University of Waterloo
Faculty of Engineering
Department of Electrical and Computer Engineering

ECE 457A - Assignment 3

Prepared By:
Mohammed Ridwanul Islam - 20492491
Vinay Padma - 20474755
Aayushya Agarwal - 20468915

Thursday, July 26, 2016

1 Genetic Algorithm PID

a)

A representation for the solution is given by a nx4 matrix where each row represents a solution and the 4 columns are used for Kp, Ti, Td and the fitness function. An example of a solution is

[8.2962 7.9795 2.2777 0.0134]

Where Kp = 8.2962, Ti = 7.9795, Td = 2.2777 and the fitness = 0.0134

b) The fitness of a solution is given by the inverse of ISE (1/ISE). ISE determines the error of the PID controller. A larger ISE is a larger error. So the inverse would mean that a larger value corresponds to a better PID. So fitness = 1/ISE

c)

To solve this problem, we used a whole arithmetic crossover, as given by the code below. The function below gets a random number and checks if it less than the crossover probability. This represents the probability that two parents will crossover. If so, the first child's values are given by x*parent1 + (1-x)*parent2 and vice versa for the second child. If the crossover does not occur, each child inherits a single parent's genes.

```
function [ child1,child2 ] = a3_crossover(parent1,parent2, p_crossover
)
%UNTITLED2 Summary of this function goes here
%   Detailed explanation goes here

x = rand();

if(x< p_crossover)
    child1(1) = parent1(1)*x+(1-x)*parent2(1);
    child1(2) = parent1(2)*x+(1-x)*parent2(2);
    child1(3) = parent1(3)*x+(1-x)*parent2(3);

    child2(1) = parent2(1)*x+(1-x)*parent1(1);
    child2(2) = parent2(2)*x+(1-x)*parent1(2);
    child2(3) = parent2(3)*x+(1-x)*parent1(3);
else
    child1(1) = parent1(1);
    child1(2) = parent1(2);
    child1(3) = parent1(3);

    child2(1) = parent2(1);
    child2(2) = parent2(2);
    child2(3) = parent2(3);
end
ISE1 = perffcn(transpose(child1));
fitness1 = 1/ISE1;
ISE2 = perffcn(transpose(child2));
fitness2 = 1/ISE2;
```

```
child1(4) = fitness1;
child2(4) = fitness2;

return;

end
```

The mutation used is a random value with a lower bound and upper bound. The lower and upper bounds for each value (Kp, Ti, Td) are given by the question.

```
function [ mutated ] = a3_mutation( child, p_mutation )
%create random integer
x = rand();

%if the random integer is less than the probability, mutation occurs
if(x<p_mutation)
    %randomly select the child's Kp value
    mutated(1) = rand()*(18-2)+2;
    %randomly select the child's Ti value
    mutated(2) = rand()*(9.42-1.05)+1.05;
    %randomly select the child's Td value
    mutated(3) = rand()*(2.37-0.26)+0.26;

    %find the fitness function of the new child
    ISE = perffcn([mutated(1);mutated(2);mutated(3)]);
    fitness = 1/ISE;
    mutated(4) = fitness;

else
    %if the probaility is less than the random integer, no mutation
occurs
    mutated = child;
end
return
end
```

Parent selection is done using roulette wheel technique. We select 10 random parents based on their fitness function for crossover. The idea is to give parents with higher fitness, a better probability.

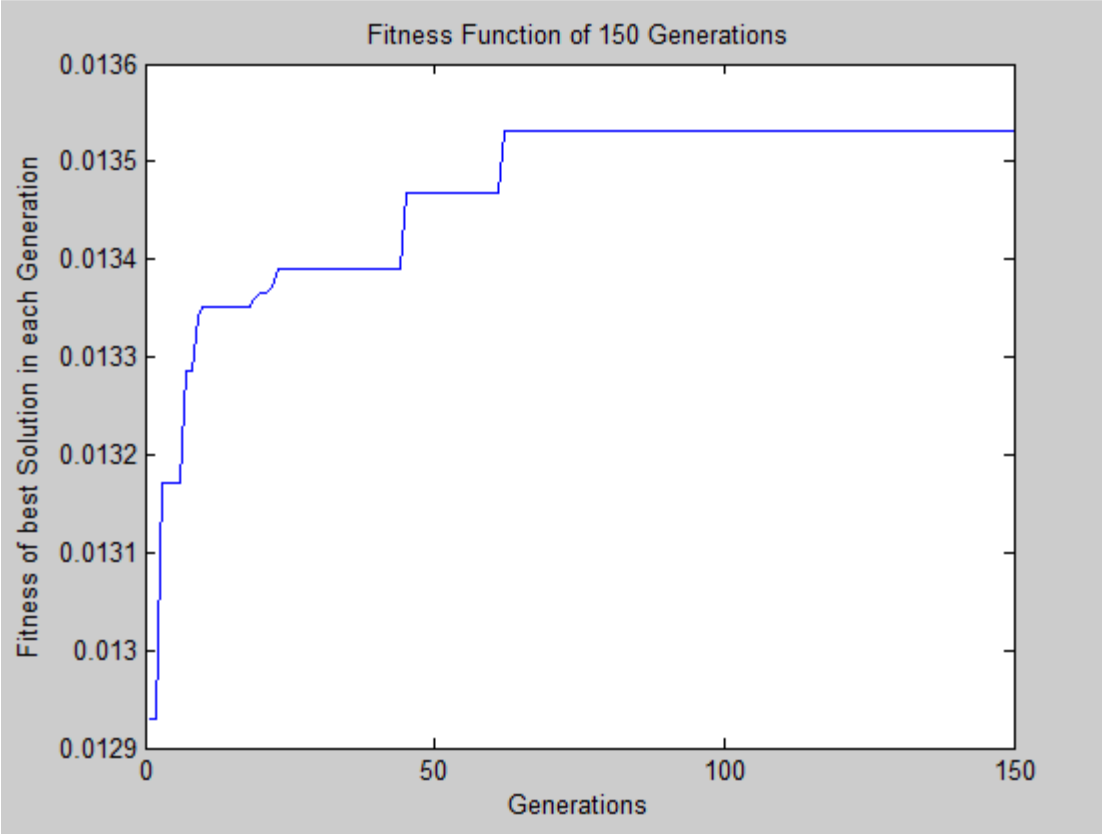
```
function [ selected_parents ] = a3_select_parents( parents )

copy = parents;
total_fitness = 0;
%array to hold selected parents
selected = [];

%loop through all the parents
for i=1:50
    %add up fitness function to find total
```

```
    total_fitness = total_fitness + parents(i,4);
end

%loop through to find 10 parents
for j= 1:10

    %create random integer
    x = rand();
    running_total = 0;
    %loop through all 50 parents
    for k= 1:50
        %find total fitness function up till current parent
        running_total = running_total+copy(k,4);
        %if the running_total/total_fitness>the random integer,select
it
        if(running_total/total_fitness>= x)
            %total fitness function is reduced
            total_fitness = total_fitness - copy(k,4);
            %add it to selected list
            selected = [copy(k,:);selected];
            %delete it from list of parents
            copy(k,:) = [0,0,0,0];
            x=1000000;
        end
    end

end

%sort the selected parents by the fitness function
selected_parents = sortrows(selected,-4);

return;

end
```

To complete the survivor selection, we replace the two worst parents with the two best children, if there is an improvement. The code below assumes the sorted_children array is sorted by fitness function. It takes the best child (sorted_children(1,:)) and checks if it is greater than the worst two parents in the population. If so, it replaces them. Similarly, we check if the second best child (sorted_children(2,:)) is better than the worst parent.

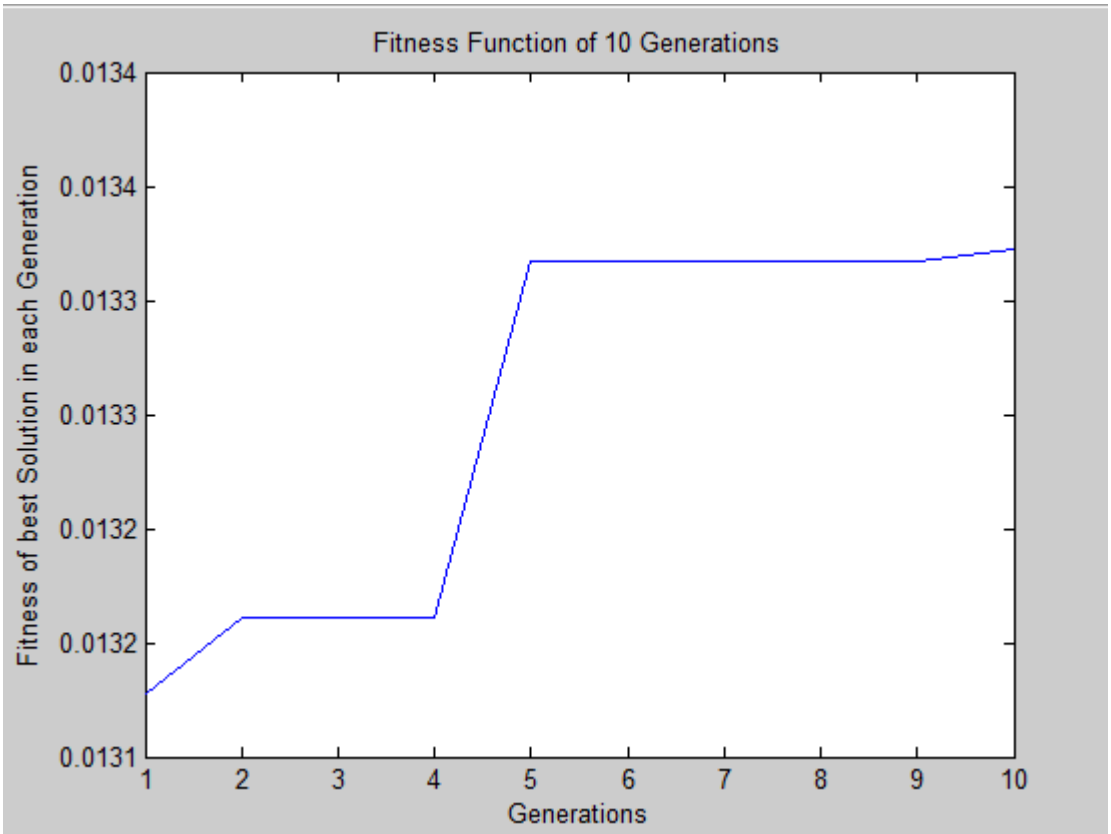
```
if(sorted_children(1,4)>population(end-1,4))
        population(end-1,:) = sorted_children(1,:);

        if(sorted_children(2,4)>population(end,4))
            population(end,:) = sorted_children(2,:);
        end
    else
        if(sorted_children(1,4)>population(end,4))
            population(end,:) = sorted_children(1,:);
        end
    end
```

d) The graph below shows the best fitness over 150 generations. There is an improvement from 0.0129 to 0.0136 (which is the inverse of ISE). One thing to note is the quick jump initially in the fitness and then the slow and steady improvements in later generations. This is a characteristic of GA, since it explores during the initial generations and exploits in the later.

