COSC 420: BIOLOGICALLY-INSPIRED COMPUTATION

PROJECT 5 - "PARTICLE SWARM OPTIMIZATION"

CLARA NGUYEN

TABLE OF CONTENTS

1. SYNOPSIS

In this project, I explored the Particle Swarm Optimization algorithm, and how can be applied. Generally, this algorithm involves placing particles in a "search space" and having them search randomly for a place that gives them the highest fitness. In our case, this is a 3D equation's maximum value.

2. DEVELOPMENT

The simulator was written **three times** in **three** different languages:

- **C++** - Initial Implementation. Console only. Prints out error values to console.
- **GML** – GUI via Direct3D. 2D Grid for showing visually how particles move around.
- **HTML/CSS/JavaScript (WebGL)** – GUI via a web port of OpenGL that runs in your web browser. This is the implementation that was used to run experiments for this paper.

All three implementations will be in the "src" directory under their respective language. The GUI ones (GML & JS [WebGL]) will allow you to configure the simulation from the application. The C++ version takes the following parameters:

```
./pso iterations size particles inertia cognition social max_velocity
```

In the GUIs, the number of iterations is infinite, though we can only consider a certain number of iterations for our observations later on. In the C++ implementation, a default setup (as par with the project writeup) will look like this.

```
./pso 50 100 40 0.99 2.0 2.0 1.0
```

It will output text in the format of a CSV file, which can be piped to a CSV file afterwards. All three simulators will have some way to write a CSV file, but the C++ one is the most straight forward.

3. HOW IT WORKS

3.1. GENERAL ALGORITHM

```
configure simulator properties and set equation to use
create particles in random spots
update()
  for every particle
    compute velocity and normalize
    update position, personal best, and global best
    update error
average error
write data file
```

Figure 1. *Pseudo Code for the Simulator.*

For all three simulators, the program structure was written with pseudo code from Figure 1 in mind. The program abides by equations was given to us in a project writeup and is shown in Figure 2.

$$Q1(p_x, p_y) = 100 * \left(1 - \frac{pdist}{mdist}\right)$$

$$Q2(p_x, p_y) = 9 * \max(0, 10 - pdist^2) + 10 * \left(1 - \frac{pdist}{mdist}\right) + 70 * \left(1 - \frac{ndist}{mdist}\right)$$

Figure 2 (Top) and 3 (Bottom). *Equations Q1 and Q2 given in the Project Writeup.*

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

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

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

Figure 4. *Variables required for equations in figures 2 and 3.*

3.2. ALGORITHM SYNOPSIS AND BREAKDOWN (UNDERGRADUATE PORTION)

The equations Q1 and Q2 are both three dimensional functions that request 2 variables, being p_x and p_y explicitly (assuming max_x and max_y are predefined). As such, these equations can be graphed via applications such as Wolfram, Matlab, etc. In the cases of a visual graph, the maximum can easily be found by simply looking. However, the computer doesn't think in the same way that we do. The Particle Swarm Optimization algorithm aims to find that maximum through particles that simply seek out the position that gives each particle the best "fitness".

3.2.1. PARTICLE POSITION MOVEMENT

The equations for *pdist* and *ndist,* given in Figure 4, are two-dimensional distance formulas. Meanwhile, *mdist* is a modification of the distance formula. The particles will go in random directions based on a random number generator, but the "randomness" is biased toward higher fitness. The next position of each particle is calculated by the following equation:

$$velocity' = inertia * velocity + cognition * rand(0,1) * \left(best_{personal} - position\right) + social * rand(0,1) * (best_{global} - position)$$

Figure 5. *Velocity Computation Equation (Applies to all axes, including X and Y)*

In each update step, the velocity is computed with the formula in Figure 5. It is then normalized to conform to the maximum velocity specified in the simulation parameters. After normalization, the velocity is simply added to the particle's position on all axes. The particle's personal best and the simulator's global best are then updated, along with the percentage of error. In our case, the percentage of error will determine whether the system has converged.

