Homework #3

Kyle MacMillan, Shashwati Shradha

April 1, 2019

keywords: Swarm Intelligence, Ant Clustering Algorithm, Particle Swarm Algorithm, Simulated Annealing

See GitHub repository for Problem 1 code. Uses `Entities 0.0.12-preview2.1`.

Problem 1

1.1 Statement

Write a version of the Ant Clustering algorithm:

1. Create a grid of 200 x 200.
2. Randomly place 200 objects. 100 red and 100 blue.
3. Randomly place 500 ants on the grid. Ants are unloaded.
4. Repeat:
   (a) For each object
      i. Find the number of same colored objects in the adjacent cell neighborhood: local density.
      ii. Find discrete rules for pickup (unloaded ants) and drop-off (loaded ants) based on the local density.
      iii. Move ants one step randomly to an adjacent grid.
      iv. Apply pickup and drop-off[1].

Plot the result. Your goal is to find rules that cause the red and blue objects to be clustered together.

1.2 Method

Our method was a little unorthodox. It was realized from the start that this would be a good opportunity to create an interactive Ant Clustering Algorithm. So the goal was to create a world we could fly around in and watch the ants move in. We gave the player controls:

- Mouse Movement - Looks around
- W - Forward

[1]This was moved to the pickup/dropoff section

- S - Backward
- A - Left
- D - Right
- Left Shift - Increase player movement speed
- Up Arrow - Increase game speed
- Down Arrow - Decrease game speed
- Space Bar - Toggle ants
- Escape - Quits

Note: Simultaneous keys are allowed (such as W + Left Shift to move forward faster)

It would have been nice to set up some game options at the start but we ran out of time.

Objective. Our objective was to see if we could get ants to cluster different colors.

Evaluation function. Evaluation was visual.

Choice of representation. Ants are represented in a Dictionary. We utilized position as the key and the entity representing the ant/ball was the value. The entire project was represented in Unity because the Ant Clustering Algorithm (ACA) can be very easily mapped to an Entity Component System

Approach. Our approach to use a Dictionary instead of grid was due to the fact that the grid was going to be sparse, it would not scale well, and speed. Dictionaries have $\mathcal{O}(1)$ look-up, insert, and remove. This was critical to a performant system that could expand beyond the initial requirements.

We determined maximum size of the grid given by $width * height - 1$ and used this with an integer random number generator to assign a grid location. Ants and balls were applied to the grid utilizing this method. Ants and balls were stored in separate Dictionaries because the `C#` does not allow for duplicate keys. This also had the benefit of keeping the code more readable by knowing which Dictionary we were working on at the time.

After the setup it was necessary to code ACA. We utilized the book algorithm:

Algorithm 1 Simple Ant Clustering Algorithm

```
Initialize every ant and ball to a random unoccupied cell
while Not quit do
  for every ant do
    Compute locality of each ball
    if Ant unloaded AND cell occupied by item $x_{color}$ then
      Compute $P_p(x_{color})$
      Pick up item if $P_p(x_{color}) >$ Random [0, 1]
    else if Ant carrying item $x_{color}$ AND cell empty then
      Compute $P_d(x_{color})$
      Drop item $x_{color}$ if $P_d(x_{color}) >$ Random [0, 1]
    end if
    Move to random neighbor without ant (or ball if carrying)
  end for
end while
```

We began with the book Probability for Pickup/Dropoff:

$$P_p(x_{color}) = \left(\frac{k_1}{k_1 + f(x_{color})}\right)^2$$

$$P_d(x_{color}) = \left(\frac{f(x_{color})}{k_2 + f(x_{color})}\right)^2$$

We tried a plethora of options and found some simple rules that will be outlined in Section 1.3.

1.3 Results

All of our results can be found in this YouTube Playlist. Just a warning, one of the videos has music because I thought it was kind of boring. I removed the music after that video. The game is also available for play **without the need to install anything**[1].
The big thing we found was that ACA tended to behave best with some rules!

Pickup Chance:

- No balls nearby – 100%
- 1 ball nearby – 76%
- 2 balls nearby – 21%
- 3 balls nearby – 0%

[1]Tested on Windows 10 64 bit and Ubuntu 18.04

Dropoff chance:

- No balls nearby – 0%
- 1 ball nearby – 78.75%
- 2 balls nearby – 90%
- 3 balls nearby – 100%

Figure 1 shows the curve used for $P_p(x_{color})$ and Figure 2 shows the curve used for $P_d(x_{color})$ that yielded the best results.

