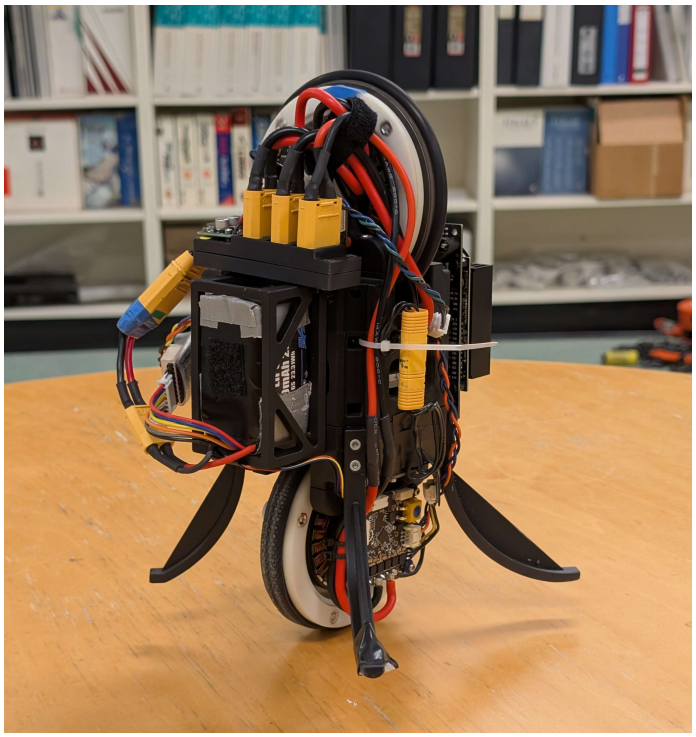
Learning to Balance

A Reaction-Wheel Unicycle Robot



Jackson Fraser
Simon Ghyselincks
Julian Lapenna
Tristan Lee
Kyle Mackenzie

ENPH 479 — Project #2454
Sponsored by the Engineering Physics Project Lab

The University of British Columbia

Vancouver, BC, Canada

April 8, 2025

Executive Summary

This two-year capstone project, sponsored by the UBC Engineering Physics Project Lab, combines robotics, controls engineering, and machine learning to develop a fully functional self-balancing unicycle robot as a platform to compare modern control theory algorithms and reinforcement learning (RL) control algorithms.

Previous projects include robots balancing using classical control and a single-axis inverted pendulum managed with RL. Furthermore, previous work in robotics has provided significant guidance and inspiration for our project. This robot extends the Wheelbot design by adding an additional axis of rotational control, incorporating RL control, and point-to-point navigation.

In this report, we present Broomy, an autonomously balancing unicycle robot. Comprehensive telemetry and sophisticated data visualization methods implemented during prototyping were critical in accelerating our design iterations and control optimization.

Our efforts began with the production of a reaction wheel inverted pendulum (RWIP) to understand unstable dynamics and build a development pipeline. Next, we designed a physical prototype of the full robot and established comprehensive telemetry and data visualization to optimize testing and design iteration. Finally, we developed modern control algorithms, validated on hardware, and RL control algorithms through a simulation pipeline. This project has demonstrated the practical effectiveness of modern control methods in robotics applications, while highlighting the potential of reinforcement learning given further optimization.

Looking ahead, future work could involve further enhancements to RL control and navigation algorithms for increased precision, improving energy efficiency, and scaling the design for diverse real-world applications.

Contents

List of Figures

1 Introduction

Control systems are an integral part of modern technologies that we rely upon. From consumer goods such as automobiles and home appliances to industrial robotics and autonomous satellites– these all require the theory and application of a controls policy to reach desired outcomes and keep us safe. However, these systems traditionally use well-established mathematical frameworks that assume simplified, linear behaviors, which may ultimately limit their performance and complexity.

Recent advances in computational hardware and algorithms have generated enormous research interest into control methods that can overcome these limitations. In particular, reinforcement learning (RL) offers a promising alternative to classical approaches by allowing a controller to learn directly through trial and error, rather than relying on a physics-based model. While RL-based controllers perform well in simulation, transferring these policies to the physical world, known as the "sim2real" transfer. remains a challenge.

This two-year senior capstone project, sponsored by the UBC Engineering Physics Project Lab, focuses on designing, building, and testing a self-balancing unicycle robot to explore these fundamental research problems. The project aims to compare and refine control algorithms that push the limits of modern robotics and controls and which form an active branch of industrial research. In the sections ahead, we detail the mechanical, electrical, and software design of the unicycle, the development of both classical and RL controllers, and a series of benchmark comparative tests and results. Finally, we will discuss the project outcomes and potential future research.

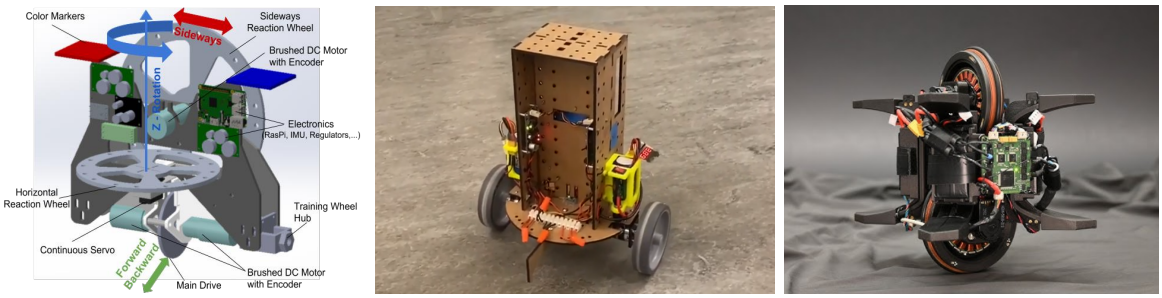
1.1 Previous Work

Several projects have addressed aspects of controlling an unstable unicycle robot. A 2019 capstone project attempted to stabilize a unicycle robot using classical control [1] and a 2022 team succeeded in achieving the sim2real transfer on a two-wheel inverted pendulum [19]. The Max Planck Institute's Wheelbot [6] demonstrated advanced control strategies and served as the starting point for our robot design. However, no previous effort has tackled the specific sim2real challenge for a unicycle with multiple reaction wheels and a third, yaw axis.

1.2 System Overview

The robot, Broomy, is composed of six main parts, shown in Figure 2. The roll wheel and pitch wheel work to balance and keep the robot upright, while an internal yaw wheel provides directional control. The motors for each wheel and the onboard Nvidia Jetson Orin Nano computer are powered with a LiPo battery pack, all held together with a 3D-printed PLA frame. It has a total height of $\sim$28cm and a mass of $\sim$2.05kg, reflecting a compact and efficient design intended to allow self-righting from a position resting on its resetting legs.

The Jetson processes sensor data, issues motor commands, and receives feedback from the motor



(a) AIUR - Team 1868, 2019 [1] *(b) TWIP - Team 2153, 2022 [19]* *(c) Wheelbot - Max Planck Institute [6]*

Figure 1: Previous work.

drivers. An external computer server communicates with the Jetson over the internet to handle web communications, telemetry, and data logging (further reading in Appendix A). While the robot is fully autonomous, the communication system enables an additional real-time control operation of the robot using an Xbox controller.

2 Design Motivation

Designing Broomy began with high-level constraints and goals for the final product:

1. Design a system that can achieve point-to-point navigation.
2. The geometry of the robot should allow for self-righting from specific rest positions.
3. The robot should carry its own power supply and computer.
4. Onboard processing should be capable of both traditional and machine-learning controlled balance.
5. Point-to-point control should be managed by external systems sending signals to the onboard processor.

These requirements informed the development of the systems, structure, and controllers detailed in the sections ahead.

2.1 Mechanical Design Considerations

The primary mechanical objective was to design a robot capable of self-righting from rest, balancing, and driving on flat ground. Achieving this required mechanical analysis and a design that meets these demands.

The first of these considerations was to make sure that the robot can self-right from rest. For this, the motor must have enough continuous torque to overcome gravity acting on the body to bring the robot to an upright position. More specifically, the torque must be greater than the moment that gravity creates at the point of ground contact, until the robot is upright. This means that having the robot's

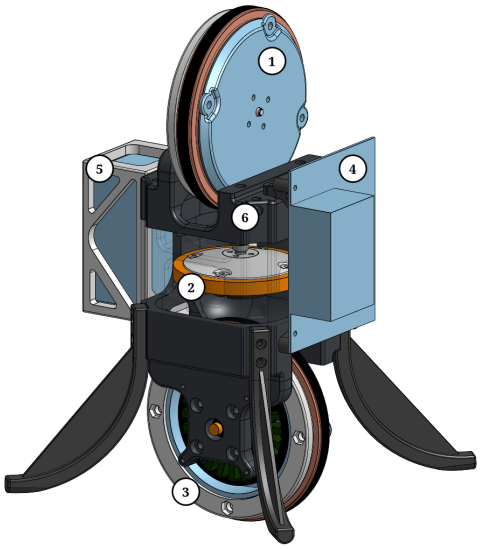


Figure 2: **Labeled robot components***(1) Roll reaction wheel, (2) yaw (azimuthal) reaction wheel, (3) pitch/drive wheel, (4) computer, (5) battery pack, (6) chassis.*

center of mass lower to the ground reduces the torque requirement. Additionally, having a lower polar moment of inertia about the contact point reduces the time to self-right, governed by Newton's second law for rotational motion:

$$\Sigma\tau = I_{polar}\alpha \tag{1}$$

Next, the three actuators are orietned in three orthogonal axes. This arrangement naturally decouples control torques into τ_x, τ_y, τ_z while balanced upright, with each motor independently controlling one axis. The benefits of this configuration are discussed further in Section 3.

The final major design consideration regards the mass of different components of the robot. As previously mentioned, having minimal mass and a low center of gravity reduces the torque requirement for self-righting. However, the motors have a maximum wheel speed at which they can no longer generate torque. Thus, it is important to ensure that the wheels never reach their maximum speed while balancing, as it requires constant small adjustment torques. Given that the torque is the product of the wheel's angular moment of inertia and its angular acceleration, it is possible to minimize the acceleration needed for any given torque by increasing the wheel inertia. Thus, we have a challenging goal to balance the minimization of the total robot mass and angular moment of inertia about the ground point, while keeping the wheel inertia high enough such that the torque required for controlling the robot never causes a wheel to accelerate to its maximum speed.

While designing the chassis, we needed to make it strong enough to withstand the forces of hitting the

ground and rapidly accelerating, as well as be resilient to vibrations that may be caused by imbalances in the flywheel's mass distribution. In our physics models used for controls, it is implicitly assumed that the whole robot is a rigid body. The implications of this are further discussed in Sections 3 and 5.4.

2.2 Actuators

This section examines the actuators used in the robot, focusing primarily on the components responsible for generating torque and enabling controlled motion across all axes.

2.2.1 Flywheel

The flywheel design is based on Wheelbot's, aiming to maximize moment of inertia while minimizing mass. Steel rings are mounted around a 3D-printed housing enclosing the BLDC motor, concentrating mass at the rotor's perimeter to optimize inertia-to-mass ratio (items (2), (4), and (1) in Figure 6).

2.2.2 Motors

The necessity for rapid, high torque maneuvers determined the selection of the robot motors and motor drivers. Detailed in Appendix B, self-righting maneuvers require approximately 1.4Nm of torque, setting the max torque needed by the motors. Brushless direct-current (BLDC) motors are most suitable for their high torque-to-mass ratio. Broomy uses the T-Motor Antigravity MN6007II [17], rated for 1.49 Nm of continuous torque, with a pancake form factor ideal for compact, direct-drive reaction wheels (item (1) in Figure 6).

2.2.3 Moteus BLDC Drivers

To achieve fine torque control, the drivers for the MN6007II's were outsourced to the moteus-n1 [12]. These boards are capable of driving the high currents necessary to produce the torques required by the robot. Since this driver runs a low-level (field-oriented [9]) control loop, the requests sent by the Jetson can be abstracted to simple single-value torque commands (Figure 3). The n1 drivers also provide

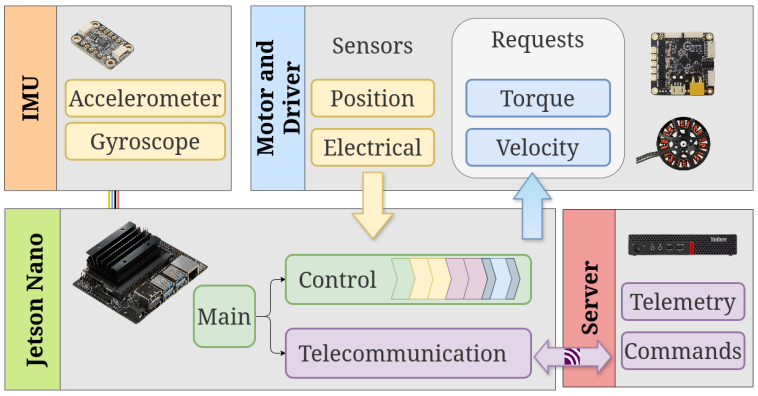


Figure 3: Software systems diagram. The central computer receives sensor data (yellow), then updates commands to the motors (blue), and maintains communication with an optional server (purple).

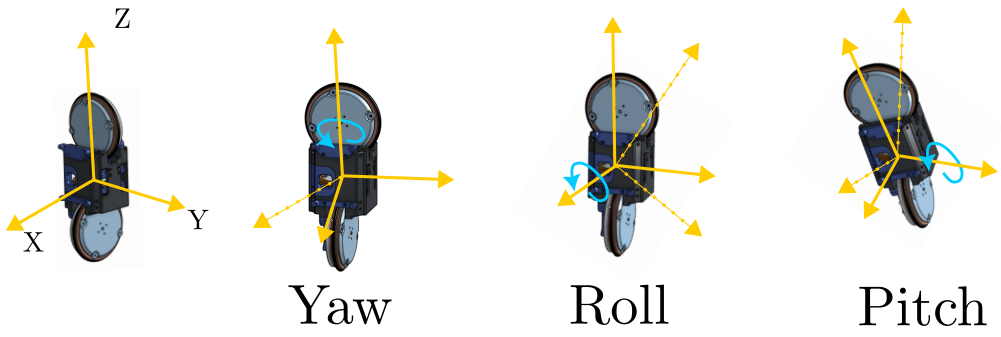


Figure 4: Three primary axes of rotation.

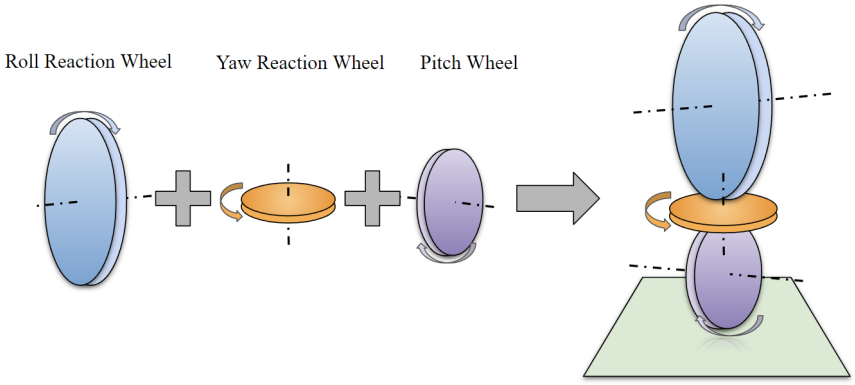


Figure 5: Orthogonal wheels reduces nonlinear coupling.

