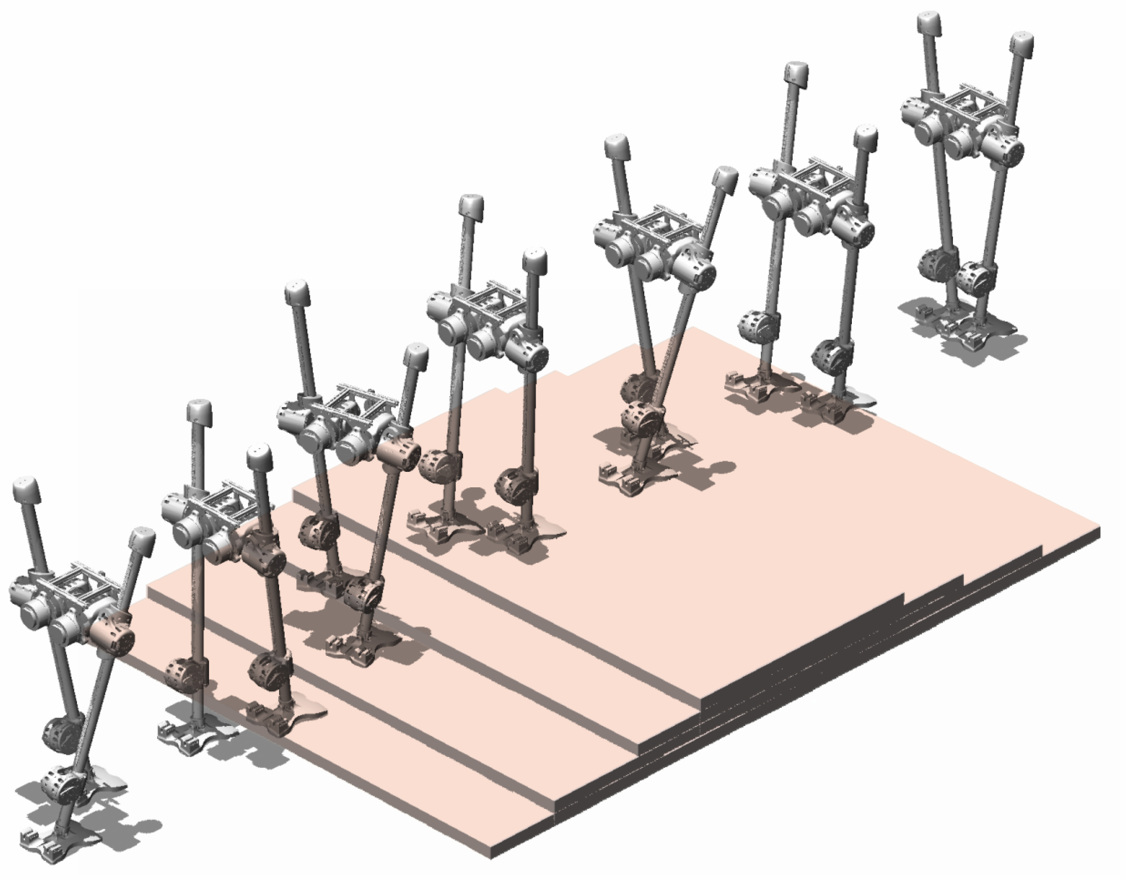
The Full Software Stack for the Sim-To-Real Transfer of a Learned Bipedal Walking Policy



Thomas Godden
Dyson School of Design Engineering
Imperial College London
June 2023

Abstract

This report details the control development for the sim-to-real transfer of a reinforcement learning bipedal walking controller. A joint controller and state estimator are written to facilitate realtime control of the physical hardware. A simulated robot model, parameter estimator, and reward function are made to enable an RL algorithm to learn how to generate and stabilise a walking gait. The policy trained in simulation is then transferred onto the real hardware. The results show that the learned controller has generalized enough to be able to stabilize the physical system despite modeling discrepancies and sensor noise.

Glossary

RL - Reinforcement Learning
PPO - Proximal Policy Optimization
Hybrid System - A system that exhibits both continuous and discrete behavior
Sim-to-Real - The transfer of a controller learned in simulation to real hardware
ROS - Robot Operating System
CAN bus - Controller Area Network communication bus
IMU - Inertial Measurement Unit
MCU - Microcontroller Unit
Pose - Position and orientation of an object
Proprioceptive - The sense of joint position, velocity and force
BLDC - Brushless DC motor
Policy - A specific control architecture and set of corresponding parameters

Contents

1 Introduction

Bipedal waking poses a unique control challenge: it is a high-dof, control-through-contact, hybrid system with real-time control constraints. In particular, the timing constraints differentiate it from manipulation where each state is generally statically stable. This means it is a good problem to stress test control strategies on. If an approach can stabilize a bipedal robot, it implies it will likely work well on other similar problems. This project will discuss writing a full stack walking controller for a bipedal robot that uses Reinforcement Learning (RL) to generate a dynamic gait capable of untethered walking.

We will start with a low level actuator interface and build up the software components required to perform joint control and full state estimation. Then a model of the robot will be generated and ported to a simulator. A reinforcement learning agent will be trained to control the simulated version of the robot and finally the learned policy will be transferred back onto the real hardware. Performance will be evaluated both in simulation and on the real hardware. The robot we are using is detailed in [32], named SLIDER for it's unconventional prismatic joints that replace the traditional knee joint.

Approaches to bipedal walking control

Walking has been studied by biomechanics researches far before engineers had the tools to create robots capable of reproducing legged locomotion [19]. Early theories such as ZMP (zero moment point) control appeared as early as 1969 [31]. These experiments and resulting models attempted to explain the energy shift along the walking cycle by tracking the point along the ground where the tipping moment due inertia and gravity equals zero. This can be thought of as the pivot point of the robot and shifts forwards under the feet as the robot walks. This was exploited heavily by the Honda Asimo robots [12], which displayed remarkable stability for slow movements but lacked true "agility".

In the late 1980s the Leg Lab started creating hopping robots, these differed significantly from others by exploring dynamic stability resulting from a simple spring-like motion instead of complex walking gait generation. They showed that a similar control scheme could control running for 1,2 and 4 legged robots [22]. Later, the Linear Inverted Pendulum LIP and the Spring Loaded Linear Inverted Pendulum model was formalised [14] and proved to be able to generate natural-like gaits. In these models, the dynamics of a walker are approximated as a point mass on either a rigid or spring leg, thus creating an inverted pendulum. In addition to the Raibert hoppers, the passive dynamic walkers studied by Tad McGeer in 1990 [18] fed into the shift of thinking towards working with the inherent dynamics of a system rather than trying to override them. These machines used the energy gained by a slope to generate a stable and convincing walking gait with no active control.

In the past 15 years, computational speed has given rise to the ability to perform online trajectory optimisation, otherwise known as model predictive control (MPC). This control scheme uses a simplified model of the dynamics to simulate trajectories into the future and pick the best control inputs to minimise some cost function. This approaches have generally been brittle on systems that cannot be modeled well, as errors compound over the trajectory rollout. However, recent approaches on quadrupeds and bipeds show that incredible performance can be achieved using a combination of offline trajectory optimisation, and online MPC [1] [17].

With the advent of learning algorithms and their success in many digital domains, people have been attempting to use learned policies for robot control. The two major routes are imitation learning and reinforcement learning.

Imitation learning attempts to create a control policy by tracking a set of expert demonstrations. With a large enough example dataset, interpolation is often achieved where the agent is able to perform under scenarios it has never seen. One of the largest benefits of imitation learning is that trajectories can be recorded on real hardware by human demonstrators, encoding far more nuance than a reward function. Another source for expert data are offline trajectory optimisation algorithms that might be too computationally intensive to run online, but can be "distilled" down into a neural network policy.

Reinforcement learning provides an agent with a set of observations and a reward per timestamp, then tasks the optimiser with tuning a control policy to maximise the agent's expected reward over it's "lifetime". These algorithms have worked phenomenally in controlling video or board games, where the agent can be trained on the real game environment. These have surpassed most human players at even complex games like Go [28] and StarCraft [30]. RL for real time robot control is still in it's early phases. Most algorithms generally require tens to hundreds of millions of timesteps to train on a complex system which can equate to hundreds or thousands of hours of real world equivalent training time. The data hurdle is often circumvented by the use of simulated enviroments to gather vast amounts of data, however the transfer from a policy trained in simulation to the real hardware (reffered to as sim-to-real) is still an open problem. Despite these challenges, a number of promising results have come out in the past 5 years. One of the best results of sim-to-real policy transfer on a legged robot has been the quadraped controller developed by ETH Zurich on their AnyMal platform [26]. The subsequent policies are likely some of the most robust legged locomotion controllers to come out of a research lab [17]. These successes have also started to be realised on bipedal robots; the current world record holder for the fastest 100m run by a robot is held by an RL controller from Oregon State [5].

Motivation

While a final year masters project is nowhere near long enough to fully explore the sim-to-real problem space, we will be taking a breadth first approach. Trying to build the full pipeline from scratch will quickly highlight the issues with current techniques and motivate further exploration. It is also a great way to be exposed to a bunch of tangential problems in robotics, control, state estimation and learning methods.

2 Control Design

Since our goal is to perform a sim-to-real transfer of a controller, a significant portion of the project centered on the implementation of a joint controller and state estimator. This section describes the control scheme, which takes in desired joint goals and sends corresponding command packets to the motors. The next section describes state estimation which provides a live estimate of the full state of the robot. An early decision was made to run the inbuilt impedance control (joint level proportional and derivative control) on each of the robot's actuators and use desired joint targets as the action space for the RL algorithm.

2.1 Architecture

Instead of writing a single or multi-threaded application to control all functions of the robot, ROS 2 Humble [23] was used to connect various processes together into a network of nodes. Each node fulfills a discrete task (or set of related tasks) and generally can be built and tested in isolation. ROS was picked due to it's

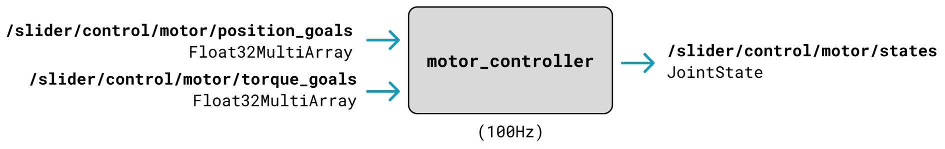
- Ease of use
- Reliability in a prototyping enviroment
- Support for mixing C++ and Python nodes
- Wide range of "off the shelf" packages and message types
- 3D transformation support [24]
- Visualization options[1]
- Inherent networking ability

A basic ROS network is made up of a set of nodes and topics. Each node functions as a single program that can produce or consume messages. Each topic is a channel through which messages are passed. Nodes can publish or subscribe to messages on any available topic. Once inside a node, a message can be unpacked and used for whatever purpose, however ROS defines a series of default message types and topics that aid the standardization node communication. It should be noted ROS provides far greater functionality than just basic message passing.