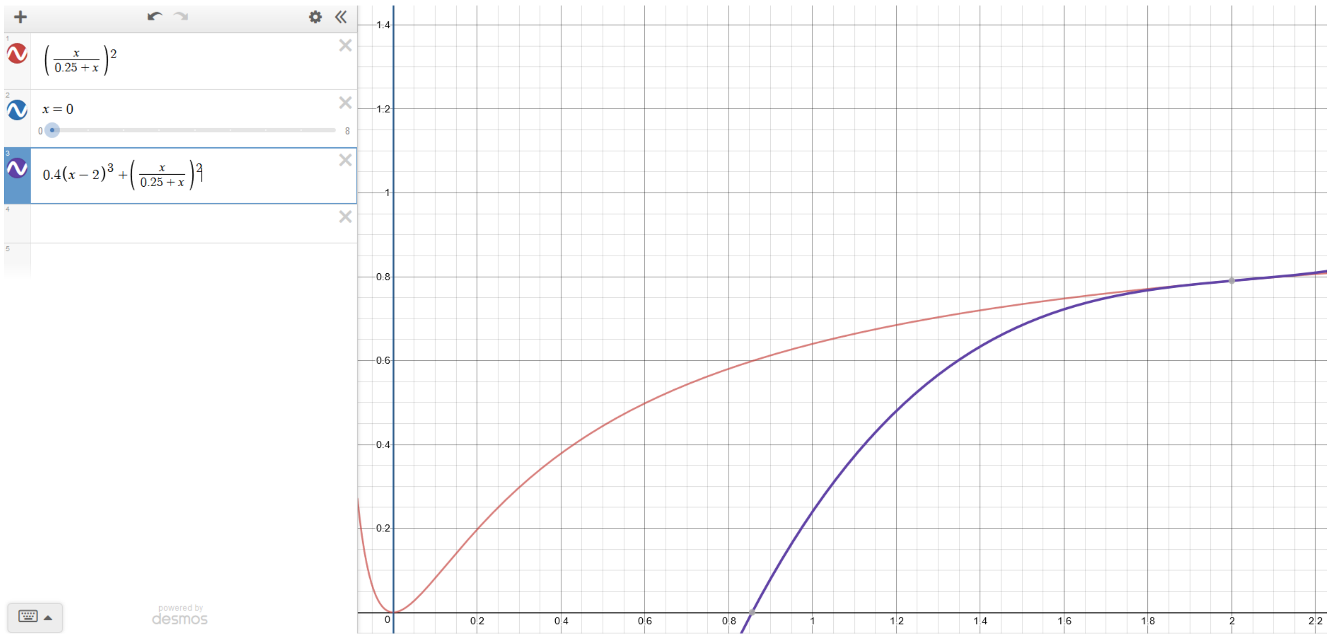


Figure 1: Probability for Pickup

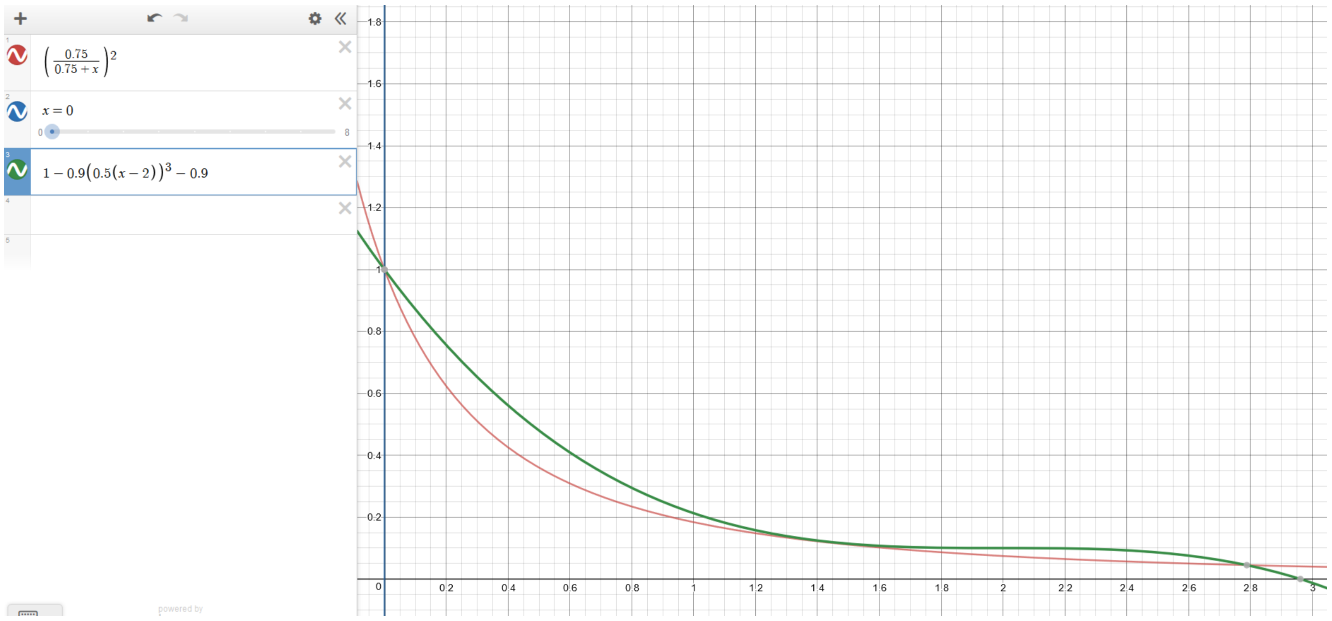


Figure 2: Probability for Dropoff

The reason for these rules was that for pickup we wanted a slightly higher rate of pickup around 1 and 2 balls but we determined it should not pick it up *at all* if there were 3 balls. With 500 ants there is a chance every frame that an ant might pick up a ball when there are 3 nearby, we deemed that unacceptable.