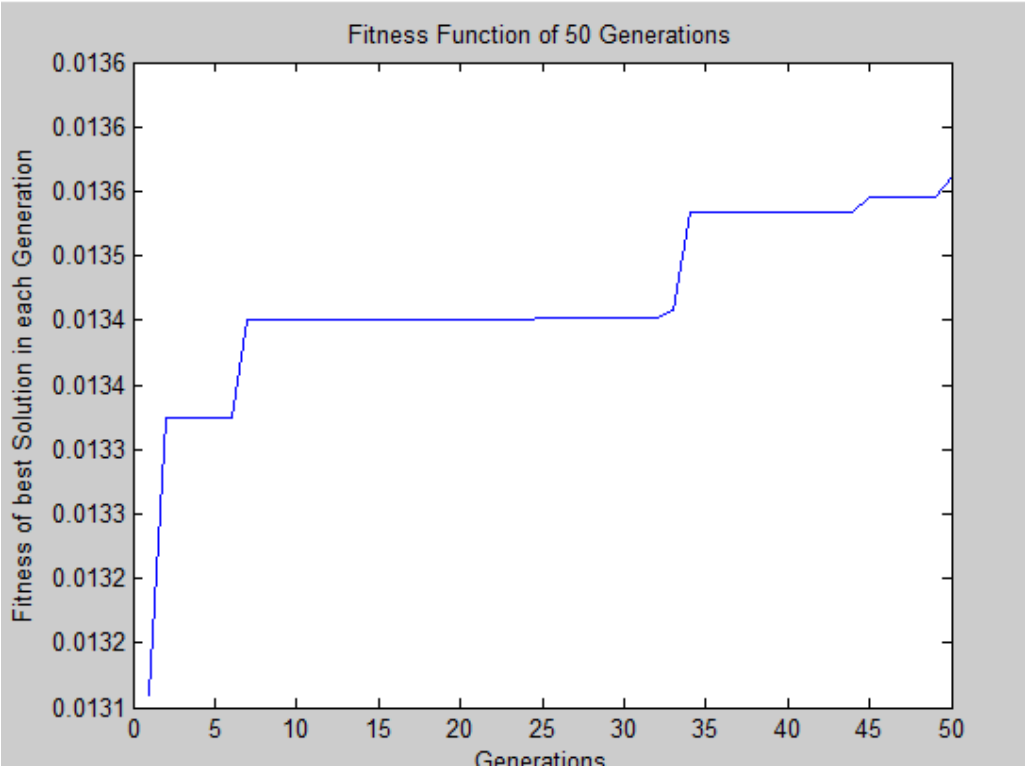


e) Below are graphs of the same function with different number of generations; 10, 50 and 100. Notice that as we increase the number of generations, the fitness function has a larger improvement. The fitness function of 100 has a much larger improvement than the fitness function of 10 generations. This is because there is more time for the fitness to improve as we increase the number of generations.

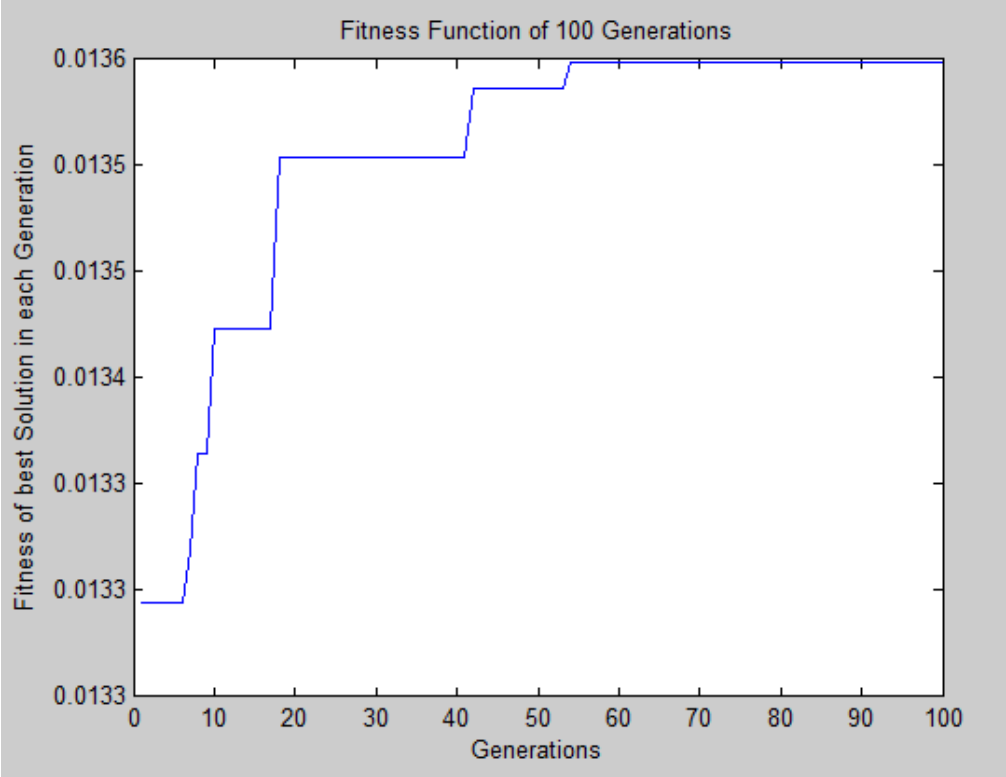
For 10 Generations:



For 50 Generations

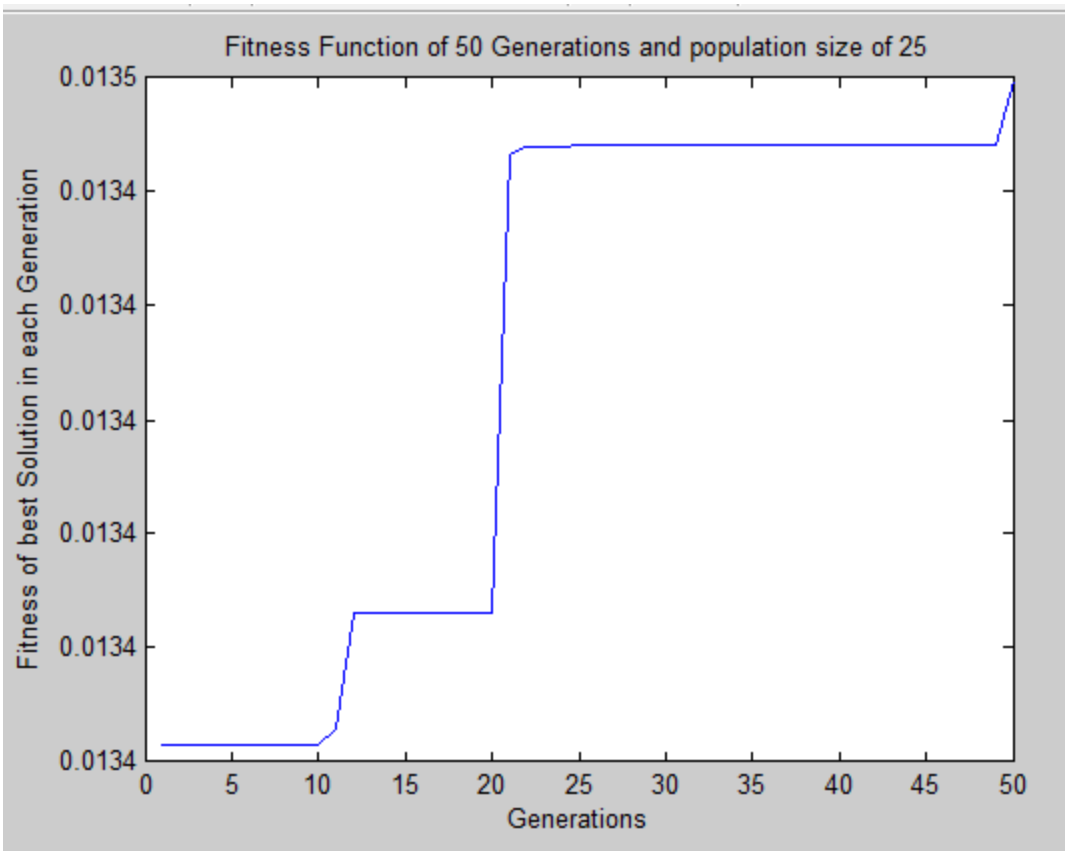


For 100 Generations:

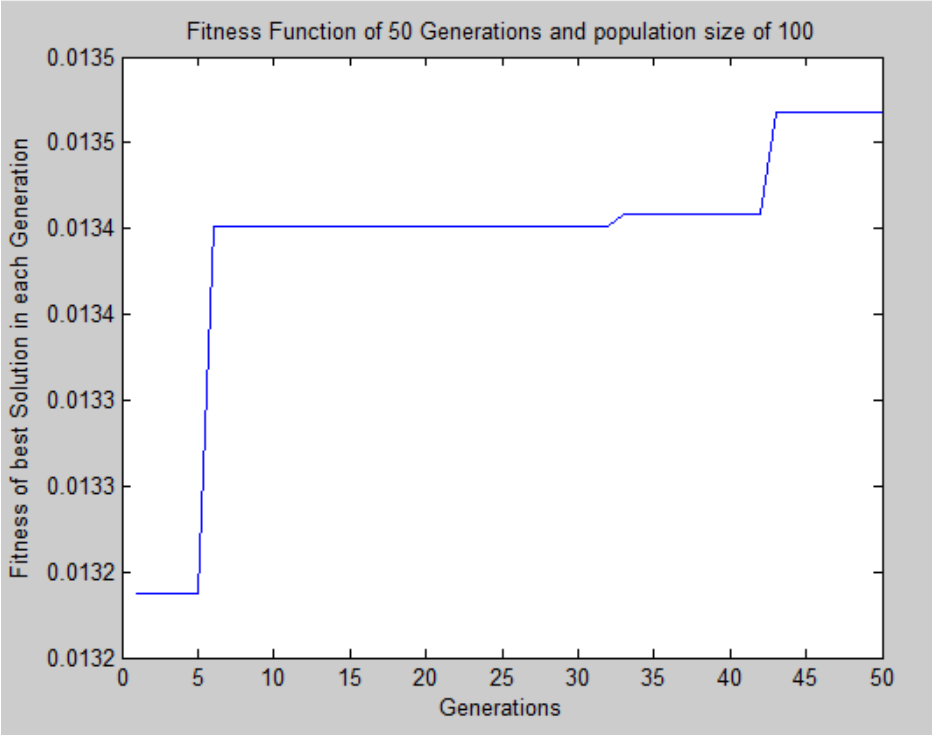


f) In this part of the experiment, we observe the effects of changing the population size. The plots below show the best fitness of each generation for different population sizes. The fitness function rapidly increases as the population size increases. As we increase the population size, there is a wider range of parents and genes. Therefore, by increasing the population size, we get a larger diversity and can find optimal solutions faster. Also, the increased diversity allows for a larger improvement. The fitness plot for a population size of 25 has a much smaller incremental improvement compared the fitness when the population size is 150.