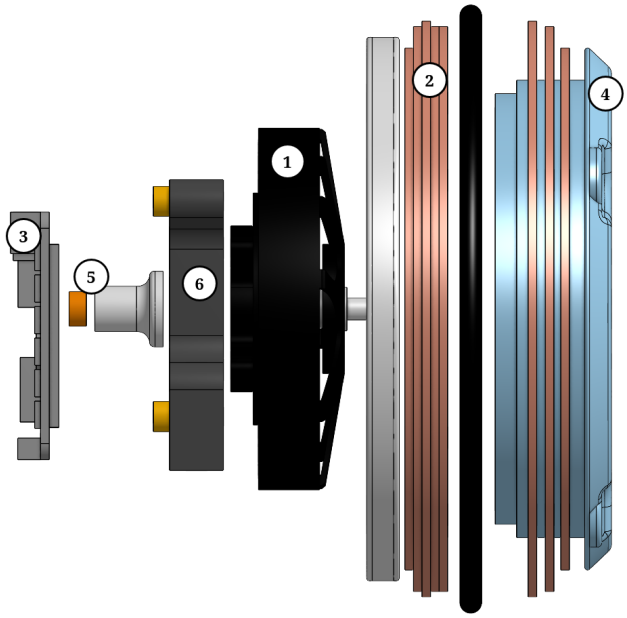
telemetry, including motor velocity, position, and phase currents (item (3) in Figure 6).

2.3 Power Supply

Broomy is powered by a six-cell, 1050mAh MaxAmps [11] lithium-polymer (LiPo) battery, rated for a 50A maximum discharge current, as necessary for the 24A current required for overcoming gravity in the rest position. This LiPo directly powered all three actuators, as well as the Jetson via a 15V step-down regulator, in total drawing an average current of 4A while balancing, resulting in 15 minutes of continuous operation. Power was distributed to the robot through a distribution board with the battery connected via an anti-spark connector to prevent inrush currents to the motor drivers, a result of their high capacitance.

2.4 Software System

The mechanical, electrical, and computational components are integrated as a system through the Orin Nano controller, running a version of Linux Ubuntu 22.04. The Linux kernel was patched with PREEMPT-RT for use in robotics applications where timing is critical, see Nvidia Forum Post for details on how this was accomplished. The software control is in Python and built around a core 80Hz control



(1) T-Motor Antigravity MN6007II
(2) Sheet metal steel rings
(3) moteus-n1 BLDC driver
(4) Resin printed housing
(5) Encoder magnet
(6) Motor and driver mount

Figure 6: Numbered exploded view of reaction wheel actuator.

cycle, a frequency which was selected based on the other similar projects, as well as the time constraints imposed by communication delays with the moteus drivers.

The central control loop shown in Figure 7 maps the stages of each cycle. The onboard inertial measurement units (IMUs) and motor states are read to get the most recent sensor readings. An estimate of the robot's current state is made and passed to a control policy to make a decision on the next torque command to send to the motors. At the end of the cycle internal data is passed out via telemetry and incoming commands are processed by handlers.

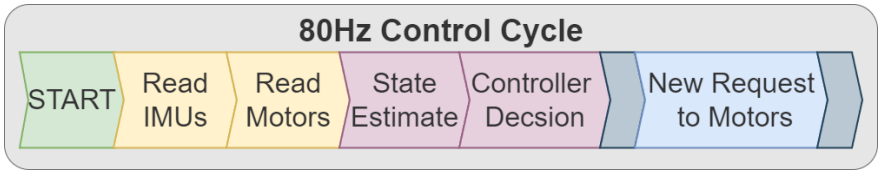


Figure 7: Orthogonal wheels reduces nonlinear coupling.

3 Dynamics and Control

This section covers the system dynamics of the robot, the state estimation from sensor data, and the control algorithms implemented.

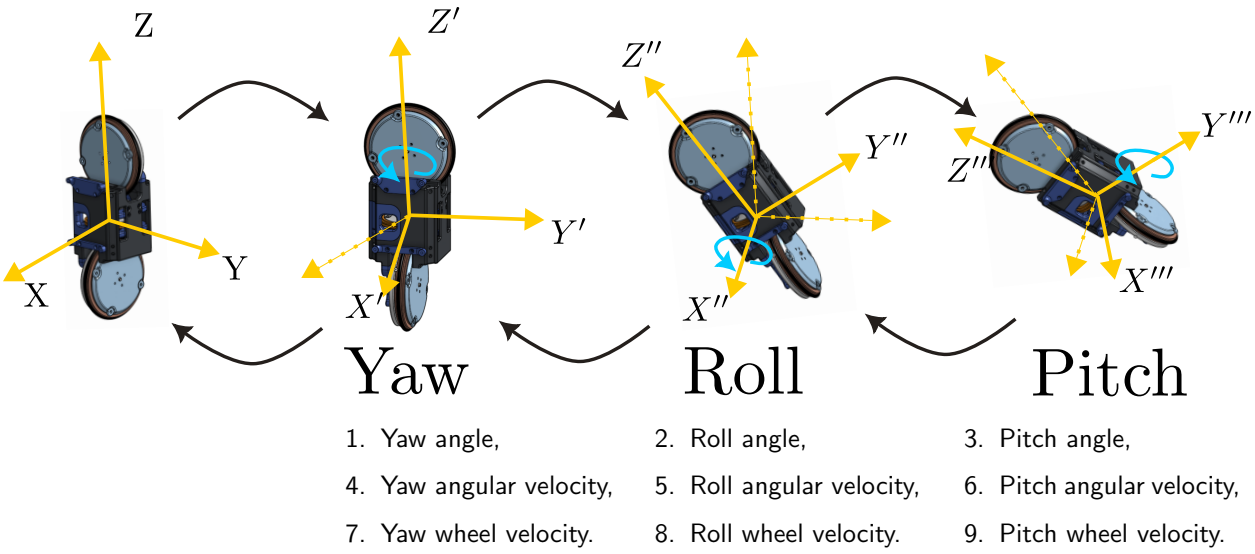


Figure 8: Robot orientation coordinates and Euler sequence. The robot orientation is described by a series of Euler rotations ZXY corresponding to yaw-roll-pitch (left to right). The reverse sequence of rotations YXZ translates from world-frame into the fixed robot frame (right to left).

3.1 Dynamics

3.1.1 Robot Coordinates

The robot's state is represented by a 9-element vector of coordinates that describes its orientation, angular velocities, and the angular velocities of each reaction wheel, fully specifying the current physical state. Each of the three spatial axes is associated with an Euler angle, an Euler rate, and an angular velocity for the wheel attached to the motor. The motor wheels are assumed to be radially symmetric and the driving surface is assumed to be a flat plane, such that the motor wheel position does not change the robot energy state and does not affect the dynamics.

Each coordinate is in SI units of radians or radians per second and follows the right-hand rule, with positive values increasing in the CCW direction. The robot angles are transformed from the inertial frame to the body-fixed frame, using a yaw-roll-pitch Euler rotation sequence. The complete system of coordinates is shown in Figure 8.

This sequence allows for reading the pitch angle rate directly from the onboard gyroscope, giving the least measurement error which is important as this wheel is the ground contact point. A more detailed discussion of the coordinate system and ordering can be found in Wheelbot [6], which uses the same convention. Making our system identification compatible with Wheelbot by using the same coordinates gives a valuable comparison point for derived dynamics.

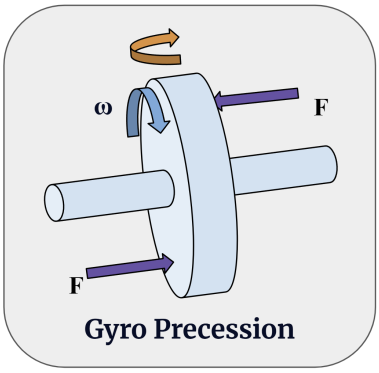


Figure 9: Torque in one axis combined with angular momentum in a second axis couples as torque in the third axis.

3.1.2 Dynamics Modeling

To create a control policy for the robot, we first require an accurate description of the dynamics of the system. A Lagrangian dynamics model of the robot was derived under the following assumptions:

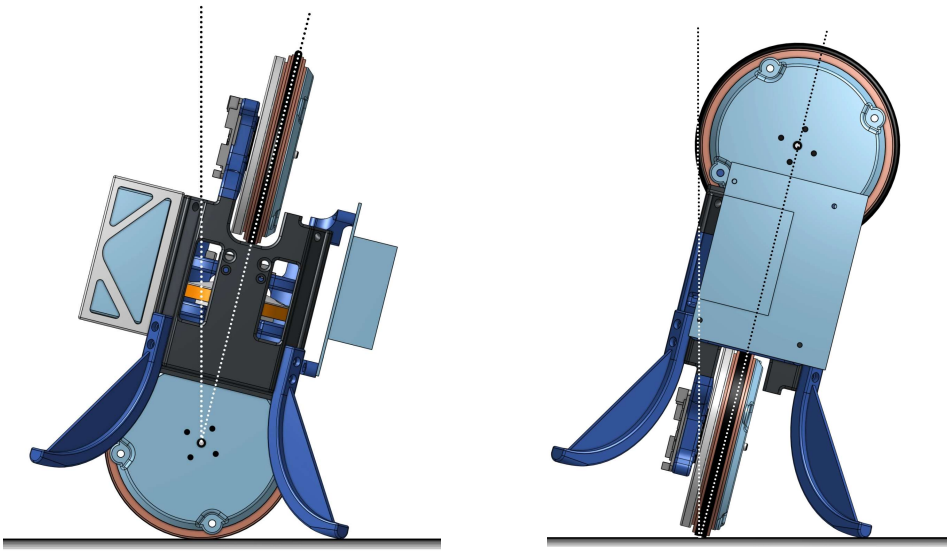
1. The robot assembly is a rigid body,
2. The drive wheel rolls without slipping,
3. Motors provide requested torques with no losses or delay.

The neglected delays and mechanical and electrical losses are expected to be compensated for by the linear feedback controller, or learned in simulation by the RL controller.

The full set of non-linear dynamics are not required for linear control policies, which instead use a simplified form linearized about the upright unstable equilibrium position or $\vec{0}$ state. The only coupling between roll, pitch, and yaw axes is via gyroscopic precession which requires two non-zero inputs, see Figure 9. The linearization removes this effect, decoupling the axes so that roll, pitch, and yaw can be treated independently. The linearized robot dynamics behaves as a composition of a wheeled inverted pendulum with a reaction wheel inverted pendulum, see Figure 10. Derivations of the roll and pitch are included in Appendix C and Appendix D.

3.1.3 State Estimation

The robot state is computed using two onboard inertial measurement units (IMUs) with onboard gyroscope and accelerometer. Estimates for the three Euler angles are made with the Madgewick [8] complementary filter, which is then combined across the two IMUs using a weighted average. The returned Euler angle estimate is combined with gyro data to estimate the three Euler rates. Using multiple IMUs provides a smaller standard deviation of error and reduces bias from the placement of the sensor. Finally, the three wheel velocities are polled from the motor drivers, completing all nine elements of the robot state vector.



(a) Front view, the point of rotation is the joint of the bottom wheel, similar to a wheeled inverted pendulum.

(b) Side view, the point of rotation is the contact point with the ground, similar to a reaction wheel inverted pendulum.

Figure 10: Resting position of robot.

3.2 Controls

While in development, multiple types of controllers were tested for capability and feasibility. The two controllers implemented on the robot were Linear Quadratic Regular (LQR) for linear control and Reinforcement Learning (RL) for machine learning-based control. A fast Model Predictive Control (fast-MPC) implementation was also developed, but early testing indicated few advantages over (LQR) for the increased complexity, so it was paused. This section covers the development of these control policies.

3.2.1 LQR Control

The linear control approach employs the standard state-space form:

$$\dot{\vec{x}} = Ax + Bu$$

where x is the state vector, u is the input vector of motor torques, and matrices A and B result from linearizing the robot's dynamics (derived in Appendix C and Appendix D).

Using LQR, the optimal control input is determined as $u = -Kx$, where the gain matrix K is computed using MATLAB's `lqr` function [10]. The calculation of K depends on tuning the weighting matrices Q and R, where Q penalizes deviations from the ideal state and R penalizes input control effort, thus balancing performance and energy efficiency. Further details of the method are found in work by Brunton

and Kutz [3].

Notably, since the linear dynamics are decoupled, the combined system matrices A and B are block diagonal matrices containing the system matrices for each control axis:

$$A \in \mathbb{R}^{9\times 9} = \begin{bmatrix} A_{\text{pitch}} & 0 & 0 \\ 0 & A_{\text{roll}} & 0 \\ 0 & 0 & A_{\text{yaw}} \end{bmatrix}, \quad B \in \mathbb{R}^{9\times 3} = \begin{bmatrix} B_{\text{pitch}} & 0 & 0 \\ 0 & B_{\text{roll}} & 0 \\ 0 & 0 & B_{\text{yaw}} \end{bmatrix}$$

and a similar composition for the input and state vectors:

$$x = \begin{bmatrix} x_{\text{pitch}} \\ x_{\text{roll}} \\ x_{\text{yaw}} \end{bmatrix}, \quad u = \begin{bmatrix} u_{\text{pitch}} \\ u_{\text{roll}} \\ u_{\text{yaw}} \end{bmatrix}.$$

Autonomous control of the yaw axis was never realized. The numerically obtained pitch and roll controllers reduce to one-dimensional feedback gains, represented as row vectors:

$$K_{\text{pitch}} = \begin{bmatrix} -3.5 & -0.25 & 0.0059 \end{bmatrix}$$
$$K_{\text{roll}} = \begin{bmatrix} -8.25 & -1.1 & -0.0045 \end{bmatrix}$$

where the first index is multiplied by the angle, the second by the angular velocity, and the third by the wheel angular velocity for their respective axis (discussed further in Section 5.1).

The primary challenges of LQR are accurate system modeling and manual adjustment of Q and R, which directly affect performance and stability. Further, as a linear model, nonlinear dynamics are not captured, and system constraints such as actuator limits are not taken into account.

3.2.2 Fast-MPC Control

Fast-MPC uses a similar set of matrices as those found in LQR control and can be used to develop a linear policy that is aware of constraints in torque inputs and the robot state, as well as the discrete time nature of the control decisions. The primary challenge with the technique is that the optimal control decision needs to be computed again at every control cycle, a computational requirement that is highly constrained by the control frequency. A full description of the implementation is found in Appendix E).

3.2.3 RL Control

The RL control approach relies on simulating a physically realistic model of the robot to iteratively improve control strategies through trial and error. An agent takes actions in an environment, receives feedback in the form of rewards, and gradually improves its decision making policy.