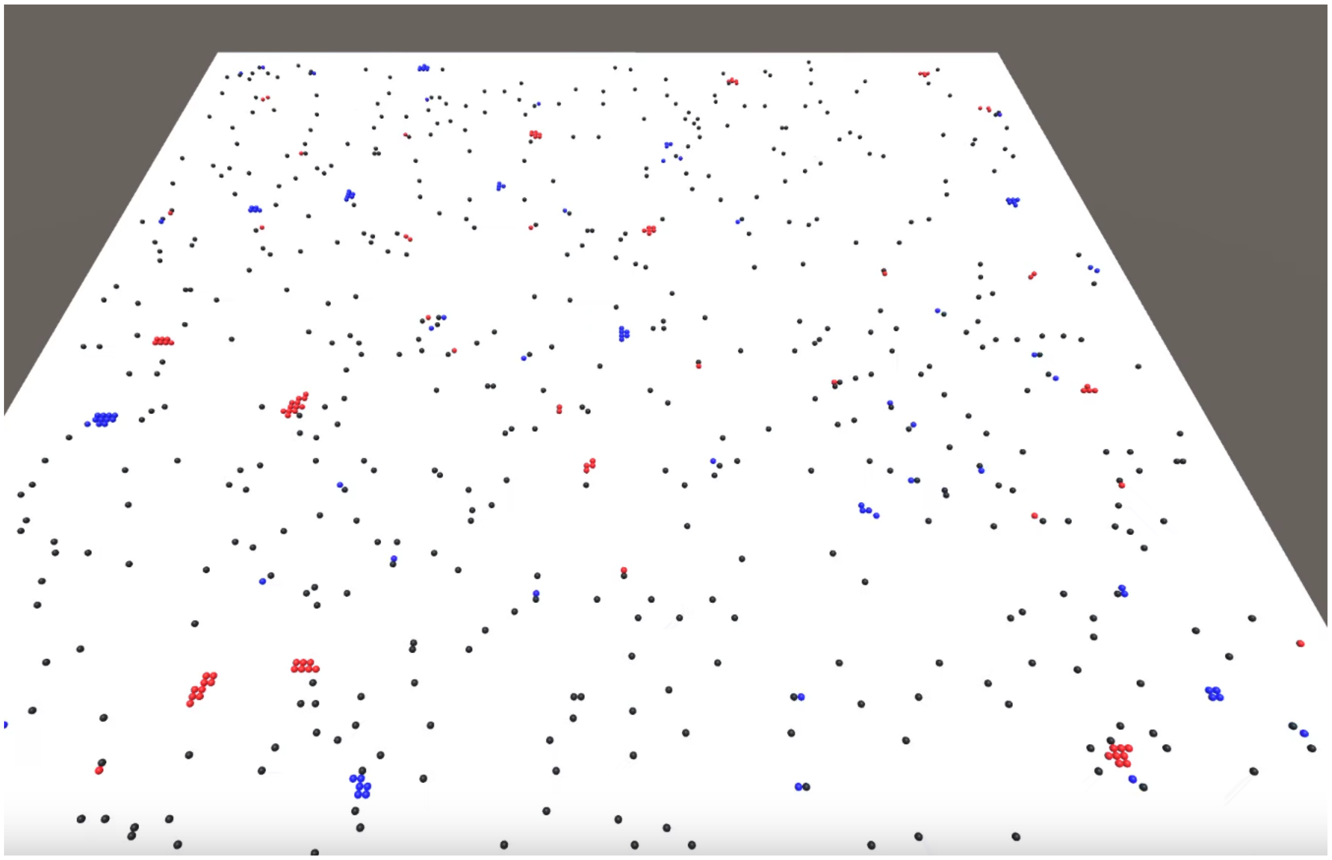


Figure 3: Default Settings: 100 Red, 100 Blue, 500 Ants

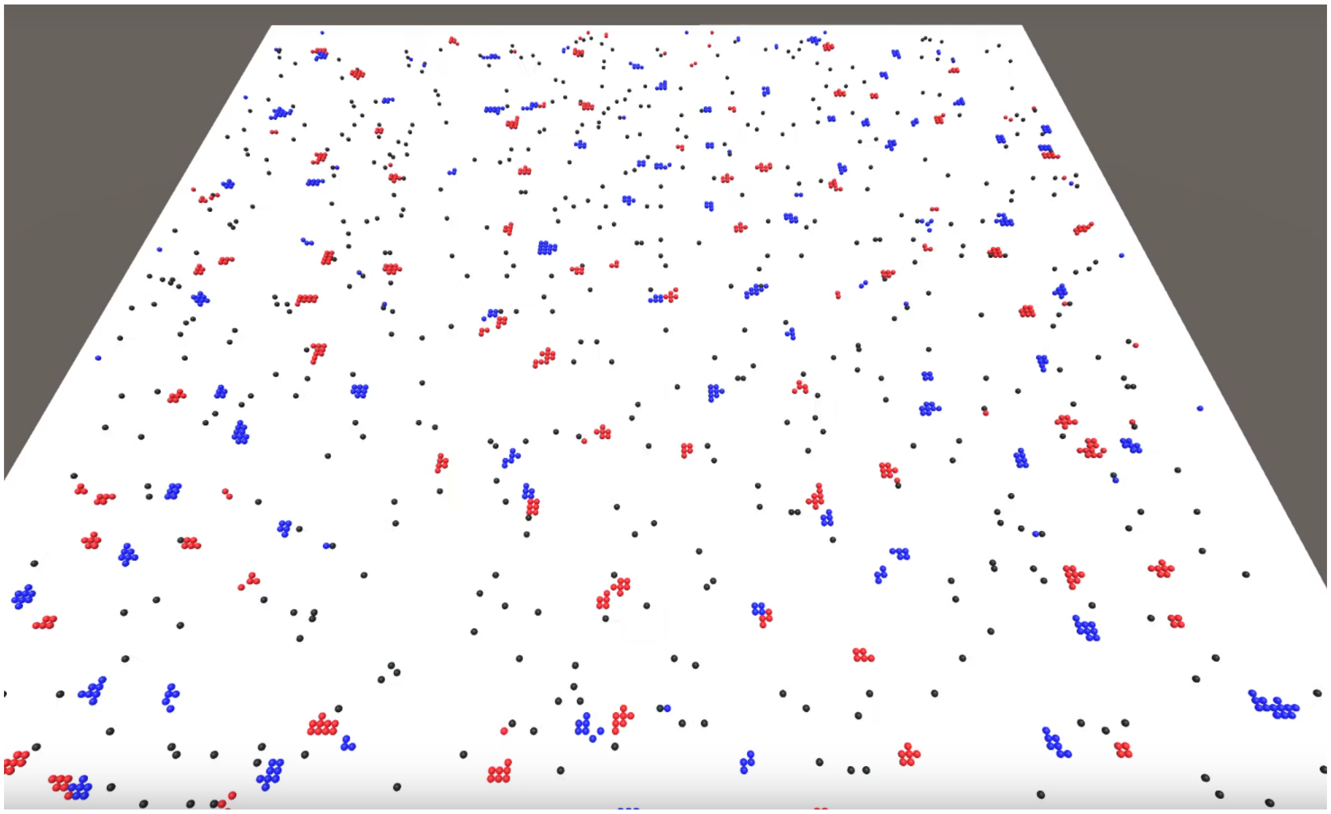


Figure 4: Settings: 500 Red, 500 Blue, 500 Ants

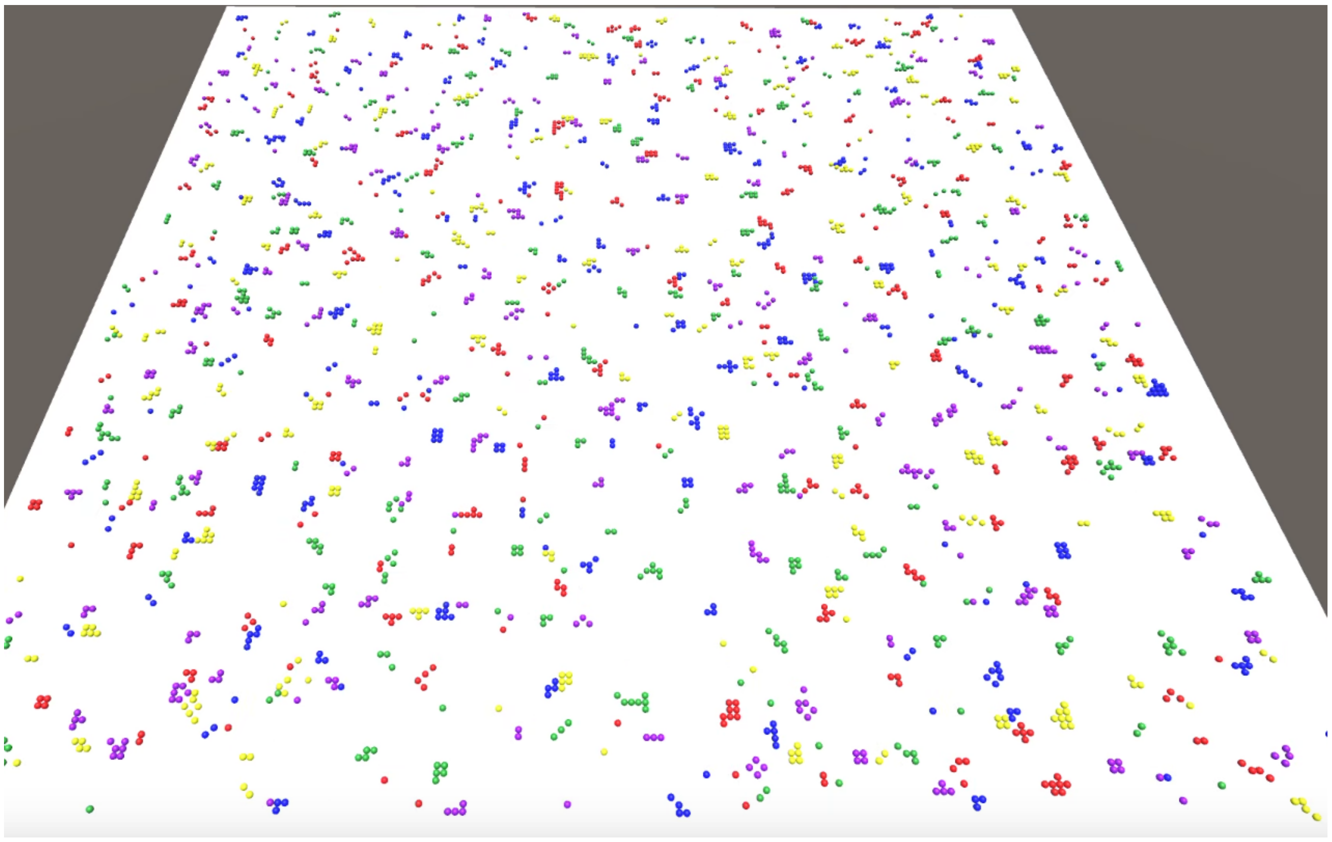


Figure 5: Settings: 500 Red/Blue/Green/Yellow/Purple, 500 Ants

By Figure 5 we realized it would be nice to be able to turn ants on/off so a toggle was made. That is also from Video 6 in the Playlist.

1.4 Game

1.4.1 Windows

Windows Zip Folder

1. Download the zip folder
2. Unzip folder
3. Open unzipped folder
4. Double click on `SwarmClustering.exe`
5. Choose the resolution settings you'd like and enjoy!

See Section 1.2 for controls.

1.4.2 Linux

Linux Zip Folder

1. Download the zip folder
2. Unzip folder
3. From a terminal navigate inside the zipped folder
4. run from terminal: `sudo chmod +x AntSwarm.x86_64`
5. run from terminal: `./AntSwarm.x86_64`
6. Choose the resolution settings you'd like and enjoy!

See Section 1.2 for controls.

1.5 Conclusion

