Quadson

Ying Pei Lin

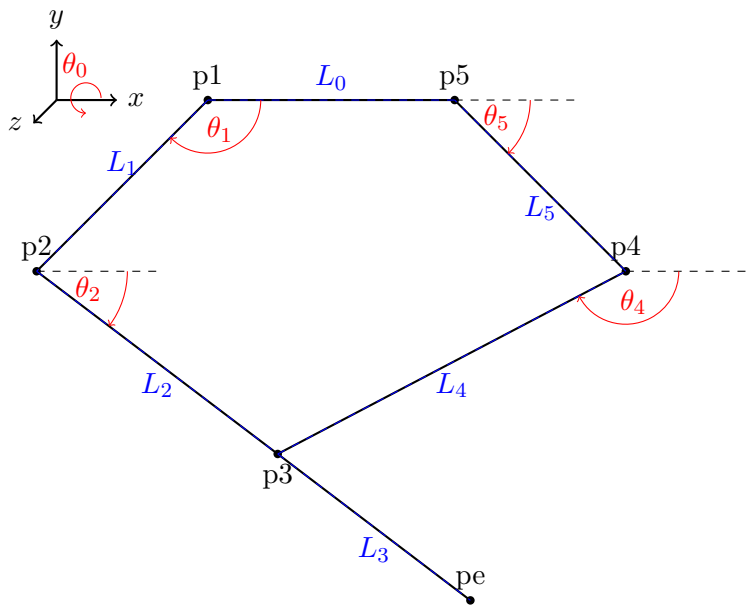
Spring 2025

1 Introduction

Quadson is a 3D printed quadruped robot designed as a platform for exploring advanced robotic motion and control. This document includes its forward kinematics, inverse kinematics, differential kinematics and expands to body kinematics and locomotion. Following the analysis, we will introduce the process of performing simultion in Pybullet and using deep reinforcement learning to improve the robot's control and performance.

2 Forward Kinematics

For each leg of the robot, we define the reference frame with its origin at point p_1. Two motors are positioned at p_1 and p_5 respectively, and the third motor connected to the entire leg assembly, controls the rotation around x axis. The remaining joints in the leg are all passive.



The following algorithm takes the angle of three motors as input and computes the position of the end-effector.

Algorithm 1 Compute Leg End-Point

Require: Motor angles $\theta_0, \theta_1, \theta_5$

1: Compute initial 2D joint positions:

$$p_2 = (L_1 \cos\theta_1, -L_1 \sin\theta_1), p_5 = (L_0, 0),$$
$$p_4 = p_5 + (L_5 \cos\theta_5, -L_5 \sin\theta_5)$$

2: Compute distances:

$$L_{14} = \|p_4 - p_1\|, \quad L_{24} = \|p_4 - p_2\|$$

3: Solve for p_3 using angles:

$$\theta_2 = \cos^{-1}\left(\frac{L_2^2 + L_{24}^2 - L_4^2}{2L_2L_{24}}\right) + \cos^{-1}\left(\frac{L_1^2 + L_{24}^2 - L_{14}^2}{2L_1L_{24}}\right) - (\pi - \theta_1)$$

$$p_3 = p_2 + L_2(\cos\theta_2, -\sin\theta_2)$$

4: Compute endpoint:

$$p_e = p_2 + (L_2 + L_3)(\cos\theta_2, -\sin\theta_2)$$

5: Transform to 3D using rotation matrix:

$$R_{\theta_0} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(-\theta_0) & -\sin(-\theta_0) \\ 0 & \sin(-\theta_0) & \cos(-\theta_0) \end{bmatrix}, p_e = R_{\theta_0} \times \begin{bmatrix} p_e[0] \\ p_e[1] \\ 0 \end{bmatrix}$$

6: **return** p_e

3 Inverse Kinematics

In real world scenarios, controlling position of the end-effector to reach to a certain position is often more pratical than setting the angles directly. For example, to move the end-effector along a defined trajectory, achieving this by manually adjusting the three motor angles is nearly impossible due to the complexity of the joint movements.

Therefore we need the following algorithm to compute the angles of the motor corresponding to a given target position.

