BIOMIMETIC ALGORITHM FOR UAV SWARM

BY: OWEN HAMRO AND JUSTIN FERRALES

AERO 470-01 – TEAM PROJECT

ABSTRACT

- Simulation utilizes Particle Swarm Optimization (PSO) for search and rescue operations employing UAVs.
- Parameters used include the number of particles, iterations, and factors influencing particle movement.
- Initializes particles with random positions and velocities, followed by iterative updates based on velocity and fitness evaluation.
- Tracks particle trajectory to observe swarm behavior and convergence towards the target location.
- Outputs final swarm position and total iteration count for analysis.
- Demonstrates the practical application of PSO in decentralized UAV swarm optimization scenarios, such as search and rescue missions.

INTRODUCTION

- Search and rescue operations with UAVs are a popular trend in today's world of engineering.
- Particle Swarm Optimization (PSO) acts a biomimetic algorithm, inspired by collective behaviors observed in nature.
- This project utilizes the PSO algorithm to simulate UAV capabilities in search and rescue missions.
- PSO, previously developed in this class, can guide the UAV swarm towards the target location while updating its trajectory.
- The project serves to demonstrate the ability of biomimetic algorithms to accurately simulate UAV technology.

[1] https://www.kdcresource.com/insights-events/the-rise-of-swarm-drones-a-look-at-the-latest-advancements-in-uav-technology/

PSO ALGORITHM

- The code defines a Particle class for Particle Swarm Optimization (PSO).
- Particle positions and velocities are initialized in the constructor.
- Position and velocity updates are performed with `updatePosition()` and `updateVelocity()` methods, incorporating random factors and the global best position.
- Fitness values are updated using `updateValue()` based on objective functions `f()` and `f1()`.
- Objective functions `f()` and `f1()` calculate Manhattan and Euclidean distances to the target location.
- Random values contribute to the stochastic nature of the PSO algorithm.

SWARM FUNCTION ALGORITHM

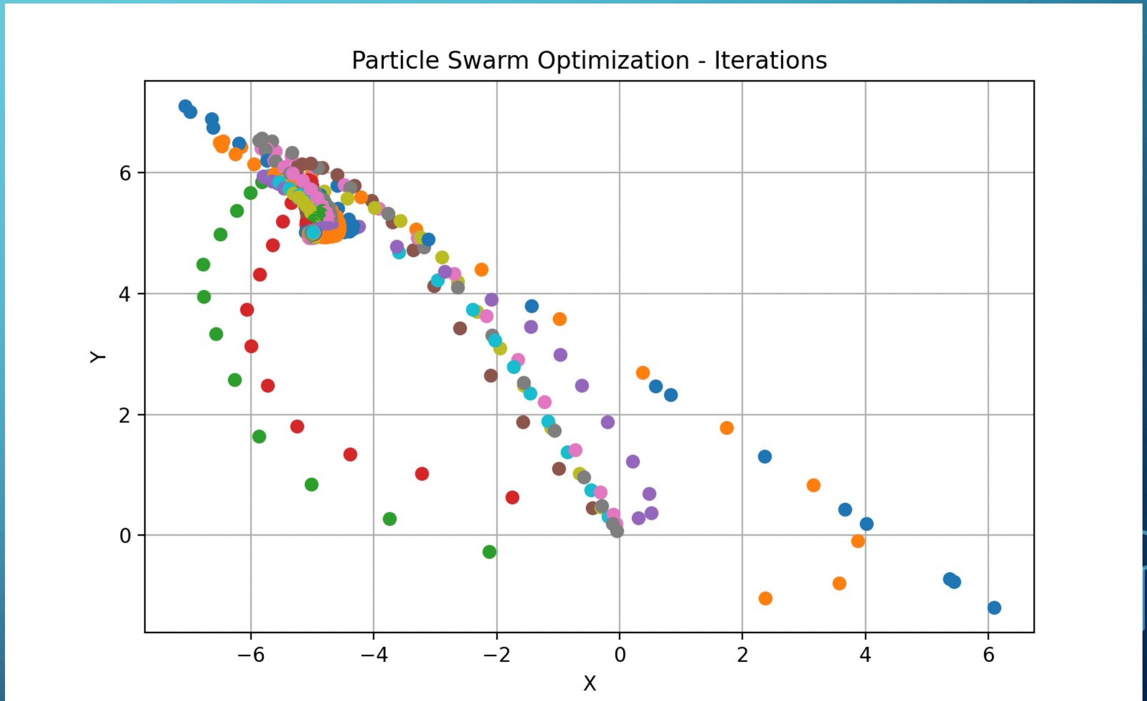
- Function `runUAVSwarm` executes the PSO algorithm for search and rescue.
- It takes parameters such as the number of particles, iterations, initialization values, and target location.

