1 Implementation

I chose to use JAVA as the programming language that I used to implement the algorithms, mainly because it is the language which I have most experience in. This enabled me to quickly convert the pseudo code functions and mathematical equations provided in the assignment JAVA for testing on my machine.

1.1 (1+1) evolution strategy algorithm

My implementation allows the user to first choose which mutation algorithm they wish to use via console input out of the four implemented. It then creates the initial search pool for the run, $x = (x_1, \ldots, x_n)$ by picking x_i uniformly at random from $[-100, 100]$. It then calls a function, iterate(), which loops T times (in my testing this was 5000) calling the selected mutation algorithm for each iteration and storing that iterations fitness calculation into an array. Finally, it outputs the array of the fitness for each iteration to a file. This whole process is then repeated 30 times for the 30 runs of the program we are required to perform and all of the results for each iteration of each run is stored in the output file (with spaces separating each result).

This has the effect of the program running and exporting the data for all 30 runs of 5000 iterations in one go which saves a lot of time and allows the data to be exported straight into a spreadsheet application (Excel).

1.2 Fitness Function

$SPHERE(x) = \sum_{i=1}^{n} x_i^2$ with $-\infty < x < \infty$ I implemented the fitness function in a dedicated method inside my code. It has a parameter X which is an array of doubles, which the function calculates the sum of all of the elements squared.

```
private double calculateFitness(double[] x) {
  double sum = 0;
  int count = n - 1;
  while (count >= 0) {
     sum = sum + (x[count] * x[count]);
     count--;
  }
  return sum;
}
```

I have checked this function against the graph provided for $X[n]$ where $n = 2$ which proved successful. Therefore I am confident in my implementation of this method.

1.3 Fitness Checking

I also implemented the method to check the fitness to see if the offspring is better than the parent in its own function. This method first calculates the fitness of the parent & offspring and then compares them to see which is better. If the offspring is better then the function returns boolean True and if not it returns False.

```
private boolean fitnessCheck(double[] offspring)  {
  // Calculate the fitness of both offspring and X
  double offspringFitness = calculateFitness(offspring);
  double parentFitness = calculateFitness(X);

  // Check which is the better fitness
  if (offspringFitness < parentFitness) {
     return true;
  } else {
     return false;
  }
}
```

1.4 Initial search point

My function for creating the initial search pool uses Java's Math.random() to generate uniform random numbers to be used as the initial values for the array X. I have included a range on the random values produced to that it satisfies $[-100 < X_i < 100]$.

```
private void createInitialSearchPool() {
  // Populate the X array with random numbers
  for (int i = 0; i < n; i++) {
     double rand = searchSpaceMin + Math.random() * (searchSpaceMax);
     X[i] = rand;
  }
  System.out.print("Initial value of 'X' set to [ ");
  for (int i = 0; i < n; i++) {
     System.out.print(X[i] + " ");
  }
  System.out.println("]");
}
```

1.5 Output File

My function to write the output results to a file uses Java's FileReader & FileWriter functions. It first tries to see if the file already exists and if so it reads it into an array of lines. These lines are then used as prefixes to the current iterations data so that the eventual file contains all 5000 iteration's fitness

calculations for each of the 30 runs (results separated by spaces). This data can then be easily parsed into a spreadsheet application.

1.6 Uniform Mutation Algorithm

My implementation of the Uniform Mutation Algorithm uses randomly generated numbers to pick an index of the X array and then replace it with a new value which is also randomly generated (within the bounds of $[-100 < x < 100]$).

```
public static double[] UniformMutation(double[] x) {
   double[] xNew = x.clone();

   // pick a random index to mutate
   int max = x.length - 1;
   int min = 0;
   Random rand = new Random();
   int index = rand.nextInt((max - min) + 1) + min;

   // Generate a new random number for that index
   xNew[index] = -10 + Math.random() * (10 - -10);

   return xNew;
}
```

1.7 Non-Uniform Mutation Algorithm

My implementation of a non-uniform mutation algorithm chooses a random index of X and then mutates this value by adding or subtracting a random value.The mutation strength, i. e., the size of this change, decreases over time.

This implements the algorithm: $\triangle(t, y) = y \cdot 1 - r^{1-\frac{t}{T}})^b$ which decreases the size of the mutation change as the number of iterations, t, increases. In my code this is also a separate function called NonUniformMutationDelta().

```
public static double[] NonUniformMutation(double[] x,
      int currentIterationNumber, int numberOfIterations, double b) {
   double[] xNew = x.clone();

   // pick a random index to mutate
   int max = x.length - 1;
   int min = 0;
   Random rand = new Random();
   int index = rand.nextInt((max - min) + 1) + min;
   double newValue;

   // randomly select a mutation
   if (rand.nextDouble() >= 0.5) {
      double nonUniformMutationDelta = NonUniformMutationDelta(
```

```
            currentIterationNumber, (100 - xNew[index]), b,
            numberOfIterations);
        newValue = xNew[index] + nonUniformMutationDelta;
    } else {
        double nonUniformMutationDelta = NonUniformMutationDelta(
            currentIterationNumber, (xNew[index] + 100), b,
            numberOfIterations);
        newValue = xNew[index] - nonUniformMutationDelta;
    }
    xNew[index] = newValue;

    return xNew;
}
```

1.8 Gaussian Mutation

Gaussian mutation adds a normally-distributed random value to each component of the search point. The size of this value is controlled by a parameter σ, which is usually called step size. I used the apache distribution library to generate a normal distribution with standard deviation of 1 and then applied the distribution * step size to each element in the X array.

```
public static double[] GaussianMutation(double[] x, double
    stepSize) {
double[] xNew = x.clone();

NormalDistribution dist = new NormalDistribution(0, 1);

for (int i = 0; i < xNew.length; i++) {
    double t = dist.sample();
    xNew[i] = xNew[i] + (stepSize * t);
}

return xNew;
}
```

1.9 (1+1) evolution strategy with 1 5-rule

This evolutionary algorithm changes the step size based on how well the algorithm is doing according to the fitness function. It keeps a track of the number of successful & unsuccessful iterations and adjusts the step size accordingly (If this is larger than 1 5 we double σ , if it is smaller σ is halved).

This algorithm uses the Gaussuan mutation as described in the previous section and the changes are implemented in the $(1 + 1)$ evolution function. Essentially, if the success rate of the evolutions after i iterations is > 0.2 then the step size is doubled, else if it is < 0.2 then it is halved. This has the effect

of broadening & narrowing possible mutation space for each iteration based on whether the algorithm is a long was off the local optimum.

1.10 Software

I have used the Eclipse IDE to program the assignment in JAVA, Microsoft Excel has been used as a spreadsheet application to process the data and produces graphs of the results. I have also uses LaTeXto write & produce the report.