Unity was a fantastic environment to handle this problem, in particular Entity Component System is very well suited for Ant Clustering Algorithms. The game environment allows users to interact with the algorithm (albeit limited at the moment) so they can watch particular aspects of the algorithm. The rules we found seemed to hold quite well for clustering randomly but it would have been nice to see fewer clusters forming. We recognize that is just part of the locality check and could potentially be solved with pheromones or a larger radii.

1.6 Future Work

Future work could include adding selection criteria at the start of the game and an ability to reset to the main menu. We implemented the ability to track larger radii but did not include it in the results because further testing, experimentation, and documentation was necessary. The Dictionary approach could potentially lead to 3D solutions as well. In our work the ants are moving in a 3 dimensional environment but were locked to 2 dimensions.

Problem 2

2.1 Statement

Write the text's Particle Swarm Algorithm, alg 5.4, on:

$$f(x) = 2^{-2((x-0.1)/0.9)^2}(\sin(5\pi x))^6 \quad \text{with} \quad x \in [0, 1].$$

Compare to your simulated annealing result.

2.2 Method

Objective. To maximize the value of *f(x)*

Evaluation function. The evaluation function is *f(x)*

Choice of representation. Each ant is represented as a value $\in [0, 1]$ in 1-D space.

Approach. Particle Swarm Algorithm

A swarm system is composed of a set of individuals capable of interacting with one another and the environment. Particle swarm optimization (PSA) is a population based stochastic optimization inspired by social behaviour of birds and insects.

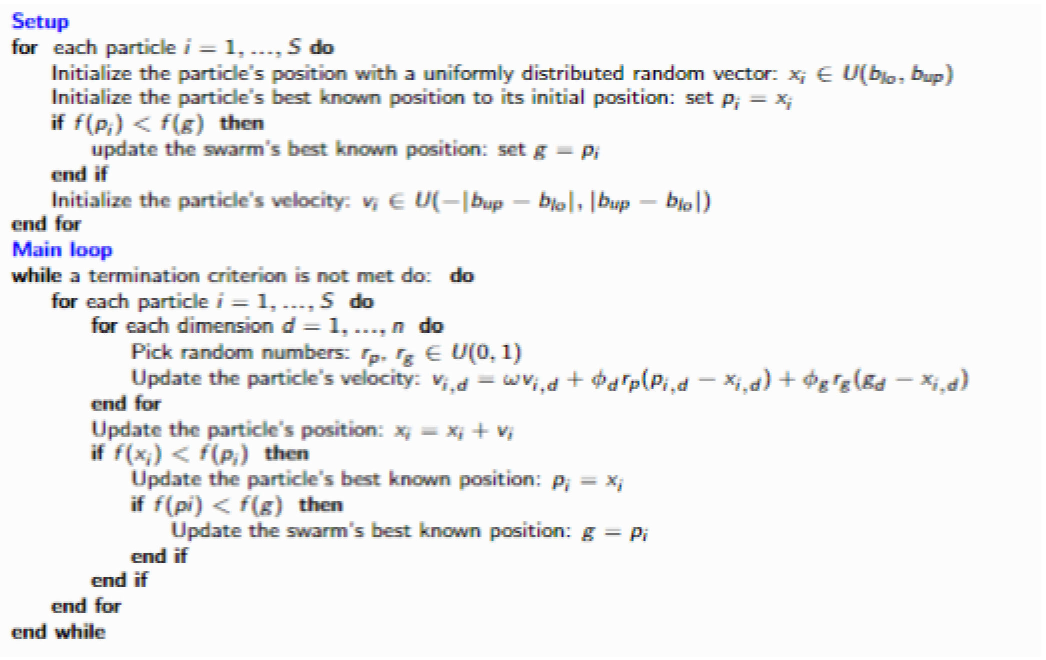
In this algorithm individuals searching for solutions to a given problem learn from own past experience and from the experience of others. Individuals evaluate themselves, compare to their neighbours and imitate only the neighbours who are superior to themselves. Therefore, individuals are able to evaluate, compare and imitate a number of possible situations the environment has to offer.