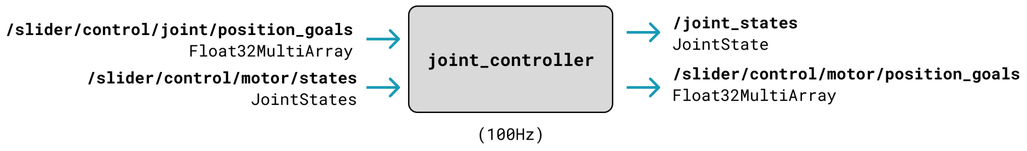
[1]Foxglove [9] was particularly handy

To begin, a rough node layout was decided on and overall robot control tasks were split up between nodes. The network evolved as follows.

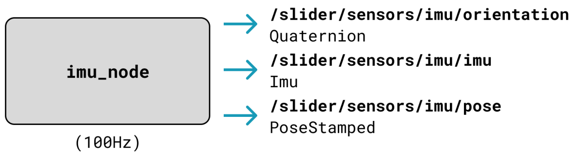
The hardware interface with the robot are two USB-to-CAN modules that interface with a CAN bus on each leg. Frames sent on this bus represent joint targets, control constants and feedback variables. Due to the complexity of handling the bus communications and the need to multithread parts of this program, this was given a full node. This also isolates the motor control from failures in other nodes and allows a set of safety features to be run that "sanitize" inputs into this node.


/slider/control/motor/position_goals
Float32MultiArray
/slider/control/motor/torque_goals
Float32MultiArray
motor_controller
(100Hz)
/slider/control/motor/states
JointState

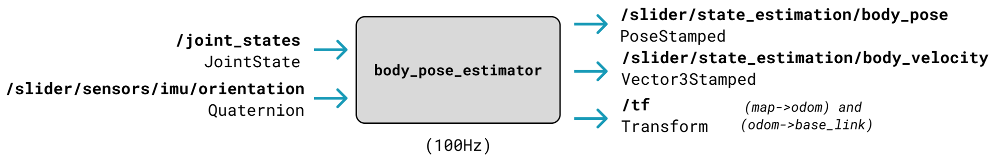
Moving a step up, the motors do not match 1:1 to the kinematic joints on the robot. Therefore a joint controller node was needed that subscribes to joint targets, and publishes motor targets. Vice-versa it subscribes to motor states and publishes joint states.


/slider/control/joint/position_goals
Float32MultiArray
/slider/control/motor/states
JointStates
joint_controller
(100Hz)
/joint_states
JointState
/slider/control/motor/position_goals
Float32MultiArray

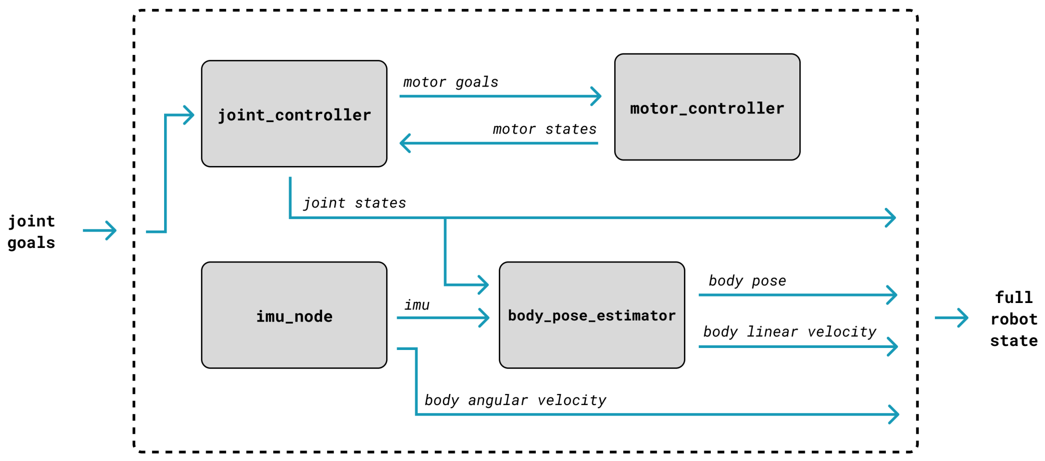
To generate the full state of the robot we must calculate the pose and velocity of the torso. To measure orientation, an IMU hardware interface node was written to receive data from an AHRS (Attitude and Heading Reference System) algorithm running on a Raspberry Pi Pico Microcontroller which in turn, polls the IMU sensor.



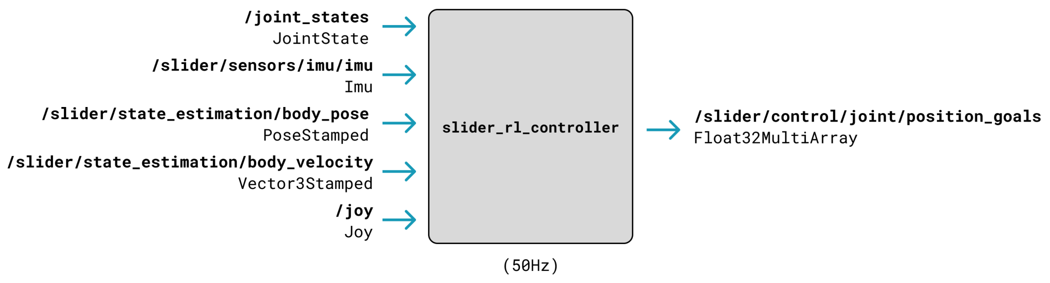
As we do not have a direct way to observe the absolute position or velocity of the robot, we must combine other measurements to estimate them. A proprioceptive state estimator was written to calculate robot pose based on joint angles and base orientation. This subscribes to the joint states and IMU topics and publishes an estimated base link pose and velocity to a robot pose topic.



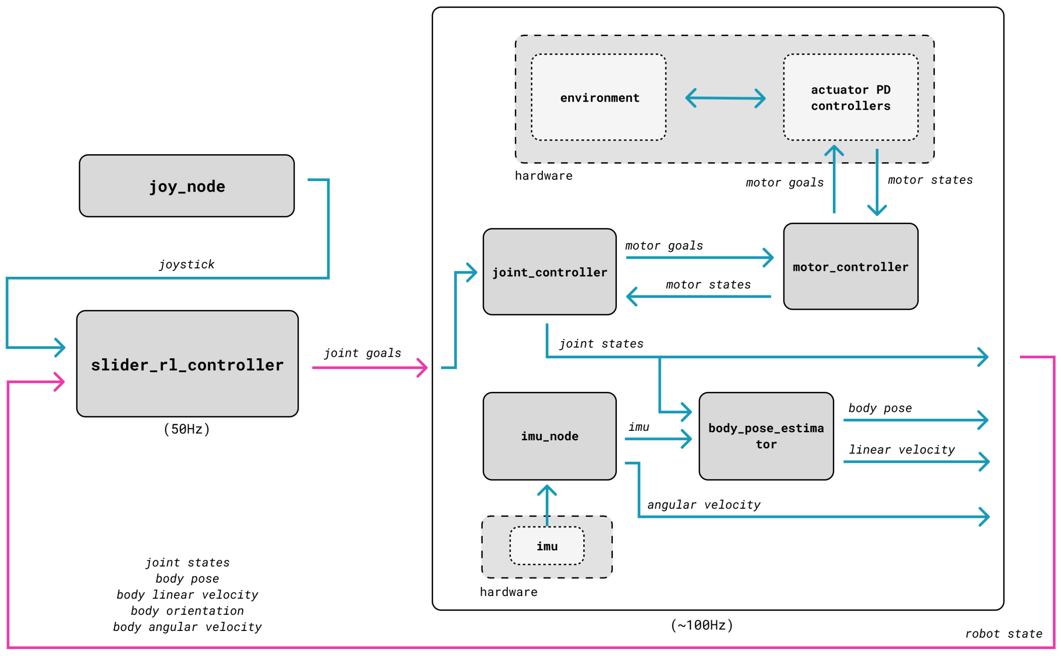
This set of nodes gives us our full system plant that subscribes to desired joint targets and publishes full state.



To round out, we have the RL policy node that closes the loop by subscribing to the full robot state and publishing desired joint angles. It also subscribes to a joystick topic.



The full node graph is shown below, the classical feedback loop path is highlighted in pink. When training the RL policy, we can abstract the entire "plant" block and replace it with a simulated environment.



2.2 Motor Control

The motors used on SLIDER are several models from CubeMars's AK Series of BLDC servos [7]. These are scaled up commercial clones of the infamous Mini-Cheetah actuator designed by Ben Katz [15]. A diagram of the robot layout with the types of actuators is shown in Fig. 1.

