Christopher Moran

Dr. MacLennan

CS 420

24 April 2020

Project 5: Particle Swarm Optimization (PSO) Algorithm

The purpose of this project was to familiarize the student with the concepts related to particle swarms. The general concept behind this project, as in other projects, can be traced back to its namesake and use of separate particles. The model attempts to simulate the velocities of different particle groupings, which allows the model to represent a number of physical phenomena and interactions present in the physical world. As an example, this program, with a few modifications and a changing inertia, would be able to simulate how a group of particles interact as the temperature decreases.

The model functions through the use of different maxima and societal values, which allow the particles to behave in a sort of group like fashion. Perhaps group like is not the correct term, but the fact of the matter remains that they share a certain set of influences, which in turn forces the separate particles to move in a similar fashion and gradient, on average. This project is entirely self-contained, which means that it does not rely on an external data set to form its model, which is useful because it allows for increased portability when compared to the alternative.

Methodology

The program is structured in more or less the same way as the instructions were laid out. This means that the first thing required of the program was the initialization. The majority of the initialization was contained within a function, though some things, such as the 5 required

arguments, had to be set before the initialization function. These 5 arguments consist of the number of epochs, the number of particles, the inertia, the cognition parameter, and the social parameter. There were also some additional parameters that had to be set before the function, such as the maximum velocity of a particle, the world width, and the world height. Once those values were initialized, it was possible to move on to the initialization function. This is where all of the different arrays were created, as the assigned a number of values for each particle. In a way, this is where the particles were created. Also, before entering this function, the mdist must be found via this function:

$$mdist = \sqrt{\max_x^2 + \max_y^2} / 2$$

Where max_x is the width, and max_y is the height.

The function starts off by setting the each particle's position to a random spot within the confines of the world. Then, it moves on to initialize each particle's velocity, though they all start with the value of 0. It also initializes each particle's personal best position to that of its current position, as there is no other option. Once these essential particle values have been set, it becomes possible to set that particle's pdist and ndist. Pdist is given by this function:

$$pdist = \sqrt{(p_x - 20)^2 + (p_y - 7)^2}$$

And ndist with this function:

$$ndist = \sqrt{(p_x + 20)^2 + (p_y + 7)^2}$$

In these functions, px represents the particles x axis position, and py represents its y axis position.

These values are required to create the performance function, which changes depending on which problem is currently being examined. As pdist and ndist are recalculated for each epoch/update/iteration of particles, they can be used to calculate the performance of each individual particle. A particle's performance is to be given by the function Q(px, py), though what Q does changes depending on which problem is being solved. If problem 1 is being solved, then the function Q can be represented by the calculation shown here:

$$Q(p_x, p_y) = 100 \cdot \left(1 - \frac{pdist}{mdist}\right)$$

And the problem 2 implementation of Q can be represented as follows:

$$Q(p_x, p_y) = 9 \cdot \max(0, 10 - pdist^2) + 10 \cdot \left(1 - \frac{pdist}{mdist}\right) + 70 \cdot \left(1 - \frac{ndist}{mdist}\right)$$

These equations are important when updating the personal bests of each particle (and therefore the global bests), which then influence how each particle's velocity changes from iteration to

iteration, which cascades into changing how their positions are updated, which circles back to influence how the pdist and ndist values are calculated. This makes it a self feeding model of sorts.

This of course brings up the matter of how, specifically, the model is able to update itself. In a similar fashion to how the model was initialized, the updating occurs on a particle by particle basis. As soon as each particle is updated, then the model goes back and updates them again, assuming the error, which is calculated after each particle is updated, is not beyond the threshold where the model must then break out of its loop. This is actually just one of a number of ways in which the model can break out of its loop. Another way would be if the model iterated up until the predetermined max number of epochs, which was initialized at the very beginning. That being said, the main idea behind the model is to influence the particles to converge in some way, and to be able to measure this in some way. Once the particles have converged to a point deemed acceptable, the model will break out of its loop and become complete to according to its initial parameters.