To find the maxima of *f(x)* PSA will model a social system as follows:

- **Individual.** Each individual is a *particle* in a 1-dimensional space. They store values in [0, 1] and any changes in the value over time is represented as movements.
- **Population.** The swarm of particles corresponds to the multiple individuals in the population.
- **Forgetting and learning.** It is seen as a decrease or increase in the value of a particle in the population. Attitude changes are seen as movements between positive and negative values.
- **Individual's experience.** Each particle has some knowledge of how it performed in the past and uses it to determine where it is going to move to.
- **Social interaction.** Each particle also has some knowledge of how other particles around itself perform and use it to determine where it is going to move to. The particles will tend to move towards one another and tend to influence one another as individuals seek agreement with their neighbors (Kennedy et al., 2001; Kennedy, 2004).

The pseudo code for PSA is as follows:

Algorithm 2 Particle Swarm Algorithm



```
Setup
for each particle $i = 1, \ldots, S$ do
    Initialize the particle's position with a uniformly distributed random vector: $x_i \in U(b_{lo}, b_{up})$
    Initialize the particle's best known position to its initial position: set $p_i = x_i$
    if $f(p_i) < f(g)$ then
        update the swarm's best known position: set $g = p_i$
    end if
    Initialize the particle's velocity: $v_i \in U(-|b_{up} - b_{lo}|, |b_{up} - b_{lo}|)$
end for
Main loop
while a termination criterion is not met do: do
    for each particle $i = 1, \ldots, S$ do
        for each dimension $d = 1, \ldots, n$ do
            Pick random numbers: $r_p, r_g \in U(0, 1)$
            Update the particle's velocity: $v_{i,d} = \omega v_{i,d} + \phi_d r_p (p_{i,d} - x_{i,d}) + \phi_g r_g (g_d - x_{i,d})$
        end for
        Update the particle's position: $x_i = x_i + v_i$
        if $f(x_i) < f(p_i)$ then
            Update the particle's best known position: $p_i = x_i$
            if $f(pi) < f(g)$ then
                Update the swarm's best known position: $g = p_i$
            end if
        end if
    end for
end while
```

The global best or the swarm best will the optimum value. Each particle describes a set of parameter values and a initial velocity (vector) v. And we compute the fitness by just plugging in those values in the cost function *f(x)*. That will give us the fitness of the particle. In each iteration we calculate fitness for each particle. We get the best fitness value of the swarm. Now we simply update the velocities i.e result of initial velocity (v), cognitive force (p_i) social force (p_g) as shown below.

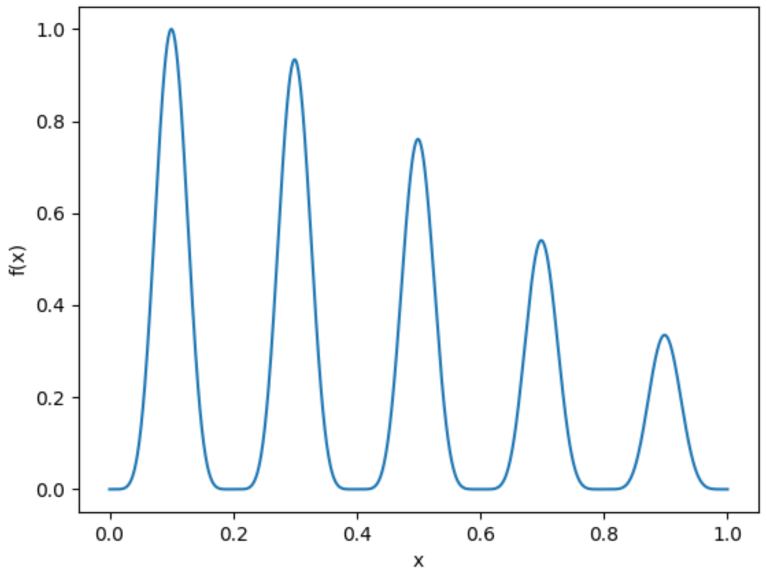
$$v_t(t+1) = v_t(t) + \phi_1 * (p_i - x_i(t)) + \phi_2 * (p_g - x_t(t))$$

where ϕ_1 and ϕ_2 represent positive random vectors composed of numbers drawn from uniform distributions with a predefined upper limit: $\phi_1 = U(0, AC_1)$ and $\phi_2 = U(0, AC_2)$; $U(0, AC)$ is a vector composed of uniformly distributed random numbers and AC is called the *acceleration* constant. The symbol * represents a dot product. For this problem, since the points are in 1-D space, it is simply multiplication.