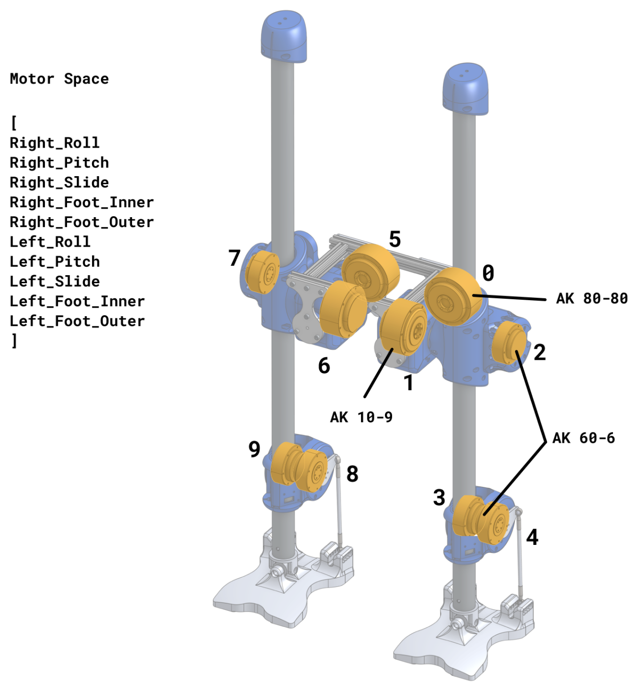
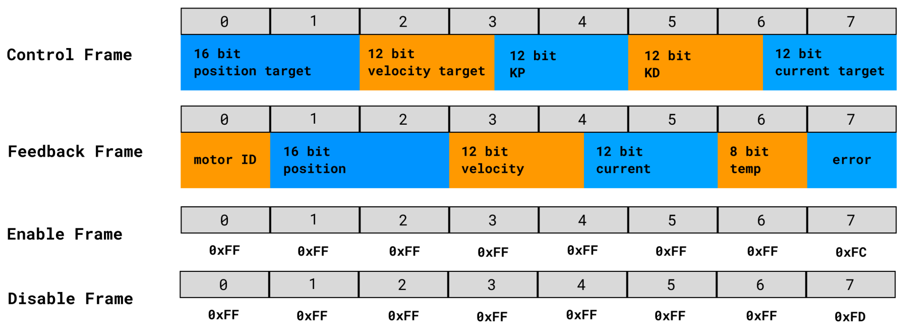


Figure 1: The motor layout of the SLIDER robot

These actuators communicate over CAN bus using several command and telemetry packets. These are split into control, feedback, enable and disable messages, as well as a zero encoder command that was not used in this project. A diagram of the byte layout is shown in Fig 2.



	0	1	2	3	4	5	6	7
Control Frame	16 bit position target		12 bit velocity target	12 bit KP		12 bit KD	12 bit current target	
Feedback Frame	motor ID	16 bit position		12 bit velocity	12 bit current		8 bit temp	error
Enable Frame	0xFF	0xFF	0xFF	0xFF	0xFF	0xFF	0xFF	0xFC
Disable Frame	0xFF	0xFF	0xFF	0xFF	0xFF	0xFF	0xFF	0xFD

Figure 2: The data packing of the motor CAN frames

The control side of the `slider_joint_control` ROS package holds all code required to interface with the actuators on the robot. This takes the form of a series of classes with increasing abstraction, `canbus`, `TMotor`, `MotorManager` and the `motor_controller` node itself.

The `canbus` class abstracts out bus parameters and allows for multiple motor objects to share the same bus parameters, helpful in a multi-bus architecture. Linux socketcan is used to interface with the CAN controllers.

The `TMotor` class handles the final command frame packing, feedback frame unpacking and low level can communication through a provided `canbus` class. Each motor type has it's own subclass that contains motor-specific constants such as torque and velocity limits.

A `MotorManager` class was written to orchestrate all motors connected a single bus. This class handles starting bus communications, enabling, and disabling all connected motors as well as updating motor variables based on telemetry messages. The SLIDER hardware has one CAN bus per leg, so two `MotorManagers` are required to run the full robot. Each of these has a separate thread that continuously reads incoming messages from the bus and routes the position, velocity and torque data into the appropriate motor object, based on the CAN packet's ID. These incorporate a mutex that is locked whenever messages are being received to prevent any consumer from reading out motor variables while they are being updated.

Finally the main `motor_controller` ROS node combines two `MotorManagers` with a third control thread and a series of callbacks to send position, velocity or torque commands. On startup, it sets each motor's constants, position limits and zero offsets. It includes three main protection features:

- Control watchdog timer. This disables all motors if the time since last received position or torque goal is greater than the specified amount. This protects against an external node sending a goal then crashing, leaving the robot stuck.
- Position discontinuity protection. If the received goal is too far away from a current motor position, throw an error and disable motors.
- Joint limits. Max and min joint limits are set per joint and enforced on the motor level. This simply clips the output and does not currently throw an error.

2.3 Joint Control

Out of the robot's 10 joints pictured in Fig 3, only four of them correspond directly to a motor; the hip pitch and roll. The two slide joints are put through a pulley and belt transmission that maps the motor's angle to leg linear displacement. On each foot, the two ankle motors are mapped to foot roll and pitch through a parallel linkage.

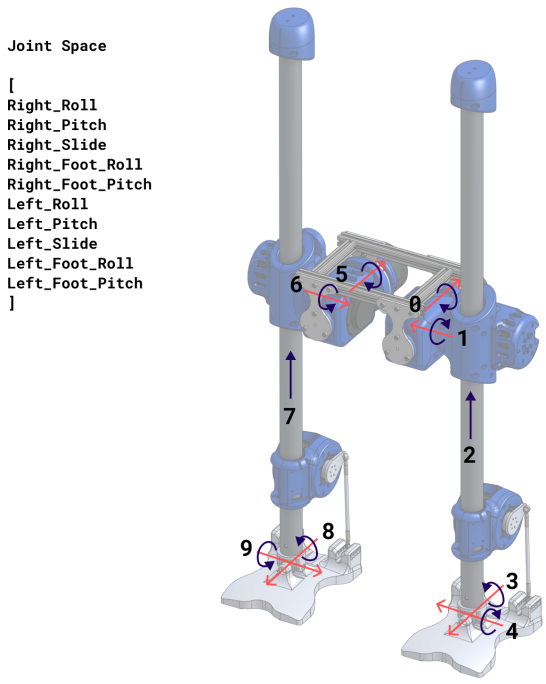


Figure 3: Joints and joint space

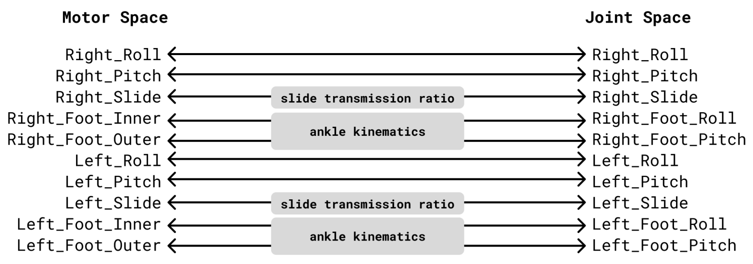


Figure 4: Mapping between motor and joint space

To handle this, a C++ node was written that converts between motor and joint space. In joint space, the `joint_control` node subscribes to target joint angles and publishes joint states. In motor space it publishes target motor angles and subscribes to motor states. This is done through topic callbacks, so happens live at the rate the messages are received to minimize delay. In addition to computation time, there is a delay introduced by having to unpack and repack a message to be sent through the ROS middleware, however this is on the order of tens to hundreds of nanoseconds [16] and therefore far lower than our control period (10 milliseconds).

The mapping between slide motor and joint states is as follows, where $\boldsymbol{q}$ represents joint positions and $\boldsymbol{\theta}$ represents motor angles.

$$\boldsymbol{q}_{\text{slide}} = \text{slide\_transmission\_ratio} \cdot \boldsymbol{\theta}_{\text{slide\_motor}}$$

$$\dot{\boldsymbol{q}}_{\text{slide}} = \text{slide\_transmission\_ratio} \cdot \dot{\boldsymbol{\theta}}_{\text{slide\_motor}}$$

$$\boldsymbol{F}_{\text{slide}} = \boldsymbol{\tau}_{\text{slide\_motor}} / \text{slide\_transmission\_ratio}$$

The following section will discuss the mapping between motor and joint space for the ankle.

2.4 Ankle Kinematics

Each foot's pitch and roll is controlled by two motors through a parallel linkage made up of two lever arms coupled by control rods. All ankle kinematics are done in a separate C++ class which provides functions that are included in the `joint_control` node.

The linkage diagram and notation is shown in Fig 5. This linkage type has been previously used in robot ankle designs and the inverse kinematics are well studied. The equation to go from joint angles to motor angles was implemented directly from [33]. A previous project had implemented a neural net based solver [11] however mechanical changes meant that it had become out of date and was inaccurate.

$$\begin{aligned}
F'_x &= \cos(\theta_{\text{pitch}}) \cdot Fo_x - \sin(\theta_{\text{roll}}) \cdot Fo_z \\
F'_y &= \sin(\theta_{\text{pitch}}) \cdot \sin(\theta_{\text{roll}}) \cdot Fo_x + \cos(\theta_{\text{pitch}}) \cdot Fo_y + \cos(\theta_{\text{roll}}) \cdot \sin(\theta_{\text{pitch}}) \cdot Fo_z \\
F'_z &= \sin(\theta_{\text{pitch}}) \cdot \sin(\theta_{\text{roll}}) \cdot Fo_x - \sin(\theta_{\text{pitch}}) \cdot Fo_y + \cos(\theta_{\text{pitch}}) \cdot \cos(\theta_{\text{roll}}) \cdot Fo_z
\end{aligned}$$

$$
\begin{aligned}
a &= F'_x - M_x \\
b &= M_z - F'_z \\
c &= ((l^2_{\text{Rod}}) - (l^2_{\text{Bar}}) - ||F' - M||^2)/(2 \cdot l_{\text{Bar}})
\end{aligned}
$$

$$\psi = \arcsin\left(bc + \sqrt{b^2c^2 - (a^2 + b^2) \cdot (c^2 - a^2)}/(a^2 + b^2)\right)$$

where

- ψ is motor angle
- θ is foot angle
- M is the motor point relative to the pivot point
- Fo is the foot link connection point relative to the foot pivot point when the foot is level
- F' is the foot link connection point relative to the foot pivot point after transformation.

The forward kinematics were not directly described, but can be calculated efficiently using an iterative solver. The psudocode for this solver is shown in Alg. 1 Over a number of iterations, the approximate motor values will converge on the true values required to generate the desired pitch and roll. The iteration number, as well as the gradient approximations were found through trial and error. If greater performance is required, an approximate to the local derivative could be found using higher order methods or the solver could be warm-started with the previous solution. However, in it's current state, the method converges in the order of microseconds on a laptop processor.

