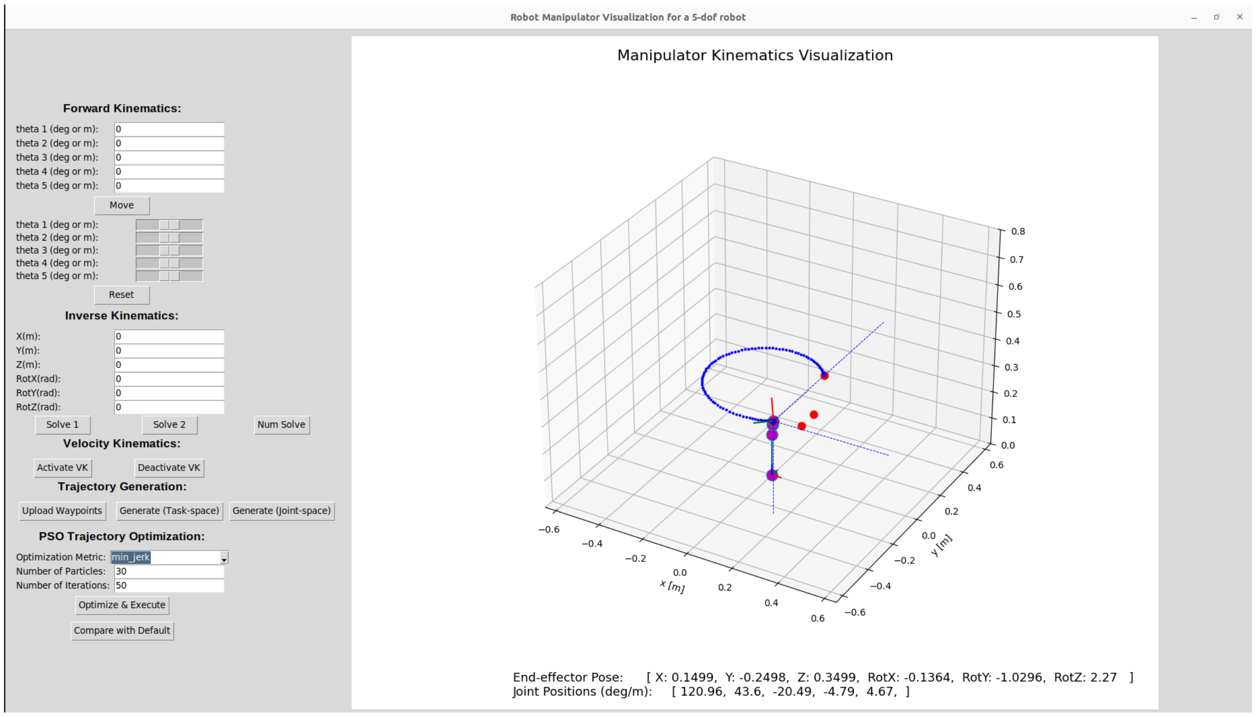
# Particle Swarm Optimization Trajectory Optimization

Optimizing the Trapezoidal Trajectory

Jun Park, Vivian Mak



```
theta 1 (deg or m):  0
theta 2 (deg or m):  0
theta 3 (deg or m):  0
theta 4 (deg or m):  0
theta 5 (deg or m):  0

X(m):       0
Y(m):       0
Z(m):       0
RotX(rad):  0
RotY(rad):  0
RotZ(rad):  0

Optimization Metric:   min_jerk
Number of Particles:   30
Number of Iterations:  50

End-effector Pose:      [ X: 0.1499,  Y: -0.2498,  Z: 0.3499,  RotX: -0.1364,  RotY: -1.0296,  RotZ: 2.27   ]
Joint Positions (deg/m):    [ 120.96,  43.6,  -20.49,  -4.79,  4.67,  ]
```

## 1 Introduction

### 1.1 Project Brief

This project focuses on replicating novel algorithms from a research paper. Specifically, we will be looking at these papers for reference:

1. Trajectory Planning of Robotic Arm Based on Particle Swarm Optimization Algorithm, 2024
2. Time-optimal Trajectory Planning of Dulcimer Music Robot Based on PSO Algorithm, 2020

The first paper focuses on enhancing the motion stability and efficiency of industrial robotic arms through improved trajectory planning with a Particle Swarm Optimization (PSO) algorithm. The Robot Trajectory Planning Particle Swarm Optimization (RTPPSO) algorithm incorporates adaptive weight strategies and random perturbation terms to address the common issue of PSO algorithms getting trapped in local optima. The RTPPSO algorithm is applied to optimize the parameters of an S-shaped velocity profile for the robotic arm's movement. Simulation experiments demonstrate that this approach significantly improves the acceleration and overall motion smoothness of the robotic arm, surpassing traditional trial-and-error velocity planning methods.

The second paper presents a time-optimal trajectory planning method for a dulcimer music robot using an improved Particle Swarm Optimization (PSO) algorithm. The goal is to enable fast and stable striking motions essential for playing the dulcimer by minimizing the total trajectory execution time while adhering to all kinematic constraints, including joint angles, velocities, and accelerations. The authors develop a kinematic model of the robot using the Denavit-Hartenberg (D-H) method and apply fifth-order polynomial interpolation for trajectory planning. The improved PSO algorithm is tailored to handle the full set of constraints, and simulations show that it significantly reduces trajectory time—by 74.35 percent—while maintaining smooth and continuous motion profiles, validating the algorithm's effectiveness.

### 1.2 Goals

The goals for this project are to:

1. Replicate a research paper within the scope of the given time period
2. Learn a novel algorithm
3. Expand on a topic covered in class

### 1.3 Motivation

We wanted to pursue this direction of replicating specific algorithms since trajectory planning is a topic we are both interested in. We wanted to delve deeper into a specific algorithm rather than implement the existing mini-project code with computer vision. Skimming over the abstract of this paper, this algorithm seemed super interesting, especially the Particle Swarm aspect. This was similar to a CompRobo project and this would be interesting to apply similar concepts to a completely different robot.

## 2 Methodology

With research papers proposing novel algorithms, resources to assist building it from scratch is scarce and usually non-existent. To make implementation easier, we broke our project into three phases

1. Do a literature review to understanding the algorithm
2. Build a skeleton pipeline to slowly test functionality
3. Integrate with the existing simulator

### 2.1 The PSO Algorithm

Particle Swarm Optimization is based on the simulating of social behavior. The algorithm uses a swarm of particles to guide its search. Each particle has a velocity and is influenced by locally and globally best-found solutions.

The social behavior is incorporated by using the globally (or locally) best-found solution in the swarm for the velocity update. Besides the social behavior, the swarm's cognitive behavior is determined by the particle's personal best solution found. The cognitive and social components need to be well balanced to ensure that the algorithm performs well on a variety of optimization problems

**PSO Equation**

$$V_d^{(i)} = \omega V_d^{(i)} + c_1 r_1 \big(P_d^{(i)} - X_d^{(i)}\big) + c_2 r_2 \big(G_d^{(i)} - X_d^{(i)}\big), \quad (1)$$

$$X_d^{(i)} = X_d^{(i)} + V_d^{(i)}. \quad (2)$$

Where

- $X_d^{(i)}$ – current position of particle $i$

- $V_d^{(i)}$ – current velocity of particle $i$
- $\omega$ – inertia weight (momentum term)
- $P_d^{(i)}$ – particle $i$'s personal–best position
- $G_d^{(i)}$ – global (or neighbourhood) best position
- $c_1, c_2$ – cognitive and social learning coefficients
- $r_1, r_2 \sim \mathcal{U}(0,1)$ – fresh random numbers each step

The equation contains three components:

**Inertia** $(\omega V)$
keeps the particle moving in its previous direction (exploration).

**Cognitive term** $c_1 r_1 (P - X)$
pulls the particle back toward its own best location (self-experience).

**Social term** $c_2 r_2 (G - X)$
pulls the particle toward the swarm's best location (group experience).

Tuning $\omega$, $c_1$, $c_2$ governs the exploration–exploitation trade-off; our implementation starts with $\omega \in [0.4, 0.9]$ (adaptive), $c_1 = c_2 = 1.5$ and then lets the adaptive rules in `update_particle()` refine them each iteration.