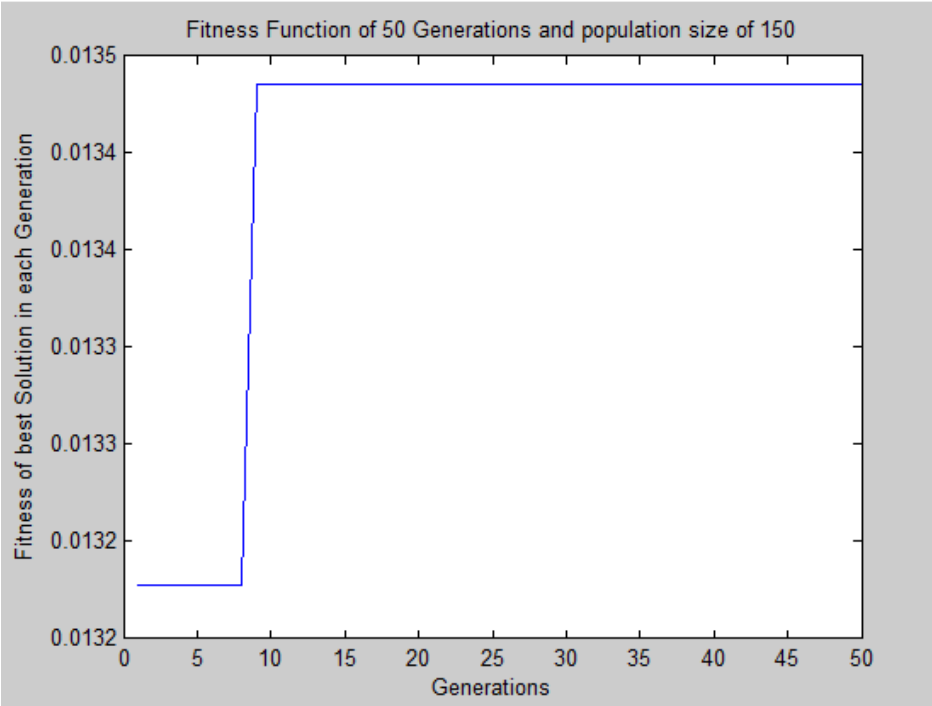
For a population size of 25:



For a population size of 100:



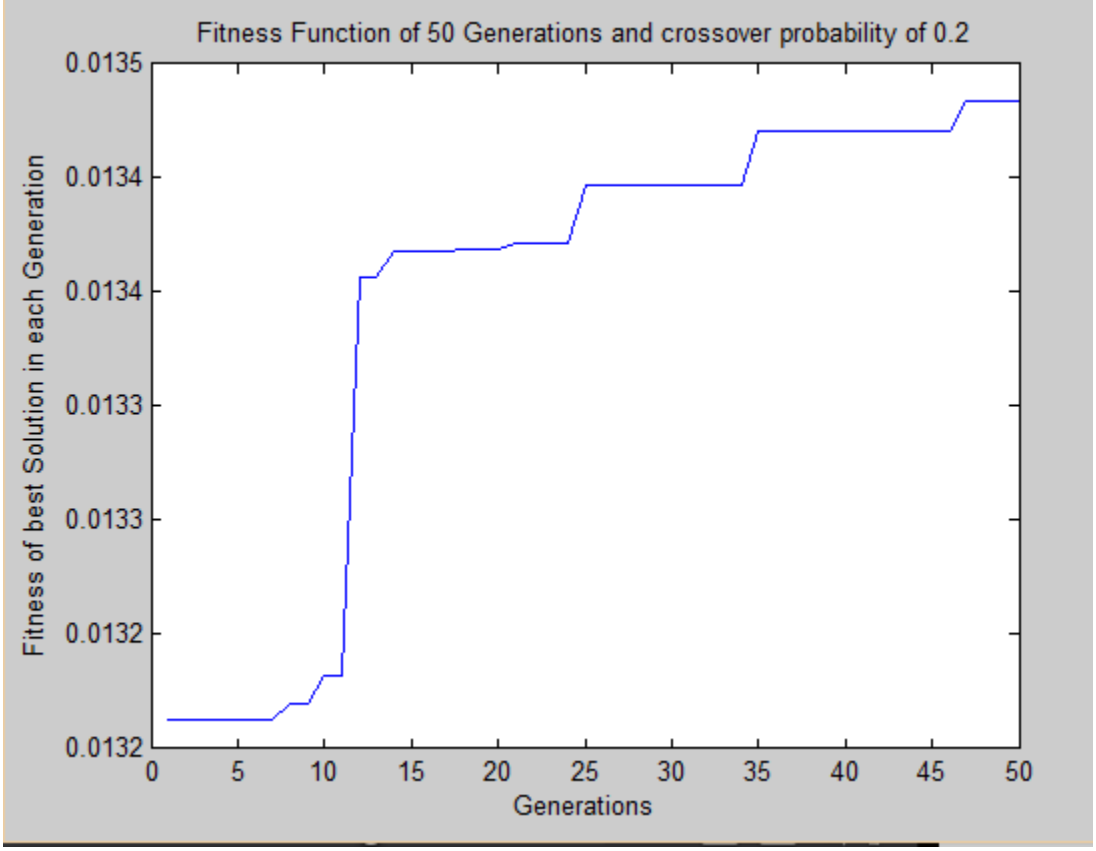
For a population size of 150:



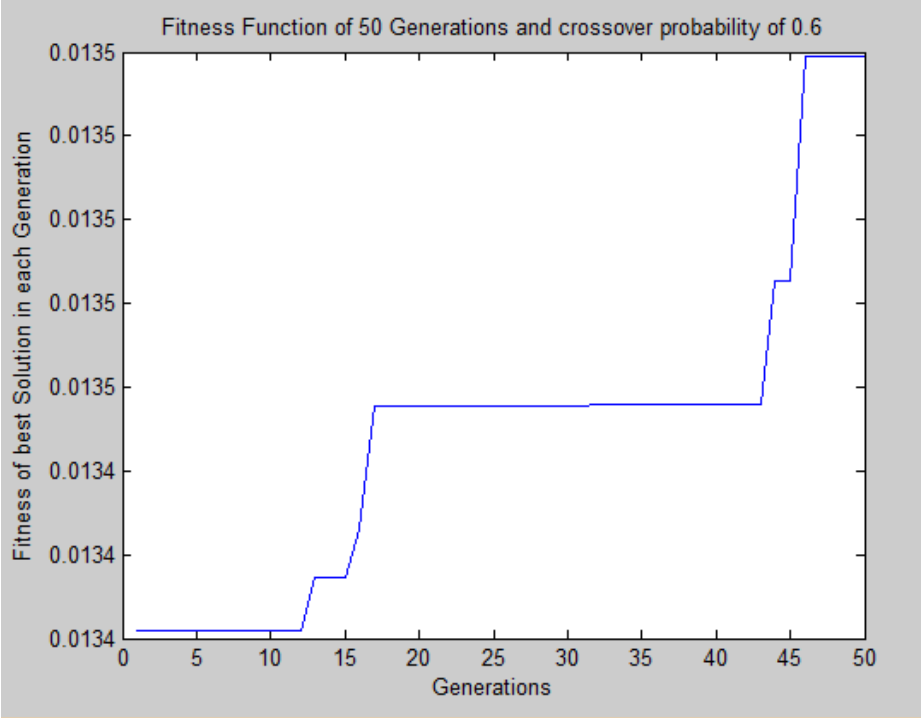
g) In this part of the experiment we change the crossover and mutation probabilities to observe the effects on the fitness in each generation.

For the following graphs, we observe the change in the fitness function as we change the crossover probability and fix the mutation probability to 0.25.