Algorithm 1 Ankle Forward Kinematics

```
roll $\leftarrow$ 0.0
pitch $\leftarrow$ 0.0

left_motor $\leftarrow$ measure_left_motor()
right_motor $\leftarrow$ measure_right_motor()

max_iterations $\leftarrow$ 10
for $i \leq$ max_iterations do
    left_motor_approx, right_motor_approx $\leftarrow$ ankleIK(pitch, roll)

    left_motor_error $\leftarrow$ left_motor - left_motor_approx
    right_motor_error $\leftarrow$ right_motor - right_motor_approx

    roll $\leftarrow$ roll $+ 0.2 \cdot$ left_motor_error $+ 0.2 \cdot$ right_motor_error
    pitch $\leftarrow$ pitch - $0.4 \cdot$ left_motor_error $+ 0.4 \cdot$ right_motor_error
end for
```

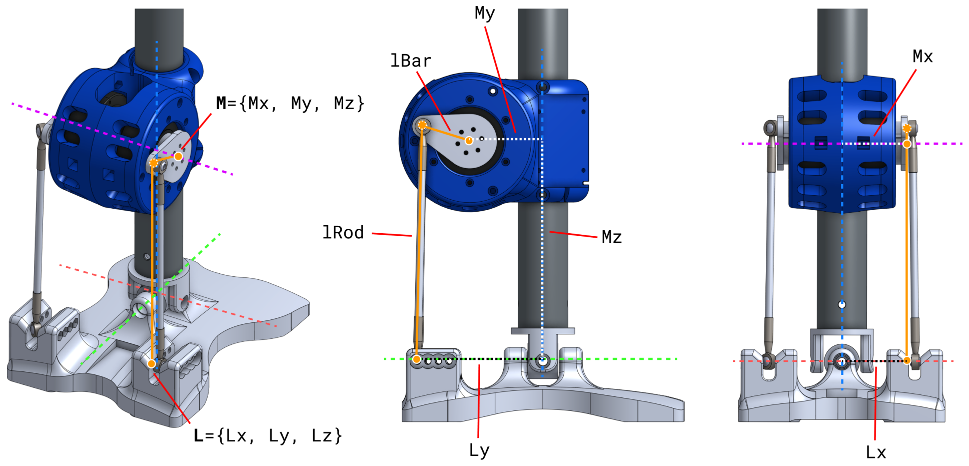


Figure 5: Diagram of the ankle linkage

Due to work done in previous projects, concerns were brought up about the speed of such an algorithm, however this solver was benchmarked and outperforms the previous neural net implementation by over 10×. Due to the network size, it is feasible that less floating point operations are being done with the iterative solver implementation, however the largest performance gain is likely from removing the python library that handles network inference. This method also allows for parameter changes and refinements without having to re-train a network.

This was tested on the robot using a level and the reported ankle angles were within 0.5 degrees of actual for both forward and inverse. This error likely stems from mechanical backlash, inaccurate measurements and mis-zeroed motors.

3 State Estimation

A fundamental assumption most RL algorithms make is that the system they are controlling is markovian; the next state must depend only on the current state and input actions. In dynamic systems, for this to be true, you need full observability, so we will need estimate the full state of the robot. You can get around this by providing a state history, which we also do, but providing full state is ideal. This means we need to measure all joint positions and velocities in addition to the torso pose and velocity. Our full state vector is

$$\boldsymbol{x} = [\boldsymbol{p}, \boldsymbol{Q}, \boldsymbol{q}, \dot{\boldsymbol{p}}, \boldsymbol{\omega}, \dot{\boldsymbol{q}}]$$

where $\boldsymbol{p}$ is torso position represented in Cartesian space, $\boldsymbol{Q}$ is torso orientation represented as a quaternion, $\boldsymbol{q}$ are joint angles, $\dot{\boldsymbol{p}}$ is torso velocity, $\boldsymbol{\omega}$ is torso angular velocity and $\dot{\boldsymbol{q}}$ are joint velocities. Our joint angles and velocities are already provided by our motor and joint control nodes, so remaining is orientation, angular velocity, position and linear velocity estimation for the torso. Previous projects on the robot have worked towards generating a visual SLAM based state estimator for the robot [3], however the particular camera was no longer installed and implementation code was in an unknown state.

3.1 Orientation Estimation

Part of estimating full robot state is base link orientation, in this case, the torso. To measure this, a LSM6DSOX 6dof IMU breakout from Adafruit [2] was installed on the robot base. This provides angular velocity and linear acceleration measurements from 12 to 6.66 KHz. 208 Hz was chosen from the list of available sampling rates as it is the slowest above our desired poll rate of 100 Hz. Picking a higher rate would result in more noise and not aid our control policy as that is targeted to run at 50Hz. To fuse these two measurement sources into an absolute pose a quaternion implementation of a standard complementary filter was used.

A minimal C library containing quaternion and vector operations was written to abstract the algorithm and aid in debugging. This library contained structs for Vector and Quaternion, and a standard set of quaternion operations such as addition, multiplication, inverse, lerp and some basic vector operations such as dot and cross.

The filtering algorithm works as follows. Initially, the local angular velocity and gravity vector are polled from the IMU. We define our IMU frame as A and the world frame as W, we also initialise a world heading estimate q, represented as a 4 element quaternion.

$$\omega_A = [\text{IMU Measured Angular Velocity}, 0]$$
$$a_A = [\text{IMU Measured Acceleration}.0]$$

$$q_W = [1, 0, 0, 0]$$

The local angular velocity, represented in the body frame, is converted into a global angular velocity, represented by the world frame, by rotating it by the inverse of the IMU heading estimate.

$$\omega_W = q_W^{-1} \omega_A q_W$$

This is then used to calculate a quaternion derivative of the heading (note that the derivative of a quaternion is not the same as a angular velocity, but they are tightly related)

$$\dot{q_W} = 0.5 \cdot \omega_w q_W.$$

Gravity is converted into an orientation by calculating the rotation required to move the up vector to the gravity vector, this is done using the standard shortest arc formula

$$\vec{\text{up}} = [0, 0, 1]$$
$$g_W = [\vec{\text{up}} \times a_A, |\vec{\text{up}}| \cdot |a_A| + \vec{\text{up}} \cdot a_A].$$

Finally these components can be used to implement a complementary filter. This filter is of a prediction-correction type, where a smooth relative measurement is combined with a noisy absolute measurement to gain a measurement that is both smooth and free of drift. The next orientation is predicted by integrating the quaternion derivative forward by our loop period and summing it to the previous estimate

$$q_W' = q_W + \dot{q_W} \cdot \text{dt}.$$

This is then combined with the absolute but noisy gravity measurement by lerping between the two using the filter constant c, in this case 0.99

$$q_{W corrected}' = q_W' \cdot c + g_W \cdot (1 - c).$$

This results in an estimate that converges to the absolute orientation over time, preventing drift, but filters out noise from the acceleration measurements. In this implementation, by not rotating the gravity quaternion by the current yaw value, we force the yaw estimate to always converge to zero. This was chosen due to not using an absolute yaw reference, such as a magnetometer. While the IMU contains one, it was deemed unnecessary as initial walking trials are short and the robot has no direct control over yaw.

Once the absolute orientation estimate is computed, the MCU prints them out over a virtual serial line at 115200 bit/s. The message ordering is shown below, values are rounded to three decimals places and each line is terminated with a line break

$$[\texttt{quat.w, quat.x, quat.y, quat.z,}$$
$$\texttt{gyro.x, gyro.y, gyro.z, acc.x, acc.y, acc.x}]$$

A ROS node then parses the incoming byte stream and publishes the resulting orientation, angular velocity and acceleration as an `Imu` message type [25]. Additionally, IMU pose (`PoseStamped`) and orientation (`Quaternion`) topics are published to make visualization easier.

3.2 Proprioceptive Pose Estimation

The challenge of object pose estimation is one far older than the robotics field. Any vehicle, manned or otherwise would ideally like to know where it is at all times. Good methods now exist for low rate, high noise absolute positioning, such as GPS, but the need for local, high bandwidth, high accuracy relative positioning still exists. These local, relative, fast methods are often referred to as odometry. This section will describe the creation of a properoceptive (in the sense that it uses limb position) relative pose estimation algorithm.

With our goal of self-contained walking, we could not use a system that relied on external senors, such as a motion tracking setup. So we have to use entirely internal sensors. A plan was devised to use the foot pose to back-calculate a transform from a locked foot point to the base link. When the next foot makes contact with the ground it is considered "locked" and the base link transform is now calculated from it. The foot in contact is simply assumed to be the foot with the lowest point in world space. This method makes several assumptions

- The body orientation is known
- All joint angles are known
- At least one foot is placed on the ground
- Feet do not slip once they have touched the ground

- The ground is perfectly flat

With the use of foot force estimation, loadcell or otherwise, the flat ground assumption can be removed and with proper inertial filtering on the body pose (such as an Extended Kalman Filter), the slip and foot contact assumptions can be partially removed as well.

We will define X as the body frame, L as the left foot frame R as the right foot frame and C as the contact point frame, which is initialised to a position of $[0, 0, 0]$. The algorithm starts out by setting all joint angles in our kinematic model to be equal to the measured joint angles on the robot. Additionally it sets the body orientation to the measured orientation reported by the IMU fusion algorithm. Given our contact point in world space ${}^{W}p^{C}$, our rotation from robot base to world space ${}^{W}R^{X}$ and our foot point in robot base space ${}^{X}p^{L}_{X}$, we can calculate the position of the robot base (the center of the torso in this case) in world space. This example is specifically for left foot support

$$ {}^{W}p^{C} - {}^{W}R^{X}{}^{X}p^{L}_{X} = {}^{W}p^{L} - {}^{X}p^{L}_{W} = {}^{W}p^{L} + {}^{L}p^{X}_{W} = {}^{W}p^{X}_{W}. $$

At the timestep where contact is swapped between feet, the new fixed contact point is calculated based on the world to foot translation of the newly contacting foot. Given our previous contact point in world space ${}^{W}p^{C}$, our body pose estimate in world space ${}^{W}p^{X}_{W}$, both feet points in robot space ${}^{X}p^{L}_{X}$, ${}^{X}p^{L}_{X}$ and the rotation from robot to world space ${}^{W}R^{X}$, the next contact point is calculated as follows. This particular example is for left to right transition where the previous contact point is assumed to equal the the previous world position of the left foot ${}^{W}p^{L} = {}^{W}p^{C}$.

$$ \begin{aligned} {}^{W}p^{C'} &= {}^{W}p^{C} + {}^{W}R^{X}({}^{X}p^{R}_{X} - {}^{X}p^{L}_{X}) \\ &= {}^{W}p^{C} + {}^{W}R^{X}({}^{L}p^{R}_{X}) \\ &= {}^{W}p^{C} + {}^{L}p^{R}_{W} \\ &= {}^{W}p^{L} + {}^{L}p^{R}_{W} = {}^{W}p^{R} \end{aligned} $$

The body pose is now calculated based on the inverse transform from the new estimated contact point until the next transition occurs, this is shown in Fig. 6. Due to this being a discrete algorithm, the transition will always happen after the foot goes lower than the floor, so the new contact point is shifted up so that the z-coordinate is 0. This is also following from our assumption of a perfectly flat floor. If foot sensors were incorporated and the crude foot contact heuristic replaced, this system could estimate elevation changes from stairs, ramps or other vertical deviations.