Using a CAD model with appropriately constrained joints, the system is simulated in NVIDIA's Isaac

Sim [15].

The primary challenges in this approach are constructing a sufficiently accurate simulation and carefully designing the reward function to incentivize the desired behaviour (discussed further in Sections 4.1 and 5.3).

4 Results

In this section we cover the results of the final product as well as the intermediary prototypes and why they were useful in the development process.

4.1 Performance in Simulation

The RL agent was trained in Isaac Gym utilizing modifications of implementations provided by [14] and [4]. We leveraged Proximal Policy Optimization (PPO) for all experiments, and focused on reward function design and tuning of other hyperparameters to improve performance instead of testing different RL algorithms.

Broomy learned to balance effectively in simulation while minimizing effort and drift from starting position.

The network architecture we used is an RNN shown in Table 1.

Layer	Type	Units	Activation	Notes
1	LSTM	32	-	Layer Normalization applied
2	Dense (MLP)	32	ELU	Default weight initializer
3	Dense (MLP)	16	ELU	No regularization applied

Table 1: Network Organization and Layer Sizes

We experimented with a wide variety of different reward functions. All contained terms related to the robot's angle from vertical, and requested torques. The reward function used in the final policy is shown below (note: the θ_z term ranges from $[0, 1]$ where zero corresponds to tipped over and one corresponds to fully upright).

$$R = \underbrace{[\theta_z \geq 0.85]}_{\text{up reward}} + \underbrace{2 \cdot \theta_z}_{\text{angle reward}} + \underbrace{1 - \|\mathbf{x}_0 - \mathbf{x}\|_2}_{\text{position reward}} - \underbrace{5 \cdot [\theta_z \leq 0.5]}_{\text{fallen penalty}} - \underbrace{\frac{1}{4} \cdot |\tau_r|}_{\text{effort penalty (roll)}} - \underbrace{\frac{1}{4} \cdot |\tau_p|}_{\text{effort penalty (pitch)}}$$
$$- \underbrace{0.15 \cdot \left(\frac{v_r}{60}\right)^4}_{\text{velocity penalty (roll)}} - \underbrace{0.5 \cdot \left(\frac{v_p}{60}\right)^4}_{\text{velocity penalty (pitch)}} - \underbrace{\frac{1}{4} \cdot \left|\sigma^2(\tau_r)\right|}_{\text{effort variance (roll)}}$$

4.2 Balance Control

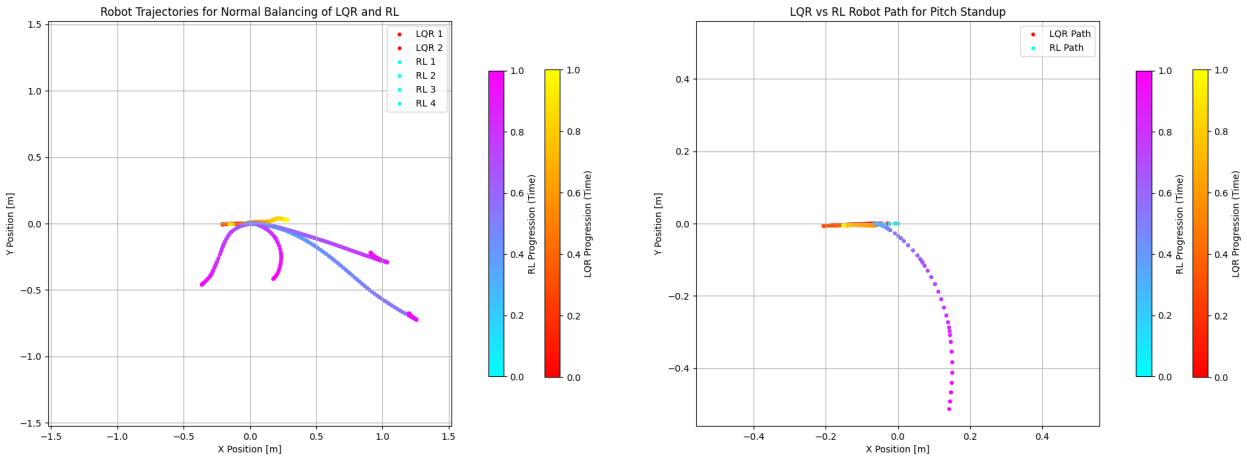
After training the RL agent in simulation, the trained policy was ported to the onboard Jetson. We then evaluated the balancing performance of both LQR and RL by taking the average balancing time across several trials.

Table 2: Balance Time Comparison between RL and LQR Controllers over Five Separate Trials. In all trials the robot started in the upright position.

Controller	Balance Time (s)
RL (Trial 1)	16.5
RL (Trial 2)	23.0
RL (Trial 3)	11.1
RL (Trial 4)	8.44
RL (Trial 5)	10.5
RL (Mean ± SD)	**13.91 ± 5.27**
LQR	Stable for full test duration (60s)

The RL controller was tested over five independent trials, achieving an average balance time of 13.91 seconds. In contrast, the LQR was able to maintain balance indefinitely, highlighting its superior stability.

While the LQR controller demonstrated effective stability, this impressive performance required extensive manual tuning of the controller gains (more discussion in Section 5.1).



(a) Balance trajectories *(b) Pitch stand-up trajectories*

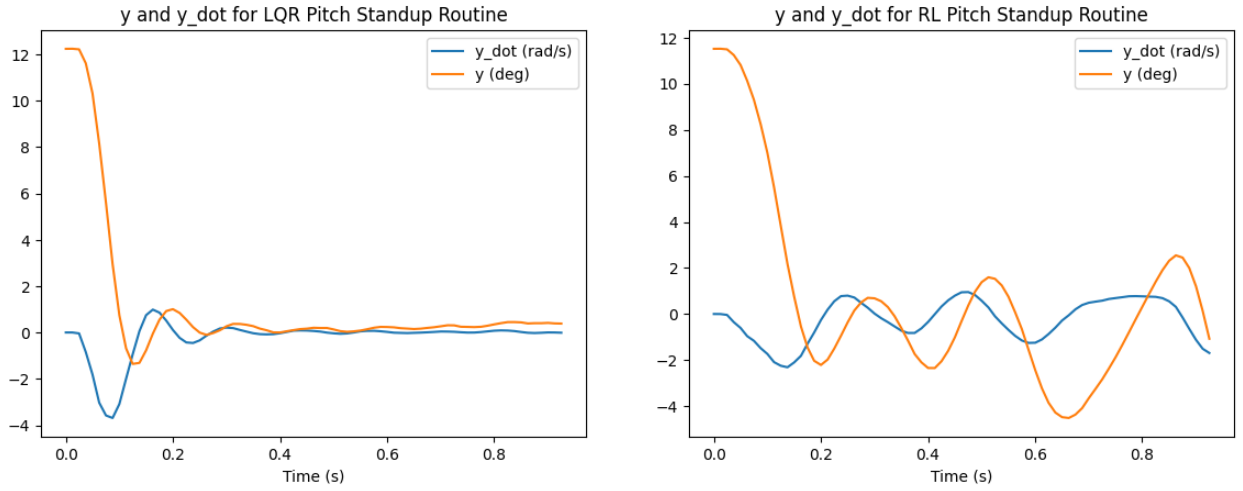
Figure 11: Control algorithm comparison: LQR and RL trajectories in the xy-plane beginning at the origin, with colour change indicating time progression. (a) shows the balance trajectories, where the LQR controller effectively maintains balance with little drift, while the RL controller consistently drifts from its starting position before falling over. (b) shows the pitch stand-up trajectories, where the LQR controller stands up within 0.2m from its starting position and maintains stability, whereas the RL controller stands up quickly but then drifts away from the start.

In contrast, the RL controller achieved balance durations averaging approximately 14 seconds. As shown in Figure 11a, the failure mode of the RL controller was a gradual drift in pitch.

Though this result is significantly shorter and less robust than LQR, this represents successful learning of the balance task through experience rather than explicit modeling. Additionally, the results achieved here were direct sim2real transfer, without any additional fine-tuning or training in the real physical environment.

4.3 Stand-up Performance

The experimental results of both "pitch" and "roll" stand-up actions for LQR and RL were evaluated. The stand-up maneuver is initiated when the robot is fully tipped over in the rest position.



(a) LQR *(b) RL*

Figure 12: Control algorithm comparison: LQR and RL standing up from rest in the pitch axis. The angle from vertical is shown in orange, and the angular velocity is shown in blue. The LQR controller manages to quickly converge to a balanced state. The RL controller manages to stand up right, but is unable to damped the oscillations and quickly goes out of control.

The LQR controller stands up and stabilizes effectively in both pitch and roll. The pitch stand-up is especially effective at staying in-place as seen in Figure 11b, while roll stand-up sometimes has large positional drift before stabilizing. The main limitation of the LQR stand-up is in roll where the robot can stabilize significantly far from the starting position.

The RL controller can also self-right in both axes, but usually cannot stabilize the robot, remaining upright for only a few seconds. This undamped oscillation can be seen in Figures 12b and 11b. The RL policy was not explicitly trained to start from the leaned over position, so stand-up performance could potentially be improved by explicitly adding this scenario into training.

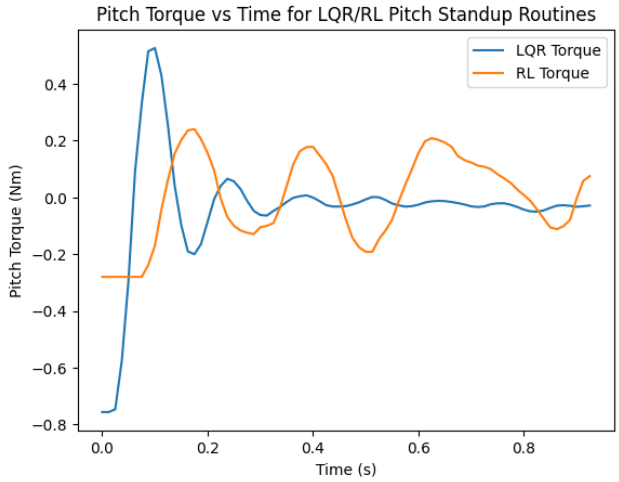


Figure 13: Control algorithm comparison: LQR and RL standing up from rest in the pitch axis. The pitch torque used by the LQR controller is shown in blue, and the RL in orange. The LQR is more aggressive and requests lots of torque immediately, converging to steady state at ∼0.4s. The RL controller attempts to perform the same damped oscillation, but it is unable to converge (reasons for the RL's torque limitations are discussed in Section 5.3).

4.4 Disturbance Rejection

We also compared the abilities of the controllers to reject external disturbances. While each was balancing, we applied disturbances in roll, pitch, and yaw.

The LQR proved to be robust to all three types of disturbances, with the biggest difficulty coming from disturbance in yaw. The RL (as we have already seen) was not as robust, but it was able to reject some of the same disturbances, performing the best in the roll direction.

Figures 14 and 15 show the responses of the two controllers to disturbances in the roll axis. The LQR controller is effective at remaining balanced, even when subject to large disturbances. The LQR has a much shorter settling time than the RL controller (see Figure 15, region three), however the RL controller has much less overshoot than the LQR.

Figure 16 shows the response of the LQR to disturbances in yaw.

Overall LQR proved to be much more robust to disturbances, especially those with larger magnitude, but the RL showed some promising results in minimizing its overshoot.

5 Discussion

This section will discuss the results, features that were partly implemented, and some difficulties of implementation.

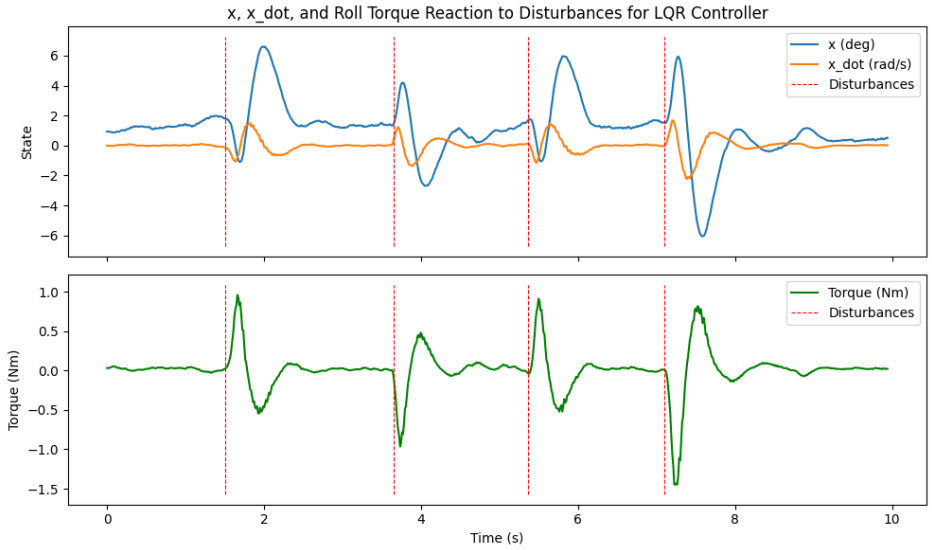


Figure 14: Control algorithm comparison: LQR Roll disturbance rejection. The controller is able to effectively respond to disturbance in the roll axis of varying amplitudes.

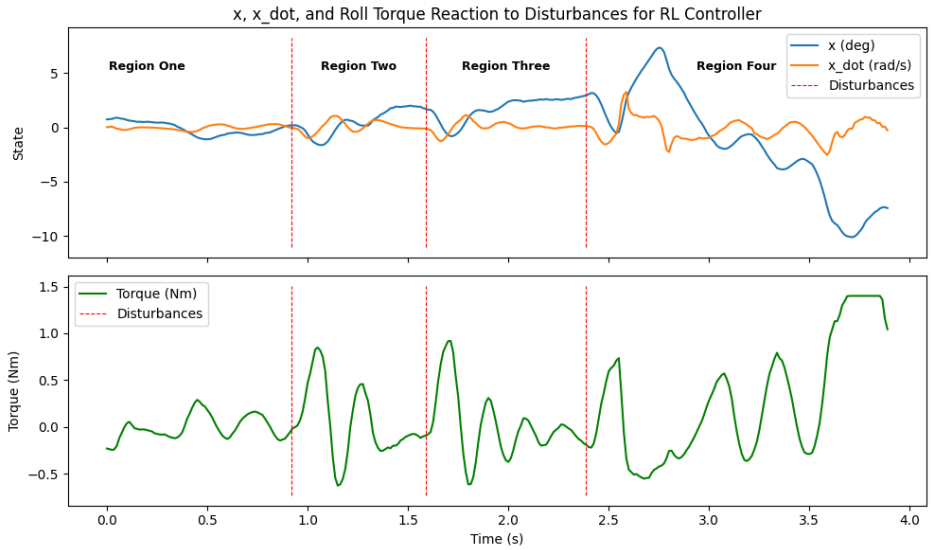


Figure 15: Control algorithm comparison: RL roll disturbance rejection. In region one we see the RL controller balancing in a steady-state. In regions two and three the RL manages to remain balanced after two disturbances in the roll axis, and it is able to dampen the applied roll torque. In region four the controller is unable to remain balanced after a larger disturbance.