In order to limit the new position of a particle so that the system does not 'explode', two values x_{min} and x_{max} are defined for the change x, thus guaranteeing that the particle oscillates within some predefined boundaries. We will update the particle to a new position only if the new particle is $\in [0, 1]$

2.3 Results

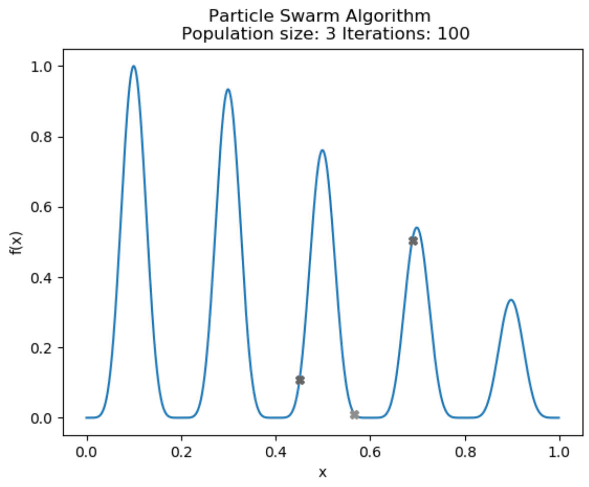
The function *f(x)* looks as follows:

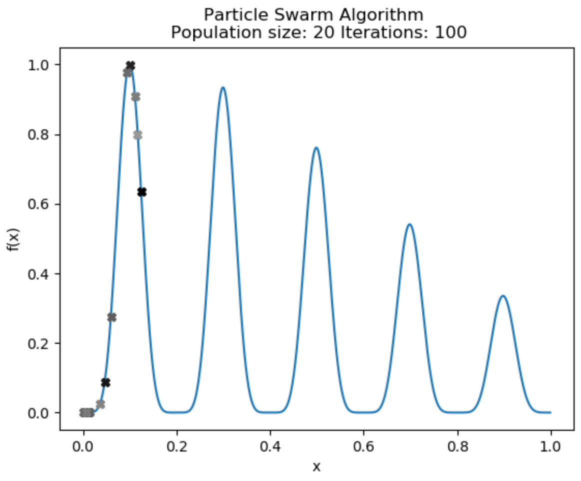


We ran the algorithm for different swarm sizes and 100 iterations:

Table 1: Swarm size vs global and fitness

size	x	*max* $f(x)$
1	0.637	0.0167
2	0.4988	0.7609
3	0.4989	0.7609
4	0.2985	0.9332
5	0.3024	0.9282
6	0.2982	0.9327
7	0.2998	0.9339
8	0.2995	0.934
9	0.1012	0.9989
10	0.0998	1.0
11	0.1002	1.0
12	0.1	1.0
13	0.1001	1.0
14	0.1	1.0
17	0.1001	1.0
18	0.0999	1.0
19	0.1	1.0
50	0.1	1.0

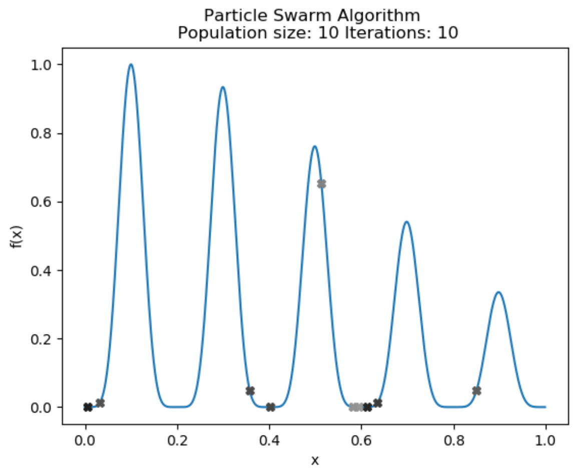


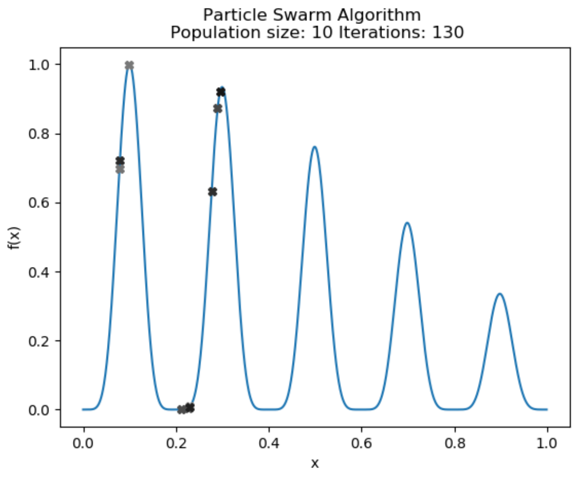


We also ran the algorithm for different iterations (generations) and swarm size of 10:

Table 2: Iterations vs global and fitness

iterations	*x*	*max f(x)*
10	0.8865	0.3028
20	0.3008	0.9329
30	0.3011	0.9322
40	0.3153	0.7752
50	0.1003	0.9999
60	0.1005	0.9998
70	0.1	1.0
80	0.0997	0.9999
90	0.0998	1.0
100	0.1001	1.0
110	0.1	1.0
120	0.1006	0.9998
130	0.0999	1.0