**Velocity-profile choice.** The reference papers apply PSO to an *S-shaped* trapezoidal velocity profile, whose acceleration ramps are themselves smoothed by a sigmoid. For this project we use the **classical trapezoidal profile** (piecewise-constant acceleration, linear velocity ramp). The simpler analytic form lets us derive closed-form expressions for acceleration, jerk, and phase durations—making both the fitness function (Section 2.3) and the simulator integration considerably easier to implement, test, and visualize.

## 2.2 Algorithm Architecture

For the algorithm, we divided them into two classes: 1) `Particle` and 2) `PSO_Trajectory_Optimization`. Each particle in the particle cloud is of class `Particle` and has attributes position, velocity, personal_best, and particle_bounds. The PSO class is more complicated with the following high-level architecture:

```python
class PSO_TrajectoryOptimizer:
    """Particle Swarm Optimization for trajectory parameters optimization.

    This optimizer finds the optimal parameters (t1, t2, v_max) for a trapezoidal
    velocity profile between waypoints, considering metrics like minimum jerk,
    minimum execution time, or a combination of both.
    """

    def __init__(self, start_pos, final_pos, metric="min_jerk", n_particles=50,
                 iterations=100, a_max=10.0, v_max=5.0, ndof=5):
        """Initialize the PSO trajectory optimizer.

        Args:
            start_pos: Starting position/joint values
            final_pos: Ending position/joint values
            metric: Optimization metric ("min_jerk", "min_time", "combined")
            n_particles: Number of particles in the swarm
            iterations: Maximum number of iterations
            a_max: Maximum acceleration constraint
            v_max: Maximum velocity constraint
            ndof: Number of degrees of freedom
        """
        self.start_pos = np.array(start_pos)
        self.final_pos = np.array(final_pos)
        self.metric = metric
        self.n_particles = n_particles
        self.iterations = iterations
        self.a_max_limit = a_max
        self.v_max_limit = v_max
        self.ndof = ndof

        # Total Distance to Cover
        self.distance = np.abs(self.final_pos - self.start_pos)

        # Parameter bounds for t1, t2, v_max
        self.bounds = [(0.1, 0.4), (0.6, 0.9), (0.1, self.v_max_limit)]

        # Particle swarm attributes
        self.particles = []

        # PSO algorithm parameters
        self.alpha = 0.5      # Random Disturbance Coefficient
        self.w_min = 0.4      # Minimum inertia weight
        self.w_max = 0.9      # Maximum inertia weight
        self.alpha = 0.5      # Used for adaptive learning factors
        self.beta = 0.5       # Used for adaptive learning factors

        # Best solution found
        self.global_best = None
        self.global_best_fitness = None

        # Fitness history for tracking progress
        self.fitness_history = []


    def initialize_particles(self):
        """Initialize the particle swarm with random parameters."""
        pass

    def calculate_fitness(self, particle):
        """
        Fitness evaluation with soft quadratic penalties instead of -inf.
        The function still returns 'higher is better'.
        """
        pass

    def update_particle(self, particle, iteration):
        """Update a particle's velocity and position based on PSO equations with adaptive parameters.

        Args:
            particle: The particle to update
            iteration: Current iteration number
        """
        pass

    def optimize(self, verbose=True):
        """Run the PSO optimization process.

        Args:
            verbose: Whether to print progress updates

        Returns:
            tuple: (best_params, fitness_history)
        """
        pass

    def plot_fitness_history(self):
        """Plot the fitness history during optimization."""
        pass

    def get_optimized_parameters(self):
        """Return the optimized trajectory parameters.

        Returns:
            dict: Dictionary containing the optimized parameters
                {'t1': float, 't2': float, 'v_max': float}
        """
        pass

    def create_trajectory_config(self):
        """Create a configuration dictionary for the trajectory generator.

        Returns:
            dict: Configuration for the MultiAxisTrajectoryGenerator
        """

```

On top of using the existing simulator code, one of the main challenges were to integrate our replicated algorithm with the code we wrote. This is done with the python script `pso_integration` that adds the PSO interface with user-chosen metrics, particle numbers, and iteration numbers.

