ES158 Lab: Upkie Pendulum Environment

Kevin Hongkai Yang

November 23, 2025

1 Environment Description

The Upkie pendulum environment is a Gymnasium wrapper that transforms the Upkie humanoid robot into a wheeled inverted pendulum system. This simplification enables focused study of balance control while abstracting away the complexity of full humanoid dynamics.

1.1 System Dynamics

The environment constrains the robot's legs to remain straight, effectively reducing the system to a single rigid body (the base) balanced on two wheels. The robot behaves as a wheeled inverted pendulum, where the control objective is to maintain an upright posture by commanding wheel velocities.

1.2 Observation Space

The observation space is a 4-dimensional continuous vector:

$$o = \begin{bmatrix} \theta \\ p \\ \dot{\theta} \\ \dot{p} \end{bmatrix} \in \mathbb{R}^4 \tag{1}$$

where:

- θ is the pitch angle of the base with respect to the world vertical, in radians. Positive values indicate forward lean. This is extracted from `base_orientation["pitch"]` in the spine observation.
- p is the position of the average wheel contact point, in meters. This is extracted from `wheel_odometry["position"]` in the spine observation.

- $\dot{\theta}$ is the body angular velocity of the base frame along its lateral axis (y-axis), in radians per second. This is the second component of `base_orientation["angular_velocity"]` in the spine observation.
- $\dot{p}$ is the velocity of the average wheel contact point, in meters per second. This is extracted from `wheel_odometry["velocity"]` in the spine observation.

The observation space is not normalized, and full spine observations remain available in the `info` dictionary returned by `reset()` and `step()`. The observation bounds are:

$$\theta \in [-\pi, +\pi] \text{ rad} \tag{2}$$
$$p \in (-\infty, +\infty) \text{ m} \tag{3}$$
$$\dot{\theta} \in [-1000, +1000] \text{ rad/s} \tag{4}$$
$$\dot{p} \in [-a_{\max}, +a_{\max}] \text{ m/s} \tag{5}$$

1.3 Action Space

The action space is a 1-dimensional continuous vector:

$$a = \left[\dot{p}^*\right] \in \mathbb{R} \tag{6}$$

where $\dot{p}^*$ is the commanded ground velocity in m/s. The action space is bounded by:

$$a \in [-a_{\max}, +a_{\max}] \tag{7}$$

where $a_{\max}$ is the maximum ground velocity (default: 1.0 m/s). This action is internally converted to wheel velocity commands using:

$$\omega_{\text{wheel}} = \frac{\dot{p}^*}{r_{\text{wheel}}} \tag{8}$$

where ω_{wheel} is the wheel angular velocity (rad/s) and r_{wheel} is the wheel radius from the robot configuration. A practical range for ground velocities is $[-1, 1]$ m/s, though the environment allows for higher values by adjusting $a_{\max}$.

1.4 Reward Function

The reward function is designed to encourage stable balancing behavior with smooth control actions. It penalizes deviations from the ideal balanced state

($\theta = 0$, $\dot{\theta} = 0$, $\dot{p} = 0$) and large or jerky control actions. The reward at each timestep is computed as:

$$r_t = 1.0 - \left(w_\theta \theta_t^2 + w_{\dot{\theta}} \dot{\theta}_t^2 + w_{\dot{p}} \dot{p}_t^2 + w_a |a_t|^2 + w_s |\Delta a_t|^2 + w_{\text{accel}} |\Delta^2 a_t|^2 \right) \quad (9)$$

where:

- θ_t is the pitch angle at timestep t (rad)
- $\dot{\theta}_t$ is the angular velocity at timestep t (rad/s)
- $\dot{p}_t$ is the linear velocity at timestep t (m/s)
- a_t is the action (commanded ground velocity) at timestep t (m/s)
- $|a_t|$ denotes the absolute value of the action
- $\Delta a_t = a_t - a_{t-1}$ is the first derivative of the action (action velocity), penalizing rapid changes in control commands
- $\Delta^2 a_t = \Delta a_t - \Delta a_{t-1}$ is the second derivative of the action (action acceleration), penalizing jerky control behavior