All that being said, this still leaves the question of how the particles are updated within each epoch. First, the velocities are updated, first for the x axis velocity, and second for the y axis velocity (as the velocities, as well as most map dependent variables, are actually 2d vectors). This is done to the specifications of the following equation.

$$velocity' = inertia * velocity + c\_1 * r\_1 * (personal\_best\_position - position) + c\_2 * r\_2 * (global\_best\_position - position)$$

In this equation, c_1 is the cognition parameter, c_2 is the social parameter, and both r_1 and r_2 are random numbers between 0 and 1.

Then, the model must ascertain that the velocities have not exceeded the maximum velocity, which can be done by using the following equation:

```
if velocity_x^2 + velocity_y^2 > maximum_velocity^2
  velocity = (maximum_velocity/sqrt(velocity_x^2 + velocity_y^2)) * velocity
```

Where velocity is either the x or y for a specific particle, though it should be said that it must be done for both x and y before moving on.

Then, once the velocity has been confirmed, the model is able to determine what the new positions of the particles will be. This can be done with a very simple formula as shown below.

```
position' = position + velocity'
```

Like before, this must be done for both the x and y position before moving on.

Then, the model is able to calculate each particle's pdist and ndist values using the equations listed further above. Once those values have been found, they can be used to discern whether or not the change in a particle's position entails a new personal or global best, which can be determined with the following two equations:

Personal best

```
if(Q(position) > Q(personal_best_position))
  personal_best_position = position
```

Global best

```
if(Q(position) > Q(global_best_position))
  global_best_position = position
```

As far as the core of the model is concerned, all that is required has been fulfilled at this point. That being said, it is important to apply bounds to any model, which is why error analysis is necessary. In this particular model, this is done by calculating the specific error for the y and x axes after each iteration.

This can be done in two steps, with only the first one requiring the model to iterate through each of the particles, as it piles on a sum into an error value, as shown with these equations:

$$error\_x \mathrel{+}= (position\_x[k] - global\_best\_position\_x)^2$$
$$error\_y \mathrel{+}= (position\_y[k] - global\_best\_position\_y)^2$$

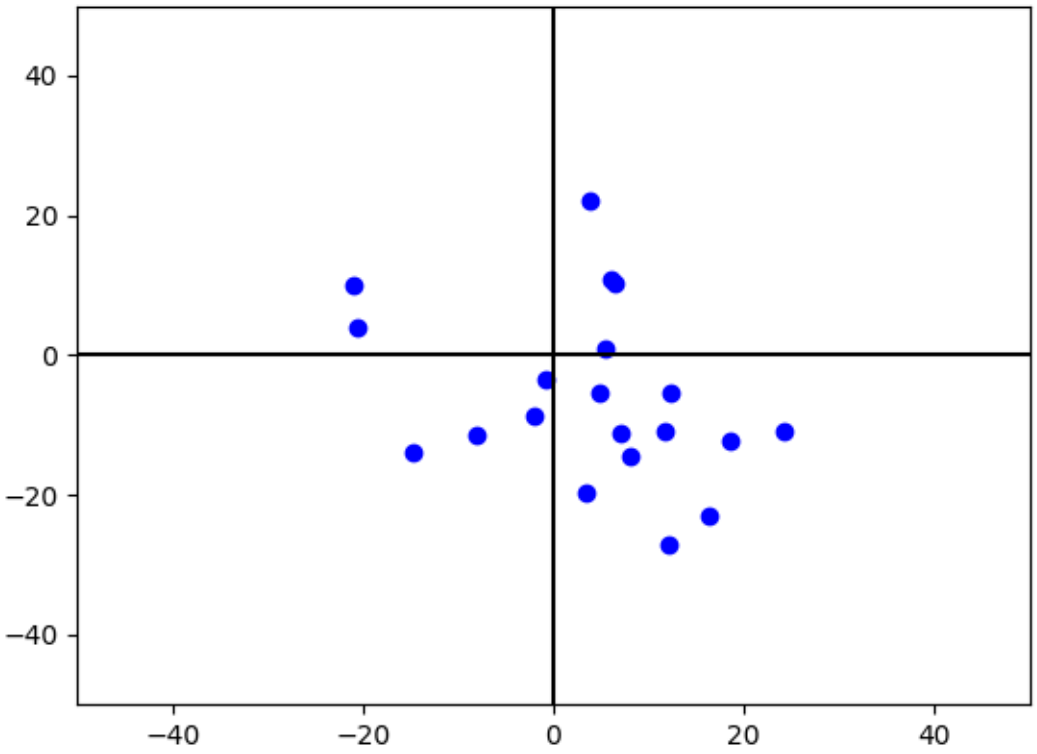
Once each particle has been iterated through, it is possible to use this equation to find the final iterative error values, using these equations:

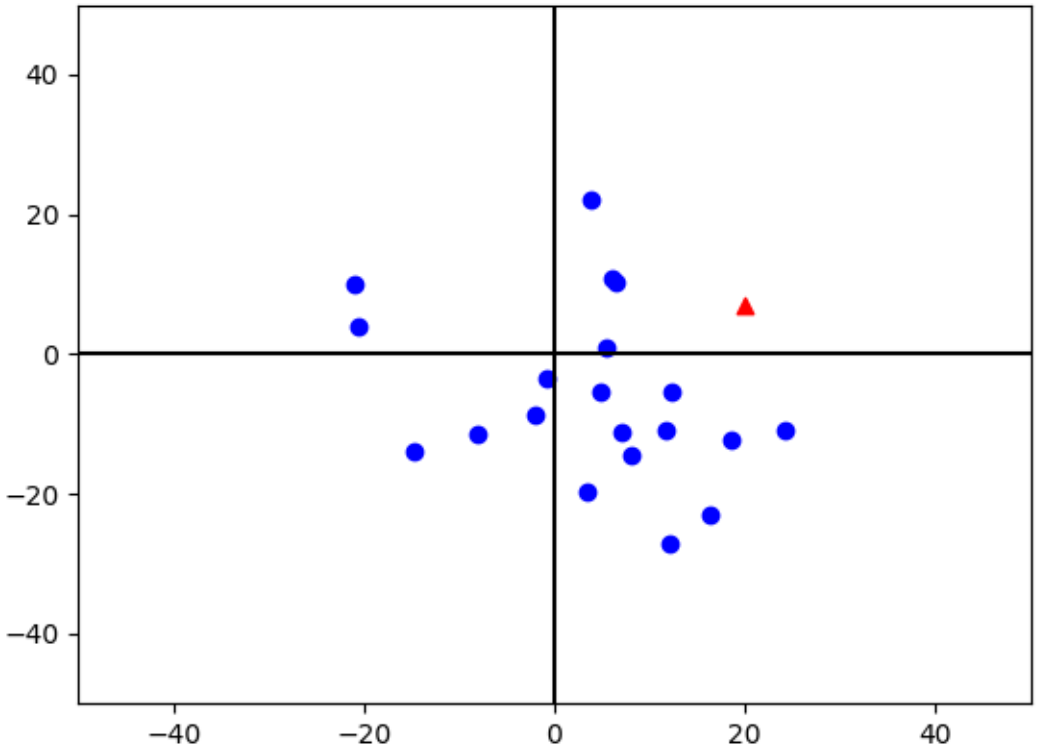
$$error\_x = \sqrt{(1/(2*num\_particles))*error\_x}$$
$$error\_y = \sqrt{(1/(2*num\_particles))*error\_y}$$

Then, all errors found are able to be plotted over the number of iterations, which should represent the performance over the course of the program, though it is technically possible to use the error value in any number of ways. As an example, it can be used as the stopping condition, or as a way of showing convergence.