5.1 LQR Controller Tuning

The tuning process for the LQR controller relied on informed trial and error of its parameters. This involved adjusting the Q and R matrices, which define the penalty attributed to each state variable and the effort attributed to each control action.

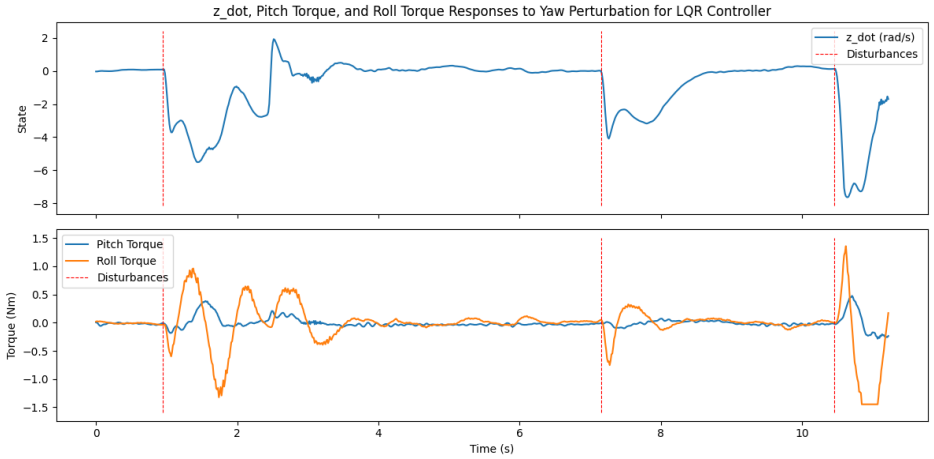


Figure 16: LQR Yaw Disturbance Rejection: Torque response of the LQR controller to disturbances in yaw is unstable. The top graph shows disturbances by reading the yaw angular velocity. The bottom graph shows the torques requested by Broomy. In comparison to the response to roll perturbations, LQR struggles to smoothly stop the robot from spinning shown in the oscillations from the first perturbation (at ∼1s). The final yaw perturbation also proved too large and the robot fell over.

Ideally, with an accurate dynamics model, only minor parameter adjustments are required for tuning. However, despite extensive efforts, the robot failed to balance with the initial LQR parameters. Depending on the tuning, various failure modes were observed, from roll motor saturation and slow tipping, to erratic instability.

Since none of the control parameter tuning worked, we began tuning the direct control outputs (i.e., the K matrix for LQR). These adjustments ultimately produced the functioning balancing controller. The likely cause of these failures was significant mismatch between the modeled and real-world dynamics.

5.2 Supervised Point-to-point Navigation & Yaw Control

The supervised point-to-point (P2P) navigation and yaw control form a separate control layer operating independently of the robot's balancing controller. Managed via MQTT telemetry, this system accepts input from an Xbox controller, enabling manual operation.

The supervised P2P navigation interprets Xbox controller inputs as a pitch angle bias. While the balancing controller maintains stability, offsetting the pitch induces forward or backward motion.

Yaw control also uses Xbox inputs, applying a proportional torque to the yaw motor.

Critically, the yaw axis lacks active stabilization; zero input does not hold the current orientation, allowing for uncontrolled spinning. This contributed to significant instability during testing (Figure 16). Moreover, yaw motor saturation amplifies gyroscopic precession from pitch and roll, further destabilizing the system. Improved yaw control would likely enhance navigation performance and would help with

the development of unsupervised P2P navigation.

5.3 Sim2Real Challenges

The transfer of the learned balancing policy from simulation to the real robot proved to be a challenging part of this project. While the final trained policy balanced effectively in simulation, when transferred to the real system it significantly underperformed relative to LQR in balancing and disturbance rejection (see Table 2 and Section 4.4).

The primary approach to closing the sim2real gap is called domain randomization. This involves randomizing various parameters in the simulation, with the assumption that the model will learn to deal with discrepancies in simulation inputs, and other robot parameters. Table 3 shows the parameters randomized in the simulation.

Another important aspect of effective sim2real transfer was randomizing the initial robot state. Applying random perturbations to the robot at later training stages also improved robustness of the final policy.

However, despite the implemented changes the learned policy still failed in real. When the policy was directly transferred to the robot the requested pitch torque acted like an undamped oscillator and the robot balanced for only a few seconds. In order to solve this, the requested pitch torque was divided by a factor of five. This resulted in a balancing model, however it is ultimately a band-aid solution which limits the pitch torque the robot can request.

Future iterations of Broomy could drastically improve upon the existing RL policy by exploring other approaches to the sim2real problem like training in real, physics informed ML approaches, or further improving upon the robustness of the simulation.

Table 3: Domain Randomization Parameters for RL Training. Correlated noise refers to unique randomization applied to each environment which remains fixed throughout training, and uncorrelated noise is random noise at each simulation step.

Parameter Type	**Description**	**Randomization Type/Range**
Observations	Correlated and uncorrelated noise on all network inputs	Gaussian noise with $\sigma = 0.005 - 0.1$
Actions	Correlated and uncorrelated noise on all torque outputs	Gaussian noise with $\sigma = 0.01 - 0.1$
Masses	Robot Body Mass	Uniform sampling within $[0.75m_0, 1.25m_0]$
Moments of Inertia	Robot Body Inertia	Uniform sampling within $[0.85I_0, 1.15I_0]$
Friction	Surface friction coefficients	Uniform sampling within $[0.5\mu_0, 2.0\mu_0]$
Gravity Vectors	Direction and magnitude of gravity vector	Uniform Sampling $g * [0.05, 0.05, 0.2]$

5.4 Vibrational Challenges

One of the main issues encountered while trying to balance the robot was chassis vibration. In deriving the equations of motion for the robot an implicit assumption was that the chassis was a rigid body, however this assumption was never tested. Observing the initial controls algorithms, the robot visibly jittered at around 20 Hz. The cause was found to be imbalance in the roll (top) wheel sending vibrations down through the chassis.

The chassis deformed under these vibrations, leading to oscillations in the IMU readings, causing the control policies fail. Notably, the IMU mounted closer to the ground wheel attachment joint had a smaller amplitude of oscillation, and less chassis deformation. By weighing that IMU's readings more heavily in our IMU data processing, the issue was largely resolved, although completely removing the oscillations may provide even better controller performance.

6 Future Development

This section covers recommendations for future development or extensions of the project, possible next steps, appendices, useful resources and references, and the deliverables.

6.1 Next Steps & Recommendations

It is our goal for teams in the future to build off this work and extend the development we have laid out here. A few immediate options that we think are achievable and would produce tangible results are:

- Control in the yaw axis
- Improved chassis resistance to vibrations (stiffer material)
- Change of motor drivers for more resilient hardware
- ML for learned dynamics error
- Positional setpoint control

6.2 Deliverables

The deliverables for the project are the following:

1. Source code: https://github.com/Team-2411-RL-Unicycle/rl-unicycle.git (Includes a media folder with select gifs and images)
2. CAD files: Main assembly
3. Omniverse Isaac Sim RL Environments Fork: https://github.com/JacksonnF/OmniIsaacGymEnvs
4. Physical prototype
5. Telemetry server

6. Spare parts and prototyping components

References

[1] *AIUR ENPH 479 Capstone Team 1868-69*. Team Project.

[2] P. Brevik. "Two-Axis Reaction Wheel Inverted Pendulum". MA thesis. Norwegian University of Science and Technology, 2017. URL: https://ntnuopen.ntnu.no/ntnu-xmlui/bitstream/handle/11250/2451060/12762_FULLTEXT.pdf?sequence=1.

[3] Steven L Brunton and J Nathan Kutz. *Data-driven science and engineering*. en. 2nd ed. Cambridge, England: Cambridge University Press, May 2022.

[4] *RLGames Repository*. URL: https://github.com/Denys88/rl_games.

[5] Richard P. Feynman, Robert B. Leighton, and Matthew Sands. "Rotation in Space". In: *The Feynman Lectures on Physics*. Vol. I. Accessed April 7, 2024. Pasadena, CA: California Institute of Technology, 1964. Chap. 20. URL: https://www.feynmanlectures.caltech.edu/I_20.html.

[6] A. René Geist et al. "The Wheelbot: A Jumping Reaction Wheel Unicycle". In: *arXiv* (July 2023). URL: http://arxiv.org/abs/2207.06988.

[7] K. Gordon et al. *OpenSim2Real Monopod Platform*. 479 Capstone Team 2166. Team Project.

[8] S. O. Madgwick, A. J. Harrison, and A. Vaidyanathan. "Estimation of IMU and MARG orientation using a gradient descent algorithm". In: *2011 IEEE International Conference on Rehabilitation Robotics*. 2011, p. 5975346. DOI: 10.1109/ICORR.2011.5975346.

[9] MathWorks. *Implement Motor Speed Control by Using Field-Oriented Control (FOC)*. MathWorks Documentation. 2024. URL: https://www.mathworks.com/help/mcb/gs/implement-motor-speed-control-by-using-field-oriented-control-foc.html.

[10] MathWorks. *lqr — Linear-Quadratic Regulator Design*. https://www.mathworks.com/help/control/ref/lti.lqr.html. Accessed: 2025-04-01. 2025.

[11] MaxAmps. *6S LiPo Battery 22.2V Collection*. MaxAmps Website. 2025. URL: https://maxamps.com/collections/6s-lipo-battery-22-2v.

[12] mjbots. *moteus-n1 Brushless Motor Controller*. mjbots Product Page. 2024. URL: https://mjbots.com/products/moteus-n1.

[13] O. D. Montoya and W. Gil-González. "Nonlinear Analysis and Control of a Reaction Wheel Pendulum: Lyapunov-Based Approach". In: *Engineering Science and Technology, an International Journal* 23.1 (2020), pp. 21–29. DOI: 10.1016/j.jestch.2019.03.004.

[14] *OmniIsaacGymEnvs Repository*. URL: https://github.com/isaac-sim/OmniIsaacGymEnvs.

[15] NVIDIA. *Isaac Sim Documentation*. Mar. 2025. URL: https://docs.isaacsim.omniverse.nvidia.com/latest/index.html.

[16] *RL-Unicycle ENPH 459 Capstone Team 2411*. Team Project.

[17] T-Motor. *MN6007 V2 Motor Antigravity Type*. T-Motor Store. 2024. URL: https://store.tmotor.com/product/mn6007-v2-motor-antigravity-type.html.

[18] California Institute of Technology. *Jacobian Linearization*. CDS Caltech. 2002. URL: https://www.cds.caltech.edu/~murray/courses/cds101/fa02/caltech/pph02-ch19-23.pdf.

[19] *TWIP ENPH 479 (Gym2Real) Capstone Team 2153*. Team Project.

[20] Berkeley University of California. *Designing Information Devices and Systems II*. EECS Berkeley. 2021. URL: https://inst.eecs.berkeley.edu/~ee16b/sp21/notes/sp21/note15.pdf.

[21] M. Veng. *Euler-Lagrange Solver*. MATLAB Central File Exchange. Oct. 2023. URL: https://www.mathworks.com/matlabcentral/fileexchange/93275-euler-lagrange-solver.

[22] Ulrike Zwiers. "Modeling of wheeled inverted pendulum systems". In: *2015 16th International Conference on Research and Education in Mechatronics (REM)* (Nov. 2015), pp. 273–277. DOI: 10.1109/rem.2015.7380406.

7 Appendices

Appendix A

Robot Telemetry Systems

A guide to robot telemetry

Telemetry and Database Systems for Capstone Projects

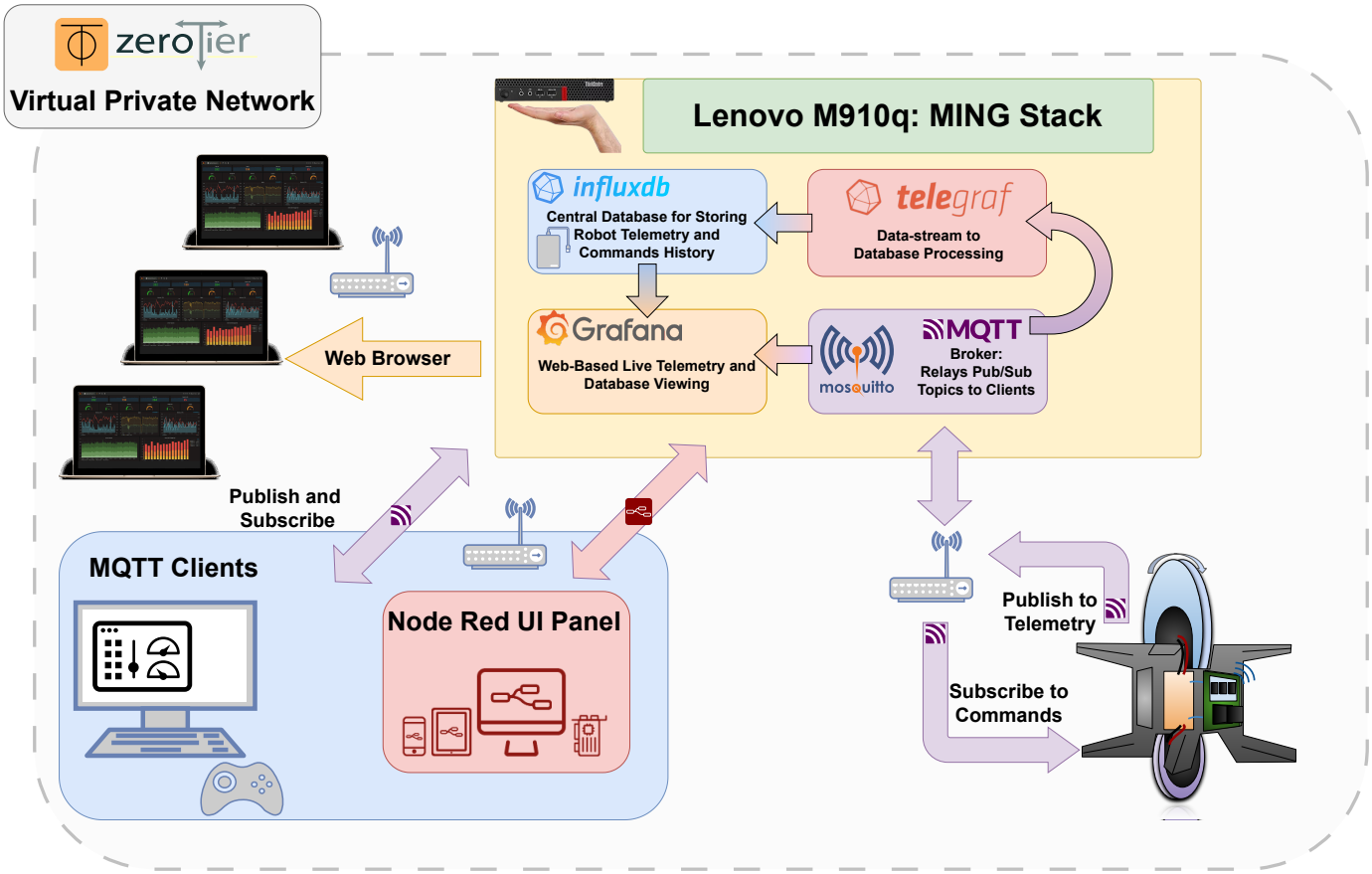
A build guide

AUTHOR
Simon Ghyselincks, Team 2411

PUBLISHED
October 12, 2024

Purpose

This guide provides a comprehensive overview of the telemetry and database systems employed in our “Learning to Balance” reinforcement learning unicycle project. It includes hardware recommendations, a detailed explanation of the networked services used, and example code to illustrate software implementation.