The reward weights are:

$$w_\theta = 2.0 \quad (10)$$
$$w_{\dot{\theta}} = 0.5 \quad (11)$$
$$w_{\dot{p}} = 0.5 \quad (12)$$
$$w_a = 1.0 \quad (13)$$
$$w_s = 2.0 \quad (14)$$
$$w_{\text{accel}} = 1.5 \quad (15)$$

The smoothness terms ($w_s |\Delta a_t|^2$ and $w_{\text{accel}} |\Delta^2 a_t|^2$) are critical additions that penalize rapid changes and jerky motion in the control actions. This encourages the policy to generate smooth, continuous control commands rather than aggressive, oscillatory behavior. The first derivative penalty (w_s) discourages large step changes in actions, while the second derivative penalty (w_{accel}) discourages acceleration in action changes, further smoothing the control profile.

Note that the reward function does not include a direct penalty on position p. Position drift is handled through termination conditions (Section 1.5), allowing the policy to focus on balance control while the termination condition prevents excessive wandering. This design choice centers

the learning objective on maintaining upright balance rather than position tracking.

During training, rewards are normalized using `VecNormalize` to stabilize value function learning and prevent value loss divergence. The normalized rewards are clipped to $[-10, 10]$ to prevent extreme values from destabilizing training.

When the robot is perfectly balanced with $\theta_t = 0$, $\dot{\theta}_t = 0$, $\dot{p}_t = 0$, $a_t = 0$, and no action changes ($\Delta a_t = 0$, $\Delta^2 a_t = 0$), the reward achieves its maximum value of $r_t = 1.0$. Deviations from this ideal state result in reduced rewards, with quadratic penalties ensuring that larger deviations are penalized more heavily.

1.5 Termination Conditions

The environment terminates an episode (sets terminated = True) under the following conditions, checked at each timestep t:

1. **Fall detection**: The episode terminates if the absolute pitch angle exceeds the fall threshold:

$$|\theta_t| > \theta_{\text{fall}} \tag{16}$$

   where $\theta_{\text{fall}} = 1.0$ rad by default. This condition indicates the robot has fallen over and cannot recover.

2. **Excessive position drift**: The episode terminates if the robot drifts too far from its initial position:

$$|p_t - p_0| > p_{\max} \tag{17}$$

   where p_t is the current position, p_0 is the initial position recorded at reset, and $p_{\max} = 5.0$ m. This prevents the robot from wandering indefinitely and encourages position maintenance while balancing.

3. **Excessive angular velocity**: The episode terminates if the angular velocity exceeds a safety threshold:

$$|\dot{\theta}_t| > \dot{\theta}_{\max} \tag{18}$$

   where $\dot{\theta}_{\max} = 10.0$ rad/s. This indicates the robot is spinning too rapidly to recover and prevents learning from unstable trajectories.

4. **Excessive linear velocity**: The episode terminates if the linear velocity exceeds a safety threshold:

$$|\dot{p}_t| > \dot{p}_{\max} \tag{19}$$

where $\dot{p}_{\max} = 2.0$ m/s. This indicates the robot is moving too fast to maintain control and prevents unsafe high-speed behavior.

Additionally, episodes are truncated (sets truncated = True) after a maximum number of steps:

$$t \geq t_{\max} \tag{20}$$

where $t_{\max} = 300$ steps. This time limit ensures reasonable episode lengths during training and prevents episodes from running indefinitely.

The termination conditions are checked in the following order: fall detection is checked first, followed by the additional termination conditions (position drift, angular velocity, linear velocity). If any condition is met, the episode terminates immediately. The time limit truncation is checked independently and can occur even if no termination condition is met.