## 2.3 Fitness Function & Optimisation Metrics

The PSO evaluates each particle with a scalar *fitness* value that trades off trajectory **smoothness**, **energy use**, and **execution time**. For a candidate parameter triple $\boldsymbol{\theta} = (t_1, t_2, v_{\max})$ we compute

$$\text{Fitness}(\boldsymbol{\theta}) \;=\; -\Big(w_{\text{jerk}}\, J(\boldsymbol{\theta}) \;+\; w_{\text{energy}}\, E(\boldsymbol{\theta})\Big) \;+\; w_{\text{time}}\, T(\boldsymbol{\theta}), \tag{3}$$

where higher fitness is better. The three component metrics are

- **Smoothness/jerk** $J(\boldsymbol{\theta}) = \dfrac{a_{\max}}{\min(t_1,\, 1-t_2)}$ estimates peak jerk at the acceleration–deceleration corners
- **Energy** $E(\boldsymbol{\theta}) = a_{\max}^2(t_1 + 1 - t_2)$ approximates electrical or mechanical effort.
- **Time** $T(\boldsymbol{\theta}) = v_{\max}$; for a fixed end-to-end distance, a higher $v_{\max}$ implies a shorter motion.

**Metric presets.** We simply adjust the weight triplet $(w_{\text{jerk}}, w_{\text{energy}}, w_{\text{time}})$ to bias the optimiser:

| **Mode** | $w_{\text{jerk}}$ | $w_{\text{energy}}$ | $w_{\text{time}}$ |
|---|---|---|---|
| `min_jerk` | 0.8 | 0.1 | 0.2 |
| `min_time` | 0.1 | 0.1 | 0.8 |
| `min_energy` | 0.2 | 0.7 | 0.1 |
| `combined` | 0.4 | 0.3 | 0.3 |

```python
# excerpt of calculate_fitness()
if self.metric == "min_jerk":
    weights = [0.8, 0.1, 0.2]
elif self.metric == "min_time":
    weights = [0.1, 0.1, 0.8]
elif self.metric == "min_energy":
    weights = [0.2, 0.7, 0.1]
else:  # combined
    weights = [0.4, 0.3, 0.3]

fitness = -(weights[0]*jerk_metric + weights[1]*energy_metric) \
          +  weights[2]*time_metric
```

### Deriving the three metric terms

For our project, we use a *trapezoidal velocity profile* with an accelerate–cruise–decelerate structure. We define the trajectory to normalize and run over $t \in [0,1]$, with $t_1$ the end of the acceleration phase, $t_2$ the start of the deceleration phase and $v_{\max}$ the speed of the cruise. Therefore, we define these as:

$$a_{\max} = \frac{v_{\max}}{t_1}, \quad a_{\min} = -\frac{v_{\max}}{1-t_2}.$$

**Smoothness/Jerk $J(\theta)$.** Ideal jerk is the time-derivative of acceleration, $J(t) = \dot{a}(t)$. In a perfect trapezoid the change from 0 to $a_{\max}$ (and back) happens over an infinitesimal interval, so the true peak jerk is unbounded. Thus, we approximate the worst-case jerk by dividing the peak acceleration by the shorter of the two ramp durations:

$$J(\theta) \approx \frac{a_{\max}}{\min\big(t_1,\, 1-t_2\big)}.$$

This metric penalizes both large accelerations *and* very sharp ramps, encouraging smoother motion.

