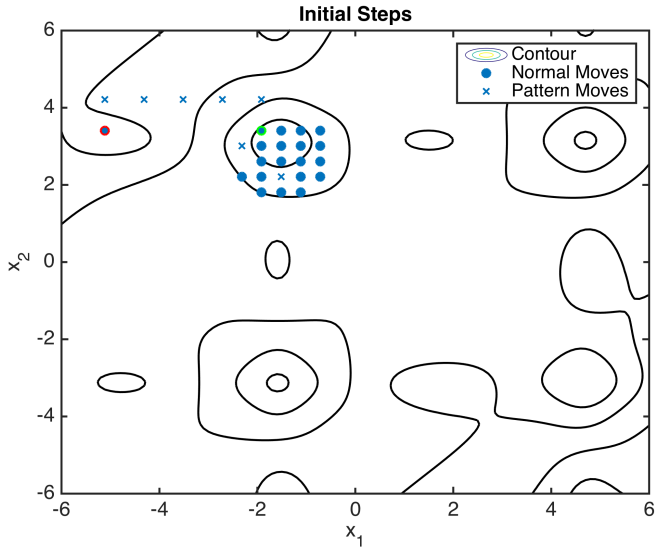


Figure 21 – Intial Steps with counters have been increased to 20, 40 and 50 for intensify, diversify and reduction counters respectively. And short term memory to 15.

4.5 Swapping the Intensification and Diversification Steps Around

Another potential way to get the algorithm to spend more time searching around before intensifying the search, is to swap the intensify and diversify steps around. To this end the diversify counter was set to 10, the intensify to 20 and the reduction counter kept at step 25. The initial increment was reduced back to 0.4. Over a 1000 runs, 9 failed again, 575 found minimum A and 341 found minimum B (see Figure 22), so 916 found one of the two global minima. Although this is a slight improvement in finding minimum A the number finding minimum B has dropped and overall it is worse. By increasing the number of searches between the counters so that intensification is done after 30, diversification after 15 and reduction after 35, this balance is addressed slightly so over 1000 runs 11 fail, 572 find minimum A and 375 find minimum B (see Figure 23). 947 find one of the two global minima compared to 944 with the default settings, this suggests that while rearranging the two counters does not have a positive effect, but this can be offset by reducing the number of times the algorithm reduces.

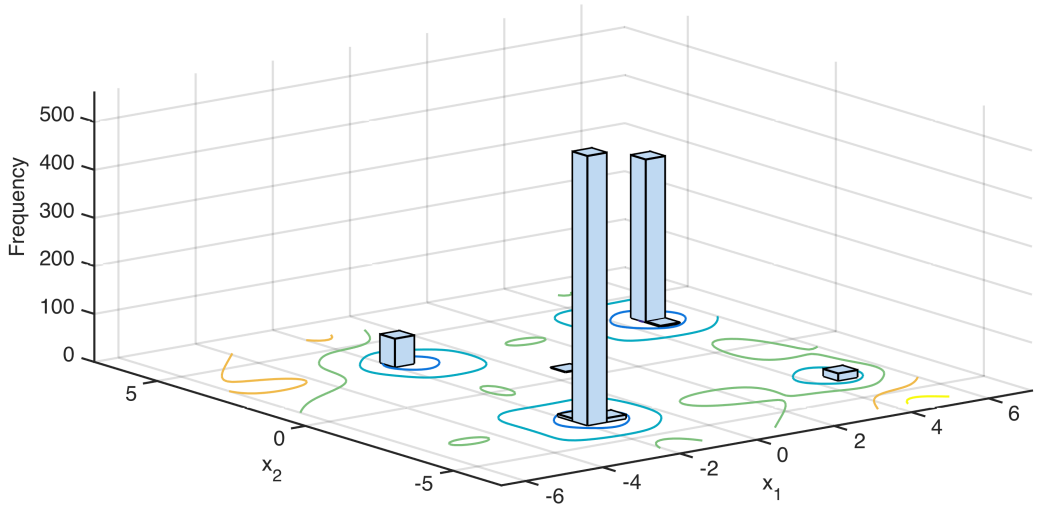


Figure 22 – Histogram showing how often the two minimums were found over 1000 differently random seeded runs (overplayed on contour map). Initial Increment 0.4. The intensification and diversification counters have been modified to 20 and 10 respectively.

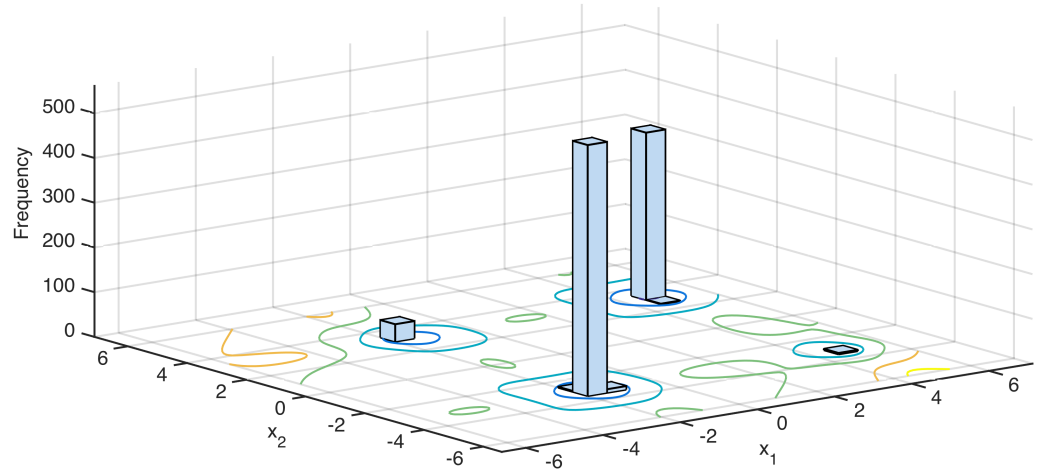


Figure 23 – Histogram showing how often the two minimums were found over 1000 differently random seeded runs (overplayed on contour map). Initial Increment 0.4. The intensification and diversification counters have been modified to 30 and 15 respectively. Reduction counter set at 35.

4.6 Changing Diversification Strategy

Section 4.2 showed how later diversifications were often useless and that initially it is advantageous to explore as much of the search space as possible. This motivates the use of increasing the number of diversification stages at the start but halting them completely in the latter stages, when the increment has become too small. Modifications were therefore made to the code to allow two sets of diversification counters at the beginning of the search and none at the end. The algorithm was then run with the intensification counter at 10, the first diversification counter at 15 and the second at 25 and the reduction counter at 35. The diversification stopped when the global counter reached 10 (see appendix B.5 for more details on this counter), this corresponded to 5 lots of diversifications. The rest of the settings were set as they were in Section 4.2.

The base point progression for the algorithm run with a random seed of -561489423 is shown in Figure 24. One can see that the lack of diversification allows the minimisation process to have no

real changes after 200 iterations (see Figure 37 in Appendix A). By looking at the evolution of base points on a finer scale (again see Figure 37) one can see that the minimum base point is actually found later in the process compared to when using the default settings. However, the new scheme also has more base points recorded; this could be because it allows more pattern moves, which take less function evaluations to assess.

The archive for the scheme is shown in Figure 25. It is fairly similar to the run with the default settings (see Figure 14), but with more points dotted around minimum A.

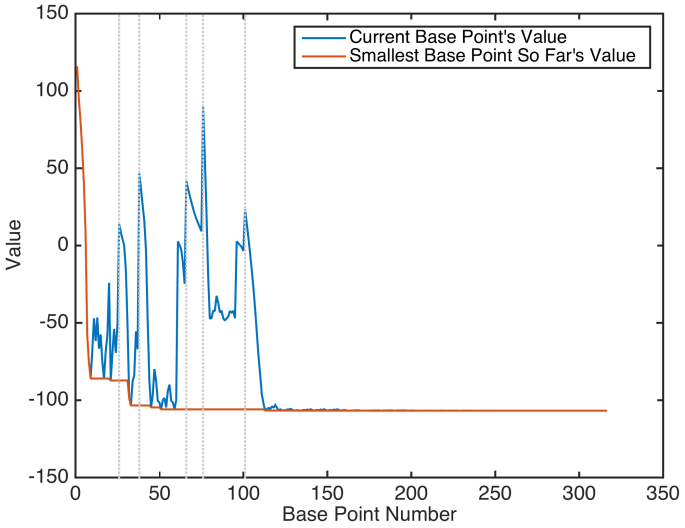


Figure 24 – Base Point's Values, for new diversify scheme. The diversification points are shown in a grey dotted line.

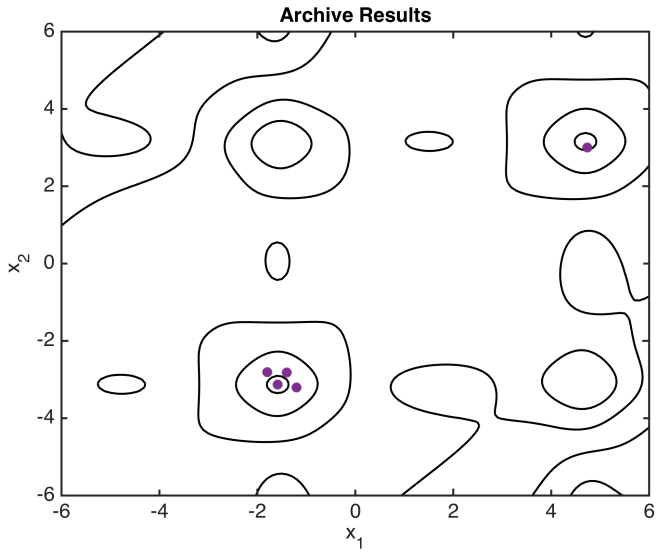


Figure 25 – Best 5 Dissimilar Solutions Archive (D_{min}=0.4, D_{sim}=0.05), with new diversify scheme.

Over a 1000 runs, 12 failed. This is 3 more than with the default settings - this could be because more early diversification leads to more chances for the algorithm to get stuck against the edge of the search area and taboo points. Figure 26 shows how spread about the search area the runs that succeeded were. 584 runs found minimum A and 393 minimum B. That is 977 out of 1000 finding one of the two global minima - the highest proportion so far, suggesting that the new diversification scheme is better. Also the average value found over the successful runs was -106.37, lower than the -105.56 found when using the default settings. This could be because the lack of diversification in the latter stages allows the algorithm to further hone in on the minimum.

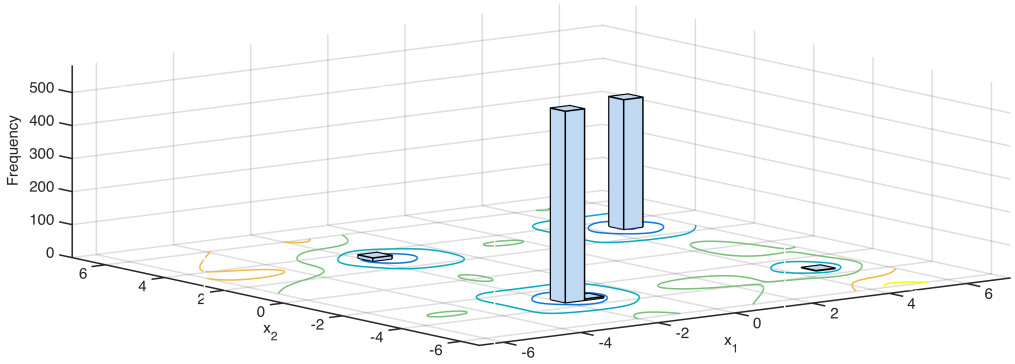


Figure 26 – Histogram showing how often the two minimums were found over 1000 differently random seeded runs (overplayed on contour map). Initial Increment 0.4. New Diversification Scheme

When running through the stages of the algorithm over several different random seeds it was found that the medium term memory would often contain points from the two global minima. This would make the intensification scheme useless, as it would pick out a point in between the two minima for further exploration. Whereas in Figure 11, the algorithm is able to escape such a situation, in the cases analysed it was not. Therefore, using a medium term memory with a size of 1, so it went back to the best point found so far, was tried. In order to get this to work, the short term memory was cleared on these occurrences to make sure that the point could not be taboo.

When run 1000 times the mean of the successful runs was -106.49, a slight improvement on before. This comes at the cost of 13 less runs finding one of the two global minima and 22 out of the 1000 failing.

When a medium term memory of size 1 is applied with the default settings, the average minimum found is -105.74, which is an improvement on -105.56 (found with the default settings and medium term memory of size 4) but not by much. The number of runs finding one of the two minima (or an area around it) is the same with the medium term memory of size 1 and 4. But when using a size of 1 with the rest of the settings at their default, the results were more evenly spread between minima A and B with 517 runs finding A and 427 finding B.

5 Particle Swarm Optimisation

5.1 Implementation Details

Particle Swarm Optimisation (PSO) is defined in detail in [3, §20.3], and the algorithm implemented follows the description there. In addition a parameter to allow the individual to move away from its own worst point was added. This can be viewed as pushing the individual down a hill analogous to the best point pulling the individual into a valley.

The general idea of the algorithm is to mimic the behaviour of a flock of birds or a school of fish. Within the code each individual is therefore described as a bird. When performing a search these animals benefit from communication and this is replicated within the algorithm. The main points of the algorithm can be summarised as follows:

- The individuals (or birds) are initially dotted around the search space. Their initial velocity is set as an input to the program (see next point).
- Each individual has an associated velocity. These are updated on each iteration by several factors:
  - A multiplication/reduction in the current velocity (called stiffness).
  - A craziness factor: a random vector.
  - An amount towards to the individual's own optimum point (called $s_{\text{cognition}}$).
  - An amount towards to the individual's neighbour's optimum point (called $s_{\text{neighbour}}$).
  - An amount away from the individual's worst point (called s_{avoid}).
  - An amount towards the best point found by the group of individuals (or flock) so far. (called s_{social}).

  The strength of all these parameters (i.e. factors in front of each term) can be adjusted as an input to the program.
- The number of individuals is also set as an input to the program.

As this algorithm was not a set part of the assignment and was pursued for interest only, less investigations were done. By fine tuning the program the inputs shown in Table 1 were made the default for the experiments as they seemed to work well.

Setting	Value
initial x velocity	0.1
initial y velocity	0.1
social	0.01
cognition	0.05
avoid	0
stiffness	0.7
neighbour	0.01
craziness	0.1
flock size	5

Table 1 – Default Settings for PSO

5.2 Search Patterns Followed

Figure 27 shows the pattern followed by the flock when the program was run. For this run the default settings were used (Table 1) and a random seed of -561489423. By looking at the Figure one can see that the high value for cognition compared to social means the individual will usually come back to their own best location rather than the flock's best position. The individuals although initially starting apart join up into groups as a result of the neighbour parameter.

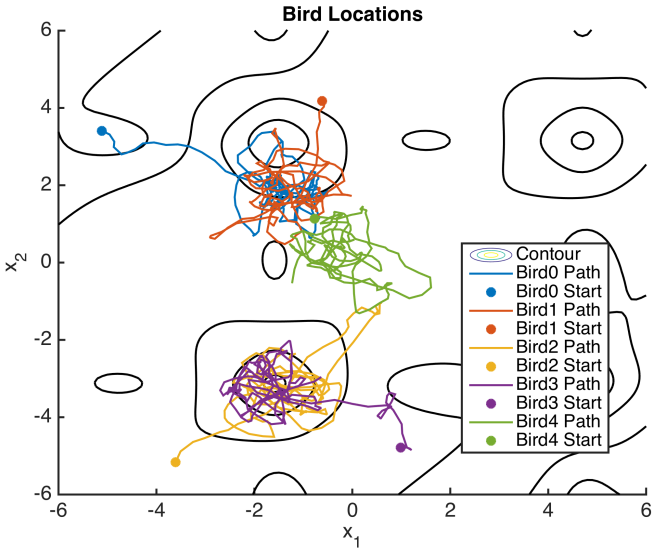


Figure 27 – Patterns of the flock.

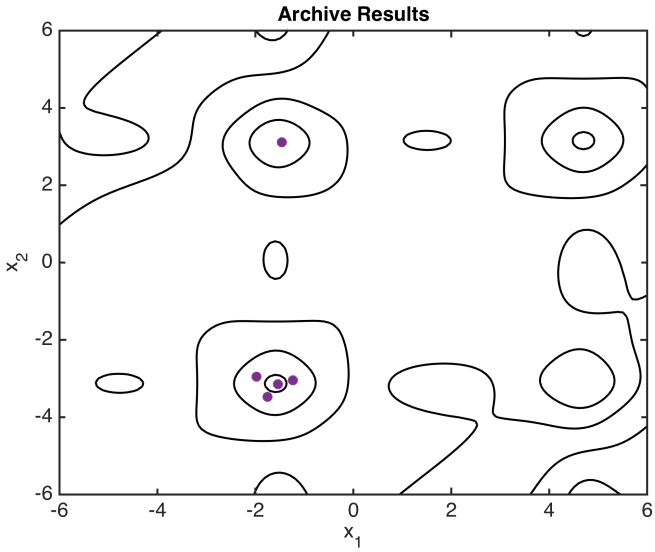


Figure 28 – Best 5 Dissimilar Solutions Archive (D_{min}=0.4, D_{sim}=0.05), of flock

Figure 28 shows the archive for this run. Unfortunately the algorithm has only picked out one of the global minima and a local minima for the archive, so for this random seed it has performed worse than the ES and TS algorithms. This corroborates with the best points found over a thousand seeds (see Figure 29) where an area around minimum A was found 490 times and an area around minimum B 241 times. Minimum A was probably found more times for the same reasons as it was for TS: it is in the centre of the area and the individuals can approach from both sides. The average minimum value found was -98.0641. An area around[1] one of the local minima was found 731 times. The PSO algorithm with these settings was more susceptible to finding local minima then the ES and TS algorithms.

[1]this includes neighbouring bars - not just the tallest bar's area

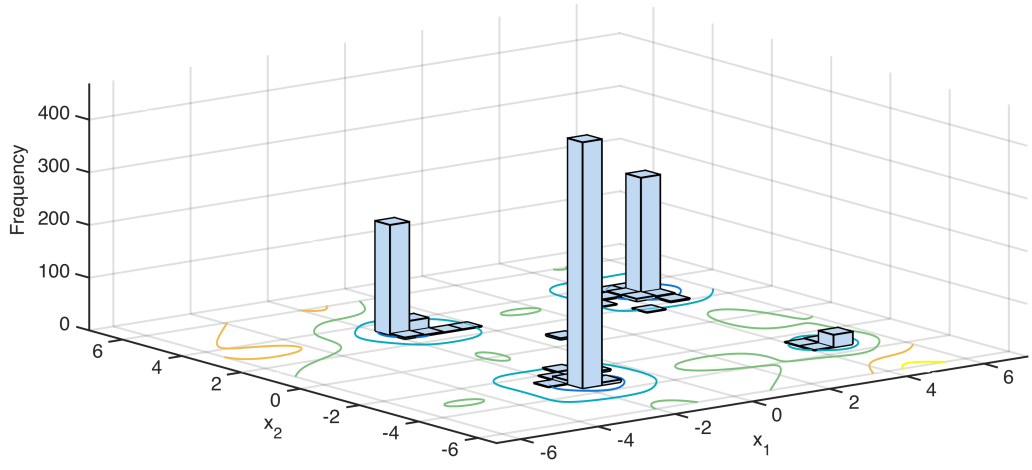
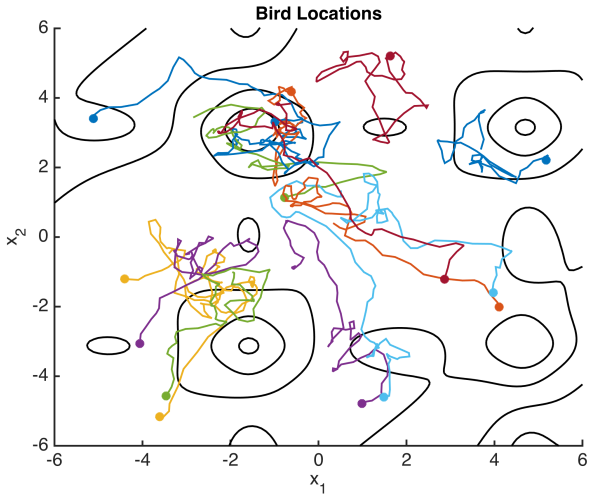


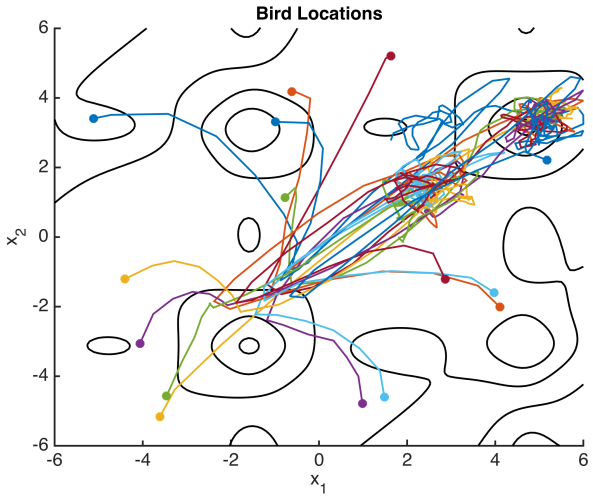
Figure 29 – Histogram showing how often the two minimums were found over 1000 differently random seeded runs (overplayed on contour map). PSO with default settings.

5.3 Changing the Flock Size

By tripling the flock size to 15 the number of iterations each bird can make is also reduced by approximately a factor of 3. This can be seen in Figure 30a. With the other settings kept the same, one can see that this change results in a wider search of the space but in less detail. Figure 31 shows the locations found when the algorithm was run a 1000 times with this setting. This algorithm does better, finding a location around one of the two global minima 829 times. It is still biased towards A, which was found 541 times.



(a) Bird Locations with flock size of 15. Rest default settings



(b) Bird Location with flock size of 15 and increased social and avoid coefficients.

Figure 30 – Path travelled with flock size of 15.

From analysing Figure 30a one can see that the individuals are not cooperating as a group as well as they could be doing, with groups of birds searching in their own distinct areas. Therefore, the social coefficient was increased to 0.1. Also the avoid coefficient was set at 0.01. Over a thousand runs this new search found an area around a global minima less times (758 in total). However, the minima was often found to greater accuracy, as shown by the fewer bars around each minima in the results histogram (Figure 32). The average minimum value found was -101.5562, which is better than using the default settings. On an individual run, one can also witness enhanced flock cooperation,

see Figure 30b. Here all the birds after their initial search join together to better search the area around minimum B.