3.2.2. PERSONAL AND GLOBAL BEST

The way how our "biased" random search works is by updating the $best_{personal}$ and $best_{global}$ variables. Though, the decision to go to one of those variables is determined by the following parameters:

- **Cognition** - Influence to seek $best_{personal}$
- **Social** - Influence to seek $best_{global}$

The two randomly chosen numbers between 0 and 1 will determine the influence that these variables have on the simulation. The variables for the best positions can be updated mid-simulation by the following pseudocode:

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

Figure 6. *Pseudo Code for updating Personal Best*

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

Figure 7. *Pseudo Code for updating Global Best*

If we look back at the formula in Figure 5, it is observed that the particle will attempt to seek either the $best_{personal}$ or the $best_{global}$. It will take the difference between those variables, and the current position, and multiply by cognition or social variables appropriately.

3.2.3. COMPUTING ERROR

Error is surprisingly simple in this simulation. In our update step, set the error for all axes to 0. Then, for each particle, append the following pseudocode after updating the personal and global best:

```
error_x += (particle.x – global_best_x)^2
error_y += (particle.y – global_best_y)^2
```

Figure 8. *Pseudo Code for updating Error*

After the particles are done being updated, the error is normalized by applying the following equation:

$$error = \sqrt{\frac{1}{2 * particle\_count}}$$

Figure 9. *Formula for normalizing the error on any axis.*

The equation in Figure 9 is applied to both the X axis and Y axis.

3.2.4. DETERMINING CONVERGENCE OF A SYSTEM

The error value can be used to help determine if a system has converged. If it is under an extremely low value, all of the particles are assumed to be extremely close to each other, and thus near the global maximum of the function given. When this happens, we can conclude that the system of particles has “converged”. Unfortunately, this assumption is not a guaranteed one. There exist cases where a particle system will never converge under these circumstances. To counter this, a “Max Iterations” variable was added to the simulator to cut it off after the epoch goes over this specified value.

4. EXPERIMENT 1: EPOCH VS. INERTIA

4.1. SYNOPSIS

Epoch is simply the number of times that the simulator has had to go through each particle and update their position, velocity, and personal best values. It can also represent the number of times that the error was updated. Point being, **epoch is a measurement of time.** And time is flowing at a constant rate throughout the simulation. This makes epoch an extremely valuable variable in graphing information. For this experiment, we will be comparing epoch to 12 different runs of the simulator, each with a different value of inertia specified.

4.2. SIMULATOR PARAMETERS

The following parameters were used in the WebGL implementation of the Particle Swarm Optimization simulator:

- **Equation:** Q1
- **Grid Size:** 100
- **Particles:** 50
- **Inertia:** {0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 0.95, 0.99, 0.999, 1.0}
- **Cognition:** 0.1
- **Social:** 1.0
- **Max Velocity:** 1.0
- **Max Iterations:** 500

The only variable that changes in this experiment is, obviously, the inertia variable. The equation in Figure 2 (Q1) will be used as well, as it is the simpler of the two to make an observation with. The equation in Figure 3 will be observed in future experiments.

4.3. SIMULATION

All of 11 inertia properties did not change the overall pattern of how error decreased per update step. They all declined, as predicted, and approached 0. The WebGL simulator exports CSV data relating to Error X, Error Y, Global Best X, and Global Best Y. To plot this, we need to get the error down to a single value, as opposed to two values. To do this, we simply use the distance formula:

$$error = \sqrt{error_x{}^2 + error_y{}^2}$$

Figure 10. *Error Computation when given 2 dimensions.*

After plotting the data in Excel, we can generate a graph of all of the error values for every single run of the experiment.