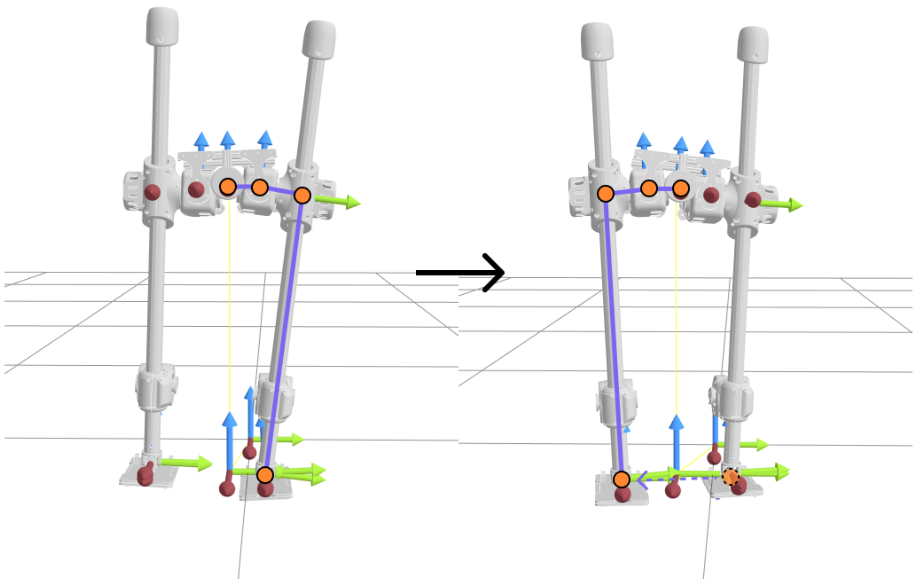


Figure 6: The transform tree change when a new contact point is made

To calculate body velocity, a 1st order derivative approximation is made using the past, current pose and the time delta between them.

4 Reinforcement Learning

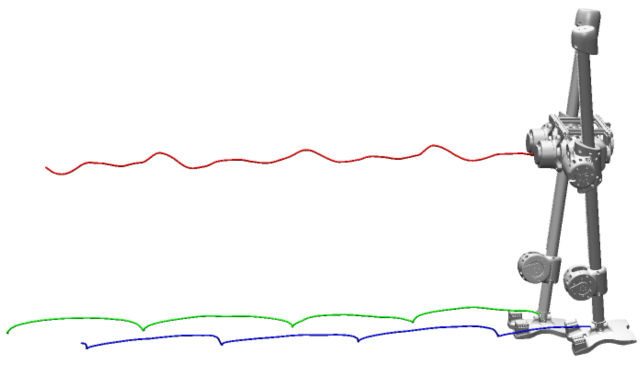


Figure 7: Trajectory of a learned walking gait

4.1 Modeling and Simulation Enviroment Setup

For a good sim-to-real transfer, an accurate simulated robot model must be created. An existing model of SLIDER was available in URDF format, however it became apparent that it's inertial properties were decently inaccurate. A simple mass check revealed that it was about half of the real robot's mass and the corresponding link inertias were also similarly different.

Parameter estimation

A new model was created using a combination of the original SLIDER CAD and measured weights on the robot, however there were some parameters that could not be directly measured or estimated from the CAD model, such as joint friction and effective actuator gains. To combat this, several trajectories were recorded on the robot and ran through a stochastic multi-parameter estimator. The operating principle of the estimator works as described in Alg. 2. The algorithm is initialised with a set of initial parameters, a parameter range, a set of measured data from the robot and a desired noise scale (somewhat equivalent to a learning rate). The scale can be lowered through the fitting run if convergence is reached. As it runs, it randomly perturbs the current parameters until a lower error parameter set is found, then swaps those into the current parameters. A set of example plots from the tool estimating stiffness, damping and inertia for both hip pitch and roll are presented in Fig. 8.

Algorithm 2 Stchocastic Multi-Parameter Estimator

```
current parameters $\leftarrow$ initial parameters
parameter range $\leftarrow$ parameter max - parameter min
best cost $\leftarrow$ Inf
for $i \leq$ number of iterations do
    nudge $\leftarrow$ parameter range $\cdot$ RandomGaussian() $\cdot$ scale

    trial parameters $\leftarrow$ current parameters $+$ nudge
    trial parameters $\leftarrow$ clamp(trial parameters, parameter max, parameter min)

    simulated trajectory $\leftarrow$ SimulateRollout(trial parameters)

    cost $\leftarrow$ MeanSquaredError(simulated trajectory, measured data)

    if cost $<$ best cost then
        current parameters $\leftarrow$ trial parameters
        best cost $\leftarrow$ cost
    end if
end for
```

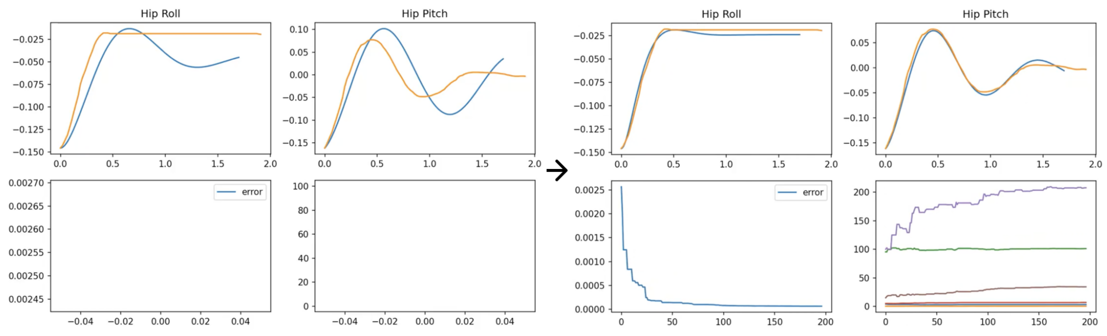


Figure 8: Initial vs Final state of the parameter estimation tool

Domain randomization

Domain randomization was performed by adding a small amount of per trial random offset and per timestep random noise to both on the action and observation vectors. This helps prevent the RL equivalent of "overfitting" and aids generalization.

Simulator

In addition to modeling the robot, a simulation framework must be chosen. For conciseness, a full review of simulators will be omitted here, but [6] contains a comprehensive summary.

MuJoCo [20] was picked due to its speed, focus on accurate contact modeling, open source code base and operating system independence. The previous SLIDER simulation in Gazebo could only run at a fraction of real time, whereas the new MuJoCo simulation runs at roughly 30x real time during training. This can be optimized further, but proved fast enough to iterate on.

The RL algorithms within our chosen learning framework interface with the environment using a standardized set of function calls, the most common of these is the Gym library from OpenAI (now Gymnasium maintained by the Farma Foundation) [10]. Gym was picked due to its low overhead, simplicity and ability to plug into the desired reinforcement learning package.

4.2 Learning Framework and Reward Function

Stable Baselines3 [29] was chosen as our learning framework due to the ability to trial different algorithms, out of the box support for gym, tensorboard support and good codebase and documentation. PPO [27] was picked due to it's support of continuous action spaces and historical ability to handle control-through-contact problems well.

At its core, reinforcement learning is an optimisation algorithm: it attempts to pick the ideal set of policy net parameters that maximize the expected reward from rollouts. PPO is a policy gradient method, which attempts to estimate the expected reward gradient for a given state and shift the policy towards higher reward. While explaining the full technical description of PPO is far outside the scope of this report[2], a short intuitive explanation will be presented. The high level update rule looks like

$$\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t)$$

where θ is our policy parameters, α is a learning rate and $\nabla J(\theta_t)$ is the derivative of expected reward over a full trajectory relative to policy parameters. This is in essence the classic gradient ascent method. While it would be nice to have an oracle that spat out the true derivative, that is infeasable, so approximations need to be made. One of the largest challenges is knowing how big of an update step to make given a policy gradient, a "low quality" gradient estimate can result in parameters being shot off into a local minima or worse. PPO attempts to combat this problem by constraining the policy updates to a range based on the difference in probabilities of taking an action in the current vs the new policy. If that ratio is high in either direction, it clips the update to prevent an action being naively

[2]The explination is large enough to be a full master's project on it's own [4]

greedily taken or being forgotten about due to an uninformative set of data. A nice technical explanation can be found in [21].

As with all optimal control methods, both learned and classical, they are highly sensitive to their reward functions. Learned methods are arguably more sensitive due to their ability to probe a more complex environment (such as a full physics simulator instead of a simplified model). This can lead to what is known as reward hacking, where unintended behavior can be used to gain large reward by exploiting various environmental features.

Initially to test the learning pipeline, a simplified model of SLIDER was created, and all joints were locked down except the slides. A jumping reward was provided where the per step reward was the square of the torso z position. With some tweaking, this produced an "enthusiastic", but solid jumping policy that proved the ability to provide a reward and use the simulated environment to learn the control actions required to maximize that reward.

To incrementally add complexity, more joints were unlocked and more complicated reward functions were generated, notably,

- Jumping in place (minimizing lateral drift)
- Running with unlocked hip pitch
- Running with unlocked hip pitch and roll
- Running with hip pitch+roll and foot pitch+roll

Once a running gait had been generated on the simplified model, it was time to work on generating a walking gait. In this context the difference between walking and running is considered to be the time the robot is in contact with the ground. When walking, the robot always maintains contact, this results in a different dynamics where you act less like a bouncing spring and more like a falling pendulum. Running is often easier to control than walking.