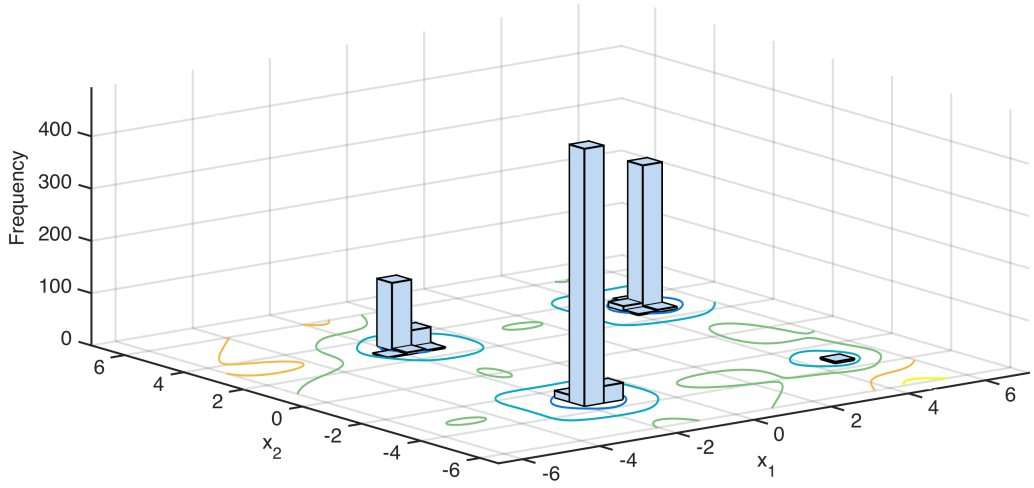


Figure 31 – Histogram showing how often the two minimums were found over 1000 differently random seeded runs (overplayed on contour map). PSO with group size of 15.

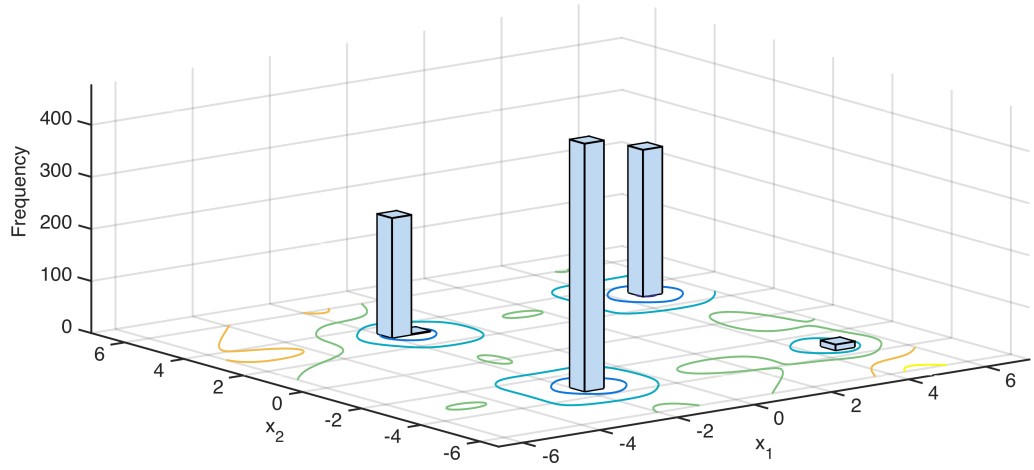


Figure 32 – Histogram showing how often the two minimums were found over 1000 differently random seeded runs (overplayed on contour map). PSO with group size of 15 and increased social and avoidance coefficents.

6 Conclusions

6.1 Comparison of the Three Schemes

This report found that ES and TS offer similar performance over a 1000 runs, finding a point near a global minimum the majority of times. PSO on the other hand was more likely to fall into a local minimum, but this could be because not as long was spent tuning the control parameters. The Bird Function is a hard function to optimise over as it has two equally valued global minima. All the algorithms usually ended at only one of the minimum points so would need to be used in combination with an archive to store the other point. This was briefly looked at but tuning the archive parameters was not done in this report.

The two global minima caused some problem specific implementation issues. The taboo search's medium term memory size could be reduced to 1 to ensure that the intensification stage did not get stuck between the two global minima. The ES did not get stuck due to the two minima but sometimes the population would leave one minimum to focus only on the other. A two tribe ES algorithm was posed in Section 3.2 as a way to prevent this.

Figures 34 and 35 show the box plots for multiple runs for 2 schemes on each algorithm. The first box plot for each algorithm used the default settings and the other used settings described in the report that worked well. The settings for each one are labelled in Figure 34. If a settings is not mentioned then it was left at its default value. The box plots are produced from 978 runs (as 22 out of the 1000 runs failed for TS2).

From these figures one can see that all four algorithms are able to find similar minimum values. Also TS2 seems to be the best of the settings and algorithms used, as the spread of the minimum points found has a low standard deviation and low difference between the upper and lower quartile marks. This may be because the TS2 algorithm is able to hone into the minimum each time, ensuring that it finds it (or a point very close to it) on each run. This confirms with intuition gained from looking at representative runs from the two schemes - ES was often able to effectively find one of the global minima but once found, the TS algorithm was often a much more effective algorithm at honing in. TS2 runs better than TS1, probably because the extra diversifications at the start allow it to find one of the global minima.

This is corroborated by Figure 33, the performance curve for ES2 and TS2. This curve shows how initially, on average, the ES algorithm is better but this only lasts 32 function evaluations (or about 1 generation in ES terms). The TS search would then overtake it before they would both find nearly the same valued minimum: ES2's was -106.38 on average and TS2's was -106.53 on average.

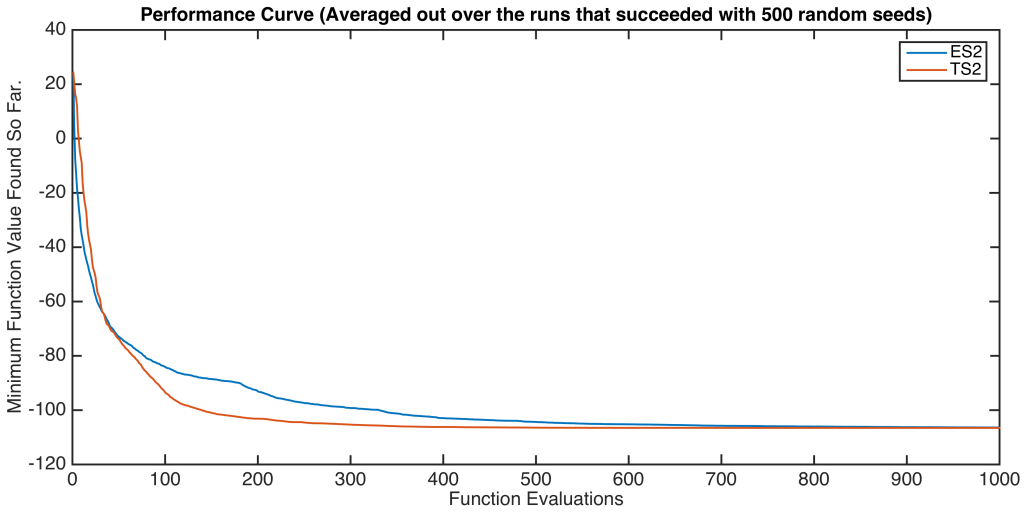


Figure 33 – Performance curve for ES2 and TS2 settings averaged out over 500 runs (Some of the TS runs failed for reasons discussed earlier)

6.2 Further Things to Explore

Due to the large level of flexibility with the three algorithms, there are far more variations that can be explored than there is time to try them. One of the interesting ideas that came out during the work was to look at varying the control parameters as the algorithm progressed. This was briefly looked into in Section 4.6, where extra diversification steps were used at the beginning of the TS program and none at the end. However, it would be more interesting to give even greater flexibility in the control variables' time evolution. The control parameters could perhaps be modelled by a function dependent on the iteration number. This would need changes to the code but, due to the object oriented approach, only to the control parameter classes.

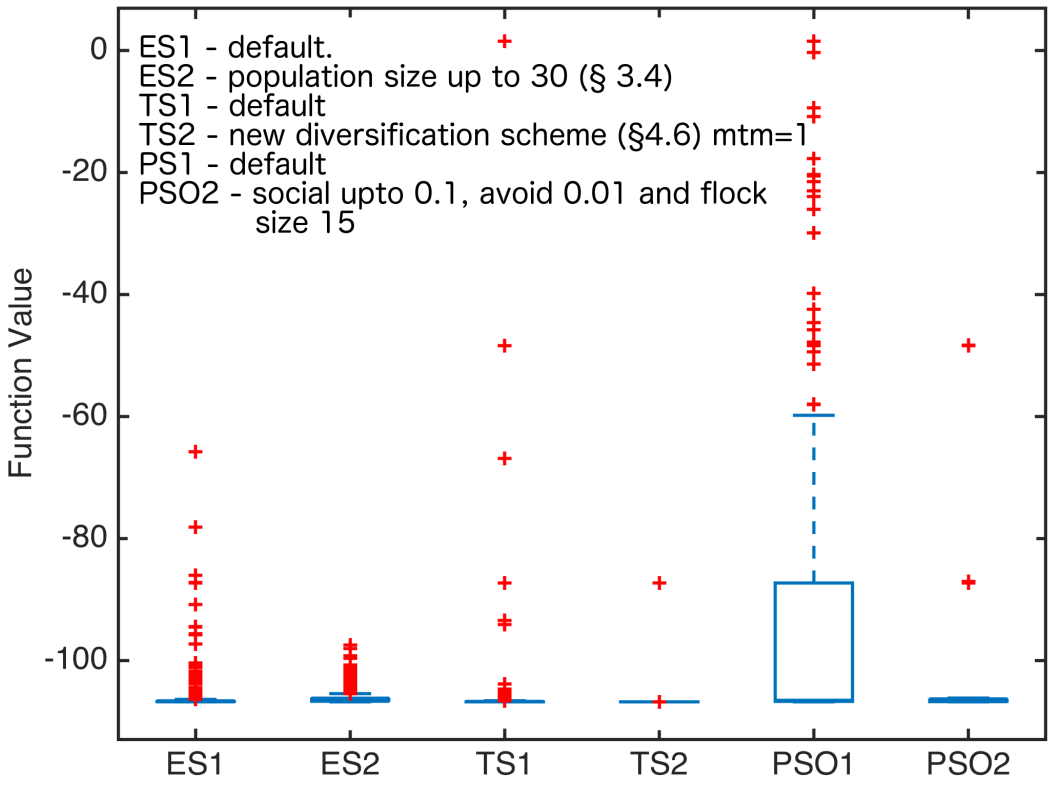


Figure 34 – Boxplots of 6 of the schemes discussed in this report. (Red x's mark outliers that are defined at `http://uk.mathworks.com/help/stats/boxplot.html`)

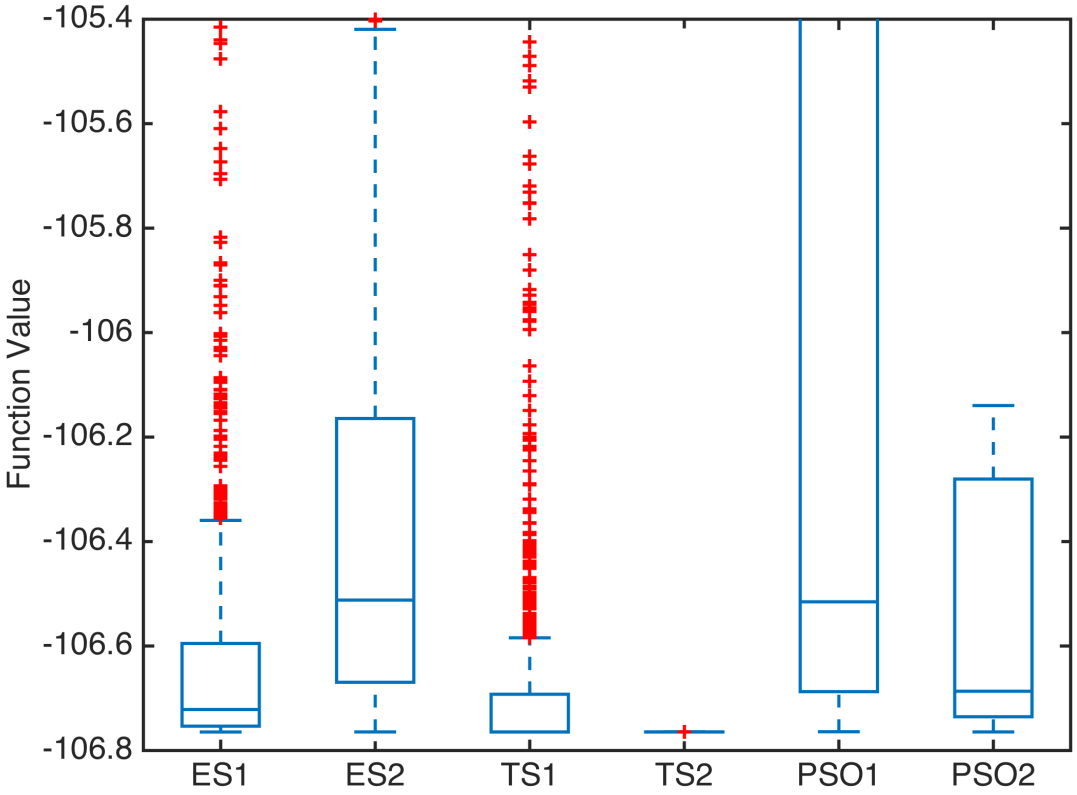


Figure 35 – Zoom in on the boxplots of 6 of the schemes discussed in this report.

Another interesting idea to look at would be to combine several optimisation techniques together. This report has shown how ES is very good at finding an area around a local minima but that TS is better at descending into the minimum. Therefore, a two step optimisation process could be used: using ES initially to find a minima and then TS to descend down in a greedy manner. This would be appropriate for the Bird Function but it would be interesting to see whether a dual algorithm approach would work well on other functions.

A third idea is to integrate derivative information, where possible, into the algorithms and the code. Although this would not be appropriate for ES, it could be implemented into the local search part of the TS algorithm. It could also be added to the PSO algorithm to influence the velocity update formula.

Finally due to time constraints the full level of tuning allowed by the PSO technique was not fully evaluated. It would be interesting to spend some further time to investigate how changing all the parameters changes the search patterns followed by the individuals and how this affects the results. For example, we have seen so far that increasing the social coefficient means the individuals team up and spend more collaborative time improving the best minimum found by one of the birds, and it would be interesting to see what changes to the other parameters did.

References

[1] Parks G, 2014, *Coursework exercise 3 (Assignment Sheet)*,

[2] Parks G, 2014, *Coursework Notes*,

[3] Schneider S.S. and Kirkpatrick S. 2006 *Stochastic Optimization*, Springer, ISBN 3-540-34559-0

[4] Bëck T., 1996 *Evolutionary Algorithms in Theory and Practice*, Oxford university Press, ISBN 0-19-509971-0

[5] Glover F. and Laguna M., 1997 *Tabu Search*, Kluwer Academic Publishers, ISBN 0-7923-9965-X

A Extra Figures for Tabu Search

The following two figures, show the base point evolution for two runs of TS and are referenced in the main report. They have odd scales on the y axis, where they have had their y values shifted up (to avoid taking negative logarithms) by an amount before taking the logarithm. The logarithm has been taken to allow small changes in values to be seen, which cannot be seen easily using a linear scale.

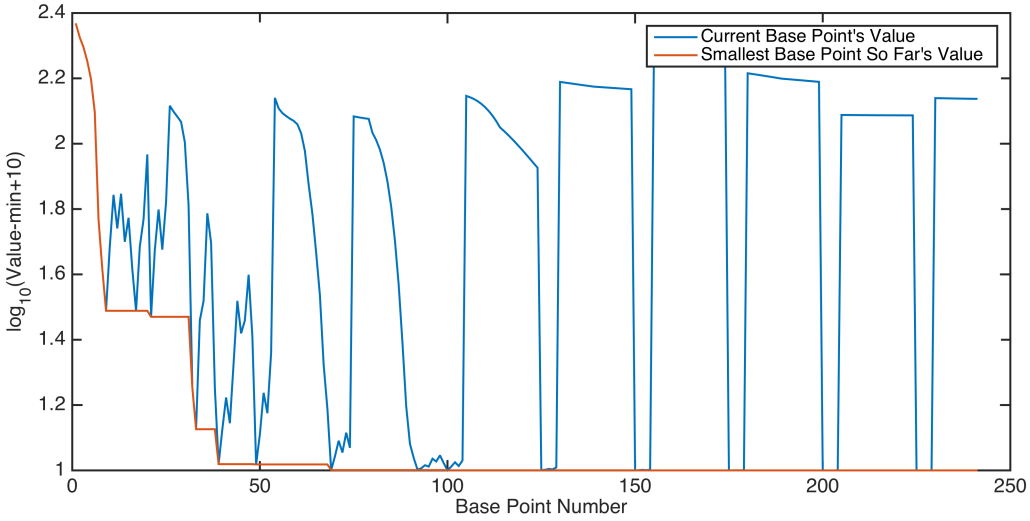


Figure 36 – Evolution of base points on an adjusted y scale. Default Settings.

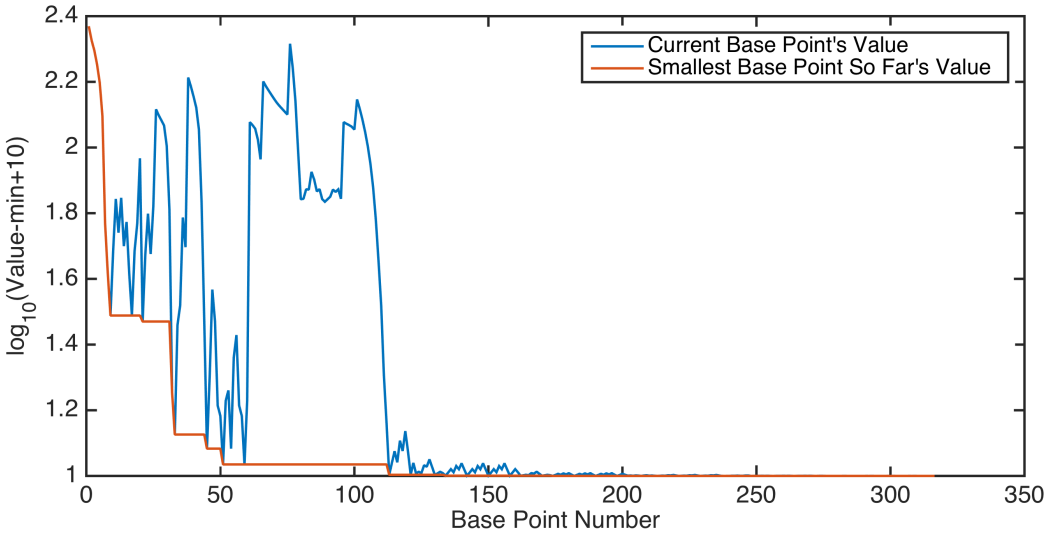


Figure 37 – Evolution of base points on an adjusted y scale. Adjusted Diversification Scheme.

B Source Code

B.1 Introduction to Code

The code was written in three main languages:

Java 8: All the algorithm source code was written in this language. As this is the actual implementation of the algorithm, this section mainly focusses on this code and explains how it works. As requested in the assignment sheet these are 'reasonably well commented'. The main body of this report deals with the overall method of these schemes (and apart from the PSO they

closely follow the lecture notes) so this is not repeated here. All the Java source code is listed in this report.

Python: To batch up multiple runs and collect their results Python was used. This is because it could easily execute system commands and was quicker than MATLAB. As these are utility scripts (and do not include algorithm implementation) only one has been included in this report as an example. The others (available online) follow similar principles but are not commented as much. The code may not be implemented in the most efficient manner but ran at a reasonable speed (approximately 30 seconds for 1000 runs) on the laptop used by the author.

MATLAB: MATLAB was used to collate all the data and produce all the graphs in this report. This graph code is available online but not in the listings. The archive code, which using the function calls could simulate an archive, is included in this report.

All of the code used can be found online at:

`https://github.com/eng-188do/4M17_assignment3`

The reason for picking Java over MATLAB was that when iterating over loops (for example members of the population) it can be a lot quicker. The reason for using Java over C++ was that it was quicker to write programs in. Also, the author wanted to improve their Java experience, with currently only a little experience in using Java from the previous assignment.

Only a little profiling and improvement for speed was done on the program, as it ran fast enough already. If the code was expanded to carry out more function evaluations more would need to be done. There exist many appropriate techniques (e.g. parallelisation) for making the implementations faster.

B.2 Program Structure

The classes were sorted into a series of sub-packages, the overall explanation of which is given in Figure 38.

The high level running of the program was done by a set of classes shown in Figure 39. The class `RunGeneral` was the entrance to the program and accepted when called the following set of arguments:

1. The random seed - this took the form of a Java `long` variable.
2. The input control parameters - this was the pathname of a csv file containing the control parameters. An example of such a csv (and so also the control parameters that can be varied) is provided in the listings for the ES algorithm (see Section B.8).
3. The output filename path stub: this stub was used to create the output files.
4. The type of optimisation: this took the value of a `string` in the set {"ES", "TS", "PS" } for the type of optimisation to be used.

The `RunGeneral` class then set up the appropriate run class for the type of optimisation. Each of these optimisation classes conformed to the `RunOptimisationInterface`. The `FunctionDef` class contained the function definition and could be used to evaluate the function, given a set of coordinates.

auxillary_maths_functions:
This package contained the maths function classes. For matrix calculations.

control_parameters:
This package contained the function to read in and store the control parameters.

coordinate_holder
This package contained the classes that recorded the coordinates and value of the current point.

evolution_strategies_specifics
This package contains the three classes that are used to run the ES algorithm.

exceptions:
This package contained the custom exceptions created.

file_operations:
This package contains the class that allows you to write an array to a cvs file.

loggers:
This package contained all the logging classes.

swarm_specifics:
This package contained the classes that performed PSO.

tabu_search_specifics:
This package contained the classes that ran the TS algorithm

tests:
This package contained all the unit test classes.

also included were 2 other classes and an interface, to compute the function evaluation and run the program from a higher level.

Figure 38 – Java Packages Created

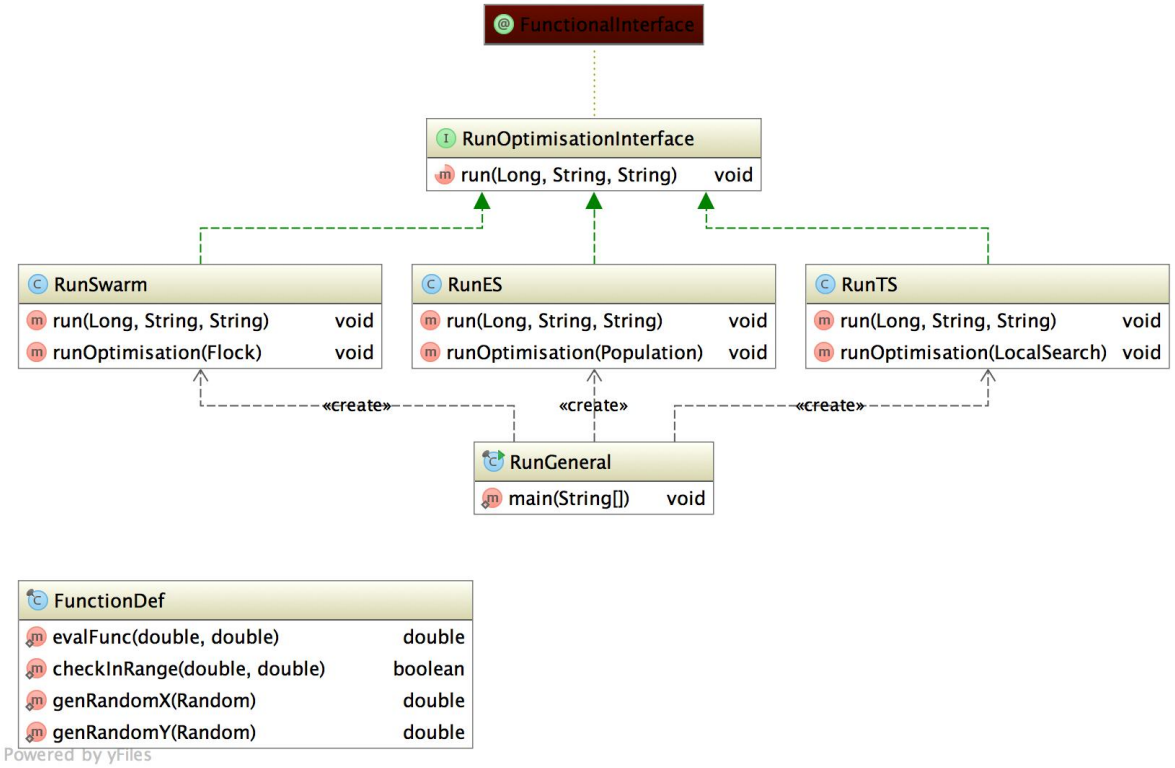


Figure 39 – High Level Classes

B.3 Overall Ideas

The purpose of this section is to elaborate some of the techniques the program used when performing its task.