Algorithm 2 Compute Joint Angles from End-Point

Require: End-point position (x, y, z)

1: Calculate angle of motor 0:

$$\theta_0 = -\tan^{-1}\left(\frac{z}{-y}\right)$$

2: Translate points from 3D to 2D using rotation matrix:

$$R_{\theta_0} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(-\theta_0) & -\sin(-\theta_0) \\ 0 & \sin(-\theta_0) & \cos(-\theta_0) \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = R_{\theta_0} \times \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

3: Calculate angle 1:

$$L_{1e} = \|(x', y')\|, \quad \theta_{e15} = \tan^{-1}\left(\frac{-y'}{x'}\right)$$

$$\theta_{e12} = \cos^{-1}\left(\frac{L_1^2 + L_{1e}^2 - (L_2 + L_3)^2}{2L_1 L_{1e}}\right)$$

$$\theta_1 = \theta_{e15} + \theta_{e12}$$

4: Compute positions of joints:

$$p_1 = (0, 0), \quad p_2 = (L_1 \cos\theta_1, -L_1 \sin\theta_1)$$

$$p_3 = \frac{(pe_2d \cdot L_2) + (p_2 \cdot L_3)}{L_2 + L_3}, \quad p_5 = (L_0, 0)$$

5: Calculate angle 5:

$$\theta_{350} = \tan^{-1}\left(\frac{-p_3[1]}{L_0 - p_3[0]}\right)$$

$$\theta_{354} = \cos^{-1}\left(\frac{L_{35}^2 + L_5^2 - L_4^2}{2L_{35}L_5}\right), \quad L_{35} = \|p_3 - p_5\|$$

$$\theta_5 = \pi - (\theta_{350} + \theta_{354})$$

6: **return** $(\theta_0, \theta_1, \theta_5)$

4 Differential Kinematics

To control a quadruped robot's leg smoothly, we employ differential kinematics, which provides the relation between joint angular velocities and linear velocity of the end-effector.

Given the motor angles $\boldsymbol{\theta} = [\theta_0, \theta_1, \theta_5]^T$ and their angular velocities $\dot{\boldsymbol{\theta}} = [\omega_0, \omega_1, \omega_5]^T$. The linear velocity of the end-effector $\mathbf{v} = [v_x, v_y, v_z]^T$ can be represented as following:

$$\mathbf{v} = \mathbf{J}(\boldsymbol{\theta}) \cdot \dot{\boldsymbol{\theta}}$$

Where $\mathbf{J}(\boldsymbol{\theta}) \in \mathbf{R}^{3\times3}$ is the Jacobian matrix. It is updated continuously as $\boldsymbol{\theta}$ changes during motion even if the given linear velocity remain the same. Note that angular velocity of the end-effector $(\omega_x, \omega_y, \omega_z)$ is ignored because the foot contact does not require rotational control in this context.

To find the inverse, the required motor angular velocities for a desired end-effector linear velocity:

$$\dot{\boldsymbol{\theta}} = \mathbf{J}^{+}(\boldsymbol{\theta}) \cdot \mathbf{v}$$

Where $\mathbf{J}^{+}$ is the Moore-Penrose pseudoinverse of $\mathbf{J}$.

Algorithm 3 Compute Joint Velocities from End-Effector Velocity

Require: End-effector velocity $\mathbf{v} = [v_x, v_y, v_z]^T$

1: Read current joint angles $\boldsymbol{\theta}$
2: Compute forward kinematics to get $\mathbf{p} = f(\boldsymbol{\theta})$
3: Compute numerical Jacobian $\mathbf{J} \in \mathbf{R}^{3\times3}$
4: Compute pseudoinverse $\mathbf{J}^{+}$
5: Compute $\dot{\boldsymbol{\theta}} = \mathbf{J}^{+} \cdot \mathbf{v}$
6: **return** $\dot{\boldsymbol{\theta}}$

Due to the difficulty in deriving $\mathbf{J}(\boldsymbol{\theta})$ analytically, we use the numerical finite difference method:

$$\mathbf{J}_{:,i} = \frac{f(\boldsymbol{\theta} + \delta \cdot \mathbf{e}_i) - f(\boldsymbol{\theta})}{\delta}$$

Where $f(\boldsymbol{\theta})$ is the forward kinematics function, δ is a small perturbation and $\mathbf{e}_i$ is a unit vector along the i-th motor axis.

5 Body Kinematics

To control the body pose of the robot when the legs are in contact with the ground, we take the positions of the end-effectors in the world frame and the orientation of the body as inputs. The goal is to transform the end-effector

positions from the world frame to the shoulder frame, which is necessary for calculating the inverse kinematics of the legs.