Inside the function:

- Particles are randomly initialized within specified bounds.
- PSO particles are created with positions, velocities, and other parameters.
- The global best position `g` is initialized with the first particle's position.
- Particle positions are iteratively updated based on velocity and position functions.
- Fitness of each particle is evaluated based on proximity to the target.
- `g` is updated if a particle's fitness is better than the current global best.
- Particle positions are plotted for visualization.
- Information regarding optimization, like final swarm position and total iterations, is printed.

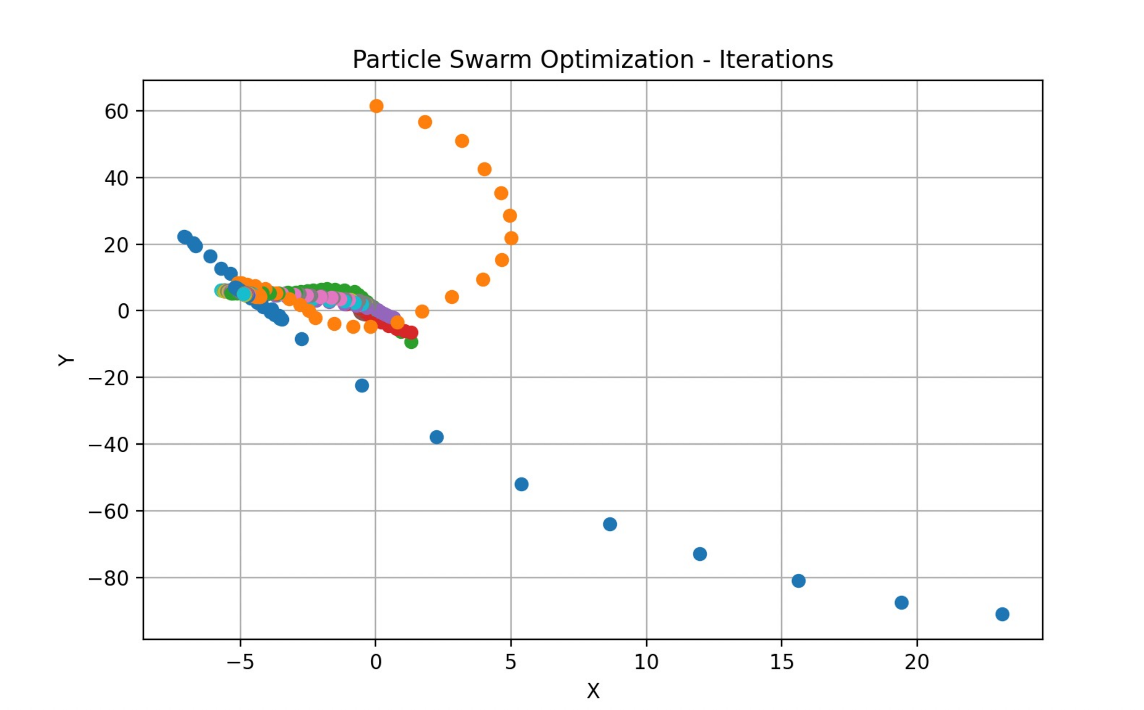
RESULTS (CASE 1)

- **Parameters:**
  - 10 particles, 50 iterations, [-10,10] initial domain, (-5,5) target location
- **Solution:**
  - The PSO flocked towards [-5.00295044 5.00016856], the PSO started at [-3.36116798 4.71399873], and the total amount of points generated: 500