Tracking the Number of Function Evaluations

The number of function evaluations used was tracked by forcing all function evaluations through a class that logged all the calls made, called `FunctionCallLogger` - shown in Figure 40. This ensured that the function evaluation limit of 1000 could be adhered to. When the function limit was met the logging class would throw an exception, which immediately stopped the evaluation of the program. This was deemed cleaner than tracking a counter through the whole program. The class also could print out all the function calls made, to allow the later simulation of an archive routine.

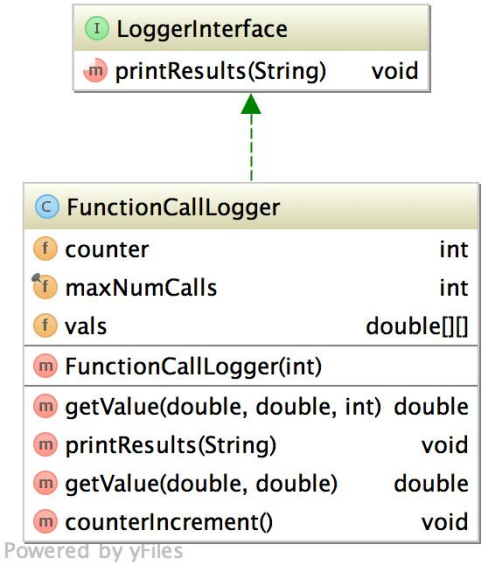


Figure 40 – Function Call Logger

Control Parameters

The control parameters were not hard coded in, so that they could easily be changed. They were input into the program in a csv file. The reading in of these parameters was controlled by the `control_parameters` classes, shown in Figure 41. The following control parameters were allowed to be defined:

ES:

beta : β (defined in Section 3.1)

tau : τ (defined in Section 3.1)

tauP : τ' (defined in Section 3.1)

variance : initial variance of the control parameters.

parents : number of parents in population

parent child ratio : number of children for each parent.

TS:

intensify : intensify counter

diversify : first diversify counter

reduce : reduction (of increment size) counter

intensification factor : reduction (of increment size) factor

initial increment : the initial increment's size

medium term memory : the medium term memory size.

short term memory : the short term memory size

diversify two : the second diversifty counter

diversify stop : the value of the global counter for when we stop diversifying.

PSO:

initial x velocity : individual's initial x velocity

initial y velocity : individual's intial y velocity

social : (defined in Section 5.1)

cognition : (defined in Section 5.1)

avoid : (defined in Section 5.1)

stiffness : (defined in Section 5.1)

neighbour : (defined in Section 5.1)

craziness : (defined in Section 5.1)

flock size : number of individuals in flock.

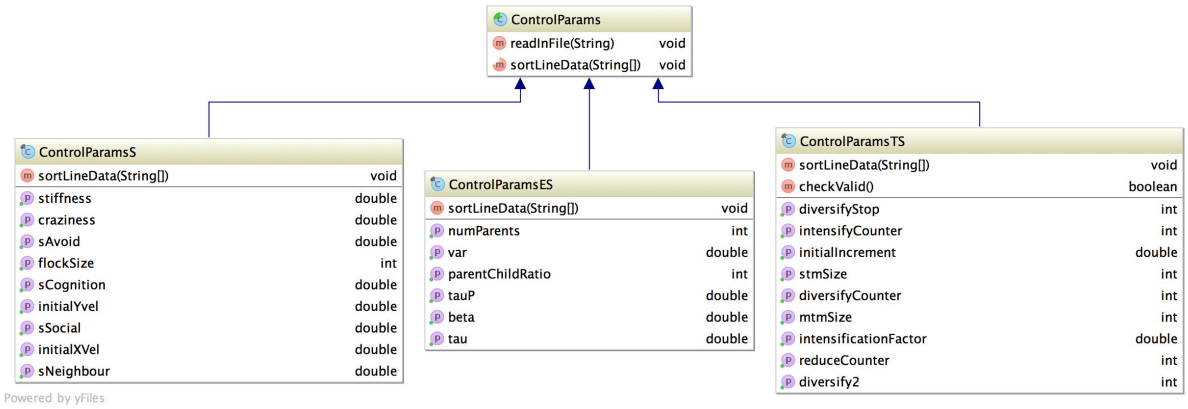


Figure 41 – Control Parameter Classes

Coordinate Holders

When travelling around the search space a class was built to contain the coordinates of the current point and the value of the function at that point. The class was built so that it would evaluate the function only when it was needed (to minimise function calls). The class was then expanded further for the ES and PSO algorithms to allow it to act as the member of the population or individual in the group moving throughout the search space. This structure is shown in Figure 42.

Testing

The testing of the lower level methods was done using Junit 4 and a series of test classes listed in this report. It was hard to test the higher level functions, as they are inherently random so their was no set true result. They were tested by ensuring the outputs were sensible and also by running through the program flow in debug mode.

Figure 42 – Coordinate Holder Classes

B.4 Evolutionary Strategies

The classes involved for ES are shown in Figure 43. The `RunES` class controlled the whole process. It updated a population of individuals (represented by the `ESMember` class) held in the `Population` class. The population would output the surviving population on each iteration into the `SurvivingPopLogger` class.

B.5 Tabu Search

Tabu search used a global counter to record the stages of the algorithm. This was increased on completing a stage, where a stage refers to the search between two counter events. For example, the intensification stage was defined as the points between hitting the intensification counter and intensifying the search until just before you hit the diversification counter and are about to diversify the search.

The classes involved for TS are shown in Figure 44. The `RunTS` class would run the overall

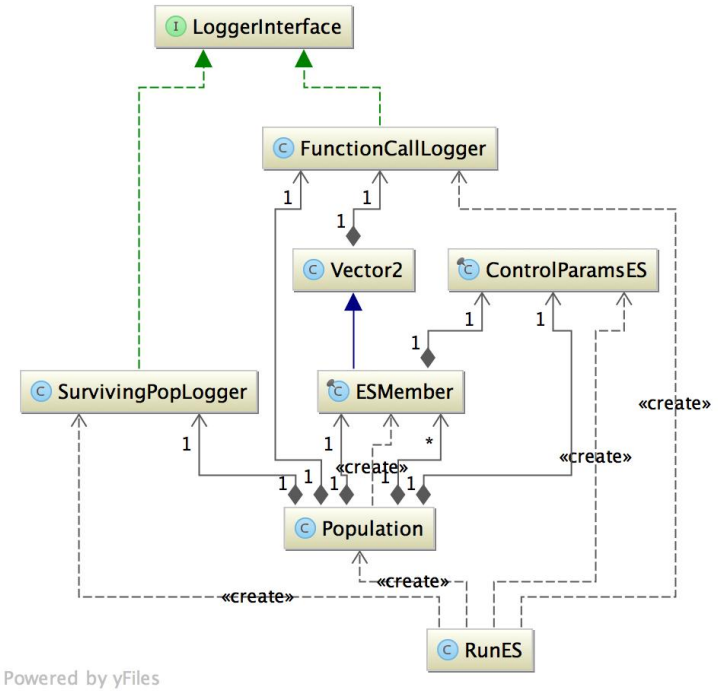


Figure 43 – ES Classes

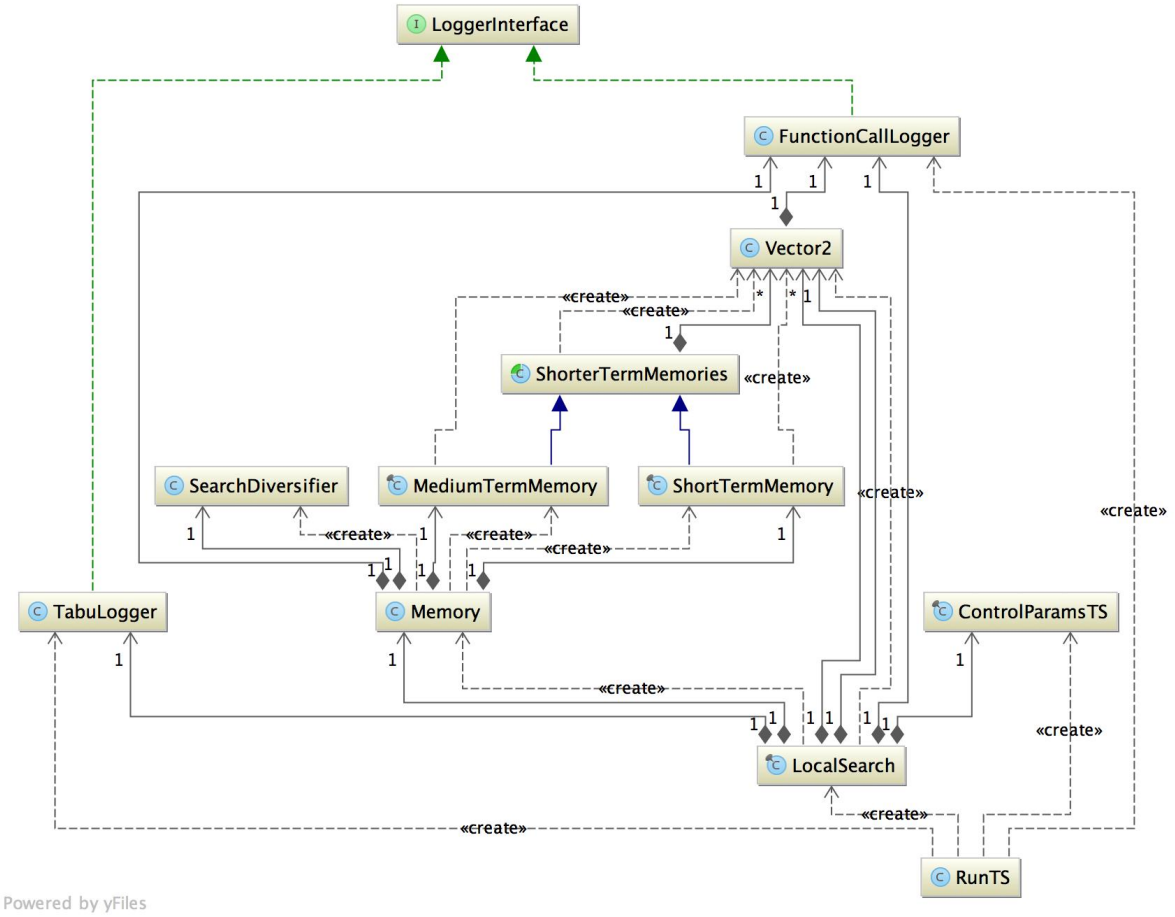


Figure 44 – TS Classes

algorithm. The `LocalSearch` class controlled the series of steps and was the heart of the algorithm. It would output its results to the `TabuLogger` class. The three memories were each controlled by the `Memory` class and this would act as an interface with the `LocalSearch` class. The medium term memory and the short term memory both shared similar features so they were derived from the same parent class: `ShorterTermMemories`. Unlike ES and PSO, the TS algorithm did not extend the `Vector2` class but used it as it was, to store potential points, pattern move points and base points.

B.6 Particle Swarm Optimisation

The classes involved for PSO are shown in Figure 45. The `RunSwarm` class would control the overall running of the program. It would create a group of individuals (held in the `Flock` class). Each individual would be represented by an occurrence of the `Bird` class. The individual would make sure their parameters were output to the `BirdLogger` class on each iteration.

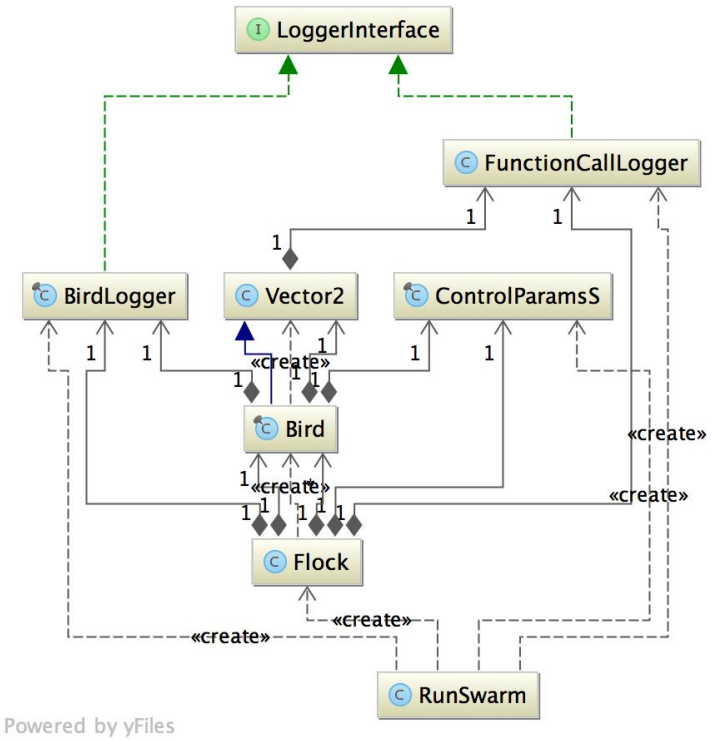


Figure 45 – PSO Classes

B.7 Java Source Code

This section contains all Java Source Code. It is split into sub packages.

High Level Classes/Interfaces

The following classes did not reside in any subpackages.

FunctionDef

```
1 package ex3;
2
3 import java.util.Random;
4
5 /**
```

```
 *
 */

/**
 * @author eng-188do
 * Class that holds the definition of the objective function
 */
public final class FunctionDef {
  //Members
  public static final double xlow=-6; //x lower bound
  public static final double xhigh=6; //x higher bound
  public static final double ylow=-6; //y lower bound
  public static final double yhigh=6; //y higher bound
  public static final double[] centre={0,0}; //marks out the centre of the region
    .

  //Methods
  /**
   * Evaluates the function at x and y
   * @param x this is x_1
   * @param y this is x_2
   * @return f(x)
   */
  static public double evalFunc(double x, double y){
    //check in range
    if (!checkInRange( x,  y))
      throw new IllegalArgumentException("The input value is out of range");

    //Now calculate the actual function
    //Three terms defined on assignment sheet.
    double result;
    result= Math.sin(x)*Math.exp( Math.pow( (1-Math.cos(y)),2 ) );
    result+= Math.cos(y)*Math.exp( Math.pow( (1-Math.sin(x)),2 ) );
    result+= Math.pow( (x-y), 2);
    return result;
  }

  /**
   * Checks input is in the allowed range
   * @param x this is x_1
   * @param y this is x_2
   * @return true if in range, false if outside.
   */
  public static boolean checkInRange(double x, double y){
    boolean val=true; //start off with true and disprove this (if applicable)
    if (x<xlow || x>xhigh)
      val=false;
    if (y<ylow || y>yhigh)
      val=false;
    return val;
  }

  /**
   * Returns a random x in the allowed range
   * @param rnd is the seed
   * @return the random x.
   */
  public static double genRandomX(Random rnd){
    double randomX=rnd.nextDouble()*(xhigh-xlow)+xlow;
    return randomX;
  }
```

```
66
67
68   /**
69    * Return a random y in the allowed range
70    * @param rnd is the seed
71    * @return the random y
72    */
73   public static double genRandomY(Random rnd){
74     double randomY=rnd.nextDouble()*(yhigh-ylow)+ylow;
75     return randomY;
76   }
77 }
```

RunGeneral

```
1  /**
2   *
3   */
4  package ex3;
5
6  import ex3.evolution_strategies_specifics.RunES;
7  import ex3.exceptions.UnknownInputParameterException;
8  import ex3.swarm_specifics.RunSwarm;
9  import ex3.tabu_search_specifics.RunTS;
10
11 /**
12  * @author eng-188do
13  * Class that holds the main method and is the entry point of our program.
14  */
15 public final class RunGeneral {
16
17   /**
18    * Our main entry for program.
19    * @param args expects
20    * -seed, (long) for random initialisation
21    * -input filename (csv format) @see ControlParams
22    * -output filename stub. (different logs have different endings ion this)
23    * -and the type of optimisation to perform!:
24    *     --'ES' for evolutionary strategy
25    *     --'TS' for tabu search
26    *     --'PS' for particle swarm search
27    */
28   public static void main(String[] args) {
29     RunOptimisationInterface optimiser;
30
31     switch (args[3]){
32     case "ES": //evolutionary strategy
33       optimiser=new RunES();
34       break;
35     case "TS": //tabu search
36       optimiser=new RunTS();
37       break;
38     case "PS": //particle swarm optimisation.
39       optimiser=new RunSwarm();
40       break;
41     default:
42       throw new UnknownInputParameterException("The input argument, " +args[3]+"
     is unknown.");
43     }
44
```

```
45      optimiser.run(Long.parseLong(args[0]), args[1], args[2]); //runs the
        optimisation
46
47      System.out.println("The optimisation has finished");
48    }
49
50  }
```

RunOptimisationInterface

```
1  /**
2   *
3   */
4  package ex3;
5
6  /**
7   * @author eng-188do
8   * This is the interface for my code. Program should implement this contract.
9   */
10 @FunctionalInterface
11 public interface RunOptimisationInterface {
12
13
14
15   /**
16    * Runs the optimisation scheme in question
17    * @param seed
18    * @param inputFile
19    * @param outputFile
20    */
21   public void run(Long seed, String inputFile, String outputFile);
22
23
24 }
```

auxillary_maths_functions

TwoDMatrices

```
1  package ex3.auxillary_maths_functions;
2  /**
3   *
4   */
5
6  /**
7   * @author eng-188do
8   *  Class to perform the matrix operations we need
9   */
10 final public class TwoDMatrices {
11
12   /**
13    * Naive (ie O(N^3)) Matrix multiplication
14    * @param one matrix 1
15    * @param two matrix 2
16    * @return product
17    */
18   static public double[][] matrixMultiplication(double[][] one, double[][] two){
```

```
19      int height0=one.length; //height of matrix one
20      int width0=one[0].length;
21      int heightT=two.length; //height of matrix 2
22      int widthT=two[0].length;
23      double[][] output=new double[height0][widthT]; //MxN x NxL=MxL matrix
24
25      if (width0!=heightT) //check can multiply them
26        throw new IllegalArgumentException("Matrix dimensions do not agree");
27
28      //NOTE: as only multiply 2by 2 using simple, not Strassen, algorithm
29      for(int i=0;i<height0;i++){ //next rows
30        for(int j=0; j<widthT; j++){ //next columns
31          output[i][j]=0;
32          for(int ii=0; ii<width0; ii++){ //down row &col
33            output[i][j]+=one[i][ii]*two[ii][j];
34          }
35        }
36      }
37
38      return output;
39    }
40
41    /**
42     * Inverse of two by two matrix
43     * @param matrix
44     * @return inverse
45     */
46    static public double[][] twoByTwoMatrixInversion(double[][] matrix){
47      double det=calcDeterminantTwobyTwo(matrix);
48      double[][] output=new double[2][2];
49
50      //Next move the elements around/times by negative 1 and diveide by
       determinant
51      output[0][0]=matrix[1][1]/det;
52      output[1][1]=matrix[0][0]/det;
53      output[0][1]=-matrix[0][1]/det;
54      output[1][0]=-matrix[1][0]/det;
55
56      return output;
57
58    }
59
60
61    /**
62     * Calculates the determinant of a two by two matrix
63     * @param matrix
64     * @return determinant
65     */
66    public static double calcDeterminantTwobyTwo(double[][] matrix){
67      if(matrix.length!=2 || matrix[0].length!=2)
68        throw new IllegalArgumentException("Matrix is not 2 by 2.");
69
70      double det=matrix[0][0]*matrix[1][1]-matrix[1][0]*matrix[0][1];
71      //NOTE: we could have numerical issues here
72
73      return det;
74    }
75
76    /**
77     * Converts an 1d array to a 2d one where values get put in the first column.
78     * @param array 1d ie { , }
```

```
79     * @return 2d array ie { { , }, { , }}
80     */
81    public static double[][] convertArrayToColVector(double[] array){
82      double[][] vector=new double[array.length][1];
83      for(int i=0; i<array.length; i++){
84        vector[i][0]=array[i];
85      }
86      return vector;
87    }
88
89    /**
90     * Converts a 2d array with values only in its column to a 1d array
91     * @param vector 2d array ie { { , }, { , }}
92     * @return 1d array ie { , }
93     */
94    public static double[] convertColVectorToArray(double[][] vector){
95      if (vector[0].length>1)
96        throw new IllegalArgumentException("Col vector has more than one column");
97
98      double[] array= new double[vector.length];
99
100     for(int i=0; i<vector.length; i++){
101       array[i]=vector[i][0];
102     }
103     return array;
104   }
105
106 }
```

control_parameters

ControlParams

```
1  package ex3.control_parameters;
2  import java.io.BufferedReader;
3  import java.io.FileNotFoundException;
4  import java.io.IOException;
5  import java.io.FileReader;
6
7  /**
8   *
9   */
10
11 /**
12  * @author eng-188do
13  * Base class for reading in control parameters.
14  */
15 public abstract class ControlParams {
16   /**
17    * Reads in csv file containing control parameters
18    * @param fileName
19    */
20   public void readInFile(String fileName){
21     BufferedReader br = null;
22     String line = ""; //temp variable for holding lines
23     String splitter = ","; //what splits our files up
24
25     try {  //try reading in file
26       br = new BufferedReader(new FileReader(fileName));
```

```
27        while ((line = br.readLine()) != null) {
28          String[] lineData = line.split(splitter); //Split up inputs
29          sortLineData(lineData);
30        }
31      } catch (FileNotFoundException e) {
32        e.printStackTrace();
33      } catch (IOException e) {
34        e.printStackTrace();
35      } finally { //try closing the file.
36        if (br != null) {
37          try {
38            br.close();
39          } catch (IOException e) {
40            e.printStackTrace();
41          }
42        }
43      }
44
45      System.out.println("Read in " + fileName);
46    }
47
48    /**
49     * Deals with the line data - specific to the kind of optimisation
50     * @param lineData
51     */
52    protected abstract void sortLineData(String[] lineData);
53  }
```