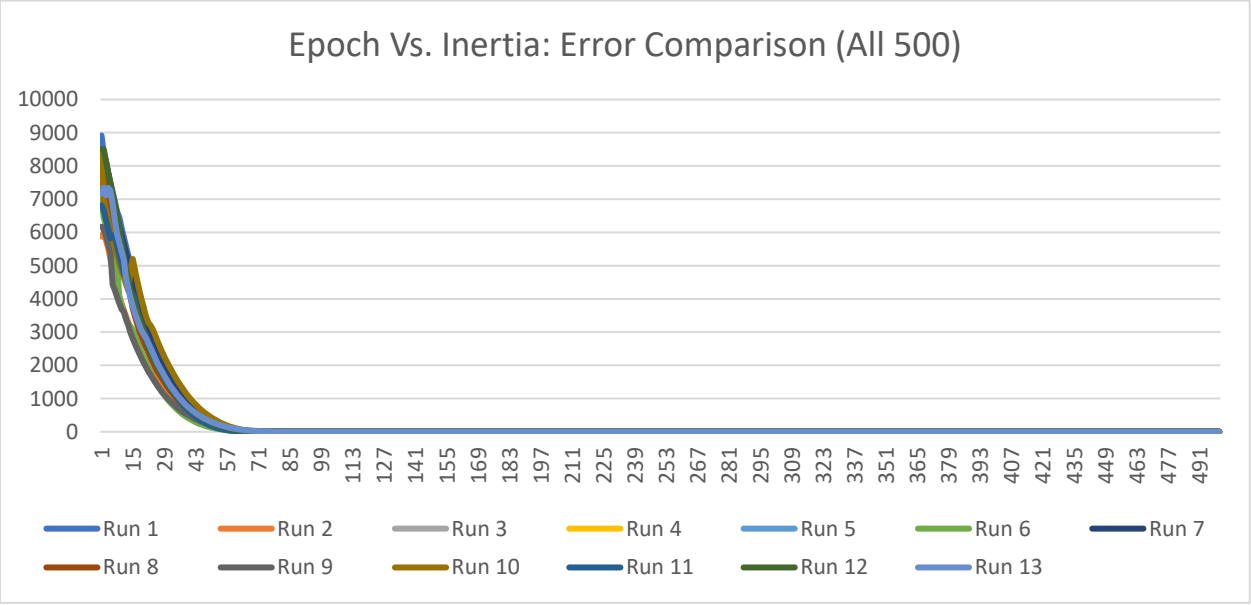

Figure 11. *All 500 error computations for runs 1 through 13.*

As previously mentioned, the pattern is consistent among runs 1 through 13. They approach 0.

However, this graph can give more detailed information if the range is more carefully chosen. Because we know it'll eventually approach a value near 0, we can set the vertical axis to a much more reasonable value… so let's set it to go from 0 to 6.

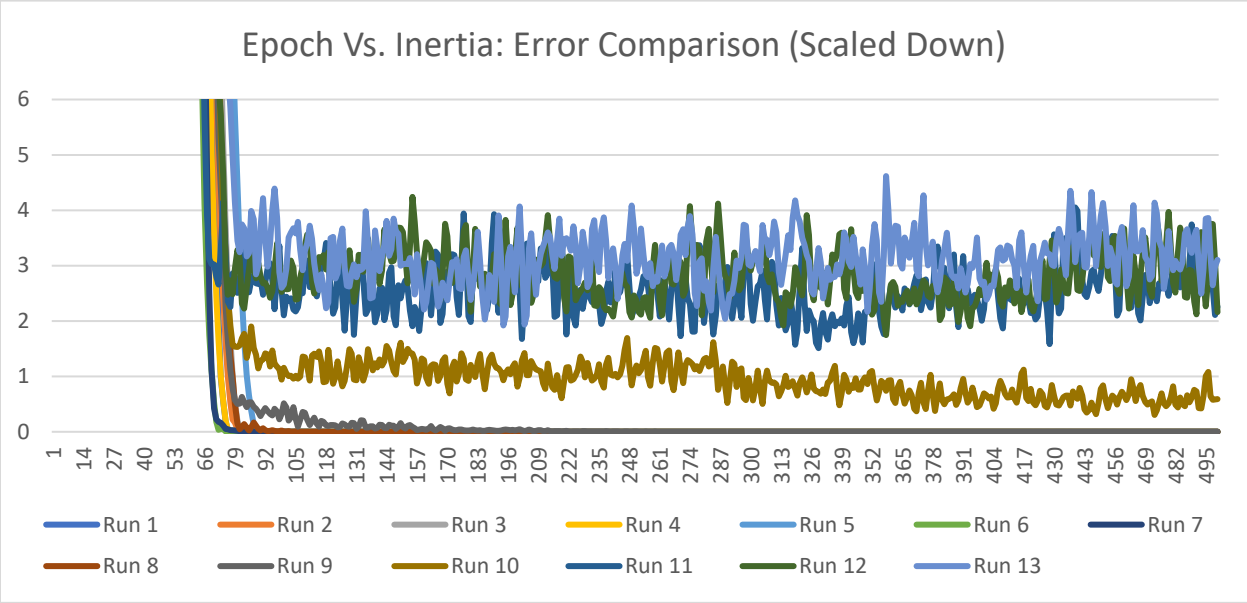

Figure 12. *All 500 error computations with a focus on lower values.*

Now we are able to actually do some reasonable analysis on what we were given. Thankfully, we are also able to do this visually due to WebGL. Experiments with a lower inertia were able to hit an error of 0. The global maximum of the function is at (20, 7), which they were able to find.