To encourage the robot to walk instead of shuffling, a cost term was added that heavily penalized sliding a foot while it was in contact. To give the robot a sense of time a set of sinusoids were provided as input to the model. This has a biological parallel in the form of central pattern generators [13] [8]. CPGs are naturally occurring neural oscillators and have been hypothesised to aid in locomotion and other periodic tasks. These inputs are always locked to a multiple of the desired stepping gait

$$\cos(t \cdot 2\pi/(\text{Desired gait period})).$$

Interestingly enough, the model almost always latched onto this input even if a gait timing wasn't directly enforced. Letting the model learn to generate a swing and stance gait is fairly unique as nearly all other learned bipedal policies explicitly enforce a contact schedule. In future iterations of this controller, the model should be given the ability to adjust it's clock timings to increase or decrease stepping frequency.

Once the simplified model was walking and the reward function structure was validated, the full URDF was imported and tweaked. An additional set of reward terms were introduced to minimize energy, control the robots heading, and minimize torso accelerations and angular velocity. The final observation space included three timesteps of full state history, the reference velocity and the sinusoid clock input.

4.3 Final Reward Function

Body Velocity

The main task cost component is a penalty on squared error between reference and measured velocity components. The term is

$$C_{vel} = 5.0 \cdot (v_{refx} - v_{bodyx})^2 + 2.0 \cdot (v_{refy} - v_{bodyy})^2$$

where v_{ref} is the desired velocity.

Power

$$c_{hip} = 1.0 \quad c_{slide} = 0.1 \quad c_{ankle} = 1.0$$

$$C_{jointfactors} = \left[c_{hip}, c_{hip}, c_{slide}, c_{ankle}, c_{ankle}, c_{hip}, c_{hip}, c_{slide}, c_{ankle}, c_{ankle}\right]$$

$$P = \boldsymbol{F}_{joint}\dot{\boldsymbol{q}}^T$$

$$C_{power} = P^T C_{jointfactors} P$$

where $\boldsymbol{F}$ is generalized force, $\dot{\boldsymbol{q}}$ is generalized velocity and $\boldsymbol{C}$ is a set of adjustment terms that allow certain joints to be weighted higher.

This cost provides a nice "grounding" effect on the controller, with physically unreasonable or implausible gaits being generally rejected. Power is also a better measure than just force, as a lot of natural gaits require high force events. With the initial force cost, the robot would often choose to lower down to the slide joint limit and sit there, so the slide was no longer supporting the weight of the robot.

Foot Slip

One of the largest issues with the initial network was it's tendency to "shuffle". To combat this, a penalty was placed on a foot having both force and velocity. The cost terms are

$$C_{Lfootslip} = |[v_{Lfootx}, v_{Lfooty}]| \cdot F_{Lfoot}$$

$$C_{Rfootslip} = |[v_{Rfootx}, v_{Rfooty}]| \cdot F_{Rfoot}$$

$$\text{Foot Slip Cost} = C_{footslip} = C_{Lfootslip} + C_{Rfootslip}.$$

Body Orientation

To keep the body frame aligned with the desired orientation, cost components penalized deviation of both the upwards and forwards axes

$$C_{bodyup} = |[u_x, u_y]|$$

$$C_{bodyforward} = |[f_y, f_z]|$$

where u is a vector aligned with the vertical ("up") axis of the robot body frame, and f is a vector aligned with the forward axis of the body frame.

Body Movement

A desired characteristic is the smoothness of the torso trajectory during the walk cycle. This aids mechanical longevity, especially for any onboard hardware and sensors and generally results in more nautral feeling gaits.

$$C_{body\ angular\ velocity} = |\omega_{body}|$$

$$C_{body\ acceleration} = |a_{body} - g|$$

where ω_{body} is the angular velocity of the torso, a_{body} is the linear acceleration of the torso and g is gravity.

Fall

If the robot falls (the torso z position is less than 0.4 meters), a fall cost is applied and the episode is reset

$$C_{fall} = \begin{cases} 1, & \text{if } (p_{body\ z}) < 0.4 \\ 0, & \text{otherwise.} \end{cases}$$

Full Cost Function
The full cost function and relative weights are as follows, note that cost terms are not pre-normalised so specific weights should be taken with a grain of salt

$$\begin{aligned}C = &C_{vel} + 0.015 \cdot C_{slip} + 0.5 \cdot 10^{-5} \cdot C_{power}\\ &+ 2.5 \cdot (C_{body\ up}) + 0.2 \cdot (C_{body\ forward})\\ &+ 0.02 \cdot (C_{body\ angular\ velocity}) + 0.01 \cdot (C_{body\ acceleration})\\ &+ 200 \cdot C_{fall}.\end{aligned}$$

Reward Function
To generate a reward function from a cost function, we sum the negated cost with a constant offset to result in positive reward. Purely negative rewards do not play well with RL algorithms which train on expected reward. This generally causes them to seek early termination, since the highest expected reward is gained by accumulating the least cost and each timestep incurrs cost. Interestingly, the choice of constant to sum makes a noticable difference in the training process, with too high of a constant resulting in the robot preffering to statically stand. The is likely because the risk of ending the episode early by falling outweights the potential gain in reward by learning to walk. A "good" constant is one where standing results in negative reward, but good velocity tracking results in positive reward. The final reward function is

$$R = 2.0 - C.$$

4.4 Learning Curriculum

During initial testing, it was found that the RL policy struggled to learn how to track omnidirectional gates well. This often took the form of never beginning to walk if a randomized reference velocity was given in the base policy. One possible explanation for this is that the average velocity was (0.0, 0.0) and before a full gait had developed, standing still provided the best expected reward for the randomized target. Thus, it would get stuck in a local minima and struggle to escape. Making this issue worse was the fact that standing still is generally favorable to the other terms within the reward function (body pose and accelerations, effort, foot sliding). A learning curriculum was designed to combat this, shown in Fig. 9.

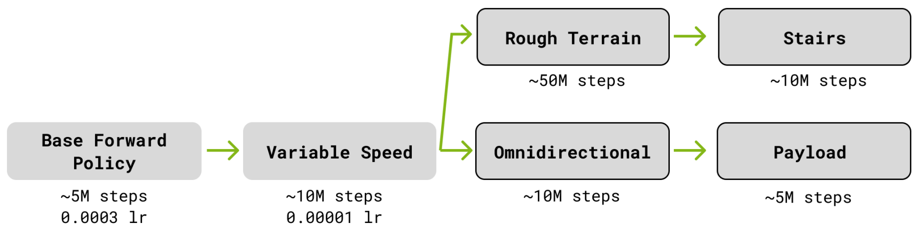


Figure 9: An example learning curriculum

5 Validation

To validate the learned policy in isolation, a series of simulated tests were performed. For impulse rejection, a normal walking model was used that had been trained on a flat ground environment with periodic perturbations. For slope and stairs, a climbing model was created that trained on an increasing series of slopes and stairs. With extra perception it would be possible to swap between these models to pick the most adequate one for the task, or with a more sophisticated policy network setup including memory, the robot could adapt on the fly.

To validate the sim-to-real transfer of the policy, a set of walking trials were performed on hardware. Some "hacks" had to be performed to get the policy running well, such as scaling all outputs down by roughly 0.75, as the joint PD controllers appeared to go unstable when the policy was run at full actuation level. This was likely due to control rate and other delays impacting full system stability. Interestingly enough, once this was done, the policy was surprisingly robust, if a bit "hesitant". A comprehensive analysis on the source of these oscillations should be performed, but it was not able to be done on the timeline of this project.

5.1 Simulated Impulse Rejection

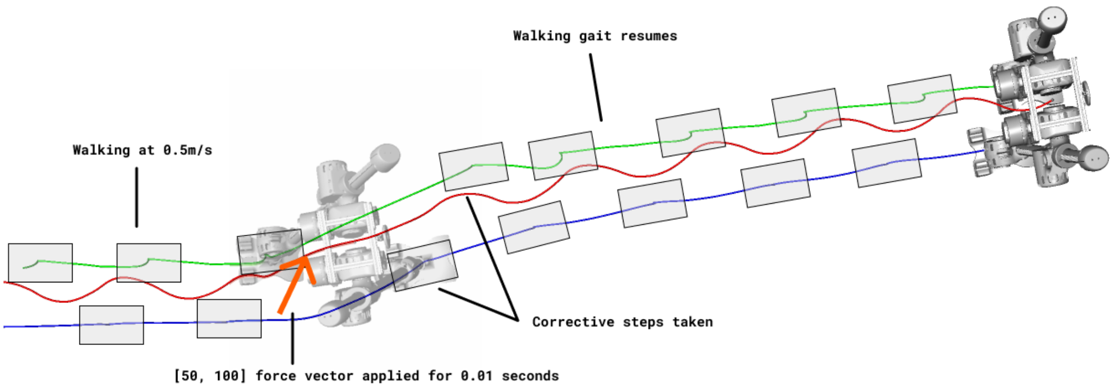


Figure 10: Diagram showing the policy recovering from a perturbation.

To validate the policy's ability to reject impulse force perturbations, a grid of force vectors was created and applied for several trials. The model was given 0.5 seconds to start walking at 0.5m/s and then the force was applied randomly between the 0.5

and 2.5 seconds into the trial, finally, an extra 4.5 seconds was given for the model to fall or recover. Presented in Fig. 11 are isolated X and Y impulses, whereas Fig. 12 shows sucsess over the full vector field.

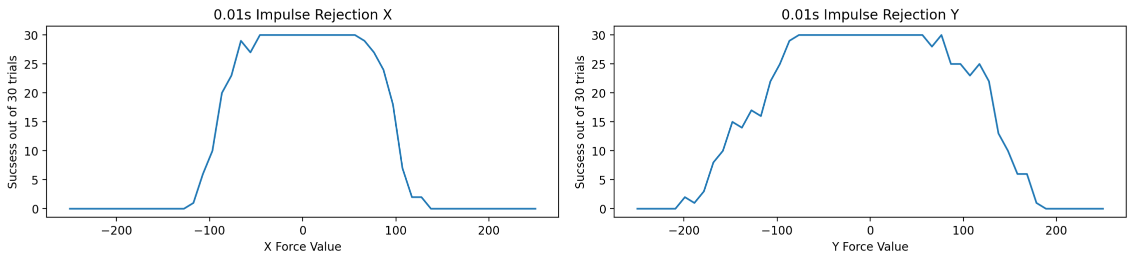


Figure 11: Impulse rejection results for pure X and Y impulses.