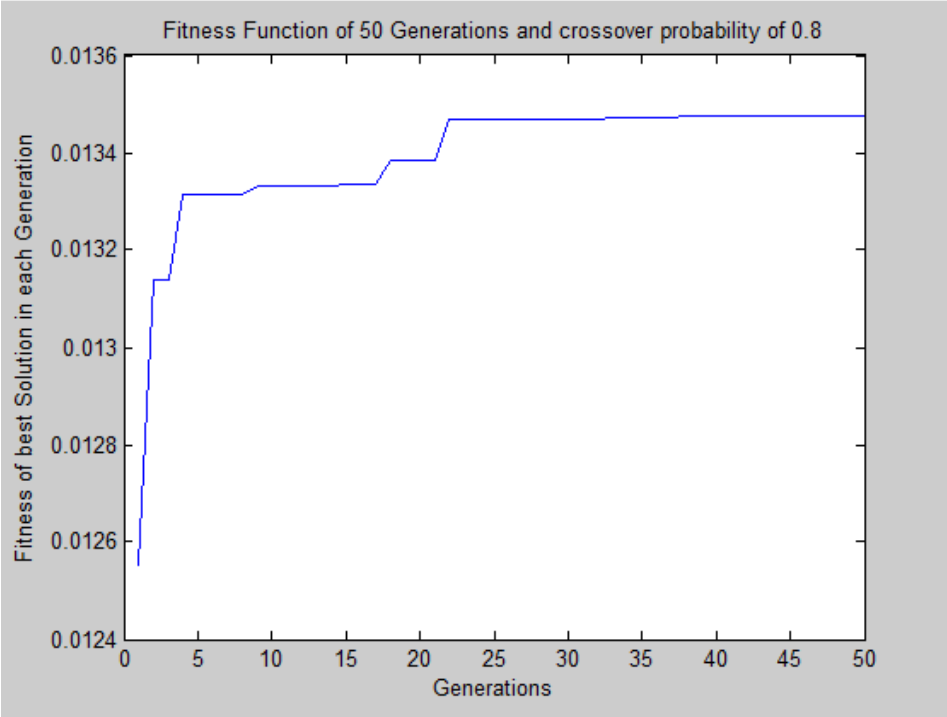
For a crossover probability of 0.2:



For a crossover probability of 0.6:



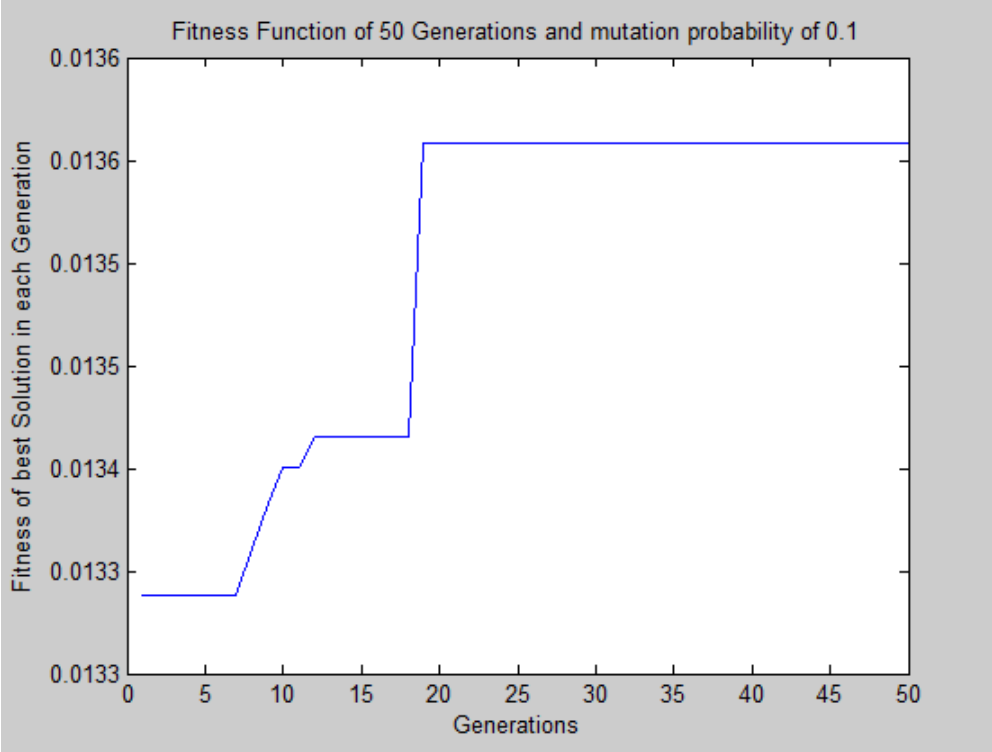
For a crossover probability of 0.8:



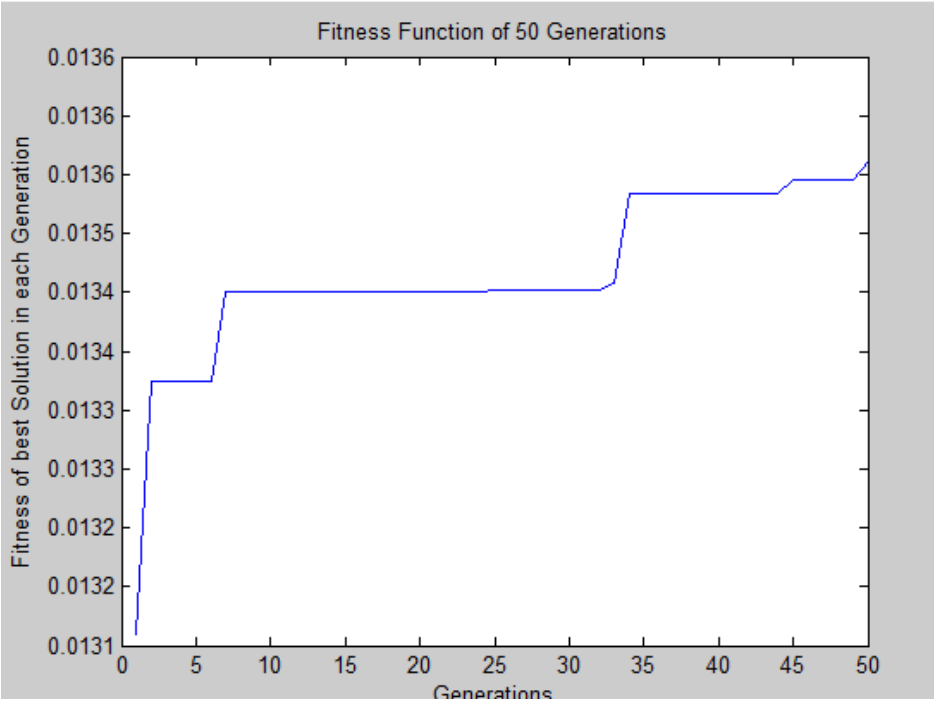
As we increase the crossover probability, it is more likely that a crossover will occur, resulting in more exploration than exploitation. This is seen in the graphs above, since the initial rate increases much more rapidly as we increase the crossover probability. This is a sign of exploration.

The plots below show the effect of varying the mutation probability on the fitness in each generation. In the experiments below, we fix the crossover probability to 0.6.

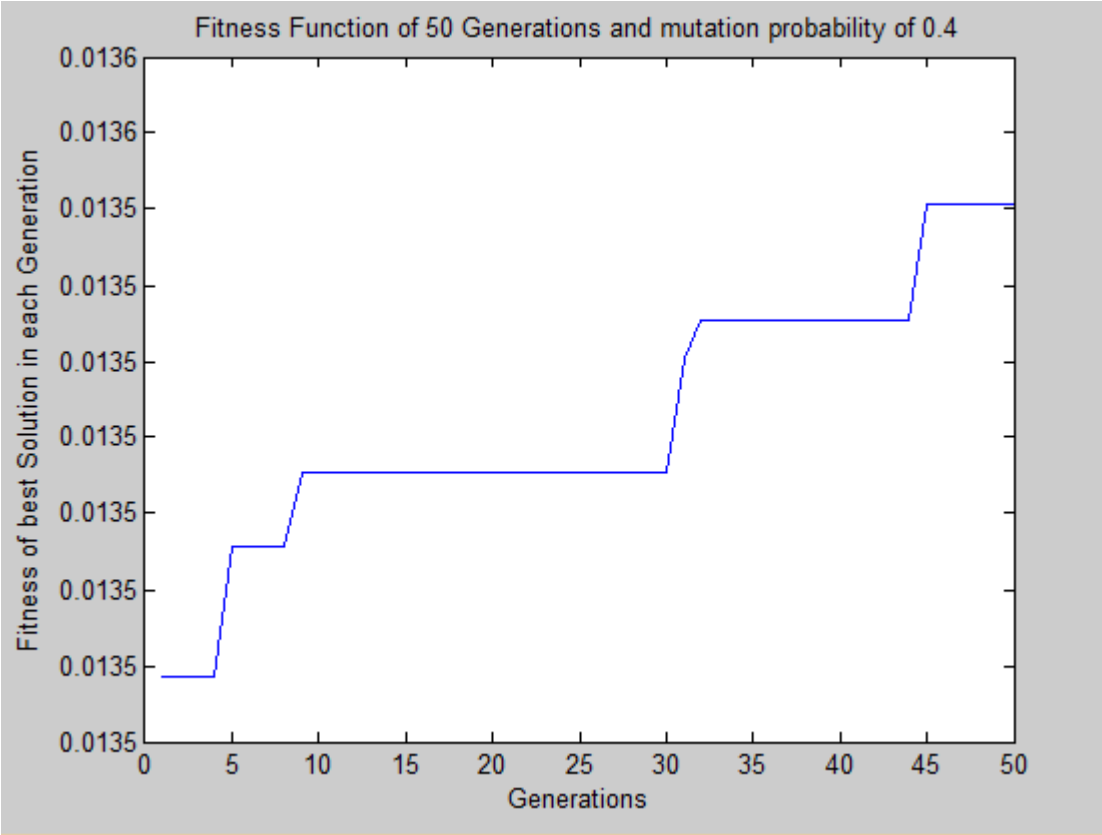
For a mutation probability of 0.1:



For a mutation probability of 0.25:



For a mutation probability of 0.4:



Increasing the mutation probability results in more mutations. This results in more exploitation and as we can see from the graphs above, there is more of a gradual improvement towards the later generations, which is a sign of exploitation. Exploitation is achieved in GA using mutations. Therefore with a higher mutation probability, we experience greater exploitation.

2 Ant Colony Optimization

PART A – NETLOGO Simulation

The following set of tests were carried out with constant positions of food sources as follows:

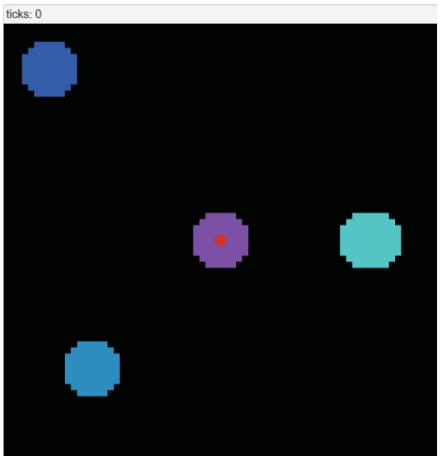


Figure 1: Position of food sources

Table 1: Showing the finish time with different population, diffusion and evaporation rates.

Population	Diffusion Rate	Evaporation Rate	Finish Time
30	40	10	9310
50	40	10	5103
100	40	10	1100
30	80	10	7250
30	40	20	6920
30	80	20	7410
100	80	20	3420

As we can see from the first set of results with the diffusion and evaporation rate kept constant as the population of ants is increased the finish time reduces significantly.