1.6 Implementation Details

The environment is implemented as a Gymnasium wrapper around the base `UpkieEnv` class. The wrapper:

- Maintains leg joints in a neutral configuration using low-pass filtering
- Converts ground velocity commands to wheel velocity commands
- Extracts the simplified 4D observation from the full spine observations
- Computes rewards and checks termination conditions at each step

The environment integrates with the Upkie spine backend, which handles all physics simulation. This design allows the same environment code to work with both simulation and real robot hardware.

2 Policy Training

2.1 Training Approach

To learn a stabilizing policy for the Upkie pendulum, we use Proximal Policy Optimization (PPO) [1], a state-of-the-art on-policy reinforcement learning algorithm. PPO is well-suited for continuous control tasks and provides stable, sample-efficient learning.

2.1.1 Algorithm Configuration

The PPO algorithm is configured with the following hyperparameters:

- **Learning rate**: 2×10^{-4} (initial) with linear schedule decaying to 1×10^{-5} - adaptive learning rate that maintains higher values longer to prevent premature convergence
- **Batch size**: 128 - number of samples per gradient update
- **Number of steps**: 2048 total steps across all parallel environments - steps collected per policy update (distributed across parallel envs)
- **Number of epochs**: 5 - optimization epochs per update
- **Discount factor** (γ): 0.99 - long-term reward discounting
- **GAE lambda** (λ): 0.95 - generalized advantage estimation parameter
- **Clipping range**: 0.2 - PPO clipping parameter for policy updates
- **Value function clipping range**: 0.2 - additional clipping for value function to prevent divergence
- **Entropy coefficient**: 0.0 - no entropy bonus (deterministic policy preferred)
- **Value function coefficient**: 0.5 - weight for value function loss
- **Max gradient norm**: 0.5 - gradient clipping threshold

The policy network architecture consists of two fully-connected hidden layers with 64 units each, using ReLU activation functions. The network takes the 4-dimensional observation vector as input and outputs a 1-dimensional action (ground velocity command).

Observations and rewards are normalized using `VecNormalize` during training to stabilize learning. Observations are clipped to $[-10, 10]$ and rewards are normalized with the same discount factor ($\gamma = 0.99$) used for value estimation, then clipped to $[-10, 10]$. This normalization is critical for preventing value function divergence and stabilizing training.

2.1.2 Training Setup

Training is performed using the PyBullet simulation backend, which provides fast, accurate physics simulation without requiring a separate spine process. The training configuration includes:

- **Environment**: `Upkie-PyBullet-Pendulum` with frequency of 200 Hz
- **Parallel environments**: 10 - enables parallel data collection for faster training using `DummyVecEnv` (same process, multiple PyBullet connections)
- **Total timesteps**: 1,000,000 - sufficient for learning stable balancing
- **Checkpoint frequency**: Every 50,000 steps - periodic checkpoints for recovery and analysis
- **Evaluation**: Disabled during training to avoid robot deletion issues; evaluation can be performed manually after training using `rollout_policy.py`

During training, periodic checkpoints are saved every 50,000 steps, including model weights, optimizer state, and normalization statistics. The final model is saved with a timestamp upon training completion. Training progress is logged to TensorBoard for monitoring, including policy loss, value loss, and smoothed loss curves via a custom `LossCurveCallback`.

2.1.3 Training Process

The training process follows these steps:

1. Initialize the PPO agent with the specified hyperparameters and learning rate schedule
2. Create 10 parallel environments using `DummyVecEnv` (each with its own PyBullet connection)
3. Wrap environments with `VecNormalize` to normalize observations and rewards
4. Collect rollouts from 10 parallel environments (2048 total steps distributed across envs)
5. Compute advantages using Generalized Advantage Estimation (GAE)

6. Update the policy and value networks using PPO's clipped objective with value function clipping

7. Save periodic checkpoints every 50,000 steps (including normalization statistics)

8. Log training metrics (losses, returns) to TensorBoard via custom callbacks

The reward function designed in Section 1.4 provides a clear learning signal, encouraging the policy to minimize pitch angle, angular velocity, linear velocity, control effort, and action smoothness. The smoothness penalties (first and second derivatives of actions) are particularly important for generating stable, non-jerky control policies. The termination conditions ensure that the policy learns from safe, recoverable states while avoiding unstable trajectories. Reward normalization via `VecNormalize` stabilizes value function learning and prevents training instability.