The world frame, $\boldsymbol{W}$, is the global reference frame in which the robot operates. The body frame, $\boldsymbol{B}$, is defined as the frame attached to the robot's center, and the shoulder frame, $\boldsymbol{S}$, is defined as the local frame attached to each leg's joint 1.

Each coordinate transformation is represented using a 4x4 homogeneous transformation matrix, including a rotation matrix $\mathbf{R} \in \mathbf{R}^{3\times3}$ and a translation vector $\mathbf{p} \in \mathbf{R}^3$:

$$T = \begin{bmatrix} R & p \\ 0 & 1 \end{bmatrix}$$

The steps to compute the foot positions in the shoulder frame are as follows:

Algorithm 4 Compute Foot Position in Shoulder Frame

Require: end-effector positions p_i^w, body orientation $(roll, pitch, yaw)$

1: Compute body-to-world transform, T_b
2: **for** each leg i **do**
3: Get shoulder-to-body transform $T_{s_i}^b$ (predefined by geometry)
4: Compute shoulder-to-world transform, $T_{s_i} = T_b \cdot T_{s_i}^b$
5: Compute world-to-shoulder transform, $T_w^{s_i} = T_{s_i}^{-1}$
6: Project world foot point into shoulder frame:

$$p_i^s = T_w^{s_i} p_i^w$$

7: **end for**
8: **return** Foot positions p_i^s in shoulder frames

6 Locomotion

Locomotion control is crucial for quadruped robots to enable them to move efficiently. The motion of the robot depends on the type of gait used, and different gaits have different characteristics such as step height, length, and timing.

The following first algorithm computes the phases of each leg based on time. Once the phase of each leg is determined, the next step is to calculate the end-effector positions for each leg. The second algorithm computes corresponding end-effector points. The position is calculated differently for the stance and swing phases of each gait cycle.

Algorithm 5 Locomotion Phase Calculation

Require: Gait type: `gait_type` ('walk', 'trot', 'pace', 'bound', or 'gallop')

1: Calculate the current phase of each leg based on the current time:

$$\text{cycle\_progress} = \frac{\text{time}\%\text{cycle\_time}}{\text{cycle\_time}}$$

2: **for** each leg in the robot's leg configuration **do**

3: Calculate phase based on cycle progress:

$$\text{phase\_dict}[\text{leg\_name}] = (\text{cycle\_progress} + \text{phase\_offset})\%1.0$$

4: **end for**

5: **return** `phase_dict`

Algorithm 6 End-Effector Points Calculation

Require: Current time: `time`

1: Get current phase for each leg using `get_current_phase(time)`

2: **for** each leg in the robot's leg configuration **do**

3: **if** phase of the leg is in the stance phase (i.e., less than `duty_factor`) **then**

4: Interpolate between the stance start and end points:

$$\begin{aligned}&\text{p\_start} = (start\_x, base\_y),\\ &\text{p\_end} = (start\_x + \text{direction} \times \text{step\_length}, base\_y)\\ &p = \text{p\_start} + (\text{p\_end} - \text{p\_start}) \times \frac{\text{phase}}{\text{duty\_factor}}\end{aligned}$$

5: Set `x, y` coordinates based on interpolation

6: **else**

7: Interpolate using cubic Bezier curve for swing phase:

$$\begin{aligned}&p_0 = (start\_x + direction \times \text{step\_length}, base\_y)\\ &p_1 = (start\_x + direction \times \text{step\_length}/2, base\_y + \text{step\_height})\\ &p_2 = (start\_x, base\_y + \text{step\_height}/2), \quad p_3 = (start\_x, base\_y)\\ &(x, y) = \text{cubic\_bezier}(t, p_0, p_1, p_2, p_3)\end{aligned}$$

8: **end if**

9: Set `ee_points[leg_name]` to the calculated coordinates

10: **end for**

11: **return** end-effector points: `ee_points`

Duty factor, the fraction of the cycle where the leg is in contact with the

ground.

The cubic Bezier curve is used for the swing phase to smoothly interpolate the leg's motion. The following formula is used to calculate the position on the cubic Bezier curve.