Next we see that with constant population and evaporation, an increase in diffusion rate decreases the finish time of the problem. This makes sense because the ants are more spread out and more explorative allowing for foraging to happen quicker.

We also notice from the table that with an increase in evaporation rate while everything else is constant reduces the finish time.

Explanation of behaviour

Concentration of pheromones on the closest food source with the shortest distance causes it to be foraged out first as can be seen here:

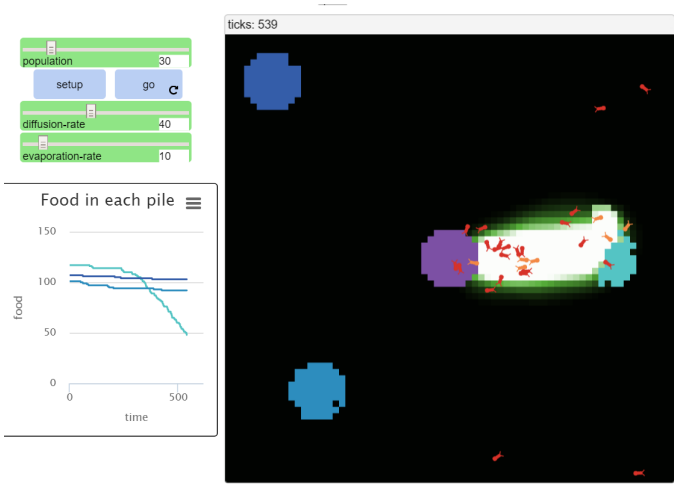


Figure 2: Explanation of behaviour (I)

The ants then get distributed and their behaviour is random until they come across more food sources.

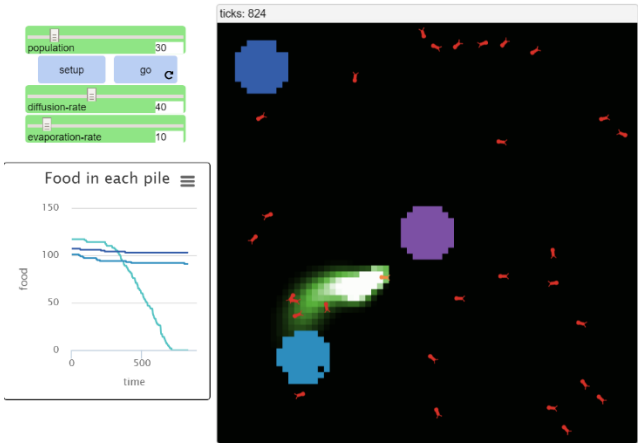


Figure 3: Explanation of behaviour (II)

More pheromones being generated for the closest food source once again:

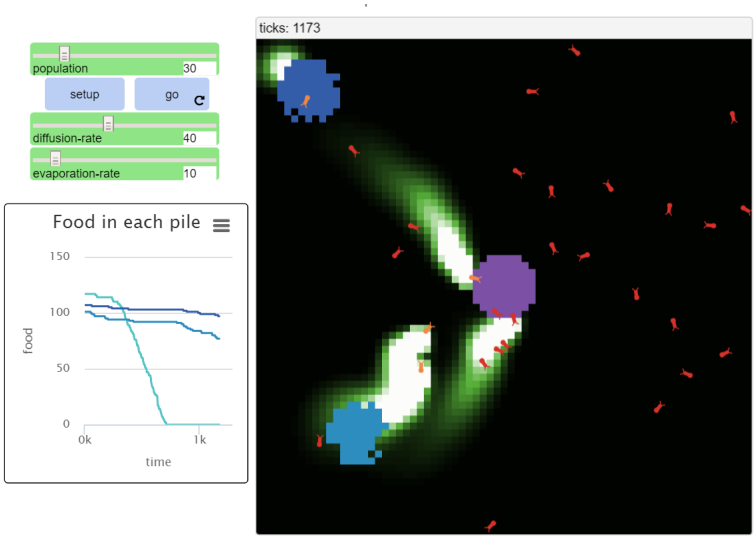


Figure 4: Explanation of behaviour (III)

This is how ants find the shortest path to food sources.

Change in positions

In this set of tests, the position was changed to have two food sources equidistant. We expect that both food sources will be foraged at equal rates.

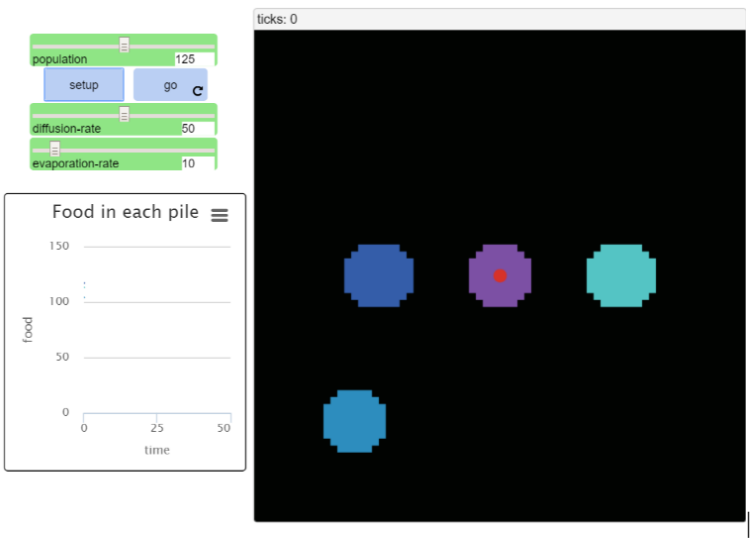


Figure 5: Change in positions

Time taken to completion: 667

As expected the ones with the shortest distances get eaten up first. In this case since they have equal distance, both of them get eaten up at the same rate.

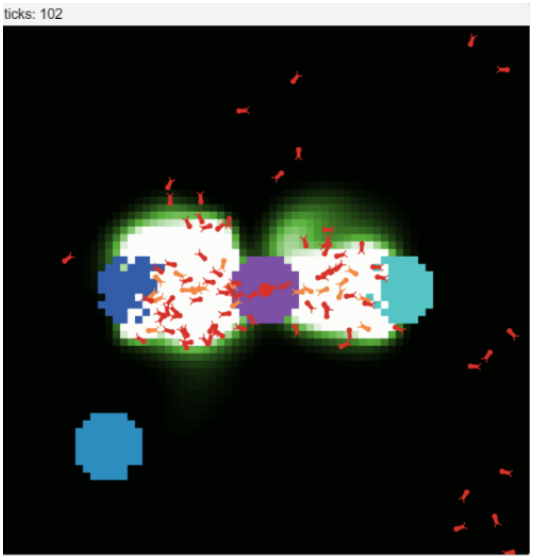


Figure 6: Pheromone distribution with equidistant food sources.

Extremely high evaporation rate

In this test we set the evaporation rate to 100. This evaporates all the pheromones so the ants have a sort of random exploratory search as expected because there is nothing to guide the pathway.

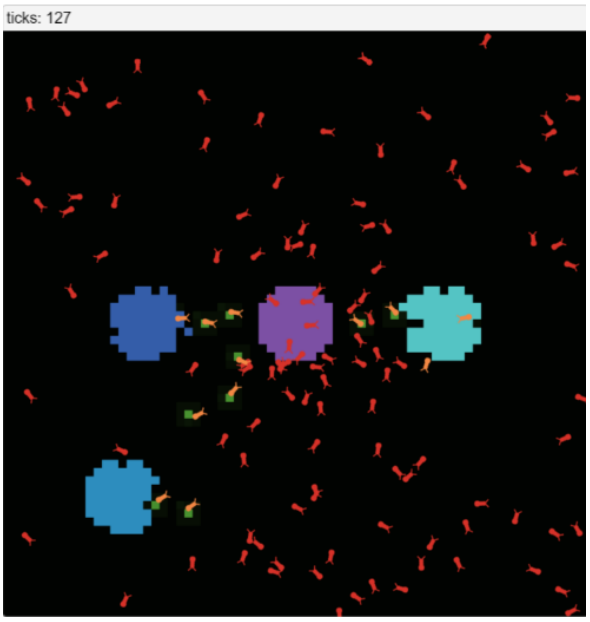


Figure 7: Behaviour of ants with an extremely high evaporation rate.

PART B – Ant Colony Optimization of Bays29

Algorithm Overview

For the ant colony method, the way our optimization algorithm works is we start off using the following properties:

1) Initial pheromone values set to 1000.
2) Initial ant population is set to 15.
3) At each node *i*, the ant has a choice to move to any of the *j* nodes connected to it. Each node $j \in Ni$ connected to *i* has a probability to be selected by ant *K* equal to:

$$p_{ij}^{k} = \begin{cases} \dfrac{\tau_{ij}^{\alpha} / d_{ij}^{\beta}}{\sum_{n \in N_i} \tau_{in}^{\alpha} / d_{in}^{\beta}} & if \quad j \in N_i \\ 0 & if \quad j \notin N_i \end{cases}$$

Formula 1: used to evaluate the probability of visiting the city

A random number is generated between [0,1], if the number is greater than the city with the highest probability, visit the city. If not visit a random city. This is our *transition rule.* Once a city is visited it is added to a taboo-array list.

4) Initially alpha and beta values are set to $\alpha = \beta = 1$. Which says that the local vs. global search ability is of equal weightage of the ant.
5) To deposit pheromones we use the *online delayed pheromone update rule (ant cycle method).* After the ant finishes building one solution, it traces the path backward by examining an solution which is an array list of cities visited. As it traces the path backward, it updates the pheromone trails on the visited arcs according to the solution quality by the following rule:

$$\tau_{ij} = \tau_{ij} + \Delta\tau$$

Formula 2: Extra pheromone deposited on the edge to the city visited.

6) Delta is determined as follows:

$$\Delta\tau_{ij} = \frac{Q}{L^{k}}$$

Formula 3: For delta tau

L^k is the length of the path found by ant *k.* *We use a value of 2000 for Q.*

7) Once the pheromones have been laid down. The pheromone is evaporated before the next ant starts constructing a solution as follows:

$$\tau_i = (1-\rho)\times\tau_i, \quad \rho \in (0,1]$$

Formula 4: Evaporation of pheromones using a set evaporation rate (rho).

8) Once all the ants have finished constructing a solution one iteration is finished.
9) Store the best solution amongst all ants.
10) If this solution is better then previous iterations, update global best solution.
11) The next iteration is started but if we want use the ***online pheromone update*** then another run of evaporation of pheromones is run.
12) Repeat the same process mentioned above for each iteration.