Telemetry Overview

The telemetry and database system functions as a two-way pipeline controlling the flow of data from sensors, motors, and control decisions. Data from the robot is efficiently offloaded to a server, considering limited processing power and the necessity for a stable control loop. The server further processes this data and manages control signals sent back to the robot for parameter adjustments or commands. By utilizing a central server and internet connectivity, the database, live telemetry, and control panel can be accessed from any networked computer via a browser or software API, globally.

Prerequisites

This guide assumes familiarity with running commands on a Linux command line and access to a terminal on your client device (e.g., personal laptop). This could be through WSL or VSCode on Windows, or a terminal on a Mac or Linux machine.

Basic understanding of Python is recommended, as some examples use Python. However, the same libraries are available in other languages such as C++.

Hardware Recommendation and Requirements

Server

We recommend the Lenovo M900 series of refurbished tiny PCs as an affordable option that meets the computational needs for a server. The device's SSD was set to dual boot into Linux Ubuntu 22.04 to run the server. This type of device is capable of handling the computational load of running multiple services simultaneously, including database management, messaging, and control services. It can also serve as a workstation for the team.

We tested the Raspberry Pi 4B 8GB with an external SSD, but its processing power is at the limit for these requirements. Therefore, it is not recommended for use as a server, especially considering the cost-effectiveness of a refurbished Lenovo.

Lenovo Server

Wifi Dongle

Robot Wi-Fi

Our robot uses an NVIDIA Jetson Nano 4GB, which does not include built-in Wi-Fi. Additionally, a real-time (PREEMPT_RT) patch has been applied to our Linux kernel. Many Wi-Fi dongle drivers are incompatible with the low-level kernel changes made by the patch; for example, the rtl8188EUS driver stopped working after the patch.

We recommend the MT7601U chipset USB Wi-Fi dongle, which works without the need for additional drivers on Ubuntu 22.04, Ubuntu 16.04 PREEMPT-RT, and Raspbian. This dongle is reliable for use with outdated and/or patched Linux kernels, is inexpensive, and can be found on Amazon or Aliexpress.

Telemetry Services Overview

The telemetry and control command communications are managed through a series of services running on a central server. Client devices, the server, and the robot are all visible to each other through a virtual network managed with ZeroTier. This allows for secure communication between devices over the internet without exposing their IP addresses to the public.

The principal components of the software stack are as follows:

- **Mosquitto MQTT Broker**: A publisher-subscriber model that allows rapid message passing between devices across topics. It is similar to the ROS topic system but is more lightweight and can be used for a wider range of applications.

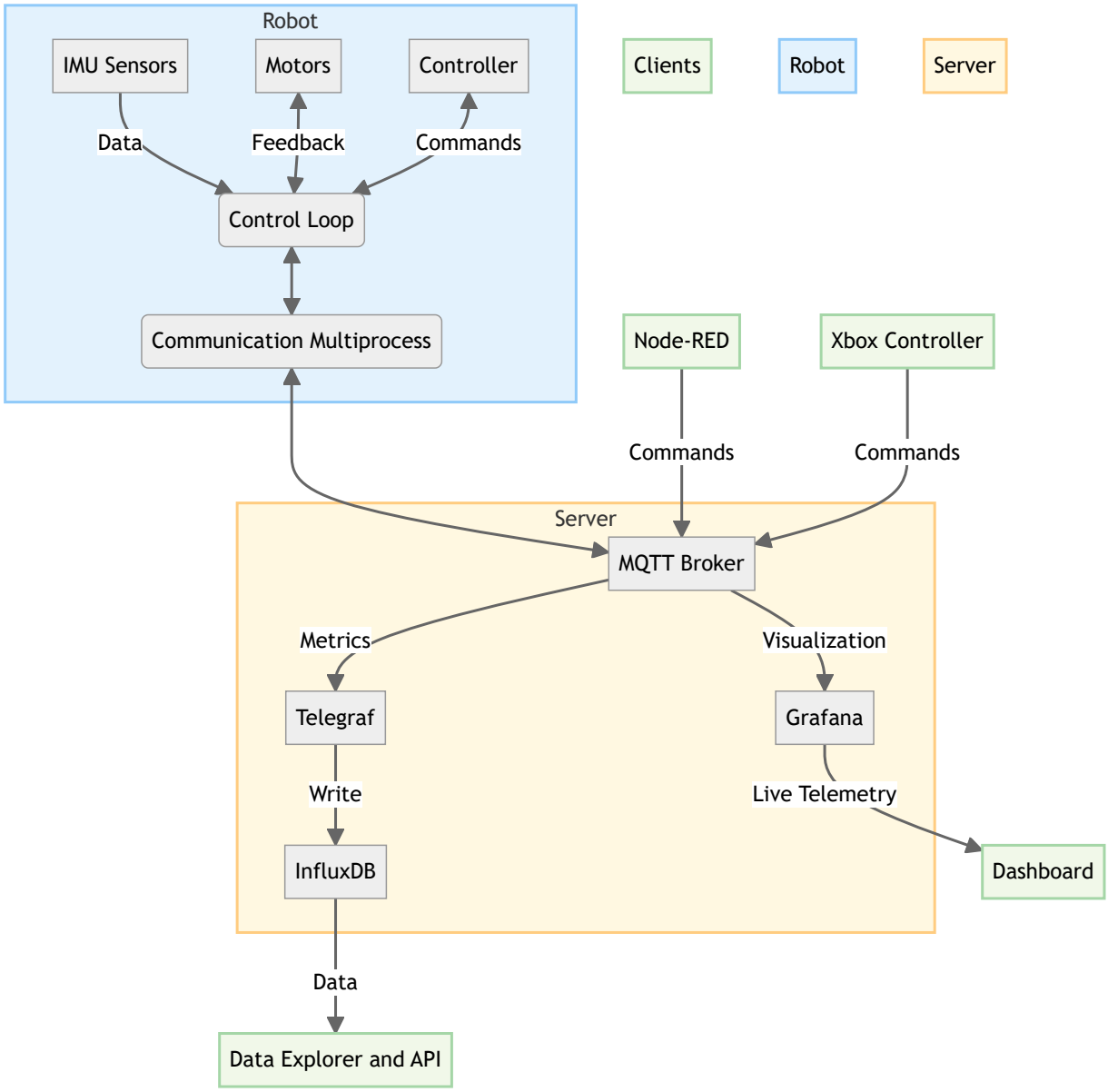
- **Telegraf**: A plugin-driven server agent for collecting and reporting metrics. It acts as a localized central switchboard for data from MQTT topics and inputs them into the InfluxDB database.

- **InfluxDB**: A time-series database used to store telemetry data streamed from the robot. It features a web browser interface for data exploration and APIs in Python and other languages for database queries.

- **Grafana**: A powerful open-source platform for creating dashboards and visualizing time-series data. It supports various data sources through plugins, including InfluxDB. For live telemetry with a fast refresh rate, the MQTT plugin can connect to the MQTT broker and display streaming data in real-time.

- **Node-Red**: A flow-based, open-source development tool for visual programming. It provides a browser-based editor that makes it easy to wire together flows using a wide range of nodes. In our case, this enables a synchronized control panel accessible through a web browser.

A simplified flowchart of the system is shown below:



We will now examine each of these services and how to configure them for a robotics project.

ZeroTier Virtual Network

ZeroTier allows all authorized devices on the network to communicate directly using assigned virtual IP addresses, similar to running a local network over a Wi-Fi router.

IP Addresses: When a website address is entered into a browser, the request is sent to a Domain Name Server (DNS), which translates the address into an IP address—a unique identifier that functions like a postal address, marking the exact location of a server on the internet. For example:

```
❯ ping google.com
PING google.com (142.251.33.78) 56(84) bytes of data.
64 bytes from sea09s28-in-f14.1e100.net (142.251.33.78): icmp_seq=1 ttl=114 time=21.6 ms
```

The Google webpage can be accessed by typing the IP address directly into the browser. The number is a unique identifier for that server on the internet.

Public and Private IP Addresses: The IP protocol reserves certain ranges of IP addresses for pivate networks. For example, the entire block of addresses `192.168.0.0 – 192.168.255.255` do not point to the wider internet but is reserved for local devices. This is why home routers can all have the same common IP address of `192.168.0.1` without creating any conflicts. It acts as a local addressing system, like apartment numbers in a building.

Network Ports: Ports differentiate between different services running on the same IP address. For example, a web server might run on port 80, while an email server might run on port 25. When entering a website address, the browser automatically connects to the server on port 80. To connect to a different service, you can specify the port using a colon, e.g., `http://172.22.1.1:8086/` connects to port `8086` which is commonly used for InfluxDB.

Local Host: The IP address `http://localhost` s a special address that points to the local machine. It is used to access services running on the same machine without needing to know the IP address.

The clients on the ZeroTier network connect to the robot and server using their assigned virtual IP addresses managed by the ZeroTier service.

Setting up ZeroTier Network

To setup a network you should first create a free account at https://my.zerotier.com/. It is advisable to set up a team email so that any team member can log in to manage the network as needed. Once you have an account, you can create a network and add devices to it. The network ID is a 16-digit number used to identify the network.

ZeroTier Client Setup

Every device intended to be part of the network – including laptops, the server, and the Jetson – should have the ZeroTier client installed. After installation, enter the network ID from the ZeroTier website into the client, and approve the device to join the network. Assign static IP addresses, especially for critical devices like the server. This can be managed via the ZeroTier website.

Installation Instructions: Download the ZeroTier client here.

Steps:

1. **Download and Install the ZeroTier client** for your operating system:
2. **Start the ZeroTier service**:
   - **On Windows**:
     - Open the ZeroTier client, which will add an icon to the system tray.
     - Right-click on the icon and select `Join Network`, then enter the network ID.
     - Set the client UI to launch on startup.
   - **On Linux**: Run the following commands:

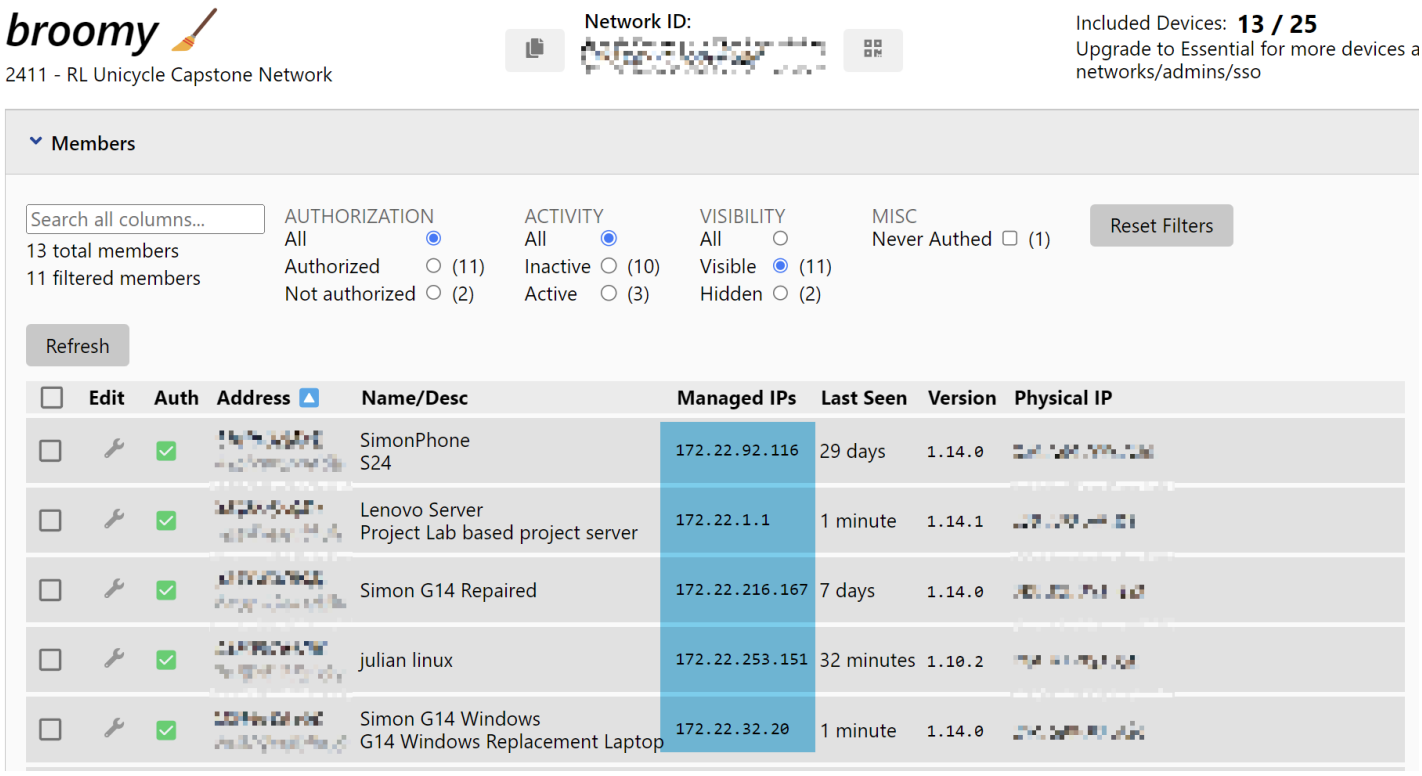
     ```
     sudo systemctl enable zerotier-one
     sudo systemctl start zerotier-one
     sudo zerotier-cli join YOUR_NETWORK_ID
     ```

3. **Approve the device**:
   Take note of the client ID and either log into the ZeroTier website or use the command-line interface to approve the device for network access.
4. **Verify the connection**:
   After approval, you can verify the connection by pinging another connected device on the network using its assigned virtual IP address.

5. **Assign a static IP (optional)**:
   For important devices like the server or the Jetson, assign static IP addresses through the ZeroTier web console under the **Members** tab. This ensures consistent IP allocation across reboots.

Managing IP Addresses

Below is an example of a ZeroTier network where the server has been assigned the static IP address `172.22.1.1` on the network:



ZeroTier Network and Access

Connecting to Robot Controller using SSH

With the virtual network established, you can enable remote access to the robot controller via SSH. SSH is a secure shell protocol that allows you to run command-line commands on a remote device. This is very useful for managing the robot, running scripts, and updating software code. Use the virtual IP address assigned to the robot on the ZeroTier network. It is recommended to assign a static IP address using the ZeroTier UI or CLI so that the address does not change between reboots.