Analysis

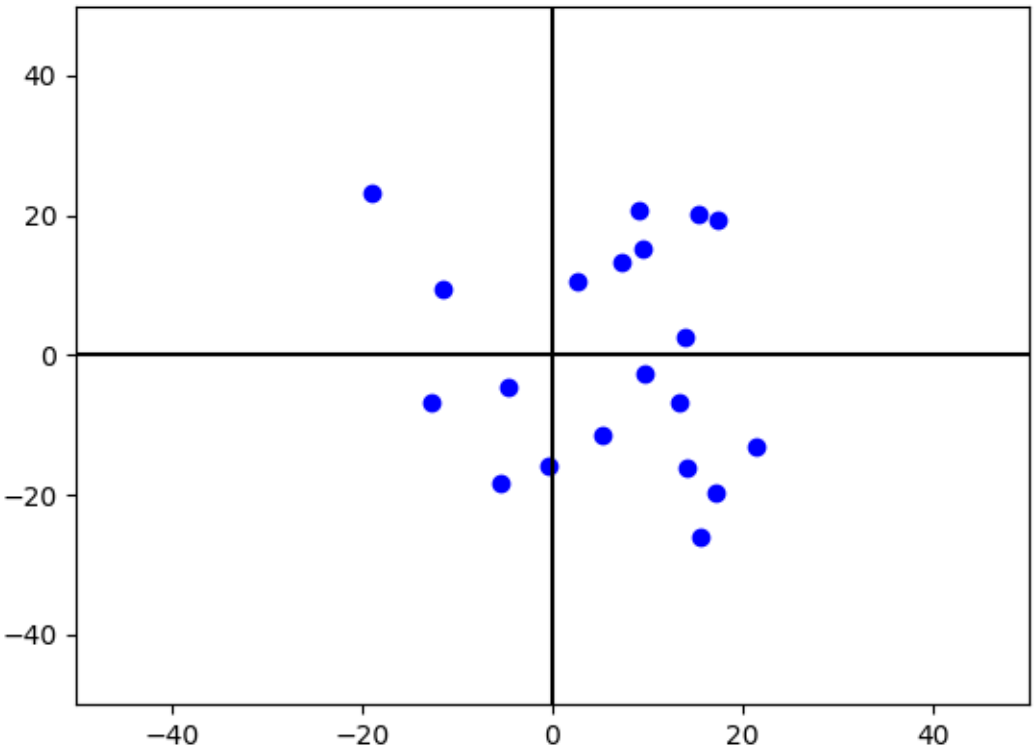
Problem 1

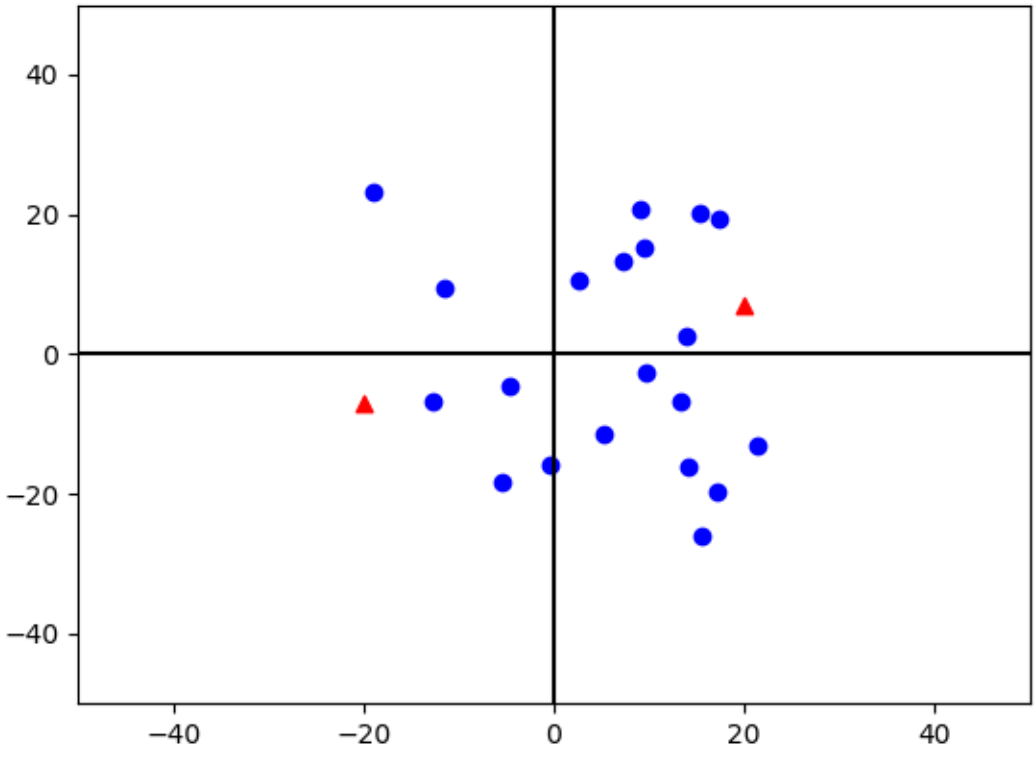




The above page shows the results of problem 1 when c_1 and c_2 are set to 2, inertia to 1, the number of epochs to 10000, and max velocity set to 1.