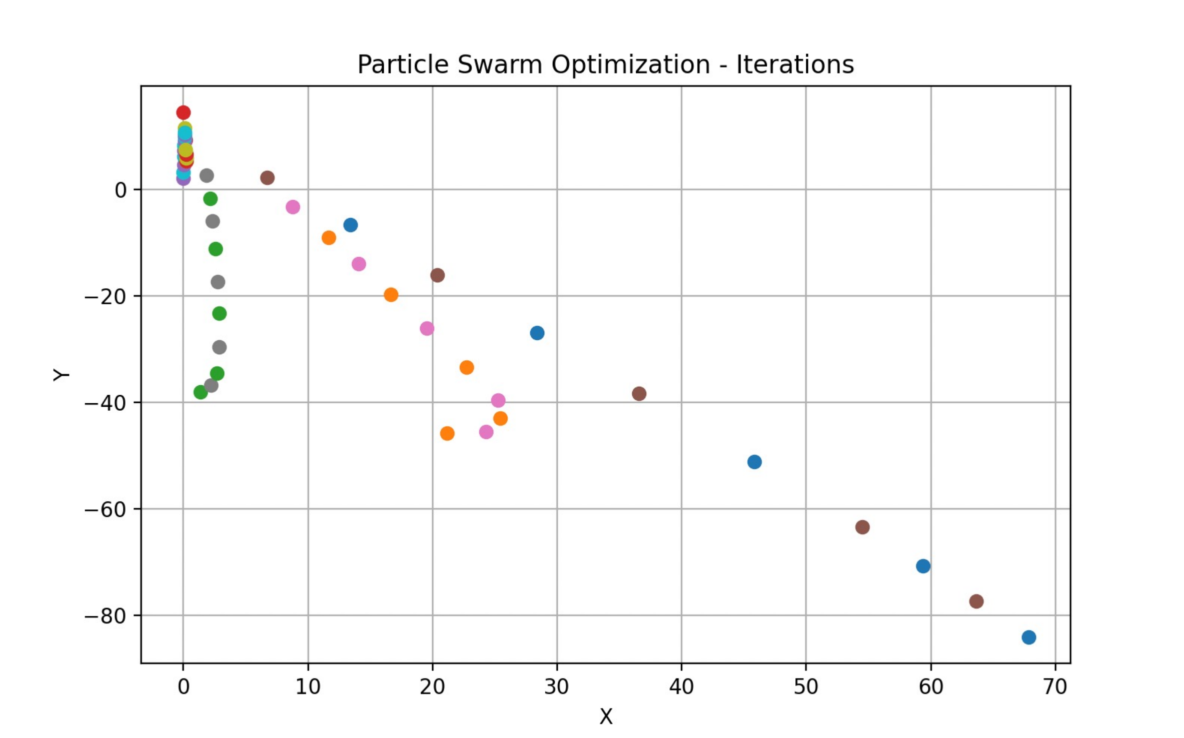
RESULTS (CASE 2)

- **Parameters:**
  - 10 particles, 50 iterations, [-100,100] initial domain, (-5,5) target location
- **Solution:**
  - The PSO flocked towards [-4.88239478 5.01546106], the PSO started at [23.13023152 -90.86826327], and the total amount of points generated: 500



RESULTS (CASE 3)

- **Parameters:**
  - 5 particles, 10 iterations, [-100,100] initial domain, (-10,10) target location
- **Solution:**
  - The PSO flocked towards [0.09438848 9.72894709], the PSO started at [67.82654405 -84.01207231], and the total amount of points generated: 50



ANALYSIS (PARAMETERS)

- The algorithm's effectiveness is heavily dependent on the parameters chosen for the simulation.
- Changing values such as the target location and the domain of the UAV's search can determine how successful the simulation is.
- Number of particles and iterations also have an effect the simulation's success, although not as drastic as the parameters.

ANALYSIS (CASE 1)

- Case 1 uses a healthy number of particles and iterations, as well as a relatively small initial domain to search.
- This allows the UAV simulation to have sufficient “resources” to locate the target destination at (-5,5) fairly quickly.
- Only after a few iterations, the UAV is able to narrow in on the target destination.
- Outcome: Success

ANALYSIS (CASE 2)

- Case 2 uses the same number of particles and iterations, but with a larger initial domain to search.
- The larger domain shows that the UAV is rather far from the target destination at the start of its search but is still able to find the target destination.
- Compared to Case 1, the printed solution displays that the particle trajectory was not as accurate due to the larger domain.
- Outcome: Success

ANALYSIS (CASE 3)

- Case 3 cuts down on the number of particles and iterations, keeping the larger initial domain to search and a different target of (-10,10).
- This forces the UAV simulation to have insufficient “resources” to locate the target destination, trending in the right direction but never making it.
- The flock’s trajectory is accurate for its y-axis at nearly 10; however, the x-axis trajectory is off by a good margin at roughly 0.
- Outcome: Failure

CONCLUSION

- The PSO algorithm and swarm implementation highlight the power of biomimetic algorithms.
- Similar to what is expected in nature, the UAV's success is determined by how large of an area it needs to search, as well as its resources in doing so.
- As long as the algorithm has a sufficient number of iterations, particles, and initial search domain, the UAV is able to find the target destination.
- Although the implementation is far from perfect, its higher-than-expected success rate reveals the power of simulating real-world tendencies.