Algorithm 7 Cubic Bezier Curve

Require: Interpolation factor: `t`, Control points: `p0, p1, p2, p3`

1: Define the control points as a matrix:

$$\text{points} = \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{bmatrix}$$

2: Define the Bernstein coefficients for the cubic curve:

$$\text{coeffs} = \begin{bmatrix} (1-t)^3 & 3(1-t)^2 t & 3(1-t)t^2 & t^3 \end{bmatrix}$$

3: Calculate the position on the curve:

$$(x, y) = \text{coeffs} \times \text{points}$$

4: **return** position `(x, y)`

7 Simulation

After implementing the algorithms, we first test them in python notebook. We use matplotlib to visualize the robot's motion and check if the end-effector positions are correct.

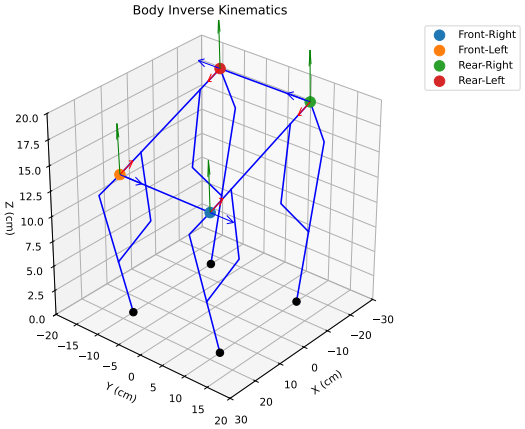


Figure 1: Visualized Body Inverse Kinematics

And for the differential kinematics, we compare the actual end-effector velocities with the desired velocities to ensure the control is accurate. As shown in the figure below, the vector of actual end-effector velocity fits well with the desired velocity.

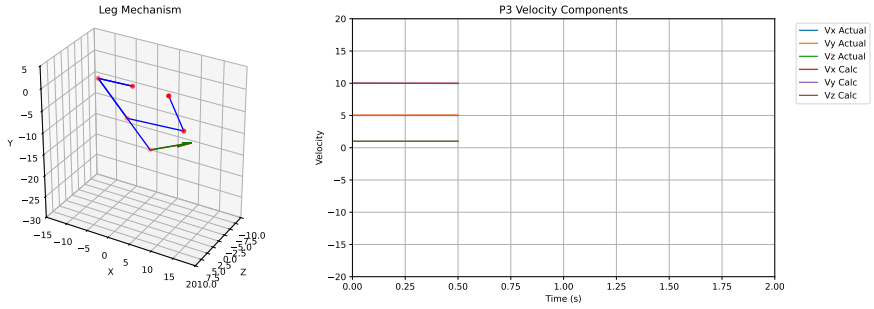


Figure 2: Visualized Differential Kinematics

Next, we move to the simulation in Pybullet. Pybullet is a physics engine that allows us to simulate the robot's motion in a realistic environment. We first use Blender and its extension, Phobos, to transform the solidwork model into a URDF file. Then we load the URDF file into Pybullet and set up the simulation environment.

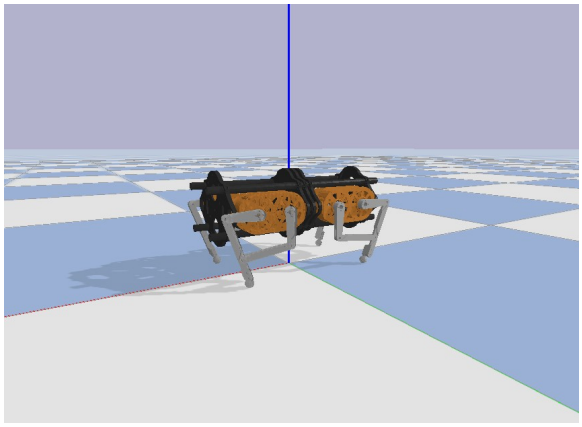


Figure 3: Simulation in Pybullet

However, Pybullet does not support the closed chain mechanism of the robot. Therefore, we need to cut the each leg into two parts: the upper leg and the lower leg. Then, in the simulation, we use the kinematic algorithm to control the position of all links directly to simulate the closed chain mechanism.

8 Deep Reinforcement Learning

To enhance Quadson's locomotion stability, we apply deep reinforcement learning (RL) to generate adaptive offsets for the end-effector positions. This section outlines the environment, RL formulation, and training process.