ControlParamsS

```
1  /**
2   *
3   */
4  package ex3.control_parameters;
5
6  import ex3.exceptions.UnknownInputParameterException;
7
8  /**
9   * @author eng-188do
10  * Control Params for particle swarm algorithm
11  */
12 public final class ControlParamsS extends ControlParams {
13
14   //Members
15   private double initialXVel; //initial x velocity of birds
16   private double initialYVel; //initial y velocity of birds
17   private double sSocial; //see p.171 Schneider & Kirkpatrick "Stochastic
       Optimisation"
18   private double sCognition; //see p.171 Schneider & Kirkpatrick "Stochastic
       Optimisation"
19   private double sAvoid; //see my report
20   private double stiffness; //see p.171 Schneider & Kirkpatrick "Stochastic
       Optimisation"
21   private double sNeighbour; //see p.171 Schneider & Kirkpatrick "Stochastic
       Optimisation"
22   private double craziness; //amount to multiply random number from standard
       gaussian see p.172 Schneider & Kirkpatrick "Stochastic Optimisation"
23   private int flockSize; //number of birds in flock.
24
25
26   //Getters
```

```
27   /**
28    * @return the flockSize
29    */
30   public int getFlockSize() {
31     return flockSize;
32   }
33
34
35   /**
36    * @return the craziness
37    */
38   public double getCraziness() {
39     return craziness;
40   }
41
42
43   /**
44    * @return the initialXVel
45    */
46   public double getInitialXVel() {
47     return initialXVel;
48   }
49
50
51   /**
52    * @return the initialYvel
53    */
54   public double getInitialYvel() {
55     return initialYVel;
56   }
57
58
59   /**
60    * @return the sSocial
61    */
62   public double getsSocial() {
63     return sSocial;
64   }
65
66
67
68   /**
69    * @return the sCognition
70    */
71   public double getsCognition() {
72     return sCognition;
73   }
74
75
76
77   /**
78    * @return the sAvoid
79    */
80   public double getsAvoid() {
81     return sAvoid;
82   }
83
84
85
86   /**
87    * @return the stiffness
```

```
88    */
89   public double getStiffness() {
90     return stiffness;
91   }
92
93
94
95   /**
96    * @return the sNeighbour
97    */
98   public double getsNeighbour() {
99     return sNeighbour;
100  }
101
102
103
104  //Methods.
105
106  /* (non-Javadoc)
107   * @see ex3.control_parameters.ControlParams#sortLineData(java.lang.String[])
108   * input:
109   * initial x velocity,#
110    initial y velocity,#
111    social,#
112    cognition,#
113    avoid,#
114    stiffness,#
115    neighbour,#
116    craziness,#
117    flock size,#
118   */
119  @Override
120  protected void sortLineData(String[] lineData) {
121    switch( lineData[0].toLowerCase() ){ //NOTE
122    case "initial x velocity":
123      this.initialXVel=Double.parseDouble(lineData[1]);
124      break;
125    case "initial y velocity":
126      this.initialYVel=Double.parseDouble(lineData[1]);
127      break;
128    case "social":
129      this.sSocial=Double.parseDouble(lineData[1]);
130      break;
131    case "cognition":
132      this.sCognition=Double.parseDouble(lineData[1]);
133      break;
134    case "avoid":
135      this.sAvoid=Double.parseDouble(lineData[1]);
136      break;
137    case "stiffness":
138      this.stiffness=Double.parseDouble(lineData[1]);
139      break;
140    case "neighbour":
141      this.sNeighbour=Double.parseDouble(lineData[1]);
142      break;
143    case "craziness":
144      this.craziness=Double.parseDouble(lineData[1]);
145      break;
146    case "flock size":
147      this.flockSize=Integer.parseInt(lineData[1]);
148      break;
```

```
149     default:
150       throw new UnknownInputParameterException("Cannot yet take in that input");
151   }
152 
153   }
154 
155 }
```

ControlParamsES

```
1  /**
2   *
3   */
4  package ex3.control_parameters;
5  
6  import ex3.exceptions.UnknownInputParameterException;
7  
8  
9  /**
10  * @author eng-188do
11  *
12  */
13 public final class ControlParamsES extends ControlParams {
14 
15   //Members
16   private double tauP; //defined in notes
17   private double tau; //defined in notes
18   private double beta; //defined in notes
19   private int parentChildRatio; //number of children for each parent
20   private int numParents; //number of parents (and in a population post selection
      )
21   private double var; //variances
22 
23   //getters/Setters
24   /**
25    * @return the parentChildRatio
26    */
27   public int getParentChildRatio() {
28     return parentChildRatio;
29   }
30 
31 
32   /**
33    * @return the numParents
34    */
35   public int getNumParents() {
36     return numParents;
37   }
38 
39 
40   /**
41    * @param var the var to set
42    */
43   public double getVar() {
44     return var;
45   }
46 
47 
48   /**
49    * @return the tauP
50    */
```

```
51    public double getTauP() {
52      return tauP;
53    }
54
55
56    /**
57     * @return the tau
58     */
59    public double getTau() {
60      return tau;
61    }
62
63
64    /**
65     * @return the beta
66     */
67    public double getBeta() {
68      return beta;
69    }
70
71
72
73
74
75    /* (non-Javadoc)
76     * @see ex3.ControlParams#sortLineData(java.lang.String[])
77     */
78    @Override
79    protected void sortLineData(String[] lineData) {
80      switch( lineData[0].toLowerCase() ){ //NOTE
81        case "beta":
82          this.beta=Double.parseDouble(lineData[1]);
83          break;
84        case "tau":
85          this.tau=Double.parseDouble(lineData[1]);
86          break;
87        case "taup":
88          this.tauP=Double.parseDouble(lineData[1]);
89          break;
90        case "variance":
91          this.var=Double.parseDouble(lineData[1]);
92          break;
93        case "parents":
94          this.numParents=Integer.parseInt(lineData[1]);
95          break;
96        case "parent child ratio":
97          this.parentChildRatio=Integer.parseInt(lineData[1]);
98          break;
99        default:
100         throw new UnknownInputParameterException("Cannot yet take in that input")
      ;
101     }
102
103   }
104
105
106
107
108 }
```

ControlParamsTS

```java
/**
 *
 */
package ex3.control_parameters;

import ex3.exceptions.UnknownInputParameterException;

/**
 * @author eng-188do
 * Class to hold and read in the control parameters for tabu search
 */
public final class ControlParamsTS extends ControlParams {

  //Members
  private int intensifyCounter; //after how long do we intensify
  private int diversifyCounter; //after how long do we diversify
  private int reduceCounter; //when to reduce the increment size
  private double intensificationFactor; //how much to reduce the increment size by
  private double initialIncrement; //initial increment size.
  private int stmSize; //short term memory size
  private int mtmSize; //medium term memory size
  private int diversify2; //counter for diversifying the second time. can set negative if don't want to use.
  private int diversifyStop; //counter for when to stop diversifying. if do not want to use just set really big.


  //Getters
  /**
   * @return the diversify2
   */
  public int getDiversify2() {
    return diversify2;
  }


  /**
   * @return the diversifyStop
   */
  public int getDiversifyStop() {
    return diversifyStop;
  }


  /**
   * @return the stmSize
   */
  public int getStmSize() {
    return stmSize;
  }


  /**
   * @return the mtmSize
   */
  public int getMtmSize() {
    return mtmSize;
  }


```

```
59
60
61
62    /**
63     * @return the intensifyCounter
64     */
65    public int getIntensifyCounter() {
66      return intensifyCounter;
67    }
68
69
70    /**
71     * @return the diversifyCounter
72     */
73    public int getDiversifyCounter() {
74      return diversifyCounter;
75    }
76
77
78    /**
79     * @return the reduceCounter
80     */
81    public int getReduceCounter() {
82      return reduceCounter;
83    }
84
85
86    /**
87     * @return the intensificationFactor
88     */
89    public double getIntensificationFactor() {
90      return intensificationFactor;
91    }
92
93
94
95
96    /**
97     * @return the initialIncrement
98     */
99    public double getInitialIncrement() {
100     return initialIncrement;
101   }
102
103   //Methods
104   /* (non-Javadoc)
105    * @see ex3.ControlParameters.ControlParams#sortLineData(java.lang.String[])
106    */
107   @Override
108   protected void sortLineData(String[] lineData) {
109     switch( lineData[0].toLowerCase() ){
110       case "intensify":
111         this.intensifyCounter=Integer.parseInt(lineData[1]);
112         break;
113       case "diversify":
114         this.diversifyCounter=Integer.parseInt(lineData[1]);
115         break;
116       case "reduce":
117         this.reduceCounter=Integer.parseInt(lineData[1]);
118         break;
119       case "intensification factor":
```

```
120         this.intensificationFactor=Double.parseDouble(lineData[1]);
121         break;
122       case "initial increment":
123         this.initialIncrement=Double.parseDouble(lineData[1]);
124         break;
125       case "medium term memory":
126         this.mtmSize=Integer.parseInt(lineData[1]);
127         break;
128       case "short term memory":
129         this.stmSize=Integer.parseInt(lineData[1]);
130         break;
131       case "diversify two":
132         this.diversify2=Integer.parseInt(lineData[1]);
133         break;
134       case "diversify stop":
135         this.diversifyStop=Integer.parseInt(lineData[1]);
136         break;
137       default:
138         throw new UnknownInputParameterException("Cannot yet take in that input")
    ;
139     }
140 
141   }
142 
143   /**
144    * Checks that the control parameters makes sense
145    * @return true if they do make sense, false otherwise
146    */
147   public boolean checkValid(){
148     boolean returnVal=true;
149     if (intensifyCounter>=diversifyCounter)
150       returnVal=false;
151     if (reduceCounter<diversifyCounter)
152       returnVal=false;
153     if (intensificationFactor>1 || intensificationFactor<0)
154       returnVal=false;
155     if (initialIncrement<0)
156       returnVal=false;
157     if (diversifyStop<0)
158       returnVal=false;
159 
160     return returnVal;
161   }
162 
163 }
```

coordinate_holder

PairDouble

```
1 /**
2  *
3  */
4 package ex3.coordinate_holder;
5 
6 /**
7  * @author eng-188do
8  * Stores pairs of values which are double
9  */
```

```
public final class PairDouble {
  //Constructors
  public PairDouble(double x, double y) {
    this.x = x;
    this.y = y;
  }

  public PairDouble(PairDouble copy) { //Copy constructor
    this.x=new Double(copy.x);
    this.y=new Double(copy.y);
  }

  //Members
  public double x;
  public double y;

}
```

Vector2

```
/**
 *
 */
package ex3.coordinate_holder;

import ex3.exceptions.AttributeNotCalulatedException;
import ex3.exceptions.FunctionEvalLimitException;
import ex3.loggers.FunctionCallLogger;

/**
 * @author eng-188do
 * Class to store the coordinate data and value
 * Note: this class has a natural ordering that is inconsistent with equals.
 */
public class Vector2 implements Comparable<Vector2> {
  //Constructors
  /**
   * Constructor
   * @param logger : is the logger for making function calls and ensuring we don'
   t go over the limit.
   * @param x : x_1 coord (which we often call x)
   * @param y : x_2 coord (which we often call y)
   */
  public Vector2(FunctionCallLogger logger,double x, double y){
    fn=logger;
    coord= new PairDouble(x,y);
    haveEvaluated=false;
  }

  /**
   * Copy constructor
   * @param copy is the item to copy
   */
  public Vector2(Vector2 copy){
    this.fn=copy.fn; //note same object
    this.coord=new PairDouble(copy.coord);
    this.value=new Double(copy.value);
    this.haveEvaluated= new Boolean(copy.haveEvaluated);
  }


```

```
41
42  //Members
43  protected FunctionCallLogger fn; //records all function calls
44  private PairDouble coord; //coordinates
45  protected double value; // value at coordinates, evaluate last minute to avoid
     unnecessary function evals
46  protected boolean haveEvaluated; //flag to say whether or not we have evaluated
     .
47
48  //Methods
49  /**
50   * Gets the function value at this point
51   * @return function value
52   * @throws FunctionEvalLimitException exception if function eval. limit has
     been met
53   */
54  public double getValue() throws FunctionEvalLimitException {
55    double val;
56    if (haveEvaluated){
57      val=value;
58    } else {
59      val=fn.getValue(getCoordX(), getCoordY());
60      value=val; //store value for future reference
61      haveEvaluated=true;
62    }
63    return val;
64  }
65
66
67  /**
68   *  @return the x coordinate
69   */
70  public double getCoordX(){
71    return coord.x;
72  }
73
74  /**
75   * @return the y coordinate
76   */
77  public double getCoordY(){
78    return coord.y;
79  }
80
81  /**
82   * Method for comparing two Vector2's by their objective function values.
83   * This method assumes that the points values have been calculated by this
     point. if not it will throw an error.
84   */
85  @Override
86    public int compareTo(Vector2 other){
87        // compareTo should return < 0 if this is supposed to be
88        // less than other, > 0 if this is supposed to be greater than
89        // other and 0 if they are supposed to be equal
90        int result;
91    try {
92      result = (this.getValue() < other.getValue()) ? -1 : ((this.getValue() >
     other.getValue()) ? 1: 0);
93    } catch (FunctionEvalLimitException e) { //All the functions should have been
     evaluated by here
94      e.printStackTrace();
```

```
95        throw new AttributeNotCalulatedException("Points values should have been
      calulated by this point.");
96      }
97          return result;
98      }
99
100   /**
101    * Sets new coordinated
102    * @param x : x_1 coord
103    * @param y : x_2 coord
104    */
105   protected void setCoord(double x, double y){
106     coord=new PairDouble(x,y);
107     haveEvaluated=false;
108   }
109
110
111
112 }
```

evolution_strategies_specifics

ESMember

```
1  /**
2   *
3   */
4  package ex3.evolution_strategies_specifics;
5
6  import java.util.Random;
7  import ex3.FunctionDef;
8  import ex3.auxillary_maths_functions.TwoDMatrices;
9  import ex3.control_parameters.ControlParamsES;
10 import ex3.coordinate_holder.Vector2;
11 import ex3.exceptions.FunctionEvalLimitException;
12 import ex3.loggers.FunctionCallLogger;
13
14 /**
15  * @author eng-188do
16  * This class holds a member of the evolutionary strategy population.
17  */
18 public final class ESMember extends Vector2 {
19   //Constructor
20   /**
21    * Constructor
22    * @param logger : function call logger. for getting function value and for
      storing all calls to it
23    * @param x : x coord (x_1 on assignment sheet)
24    * @param y : y coord (x_2 on assignment sheet)
25    * @param seed : random number generator to use
26    * @param cntrlIn : control parameters
27    * @param varsIn : variances for mutation
28    * @param alphasIn : alphas (ie rotation angles) for mutation
29    * @param genIn : generation number
30    */
31   public ESMember(FunctionCallLogger logger,double x, double y, Random seed,
      ControlParamsES cntrlIn, double[] varsIn, double[] alphasIn, int genIn){
32     super( logger, x,  y);
33     rnd=seed;
```

```
34      cntrl=cntrlIn;
35      vars=varsIn;
36      alphas=alphasIn;
37      gen=genIn;
38    }
39
40    /**
41     * Copy Constructor
42     * @param copy : ESMember to copy.
43     */
44    public ESMember(ESMember copy){
45      super(copy); //TODO: check this is working as i am expecting it to.
46
47      this.vars=new double[copy.vars.length];
48      System.arraycopy( copy.vars, 0, this.vars, 0, copy.vars.length);
49
50      this.alphas=new double[copy.alphas.length];
51      System.arraycopy( copy.alphas, 0, this.alphas, 0, copy.alphas.length);
52
53      this.gen=new Integer(copy.gen);
54      this.rnd=copy.rnd; //want reference to same object
55      this.cntrl=copy.cntrl; //want reference to same object
56    }
57
58    //Members
59
60    /* Strategy Parameters, these are protected to allow population to access them
       if it needs to. */
61    protected double[] vars; //variances
62    protected double[] alphas; //define as {axx,axy,ayx,ayy}.
63
64    Random rnd; //random number generaetor
65    private int gen; //generation number
66    private ControlParamsES cntrl; //control parameters
67
68
69    /* (non-Javadoc)
70     * @see ex3.CoordinateHolder.Vector2#getValue()
71     * gets the value at the current coords
72     */
73    @Override
74    public double getValue() throws FunctionEvalLimitException {
75      double val;
76      if (haveEvaluated){
77        val=value;
78      } else {
79        val=fn.getValue(getCoordX(), getCoordY(), gen);
80        value=val; //store value for future reference
81        haveEvaluated=true;
82      }
83      return val;
84    }
85
86    /**
87     * Mutates the member of the population
88     * overloaded method
89     * @see ESMember#mutate(int)
90     */
91    public void mutate(){
92      mutate(0);
93    }
```

```
94
95  /**
96   * Mutates the member of the population
97   * @param counter : records number of times function has been called
98   * Note: calls itself recursively
99   */
100 public void mutate(int counter){
101
102   //mutate strategy parameters
103   mutateStratParams();
104
105    //mutate X
106    double[] c=calcMutateXChange(); //change in coords as the result of the
     mutation
107
108    double newX=getCoordX()+ c[0];
109    double newY=getCoordY()+c[1];
110    //Change x and y
111    if (!FunctionDef.checkInRange(newX, newY)){//if out of range try mutating
     again.
112        if (counter > 100) // I found that the variances could run off with
     themselves and get too big. So I would call this function until stack
     overflow.
113          resetVars();  //So I put this hack in to try to stop that happening.
     Will be discussed in the report.
114       mutate(++counter);
115       return;
116     }
117     setCoord(newX,newY);
118 }
119
120 /**
121  * resets the variances to their initial values
122  */
123 private void resetVars(){
124   for(int i=0; i<vars.length; i++){
125     vars[i]=cntrl.getVar();
126   }
127 }
128
129 /**
130  * @return  the change in coordinates due to mutation
131  */
132 private double[] calcMutateXChange(){
133    //form rotation matrix (see p.70 beck)
134    double[][] rot={ {Math.cos(alphas[0]),-Math.sin(alphas[1])},
135        {Math.sin(alphas[2]),Math.cos(alphas[3])} };
136
137    //form matrix to multiply this with
138    double[][] multiplier={ {vars[0],0},
139                {0,vars[1]} };
140
141    //random vector
142    double[][] randVec={{rnd.nextGaussian()},{rnd.nextGaussian()}};
143
144    //multiply it out
145    double[][] change=TwoDMatrices.matrixMultiplication(rot, multiplier);
146    change=TwoDMatrices.matrixMultiplication(change, randVec);
147    double [] c=TwoDMatrices.convertColVectorToArray(change);
148    return c;
149 }
```

```
150
151   /**
152    * Mutates the strategy parameters  (ie varaiances and alphas) see Evolution
       Strategy Notes for how it does this.
153    */
154   private void mutateStratParams(){
155      double[] n=new double[7]; //array  {N_0, N_1,N_2,N_11,N_12,N_21,N_22}
156
157      //find random numbers for Ns
158      for (int i=0; i<n.length;i++)
159       n[i]=rnd.nextGaussian();
160
161      //mutate variances
162      for (int j=0; j<vars.length; j++){
163        vars[j]*=Math.exp(cntrl.getTauP()*n[0]+cntrl.getTau()*n[j+1]);
164      }
165
166      //mutate alpha's
167      for (int k=0; k<alphas.length; k++){
168        alphas[k]+=cntrl.getBeta()*n[3+k];
169      }
170   }
171
172
173
174 }
```