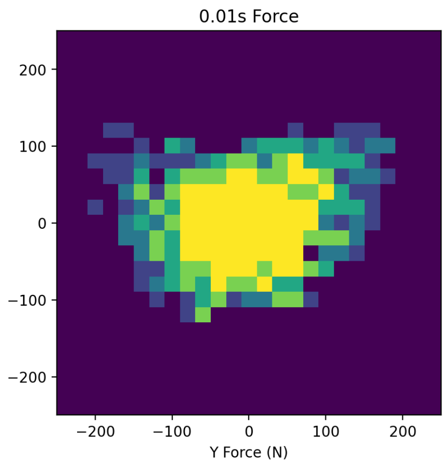
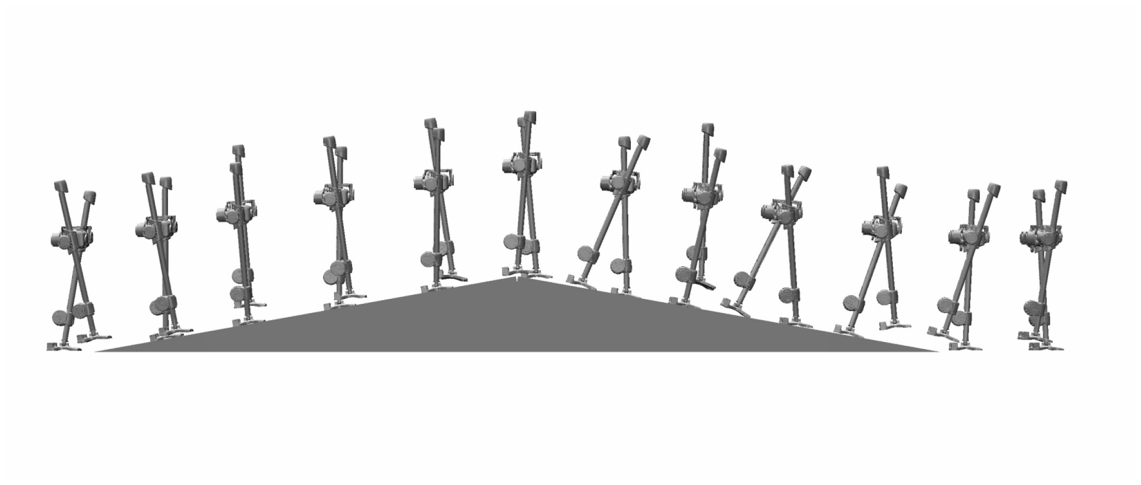


Figure 12: Impulse rejection successes from a grid of force vectors. Yellow represents full success.

5.2 Simulated Slope Climbing



The model walking at 0.5m/s was trialed against a series of increasing slope angles. The policy was evaluated on a "peak" where the slope increased and then decreased at the same rate. A full traversal both up and down the obstacle with a return to normal walking was considered a success. The model reached full success on a 12.5° slope. The results are plotted in Fig. 13.

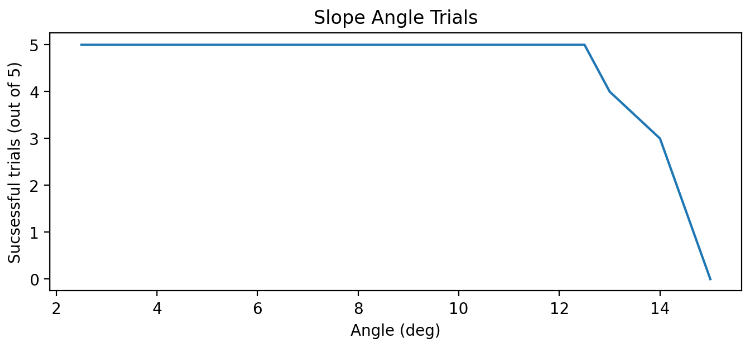
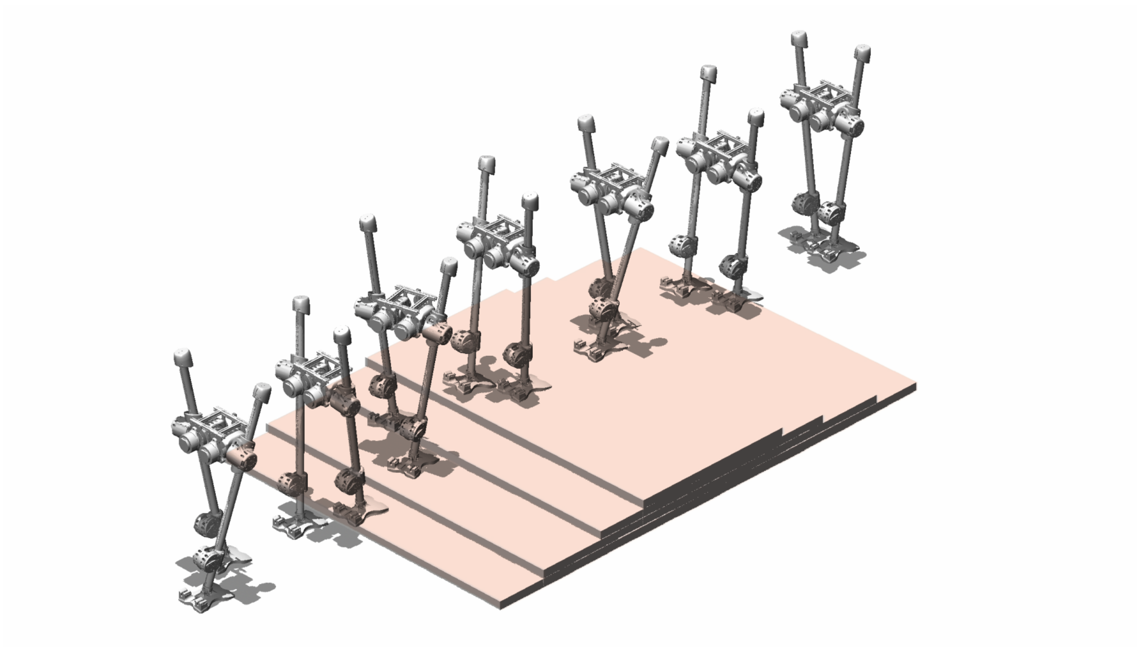


Figure 13: Results from increasing slope angle

5.3 Simulated Stair Climbing



The model walking at 0.5m/s was trialed against a series of increasing stair heights, each step was given a depth of 50cm. A full traversal both up and down the stairs with a return to normal walking gait was considered a success. The model reached full success with a 3cm step height, but could likely be improved with more stair specific training. The results are plotted in Fig. 14.

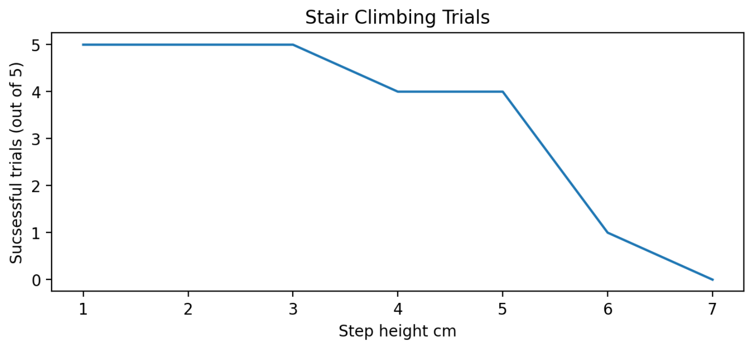


Figure 14: Results from increasing stair step height

5.4 Sim-to-Real Flat Ground Walking Trials

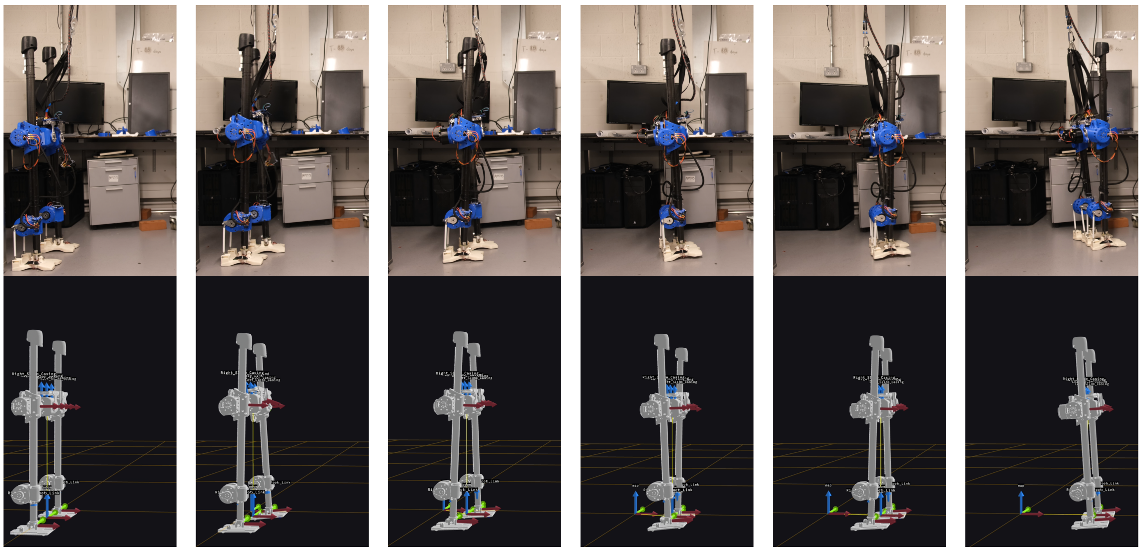


Figure 15: State estimate results from a walking trial

A series of forward walking trials were performed with the real hardware to test the controller transfer. Due to the harness, only about a meter of walking was achieved per trial. Out of 27 trials, only one fall not due to external perturbation was experienced, which was due to starting the controller too fast. The robot was robust to fairly significant perturbations as shown in Fig 16.