Code overview

To store the coordinates that were given to us we use a Point2D[] type array, which is a built in Java type used to store (x,y) coordinate tuples. The coordinates array stores an array of Point2D values.

Phermone values are stored in a symmetric 2-dimensional array an are initialized to 100 for all nodes except the diagonals. Because there is no pheromones from the edge between city 3 to city 3 for e.g.

We use an ant data structure which has certain key properties like a Point2D current_node variable which stores the position of the ant. A taboo list array list storing city indexes visited and a solution list. It also has a constructor to put it to a random city when initialized.

We have a best_ant variable which stores the iteration best ant. A global_best_cost and iteration_best_cost variable as well.

At the beginning we carry out all the initializations and initialize our coordinates array and our pheromone matrix.

Next we carry out a while loop until all num_iterations is decremented to zero. In each while loop, we have another for loop until each ant has a solution. Then generate a random number between 0 and 29 and create a new instance of an ant and set its current node to that random number.

Next we call the generate_solution function. In the generate_solution function the ants current (initial starting city) is placed in taboo list and in the solution list.

A while loop is called which will iterate until the solution array is of size 28. We then go over each city i=0 to i<29 and check if i'th city is in the taboo list. If yes, do nothing, if not then calculate the sum of the distance from current city to every neighbor i.e. total (the denominator of the probability function).

Next we go through another for loop from i=0 to i<28, and for each I'th city check if it is in the taboo list. If not calculate the probability of visiting that city and also the update the best so far probability and store it into best_prob and store the index in best_city.

Once that is done if there are cities left to be visited, generate a random number between 0 and 1and see if it is better then best_prob. If yes add the city to the solution and taboo list and update the ants current_node.

If not, visit a random city and add it to solution and taboo list and update the ants current_node.

Reset the best_prob, best_city and set cities_left_to_visit to false and continue with the while loop until the solution has 28 values in it.

Once the while loop has been executed, we call the update function, which updates the ant's cost variable by summing up the Euclidean distance between each cities in the solution array.

Here we also carry out the online delayed pheromone update and lay the pheromone as discussed above with Q = 2000.

We then carry out the evaporatePhermone() function which evaporates the pheromones by an update_factor.

Now that one ant has finished generating its solution, we do some check to see if it is the best solution in the current iteration, if so we update iteration_best_cost and iteration_best_solution. Similarly repeat the loop for all ants. Once the first iteration is done, if we want to do offline pheromone update then we call the update function again but this time with the argument as the best_ant in that iteration. The update function lays extra pheromones on the best path in that iteration.

We then update the global_best_cost and global_solution if the iteration solution is better then other iterations solutions.

Repeat for all iterations.

Results

2.6.1 a) Change the values of pheromone persistence constant 3 times.

The following shows the console output from 3 runs using different parameters. **Offline pheromone update is ON.**

Evaporation factor: 0.2
Alpha: 1
Beta: 1
Num ants: 13
Num iterations: 80

```
Global best cost: 9480.753849853065
```

```
Global best solution: [27, 8, 24, 1, 28, 6, 12, 9, 26, 29, 3, 5, 21, 2,
20, 10, 4, 15, 18, 14, 22, 17, 11, 19, 25, 7, 23, 16, 13]
```

Evaporation factor: 0.3
Alpha: 1
Beta: 1
Num ants: 13
Num iterations: 80

```
Global best cost: 9778.300660245313
Global best solution: [3, 29, 26, 5, 9, 6, 12, 28, 1, 4, 15, 18, 14, 22,
17, 11, 19, 25, 7, 23, 27, 8, 24, 16, 13, 10, 20, 2, 21]
```

Evaporation factor: 0.14
Alpha: 1
Beta: 1
Num ants: 13
Num iterations: 80

```
Global best cost: 9635.345248306043
Global best solution: [21, 2, 20, 10, 4, 15, 22, 14, 18, 17, 11, 19, 25,
7, 23, 27, 8, 24, 16, 13, 1, 28, 6, 12, 9, 5, 26, 29, 3]
```

2.6.2 b) Change the values of the state transition control parameter 3 times.

To change the state transition control parameters, we changed the alpha and beta parameters.

Evaporation factor: 0.15
Alpha: 3
Beta: 1
Num ants: 15
Num iterations: 80

```
Global best cost: 9621.14492516223
Global best solution: [29, 26, 5, 9, 6, 12, 28, 1, 21, 2, 20, 10, 4, 15,
18, 14, 22, 17, 11, 19, 25, 7, 23, 27, 8, 24, 16, 13, 3]
```

Evaporation factor: 0.15
Alpha: 1
Beta: 3
Num ants: 15
Num iterations: 80

```
Global best cost: 9866.271470268872
Global best solution: [8, 12, 6, 5, 9, 26, 29, 3, 2, 21, 1, 28, 24, 27,
16, 13, 10, 20, 4, 15, 18, 14, 22, 17, 11, 19, 25, 7, 23]
```

Evaporation factor: 0.15
Alpha: 1.5
Beta: 3
Num ants: 15
Num iterations: 80

```
Global best cost: 9773.778085757664
Global best solution: [16, 27, 8, 24, 1, 28, 6, 12, 9, 5, 21, 26, 29, 3,
2, 20, 10, 4, 15, 18, 14, 22, 17, 11, 19, 25, 7, 23, 13]
```

2.6.3 c) Change the population size twice

Number of ants is changed in this section.

Evaporation factor: 0.2
Alpha: 1
Beta: 1
Num ants: 2
Num iterations: 80

```
Global best cost: 10047.085269687028
Global best solution: [27, 8, 28, 1, 24, 16, 13, 10, 20, 2, 21, 5, 9, 6,
12, 26, 29, 3, 4, 15, 18, 14, 22, 17, 11, 19, 25, 7, 23]
```

Evaporation factor: 0.2
Alpha: 1
Beta: 1
Num ants: 7
Num iterations: 80

```
Global best cost: 9964.781274996829
Global best solution: [21, 2, 20, 10, 4, 15, 18, 14, 22, 17, 11, 19, 25,
7, 23, 27, 8, 24, 1, 28, 6, 12, 9, 5, 26, 29, 3, 13, 16]
```

Evaporation factor: 0.2
Alpha: 1
Beta: 1
Num ants: 20

Num iterations: 80

```
Global best cost: 9730.0297542851
Global best solution: [21, 24, 27, 8, 28, 1, 6, 12, 9, 5, 26, 29, 3, 2,
20, 10, 4, 15, 18, 14, 22, 17, 11, 19, 25, 7, 23, 16, 13]
```

2.6.4 d) Turn off online pheromone update

Offline pheromone update is switched off in this section.

Evaporation factor: 0.2
Alpha: 1
Beta: 1
Num ants: 13
Num iterations: 80

```
Global best cost: 9519.958710208139
Global best solution: [21, 2, 20, 10, 4, 15, 18, 14, 22, 17, 11, 19, 25,
7, 23, 27, 8, 24, 13, 16, 1, 28, 6, 12, 9, 5, 26, 29, 3]
```

Evaporation factor: 0.13
Alpha: 1
Beta: 1
Num ants: 17
Num iterations: 80

```
Global best cost: 9425.685843070773
Global best solution: [1, 28, 6, 12, 9, 5, 26, 29, 3, 2, 21, 20, 10, 4,
15, 18, 14, 22, 17, 11, 19, 25, 7, 23, 27, 8, 24, 16, 13]
```

3 Particle Swarm Optimization

PART A – NETLOGO Simulation

NetLogo was used to see the effects that different parameters have on the PSO.

Pop.	Speed Limit	Inertia	Att. PBest and GBest	Speed of Converge (ticks)	Found GOptima?
30	2	0.6	1.7	~60	25%
30	2	0.6	1.494	~40	60%
30	2	0.729	1.7	~52	0%
30	2	0.729	1.494	~80	20%
30	6	0.6	1.7	~30	40%
30	6	0.6	1.494	~32	80%
30	6	0.729	1.7	~34	0%
30	6	0.729	1.494	~35	20%
60	2	0.6	1.7	~60	40%
60	2	0.6	1.494	~50	20%
60	2	0.729	1.7	~50	60%
60	2	0.729	1.494	~50	40%
60	6	0.6	1.7	~20	20%
60	6	0.6	1.494	~23	20%
60	6	0.729	1.7	~25	60%
60	6	0.729	1.494	~25	70%

Table 3.1: NetLogo PSO results with varying parameters

Note: In the above table, each scenario is run 5 times and averaged in order to find the percentage of success and speed of convergence. This is done since the algorithm is stochastic. We acknowledge 5 runs is not a large sample size, but it is better than a single run.

From Table 3.1 one can see that speed of convergence is highly influenced by the speed limit of the PSO. Also, although there were not a lot of samples taken, there is a slight trend one can see with the population parameter: the more agents in the PSO, the greater chance there is for the PSO to find the most optimal solution. The parameters inertia_weight = 0.729 and c1=c2=1.494 are the parameters used while using the Constriction Factor. In the samples taken above we do not see any improvement using those parameters compared to inertia_weight = 0.6 and c1=c2=1.7. A viable reason is that, for the problem the NetLogo program is attempting to solve, the parameters (0.6, 1.7) are a more optimal set than (0.729, 1.494).

Comparing NetLogo Implementation to the Classical PSO

The equation for velocity is slightly different, it includes the following term in the equation:

$$different_{term} = (1 - particleInertia) * C_1$$

Or, for Global Best

$$different_{term} = (1 - particleInertia) * C_2$$

This is referred to in the code snippet below:

```
    set vx vx + (1 - particle-inertia) * attraction-to-personal-best *
(random-float 1.0) * dist * dx
    set vy vy + (1 - particle-inertia) * attraction-to-personal-best *
(random-float 1.0) * dist * dy
```

And

```
    set vx vx + (1 - particle-inertia) * attraction-to-global-best *
(random-float 1.0) * dist * dx
    set vy vy + (1 - particle-inertia) * attraction-to-global-best *
(random-float 1.0) * dist * dy
```

The author notes this change was included for the following reason:

"It was added because it allows the "particle-inertia" slider to vary particles motion on the full spectrum from moving in a straight line (1.0) to always moving towards the "best" spots and ignoring its previous velocity (0.0)."