Population

```
1  /**
2   *
3   */
4  package ex3.evolution_strategies_specifics;
5
6  import java.util.*;
7
8  import ex3.FunctionDef;
9  import ex3.control_parameters.ControlParamsES;
10 import ex3.exceptions.FunctionEvalLimitException;
11 import ex3.loggers.FunctionCallLogger;
12 import ex3.loggers.SurvivingPopLogger;
13
14 /**
15  * @author eng-188do
16  * Class to hold whole population of evolutionary strategy members
17  */
18 public class Population {
19   //Constructor.
20   /**
21    * Constructor
22    * @param rndIn : random number generator
23    * @param fnin : function logger
24    * @param cntrlIn : control parameters
25    * @param logSurvPopIn : logger for the surviving population for each
       generation
26    */
27   public Population(Random rndIn, FunctionCallLogger fnin, ControlParamsES
       cntrlIn, SurvivingPopLogger logSurvPopIn){
28     rnd=rndIn;
29     cntrl=cntrlIn;
30     fn=fnin;
```

```
31      logSurvPop=logSurvPopIn;
32    }
33
34    //Members
35    private List<ESMember> pop= new ArrayList<ESMember>(); //holds population
36    private List<ESMember> mutParents= new ArrayList<ESMember>(); //holds the
       parents after mutation
37    private List<ESMember> children= new ArrayList<ESMember>(); //holds the
       children pre-selection.
38    private ControlParamsES cntrl; //control parameters
39    private Random rnd; //random number generator
40    private FunctionCallLogger fn; //function logger (ensures we do not go over
       evaluation limit).
41    private int gen=0; //stores generation count.
42    private SurvivingPopLogger logSurvPop; //logs the surviving population at each
       generation.
43
44    //Members
45    /**
46     * Mutate the population and put it into the mutated parents list.
47     */
48    public void mutate(){
49      mutParents=Population.cloneList(pop);
50      for(ESMember mem: mutParents){
51        mem.mutate();
52      }
53    }
54
55    /**
56     * Combine the mutated parents to form children, which go into the children
       list.
57     * @throws FunctionEvalLimitException
58     */
59    public void recombineToFormChildren() throws FunctionEvalLimitException{
60      //NOTE I am assuming that this only results in valid solutions - which is the
       case for my problem as valid space is square
61      int numChildToForm=cntrl.getParentChildRatio()*cntrl.getNumParents();
62      children.clear(); //clear old.
63      gen++; //we are creating a new generation
64
65      for(int counter=0;counter<numChildToForm;counter++){
66        //Choose parents:
67        int parent1=genRandomParent();
68        int parent2=genRandomParent();
69
70        //Choose new points:
71        double x=mutParents.get(parent1).getCoordX();
72        double y=mutParents.get(parent2).getCoordY();
73
74        //Choose new strategy params:
75        double[] vars=recombineVars(parent1,parent2);
76        double[] alphas=recombineAlphas(parent1,parent2);
77
78        //Make child!
79        ESMember child=new ESMember( fn, x,  y, rnd, cntrl, vars, alphas, gen);
80        child.getValue(); //make sure that it knows its value by this point.
81        children.add(child);
82      }
83    }
84
85    /**
```

```
86    * We add the children back to the general population. note we leave the
      original (unmutated) population how it is.
87    * This is an elitist scheme.
88    * NOTE NOT CURRENTLY USED
89    */
90   public void addChildrenToPopulationKeepParents(){
91     for (ESMember child:children){
92       pop.add(child);
93     }
94   }
95
96   /**
97    * We add the children back to the general population. and we remove the
      parents
98    *  @see Population#addChildrenToPopulationKeepParents()   elitist version
99    */
100  public void addChildrenToPopulation(){
101    pop.clear();
102    for (ESMember child:children){
103      pop.add(child);
104    }
105  }
106
107
108  /**
109   * We are just going to keep the best lambda of the population
110   */
111  public void assessAndCullPop(){
112    Collections.sort(pop);
113    pop.subList(cntrl.getNumParents(),pop.size()).clear();
114    addToSurvivingPopLog();
115
116  }
117
118
119  /**
120   * Adds the surving population to the correct log.
121   */
122  private void addToSurvivingPopLog(){
123    //And add to the log
124        for(ESMember survivingMember: pop){ //go through population
125          double x,y,val = 0;
126
127          //get coordinates
128          x=survivingMember.getCoordX();
129          y=survivingMember.getCoordY();
130
131          //try getting the value - it should be calculated by this point in the
       program:
132          try {
133            val=survivingMember.getValue();
134          } catch (FunctionEvalLimitException e) { //they should all be
       calculated by now so we will write a message and quit
135            e.printStackTrace();
136            System.out.println("Program isn't working correctly, debug!");
137            System.exit(1); //we're going to exit here as program is not working,
       as designned.
138          }
139          logSurvPop.add(x, y, val, gen);
140        }
141  }
```

```
142
143   /**
144    * Clones a list (this is a deep copy)
145    * @param origList : original list to clone from
146    * @return the cloned list
147    */
148   public static List<ESMember> cloneList(List<ESMember> origList) {
149       List<ESMember> clonedList = new ArrayList<ESMember>(origList.size());
150       for (ESMember mem : origList) { //look through and copy each member in
       original list.
151           clonedList.add(new ESMember(mem));
152       }
153       return clonedList;
154   }
155
156   /**
157    * @return a random parent's index
158    */
159   private int genRandomParent(){
160     int num=rnd.nextInt(mutParents.size());
161     return num;
162   }
163
164
165   /**
166    * Perform recombination on strategy parameters by returning average of parents
        (intermediate recombination).
167    * @param strat1 strategy parameters of parent 1
168    * @param strat2 strategy parameters of parent 2
169    * @return the new strategy parameters of the child.
170    */
171   private double[] recombineStrategy(double[] strat1, double[] strat2){
172     double[] stratNew=new double[strat1.length];
173
174     for(int i=0;i<strat1.length;i++){
175       stratNew[i]=0.5*(strat1[i]+strat2[i]);
176     }
177     return stratNew;
178   }
179
180
181   /**
182    * Recombination strategy on the variances
183    * @param parent1 index of parent 1
184    * @param parent2 index of parent 2
185    * @return  the new variance
186    */
187   private double[] recombineVars(int parent1, int parent2){
188     double[] vars1, vars2;
189     vars1=mutParents.get(parent1).vars;
190     vars2=mutParents.get(parent2).vars;
191
192     return recombineStrategy(vars1, vars2)  ; //(intermediate recombination)
193   }
194
195   /**
196    * Recombination strategy on the alphas
197    * @param parent1 index of parent 1
198    * @param parent2 index of parent 2
199    * @return  the new alpha
200    */
```

```
201   private double[] recombineAlphas(int parent1, int parent2){
202     double[] alpha1, alpha2;
203     alpha1=mutParents.get(parent1).alphas;
204     alpha2=mutParents.get(parent2).alphas;
205 
206     return recombineStrategy(alpha1, alpha2); //(intermediate recombination)
207   }
208 
209   /**
210    * Initialised the random population. X's are chosen randomly. Strategy
       parameters are defined deterministically.
211    * see my report for discussion on this point.
212    */
213   public void initialisePopulation(){
214     pop.clear();
215 
216     double x,y;
217     double vars[]={cntrl.getVar(), cntrl.getVar()}; //start covariance functions
       are defined.
218     double alphas[]={1,0,0,1}; //deterministic and no covariance to start off
       with.
219 
220     for (int i=0; i<cntrl.getNumParents(); i++){
221       x=FunctionDef.genRandomX(rnd);
222       y=FunctionDef.genRandomY(rnd);
223       pop.add(new ESMember(fn, x, y, rnd, cntrl, vars, alphas, 0));
224     }
225 
226     addToSurvivingPopLog();
227   }
228 }
```

RunES

```
1  /**
2   *
3   */
4  package ex3.evolution_strategies_specifics;
5 
6  import java.util.*;
7 
8  import ex3.RunOptimisationInterface;
9  import ex3.control_parameters.ControlParamsES;
10 import ex3.exceptions.FunctionEvalLimitException;
11 import ex3.loggers.FunctionCallLogger;
12 import ex3.loggers.SurvivingPopLogger;
13 
14 
15 
16 /**
17  * @author eng-188do
18  * Class runs the evolution strategy program.
19  */
20 public class RunES implements RunOptimisationInterface {
21 
22 
23 
24   /* (non-Javadoc)
25    * @see ex3.RunOptimisation#run(java.lang.Long, java.lang.String, java.lang.
       String)
26    */
```

```
27   @Override
28   public void run(Long seed, String inputFile, String outputFile) {
29     //Set up loggers
30     FunctionCallLogger log=new FunctionCallLogger(1000); //only 1000 function
      calls allowed
31     SurvivingPopLogger logSurvPop=new SurvivingPopLogger();
32
33     //set up then read in control parameters
34     ControlParamsES cntrl=new ControlParamsES();
35     cntrl.readInFile(inputFile);
36
37     //set up random seed
38     Random rnd=new Random(seed);
39
40     //set up population
41     Population population=new Population(rnd, log, cntrl,logSurvPop);
42
43     //run optimisation
44     try{
45       runOptimisation(population);
46     } catch (FunctionEvalLimitException e){//we've met our allowed func evals
47       //print out results:
48       log.printResults(outputFile);
49       logSurvPop.printResults(outputFile);
50     }
51   }
52
53   private void runOptimisation(Population pop) throws FunctionEvalLimitException{
54     pop.initialisePopulation();
55       while (true){ //go through cycles
56         pop.mutate(); //mutate
57         pop.recombineToFormChildren(); //recombine to generate offspring
58         pop.addChildrenToPopulation(); //add these children to the population
59         //pop.addChildrenToPopulationKeepParents(); //add these children to the
      population (elitist)
60         pop.assessAndCullPop(); //assess and calualte the survivors of the
      population.
61       }
62   }
63
64
65
66
67 }
```

exceptions

AttributeNotCalulatedException

```
1 /**
2  *
3  */
4 package ex3.exceptions;
5
6 /**
7  * @author eng-188do
8  * Exception thrown when attribute not found.
9  * It is a runtime exception so signifies a problem of my 9the programmer's fault
     ). I have used this mostly to signal
```

```
 * when the value of a point has not been calculated when it should have been already and we have run out of function eavluations to
 * evaluate it.
 */
public class AttributeNotCalulatedException extends RuntimeException {
  public AttributeNotCalulatedException(){super();};
  public AttributeNotCalulatedException(String message){super( message);};
  public AttributeNotCalulatedException(String message, Throwable cause){super( message, cause);};
  public AttributeNotCalulatedException(Throwable cause){super( cause);};
}
```

UnknownInputParameterException

```
/**
 *
 */
package ex3.exceptions;

import javax.lang.model.UnknownEntityException;

/**
 * @author eng-188do
 * Exception thrown when an unknown input parameter is entered.
 */
public class UnknownInputParameterException extends UnknownEntityException {
  public UnknownInputParameterException(String message){ super(message);};
}
```

FunctionEvalLimitException

```
/**
 *
 */
package ex3.exceptions;

/**
 * @author eng-188do
 * Exception for when the limit on the number of function evaluations is met.
 * Use this to easily meet the limit of 1000 function evaluations given in the assignment.
 */
public class FunctionEvalLimitException extends Exception {
  public FunctionEvalLimitException() { super(); }
  public FunctionEvalLimitException(String message) { super(message); }
  public FunctionEvalLimitException(String message, Throwable cause) { super( message, cause); }
  public FunctionEvalLimitException(Throwable cause) { super(cause); }

}
```

file_operations

FileArrayWriter

```
/**
 *
```

```
 */
package ex3.file_operations;

import java.io.BufferedWriter;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStreamWriter;
import java.io.Writer;
import java.util.List;

/**
 * @author eng-188do
 * Class to conveniently hold the code for outputting an array into a csv file.
 */
public class FileArrayWriter {
  /**
   * Print out array information to a file csv format.
   * @param distances distances to print out
   * @param location which file to output to
   */
  public static void printOutArray(double[][] array, String location){
    Writer writer = null;
    location+=".csv";
    try {
        writer = new BufferedWriter(new OutputStreamWriter(
              new FileOutputStream( location), "utf-8"));
        for(int i=0; i<array.length;i++){ //print out "i, $arrayValue"
          String line="";
          for(double item: array[i]){
            line+=Double.toString(item) +",";
          }
          writer.write(line +"\n");
        }

    } catch (IOException e) {
      e.printStackTrace();
    } finally { //now close the file
       try {
         writer.close();
       } catch (Exception e) {
        e.printStackTrace();
       }
    }
  }

  /**
   * converts a list of fixed size double arrays to an fixed multi-dimensional
   double array.
   * @param log : this is a list of double arrays (which should all be the same
   size (although I don't currently check this)
   * @return the list in a 2d double array format.
   */
  public static double[][] convertListToDouble(List<double[]> log){
    double[][] logArray = new double[log.size()][log.get(0).length];
    for(int i=0; i<log.size();i++){
      logArray[i]=log.get(i);
    }
    return logArray;
  }


```

```
  /*static void printOutArray(ArrayList<double[]> array, String location){
   //TODO:remove this dupliacte code
    Writer writer = null;
    location+=".csv";
    try {
        writer = new BufferedWriter(new OutputStreamWriter(
              new FileOutputStream( location), "utf-8"));
        for(int i=0; i<array.size();i++){
          String line="";
          for(double item: array.get(i)){
            line+=Double.toString(item) +",";
          }
          writer.write(line +"\n");
        }

    } catch (IOException e) {
      e.printStackTrace();
    } finally { //now close the file
       try {
         writer.close();
       } catch (Exception e) {
        e.printStackTrace();
       }
    }
  }*/


}
```

loggers

BirdLogger

```
/**
 *
 */
package ex3.loggers;

import java.util.ArrayList;
import java.util.List;

import ex3.file_operations.FileArrayWriter;

/**
 * @author eng-188do
 * Class for storing the Particle Swarm Optimisation results.
 */
public final class BirdLogger implements LoggerInterface {
  //Members
  private List<double[]> birds=new ArrayList<double[]>(); //log of all the birds
        //it will be a double of length 7:
        // with birdnum,x coord, y coord, value, x velocity, y velocity, counter


  //Methods
  /**
   * Adds bird to log
   * @param birdNum unique identifier of bird
   * @param coords birds coordinates {x,y}
```

```
27     * @param val the bird's value
28     * @param vel the birds velocity {xvel,yvel}
29     * @param birdIndividualCounter how many steps has the bird in question taken.
30     */
31    public void addMember(int birdNum, double[] coords, double val, double[] vel,
       int birdIndividualCounter){
32     //add values raqther than bird as easier to make sure we aren't taking a
       reference
33     //convert to double if applicable
34     double birdNumDoub=(double)birdNum;
35     double birdIndividualCounterDoub=(double) birdIndividualCounter;
36
37     //group up into array
38     double[] input={birdNumDoub, coords[0], coords[1], val, vel[0], vel[1],
       birdIndividualCounterDoub};
39
40     //add to list
41     birds.add(input);
42    }
43
44    /* (non-Javadoc)
45     * @see ex3.Loggers.Logger#printResults(java.lang.String)
46     */
47    @Override
48    public void printResults(String outputFile) {
49      outputFile+="_BirdLogger";
50     FileArrayWriter.printOutArray(FileArrayWriter.convertListToDouble(birds),
       outputFile);
51    }
52
53 }
```

LoggerInterface

```
1 /**
2  *
3  */
4 package ex3.loggers;
5
6 /**
7  * @author eng-188do
8  * Interface for the logger classes. I want to ensure that they all print out in
     the same way.
9  */
10 @FunctionalInterface
11 public interface LoggerInterface {
12   /**
13    * Print out log to file specified.
14    * @param outputFile : name stub of file to output to (can append to it the
       name of the logger).
15    */
16   public  void printResults(String outputFile);
17 }
```

TabuLogger

```
1 /**
2  *
3  */
4 package ex3.loggers;
```

```
5
6  import ex3.file_operations.FileArrayWriter;
7
8  import java.util.*;
9
10 /**
11  * @author eng-188do
12  *  This class logs the base points in the Tabu search
13  */
14 public class TabuLogger implements LoggerInterface {
15
16   //Members
17   private List<double[]> basePts=new ArrayList<double[]>(); //log of all the base
       points
18         //this will be a double of length 7:
19         // with x,y,val,globalCounter,counter, increment, pattern move (boolean)
20
21
22   //Methods
23   /**
24    * Adds base point to list
25    * @param x value : x coord (x_1)
26    * @param y value : y coord (x_2)
27    * @param val : (f(x,y))
28    * @param globalCounter which base point is it
29    * @param counter which stage of the local search are we on
30    * @param increment how much local search is moving
31    * @param patternMove was this base point the result of a pattern move?
32    */
33   public void addMember(double x, double y, double val, int globalCounter, int
      counter, double increment, boolean patternMove){
34     //convert to double if applicable
35     double globalCounterDoub=(double)globalCounter;
36     double counterDoub=(double) counter;
37     double patternMoveDoub = patternMove ? 1.0 : 0.0;
38
39     //group up into array
40     double[] input={x, y, val, globalCounterDoub, counterDoub, increment,
      patternMoveDoub};
41
42     //add to list
43     basePts.add(input);
44   }
45
46   /* (non-Javadoc)
47    * @see ex3.Loggers.Logger#printResults(java.lang.String)
48    */
49   @Override
50   public void printResults(String outputFile) {
51     outputFile+="_TabuSearchLogger";
52     FileArrayWriter.printOutArray(FileArrayWriter.convertListToDouble(basePts),
      outputFile);
53   }
54
55 }
```

FunctionCallLogger

```
1 /**
2  *
3  */
```

```java
package ex3.loggers;

import ex3.FunctionDef;
import ex3.exceptions.FunctionEvalLimitException;
import ex3.file_operations.FileArrayWriter;

/**
 * @author eng-188do
 * Holds all the function calls
 */
public class FunctionCallLogger implements LoggerInterface {
  //Constructor
  /**
   * Constructor
   * @param maxNumCallsIn : is the maximum number of calls allowed.
   */
  public FunctionCallLogger(int maxNumCallsIn){
    maxNumCalls=maxNumCallsIn-1; //-1 is just as arrays start at 0 index
    vals= new double[maxNumCallsIn][4]; //1: x, 2: y, 3: value, 4: generation (if applicable)
  }

  //Members
  private int counter=-1; //for arrays starting at 0 index.
  private final int maxNumCalls; //maximum number of calls allowed
  private double[][] vals; //stores the log results


  //Methods
  /**
   * Gets the fucntion's value and adds it to the log.
   * Overloaded version of
   * @see FunctionCallLogger#getValue(double, double, int)
   * @param x : x coord (x_1)
   * @param y : y coord (x_2)
   * @param gen : generation number (only applicable to ES)
   * @return the function's vallue
   * @throws FunctionEvalLimitException : thrown when the function evaluation limit is met.
   */
  public double getValue(double x, double y, int gen) throws FunctionEvalLimitException {
    //NOTE: as we are only calculating at max 1000 values perhaps for ones we have already calculated we can just look it up
    //will discuss this in the report.

    double val=getValue(x, y); //Get value and put other items in memory.

    //Put generation into memory
    vals[counter][3]=gen;

    return val;
  }


  /**
   * prints out the log
   */
  public  void printResults(String outputFile) {
    outputFile+="_FunctionCallLogger";
    FileArrayWriter.printOutArray(vals, outputFile);
```

```
61
62    }
63
64    /**
65     * Overloaded version of getValue
66     * @see FunctionCallLogger#getValue(double, double, int)
67     */
68    public double getValue(double x, double y) throws FunctionEvalLimitException {
69      counterIncrement();
70
71      //calculate values
72      double val=FunctionDef.evalFunc(x, y);
73
74      //put values in memory
75      vals[counter][0]=x;
76      vals[counter][1]=y;
77      vals[counter][2]=val;
78      return val;
79    }
80
81    /**
82     * Increment the counter
83     * @throws FunctionEvalLimitException if we have used up our allowed function
       evaluations an exception is thrown.
84     */
85    private void counterIncrement() throws FunctionEvalLimitException {
86      if (++counter>maxNumCalls)
87        throw new FunctionEvalLimitException();
88    }
89
90
91
92
93
94 }
```

SurvivingPopLogger

```
1  /**
2   *
3   */
4  package ex3.loggers;
5
6  import java.util.*;
7
8  import ex3.file_operations.FileArrayWriter;
9
10 /**
11  * @author eng-188do
12  * class that logs the surviving population at each generation
13  */
14 public class SurvivingPopLogger implements LoggerInterface {
15
16   //Members
17   private List<double[]> log=new ArrayList<double[]>(); //arrays will be 4 by 4.
18
19
20   //Methods
21   /**
22    * Adds point to log
23    * @param x : x coord (x_1)
```

```
24     * @param y : y coord (x_2)
25     * @param val : value found at those coordinates
26     * @param gen : generation number of the point.
27     */
28    public void add(double x, double y, double val, int gen){
29      double genDoub=(double) gen;
30      double[] listEntry={x,y,val,genDoub};
31      log.add(listEntry);
32    }
33
34    /* (non-Javadoc)
35     * @see ex3.Logger#printResults(java.lang.String)
36     */
37    @Override
38    public void printResults(String outputFile) {
39      outputFile+="_SurvivingPopulationLogger";
40      FileArrayWriter.printOutArray(FileArrayWriter.convertListToDouble(log),
       outputFile);
41
42    }
43
44
45
46 }
```