However, as we approach an inertia of 1 (without actually hitting it), it seems to have a major impact on how fast the simulator is able to converge down to a single point. This is especially the case with runs where the inertia is 0.9 or higher (Runs 9 and beyond). This is why the simulation was configured to cut off at 500 updates. It appears that it would go almost infinitely without a cutoff.

4.4. A Visual Proof Of Concept

Running the WebGL simulator, we can actually see, visually, how the cluster of particles move. We can also use this visual aspect to prove our observations about inertia and its relation to how fast it can converge down to a single point… if ever.

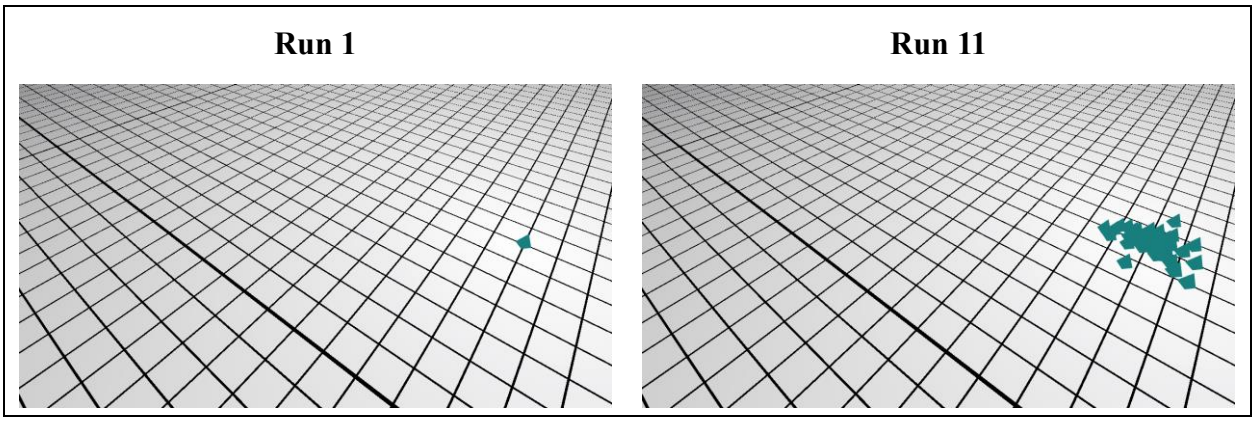

Figure 13. *Visual Comparisons of Run 1 to Run 11 after the 500th update.*

From the visual analysis, the particles are moving around the point at (20, 7) but are, indeed, unable to approach that point and stabilize there. After leaving the simulation to run for several hours, the particles still have not been able to converge, with the error being seemingly random. Of course, if we look at the equation given in Figure 5, a higher inertia would explain why this phenomenon occurs. Here's a segment of the equation:

$$velocity' = inertia * velocity \ldots$$

Figure 14. *A portion of Figure 5.*

The lower the inertia, the faster the velocity slows down. When it is at a value like 1.0, this multiplication is essentially voided. However, what would happen if we were to expand inertia to values beyond 1.0?

4.5. Exploring the Impossible: Inertia Values Over One

We will add two new runs into the series: 2.0 and 5.0. This makes for a total of 15 runs. It turns out that, if we set inertia to higher values, the range of error actually multiplies and circles around the point. The initial graph, including all 15 runs looks the same as the previous one. Therefore, it will not be shown (It is included in the Excel document, regardless). However, the interesting case comes when we set the range of the vertical axis down to 0 to 60.