APPENDICES

- A.1: PSO Code
- A.2: Swarm Implementation

A.1

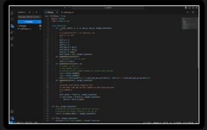
```python
import random
import numpy as np

class Particle:
    def __init__(self, x, v, w, phi_p, phi_g, target_location):
        """
        x is position R^n, v is velocity, etc
        self.x = x, etc
        """
        self.x = x
        self.v = v
        self.w = w
        self.phi_p = phi_p
        self.phi_g = phi_g
        self.p = x.copy()
        self.value = f(x, target_location)
    def updatePosition(self):
        # implement x = x+v on self
        self.x += self.v
    def updateVelocity(self, g):
        # random r_p and r_g
        # then update self.v = ...
        # picking uniform random randoms on uniform distribution
        r_p = random.random()
        r_g = random.random()
        self.v = (self.w*self.v) + ((self.phi_p*r_p)*(self.p - self.x)) + ((self.phi_g*r_g)*(g-self.x))
    def updateValue(self, target_location):
        """
        calculate self.value, based on f(x)
        in the mean time may as well update my best know position
        # ie update p
        """
        self.value = f(self.x, target_location)
        if self.value < f(self.p, target_location):
            self.p = self.x.copy()


def f(x, target_location):
    # Manhattan distance-based objective function
    # Assuming the target location is at (0, 0)
    return np.sum(np.abs(x - target_location))

def f1(x, target_location):
    distance = np.linalg.norm(x - target_location)
    return distance

```



Ln 1, Col 1 Spaces: 4 UTF-8 LF Python 3.12.1 64-bit

A.2

```python
# Team Project Main Script
import numpy as np
import matplotlib.pyplot as plt
import random
import PSO

# Define UAV swarm function
def runUAVSwarm(numParticles, numIterations, x, v, w, phi_p, phi_g, targetLocation):
    """
    Using PSO to do a search and rescue/find on a target location.
    Args:
        numParticles (int): how many particles per iteration
        numIterations (int): how many iterations do you want to run
        x (int): random value, negative and positive used in initializtion of particles
        v (int): random value, used as random initial velocities
        w (double): *must be between 0-1* self declared variables used in updating particle velocity
        phi_p (double): *must be between 0-1* self declared variables used in updating particle velocity
        phi_g (double): *must be between 0-1* self declared variables used in updating particle velocity
        targetLocation ([int, int]): must be in form np.array([int, int])
    """
    S = numParticles
    particles = []

    # Generating numParticles with random initial positions and velocities
    for s in range(S):
        x = np.random.uniform(-x, x, 2)
        v = np.random.uniform(-v, v, 2)
        particles.append(PSO.Particle(x, v, w, phi_p, phi_g, targetLocation))

    # Initializes global best position
    g = particles[0].x.copy()

    # Used to store position of particles
    positionVals = []
    totalPoints = 0
    n = 0

    # Create figure
    fig, ax = plt.subplots(figsize=(8,5))

    # Iterates and updates each position and velocity
    # If particle is better than global best, update g
    # Also plots particle locations
    while n < numIterations:
        for particle in particles:
            particle.updateVelocity(g)
            particle.updatePosition()
            particle.updateValue(targetLocation)
            if particle.value < PSO.f(g, targetLocation):
                g = particle.x.copy()
                positionVals.append(g)
            ax.plot(particle.x[0], particle.x[1], 'o')
            totalPoints += 1
        n += 1

    # Plot characteristics
    print("The PSO flocked towards", positionVals[-1])
    print("The PSO started at", positionVals[0])
    print("The total amount of points generated")
    print(totalPoints)
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_title('Particle Swarm Optimization - Iterations')
    plt.grid()
    plt.show()

# Calls function with determined area to search, target location,...
runUAVSwarm(5, 10, 100, 5, 0.8, 0.1, 0.1, np.array([-10, 10]))
```