**Energy** While our hardware uses servo motors, usually, loss in an electrical motor winding scales with the square of current, $P_{\text{Cu}} \propto I^2 R$. Because motor current is almost linearly proportional to torque ($\tau = K_t I$) and since torque is proportional to acceleration ($\tau = I\,\ddot{q}$), loss can be estimated as $\propto \tau^2 \propto (\ddot{q})^2$. In a trapezoidal profile, the magnitude of acceleration is held constant at $+a_{\max}$ during the accelerate phase (duration $t_1$) and at $-a_{\max}$ during the decelerate phase (duration $1-t_2$). Hence, for simplicity of our simulation, we can assume that the total energy loss is proportional to:

$$a_{\max}^2\big(t_1 + 1 - t_2\big)$$

.

**Time $T(\theta)$.** With a fixed point-to-point distance $D = \|q_f - q_0\|$, the overall duration is:

$$\underbrace{t_1}_{\text{accel}} + \underbrace{(1-t_1) - (1-t_2) = t_2 - t_1}_{\text{cruise}} + \underbrace{1-t_2}_{\text{decel}} = 1.$$

Because we normalize the total window to 1, minimizing time collapses to *maximising* the cruise speed. Hence we choose the simple equation of:

$$T(\theta) = v_{\max}$$

.

These metric expressions are simplified for us to evaluate with respect to the physical quantities we care about, making them suitable for our simulated PSO scoring. Most importantly, this formulation lets us switch optimization priorities with a single flag while keeping the rest of the PSO loop unchanged.

## 2.4 Algorithm Pipeline

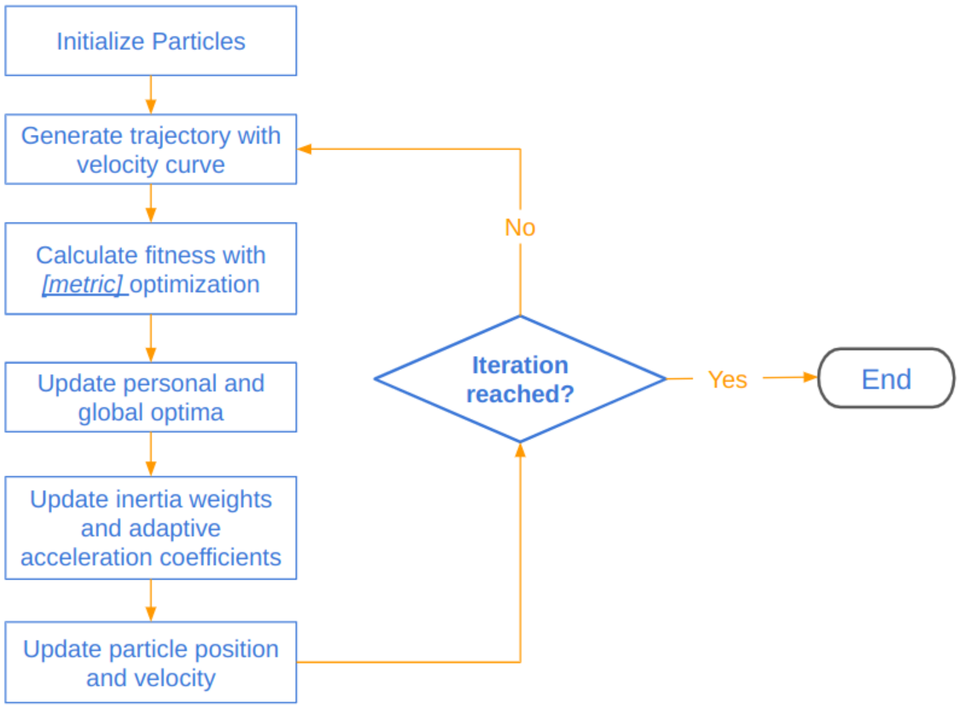

Initialize Particles
Generate trajectory with velocity curve
Calculate fitness with [metric] optimization
Update personal and global optima
Update inertia weights and adaptive acceleration coefficients
Update particle position and velocity
Iteration reached?
No
Yes
End

Figure 1: Diagram of the Particle Swarm Optimization algorithm pipeline

All code is linked in our Github Repo and instructions to run the simulator is on our Repo README