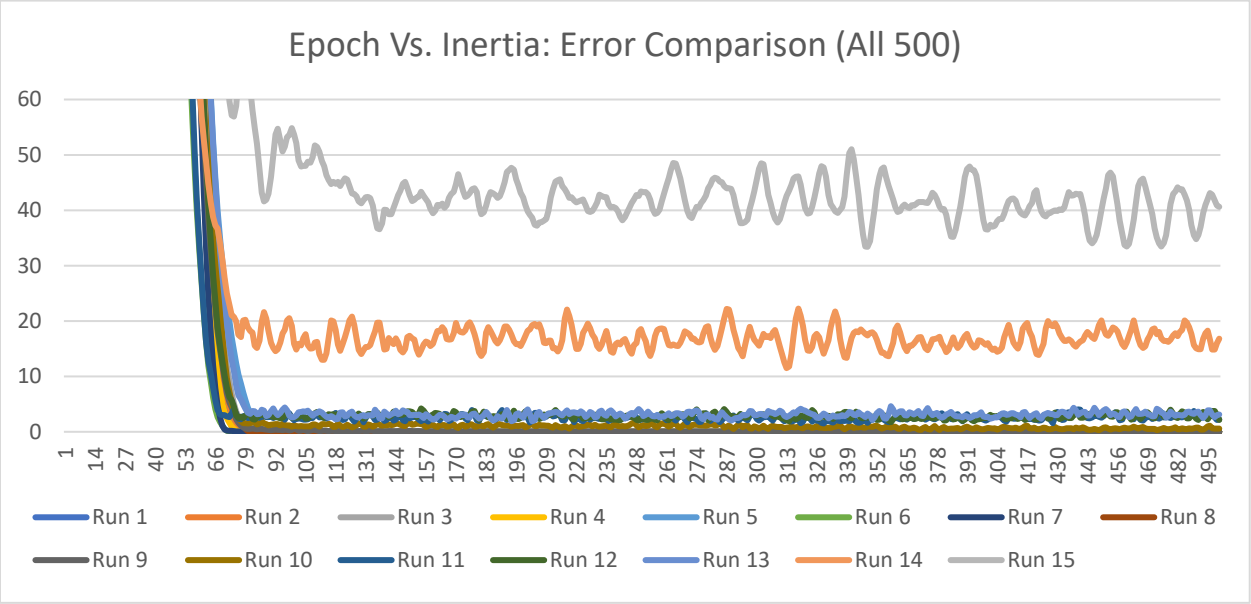

Figure 15. *All 500 error computations with a focus on lower values.*

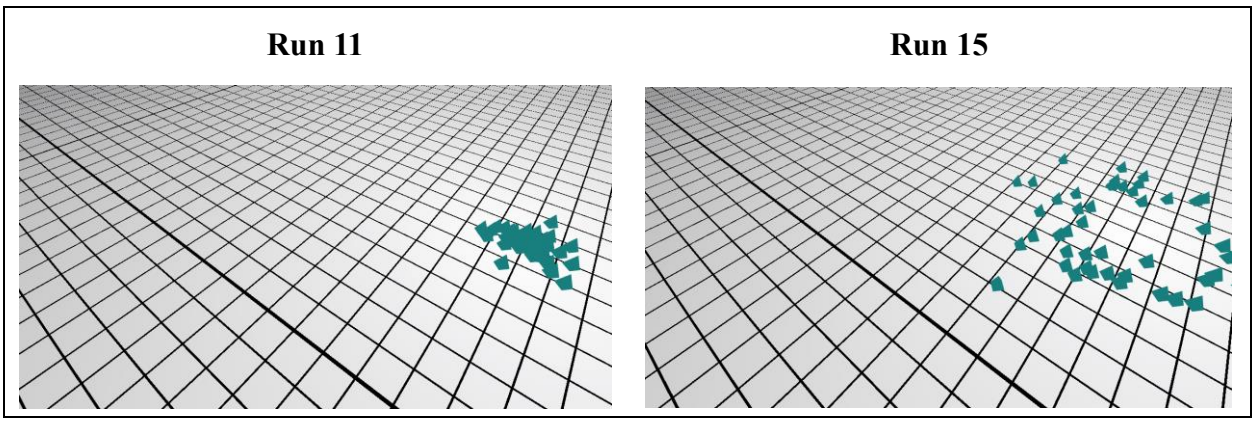

Figure 16. *Visual Comparisons between Run 11 and Run 15 after the 500th update.*