2.2 Results

After training for 1,000,000 timesteps, the policy successfully learns to stabilize the Upkie robot in an upright position. The trained policy demonstrates the following characteristics:

- **Stability**: The policy maintains the robot's pitch angle close to zero, with small oscillations around the upright position

- **Position control**: The policy minimizes position drift, keeping the robot near its initial position

- **Smooth control**: The policy generates smooth, bounded control actions rather than aggressive jerky motions

- **Robustness**: The policy can recover from small perturbations and maintain balance

Evaluation metrics from testing the trained policy show:

- Mean episode return: Approximately 250-300 (depending on episode length)

- Mean episode length: Close to the 300-step maximum, indicating successful long-term balancing

- Low variance: Consistent performance across multiple evaluation episodes

Training progress is monitored through loss and KL divergence metrics, as shown in Figures 1 and 2. The loss curve (Figure 1) shows the total training loss decreasing over time, indicating successful learning. The loss is smoothed using a rolling average with a window size of 50 to better visualize the overall trend. The KL divergence curve (Figure 2) tracks the approximate KL divergence between the old and new policies during PPO updates, which helps monitor policy update stability. Both metrics demonstrate stable training convergence over the course of 1,000,000 timesteps.

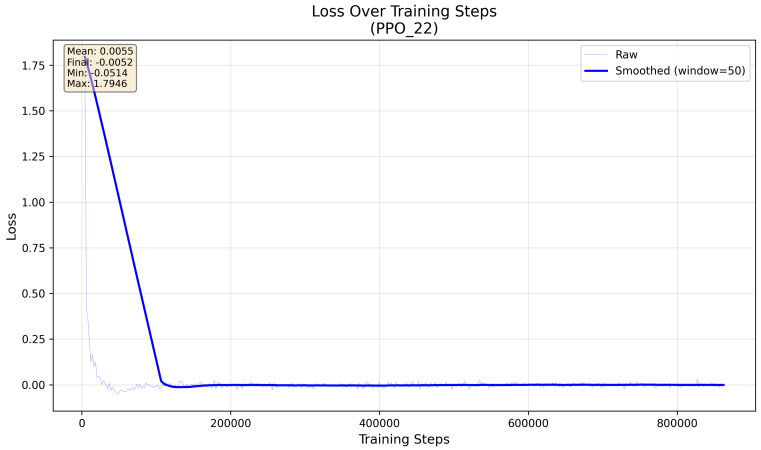

Figure 1: Training loss over time. The raw loss values (semi-transparent) and smoothed curve (window size 50) are shown. The loss decreases over training, indicating successful learning.

The policy successfully balances the robot by making small corrective wheel velocity adjustments based on the current pitch angle and angular velocity. When the robot leans forward (positive θ), the policy commands forward wheel motion to catch up with the center of mass. When the robot leans backward (negative θ), the policy commands backward wheel motion. This creates a stabilizing feedback loop that maintains the upright position.

2.3 Testing

The trained policy is tested using the `rollout_policy.py` script, which:

1. Loads the best saved model from training

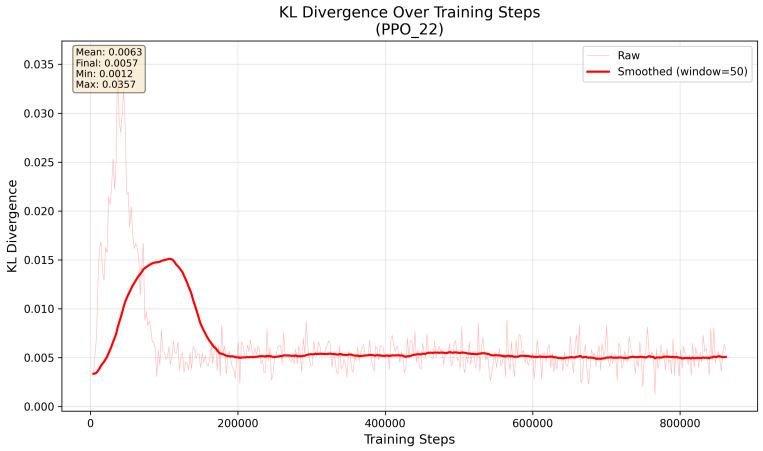

Figure 2: Approximate KL divergence between old and new policies during PPO updates. The raw values (semi-transparent) and smoothed curve (window size 50) are shown. Stable KL divergence indicates controlled policy updates.

2. Creates a test environment (PyBullet with GUI enabled for visualization)
3. Runs multiple episodes with the trained policy
4. Reports statistics including mean return, standard deviation, and episode lengths

The policy can be tested in both simulation (PyBullet) and on real hardware (Spine backend) by changing the environment ID in the rollout script. This demonstrates the transferability of the learned policy from simulation to reality.

3 Design Rationale

The reward function design balances several objectives:

- **Stability**: The quadratic penalties on θ, $\dot{\theta}$, and $\dot{p}$ encourage the robot to maintain an upright, stationary pose. The increased weight on pitch angle ($w_\theta = 2.0$) prioritizes balance maintenance.
- **Smooth control**: The action penalty ($w_a = 1.0$) and smoothness penalties ($w_s = 2.0$, $w_{\text{accel}} = 1.5$) work together to encourage the pol-

icy to use smooth, continuous control inputs rather than aggressive jerky motions. The smoothness terms penalize both rapid changes (first derivative) and acceleration in changes (second derivative), creating a strong incentive for smooth control profiles.

- **Position centering**: Position p is not directly penalized in the reward function, allowing the policy to focus on balance control. Position drift is handled through termination conditions, which prevent excessive wandering while not constraining the learning objective.
- **Learning signal**: The reward structure provides a clear gradient toward the desired behavior, with maximum reward at the ideal state. Reward normalization during training further stabilizes the learning process.

The termination conditions are designed to:

- **Prevent unsafe states**: Early termination when the robot falls or moves dangerously fast prevents learning from unstable trajectories.
- **Encourage focused learning**: By terminating episodes that drift too far, the policy learns to maintain position while balancing.
- **Enable efficient training**: Reasonable episode lengths (300 steps) allow for efficient exploration and learning.

This environment design enables reinforcement learning algorithms to learn effective balancing policies for the wheeled inverted pendulum system.

4 Task 3: Full Model Servos Control

4.1 Environment Overview

Task 3 extends the pendulum control problem to the full Upkie robot model with direct servo-level control. Unlike the simplified pendulum environment, this task requires controlling six servomotors directly: left hip, left knee, left wheel, right hip, right knee, and right wheel. This provides significantly more control authority and complexity, as the policy must coordinate all six joints to maintain balance.

4.2 Action Space

The action space for the Upkie-Servos environment is a dictionary with one key for each of the six servos. Each servo action is itself a dictionary containing:

- `position` θ^* (rad): Commanded joint angle, or `NaN` to disable the position loop
- `velocity` $\dot{\theta}^*$ (rad/s): Commanded joint velocity
- `feedforward_torque` τ_{ff} (N·m): Feedforward torque command
- `kp_scale` k_p^{scale}: Scaling factor for position feedback gain, $\in [0, 1]$
- `kd_scale` k_d^{scale}: Scaling factor for velocity feedback gain, $\in [0, 1]$
- `maximum_torque` $\tau_{\max}$ (N·m): Maximum torque limit

The servo applies torque according to:

$$\tau = \text{clamp}_{[-\tau_{\max}, \tau_{\max}]} \left(\tau_{\text{ff}} + k_p k_p^{\text{scale}} (\theta^* - \theta) + k_d k_d^{\text{scale}} (\dot{\theta}^* - \dot{\theta}) \right) \tag{21}$$

where k_p and k_d are fixed controller gains configured in the moteus controllers (running at ∼40 kHz, faster than the agent-spine loop).