## 3 Results

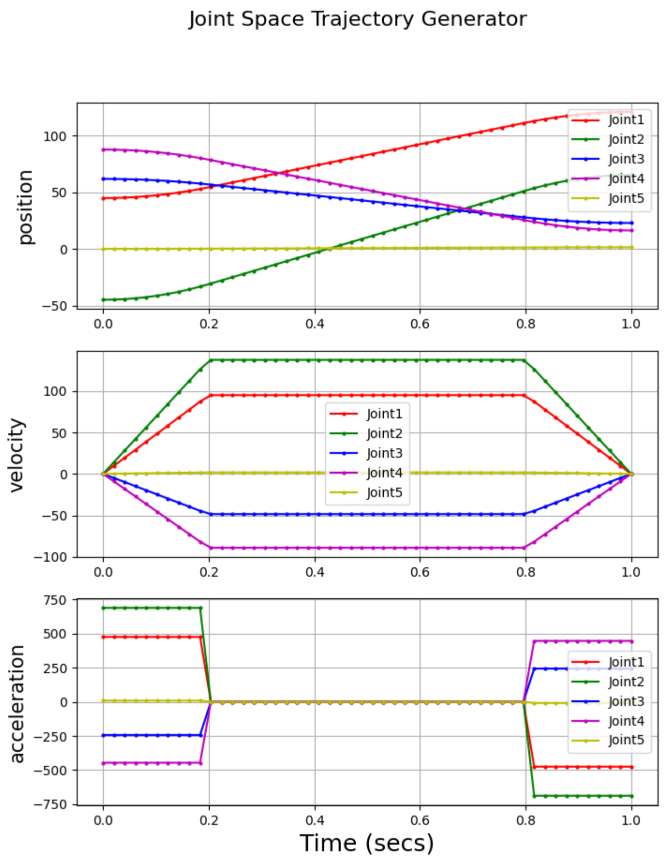


Figure 2: Graph of the position, velocity, and acceleration, over trajectory time for each join

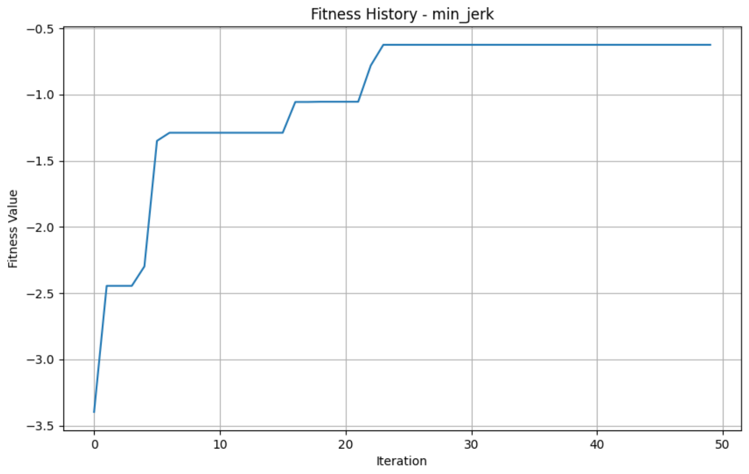


Figure 3: Calculation of the best fitness over each iteration, optimizing to minimize jerk