4.6. PREDICTING ERROR

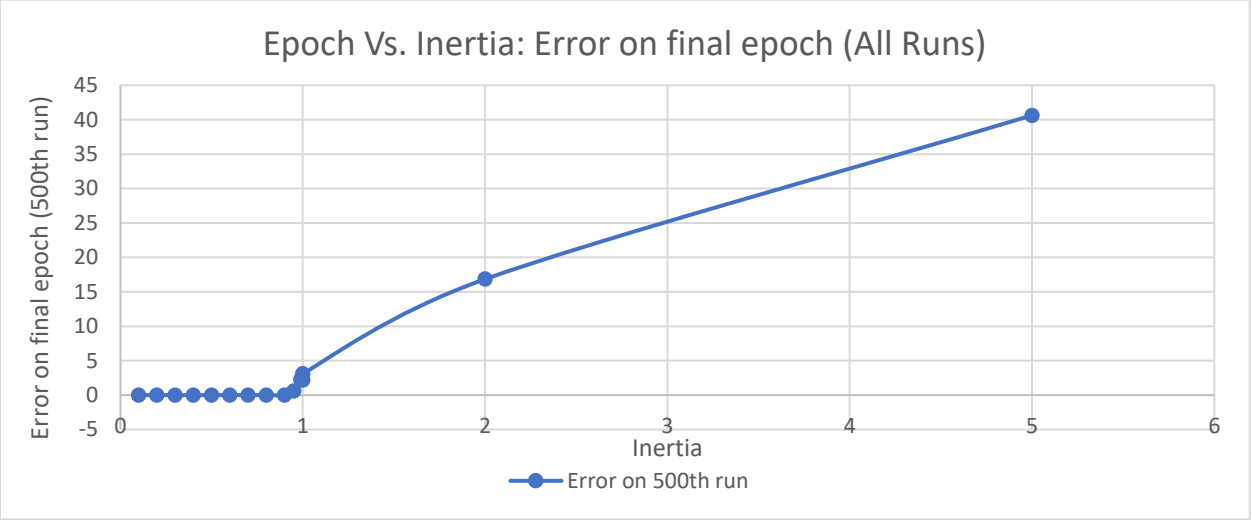

Figure 17. *Relation between Epoch and Inertia based on the 500th update for all runs.*

Based on the graph above, I began to investigate if there was a way to predict the error based on the value of inertia given. Aside from the values prior to 1, it appears that there is a regression that can be used to predict the error for values for 2.0 and beyond. It turns out, there is a relation.

$$error_{r(500)} = -5.8346 + 8.928579x + 0.016795086x^2$$

Figure 18. *Logarithmic Regression Formula for the relation between Inertia and Error.*

We can prove this by simply running more tests.

Inertia	2	3	5	10	20	50	1000
500th Update Error	16.8442	29.224	40.6278	87.6966	153.186	492.029	25717.8
Predicted Error	12.0897	21.1023	39.2282	85.1307	179.455	482.582	25717.8

Figure 19. *Table comparing simulation to equation in Figure 18.*

The results are not perfect. As shown in Figure 15, the results are unstable, but are shown to be fit within a certain range. We could get more accurate by simply taking the average of each equation, but seeing as we get accurate results among those 7 values of inertia (especially up to 1000), the equation will suffice. It should also be noted that the inertia should never actually go over 1. Though, the fact that a pattern actually exists for values above 1.0 is an interesting observation. The fact that the particles orbit around the global maximum, and how that orbiting radius gets larger as the inertia gets larger is a nice effect. It's also possible to take the average of every single point and get the global maximum through this method, as all of the points are orbiting around it.

4.7. Determining when convergence occurs

Saving the simplest observation for last, we can use the data given to determine if inertia affects when the system converges. We will use a threshold or error to determine whether the system has converged or not. In this case, an error below 0.05 will suffice.