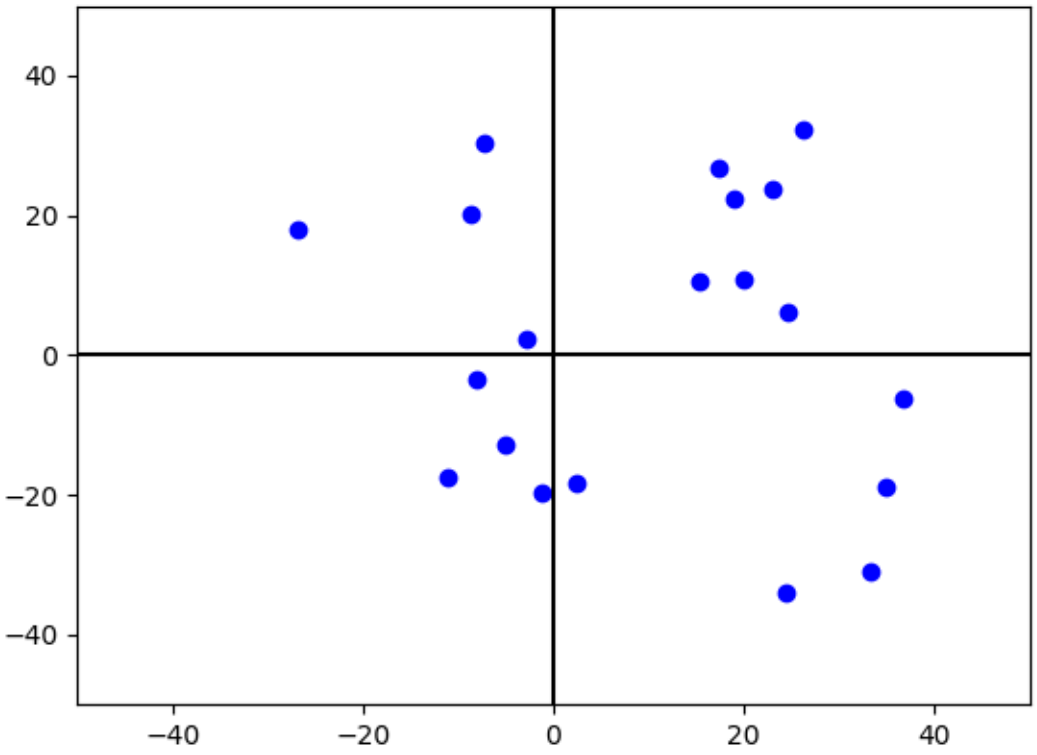
Problem 2

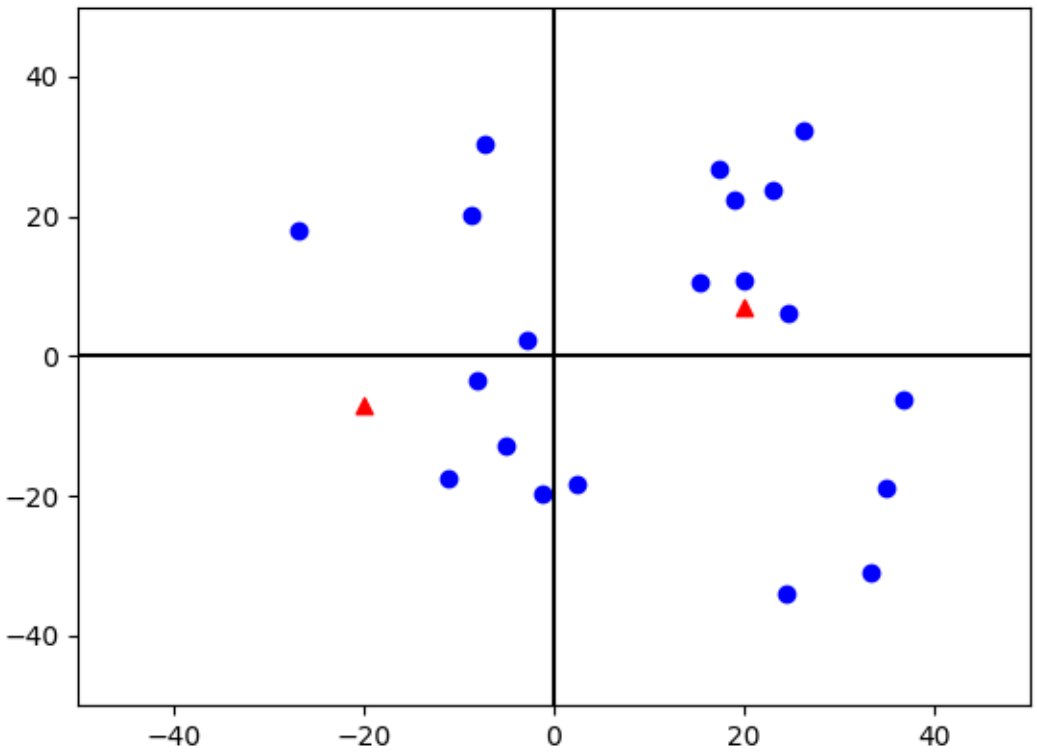




The above page shows the results of problem 1 when c_1 and c_2 are set to 2, inertia to 1, the number of epochs to 10000, and max velocity set to 1.