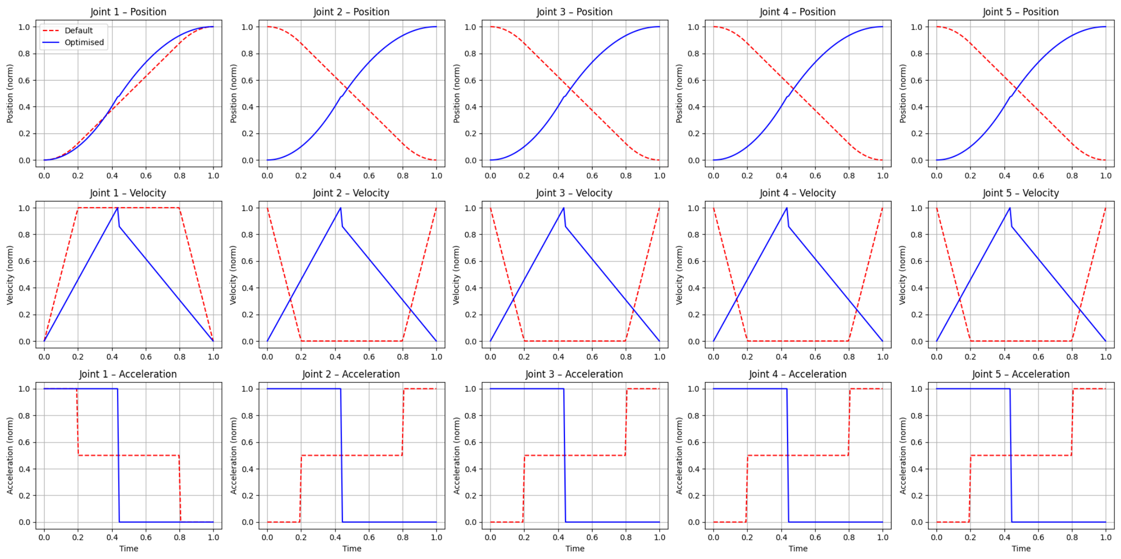


Figure 4: Comparison of default v optimized of the position, velocity, and acceleration, over trajectory time for each joint (Normalized data)

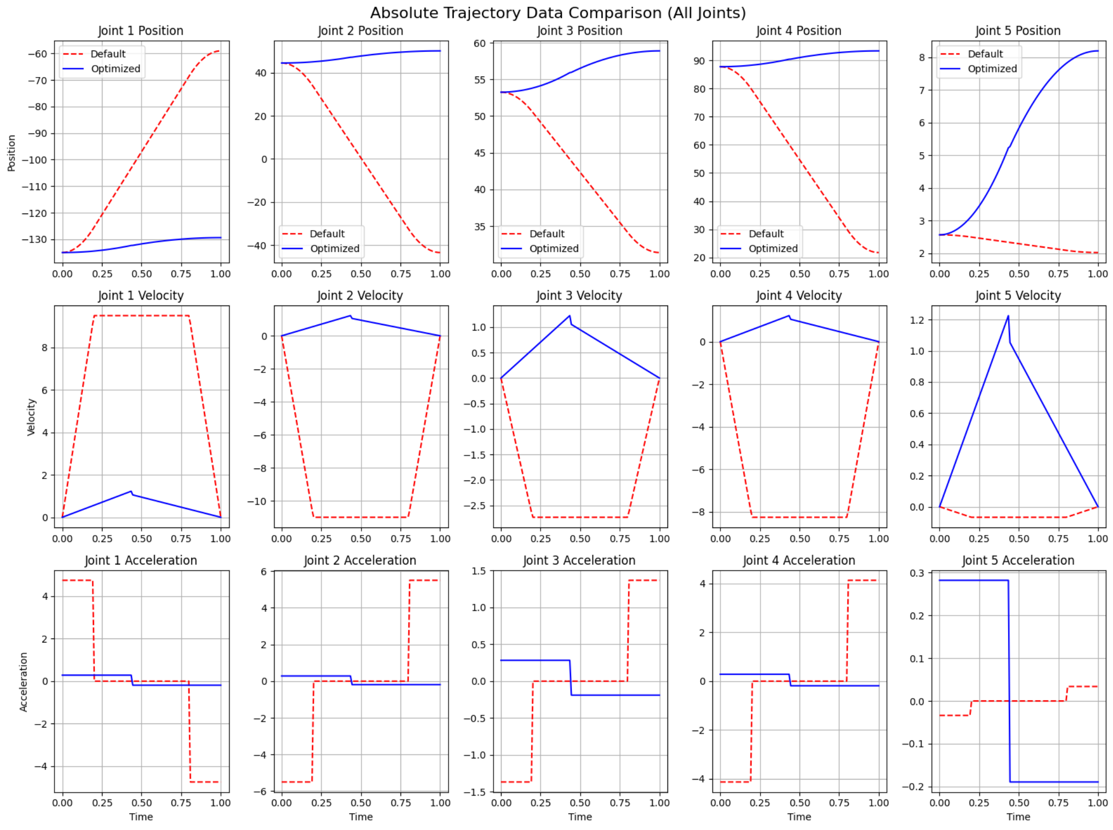


Figure 5: Comparison of default v optimized of the position, velocity, and acceleration, over trajectory time for each joint (Raw data)