For faster access to SSH devices, consider setting up an alias or an SSH key once SSH is verified working for a device. More information on SSH can be found here

Example: Our robot has the static ip address `172.22.0.5`. To connect from a linux terminal on a computer connected to the private network with zerotier client installed, run the following command:

```
ssh jetson@172.22.0.5
```

This will attempt to log in to the username `jetson` on the robot controller, prompting for the user password (the same as if logging in directly on the robot). For a controller running Linux, we recommend setting up different users on the system so that each team member can log in to their own account and manage their own files and credentials. Software between user accounts can be shared using symbolic links to a central repository or using GitHub to manage individual software branches.

VSCode Server

The Jetson is also able to handle VSCode Server (https://code.visualstudio.com/docs/remote/ssh), although the outdated Ubuntu 16.04 requires running an older version of the VSCode IDE to SSH in. Be aware that VSCode Server is active on the Jetson when SSHed in, which consumes some system resources; however, in practice, it has not been a performance issue. For best performance, running the robot through a simple terminal is recommended. The benefit

of SSHing through VSCode is that it provides a GUI interface for software development and file management on the robot, as if you were using VSCode on your own computer.

MQTT Overview

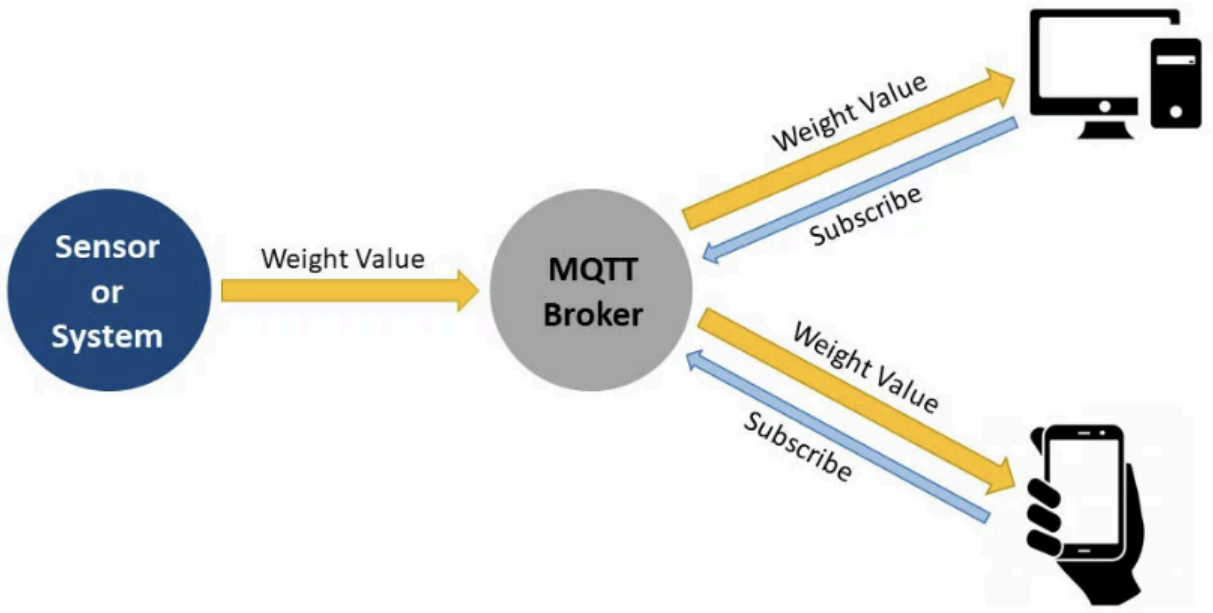
MQTT is a lightweight messaging protocol that provides an efficient and cost-effective method for telemetry-based communication between devices. MQTT messages are routed through the Lenovo server acting as the broker using Mosquitto. The robot can publish data to a topic, which can be picked up by various subscribers such as Grafana or other laptops and devices connected to the broker and subscribed to the topic. Similarly, commands can be sent back to the robot via a command topic to turn it on or off or adjust parameters. The default port for MQTT is `1883`.

For setup, installation, and maintenance of the broker, we recommend installing MQTT Explorer on any device connected to the network. This allows for monitoring all messaging and client connections across the system.

Download from MQTT Explorer

Key Features of MQTT

Feature	Description
Lightweight Protocol	Ideal for constrained devices and networks with limited bandwidth.
Publish-Subscribe Model	Allows devices to publish messages to a topic and any client subscribed to that topic will receive the messages.
Reliable Message Delivery	Offers various levels of Quality of Service (QoS) to guarantee message delivery.
Minimal Overhead	Adds only a small overhead to each message, ensuring efficient use of network resources.



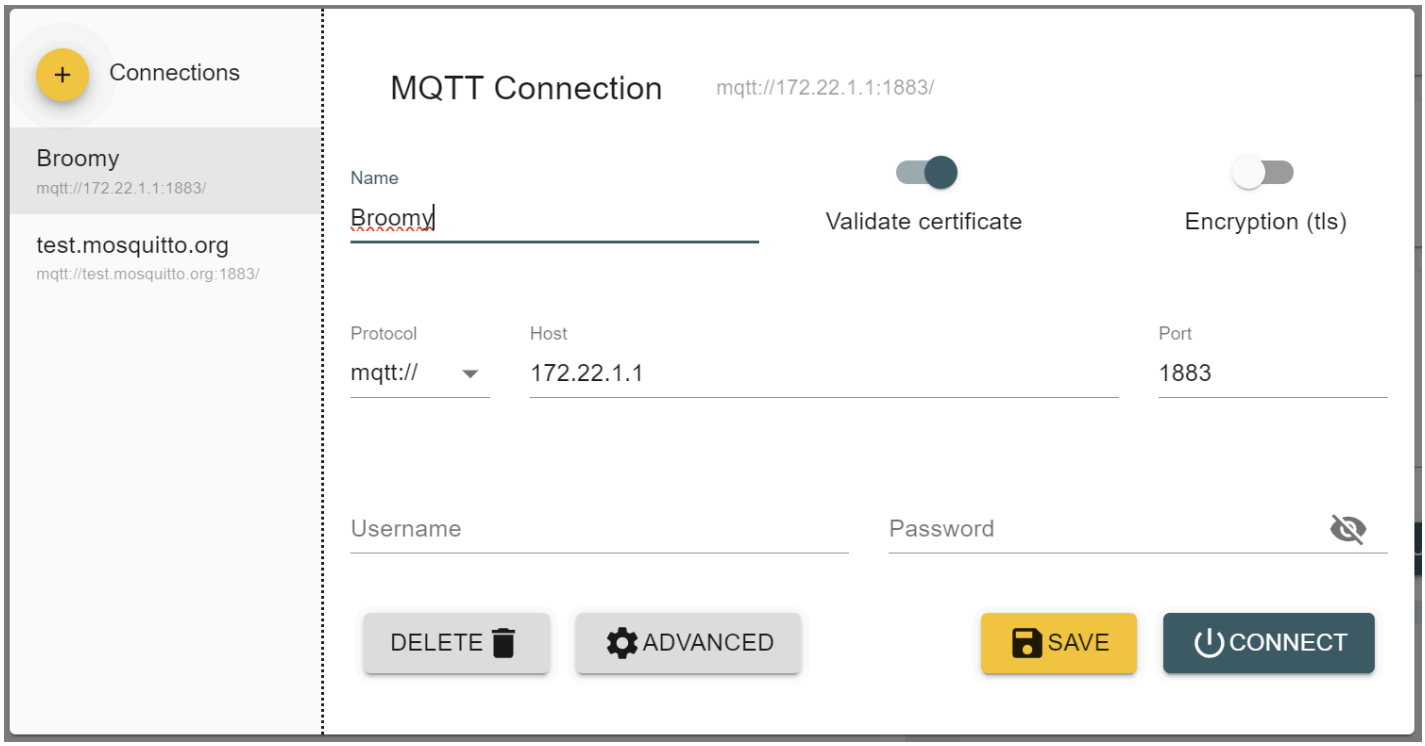
Installing Mosquitto MQTT Broker

From the server open a terminal and run the following commands to install the MQTT broker:

```
sudo apt update
sudo apt install mosquitto mosquitto-clients
```

```
sudo systemctl enable mosquitto
sudo systemctl start mosquitto
```

After installation, open MQTT Explorer and connect to the broker using the IP address of the server and the default port. You should see the server as a client connected to the broker.



Connecting to MQTT Broker

Once connected, some test messages can be sent through the GUI and verified that they are being received by the server.

For more detailed instructions or help, consider using additional resources or consult the Mosquitto documentation.

Interfacing with Software

MQTT interfaces with Python, Node-RED, and Grafana to provide a network of communication topics. The broker can be accessed by any device on the ZeroTier network; messages can be sent to a topic, or actions can be taken based on a message received from a topic.

Below is an example Python script for publishing messages to the MQTT broker. This script publishes a test message to the topic `jetson/telemetry` every second. Note that the two key components of a successful message are the topic and the message payload. The topic is the address to which the message is sent, and the payload is the data being sent. The payload can be a string, a number, or a JSON object. For our project, we use JSON objects to send data.

The package is installed using pip: `pip install paho-mqtt`

```
import json
import time
import paho.mqtt.client as mqtt
import random

# Define the MQTT settings
broker_address = "172.22.1.1"  # Lenovo's IP address (replace with your broker IP)
port = 1883
topic = "jeston/telemetry"

# Create an MQTT client instance
client = mqtt.Client()

# Define the callback for receiving messages
def on_message(client, userdata, message):
    print(f"Message received on topic {message.topic}: {message.payload.decode()}")

# Define the callback for connecting to the broker
def on_connect(client, userdata, flags, rc):
    print("Connected to broker with result code " + str(rc))
    # Subscribe to the topic when connected
```

```
    client.subscribe(topic)

# Assign the callbacks
client.on_message = on_message
client.on_connect = on_connect

# Connect to the broker
client.connect(broker_address, port)

# Start the loop to process messages
client.loop_start()

# Publish some test messages to the topic every second
try:
  range(3)
  for i in range(3):
        message = {"sensor": "temperature", "value": 20 + random.random() * 5}
        client.publish(topic, json.dumps(message))
        print(f"Published message: {message}")
        time.sleep(1)
except KeyboardInterrupt:
    print("Exiting...")

# Stop the loop and disconnect
client.loop_stop()
client.disconnect()
```

```
Published message: {'sensor': 'temperature', 'value': 20.3095790384297}
```

```
C:\Users\sghys\AppData\Local\Temp\ipykernel_15800\406674647.py:12: DeprecationWarning: Callback API version 1 is deprecated, update to latest
version
  client = mqtt.Client()
```

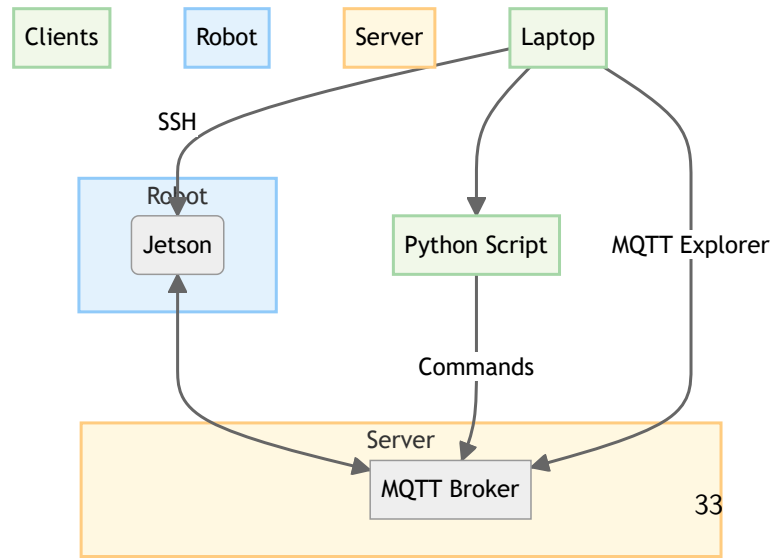
```
Connected to broker with result code 0
Published message: {'sensor': 'temperature', 'value': 23.604692472457177}
Message received on topic jeston/telemetry: {"sensor": "temperature", "value": 23.604692472457177}
Published message: {'sensor': 'temperature', 'value': 22.55116248680919}
Message received on topic jeston/telemetry: {"sensor": "temperature", "value": 22.55116248680919}
```

```
<MQTTErrorCode.MQTT_ERR_SUCCESS: 0>
```

In this demo script, the client is both publishing and subscribing to the same topic. In practice, we use multiple topics for different data streams and commands. A more advanced implementation for assigning topics and managing data can be found in our repository: RLUnicycle

This test script is useful for publishing test data when it comes to verifying the installation of InfluxDB and Grafana ahead.

We are now at this stage in the setup:



Telegraf and InfluxDB

Telegraf and InfluxDB are free and open-source products from InfluxData. Telegraf is driven by a configuration file that organizes incoming data from multiple sources for processing and forwarding into the InfluxDB database. InfluxDB is a time-series database used to store telemetry data streamed from the robot. It features a web browser interface for data exploration and APIs in Python and other languages for database queries.

Follow the instructions in the links above to install both services on the server.

InfluxDB Configuration

The InfluxDB database can be accessed through a web browser by navigating to the IP address of the Lenovo server on port 8086. For example, `http://172.22.1.1:8086/`. A login process will establish a username, password, and organization. The organization is simply a way to group data together across users. A bucket is a way to group data together within an organization. For our database we have assigned the organization name as `Capstone` and the bucket name as `telegraf` but these are free to choose. Once the organization and bucket have been created, the database is ready to receive data.

Telegraf Configuration

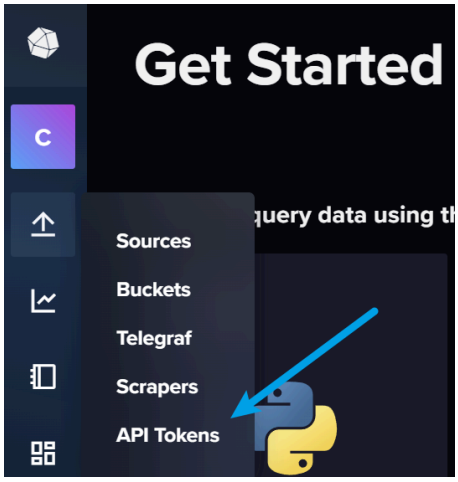
The Telegraf configuration file is located at `/etc/telegraf/telegraf.conf` by default. The default configuration is extensive, with many lines commented out. It is advisable to back up the original file and then remove unnecessary commented lines for clarity.

Telegraf acts as a central messaging switchboard. To utilize it, we need to connect the MQTT topics into the switchboard and connect the output to the InfluxDB database for storage. The main changes suggested from the default configuration are:

1. Remove the logging of server stats from the pool of inputs.
2. Add the MQTT input plugin to the configuration file.
3. Add the InfluxDB output plugin to the configuration file.