swarm_specifics

Bird

```
1  /**
2   *
3   */
4  package ex3.swarm_specifics;
5
6  import java.util.*;
7
8  import ex3.FunctionDef;
9  import ex3.control_parameters.ControlParamsS;
10 import ex3.coordinate_holder.Vector2;
11 import ex3.exceptions.AttributeNotCalulatedException;
12 import ex3.exceptions.FunctionEvalLimitException;
13 import ex3.loggers.BirdLogger;
14 import ex3.loggers.FunctionCallLogger;
15
16 /**
17  * @author eng-188do
18  * Class to act as a bird in the swarm optimisation.
19  */
20 public final class Bird extends Vector2 {
21   //Constructor
22   /**
23    * Constructor
24    * @param logger : function call logger
25    * @param x : x coord (x_1)
26    * @param y : y coord (x_2 on assignment sheet)
27    * @param rndIn : random number generator
28    * @param logIn : log for the birds
29    * @param cntrlIn : control parameters class
```

```
30    * @param birdIdIn : bird's id, this should be unique, and is used for tracking
       birds in the log.
31    */
32   Bird(FunctionCallLogger logger,double x, double y, Random rndIn, BirdLogger
       logIn, ControlParamsS cntrlIn, int birdIdIn){
33     super(logger, x, y);
34     this.rnd=rndIn; //copy ref
35     this.log=logIn;
36     this.cntrl=cntrlIn;
37     this.birdId=birdIdIn;
38   }
39
40   /**
41    * Copy constructor
42    * @param copy : bird we are taking a copy of.
43    */
44   Bird(Bird copy){
45     super(copy);
46     this.counter=copy.counter;
47     this.bestCoords=new Vector2(copy.bestCoords);
48     this.worstCoords=new Vector2(worstCoords);
49     this.offEdgeFlag=copy.offEdgeFlag;
50     System.arraycopy(copy.velocity, 0, this.velocity, 0, copy.velocity.length);
       //performs shallow copy (but on double so effectively deep).
51     this.rnd=copy.rnd; //copy ref
52     this.log=copy.log;
53     this.cntrl=copy.cntrl;
54     this.birdId=copy.birdId;
55   }
56
57   //Members
58   private int counter=0; //measures how many steps the bird has taken.
59   private Vector2 bestCoords; //measures the best coordinates witnessed
       personally by this bird.
60   private Vector2 worstCoords; //measures the worst coordinates witnessed
       personally by this bird.
61   private boolean offEdgeFlag; //flag set to true when inside the allowable
       region and not on the edge.
62   private double[] velocity=new double[2]; //{xvelocity,yvelocity}
63   Random rnd; //random number generator
64   BirdLogger log; //log for birds.
65   ControlParamsS cntrl; //control parameters for birds.
66   public final int birdId; //stores the unique identifier of the bird.
67
68   //Methods
69   /**
70    * Initialise all the bird fields.
71    * position is random, velocity is as set in the control parameters
72    * @throws FunctionEvalLimitException
73    */
74   public void initialise() throws FunctionEvalLimitException{
75     //generate random x's and y's
76     double x=FunctionDef.genRandomX(rnd);
77     double y=FunctionDef.genRandomY(rnd);
78     offEdgeFlag=true; //should have generated it inside the region
79     assert FunctionDef.checkInRange(x, y) :
80       "Generating initial birds outside allowable region"; //check in debug mode
       that working as expected.
81
82     //set the point and make sure the value has been calculated
83     this.setCoord(x, y); //set position
```

```
84      this.getValue(); //make sure velocity is calulated
85      bestCoords=new Vector2(this); //put a copy into best point
86      worstCoords=new Vector2(this); //puty a copy into worst point
87      velocity[0]=cntrl.getInitialXVel(); //set up velocities.
88      velocity[1]=cntrl.getInitialYvel();
89
90      //finally chuck the entry into the log.
91      addToLog();
92    }
93
94
95    /**
96     * Updates the bird by calculating its velocity and then applying it.
97     * @param otherBirds : list of other birds so we can find the best one.
98     * @throws FunctionEvalLimitException
99     */
100   public void updateBird(ArrayList<Bird> otherBirds) throws
       FunctionEvalLimitException{
101     updateVel(otherBirds);
102
103     //find and set the new coordinates
104     double[] newCoords=updateCoords();
105     this.setCoord(newCoords[0], newCoords[1]);
106
107     //update counter and put in log if inside.
108     counter++;
109     assert FunctionDef.checkInRange(this.getCoordX(), this.getCoordY()) : "birds
       have escaped!";
110     this.getValue();//make sure if we adding to log we have value.
111     addToLog();
112
113     //update best and worst positions if applicable
114     if (this.getValue()<bestCoords.getValue())
115       bestCoords=new Vector2(this); //assign a copy
116     if (this.getValue()>worstCoords.getValue())
117       worstCoords=new Vector2(this);
118
119
120   }
121
122
123   /**
124    * Updates the coordinates by adding the velocities to each coordinate.
125    * If the bird was going to stray off the region it is brought back and
       positioned on the edge with the relevant values set.
126    * the velocity in that direction is set to 0.
127    * @return : new coordinates
128    */
129   private  double[] updateCoords(){
130     double x=this.getCoordX()+velocity[0]; //set out where we want to go if no
       boundaries
131     double y=this.getCoordY()+velocity[1];
132
133     offEdgeFlag=true; //assume for now we are off the edge - will correct later
       if not.
134     x=setInRange(x, FunctionDef.xlow, FunctionDef.xhigh, 0); //check whether
       strayed over x boundary
135     y=setInRange(y, FunctionDef.ylow, FunctionDef.yhigh, 1); //check whether
       strayed over y boundary
136
137     double[] newCoordinates={x,y};
```

```
138     return newCoordinates;
139   }
140
141   /**
142    * Checks a particular coordinate for in range. if it isn't then it sets it to
       boundary and the velocity to 0.
143    * @param value : coordinates current value.
144    * @param lowBound : the coordinate's lower bound
145    * @param highBound : the coordinate's upper bound
146    * @param index : the index of the coordinate
147    * @return the coordinate
148    */
149   private double setInRange(double value, double lowBound, double highBound, int
       index){
150     if(value<lowBound) { //check whether strayed over y boundary.
151       value=lowBound;
152       offEdgeFlag=false;
153       velocity[index]=0;
154     } else if (value>highBound){
155       value=highBound;
156       offEdgeFlag=false;
157       velocity[index]=0;
158     }
159     return value;
160   }
161
162   /**
163    * Updates the velocity of the bird
164    * @param otherBirds : list of the other birds in the flock.
165    * @throws FunctionEvalLimitException
166    */
167   private void updateVel(ArrayList<Bird> otherBirds) throws
       FunctionEvalLimitException {
168     //get the best point's  (of the whole flock).
169     double[] bestCoords=findBestBirdCoords( otherBirds);
170
171     //get the neighbour's bird best point coords
172     double[] neighbourCoord=neighbourCoords(otherBirds);
173
174     setVelComponent(0,bestCoords[0],neighbourCoord[0],this.bestCoords.getCoordX()
       ,this.worstCoords.getCoordX(), this.getCoordX()); //set x velocity
175     setVelComponent(1,bestCoords[1],neighbourCoord[1],this.bestCoords.getCoordY()
       ,this.worstCoords.getCoordY(), this.getCoordY()); //set y velocity
176
177   }
178
179
180
181
182   /**
183    * Sets a particular velocity component
184    * @param index : index of the velocity component we are setting. signifies the
       coordinate of interest.
185    * @param bestCoord : the corresponding position coordinate of the best point
       found by the rest of the birds in the flock.
186    * @param bestNeighbourCoord : the coordinate corresponding to the best point
       found by this bird's neighbour
187    * @param bestSelfCoord : the coordinate corresponding to the best point found
       by this bird
188    * @param worstSelfCoord : the coordinate corresponding to the worst point
       found by this bird
```

```
189     * @param currentCoord : the current value of the coordinate of interest for
        this bird.
190     */
191    private void setVelComponent(int index, double bestCoord, double
        bestNeighbourCoord, double bestSelfCoord,
192        double worstSelfCoord, double currentCoord) {
193
194     double currentVel=velocity[index];
195     currentVel*=cntrl.getStiffness(); //scale by stiffness
196
197     if (!offEdgeFlag){ //if outside the allowed area push it back towards by an
        amount proportional to how far it is outside
198        currentVel=0.1*(FunctionDef.centre[index]-currentCoord); //kickstart it in
        the right direction.
199     } else{ //otherwise continue by adding different factors.
200        currentVel+=cntrl.getsSocial()*(bestCoord-currentCoord); //head towards the
        best point of the rest of the flock
201        currentVel+=cntrl.getsNeighbour()*(bestNeighbourCoord-currentCoord); //head
        towards the neighbours's best point.
202        currentVel+=cntrl.getsCognition()*(bestSelfCoord-currentCoord); //head
        towards own best point
203        currentVel+=-cntrl.getsAvoid()*(worstSelfCoord-currentCoord); //head away
        from own worst point
204        currentVel+=cntrl.getCraziness()*rnd.nextGaussian(); //add some craziness
205     }
206
207     velocity[index]=currentVel; //update velocity
208    }
209
210    /**
211     * Finds the best point found by the rest of the flock.
212     * @param otherBirds : the rest of the flock.
213     * @return : best point found by the rest of the flock. (copy of coordinates)
214     * @throws FunctionEvalLimitException
215     */
216    private double[] findBestBirdCoords(ArrayList<Bird> otherBirds) throws
        FunctionEvalLimitException{
217     double min=otherBirds.get(0).bestCoords.getValue(); //set minimum as the
        first bird.
218     double[] coords={otherBirds.get(0).bestCoords.getCoordX(),otherBirds.get(0).
        bestCoords.getCoordY()};
219     double[] coordsOut=new double[2]; //array for passing out - make sure we pass
        out a deep copy not the value to the reference.
220
221     for(Bird eachBird: otherBirds){
222        if(min>eachBird.bestCoords.getValue()){ //if this bird has a better value
        than the other then set it as the minimum.
223           min=eachBird.bestCoords.getValue();
224           coords=new double[2];
225           coords[0]=eachBird.bestCoords.getCoordX();
226           coords[1]=eachBird.bestCoords.getCoordY();
227        }
228     }
229
230     System.arraycopy(coords, 0, coordsOut, 0, coords.length);
231     return coords;
232    }
233
234    /**
235     * Loops through the birds and finds the one currently closest.
236     * @param otherBirds : rest of flock.
```

```
237     * @return :  the coordinates of the best point found by the neighbouring bird.
238     */
239    private double[] neighbourCoords(ArrayList<Bird> otherBirds) {
240      //initially set the first bird as the closest.
241      double closest=calcDistanceToBird(otherBirds.get(0));
242      double[] coords={otherBirds.get(0).bestCoords.getCoordX(),otherBirds.get(0).
       bestCoords.getCoordY()};
243
244      for(Bird eachBird: otherBirds){
245        if (calcDistanceToBird(eachBird)<closest){ //if closer set this as the new
       closest bird.
246          coords[0]=eachBird.bestCoords.getCoordX();
247          coords[1]=eachBird.bestCoords.getCoordY();
248          closest=calcDistanceToBird(eachBird);
249        }
250      }
251      return coords;
252    }
253
254    /**
255     * Adds the bird to the bird logger.
256     */
257    private void addToLog(){
258      try {
259        double[] coords={this.getCoordX(), this.getCoordY()};
260        double val = this.getValue();
261        log.addMember(birdId, coords, val, velocity, counter);
262      } catch (FunctionEvalLimitException e) { //should have calculated value by
       this point. so my problem if exception.
263        e.printStackTrace();
264        throw new AttributeNotCalulatedException();
265      }
266    }
267
268    /**
269     * Calculates the distance from this bird to another.
270     * @param other : the 'another' bird.
271     * @return : the euclidean distance between the two birds.
272     */
273    private double calcDistanceToBird(Bird other){
274      //get other bird's coordinates
275      double xOther=other.getCoordX();
276      double yOther=other.getCoordY();
277
278      //calculate the changes in x and y
279      double xDiff=this.getCoordX()-xOther;
280      double yDiff=this.getCoordY()-yOther;
281
282      double squaredDist=Math.pow(xDiff, 2)+Math.pow(yDiff, 2);
283      return Math.sqrt(squaredDist);
284    }
285
286 }
```

Flock

```
1 /**
2  *
3  */
4 package ex3.swarm_specifics;
5
```

```
6
7  import java.util.*;
8
9  import ex3.control_parameters.ControlParamsS;
10 import ex3.exceptions.FunctionEvalLimitException;
11 import ex3.loggers.*;
12
13 /**
14  * @author eng-188do
15  * Class to hold the flock details.
16  */
17 public class Flock {
18   //Constructor.
19   /**
20    * Constructor
21    * @param rndIn : random number generator
22    * @param cntrlIn : control parameters
23    * @param logIn : logger for the birds.
24    * @param fnIn : logger for function and gateway for evaluating functions
      ensuring that the evaluation limit is adhered to.
25    */
26   public Flock(Random rndIn, ControlParamsS cntrlIn, BirdLogger logIn,
      FunctionCallLogger fnIn){
27     rnd=rndIn;
28     cntrl=cntrlIn;
29     log=logIn;
30     fn=fnIn;
31     flockSize=cntrl.getFlockSize();
32   }
33
34   //Members
35   private final int flockSize; //number of birds
36   private Random rnd; //random number generator
37   private ControlParamsS cntrl; //control parameters
38   private BirdLogger log; //log for bird positions
39   private FunctionCallLogger fn; //function logger and caller.
40
41   private List<Bird> flock=new ArrayList<Bird>(); //for storing the flock.
42
43   //Methods
44   /**
45    * Initialising each bird randomly in the space and with the set velocities.
46    * @throws FunctionEvalLimitException
47    */
48   public void initialiseFlock() throws FunctionEvalLimitException{
49     for(int i=0 ; i<flockSize; i++){
50       flock.add(new Bird(fn,0,0,rnd,log,cntrl,i)); //add bird to flock.
51       flock.get(i).initialise(); //Initialise the bird randomly.
52     }
53   }
54
55   /**
56    * Updates the whole flock by getting them to move on one iteration.
57    * @throws FunctionEvalLimitException
58    */
59   public void updateFlock() throws FunctionEvalLimitException{
60     ArrayList<Bird> otherBirds=new ArrayList<Bird>();
61     for(Bird eachBird: flock){ //go through each bird and update it
62       otherBirds=shallowCopyFlock();
63       otherBirds.remove(eachBird);
64       eachBird.updateBird(otherBirds);
```

```
65      }
66    }
67
68    /**
69     * Performs a shallow copy of the flock (shallow in the sense that the
      references to the objects in the list remain constant)
70     * @return
71     */
72    private ArrayList<Bird> shallowCopyFlock(){
73      ArrayList<Bird> flockCopy=new ArrayList<Bird>();
74      for(Bird eachBird: flock){ //go through each bird and also put it in the new
      list.
75         flockCopy.add(eachBird);
76      }
77      return flockCopy;
78    }
79
80  }
```

RunSwarm

```
1  /**
2   *
3   */
4  package ex3.swarm_specifics;
5
6  import java.util.Random;
7
8  import ex3.RunOptimisationInterface;
9  import ex3.control_parameters.ControlParamsS;
10 import ex3.exceptions.FunctionEvalLimitException;
11 import ex3.loggers.*;
12
13
14 /**
15  * @author eng-188do
16  * Class to run the swarm optimisation algorithm.
17  */
18 public class RunSwarm implements RunOptimisationInterface {
19
20   /* (non-Javadoc)
21    * @see ex3.RunOptimisationInterface#run(java.lang.Long, java.lang.String, java
      .lang.String)
22    */
23   @Override
24   public void run(Long seed, String inputFile, String outputFile) {
25     //Set up loggers
26     FunctionCallLogger log=new FunctionCallLogger(1000); //only 1000 function
      calls allowed
27     BirdLogger logBird=new BirdLogger();
28
29     //set up then read in control parameters
30     ControlParamsS cntrl=new ControlParamsS();
31     cntrl.readInFile(inputFile);
32
33     //set up random seed
34     Random rnd=new Random(seed);
35
36     //set up flock
37     Flock flock=new Flock(rnd,cntrl,logBird,log);
38
```

```
39      //run optimisation
40      try{
41        runOptimisation(flock);
42      } catch (FunctionEvalLimitException e){//we've met our allowed func evals
43        //print out results:
44        log.printResults(outputFile);
45        logBird.printResults(outputFile);
46      }
47
48    }
49
50    private void runOptimisation(Flock flock) throws FunctionEvalLimitException{
51      flock.initialiseFlock(); //initialise randomly in space
52        while (true){ //go through cycles
53          flock.updateFlock(); //perform one iteration - update all points and
      velocities.
54        }
55    }
56
57 }
```

tabu_search_specifics

LocalSearch

```
1  /**
2   *
3   */
4  package ex3.tabu_search_specifics;
5
6  import ex3.FunctionDef;
7  import ex3.control_parameters.ControlParamsTS;
8  import ex3.coordinate_holder.Vector2;
9  import ex3.exceptions.AttributeNotCalulatedException;
10 import ex3.exceptions.FunctionEvalLimitException;
11 import ex3.loggers.*;
12
13 import java.util.*;
14
15 /**
16  * @author eng-188do
17  * Class to act as an object performing the tabu search
18  */
19 public final class LocalSearch {
20   /**
21    * Constructor
22    * @param rndIn : random number generator
23    * @param cntrlIn : control parameters
24    * @param fnIn : function logger and caller (ensures function evaluation limit
      is met)
25    * @param tabLoggerIn : tabu search logger records the progress of the base
      points.
26    */
27   public LocalSearch(Random rndIn, ControlParamsTS cntrlIn, FunctionCallLogger
      fnIn, TabuLogger tabLoggerIn){
28     rnd=rndIn;
29     cntrl=cntrlIn;
30     fn=fnIn;
31     tabLogger=tabLoggerIn;
```

```
32      mem=new Memory(rnd,tabLogger,cntrl);  //set up the memories
33    }
34
35    //Members
36    private Memory mem; //stores all the memories (ie short term, medium term and
       long term)
37    private Random rnd; //random number generator
38    private ControlParamsTS cntrl; //control parameters
39    private FunctionCallLogger fn; //function logger and evaluator
40    private TabuLogger tabLogger; //tabu logger - stores base points.
41
42    private int globalCounter=0; //this increases each time we start a new
       diversification/reduction etc to help with plotting.
43    private int counter=0; //this is COUNTER on page TS4 of the notes
44    private Vector2 basePoint; //this stores the current base point
45    private List<Vector2> potentialPoints=new ArrayList<Vector2>(); //stores
       potential points we are currently looking into.
46    private double increment; //this is how much we look in each direction.
47
48
49    //Methods
50    /**
51     * Initialise the search in some random location.
52     * @throws FunctionEvalLimitException if the number of function evaluations is
       over the limit.
53     */
54    public void initialise() throws FunctionEvalLimitException{
55      basePoint=new Vector2(fn, FunctionDef.genRandomX(rnd),FunctionDef.genRandomY(
       rnd)); //generate randomly in the space.
56      addBasePointToLogAndMem(basePoint, false); //adds to log and memory.
57      increment=cntrl.getInitialIncrement(); //initial increment
58    }
59
60    /**
61     * Performs one iteration of local search.
62     * @throws FunctionEvalLimitException
63     */
64    public void performOneIterLocalSearch() throws FunctionEvalLimitException{
65      double currentBest=mem.getBstSolution(); //current best solution in the
       medium term memory
66
67      if (counter==cntrl.getIntensifyCounter()) {
68        globalCounter++;
69        intensify(); //go to average best location, counter increases by 1
70      }
71      if(   globalCounter<cntrl.getDiversifyStop() && //only diversify if still
       doing it.
72           ( counter==cntrl.getDiversifyCounter() || counter==cntrl.getDiversify2()
       ) //and we have reached one of the two counters.
73                          ) {
74        globalCounter++;
75        diversify();  //go to new area to search, counter increases by 1
76      }
77      if(counter==cntrl.getReduceCounter()){
78        globalCounter++;
79        reduce(); //reduce increment and reset counter.
80      }
81
82
83      populatePotentialMoves(); //assess four points around base point.
84
```

```
85      //make the best move (+pattern move if applicable)
86      Vector2 newMove = makeBestPotentialMove();
87      if (    newMove.getValue()<=basePoint.getValue()){  //note we change newMove
        here.
88         tryPatternMove(newMove);//if move was lower, we come here, so try pattern
        move
89      } else{ //if not add it to the base points and move on.
90         basePoint=newMove;
91         addBasePointToLogAndMem(basePoint,false);
92      }
93
94
95      if (basePoint.getValue()>currentBest) //if we haven't found a new better
        solution then the regular counter goes up.
96         counter++;
97
98    }
99
100   /**
101    * Reduces the increment size and resets the counter.
102    */
103   private void reduce(){
104     increment*=cntrl.getIntensificationFactor();
105     counter=0;
106   }
107
108   /**
109    * Intensifies the search by moving the base point to the average best location
        . Counter increases by 1
110    * @throws FunctionEvalLimitException
111    */
112   private void intensify() throws FunctionEvalLimitException{
113     double[] coord=mem.intensify(); //finds average best of all points in med.
        term memory
114     if (cntrl.getMtmSize() ==1)
115       mem.clearSTM();
116     basePoint=new Vector2(fn,coord[0], coord[1]);
117     addBasePointToLogAndMem(basePoint, false);
118     counter++;
119   }
120
121   /**
122    * Diversifies the search by moving the abse point to an area of search space
        that has not been explored that thoroughly.
123    * Counter increases by 1.
124    * @throws FunctionEvalLimitException
125    */
126   private void diversify() throws FunctionEvalLimitException {
127     double[] coord=mem.diversify(); //goes to an area we haven't looked that much
        at.
128     basePoint=new Vector2(fn,coord[0], coord[1]);
129     addBasePointToLogAndMem(basePoint, false);
130     counter++;
131   }
132
133
134   /**
135    * Adds a base point to the taboo logger (we are not interested in the
        potential pooints here)- this stores all the points.
136    * @param point in Vector2 form
137    * @param patternMove whether the point came from a pattern move.
```

```
138     * @throws FunctionEvalLimitException
139     */
140    private void addPointToTabLogger(Vector2 point, boolean patternMove) throws
        FunctionEvalLimitException{
141      tabLogger.addMember(point.getCoordX(), point.getCoordY(), point.getValue(),
        globalCounter, counter, increment, patternMove);
142    }
143
144    /**
145     * Populates the list of potential moves by checking the 4 potential moves are
        valid.
146     * @throws FunctionEvalLimitException
147     */
148    private void populatePotentialMoves() throws FunctionEvalLimitException{
149      potentialPoints.clear();
150
151      double x=basePoint.getCoordX();
152      double y=basePoint.getCoordY();
153
154      //add the four new points (if they are allowed ie in range and not tabu
155      addnewPoint(x+increment,y);
156      addnewPoint(x-increment,y);
157      addnewPoint(x,y+increment);
158      addnewPoint(x,y-increment);
159
160      if (potentialPoints.isEmpty()) //if we have not been able to generate any
        points something is probably wrong with the setup (eg too small area to
        search in)
161        throw new AttributeNotCalulatedException("We have not been able to find a
        new potential point - either they were out of range or taboo");
162    }
163
164    /**
165     * Makes the best move based on all the the potential points
166     * @return ewBasePoint the new base point comes out in here (doesn't matter
        waht you feed in)
167     * Note that we copy the best potential value to the newBasePoint, probably bad
         programming pratise but means less lines.
168     */
169    private Vector2 makeBestPotentialMove(){
170        Vector2 newBasePoint;
171        Collections.sort(potentialPoints);
172        newBasePoint=potentialPoints.get(0); //get the lowest
173        return newBasePoint;
174    }
175
176    /**
177     * Adds a single potential point to the list of potential points if it is valid
          (ie in search space and not taboo)
178     * @param x coord
179     * @param y coord
180     * @throws FunctionEvalLimitException
181     */
182    private void addnewPoint(double x, double y) throws FunctionEvalLimitException{
183      if(isNewPointValid(x,y)){ //is the point within the allowed range?
184        Vector2 point= new Vector2(fn,x,y);
185        if (mem.addPotentialPoint(point)){//check if it's tabu
186          potentialPoints.add(point); //if it is not tabu we add it.
187        }
188      }
189    }
```

```
190
191   /**
192    * Checks new point is in range. (just a wrapper for FunctionDef's method for
       convenience)
193    * @param x x coord
194    * @param y y coord
195    * @return true if allowed
196    */
197   private boolean isNewPointValid(double x, double y){
198     return FunctionDef.checkInRange(x, y);
199   }
200
201   /**
202    * Assesses a pattern move and updates the base point accordingly (ie either
       the halfWayPoint, or the pattern move result).
203    * @param halfWayPoint the point which will be made the next base3 point.
204    * @throws FunctionEvalLimitException thrown if the number of function
       evaluations is exceeded.
205    */
206   private void tryPatternMove(Vector2 halfWayPoint) throws
       FunctionEvalLimitException{
207     double xChange=halfWayPoint.getCoordX()-basePoint.getCoordX(); //look at
       current change in x. we will repeat for pattern move (if applicable).
208     double yChange=halfWayPoint.getCoordY()-basePoint.getCoordY(); //same as
       above but with y.
209
210     Vector2 patternResult=new Vector2(fn,halfWayPoint.getCoordX()+xChange,
       halfWayPoint.getCoordY()+yChange);
211
212     if (isNewPointValid(patternResult.getCoordX(), patternResult.getCoordY()) &&
213       mem.addPotentialPoint(patternResult) &&
214       patternResult.getValue()<halfWayPoint.getValue() ){ //perform pattern move
       if it's valid (ie not taboo and in bounds) and lower
215                         //Note the subtlety with the above if statement. && short
       circuits so we only add the potential point to memory IF
216                         //it is in range - same as we would when we are assessing
       potential points.
217                         //short circuiting means we also only get value if
       necessary.
218       basePoint=patternResult; //pattern move a success so make it the new base
       point.
219       addBasePointToLogAndMem(basePoint,true);
220     } else{ //don't perform the pattern move.
221       basePoint=halfWayPoint; //pattern move wasn't a success so don't make it
       the new base point.
222       addBasePointToLogAndMem(basePoint,false);
223     }
224   }
225
226   /**
227    * Adds a base point to both the memory and the log.
228    * @param point the base point in question
229    * @param patternMove whether it came from a pattern move or not.
230    * @throws FunctionEvalLimitException
231    */
232   private void addBasePointToLogAndMem(Vector2 point, boolean patternMove) throws
       FunctionEvalLimitException{
233     addPointToTabLogger(point, patternMove); //log
234     mem.addBasePt(point); //memory
235   }
236
```

```
237 }
```

RunTS

```
1  /**
2   *
3   */
4  package ex3.tabu_search_specifics;
5
6  import java.util.Random;
7
8  import ex3.RunOptimisationInterface;
9  import ex3.control_parameters.ControlParamsTS;
10 import ex3.evolution_strategies_specifics.Population;
11 import ex3.exceptions.FunctionEvalLimitException;
12 import ex3.loggers.*;
13
14
15 /**
16  * @author eng-188do
17  * Runs tabu search.
18  */
19 public class RunTS implements RunOptimisationInterface {
20
21   /* (non-Javadoc)
22    * @see ex3.RunOptimisation#run(java.lang.Long, java.lang.String, java.lang.
      String)
23    */
24   @Override
25   public void run(Long seed, String inputFile, String outputFile) {
26     //Set up loggers
27     FunctionCallLogger log=new FunctionCallLogger(1000); //only 1000 function
      calls allowed
28     TabuLogger logTabu=new TabuLogger();
29
30     //set up then read in control parameters
31     ControlParamsTS cntrl=new ControlParamsTS();
32     cntrl.readInFile(inputFile);
33
34     //set up random seed
35     Random rnd=new Random(seed);
36
37     //set up tabu search member
38     LocalSearch tabuSearch=new LocalSearch(rnd,cntrl,log,logTabu);
39
40     //run optimisation
41     try{
42       runOptimisation(tabuSearch);
43     } catch (FunctionEvalLimitException e){//we've met our allowed func evals
44       //print out results:
45       log.printResults(outputFile);
46       logTabu.printResults(outputFile);
47     }
48   }
49
50   private void runOptimisation(LocalSearch tabuSearch) throws
      FunctionEvalLimitException{
51     tabuSearch.initialise(); //Initialise randomly in space
52       while (true){ //go through cycles
53         tabuSearch.performOneIterLocalSearch(); //perform one iteration
54       }
```

```
55   }
56
57 }
```

