Robotic Arm Control Using Kinematic Modeling and PSO

Introduction

This project focuses on controlling a four-link robotic arm using forward kinematics and Particle Swarm Optimization (PSO) to find optimal joint configurations. The goal is to ensure that the robotic arm can reach a target position while avoiding collisions with obstacles in its environment. The system is implemented in Unity, utilizing scripts to manage kinematic updates, optimize joint angles using PSO, and visualize the arm's movement in real-time. This report details each component of the system, including the mathematical foundations of forward kinematics, PSO optimization, collision avoidance, and visualization techniques.

Kinematic Calculations

Kinematic modeling is essential for calculating the position and orientation of each joint in the robotic arm. In this project, the kinematics of the four-link robotic arm are computed using forward kinematics, which requires the use of Denavit-Hartenberg (DH) parameters to define the spatial relationships between the links.

The transformation matrix for each joint can be written as follows:

$$T_i = \begin{bmatrix} R_i & p_i \\ 0 & 1 \end{bmatrix}$$

where:

- T_i is the transformation matrix for joint i,
- R_i is the rotation matrix, and
- p_i is the position vector.

The rotation matrix R_i can be expressed in terms of Euler angles θ_i, α_i, and β_i for the rotations around the z, y, x axes, respectively:

$$R_i = R_z(\theta_i) * R_y(\alpha_i) * R_x(\beta_i)$$

In Unity, these rotations are applied using quaternions to avoid the problems of gimbal lock. Unity's Quaternion.Euler function is used to convert Euler angles into quaternions, which are then applied to the joints of the robot. For example, the rotation of the base joint is updated as follows in the script:

```
Base.transform.rotation = Quaternion.Euler(Angles1);
```

Class: FourLinkRobot

Method: Update

The "*Update*" method is invoked every frame, ensuring that the robot's kinematics are updated continuously based on the current joint angles. The method calls "CalculateKinematics", which computes the forward kinematics for each joint and link.

Method: CalculateKinematics

This method is responsible for calculating the position and orientation of each link using forward kinematics. The key formula used is the transformation matrix that defines the relationship between consecutive links. The method first calculates the base rotation and then propagates this transformation through each joint, updating the position and rotation of the links accordingly.

For example, the position of Joint1 is calculated as:

```
Joint1.transform.position = Base.transform.position + Base.transform.up *
                            linkLength;
```

Particle Swarm Optimization (PSO)

PSO is used in this project to optimize the joint angles of the robotic arm. Each particle in the swarm represents a potential solution (i.e., a set of joint angles), and the fitness of each particle is evaluated based on how well the solution moves the end-effector to the target position without causing a collision.

The position and velocity of each particle are updated using the following equations:

$$V_{new} = w * V_{old} + c1 * r1 * (p_{best} - X) + c2 * r2 * (g_{best} - X)$$

$$X_{new} = X_{old} + V_{new}$$

Where:

- w: Inertia weight,
- c1, c2: Cognitive and social coefficients,
- r1, r2: Random values between 0 and 1,
- p_best: Personal best position,
- g_best: Global best position.

Also, these variables are tuned for optimal balance between goals:

Variables and Parameters in RA_PSO

General Parameters

- ***numParticles***:
    - **Description**: The number of particles in the swarm.
    - **Purpose**: A higher number of particles allows for better exploration of the solution space, but increases computational time.
    - **Typical Value**: 1000 particles.
- ***numIterations***:
    - **Description**: The maximum number of iterations for the PSO algorithm.
    - **Purpose**: Determines how long the PSO will search for an optimal solution. More iterations increase the likelihood of finding a better solution but require more computation.
    - **Typical Value**: 100000 iterations (can be reduced for hardware limitations).
- ***fitnessThreshold***:
    - **Description**: A threshold value for fitness, below which the algorithm stops.
    - **Purpose**: Enables early stopping if a sufficiently good solution is found.
    - **Typical Value**: 1.0 (indicating that fitness values below this are considered optimal).