A simplified header is shown below:

```
# Default Header
[global_tags]

# Configuration for telegraf agent
[agent]
  ## Default data collection interval for all inputs
  interval = "10s"
  round_interval = true
  metric_batch_size = 1000
  metric_buffer_limit = 10000
  collection_jitter = "0s"
  flush_interval = "10s"
  flush_jitter = "0s"
  precision = ""

  ## Override default hostname, if empty use os.Hostname()
  hostname = ""
  ## If set to true, do no set the "host" tag in the telegraf agent.
  omit_hostname = true
```



Now, add InfluxDB as an output plugin and MQTT as an input plugin. The MQTT plugin listens to messages on particular topics, and the InfluxDB plugin writes the data to the database. The configuration for the InfluxDB output plugin is shown below.

Note the **token** above can be generated through the InfluxDB web interface.

```
[[outputs.influxdb_v2]]
  # localhost assumes telegraf and influxdb are on the same server
  urls = ["http://localhost:8086"]
  token = "api token from InfluxDB"
  organization = "Capstone"
  bucket = "telegraf"

[[outputs.prometheus_client]]
    listen = ":9273"
    metric_version = 2
```

Finally the incoming messages from MQTT are processed. A very important consideration here is to fully automate the process of message conversion from JSON to adopt a robot-driven database. The core principle is that changes in the robot software and telemetry should not change either this configuration file or the database schema. In our case, all messages that are to be databased start with `robot/` and the topic indicates the data category. For example `robot/imu1` is the MQTT topic that recieves information on the imu sensor

```
{ax: 0.1, ay: 0.2, az: 0.3, gx: 0.4, gy: 0.5, gz: 0.6}
```

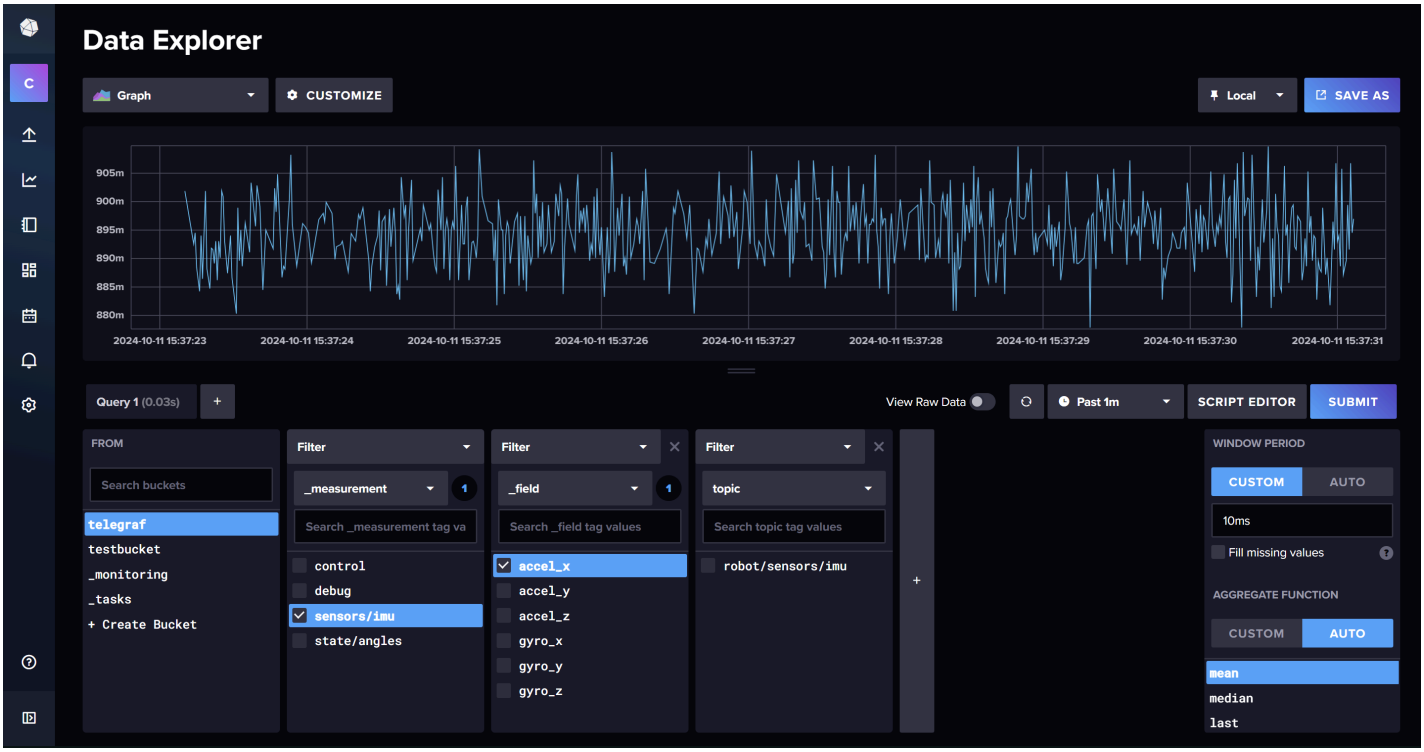
Telegraph identifies that this is to be databased, removes the `robot/` prepend, records the json message `_measurement` as `imu1` and the `_field` as `ax`, `ay`, `az`, `gx`, `gy`, `gz`.

```
[[processors.starlark]]
  source = '''
def apply(metric):
    # Get the topic tag value (e.g., "robot/motor")
    topic = metric.tags.get("topic")

    # Extract the part after "robot/"
    if topic.startswith("robot/"):
        measurement = topic.split("robot/", 1)[1]
        # Set the new measurement based on the tail of the topic
        metric.name = measurement

    return metric
'''

# MQTT Consumer Input Plugin
[[inputs.mqtt_consumer]]
  servers = ["tcp://localhost:1883"]

  topics = [
    "robot/#"  # Subscribe to all subtopics under robot/
  ]
  qos = 0
  client_id = "telegraf_mqtt_consumer"
  data_format = "json"
  ## Use a part of the topic or JSON structure as the measurement name.
  json_name_key = "measurement"
```

This completes the configuration file. The other components that record server metrics can be removed to keep the database focused.

Testing the Configuration with Data Explorer

Return to the MQTT Python script and send messages to a `robot/` topic for testing. They should now be automatically processed into the database as described. Verify that the messages are passing through the MQTT broker, then confirm they are reaching the InfluxDB database using the Data Explorer.



f successful, the Data Explorer will show that the bucket has new data. The measurement filter will display the topic passed to MQTT, the field will show the keys from the passed JSON, and the data will show the values. Note that InfluxDB automatically applies data operations such as aggregation to reduce the number of sample points. This can be managed using the window period on the right-hand side.

InfluxDB has its own query language, which can be previewed by clicking the **Script Editor** button. This provides direct insight into how the data is processed when a query is sent and can be edited to fine-tune the settings or used as an API call from elsewhere (e.g., from a Python script to create plots).

```
from(bucket: "telegraf")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
  |> filter(fn: (r) => r["_measurement"] == "sensors/imu")
  |> filter(fn: (r) => r["_field"] == "accel_x")
  |> aggregateWindow(every: 100ms, fn: mean, createEmpty: false)
  |> yield(name: "mean")
```

The `accel_x` is being aggregated into 100ms sample periods using the mean of all values in that window. This can be modified to get raw data or changed to a different aggregation function. The range values can also be set to relative times to get the last 10 minutes of data, for example.

```
from(bucket: "telegraf")
  |> range(start: -10m)
  |> filter(fn: (r) => r["_measurement"] == "sensors/imu")
  |> filter(fn: (r) => r["_field"] == "accel_x")
```

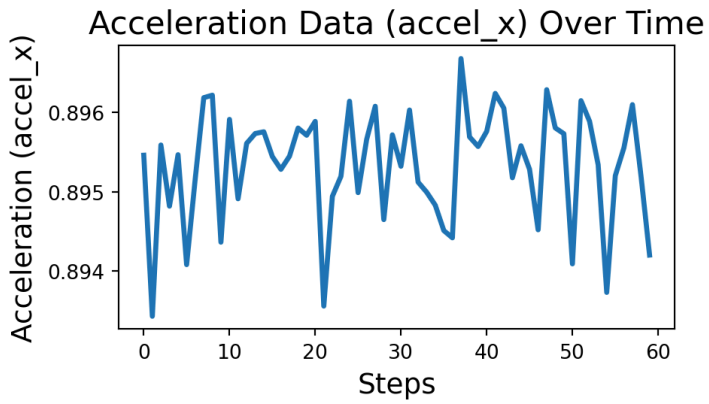
Building these queries through the script editor is a good way to get the correct string to use in a Python script to query the database.

Example Python Query

To illustrate how this can be integrated into Python for data analysis, here is a simple example. This script queries the last one minute of data from the database and plots the acceleration data over time.

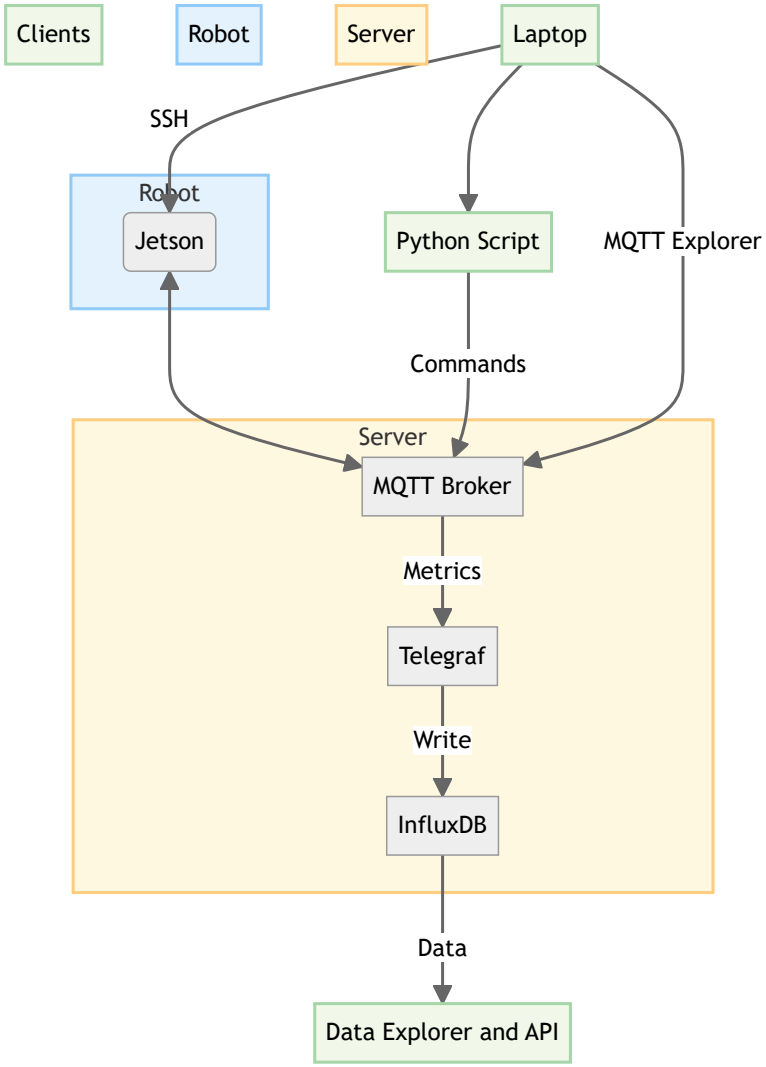
```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from influxdb_client import InfluxDBClient

# Server IP address with InfluxDB port
url = "http://172.22.1.1:8086"
token = "gGu-3t4Avltf6-yHamGXItRfOKBQIDLgWEfhdURE7wURQazK_yvIa8O9k0O-_doXX8Q0Acy82vVavb5AcM2Lhw=="
org = "Capstone"
bucket = "telegraf"

client = InfluxDBClient(url=url, token=token, org=org)
```

```
# Query for the last 10 minutes of data
last_mins = 1
query = f'''
from(bucket: "{bucket}")
  |> range(start: -{last_mins}m)
  |> filter(fn: (r) => r["_measurement"] == "sensors/imu")
  |> filter(fn: (r) => r["_field"] == "accel_x")
  |> aggregateWindow(every: 1s, fn: mean, createEmpty: false)
'''

# Query the data
query_api = client.query_api()
tables = query_api.query(org=org, query=query)

# Extract values (accel_x) from query response
values = [record.get_value() for table in tables for record in table.records]

# Plot using Seaborn
plt.figure(figsize=(5, 3))
sns.lineplot(data=values, linewidth=2.5)

# Customize plot
plt.title('Acceleration Data (accel_x) Over Time', fontsize=16)
plt.xlabel('Steps', fontsize=14)
plt.ylabel('Acceleration (accel_x)', fontsize=14)
plt.tight_layout()

# Display the plot
plt.show()
```



This concludes the setup of the MQTT, Telegraf, and InfluxDB services. The next step is to setup Grafana for live telemetry and database dashboards.

At this stage in the setup, the system architecture is as follows:



Grafana Live Telemetry

Grafana is a powerful open-source platform for creating dashboards and visualizing time-series data. Grafana supports a wide range of data sources and can display both live and historical data. While it can refresh data from InfluxDB at a rate of every 5 seconds, this is too slow for live monitoring of a dynamic system. Instead, live telemetry is pulled directly from the MQTT broker.

To begin, download and install Grafana on the server. The default port for Grafana is `3000`. Once installed, open up a web browser and navigate to `http://localhost:3000/` to access the Grafana dashboard. The default login is `admin` with the password `admin`.

Adding Data Sources

Grafana needs to be configured with data sources:

1. **InfluxDB Data Source**:
   - The InfluxDB data source is included by default as a plugin.
   - Specify the correct query language and provide the necessary credentials and address (`http://localhost:8086`).
2. **MQTT Data Source**:
   - To view MQTT data in real time, install a plugin to connect Grafana to the Mosquitto broker: MQTT Datasource Plugin.
   - In Grafana, navigate to **Configuration** > **Data Sources** > **Add data source**.
   - Select MQTT from the list of available data sources.

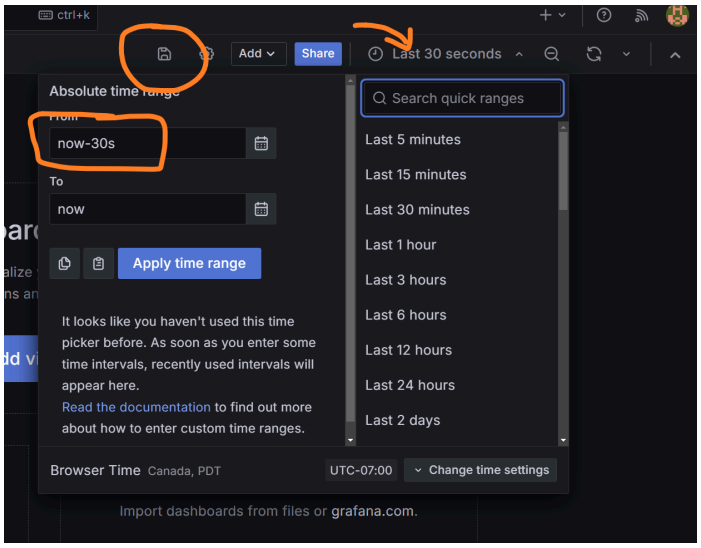
- Name the data source and specify the connection to the MQTT broker (`tcp://localhost:1883`).
- Add a username and password if configured for the broker.