MediumTermMemory

```
1  /**
2   *
3   */
4  package ex3.tabu_search_specifics;
5
6  import ex3.coordinate_holder.Vector2;
7  import ex3.exceptions.AttributeNotCalulatedException;
8  import ex3.exceptions.FunctionEvalLimitException;
9
10 import java.util.*;
11
12 /**
13  * @author eng-188do
14  * Class to hold the medium term memory (mtm).
15  */
16 public final class MediumTermMemory extends ShorterTermMemories {
17   //Constructor
18   /**
19    * Constructor
20    * @param numPoints : is the number of points in the memory
21    * @param rndIn : random number generator
22    */
23   public MediumTermMemory(int numPoints, Random rndIn) {
24     super(numPoints);
25     rnd=rndIn;
26   }
27
28   //Members
29   private Random rnd; //random number generator.
30
31   //Methods
32
33   /* (non-Javadoc)
34    * @see ex3.TSSpecificClasses.ShorterTermMemories#addPoint(ex3.CoordinateHolder
      .Vector2)
35    * Adds (copy of) point into the medium term memory if its value displaces that
       already there.
36    */
37   @Override
38   public void addPoint(Vector2 point) {
39     //Note I have thought about the fact that we are using a Vector2 array rather
       than a map or some other structure to make searching more efficient.
40     //this is because medium term memory is likely to be small notes say 4 items
     so array searching is unlikely to be significant and it is simpler to code
     and
41     //understand.
42
43     if (itemsInMemory==0){ //if memory is empty add it straight away and return
44       memory[0]=new Vector2(point); //add a copy
45       itemsInMemory++;
46     } else{
47       if (checkIfMember(point)==true) // check if already in memory, if it is,
     return
48         return;
49
```

```
50        if(itemsInMemory==memorySize){// is memory full?
51          try{
52            int index=findMax();
53            if (memory[index].getValue()>point.getValue()) //is is smaller than
    current worst?
54              memory[index]=new Vector2(point); //replace current biggest
55          }  catch (FunctionEvalLimitException e) { //All the functions should have
     been evaluated by here.
56            e.printStackTrace();
57            throw new AttributeNotCalulatedException("Points values should have
    been calulated by this point.");
58          }
59        } else{ //memory isn't full so add it to the end and increment counter.
60          memory[itemsInMemory++]=new Vector2(point); //add a copy
61        }
62      }
63    }
64
65    /**
66     * Returns the average from all the stored points
67     * (will be interesting to see what this ends up doing if the bests are in the
     two equal optimums)
68     * @return
69     */
70    public double[] averageBest(){
71      double xPoints=0, yPoints=0;
72
73      //go through all memory points and calulate average
74      int counter;
75      for (counter=0; counter<itemsInMemory; counter++){
76        xPoints+=memory[counter].getCoordX();
77        yPoints+=memory[counter].getCoordY();
78      }
79      xPoints/= (double) counter;
80      yPoints/= (double) counter;
81
82      double[] coord={xPoints, yPoints};
83      return coord;
84    }
85
86    /**
87     * Finds the index of the maximum value in the array.
88     * if several it will return one randomly.
89     * If nothing in memory it will return 0.
90     * @return index of largest
91     */
92    private int findMax() {
93      //NOTE: this is a rather clunky function, in the fact that it looks through
     the whole array twice.
94      // we could perhaps use information in the first search to narrow down our
     second search.
95      // as we are only dealing with small arrays doesn't really matter, but still
     ...
96      try{
97        double max=memory[0].getValue();
98        List<Integer> indices = new ArrayList<Integer>();
99
100       //search through and find the maximum
101       for(int i=0; i<itemsInMemory; i++){
102         if(memory[i].getValue() > max){
103           max=memory[i].getValue();
```

```
104         }
105       }
106
107       //next find the index
108       for(int i=0; i<itemsInMemory; i++){
109         if(memory[i].getValue() == max){
110           indices.add(i);
111         }
112       }
113       //now return a random entry from this list
114       int randomEntry=rnd.nextInt(indices.size());
115       return indices.get(randomEntry);
116
117     }  catch (FunctionEvalLimitException e) { //All the functions should have
    been evaluated by here.
118       e.printStackTrace();
119       throw new AttributeNotCalulatedException("Points values should have been
    calulated by this point.");
120     }
121
122
123   }
124
125   /**
126    * gets the vaue of the minimum in the memory.
127    * @return
128    */
129   public double getMin(){
130     try{
131       double min=memory[0].getValue();
132
133       //search through and find the maximum
134       for(int i=0; i<itemsInMemory; i++){
135         if(memory[i].getValue() < min){
136           min=memory[i].getValue();
137         }
138       }
139
140       return min;
141
142     }  catch (FunctionEvalLimitException e) { //All the functions should have
    been evaluated by here.
143       e.printStackTrace();
144       throw new AttributeNotCalulatedException("Points values should have been
    calulated by this point.");
145     }
146
147   }
148
149 }
```

Memory

```
1 /**
2  *
3  */
4 package ex3.tabu_search_specifics;
5
6 import java.util.*;
7
8 import ex3.control_parameters.ControlParamsTS;
```

```
9  import ex3.coordinate_holder.Vector2;
10 import ex3.exceptions.FunctionEvalLimitException;
11 import ex3.loggers.FunctionCallLogger;
12 import ex3.loggers.TabuLogger;
13
14 /**
15  * @author eng-188do
16  * This function controls the 3 memories and makes sure when making function
      evaluations that they are not tabu.
17  */
18 public class Memory {
19   //Constructor
20   /**
21    * Constructor
22    * @param rndIn : random number generator
23    * @param logIn : tabu logger for base points
24    * @param cntrl : control paramerters
25    */
26   Memory(Random rndIn, TabuLogger logIn, ControlParamsTS cntrl){
27     rnd=rndIn;
28
29     //& set up the memories:
30     mtm=new MediumTermMemory(cntrl.getMtmSize(), rnd);
31     stm=new ShortTermMemory(cntrl.getStmSize());
32     ltm=new SearchDiversifier(rnd);
33   }
34
35   //Members
36   private Random rnd; //random number generator
37   private FunctionCallLogger fn; //function logger and evaluator
38   private SearchDiversifier ltm; //long term memory
39   private MediumTermMemory mtm; //medium term memory
40   private ShortTermMemory stm; //short term memory
41
42
43   //Methods
44   /**
45    * This gets the value for a potential base point and adds it to the relevant
      memories.
46    * @param point we work on this point.
47    * @return false if point is tabu and
48    * @throws FunctionEvalLimitException thrown if the point is tabu!
49    *
50    */
51   public void addBasePt(Vector2 point) throws FunctionEvalLimitException{
52     if (stm.checkIfMember(point))
53       throw new IllegalArgumentException("this point should not be being made a
     base point: it is tabu!");
54
55     point.getValue(); //make sure that the value is calculated
56     stm.addPoint(point);
57     mtm.addPoint(point); //should already have been added as part of the
     potential points but if not we can just add it again
58     ltm.addPt(point.getCoordX(), point.getCoordY());
59
60   }
61
62   /**
63    * Adds potential point to the memory and if it is tabu it will return false
64    * @param point
65    * @return true if not tabu
```

```
66     * @throws FunctionEvalLimitException
67     */
68    public boolean addPotentialPoint(Vector2 point) throws
       FunctionEvalLimitException {
69      if(stm.checkIfMember(point))
70        return false; //this is tabu
71
72      point.getValue(); //make sure we evaluate the points value
73      mtm.addPoint(point); //gets potentially added to the medium term memory
       regardless of whether we go there or not
74      return true; //if we have got here then not tabu
75    }
76
77    /**
78     * Give coordinates for intensifying
79     * @return {x,y}
80     */
81    public double[] intensify(){
82      return mtm.averageBest();
83    }
84
85
86    /**
87     * Get new coordinates for diversification
88     * @return {x,y}
89     */
90    public double[] diversify(){
91      return ltm.genNewPoint();
92    }
93
94    /**
95     * Returns the best solution so far
96     * @return the value at the best solution so far.
97     */
98    public double getBstSolution(){
99      return mtm.getMin();
100   }
101
102   /**
103    * Clears the Short term memory.
104    */
105   public void clearSTM(){
106     stm.clearMemory();
107   }
108
109 }
```

SearchDiversifier

```
1  /**
2   *
3   */
4  package ex3.tabu_search_specifics;
5
6  import java.util.*;
7
8  import ex3.FunctionDef;
9
10 /**
11  * @author eng-188do
```

```
12  * This class holds the long term memory and sorts out the whole diversification
      process.
13  * This class stores a 6 by 6 frequency grid of the search space. And notes how
      often we have had successful moves in each area
14  */
15 public class SearchDiversifier {
16   //Constructor
17   /**
18    * Constructor
19    * @param rndIn : random number generator
20    */
21   public SearchDiversifier(Random rndIn){
22     rnd=rndIn;
23   }
24
25   //Members
26   private int[][] freqTable=new int[6][6]; //this stores the number of times we
      have visited each area of the space.
27                       //this will initialise them all to zero
28                       //storing y's then x's
29   private Random rnd; //random number generator
30
31   //Methods
32   /**
33    * Adds a point to the frequency table
34    * @param x coord
35    * @param y coord (or x_2).
36    */
37   public void addPt(double x, double y){
38     freqTable[findYIndex(y)][findXIndex(x)]++;
39   }
40
41   /**
42    * generates a new value to search from that hasn't been searched before
43    * @return {x,y}
44    */
45   public double[] genNewPoint(){
46     int[] indexToPlacePointsIn=findMin(); //find in which square we should place
      the results in
47
48     //now generate some results in that square
49     double[] returnResults=new double[2];
50     returnResults[0]=genX(indexToPlacePointsIn[0]);
51     returnResults[1]=genY(indexToPlacePointsIn[1]);
52
53     return returnResults;
54   }
55
56
57   /**
58    * Find where in the freq table to put the x value
59    * @param x coord
60    * @return
61    */
62   private int findXIndex(double x){
63     double location=freqTable[0].length*(x-FunctionDef.xlow)/(FunctionDef.xhigh-
      FunctionDef.xlow);  //convert range to 0-6
64     return (int) location; //round down
65   }
66
67   /**
```

```
68    * Find where in the freq table to put the y value
69    * @param y coord
70    * @return index value it fallsin
71    */
72   private int findYIndex(double y){
73     double location=freqTable.length*(y-FunctionDef.ylow)/(FunctionDef.yhigh-
      FunctionDef.ylow);  //convert range to 0-6
74     return (int) location; //round down
75   }
76
77   /**
78    * Finds the minimum indices of the frequency array
79    * Note: if multiple mins then it returns a random one of these.
80    * @return {x,y}
81    */
82   private int[] findMin(){
83     int min=0;
84     List<int[]> indices=new ArrayList<int[]>(); //this expanding list is where we
       will store our potential candidates
85
86     //Find minimum
87     for(int i=0; i<freqTable.length;i++){
88       for(int j=0; j<freqTable[0].length; j++){
89         if(min>freqTable[i][j]){
90           min=freqTable[i][j];
91         }
92       }
93     }
94
95     //Now go through and find where it occurs (in case it occurs multiple times)
96     int[] entry; //{x,y}'s of minimums
97     for(int i=0; i<freqTable.length;i++){
98       for(int j=0; j<freqTable[0].length; j++){
99         if(min==freqTable[i][j]){
100          entry=new int[2];
101          entry[0]=j; //x
102          entry[1]=i; //y
103          indices.add(entry);
104        }
105      }
106    }
107
108    //Now generate random entry from this
109    int randomIndex=rnd.nextInt(  indices.size() );
110    return indices.get(randomIndex);
111  }
112
113  /**
114   * generate a random x coord in the right place
115   * @param xIndex
116   * @return radnom x coord
117   */
118  private double genX(int xIndex){
119    double intervalWidth=(FunctionDef.xhigh-FunctionDef.xlow)/freqTable[0].length
      ; //get how wide an interval is
120    double intervalValue=rnd.nextDouble()*intervalWidth;//get where in the
      interval to place our value
121
122    double xOffset=((double) xIndex)*intervalWidth+FunctionDef.xlow; //find lh
      edge of interval
123    return xOffset+intervalValue;
```

```
124   }
125
126   /**
127    * Generate a random y value in the right place.
128    * @param yIndex
129    * @return random y coord
130    */
131   private double genY(int yIndex){
132     double intervalWidth=(FunctionDef.yhigh-FunctionDef.ylow)/freqTable.length;
      //get how wide an interval is
133     double intervalValue=rnd.nextDouble()*intervalWidth;//get where in the
      interval to place our value
134
135     double yOffset=((double) yIndex)*intervalWidth+FunctionDef.ylow; //find lh
      edge of interval
136     return yOffset+intervalValue;
137   }
138 }
```

ShortTermMemory

```
1  /**
2   *
3   */
4  package ex3.tabu_search_specifics;
5
6  import ex3.coordinate_holder.Vector2;
7
8  /**
9   * @author eng-188do
10  *
11  */
12 public final class ShortTermMemory extends ShorterTermMemories {
13   //Constructor
14   /**
15    * Constructor
16    * @param numPoints: memory capacity ie max number of points it can hold.
17    */
18   public ShortTermMemory(int numPoints) {
19     super(numPoints);
20
21   }
22
23   //Methods
24   /* (non-Javadoc)
25    * @see ex3.TSSpecificClasses.ShorterTermMemories#addPoint(ex3.CoordinateHolder
      .Vector2)
26    * For Short term memory we add this point to the memory replacing the first
      stored memory. ie first in first out
27    * Note you technically can get the same Methods in here. This should be
      stopped by the LocalSearch class not coming back to a point that is taboo.
28    */
29   @Override
30   public void addPoint(Vector2 point) {
31     //Shift array up (deleting last member). if nothing there will just shift
      zeros up, which is fine.
32     System.arraycopy(memory, 0, memory, 1, memory.length-1);
33
34     memory[0]=new Vector2(point); //store a copy
35
36     if (itemsInMemory<memorySize)
```

```
37        itemsInMemory++;
38    }
39
40
41
42 }
```

ShorterTermMemories

```
1  /**
2   *
3   */
4  package ex3.tabu_search_specifics;
5
6  import ex3.coordinate_holder.Vector2;
7
8  /**
9   * @author eng-188do
10  * This is the base class for the shorter term memories: ie the short term memory
        and the medium term memory.
11  */
12 public abstract class ShorterTermMemories {
13    //Constructor
14    /**
15     * Constructor
16     * @param numPoints :max number of points in memory
17     */
18    ShorterTermMemories(int numPoints){
19      memorySize=numPoints;
20      memory= new Vector2[numPoints];
21      itemsInMemory=0;
22    }
23
24    //Members
25    protected final int memorySize; //memory capacity - number of points we can
        have.
26    protected int itemsInMemory=0; //current number of items in memory
27    protected Vector2[] memory; //memory items.
28
29    //Methods
30    /**
31     * Adds a point to memory
32     * @param point : point to add to the memory
33     */
34    public abstract void addPoint(Vector2 point);
35
36    /**
37     * Empties the memory
38     */
39    public void clearMemory(){
40      itemsInMemory=0;
41      memory= new Vector2[memorySize];
42    }
43
44
45    /**
46     * Checks if point is a member (ie same coords) of the short term memory.
47     * @param point
48     * @return true if it is a member
49     */
50    public boolean checkIfMember(Vector2 point){
```

```
51      double xCoord=point.getCoordX();
52      double yCoord=point.getCoordY();
53      Vector2 memoryItem;
54
55      for (int i=0; i<itemsInMemory; i++){ //loop through and check whether the x
        and y coords already exist in memory
56        memoryItem=memory[i];
57        if ( memoryItem.getCoordX()==xCoord && memoryItem.getCoordY()==yCoord){
58          //TODO: we want to check above works and we don't need to use tolerance
        instead. like Math.abs() rather than ==
59          return true;
60        }
61      }
62
63      return false; //if got here then it's not a member of the memory
64    }
65  }
```

tests

ControlParamsESTest

```
1  /**
2   *
3   */
4  package ex3.tests;
5
6  import static org.junit.Assert.*;
7
8  import org.junit.Test;
9
10 import ex3.control_parameters.ControlParamsES;
11
12 /**
13  * @author eng-188do
14  *
15  */
16 public class ControlParamsESTest {
17
18   /**
19    * Test method for {@link ex3.control_parameters.ControlParams#readInFile(java.
      lang.String)}.
20    */
21   @Test
22   public void testReadInFile() {
23     //initialise
24     ControlParamsES cntrl = new ControlParamsES();
25     double expectedBeta=1.056;
26     double expectedTau=0.32;
27     double expectedTauP=1.56;
28     double expectedVariance=4.35;
29     int expectedParents=5;
30     int expectedPAndCratio=3;
31     /* I have also made a file with the following contents:
32       beta,1.056
33       tau,0.32
34       tauP,1.56
35       variance,4.35
36       parents,5
```

```
37        parent child ratio,3*/
38      String filename=System.getProperty("user.home") + "/Documents/uniwork/4M17/
        ex3/code/esTestInputs.csv";
39
40      //run
41      cntrl.readInFile(filename);
42
43
44      //assert
45      assertEquals("beta",expectedBeta,cntrl.getBeta(),TestConstants.DOUBLE_EPSILON
        );
46      assertEquals("tau",expectedTau,cntrl.getTau(),TestConstants.DOUBLE_EPSILON);
47      assertEquals("tau_p",expectedTauP,cntrl.getTauP(),TestConstants.
        DOUBLE_EPSILON);
48      assertEquals("variance",expectedVariance,cntrl.getVar(),TestConstants.
        DOUBLE_EPSILON);
49      assertEquals("parents number",expectedParents,cntrl.getNumParents());
50      assertEquals("parent child ratio",expectedPAndCratio,cntrl.
        getParentChildRatio());
51
52    }
53
54 }
```

ESMemberTest

```
1  /**
2   *
3   */
4  package ex3.tests;
5
6  import static org.junit.Assert.*;
7
8  import java.util.*;
9
10 import org.junit.Test;
11
12 import ex3.FunctionDef;
13 import ex3.control_parameters.ControlParamsES;
14 import ex3.evolution_strategies_specifics.ESMember;
15 import ex3.exceptions.FunctionEvalLimitException;
16 import ex3.loggers.FunctionCallLogger;
17
18 /**
19  * @author eng-188do
20  *
21  */
22 public class ESMemberTest {
23
24    /**
25     * Test method for {@link ex3.evolution_strategies_specifics.ESMember#getValue
        ()}.
26     */
27    @Test
28    public void testGetValue() {
29      //initialise
30      FunctionCallLogger fn=new FunctionCallLogger(10);
31      ControlParamsES cntrl=new ControlParamsES();
32      double[] dummyVars={1,1};
33      double[] dummyAlphas={0,0,0,0};
```

```
34      ESMember mem=new ESMember(fn, 0.2345, -4.561, new Random(1), cntrl, dummyVars
        , dummyAlphas, 0);
35      double expectedOut=23.59857075; //on calulator
36      double actualOut=0;
37
38
39      //run
40      try {
41        actualOut=mem.getValue();
42      } catch (FunctionEvalLimitException e) {
43        e.printStackTrace();
44      }
45
46      //assert
47      assertEquals("getVal function does not work",expectedOut,actualOut,
        TestConstants.DOUBLE_EPSILON);
48
49    }
50
51
52
53    /**
54     * Test method for {@link ex3.coordinate_holder.Vector2#compareTo(ex3.
        coordinate_holder.Vector2)}.
55     * I'm going to test this by sorting a simple list. As this is really what I
        want this method to help me do.
56     */
57    @Test
58    public void testCompareTo() {
59      //initialise
60      FunctionCallLogger fn=new FunctionCallLogger(7);
61      ControlParamsES cntrl=new ControlParamsES();
62      double[] dummyVars={1,1};
63      double[] dummyAlphas={0,0,0,0};
64
65        //Set up 5 members in a list
66      ESMember mem1=new ESMember(fn, 0.2345, -4.561, new Random(1), cntrl,
        dummyVars, dummyAlphas, 0);//value 23.59857075
67      ESMember mem2=new ESMember(fn, 0.2345, -4.561, new Random(1), cntrl,
        dummyVars, dummyAlphas, 0);  //value 23.59857075
68      ESMember mem3=new ESMember(fn, 0.5, -2.4, new Random(1), cntrl, dummyVars,
        dummyAlphas, 0); //value: 17.2528
69      ESMember mem4=new ESMember(fn, 0.6, 3.2, new Random(1), cntrl, dummyVars,
        dummyAlphas, 0); //value: 36.1723
70      ESMember mem5=new ESMember(fn, -5.32, -0.1, new Random(1), cntrl, dummyVars,
        dummyAlphas, 0); //value: 29.0968
71      List<ESMember> listMems=new ArrayList<ESMember>();
72      listMems.add(mem1);
73      listMems.add(mem2);
74      listMems.add(mem3);
75      listMems.add(mem4);
76      listMems.add(mem5);
77      double[] expectedXValsOut={0.5, 0.2345, 0.2345, -5.32, 0.6};
78      double[] expectedValsOut={17.2528, 23.59857075, 23.59857075, 29.0968,
        36.1723};
79
80      for(ESMember member: listMems){
81        try {
82          member.getValue();
83        } catch (FunctionEvalLimitException e) { //shouldn't happen as less than 10
        evals
```

```
84          e.printStackTrace();
85        } //make sure all the values have been calculated.
86      }
87
88      //run
89      Collections.sort(listMems);
90
91      //assert
92      for(int i=0; i<listMems.size(); i++){
93        try {
94          assertEquals("Sorting function does not work",expectedValsOut[i],listMems
    .get(i).getValue(),TestConstants.DOUBLE_EPSILON);
95        } catch (FunctionEvalLimitException e) { //shouldn't throw this exception
    as should no eval again - we're also checking this
96          e.printStackTrace();
97        }
98        assertEquals("Sorting function does not work",expectedXValsOut[i],listMems.
    get(i).getCoordX(),TestConstants.DOUBLE_EPSILON);
99      }
100   }
101
102 }
```

FunctionDefTest

```
1  package ex3.tests;
2  import static org.junit.Assert.*;
3
4  import org.junit.Rule;
5  import org.junit.Test;
6  import org.junit.rules.ExpectedException;
7
8  import ex3.FunctionDef;
9
10
11 /**
12  *
13  */
14
15 /**
16  * @author eng-188do
17  * Class to test functions in FunctionDef.
18  */
19 public class FunctionDefTest {
20
21   /**
22    * Test method for {@link FunctionDef#evalFunc(double, double)}.
23    */
24   @Test
25   public void testEvalFunc1() {
26     //intialise
27     double inputX=0.2345;
28     double inputY=-4.561;
29     double expectedOut=23.59857075;
30     double actualOut=0;
31
32     //run
33     actualOut=FunctionDef.evalFunc(inputX, inputY);
34
35     //assert
```

```
36      assertEquals("Eval function does not work",expectedOut,actualOut,
     TestConstants.DOUBLE_EPSILON);
37
38    }
39
40
41    @Rule
42    public ExpectedException ex = ExpectedException.none();
43
44    /**
45     * Test method for {@link FunctionDef#evalFunc(double, double)}.
46     */
47    @Test
48    public void testEvalFunc2() {
49      //intialise
50      double inputX=0.2345;
51      double inputY=-7.2536;
52      double actualOut;
53
54      //run
55      ex.expect(IllegalArgumentException.class);
56      actualOut=FunctionDef.evalFunc(inputX, inputY);
57
58    }
59
60
61
62    /**
63     * Test method for {@link FunctionDef#checkInRange(double, double)}.
64     */
65    @Test
66    public void testCheckInRange1() {
67      //intialise
68      double inputX=6.0;
69      double inputY=-5.9;
70      boolean expectedOut=true;
71      boolean actualOut=false;
72
73      //run
74      actualOut=FunctionDef.checkInRange(inputX, inputY);
75
76      //assert
77      assertEquals("Range function does not work",expectedOut,actualOut);
78    }
79
80
81    /**
82     * Test method for {@link FunctionDef#checkInRange(double, double)}.
83     */
84    @Test
85    public void testCheckInRange2() {
86      //intialise
87      double inputX=6.01;
88      double inputY=-5.9;
89      boolean expectedOut=false;
90      boolean actualOut=true;
91
92      //run
93      actualOut=FunctionDef.checkInRange(inputX, inputY);
94
95      //assert
```

```
96      assertEquals("Range function does not work",expectedOut,actualOut);
97    }
98
99  }
```

