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 it is applied. Generally, this algorithm is used to find the global maximum in an equation given. The equation is (in this experiment) a three dimensional equation, where x and y are both used, and return a z coordinate. These variables can be graphed onto an actual 3D terrain and be visually shown.

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 constantly 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}
- **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