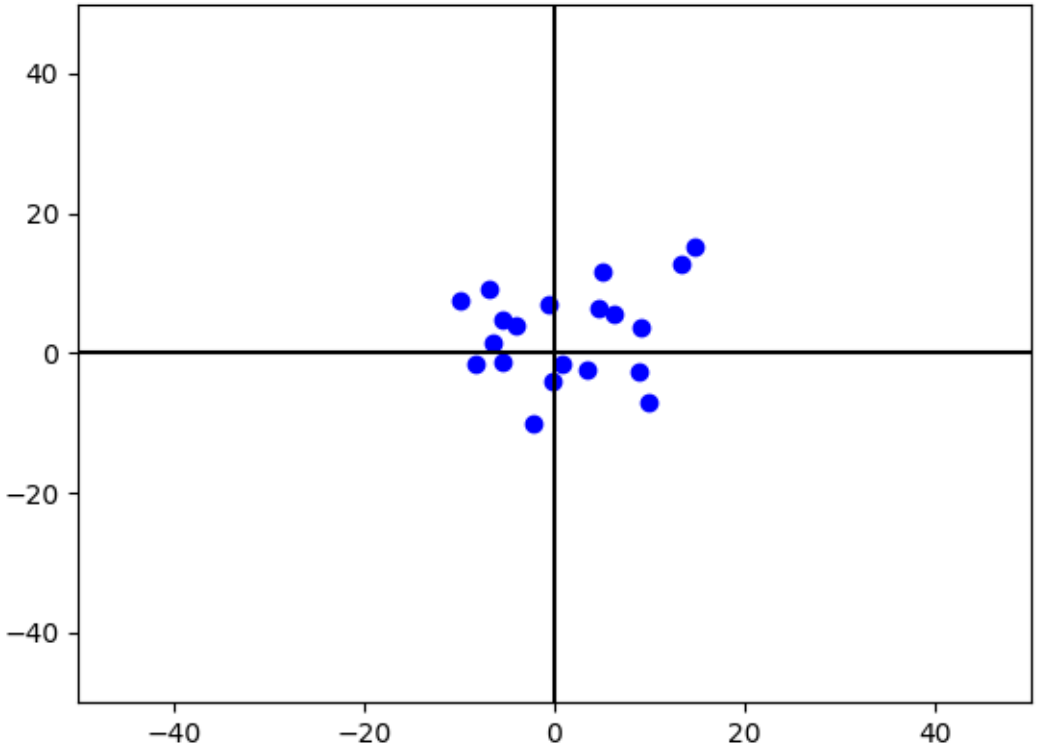
Large c1 Small c2

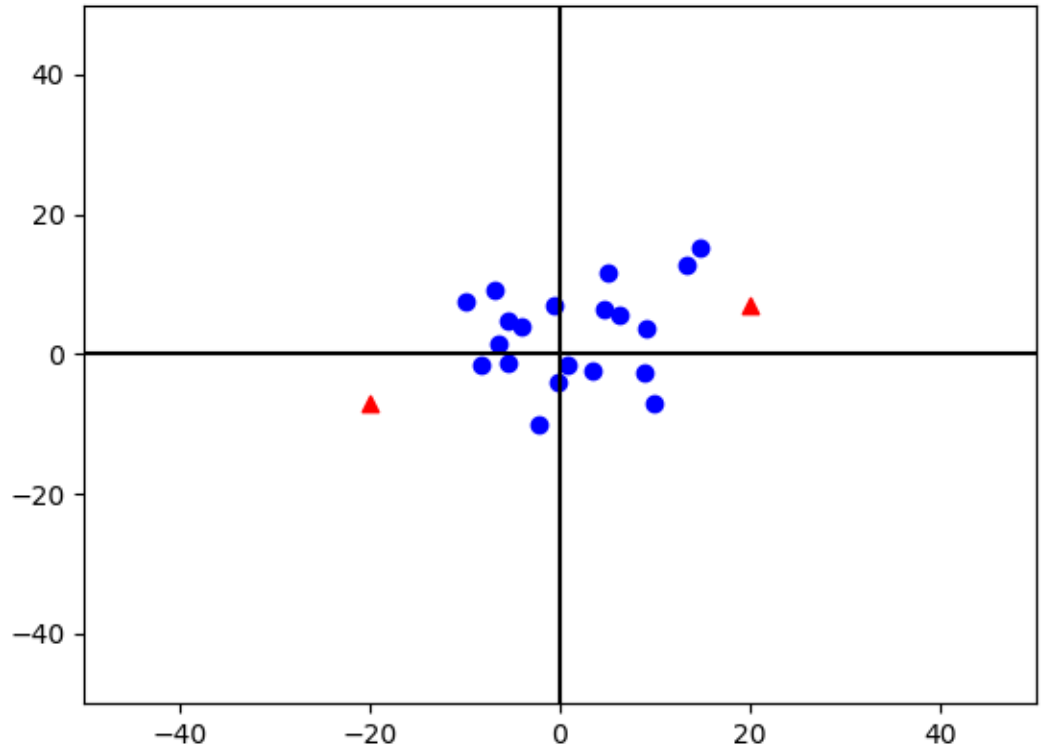




The above page shows the results of problem 1 when c_1 is 3 and c_2 is set to 1, inertia to 1, the number of epochs to 10000, and max velocity set to 1.