For training with PPO, we use an action wrapper that converts a normalized Box action space to the servo dictionary format:

- **Wheels**: Velocity commands in $[-1, 1]$ are scaled to velocity limits, with position loop disabled (`NaN`)
- **Legs (hips/knees)**: Position commands in $[-1, 1]$ are mapped to joint position limits, with velocity set to zero

The action wrapper uses fixed gain scales: $k_p^{\text{wheel}} = 0.0$, $k_d^{\text{wheel}} = 1.7$, $k_p^{\text{leg}} = 2.0$, $k_d^{\text{leg}} = 1.7$. This design choice simplifies the action space while maintaining effective control authority.

4.3 Observation Space

The observation space is a dictionary with one key per servo, where each servo observation contains:

- `position` (rad): Current joint angle

- `velocity` (rad/s): Current joint velocity
- `torque` (N·m): Current joint torque
- `temperature` (°C): Servo temperature
- `voltage` (V): Power bus voltage

For training, we flatten this dictionary into a 12-dimensional vector (6 joints × 2: position + velocity), preserving the order established by the action wrapper. The full backend observation dictionary remains available in the `info` dictionary for custom reward shaping and termination logic.

4.4 Reward Function

The reward function for the servos environment is designed to encourage stable balancing while penalizing excessive control effort. The reward at each timestep is:

$$r_t = 1.0 - \left(w_\theta \theta_t^2 + w_{\dot{\theta}} \dot{\theta}_t^2 + w_{\dot{p}} \dot{p}_t^2 + w_a |a_t|^2 + w_s |\Delta a_t|^2 + w_{\text{accel}} |\Delta^2 a_t|^2 \right) \quad (22)$$

where:

- θ_t is the pitch angle at timestep t (rad)
- $\dot{\theta}_t$ is the pitch angular velocity at timestep t (rad/s)
- $\dot{p}_t$ is the linear velocity at timestep t (m/s)
- $|a_t|$ is the action magnitude (sum of absolute values of all servo commands)
- Δa_t is the change in action magnitude (first derivative)
- $\Delta^2 a_t$ is the change in action magnitude change (second derivative)

The reward weights are:

$$w_\theta = 0.5 \quad (23)$$
$$w_{\dot{\theta}} = 0.1 \quad (24)$$
$$w_{\dot{p}} = 0.1 \quad (25)$$
$$w_a = 0.05 \quad (26)$$
$$w_s = 0.01 \quad (27)$$
$$w_{\text{accel}} = 0.005 \quad (28)$$

The action magnitude is computed by summing the absolute values of velocity commands (for wheels) and position commands (for legs) across all servos. This provides a measure of overall control effort. The smoothness penalties (w_s and w_{accel}) encourage gradual changes in control effort, promoting stable and smooth control policies.

4.5 Termination Conditions

The termination conditions are identical to the pendulum environment (Section 1.5):

1. **Fall detection**: $|\theta_t| > 1.0$ rad
2. **Position drift**: $|p_t - p_0| > 5.0$ m
3. **Angular velocity**: $|\dot{\theta}_t| > 10.0$ rad/s
4. **Linear velocity**: $|\dot{p}_t| > 2.0$ m/s

Episodes are also truncated after 300 steps. These conditions ensure safe training and prevent learning from unstable trajectories.