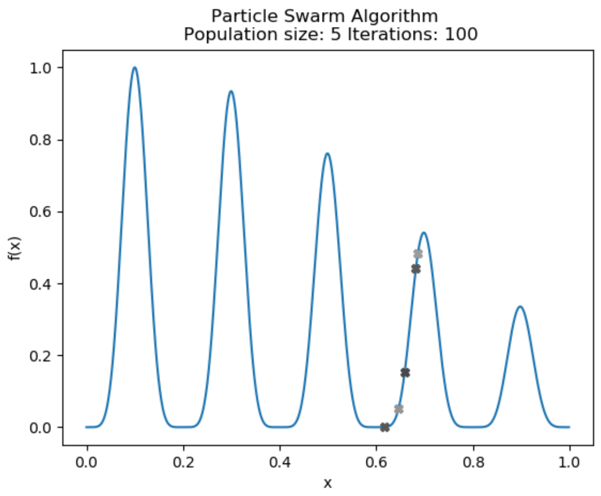
2.4 Conclusion

Particle Swarm Algorithm is a great way to find numeric parameters that optimizes a function. Some interesting observations that we made were as follows:

- The number of particles in a swarm affected optimization when the maximum iteration was fixed. When the swarm size was small, here less than 5, the chances of getting stuck in a local maxima was high. However, for higher swarm size, it was low if the particles were uniformly distributed. This is shown in Table 1.

- The maximum number of iterations also affected optimization for a fixed swarm size. When the number of iterations were low, the result would be a local maxima. When the number of iterations were high, the chances of it being stuck in a local maxima was low. This is shown in Table 2.

- The uniform distribution of the particles affected optimization more than the number of iterations. If the particles were concentrated more towards the x = 1, the particles will move towards x= 1, i.e., away for the optimum x = 0.1, no matter what the number of iterations were. This is shown below.



Simulated Annealing. Particle swarm algorithm was faster at finding the maxima compared to simulated annealing. However, if the particles were not uniformly distributed, even for high population and iteration, PSA would lead to a local maxima. Simulated annealing never leads to a local maxima in those conditions.

2.5 Code

```python
import random
import numpy as np
import matplotlib.pyplot as plt


class SwarmIntelligence:
    __POP = 10
    __MAX_ITER = 200
    __swarm = []
    __fitness = []
    __velocity = []
    __best_ind = []
    __best_fit = []
    __X_LOW = 0
    __X_HIGH = 1
    __global = 0

    def __init__(self):
        best_fitness = 0
        for i in range(self.__POP):
            individual = random.random()
            fitness = self.__f(individual)

            self.__swarm.append(individual)
            self.__fitness.append(fitness)
            self.__velocity.append(0)
            self.__best_ind.append(individual)
            self.__best_fit.append(fitness)

            if best_fitness < fitness:
                best_fitness = fitness
                self.__global = self.__swarm[i]

    # Evaluation function
    def __f(self, x):
        return np.power(2, -2*(((x-0.1)/0.9)**2)) * np.sin(5*np.pi*x)**6

    # prints current population data
    def print_swarm(self):
        for i in range(self.__POP):
            print(self.__swarm[i], self.__fitness[i])
        print("Best: ", self.__global)

    # runs the algorithm
    def PSA(self):
        top = self.__f(self.__global)
        for iteration in range(self.__MAX_ITER):
            for i in range(self.__POP):
                phi1 = 0.001*random.random()
                phi2 = 0.001*random.random()

                self.__velocity[i] += phi1 * (self.__best_ind[i] - \
                     self.__swarm[i]) + phi2 * (self.__global - \
                     self.__swarm[i])

                if (self.__swarm[i] + self.__velocity[i] > self.__X_LOW) \
                 and (self.__swarm[i] + self.__velocity[i] < self.__X_HIGH):
                    self.__swarm[i] += self.__velocity[i]
                    self.__fitness[i] = self.__f(self.__swarm[i])

            if iteration % 10 == 0: self.__plot()

            for i in range(self.__POP):
                if self.__fitness[i] > self.__best_fit[i]:
                    self.__best_ind[i] = self.__swarm[i]
                    self.__best_fit[i] = self.__fitness[i]
                if self.__fitness[i] > top:
                    self.__global = self.__swarm[i]
                    top = self.__fitness[i]
        return self.__global, top

    # Plotting results
    def __plot(self):
        plt.clf()
        color = list(np.random.choice(range(100, 256), size=self.__POP))
        x = np.arange(self.__X_LOW, self.__X_HIGH, 0.001)
        y = [self.__f(i) for i in x]
        plt.plot(x, y)
        plt.title("Particle Swarm Algorithm \n Population size:{} \
        Iterations: {}".format(self.__POP, self.__MAX_ITER))
        plt.xlabel('x')
        plt.ylabel('f(x)')
        for i in range(self.__POP):
            plt.plot(self.__swarm[i], self.__fitness[i], \
            color = str(color[i]), marker='X')
        plt.show()
        plt.pause(0.001)


plt.ion()
best, val = SwarmIntelligence().PSA()
print("Best individual: ", round(best, 4), "  Fitness: ", round(val, 4))
```

```
plt.ioff()
plt.show()
```