8.1 Environment Setup

We developed a custom Gymnasium environment, `QuadsonEnv`, using Pybullet for simulation. The environment includes:

- **Physics**: Gravity at $-9.81\,\mathrm{m/s^2}$, with a plane having friction (lateral: 0.8, spinning: 0.1, rolling: 0.01) and restitution (0.2).
- **Robot**: Quadson with 12 actuated joints, controlled via end-effector offsets.
- **Time Step**: $1/240\,\mathrm{s}$ for stable simulation.

8.2 RL Formulation

The RL problem is defined as follows:

- **Observation Space**: A 29-dimensional vector:

  - Body state: roll, pitch, yaw, linear velocity (v_x, v_y, v_z), angular velocity $(\omega_x, \omega_y, \omega_z)$ — 9 dimensions.
  - Joint state: Positions of 12 joints.
  - Leg phases: Sine and cosine of phases for four legs (LF, RF, LH, RH) — 8 dimensions.

- **Action Space**: A 12-dimensional vector of end-effector offsets $(x, y, z$ for each leg), bounded between -5 and 5 cm.

- **Reward Function**: Combines multiple terms:

$$\begin{aligned} r = & 1.0 \cdot \exp(-0.5(v_x - 0.5)^2) && \text{(forward velocity)} \\ & - 0.8 \cdot (0.5v_y^2 + 0.3\omega_y^2) && \text{(lateral penalty)} \\ & - 0.8 \cdot (0.5v_z^2 + 0.3\omega_z^2) && \text{(vertical penalty)} \\ & - 0.7 \cdot (1.7\text{roll}^2 + 1.5\text{pitch}^2 + 0.5\text{yaw}^2) && \text{(orientation penalty)} \\ & - 0.5 \cdot (\text{additional terms for stability and efficiency}). \end{aligned}$$

  Key goals include maintaining forward speed (0.5 m/s), height (0.2 m), and stability.

- **Termination**: Episode ends if roll or pitch exceeds 45°, height drops below 0.1 m, or 4000 steps are reached.

8.3 Training Process

We use the Proximal Policy Optimization (PPO) algorithm from Stable Baselines3 due to its stability and sample efficiency. Training occurs in Pybullet with:

- **Callback**: A custom `PlottingCallback` tracks rewards, computes moving averages, and plots progress every episode.

- **Visualization**: Real-time plots of episode rewards, moving averages, and standard deviations saved as `rl_training_progress.pdf`.

Hyperparameters (e.g., learning rate, batch size) are under optimization during training.

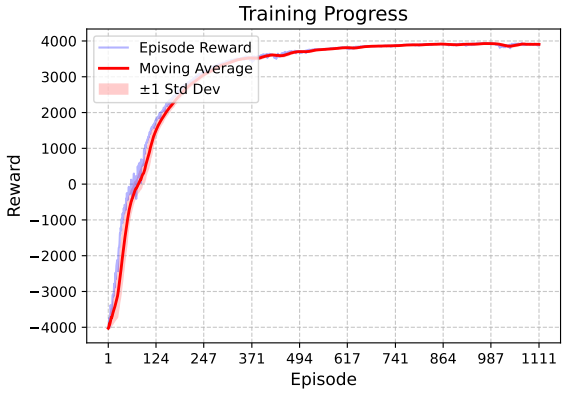
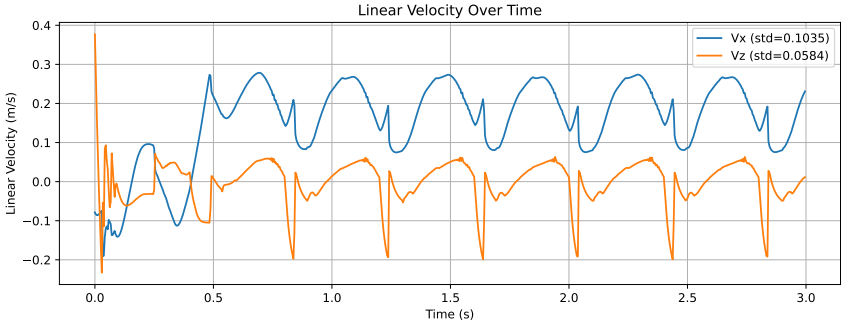