PART B – Implementation of PSO

First a simple PSO was coded into MATLAB to solve the six hump camelback problem:

$$z = \left(4 - 2.1x^2 + \frac{x^4}{3}\right)x^2 + xy + (-4 + 4y^2)y^2$$

The simple PSO converged very slowly and did not frequently find a solution close to the optimal solution. Below is an image of a run with 64 agents, and a terminal condition of 100 iterations (c1=c2=2, inertia_weight = 1).

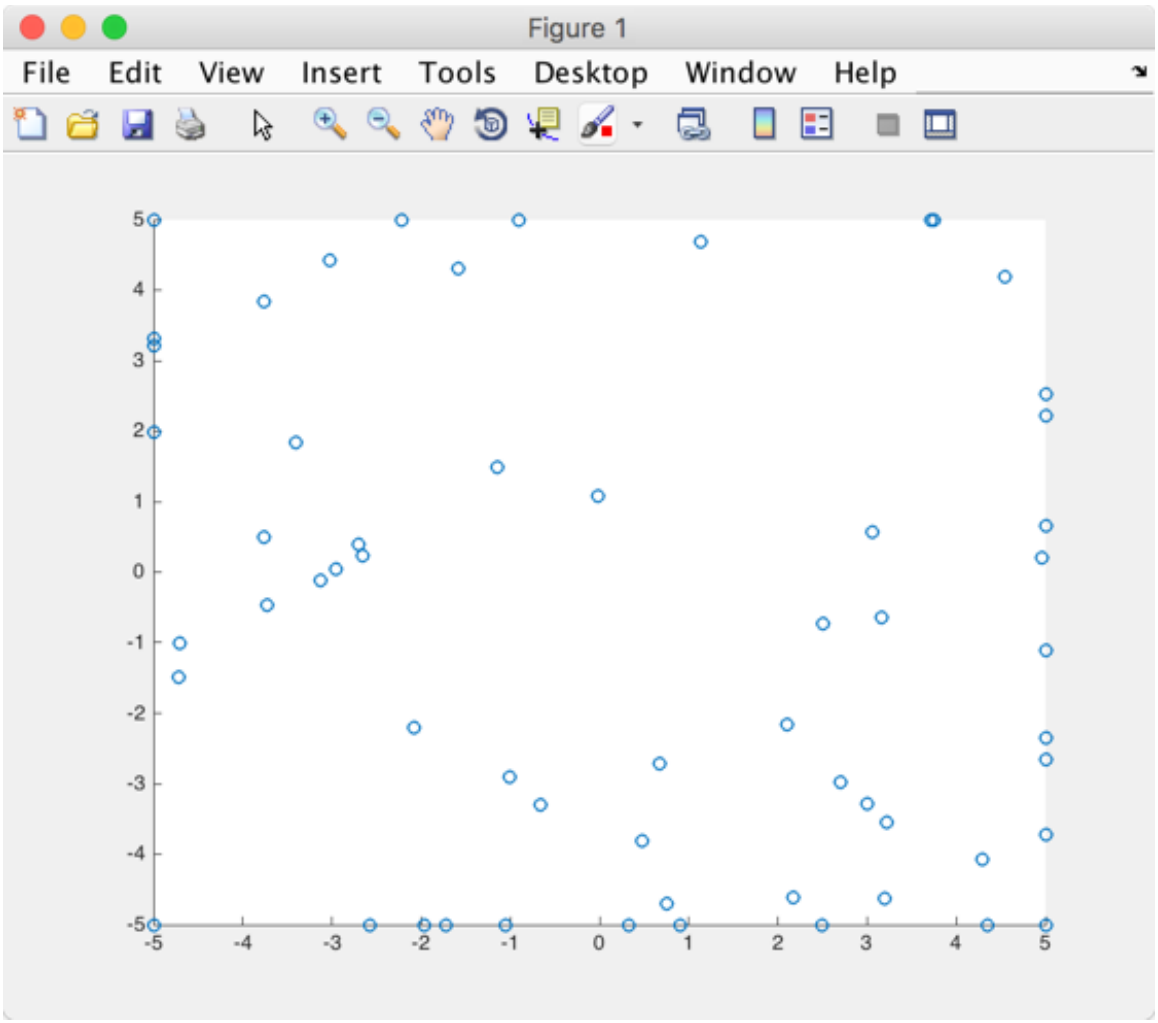


Figure 1 – Simple PSO after 100 iterations, 64 agents, c1=c2=2, inertia_weight = 1

Then the Inertia Weight version of the velocity update equation with Global Best was added. Under the same parameters, there was not much noticeable improvement (still very little/slow converging). Afterwards a V_max was introduced.

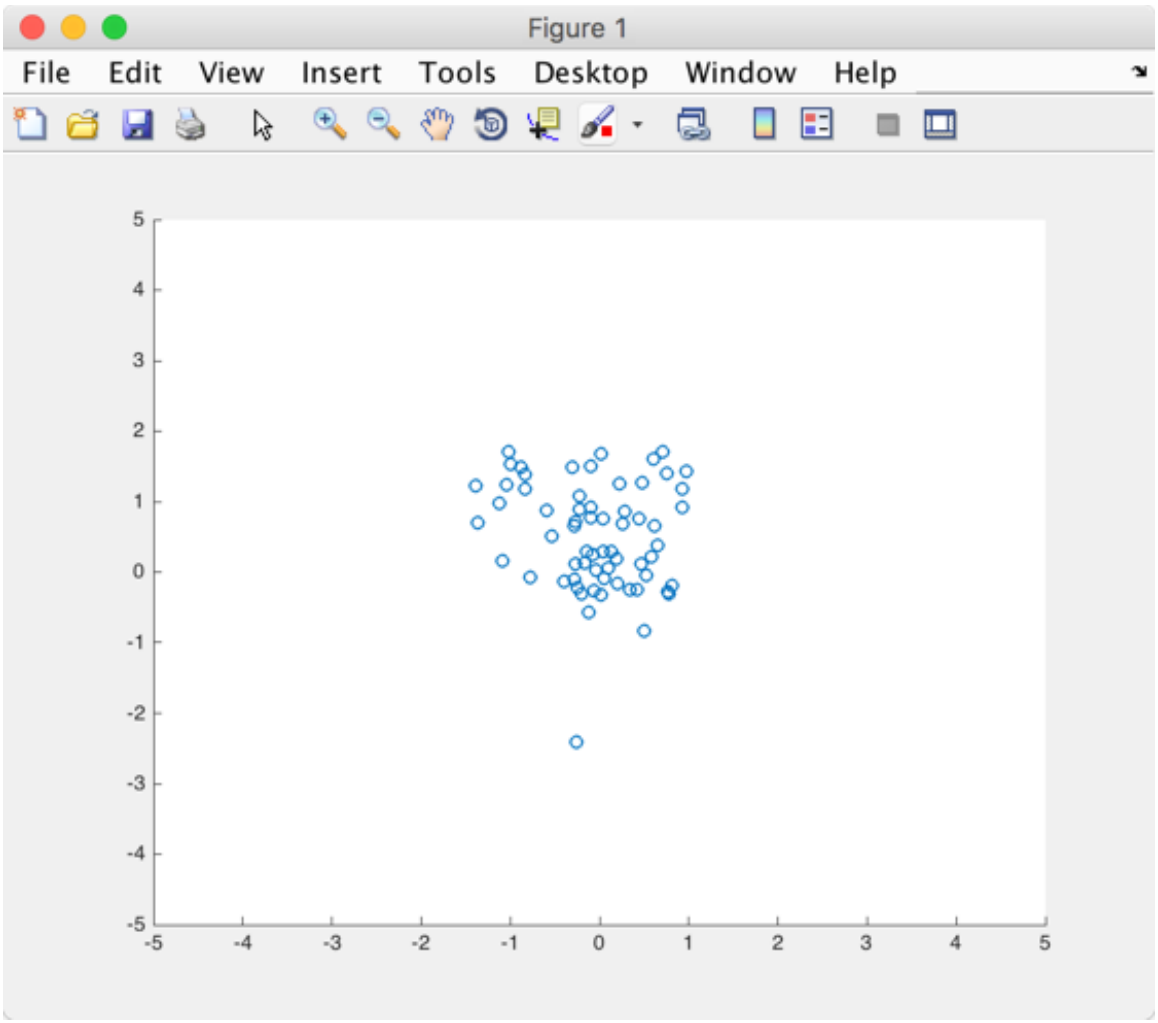


Figure 2 - PSO after 100 iterations, 64 agents, c1=c2=2, inertia_weight = 1. V_max = 1

From this we can see there is much more 'swarming' of particles. The prior implementation, without a limit on velocity tended to 'overshoot' the motion of particle so an equilibria was never reached. However, when velocity if forcibly slowed down, the algorithm begins to show some of its merits.

Next the Constriction Factor was applied to the PSO (inertia_weight = 0.792 and c1=c2=1.4944).