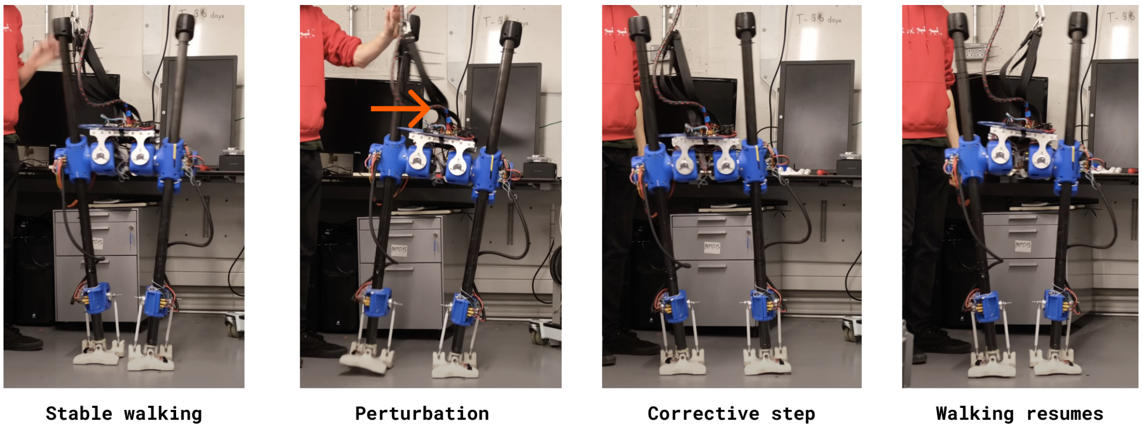


Figure 16: Robot recovering from a push

5.5 State Estimation Accuracy

To evaluate the performance of the state estimator, a walking trial was performed while recording all data coming from the robot. The final estimated x position was compared against the real measured displacement. The estimated forward distance was $0.47m$ and the real measurement was $0.67m$, giving an error of 30%. This is likely unacceptably large for odometry, but somewhat impressive given the number of assumptions the algorithm makes and the lack of a filter. While there was not time to implement in this project, a Kalman filter can be applied using the other IMU readings to help combat this. Another source of discrepancy is the naive ground contact heuristic, which can be replaced with foot contact forces either through load cells or using the leg force estimate. Fig. 15 shows a side by side of the full robot pose relative to a frames from the walking trial and Fig. 17 shows the resulting x and y plot, of note is the characteristic side to side swing present in the y trace.

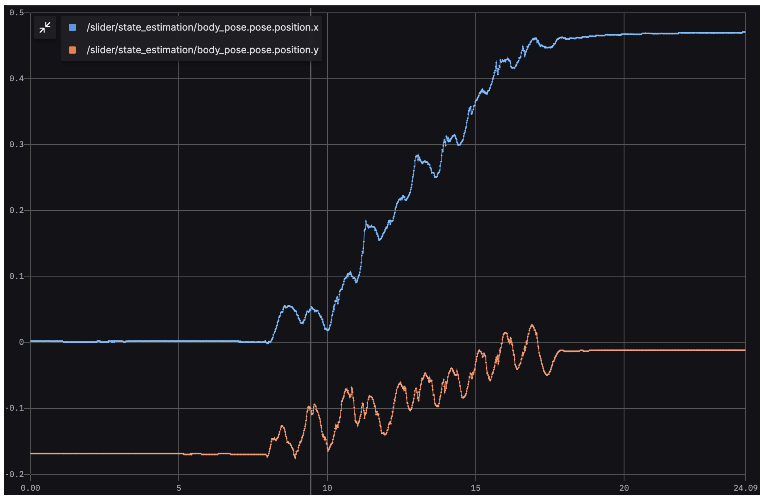


Figure 17: Position estimate plot from a walking trial

6 Conclusion

In this report, the full software stack for the sim-to-real transfer of a RL bipedal walking policy was described. The final policy was capable of slow, stable walking with disturbance rejection on the real hardware and fast, highly dynamic walking in simulation. It matches or outperformes all existing policies on the robot. To achieve the sim-to-real transfer of this policy, a joint control and state estimation system was built that can estimate full robot state using only internal sensors. A stochastic parameter estimation tool was also written to aid refinement of the model's inertia and gain parameters. The partially successful transfer onto hardware implies that the learned policy had begun to generalize.

A set of takeaways from this project are:

- **State estimation and system modeling are the largest hurdles to correctly applying a sim-to-real RL policy.** There are many great "off the shelf" frameworks for reinforcement learning and efficient simulation but minimal existing libraries for doing state estimation on legged systems. Additionally, performing automated system identification on high-dof systems with partial observantly (e.g. estimating backlash) is still an open problem.
- **Tuning the physical system should be given just as much care as tuning the control policy.**
- **Using impedance joint control significantly speeds up training and improves policy robustness.**
- **Rigid robots do not walk well.** From playing around with joint gains, there was a compliant region where the policy learned and performed best. Giving the network the ability to change joint impedance throughout the gait cycle is likely desirable and has natural precedence.
- **Feedforward plays a significant part of a walking controller.** When training, the early tests did not include the CPG input nor state history. This resulted in a purely feedback controller which did not function well, nor result in natural feeling trajectories. With joint level impedance control, a fully feedforward walking gait is feasible to generate and frees up the rest of the network to do high level feedback adjustment.
- **Given the right reward function, a periodic gait can emerge without enforcing a contact schedule.**

Bibliography

[1] *"Taskable Agility: Making Useful Dynamic Behavior Easier to Create" lecture by Scott Kuindersma.* URL: `https://mediacentral.princeton.edu/media/%22Taskable+AgilityA+Making+Useful+Dynamic+Behavior+Easier+to+Create%22+lecture+by+Scott+Kuindersma/1_z4r6sz39`.

[2] *Adafruit LSM6DSOX + LIS3MDL.* URL: `https://www.adafruit.com/product/4517`.

[3] Xinyu Bai. "Design and Implementation of a State Estimator for Bipedal Walking Robot SLIDER". MA thesis. Imperial College London, 2022.

[4] Daniel Bick and MA Wiering. "Towards Delivering a Coherent Self-Contained Explanation of Proximal Policy Optimization". MA thesis. 2021.

[5] *Cassie Sets World Record for 100M Run.* URL: `https://www.youtube.com/watch?v=DdojWYOK0Nc`.

[6] Jack Collins et al. "A review of physics simulators for robotic applications". In: *IEEE Access* 9 (2021), pp. 51416–51431.

[7] *CubeMars AK Series Dynamical Modular.* URL: `https://www.cubemars.com/category-122-AK+Series+Dynamical+Modular.html`.

[8] Milan R Dimitrijevic, Yuri Gerasimenko, and Michaela M Pinter. "Evidence for a spinal central pattern generator in humans a". In: *Annals of the New York Academy of Sciences* 860.1 (1998), pp. 360–376.

[9] *Foxglove.* URL: `https://foxglove.dev/`.

[10] *Gymnasium Python Library.* URL: `https://github.com/Farama-Foundation/Gymnasium`.

[11] Matthew Hayes. "Torque Control for a 2-DOF Parallel Ankle Mechanism". MA thesis. Imperial College London, 2022.

[12] Kazuo Hirai et al. "The development of Honda humanoid robot". In: *Proceedings. 1998 IEEE international conference on robotics and automation (Cat. No. 98CH36146).* Vol. 2. IEEE. 1998, pp. 1321–1326.

[13] Auke Jan Ijspeert. "Central pattern generators for locomotion control in animals and robots: a review". In: *Neural networks* 21.4 (2008), pp. 642–653.

[14] Shuuji Kajita et al. “The 3D linear inverted pendulum mode: A simple modeling for a biped walking pattern generation”. In: *Proceedings 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems. Expanding the Societal Role of Robotics in the the Next Millennium (Cat. No. 01CH37180).* Vol. 1. IEEE. 2001, pp. 239–246.

[15] Benjamin G Katz. “A low cost modular actuator for dynamic robots”. MA thesis. Massachusetts Institute of Technology, 2018.

[16] Tobias Kronauer et al. “Latency analysis of ros2 multi-node systems”. In: *2021 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI)*. IEEE. 2021, pp. 1–7.

[17] Joonho Lee et al. “Learning quadrupedal locomotion over challenging terrain”. In: *Science robotics* 5.47 (2020), eabc5986.

[18] Tad McGeer et al. “Passive dynamic walking”. In: *Int. J. Robotics Res.* 9.2 (1990), pp. 62–82.

[19] Robert B McGhee. “Some finite state aspects of legged locomotion”. In: *Mathematical Biosciences* 2.1-2 (1968), pp. 67–84.

[20] *MuJoCo Advanced physics simulation.* URL: `https://mujoco.org/`.

[21] *Proximal Policy Optimization (PPO), Deep Reinforcment Learning Class.* URL: `https://huggingface.co/blog/deep-rl-ppo`.

[22] Marc Raibert, Michael Chepponis, and HBJR Brown. “Running on four legs as though they were one”. In: *IEEE Journal on Robotics and Automation* 2.2 (1986), pp. 70–82.

[23] *ROS 2 Humble Hawksbill.* URL: `https://docs.ros.org/en/foxy/Releases/Release-Humble-Hawksbill.html`.

[24] *ROS 2 Humble TF library.* URL: `https://docs.ros.org/en/humble/Tutorials/Intermediate/Tf2/Tf2-Main.html`.

[25] *ROS IMU Message.*

[26] Nikita Rudin et al. “Learning to Walk in Minutes Using Massively Parallel Deep Reinforcement Learning”. In: *CoRR* abs/2109.11978 (2021). arXiv: `2109.11978`. URL: `https://arxiv.org/abs/2109.11978`.

[27] John Schulman et al. “Proximal Policy Optimization Algorithms”. In: *CoRR* abs/1707.06347 (2017). arXiv: `1707.06347`. URL: `http://arxiv.org/abs/1707.06347`.

[28] David Silver et al. “Mastering the game of go without human knowledge”. In: *nature* 550.7676 (2017), pp. 354–359.

[29] *Stable Baselines 3.* URL: `https://stable-baselines3.readthedocs.io/en/master/`.

[30] Oriol Vinyals et al. “Grandmaster level in StarCraft II using multi-agent reinforcement learning”. In: *Nature* 575.7782 (2019), pp. 350–354.

[31] Miomir Vukobratovic and Davor Juricic. “Contribution to the synthesis of biped gait”. In: *IEEE Transactions on Biomedical Engineering* 1 (1969), pp. 1–6.

[32] Ke Wang. “Design, Modelling and Control of SLIDER: An Ultralightweight, Knee-Less, Low-Cost, Bipedal Walking Robot”. PhD thesis. Imperial College London, 2022.

[33] Chengxu Zhou and Nikos Tsagarakis. “On the comprehensive kinematics analysis of a humanoid parallel ankle mechanism”. In: *Journal of Mechanisms and Robotics* 10.5 (2018), p. 051015.