Penalty Weights in Fitness Function

- ***positionPenaltyWeight***:
    - **Description**: Weight for the penalty on positional errors.
    - **Purpose**: Scales the penalty applied based on how far the robot's end-effector is from the target position. A higher weight increases the importance of positional accuracy in the fitness evaluation.
    - **Typical Value**: 1.0 (normalizes the position penalty).
- ***rotationPenaltyWeight***:

- **Description**: Weight for the penalty on rotation errors.
- **Purpose**: Scales the penalty based on how much the end-effector's orientation deviates from the target orientation. A higher value makes orientation accuracy more important.
- **Typical Value**: 10 (suggesting that rotation accuracy is crucial in this application).

- ***`overfittingPenaltyWeight`***:
  - **Description**: Weight for the penalty on joint overfitting.
  - **Purpose**: Prevents the robot from making large, unnecessary joint movements that could result in inefficient paths.
  - **Typical Value**: 1.0 (moderate overfitting penalty).
- ***`movementPenaltyWeight`***:
  - **Description**: Weight for penalizing large joint movements between consecutive iterations.
  - **Purpose**: Encourages the robot to move its joints smoothly, avoiding drastic changes between consecutive configurations.
  - **Typical Value**: 1.0 (ensures balanced movement).
- ***`timePenaltyWeight`***:
  - **Description**: Weight for penalizing long transition times between joint configurations.
  - **Purpose**: Encourages faster movement between positions, while avoiding large unnecessary movements.
  - **Typical Value**: 1.0 (suggesting moderate priority on minimizing time).
- ***`collisionPenalty`***:
  - **Description**: A large penalty for collisions between the robot and obstacles.
  - **Purpose**: Discourages configurations where any part of the robot collides with obstacles.
  - **Typical Value**: 1,000,000 (a large value to strongly penalize any collision).

Other Key Variables

- ***globalBestPosition***:
    - **Description**: The best set of joint angles found across all particles and iterations.
    - **Purpose**: This is the optimal solution for the robot's joint angles based on all iterations and particles.
- ***globalBestFitness***:
    - **Description**: The fitness value associated with globalBestPosition.
    - **Purpose**: The lower the globalBestFitness, the better the solution. This value indicates how well the best solution performs in terms of the objective.
- ***allBestPositions***:
    - **Description**: A list storing the best positions of the swarm for every iteration.
    - **Purpose**: Used for visualization, showing the progress of the PSO algorithm over time.
- ***finalPosition***:
    - **Description**: The final computed position of the robot's end-effector after the optimization process.
    - **Purpose**: Provides the last computed end-effector position based on the global best joint angles.
- ***finalRotation***:
    - **Description**: The final computed rotation of the robot's end-effector.
    - **Purpose**: Stores the final orientation of the end-effector after optimization.
- ***particles***:
    - **Description**: An array of Particle objects representing each possible solution in the swarm.
    - **Purpose**: Each particle stores its current position, velocity, best-known position, and fitness value. Particles update their velocities and positions at each iteration based on PSO rules.

Class: PSOManager

Method: Start

The "*Start*" method initializes the PSO process by validating the target position and creating an instance of the RA_PSO class. This is crucial for ensuring that the optimization has a valid target to aim for.

Method: Run

The "*Run*" method in RA_PSO manages the main PSO loop, where particles are evaluated based on their fitness, and their velocities and positions are updated accordingly. The fitness function combines several penalties to ensure that the robot avoids collisions and reaches its target efficiently.

Class: RA_PSO

Method: FitnessFunction

The fitness function in PSO evaluates how well a given set of joint angles moves the robot to its target. It consists of several penalties:

```
fitness = positionPenalty + rotationPenalty + overfittingPenalty +
    movementPenalty + timePenalty + collisionPenaltyValue
```

Collision Avoidance

Collision avoidance is handled by checking intermediate joint configurations for collisions with obstacles. The method "*CheckForIntermediateCollisions*" interpolates between joint configurations to ensure that no collisions occur along the robot's path.

Visualization

The visualization system updates the robot's joint angles in real-time based on the results from PSO. The "*RobotVisualization*" script handles the interpolation between joint angles to ensure smooth transitions between movements.