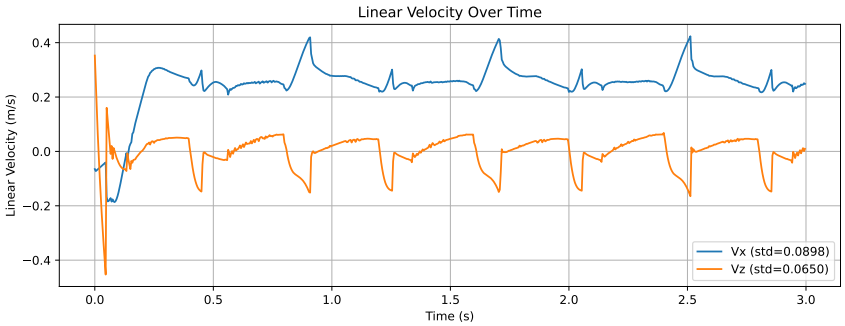
Figure 4: Reward

8.4 Preliminary Results

Training is ongoing. Initial observations suggest the agent learns to stabilize Quadson's gait. Final results will include average reward, stability metrics, and locomotion performance.

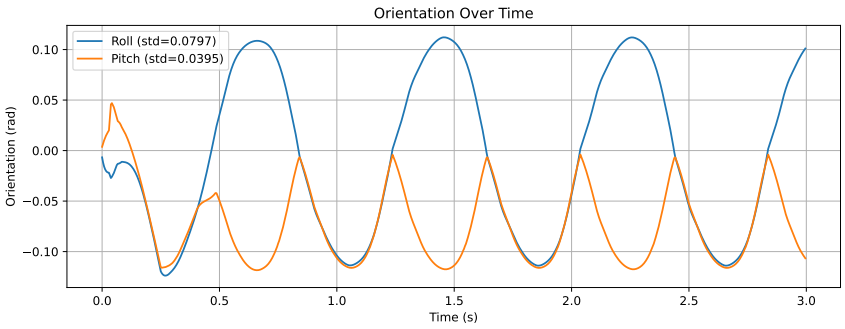


(a) Locomotion Alogrithm

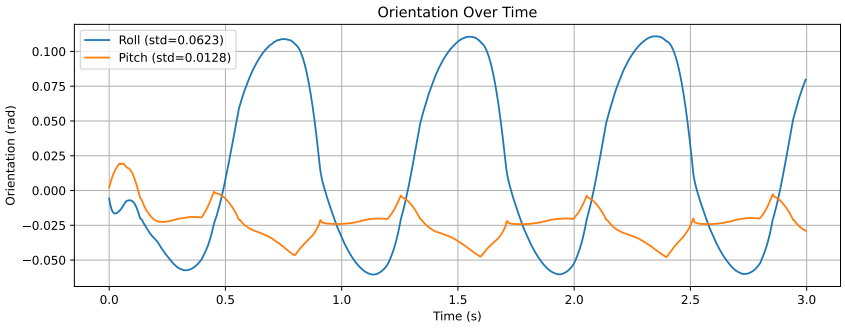


(b) RL Model

Figure 5: Velocity Comparision of Different Control Method Over a 3-Second Trial.



(a) Locomotion Alogrithm



(b) RL Model

Figure 6: Orientation Comparision of Different Control Method Over a 3-Second Trial.

9 Future Work

So far, we have implemented and valided kinematic algorithms, setting up a virtual environment for motion testing and reinforcement learning training. The RL model aims to improve locomotion stability with adaptive gait offsets. The next steps are:

1. **Simulation-to-Reality Transfer**: Deploy algorithms and RL model to the physical robot, deal with factors like friction and motor latency through tuning or domain randomization.

2. **Walking Stability**: Refine RL policy to handle the movement in different terrians, such as slopes or uneven surfaces.

3. **Sensor Integration**: Add IMU or LiDAR for real-time feedback and obstacle detection.

4. **Expanded Locomotion**: Enable movement in various directions other than forward, such as sideways.

5. **Recovery Mechanisms**: Enable self-righting or failure adaptation to protect the robot from falling.

These steps will transition Quadson from simulation to a functional real-world robot.

10 Conclusion

We developed a kinematic framework for Quadson, covering forward, inverse, differential, and body kinematics, alongside locomotion control. These algorithms were validated in Pybullet, where simulations confirmed accurate end-effector positioning and velocity tracking. To enhance stability, we initiated deep reinforcement learning (RL) with PPO to adjust gait offsets, with training ongoing. Despite Pybullet's limitation with closed-chain mechanisms, we adapted by controlling link positions directly. This work establishes Quadson as a platform for robotic motion studies, blending classical kinematics with RL. The next step is real-world deployment, building on this foundation to achieve practical functionality.