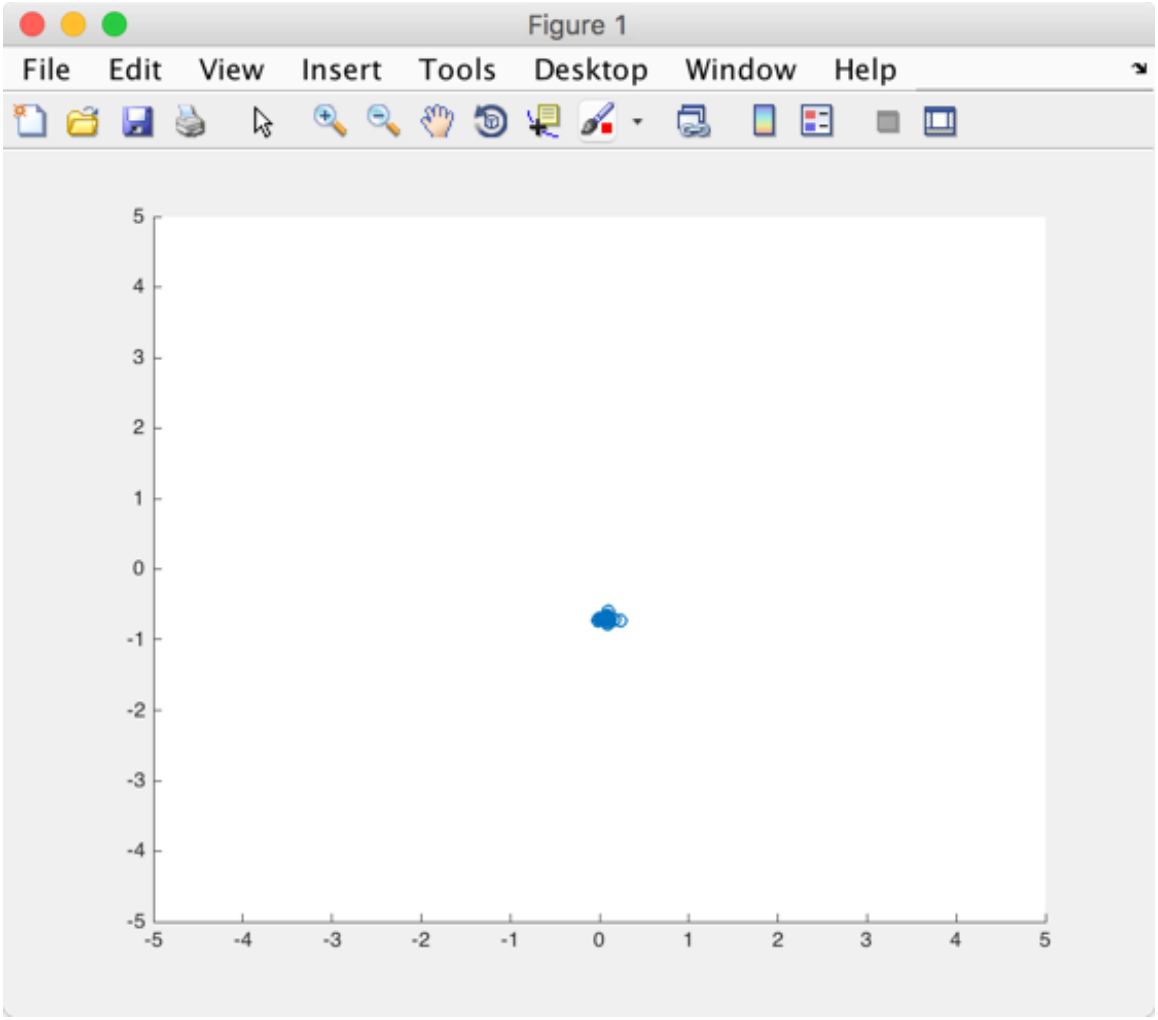


Figure 3 – PSO with the Constriction Factor. Population = 64, Max_iterations = 100. V_max =1

It is here we see consistent clustering and optimal solutions being found most of the time. Converging occurs very quickly and consistently.

Next the Guaranteed Convergence PSO [1] was added to the algorithm. Below is the GCPSO with the same parameters as the previous run using the Constriction Factor.

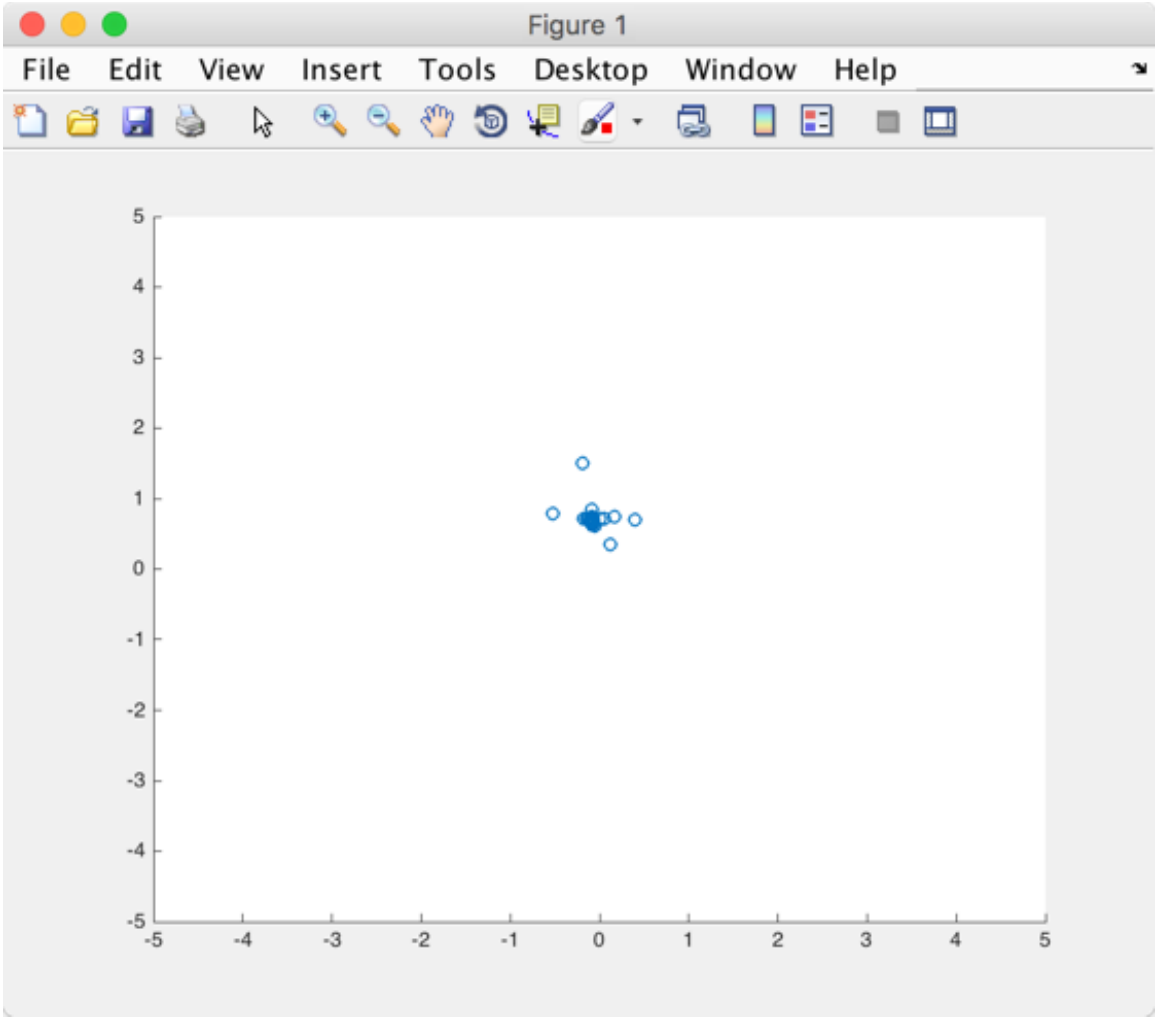


Figure 4 – GCPSO with the Constriction Factor. Population 64, Max_iterations = 100. V_max=1. Sc = 15, fc = 5

With the GCPSO, convergence still occurs but somewhat slower than in the case without it. This is puzzling as the purpose of the GCPSO is to improve convergence times. However there may be some differences in the applied algorithm when compared to the algorithm described in [1]. The paper does not specify what occurs happens to the GCPSO parameters (scalefactor, successes, failures) when the best particle changes (say gbest is located at particle 3, but then moves to particle 10 on a later iteration). The implementation of the GCPSO provided here resets all of these parameters to their initial values when another particle takes the position of gbest (scalefactor = 1, successes = 0, failures = 0). With a somewhat low number of iterations, the gbest particle changes frequently.

Code Overview

The implementation of PSO showcased in this report was done in MATLAB.

Data Structures

The particle swarm is represented by an Nx2 matrix called swarm (where N is the number of agents, or population of the swarm). Every row within the swarm matrix represents an agent, and this matrix also holds the current x position (column 1) and current y position (column 2).

Algorithm

1. Initialization
    a. Create a swarm matrix (Nx2) matrix, with agents at random allowable positions
    b. Initialize the pBest matrix = swarm matrix
    c. Evaluate all tuples in the swarm matrix
    d. Store the indices belonging to the best result into nBest
    e. Create velocity matrix (Nx2), with all elements set to zero
    f. Initialize GCPSO parameters
        i. p(0) = 1
        ii. sc = 15
        iii. fc = 5
        iv. successes = 0
        v. failures = 0
2. Main PSO Loop

While iterations < max_iterations:

For every agent in swarm:

a. Update velocity first
    i. Iterate over the velocity matrix, updating each cell using the following equations:

    IF particle is gBest:

$$v_x(t+1) = inertiaWeight \times v_x(t) - swarm_x(t) + pBest_x(t) + \rho(t) \times rand(-1,1)$$

$$v_y(t+1) = inertiaWeight \times v_y(t) - swarm_y(t) + pBest_y(t) + \rho(t) \times rand(-1,1)$$

$$where\ \rho(t+1) = \begin{cases} 2\times\rho(t), if\ successes > sc \\ 0.5 \times \rho(t), if\ failures < fc \\ \rho(t), otherwise \end{cases}$$

ELSE

$$v_x(t+1) = inertiaWeight \times v_x(t) + c_1 \times rand() \times (pBest_x(t) - swarm_x(t)) + c_2 \times rand() \times (nBest_x(t) - swarm_x(t))$$

$$v_y(t+1) = inertiaWeight \times v_y(t) + c_1 \times rand() \times (pBest_y(t) - swarm_y(t)) + c_y \times rand() \times (nBest_y(t) - swarm_y(t))$$

ii. Check that absolute velocity is within max velocity set out by parameters. If it isn't, set the velocity of the particle to maximum velocity allowed (or minimum, if the velocity is negative)

b. Update position

i. Update position based off of the following equation:

$$x_{i+1} = x_i + v_x$$

$$y_{i+1} = y_i + v_y$$

ii. Check the new position is within the boundary conditions, if not reset the position to the maximum allowable position (i.e. (-5.5, -6.5) -> (-5,-5))

iii. Evaluate function at new position.

1. Update the agent's row in pBest if the particle is at a personal best.
2. If the agent is the current gbest:
    a. Increment successes and set failures to 0 if a better solution was found
    b. Increment failures and set successes to 0 if a better solution was not found

Update nBest after every iteration of the while loop (Synchronous update), if a better pair of positions was found. If a new pair is found to have a better position, also reset GCPSO parameters (successes, failures and p(t) = p(0)).

4 Contributions

All three members of the group (Mohammed Ridwanul Islam, Aayushya Agarwal, Vinay Padma) contributed equally to this assignment and to this report.

5 References

[1] E. S. Peer, F. van den Bergh and A. P. Engelbrecht, "Using neighbourhoods with the guaranteed convergence PSO," *Swarm Intelligence Symposium, 2003. SIS '03. Proceedings of the 2003 IEEE*, 2003, pp. 235-242. doi: 10.1109/SIS.2003.1202274
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1202274&isnumber=27067