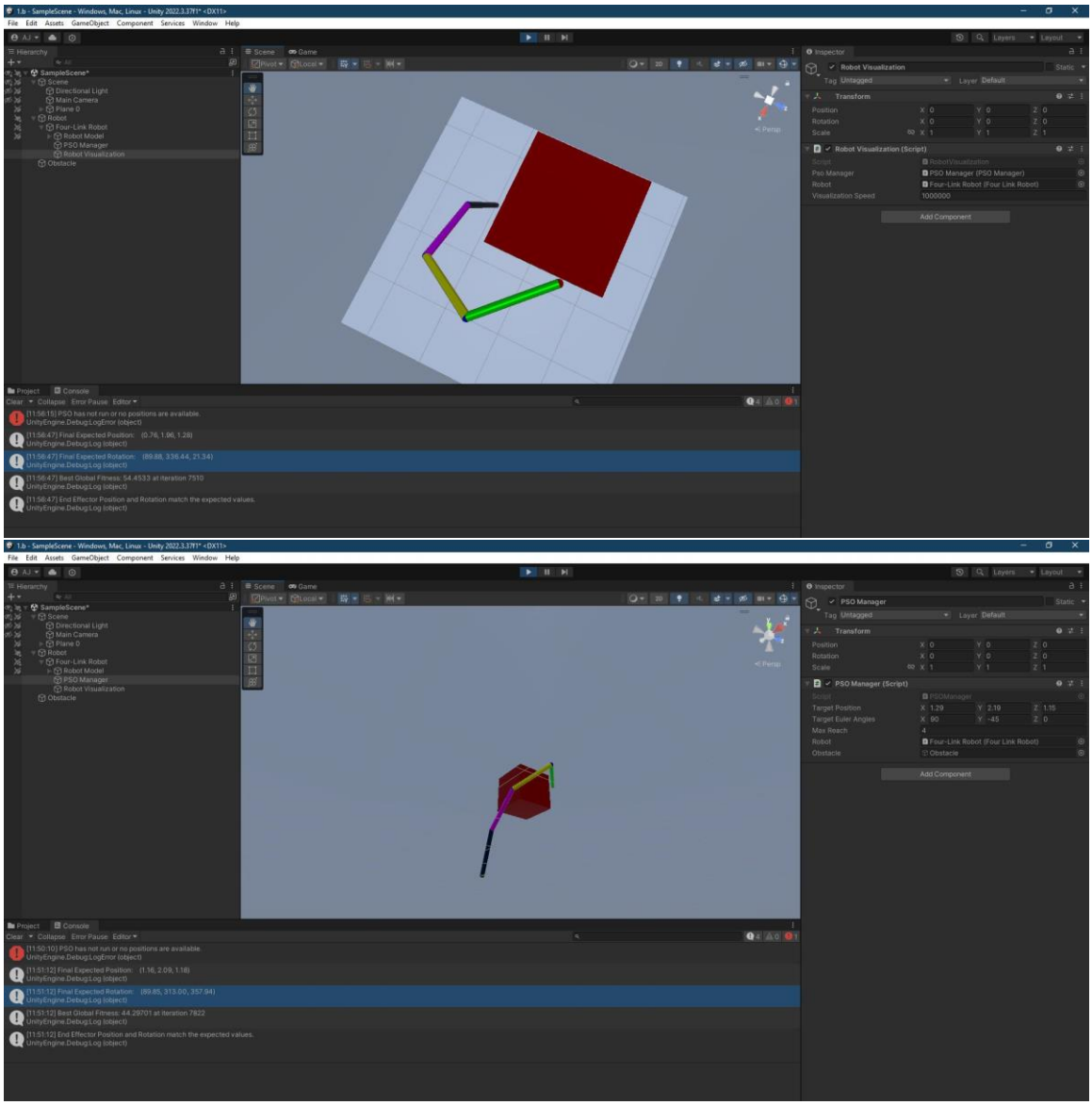

Fig 1 & 2: a sample screenshot of the unity environment showcasing collision-free final position and rotation of the end-effector, and satisfaction of balance between tuned penalties and goals. In this scenario the target position of end-effector is put inside the red cube which serves as an obstacle.

Conclusion

In this report, we have explored the components of the robotic arm control system, including kinematic modeling, optimization using PSO, collision avoidance, and visualization. These components work together to ensure that the robotic arm can efficiently reach target positions while avoiding obstacles, making use of forward kinematics and swarm intelligence.