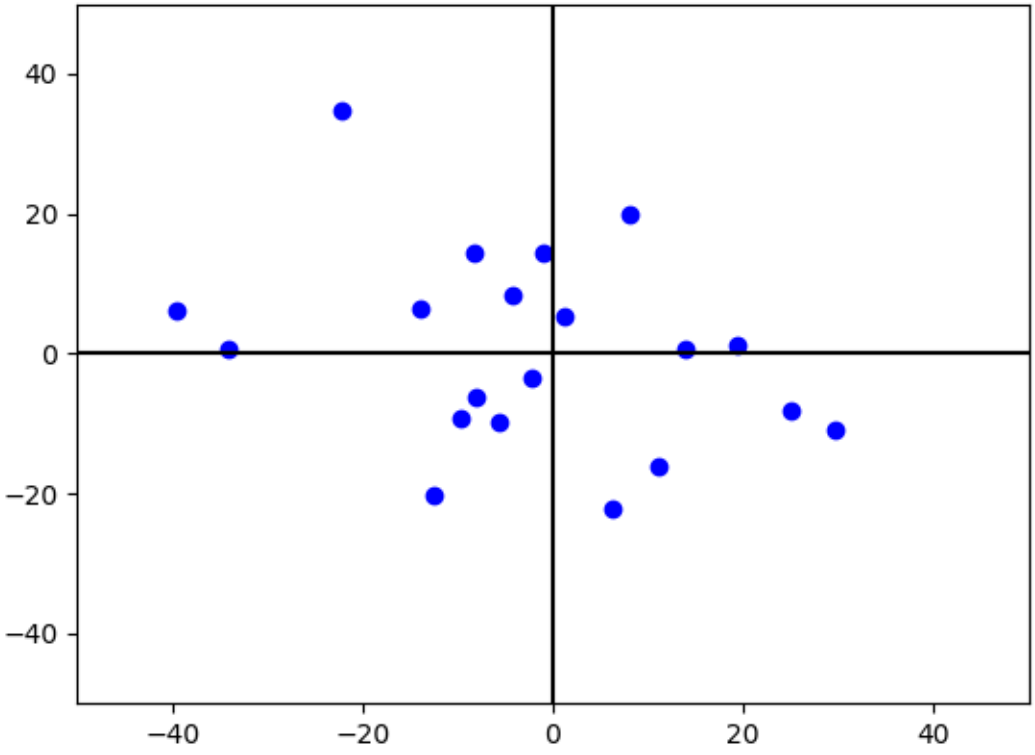
Small c1 Large c2

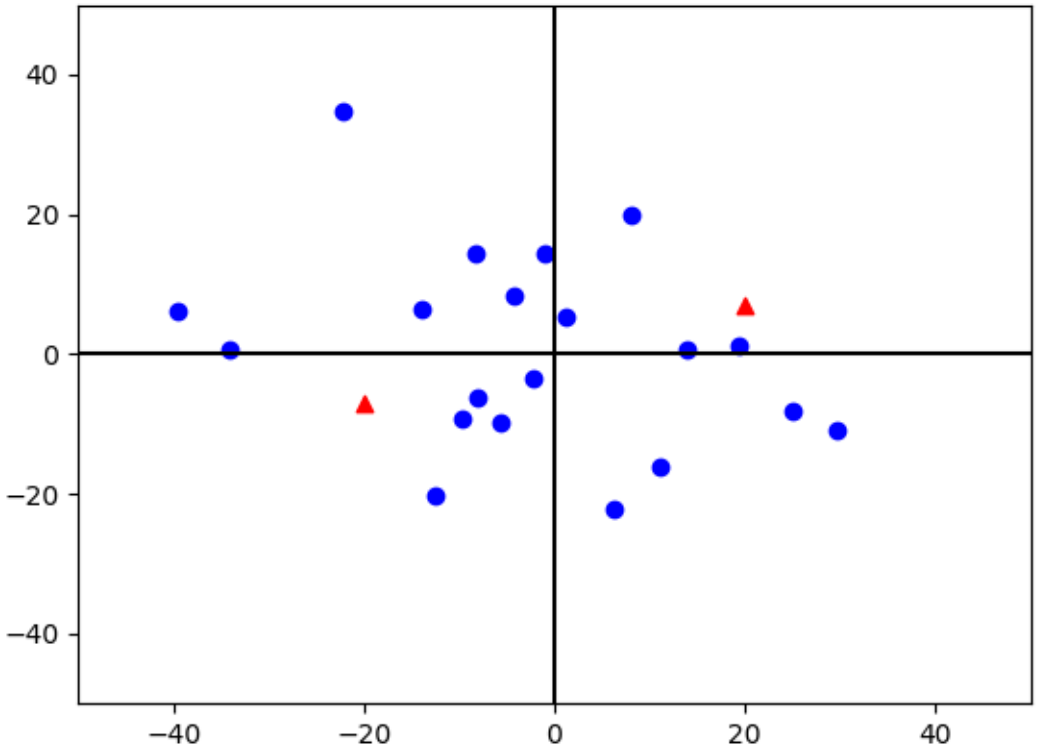




The above page shows the results of problem 1 when c_1 is 1 and c_2 is set to 3, inertia to 1, the number of epochs to 10000, and max velocity set to 1.

Max Velocity 10





The above page shows the results of problem 1 when c_1 and c_2 are set to 2, inertia to 1, the number of epochs to 10000, and max velocity set to 10.

Conclusion

The project was a success, though there due to the fact that complete convergence could never be seen, it is uncertain whether or not it behaves perfectly. That being said, the results seem reasonable enough, especially when changing the parameters. When emphasizing the cognition parameter, c1, more spread can be observed. Conversely, when the social parameter is emphasized, much less spread is observed. It is also interesting to note that even though the max velocity was increased tenfold in the last experiment, it did not increase spread tenfold when compared to the problem 2 experiment.