4.6 Training Configuration

Training uses the same PPO algorithm as the pendulum task, with modifications for the increased complexity:

- **Environment**: `Upkie-PyBullet-Servos` with frequency of 200 Hz
- **Parallel environments**: 8 (reduced from 10 due to increased computational cost)
- **Total timesteps**: 2,000,000 (increased to accommodate higher complexity)
- **Network architecture**: Two hidden layers with 128 units each (increased from 64 to handle 12D observations)
- **Learning rate**: 2×10^{-4} (initial) with linear schedule to 1×10^{-5}
- **Batch size**: 128
- **Number of epochs**: 5
- **Other hyperparameters**: Same as pendulum task (Section 2)

The environment wrapper chain is:

1. Base `UpkieServos` environment
2. `ServosRewardWrapper`: Adds reward shaping and termination conditions
3. `ServoVelActionWrapper`: Converts Box actions to servo dictionary format
4. `ServoObsFlattenWrapper`: Flattens dictionary observations to Box
5. `TimeLimit`: Limits episode length to 300 steps

Observations and rewards are normalized using `VecNormalize` to stabilize training, with the same normalization parameters as the pendulum task.

4.7 Implementation Details

The reward wrapper extracts state information from the full spine observation dictionary:

- Pitch angle from `base_orientation["pitch"]`
- Pitch velocity from `base_orientation["angular_velocity"][1]` (y-component)
- Position and linear velocity from `wheel_odometry`

The action magnitude is computed by extracting velocity commands from wheel servos and position commands from leg servos, then summing their absolute values. This provides a unified measure of control effort across the different control modes (velocity for wheels, position for legs).

4.8 Results and Challenges

Training the full servos model presents several challenges compared to the simplified pendulum:

- **Increased dimensionality**: 12D observation space and 6D action space (after conversion) vs. 4D observation and 1D action for pendulum

- **Control coordination**: The policy must coordinate all six servos simultaneously, requiring more sophisticated control strategies
- **Training stability**: The higher-dimensional action space can lead to more unstable training, requiring careful hyperparameter tuning
- **Diverse solutions**: Unlike the pendulum task, the servos environment allows for diverse stabilization patterns (e.g., different leg configurations, wheel movements)

The training process demonstrates that the policy learns to improve over time, though achieving excellent performance can be challenging. The reward function successfully guides the policy toward stable balancing behavior, with the policy learning to coordinate servo commands to maintain upright posture.

Evaluation of trained policies shows that the robot can exhibit various stabilization strategies, including:

- Active leg movements to shift center of mass
- Coordinated wheel and leg motions for balance
- Different leg configurations for stability

This diversity in solutions is expected and demonstrates the flexibility of the full control model compared to the constrained pendulum system.

4.9 Design Rationale

The reward function design for the servos environment balances several considerations:

- **Balance priority**: The pitch angle penalty ($w_\theta = 0.5$) encourages upright posture, though with lower weight than the pendulum task to allow for more diverse control strategies
- **Control effort**: The action magnitude penalty ($w_a = 0.05$) is relatively small, allowing the policy to use the full control authority when needed while still discouraging excessive control effort
- **Smoothness**: The smoothness penalties are smaller than the pendulum task, recognizing that coordinated multi-joint control may require more dynamic movements

- **State extraction**: Using the full spine observation allows the reward function to access all relevant state information while the flattened observation provides a clean input to the policy network

The action wrapper design (velocity for wheels, position for legs) reflects the natural control modes of these joints: wheels are best controlled via velocity for balance, while legs benefit from position control for maintaining desired configurations.

5 Conclusion

We have successfully implemented a complete Gymnasium environment for the Upkie wheeled inverted pendulum, including a reward function that encourages stable balancing and appropriate termination conditions. Using PPO, we trained a policy that successfully stabilizes the robot in an upright position. The policy demonstrates robust balancing behavior with smooth control actions, validating the effectiveness of the environment design and training approach.

The modular design of the environment allows for easy experimentation with different reward functions, termination conditions, and training algorithms. Future work could explore curriculum learning, domain randomization, or transfer learning from simulation to real hardware.

References

[1] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.