MediumTermMemoryTest

```
1  /**
2   *
3   */
4  package ex3.tests;
5
6  import static org.junit.Assert.*;
7
8  import java.util.Random;
9
10 import org.junit.Test;
11
12 import ex3.coordinate_holder.Vector2;
13 import ex3.exceptions.FunctionEvalLimitException;
14 import ex3.loggers.FunctionCallLogger;
15 import ex3.tabu_search_specifics.MediumTermMemory;
16
17 /**
18  * @author eng-188do
19  *
20  */
21 public class MediumTermMemoryTest {
22
23    /**
24     * Test method for {@link ex3.tabu_search_specifics.MediumTermMemory#
       averageBest()}.
25     */
26    @Test
27    public void testAverageBest() {
28      //arrange
29      FunctionCallLogger fn=new FunctionCallLogger(100);
30      MediumTermMemory mem=new MediumTermMemory(20, new Random(1));
31      Vector2 point1 = new Vector2(fn,0,2);
32      Vector2 point2 = new Vector2(fn,1,3);
33      Vector2 point3 = new Vector2(fn,4,1);
34      double[] expectedAvergageBest={1.6666666667, 2}; //from calculator.
35      double [] actualAverageBest=new double[2];
36
37      //run
38      mem.addPoint(point1);
39      mem.addPoint(point2);
40      mem.addPoint(point3);
41      actualAverageBest=mem.averageBest();
42
43      //assert
44      assertEquals("Incorrect average best x calulated"  ,expectedAvergageBest[0],
       actualAverageBest[0],TestConstants.DOUBLE_EPSILON);
45      assertEquals("Incorrect average best y calulated"  ,expectedAvergageBest[1],
       actualAverageBest[1],TestConstants.DOUBLE_EPSILON);
46    }
47
48    /**
49     * Test method for {@link ex3.tabu_search_specifics.MediumTermMemory#getMin()}.
50     */
51    @Test
```

```
52  public void testGetMin1() {
53    //arrange
54    FunctionCallLogger fn=new FunctionCallLogger(100);
55    MediumTermMemory mem=new MediumTermMemory(20, new Random(1));
56    Vector2 point1 = new Vector2(fn,0,2); //val:2.868795616243186
57    Vector2 point2 = new Vector2(fn,1,3); //val: 47.129226041767765
58    Vector2 point3 = new Vector2(fn,4,1); //val: 19.896177372947569
59    //need to ensure that values have been calculated.
60    try {
61      point1.getValue();
62      point2.getValue();
63      point3.getValue();
64    } catch (FunctionEvalLimitException e) { //given it a 100 trials so should be
      fine.
65      e.printStackTrace();
66    }
67
68    double expectedMin=2.868795616243186; //by hand on calc
69    double actualMin=0;
70
71    //run
72    mem.addPoint(point1);
73    mem.addPoint(point2);
74    mem.addPoint(point3);
75    actualMin=mem.getMin();
76
77    //assert
78    assertEquals("Incorrect min returned"  ,expectedMin, actualMin,TestConstants.
     DOUBLE_EPSILON);
79  }
80
81  /**
82   * Test method for {@link ex3.tabu_search_specifics.MediumTermMemory#getMin()}.
83   */
84  @Test
85  public void testGetMin2() {
86    //arrange
87    FunctionCallLogger fn=new FunctionCallLogger(100);
88    MediumTermMemory mem=new MediumTermMemory(4, new Random(1)); //only 4 items
     in memory.
89    Vector2 point1 = new Vector2(fn,0,2); //val:2.868795616243186
90    Vector2 point2 = new Vector2(fn,1,3); //val: 47.129226041767765
91    Vector2 point3 = new Vector2(fn,4,1); //val: 19.896177372947569
92    Vector2 point4 = new Vector2(fn,-2,5); //val: 58.344893672224373
93    Vector2 point5 = new Vector2(fn,5.9,-5.9); //val: 145
94    Vector2 point6 = new Vector2(fn,-1.5,3); //val: -85.592787269845118
95    Vector2 point7 = new Vector2(fn,1,3); //(identical to 2
96    //need to ensure that values have been calculated.
97    try {
98      point1.getValue();
99      point2.getValue();
100     point3.getValue();
101     point4.getValue();
102     point5.getValue();
103     point6.getValue();
104     point7.getValue();
105   } catch (FunctionEvalLimitException e) { //given it a 100 trials so should be
      fine.
106     e.printStackTrace();
107   }
108
```

```
109     double expectedMin=-85.592787269845118; //by hand on calc
110     double actualMin=0;
111 
112     //run
113     mem.addPoint(point1);
114     mem.addPoint(point2);
115     mem.addPoint(point3);
116     mem.addPoint(point4);
117     mem.addPoint(point5);
118     mem.addPoint(point6);
119     mem.addPoint(point7);
120     actualMin=mem.getMin();
121 
122     //assert
123     assertEquals("Incorrect min returned"  ,expectedMin, actualMin,TestConstants.
      DOUBLE_EPSILON);
124   }
125 
126 }
```

SearchDiversifierTest

```
1  /**
2   *
3   */
4  package ex3.tests;
5 
6  import static org.junit.Assert.*;
7 
8  import java.util.Random;
9 
10 import org.junit.Test;
11 
12 import ex3.tabu_search_specifics.SearchDiversifier;
13 
14 /**
15  * @author eng-188do
16  *
17  */
18 public class SearchDiversifierTest {
19 
20   /**
21    * Test method for {@link ex3.tabu_search_specifics.SearchDiversifier#
      genNewPoint()}.
22    */
23   @Test
24   public void testGenNewPoint() {
25     //initialise
26     Random rnd = new Random(1);
27     SearchDiversifier diversifier=new SearchDiversifier(rnd);
28 
29     //run
30     //run by adding new points in every location but one and getting new location
31     /*   6
32      *   4  _|_|_|_|_|_|_
33      *   2  _|_|_|_|_|_|_
34      *   0  _|_|_|_|_|_|_
35      *  -2  _|_|_|_|_|_|_
36      *  -4  _|_|_|_|_|_|_
37      *  -6   | | | | | |
38      *   -6,-4,-2,0,2,4,6
```

```
     */
    diversifier.addPt(-4.5, 5);
    diversifier.addPt(-2.5, 5);
    diversifier.addPt(-0.5, 5);
    diversifier.addPt(0.5, 5);
    diversifier.addPt(2.5, 5);
    diversifier.addPt(4.5, 5);
    diversifier.addPt(-4.5, 3);
    diversifier.addPt(-2.5, 3);
    diversifier.addPt(-0.5, 3);
    diversifier.addPt(0.5, 3);
    diversifier.addPt(2.5, 3);
    diversifier.addPt(4.5, 3);
    diversifier.addPt(-4.5, 1);
    diversifier.addPt(-2.5, 1);
    diversifier.addPt(-0.5, 1);
    diversifier.addPt(0.5, 1);
    diversifier.addPt(2.5, 1);
    diversifier.addPt(4.5, 1);
    //diversifier.addPt(-4.5, -1); so point we generate should be between -4>x>-6
    & -2<y<0
    diversifier.addPt(-2.5, -1);
    diversifier.addPt(-0.5, -1);
    diversifier.addPt(0.5, -1);
    diversifier.addPt(2.5, -1);
    diversifier.addPt(4.5, -1);
    diversifier.addPt(-4.5, -3);
    diversifier.addPt(-2.5, -3);
    diversifier.addPt(-0.5, -3);
    diversifier.addPt(0.5, -3);
    diversifier.addPt(2.5, -3);
    diversifier.addPt(4.5, -3);
    diversifier.addPt(-4.5, -5);
    diversifier.addPt(-2.5, -5);
    diversifier.addPt(-0.5, -5);
    diversifier.addPt(0.5, -5);
    diversifier.addPt(2.5, -5);
    diversifier.addPt(4.5, -5);
    double coord[]=diversifier.genNewPoint();

    //assert
    boolean xInRange=(-4>=coord[0] && coord[0]>=-6);
    boolean yInRange=(0>=coord[1] && coord[1]>=-2);
    assertTrue(xInRange);
    assertTrue(yInRange);

  }

}
```

ShortTermMemoryTest

```
/**
 *
 */
package ex3.tests;

import static org.junit.Assert.*;

import org.junit.Test;

```

```
10 import ex3.coordinate_holder.Vector2;
11 import ex3.loggers.FunctionCallLogger;
12 import ex3.tabu_search_specifics.ShortTermMemory;
13
14 /**
15  * @author eng-188do
16  *
17  */
18 public class ShortTermMemoryTest {
19
20   /**
21    * Test method for {@link ex3.tabu_search_specifics.ShorterTermMemories#
      checkIfMember(ex3.coordinate_holder.Vector2)}.
22    * This point checks tabu bit is working by checking it remembers the last
      members.
23    */
24   @Test
25   public void testCheckIfMember() {
26     //initialise
27     FunctionCallLogger fn=new FunctionCallLogger(100);
28     ShortTermMemory mem= new ShortTermMemory(5);
29     Vector2 point1 = new Vector2(fn,0,2);
30     Vector2 point2 = new Vector2(fn,1,3);
31     Vector2 point3 = new Vector2(fn,4,1);
32     Vector2 point4 = new Vector2(fn,-2,5);
33     Vector2 point5 = new Vector2(fn,5.9,-5.9);
34     Vector2 point6 = new Vector2(fn,-1.5,3);
35     Vector2 point7 = new Vector2(fn,1,3);
36
37     //run and assert
38     assertFalse(mem.checkIfMember(point1)); //should be nothing in memory.
39     mem.addPoint(point1);
40     mem.addPoint(point2);
41     mem.addPoint(point3);
42     mem.addPoint(point4);
43     mem.addPoint(point5);
44     assertTrue(mem.checkIfMember(point1)); //check points 1 and 3 went in.
45     assertTrue(mem.checkIfMember(point3));
46     mem.addPoint(point6);
47     mem.addPoint(point7);
48     assertFalse(mem.checkIfMember(point1)); //point1 should have been pushed out
      of memory now.
49     assertTrue(mem.checkIfMember(point3)); //but point3 should still be there
50
51   }
52
53 }
```

TestConstants

```
1 package ex3.tests;
2 /**
3  *
4  */
5
6 /**
7  * @author eng-188do
8  * Constants used in the test class
9  */
10 public class TestConstants {
11   static final double DOUBLE_EPSILON=0.0001;
```

```
}
```

TwoDMatricesTest

```
package ex3.tests;
import static org.junit.Assert.*;

import org.junit.Test;

import ex3.auxillary_maths_functions.TwoDMatrices;

/**
 *
 */

/**
 * @author eng-188do
 *
 */
public class TwoDMatricesTest {

  /**
   * Test method for {@link TwoDMatrices#matrixMultiplication(double[][], double
   [][])}.
   */
  @Test
  public void testMatrixMultiplication1() {
    //intialise
    double[][] mat1={ {1.30,  3.10},
            {-4.10, 2.25}};
    double[][] mat2={ {0.7, 0.1},
              {4, -2.5}};
    double[][] expectedOut={ {13.31,  -7.62},
              {6.13,  -6.035}}; //from MATLAB
    double[][] actualOut={ {0,  0},
                {0, 0}};

    //run
    actualOut=TwoDMatrices.matrixMultiplication(mat1, mat2);

    //assert
    for(int i=0; i<2 ; i++){
      for(int j=0;j<2;j++){
        assertEquals("Eval function does not work",expectedOut[i][j],actualOut[i
   ][j],TestConstants.DOUBLE_EPSILON);
      }
    }

  }


  /**
   * Test method for {@link TwoDMatrices#matrixMultiplication(double[][], double
   [][])}.
   */
  @Test
  public void testMatrixMultiplication2() {
    //intialise
    double[][] mat1={ {1.30,  3.10},
            {-4.10, 2.25}};
    double[][] mat2={ { 0.1},
```

```
55                { -2.5}};
56      double[][] expectedOut={    {-7.6200},
57                        {-6.0350}}; //from MATLAB
58      double[][] actualOut={ {  0},
59                  { 0}};
60
61      //run
62      actualOut=TwoDMatrices.matrixMultiplication(mat1, mat2);
63
64      //assert
65      for(int i=0; i<1 ; i++){
66        assertEquals("Eval function does not work",expectedOut[i][0],actualOut[i
      ][0],TestConstants.DOUBLE_EPSILON);
67      }
68
69    }
70
71    /**
72     * Test method for {@link TwoDMatrices#twoByTwoMatrixInversion(double[][])}.
73     */
74    @Test
75    public void testTwoByTwoMatrixInversion() {
76      double[][] mat={ {0.7,  0.1},
77          {4, -2.5}};
78      double[][] expectedOut={ {1.16279069767442, 0.0465116279069767},
79          {1.86046511627907,  -0.325581395348837}}; //from MATLAB
80      double[][] actualOut={ {0,  0},
81          {0, 0}};
82
83      //run
84      actualOut=TwoDMatrices.twoByTwoMatrixInversion(mat);
85
86      //assert
87      for(int i=0; i<2 ; i++){
88        for(int j=0;j<2;j++){
89          assertEquals("Mat inv function does not work",expectedOut[i][j],actualOut
      [i][j],TestConstants.DOUBLE_EPSILON);
90  }
91  }
92    }
93
94    /**
95     * Test method for {@link TwoDMatrices#calcDeterminantTwobyTwo(double[][])}.
96     */
97    @Test
98    public void testCalcDeterminantTwobyTwo() {
99      double[][] mat={ {0.7,  0.1},
100         {4, -2.5}};
101     double expectedOut=-2.15; //from MATLAB
102     double actualOut=0;
103
104     //run
105     actualOut=TwoDMatrices.calcDeterminantTwobyTwo(mat);
106     //assert
107     assertEquals("Mat det function does not work",expectedOut,actualOut,
      TestConstants.DOUBLE_EPSILON);
108   }
109
110   /**
111    * Test method for {@link TwoDMatrices#convertArrayToColVector(double[])}
112    */
```

```
113   @Test
114   public void testConvertArrayToColVector(){
115     //intialise
116     double[] mat1={7.35,  -3.25};
117 
118     double[][] expectedOut={    {7.35},
119                     {-3.25}};
120     double[][] actualOut={ {  0},
121         { 0}};
122 
123     //run
124     actualOut=TwoDMatrices.convertArrayToColVector(mat1);
125 
126     //assert
127     for(int i=0; i<1 ; i++){
128       assertEquals("Eval function does not work",expectedOut[i][0],actualOut[i
      ][0],TestConstants.DOUBLE_EPSILON);
129     }
130   }
131 
132   /**
133    * Test method for {@link TwoDMatrices#convertColVectorToArray(double[][])}
134    */
135   @Test
136   public void convertColVectorToArray(){
137     //intialise
138     double[][] mat1={{7.35},  {-3.25}};
139 
140     double[] expectedOut={    7.35,-3.25};
141     double[] actualOut={  0,    0};
142 
143     //run
144     actualOut=TwoDMatrices.convertColVectorToArray(mat1);
145 
146     //assert
147     for(int i=0; i<1 ; i++){
148       assertEquals("Eval function does not work",expectedOut[i],actualOut[i],
      TestConstants.DOUBLE_EPSILON);
149     }
150   }
151 
152 }
```

B.8 Example Inputs

The input csv when running the ES program had the following form:

Example Program Inputs

```
1 beta,0.0873
2 tau,0.595
3 tauP,0.5
4 variance,1
5 parents,30
6 parent child ratio,5
```

B.9 Python Batch Script

The following Python script was used to run the Java program 1000 times for TS, extracting the best function evaluation on each run and storing this in a new csv file for reading in and then plotting

using MATLAB. Online you can find similar scripts for ES and PSO and a script for ES, which recorded the mean and average member of the population for each generation over 1000 runs.

Python Batch Script to run TS 1000 times

```
__author__ = 'eng-188do'

import random
import math
import subprocess
import csv
from os.path import expanduser
import os

###### INFO
# Script to extract the best point found in the whole TS over multiple runs - ie what would the best value in the
# archive be at the end of the run.
# Haven't yet profiled and made efficient - the for loops may be slow.
#   eng-188do Dec 2014


###### FUNCTIONS

# Function to find the minimum in the log entries
def findMin(populationLogEntries):
    minVal = float('inf')

    for row in populationLogEntries: #go through rows and extract the value, if smaller than current min store.
        value=float(row[2])
        if value<minVal:
            minVal=value
            bestRow=row

    return bestRow

###### SCRIPT

home = expanduser("~") #get my home directory - so you cannot see my name!

random.seed(14846126)  # set up random seed
bestLocations = []  # set up list to store the best locations in each run.

iter=1000 #number of iterations to carry out
failCount=0 #variable to store how many times in the run the program failes
while iter: # run the program 100 times recording the outputs.
    # Run the java program
    seed=str(random.randint(-math.pow(2, 63), math.pow(2, 63) - 1))
    commandString = "java -cp ~/Documents/uniwork/4M17/ex3/code/Exercise3/bin ex3.RunGeneral "
    commandString += seed
    commandString += " ~/Documents/uniwork/4M17/ex3/code/defaultTSInputs.csv ~/Documents/uniwork/4M17/ex3/code/outputTS_python TS"
    failure = subprocess.call(commandString, shell=True) #record if it has failed or not

    if not failure: #if not failed read in the file and get the best result found.
        print("Program Executed Successfully")
        fileLocation = home + '/Documents/uniwork/4M17/ex3/code/outputTS_python_FunctionCallLogger.csv'
```

```
        with open(fileLocation, newline='') as csvfile:
            populationLogEntries = csv.reader(csvfile)
            bestLocations.append(findMin(populationLogEntries))
        os.remove(fileLocation)
    else: #if failed then say so and up the count.
        print("Could not execute program, seed was " + seed)
        failCount+=1

    iter+=-1 #we have done one more iteration so only iter more to go...


# Write out the best one
writerLocation=home+ '/Documents/uniwork/4M17/ex3/code/outputTS_python_best100.csv' #storage location
with open(writerLocation, 'w', newline='') as csvfile:
    writer = csv.writer(csvfile, delimiter=',')
    writer.writerows(bestLocations) #write out the best rows in each one in a new csv for reading by MATLAB for plotting.


print('failcount was ' +str(failCount)) #print out fail count
```

B.10 MALAB Archive Script

The following archive script read in the output of the function call logger, ran the archive function, plotted the contours of the Bird Function and then plotted the results.

Archive Script

```
%close all; clear all;
M = csvread('outputS_FunctionCallLogger.csv'); %read in csv
archiveResults  = archive( M, 5 ); %run archive program

%%Calculate the Bird Function on a mesh grid of the area.
x=[-6:0.1:6];
[X,Y] = meshgrid(x,x);
Z=sin(X).*exp((1-cos(Y)).^2)+cos(Y).*exp((1-sin(X)).^2)+(X-Y).^2;

%%Now plot - contours first
figure
contour(X,Y,Z,'LineColor','k');
 hold on
xlabel('x_1')
ylabel('x_2')
title('Archive Results')
sub=M(:,4)==i;
plot(archiveResults(1,:),archiveResults(2,:),'o','MarkerFaceColor',      [
    0.4940    0.1840    0.5560],'MarkerEdgeColor',      [   0.4940    0.1840
    0.5560])
```

The archive function is listed next. As it is a part of the optimisation routine it has more comments to explain what it does.

Archive Function

```
function [ archiveResults ] = archive( functionCalls, L )
%archive best L dissimilar Solutions archive
%   functionCalls: record of all the function calls
%   L= number of calls to store
%
```

```
%archive is a matrix with the points as columns and the rows as x coord,
%y coord and value.
% ie archive=[ xcoord1, xcoord2,....;
%              ycoord1, ycoord2,....;
%              value1 , value2 , ...]


archiveResults=functionCalls(1,1:3)'; %put first item in archive as always goes
    in.
dMin=0.4;
dSim=0.05;

numInArchive=size(archiveResults,2);
assert(numInArchive==1); %should be 1 in at this point.

% go through all function calls (starting at second one as we have already added
    first.)
for i=2:size(functionCalls,1)
    numInArchive=size(archiveResults,2); %update number in archive.
    point=functionCalls(i,1:3)'; %get in the same format as archive - ie column
    vector

    [index, distanceToClosest] = findClosestSolution(archiveResults,point); %
    find closest point and distance to it.

    if (numInArchive<L) %archive not full
        if(distanceToClosest>dMin) %if far emough away from others then archive
    it
            archiveResults(:,end+1)=point;
        end
    else %archive is full
        if (distanceToClosest>dMin) %sufficiently dissimilar to other points in
    archive
            [indexWorst, valueWorst]=findWorstInArchive(archiveResults);
            if (point(3)<valueWorst) %archive if it is better than the worse
    point
                archiveResults(:,indexWorst)=point; % replaces the worst point in
     the archive
            end
        else %not dissimilar to other points in archive
            if point(3)<min(archiveResults(3,:)) %archive it if it is the best
    point so far
                archiveResults(:,index)=point; %then it replaces the closest
    point
            elseif (point(3)<archiveResults(3,index) && distanceToClosest<dSim) %
    or if it close enough to its closest neighbour and is better than it then it
    is archived.
                archiveResults(:,index)=point;
            end
        end
    end
end
end

function [index, distance] = findClosestSolution(archive,point)
%finds the index of the closest point. if several returns the one with the
%maximum value. If there are several of these then it will return the first
%one.

 %set the minimum to the first entry at first.
minDistance=norm(archive(1:2,1)-point(1:2));
```

```
55 index=1;
56 value=archive(3,1);
57
58     for i=2:size(archive,2) %we could consider vectorising this loop to speed it
    up. but archive is only small so probably insignificant.
59         if (norm(archive(1:2,i)-point(1:2))< minDistance)
60             minDistance=norm(archive(1:2,i)-point(1:2)); %replace it with new
    index.
61             index=i;
62             value=archive(3,i);
63         elseif (norm(archive(1:2,i)-point(1:2))== minDistance && archive(3,i) >
    value) %%if same distance as other one replace if the value is better.
64             minDistance=norm(archive(1:2,i)-point(1:2)); %replace it with new
    index.
65             index=i;
66             value=archive(3,i);
67         end
68     end
69
70 distance=minDistance;
71 end
72
73 function [index, value]=findWorstInArchive(archive)
74 %finds the index and the value of the point with the highest value in the
75 %archive.
76 [value index]=max(archive(3,:));
77 end
```