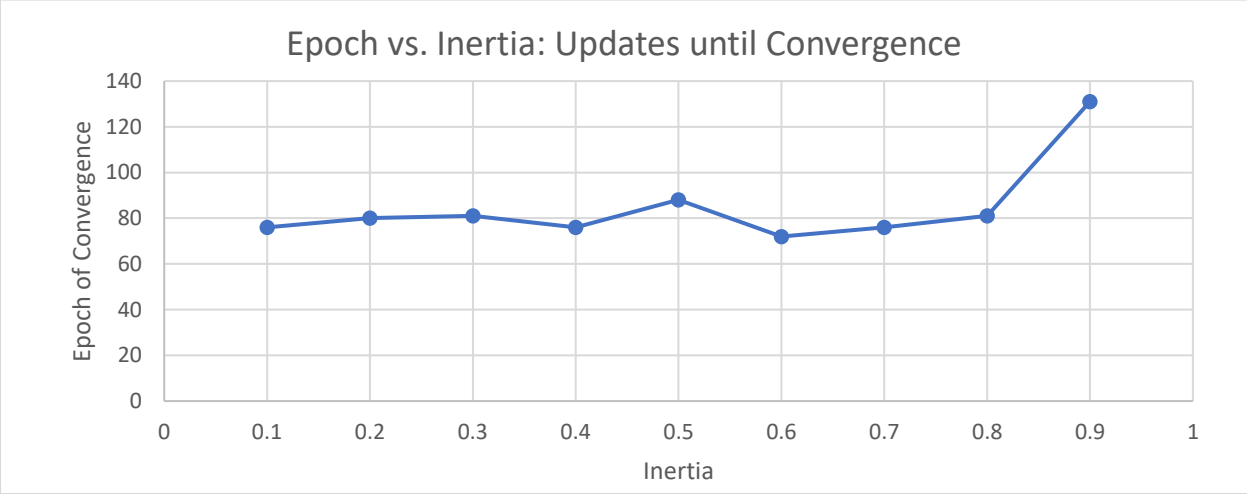

Figure 20. *Moments each run converged.*

The runs where the inertia was 0.95 or higher did not converge at all within the 500 updates.

5. EXPERIMENT 2: EXPLORING MULTIPLE FUNCTIONS

Q1 (Figure 2 in Section 3.1) was only one equation that happened to have a single maximum. However, there are other functions that can be plugged into the simulator. Q2 exists, and we can also implement our own equations as well.

5.1. Q2. A CASE OF MULTIPLE MAXIMA

Q2 has a local maximum at (-20, -7) and a global maximum at (20, 7). This equation is set to prove a point. The algorithm given is not guaranteed to always go to the global maximum.

5.2. Q3 (THE RIPPLE)

Now it's time to get creative. We will use a custom function that builds off of the cosine trigonometric function and see how it acts in the simulation. Behold:

$$Q3(p_x, p_y) = \cos\left(\sqrt{x^2 + y^2 + \sin(10 - x)}\right)\sqrt{x^2 + y^2}$$

Figure X. *Q3 (The Ripple), a 3D equation with infinite maxima.*

This equation is accessible in the WebGL simulation in the "Equation" tab, being labeled "Q3".

5.2.1. THE CATCH

This function is shaped like a ripple (hence the name), but it has an interesting twist to it, because it has an infinite number of maxima that is defined depending on the range of the search area. Its maximum areas are guaranteed to be at corners of the search area, and it can be any one of the corners.

6. EXPERIMENT 3: TRANSCENDING DIMENSIONS. THE 4TH DIMENSION

6.1. THE THEORY

This algorithm can be expanded into support an infinite number of dimensions. As long as the equations given support N dimensions, the search space is in N dimensions, and the particles are allowed to roam freely in those N dimensions, it will work. Therefore, for our last experiment, we will transcend a dimension. In the previous experiments, we were given equations that returned a Z value. The goal was simple. Just have the particles seek the spot where the function returns the best value. However, in theory, we can do this with a four-dimensional formula as well, and have it display in three-dimensional space, just like our previous example had a three-dimensional formula display on a two-dimensional plane.

Thankfully, the WebGL simulator was written with this extension in mind, and we can expand the simulator into the third dimension by simply modifying the formulas to tack on an additional dimension. Equations such as the distance formula are easily expandable. Meanwhile, equations for velocity above were specified as "handling for all axes" for a reason. In the simulator, a value was computed for each dimension, being the X and Y axes. But since we are transcending a dimension, we can simply add on a third computation to calculate the Z axis value. This will work for Z velocity, Z positioning, Z personal best, and Z global best.

For the sake of simplicity, we will use a modified version of Q1, and have its max value shown.

$$\max\left\{Q1_{4D}(p_x,p_y,p_z) = 100 * \left(1 - \frac{\sqrt{(p_x-20)^2 + \left(p_y-7\right)^2 + (p_z-7)^2}}{\frac{1}{2}\sqrt{{max_x}^2 + {max_y}^2 + {max_z}^2}}\right)\right\} = (20,7,7)$$

Figure X. *Q1 equation extended to the fourth dimension with max (pdist and mdist embedded).*