## 3.1 Analysis

In these diagrams, we ran the simulator to optimize jerk of the robot arm. To minimize jerk, the goal is to have a near constant or zero acceleration - the less sudden movement, the less deviation from the intended trajectory.

From Figure 3, graphing the fitness history, you can see the minimization of jerk over each iteration. From running the simulator (with 30 particles) several times, the jerk often converges around iteration 25.

Figure 2 shows all the position, velocity, and acceleration of each joint individually. Comparing the default with the optimized curves, shown in Figure 4 (Normalized) and Figure 5 (Raw data), we can see that the velocity curve of each of the joints yields minimum acceleration. Unlike the trapezoidal curve's acceleration which has a very sudden change in acceleration, this optimization keeps the change in acceleration at a minimum. Obviously, it is difficult to have it perfect. The acceleration on joint 5 still has a big spike for acceleration, but the results seem very stable for all the other joints. We can also clearly see how the optimized path (shown in the position graph), fundamentally takes a different path.

This provides evidence that this optimization is able to minimize the jerk to a noticeable level.

## 3.2 Running the Simulator

- Running the simulator to minimize jerk (Youtube Video)

# 4 Conclusion

## 4.1 Project Outcomes

Our implementation achieved two tangible results. First, by tuning the trapezoidal parameters ($v_{\max}$, $t_1$, $t_2$) the PSO optimizer consistently showed differed from the traditional trapezoid, producing visibly smoother movements when minimizing jerk and shorter movements when minimizing time. Second, we wrapped the optimizer in an interactive GUI for the supplied simulator, allowing users to choose a metric, particle count, and iteration with a single click.

## 4.2 Main Challenges

Our largest hurdle was **deciphering the reference paper**. Key definitions (e.g. how "jerk" or "energy" were computed) changed terminology between sections, and several equations omitted units or normalization factors altogether. That ambiguity taught us two lessons:

1. A published paper is not automatically a "ground truth" source; critical reading is indispensable.
2. When reproducing work, you must rebuild missing context (constraints, units, limits) before copying formulas—especially for algorithms like PSO whose behavior depends strongly on the fitness function.

Once the theoretical gaps were bridged, implementation was straightforward, but documentation quality—not code—was the bottleneck.

PSO's behavior turned out to be *extremely* sensitive to the bounds and penalties we supplied. This taught us that an optimizer is only as good as the physical model and constraints it is given—and that users must understand their own system's feasible operating region before pressing the "optimize" button.

On the positive side, PSO adapted gracefully when we switched from minimizing jerk to minimizing time or energy: the same code converged after only minor retuning of weight vectors. Seeing that flexibility firsthand clarified why PSO is popular in domains with conflicting objectives.

## 4.3 Improvements and Next Steps

- **Broader profile library.** Extend the optimiser to cubic, quintic, and S-curve velocity profiles to gauge how sensitive PSO is to profile topology.
- **Real-system constraints.** Incorporate actuator limits (joint speed, torque, power) directly into the fitness function so the optimiser never proposes infeasible moves.
- **Convergence visualisation.** Add a live scatter plot of particle positions and the globe's best point to illustrate convergence in real time.
- **Side-by-side playback.** Enhance the GUI to animate the traditional and optimised trajectories simultaneously, making the differences obvious without switching plots.
- **Hardware Implementation.** Apply the optimization to the HiWonder robot arm to see how this algorithm is translated in the real world.

Ultimately, this project sharpened our ability to *interpret* and *implement* research algorithms, highlighted the importance of well-defined metrics, and demonstrated how a relatively simple optimizer can yield measurable gains in motion quality.