Creating Dashboards and Panels

Now it is time to setup a dashboard, a collection of data panels. The Grafana interface is user friendly, but we are interested in some key settings.

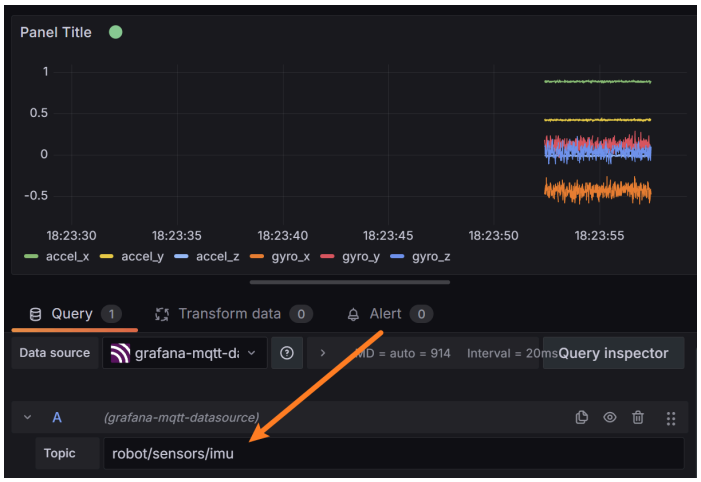
1. The window of time that is being displayed in the dashboard.
2. The MQTT topics that are being fetched for display.
3. The keys from the JSON that are being displayed.

For this stage, it is recommended to either have a robot sensor streaming data, or a surrogate Python script sending out data to the MQTT broker so that there are live streaming messages to display.



Step 1: Create a New Dashboard

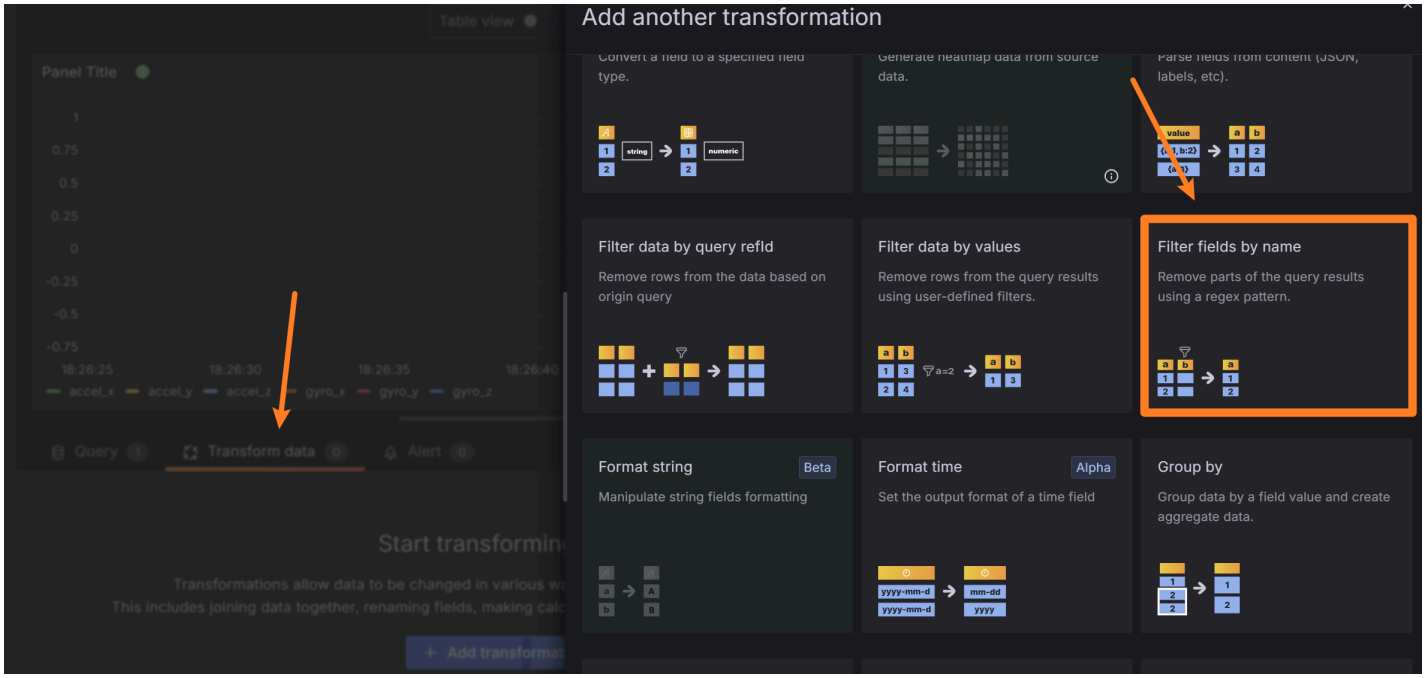
- Log into the Grafana homepage from any device connected to the private network using the server IP: `http://172.22.1.1:3000/`.
- Navigate to **Dashboards** and select **New Dashboard**.
- Save the new dashboard. Remember to save periodically to avoid losing changes.
- Adjust the time range of the dashboard to the last 30 seconds to see recent data streaming in. Apply the time range.



Step 2: Add a Visualization Panel

- Select the **Add a visualization** button to make the first panel.
- Enter the MQTT topic and verify that data is streaming using MQTT Explorer.
- The data should begin streaming in the panel preview.
- Use the **Query Inspector** and **Data** tab to verify that data is being received and processed correctly if the visualization is not showing as expected.

Step 3: Customize the Panel



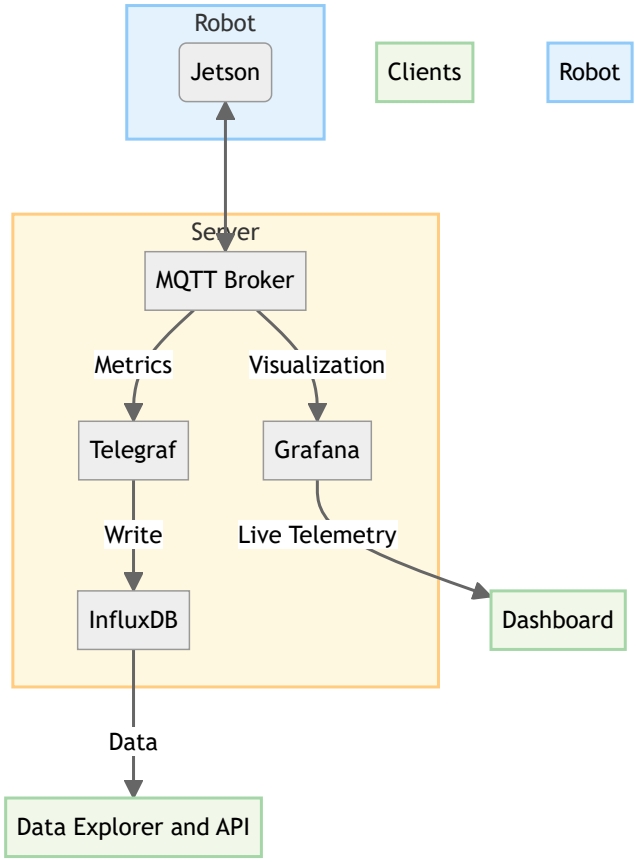
Select the **Transform Data** tab and then **Filter fields by name** option. Fields that are to be omitted can be removed from the identifier list and will not be displayed in the panel. Finally the right hand side of the panel configuration can be used to fully customize the display of data, panel title, etc.

Save and apply the panel change. Now is a good time to bookmark the dashboard for easy access in the future. The live telemetry has limitations in how much data can be displayed at once, since it is sampling from a moving buffer and not storing the data like the database. Too many panels with too much data will cause the system to stutter, so the recommendation is to downsample the data to about 10Hz or less unless full sample resolution is needed.

Notes on Downsampling

As of now, an effective way to downsample the incoming data in Grafana has not been found. Telegraf has some data processing capabilities for downsampling, but it would require rebroadcasting over a new MQTT topic. The simplest solution we have found is to downsample data on the robot side by sending full-resolution data to each of the `robot/` topics and every n th message to a `downsampled/` topic. This can be implemented with few lines of code and does not add significant overhead.

At this stage in the setup, the system architecture is as follows:



Robot Software Architecture

An automated data pipeline now exists between the robot software system and the database and telemetry panel. However, there are considerations when handling data within the robot controller, especially since the control loop is sensitive to timing and delays.

To address these issues, the telemetry handling and output of the robot are separated into a parallel process that is not part of the control loop. Crucially, the process has lower priority and will not interfere or cause delays in the time-sensitive portions of the code.

We follow two core principles in our software implementation:

1. Data-driven design: The data classes within the robot system dictate the structure of the database and communication topics. IMU data is naturally packaged into a single topic, as is motor data, etc. This approach simplifies data management and ensures correct data processing.

2. Asynchronous processing: The telemetry process runs on a separate core from the main control loop. In Python, this is achieved using multiprocessing since the Python Global Interpreter Lock (GIL) prevents a single process from running on multiple cores. The two processes communicate through two unidirectional queues: one for data out and one for commands in.

The details of the implementation can be viewed through our repository: **RLUnicycle**. The multiprocessing with queues are initiated in `main.py`, data packaging is handled by `teledata.py`, and the communication thread is handled by `mqtt.py`.

`main.py`

The main entry point for the robot handles the asynchronous processes using data queues for input and output to send messages between the control loop and communications:

Code Example: Main entry point for the robot handling asynchronous processes.

▶ Code

`robot/teledata.py`

The `@dataclass` decorator in Python is used to define structures that hold the data in fields. A method to convert them to dictionary/JSON is also included in a TelemetryData base class. A generalist debug data class is also provided to allow for ad-hoc data to be sent to the telemetry system during development that has not been assigned a specific topic. The fields are coupled with the MQTT topic and the database measurement name within the dataclass object.

A sample of the abstract class and a data class for IMU data is shown below:

Code Example: A sample of the abstract class and a data class for IMU data.

▶ Code

`robot/rwip.py`

At the end of a control cycle, all data classes are packed into a single list and sent to the telemetry process. Opening the inter-process queue incurs an overhead, so it is more efficient to send one single outgoing packet per control cycle.

Code Example: Packaging of TelemetryData objects into a list and sending them to the telemetry process.

▶ Code

Telemetry Process `communication/mqtt.py`

The telemetry process is essentially a wrapper around the existing Paho MQTT client. It is designed to unpack the data classes from the incoming queue and send them to their corresponding topics. It additionally handles downsampling the data to a lower rate for the Grafana panels. It has a standby loop that listens for incoming commands over a control topic and routes them to the control cycle through the command queue as needed.

Code Example: Parsing of TelemetryData objects and sending them to the MQTT broker.

▶ Show the code

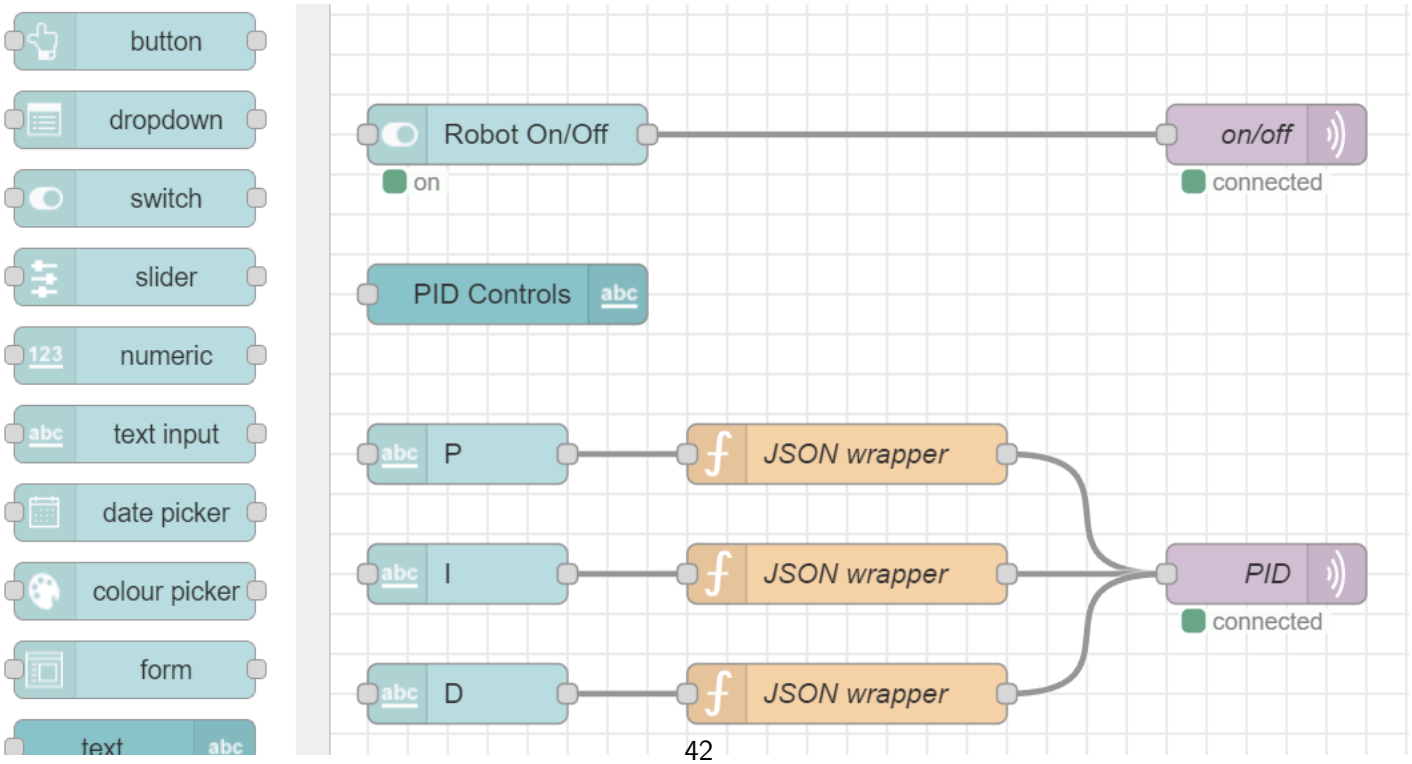
Command Management

The established connection with the robot controller also allows for remote commands to be sent in real-time. These commands can adjust tuning parameters or control the robot. We have a control panel that is synchronized across all devices through a web browser using Node-RED.

Node Red

NodeRed is a flow-based, open-source development tool for visual programming developed by IBM. It is used for wiring together hardware devices, APIs, and online services in new and interesting ways. It provides a browser-based editor that makes it easy to wire together flows using a wide range of nodes. In our case, this enables a synchronized control panel accessible through a web browser.

NodeRed can be installed on the server, with default port of `1880`.



Node Red Dashboard

The Node-RED dashboard allows for the creation of custom dashboards that provide a GUI for the robotics project.

XBox Controller

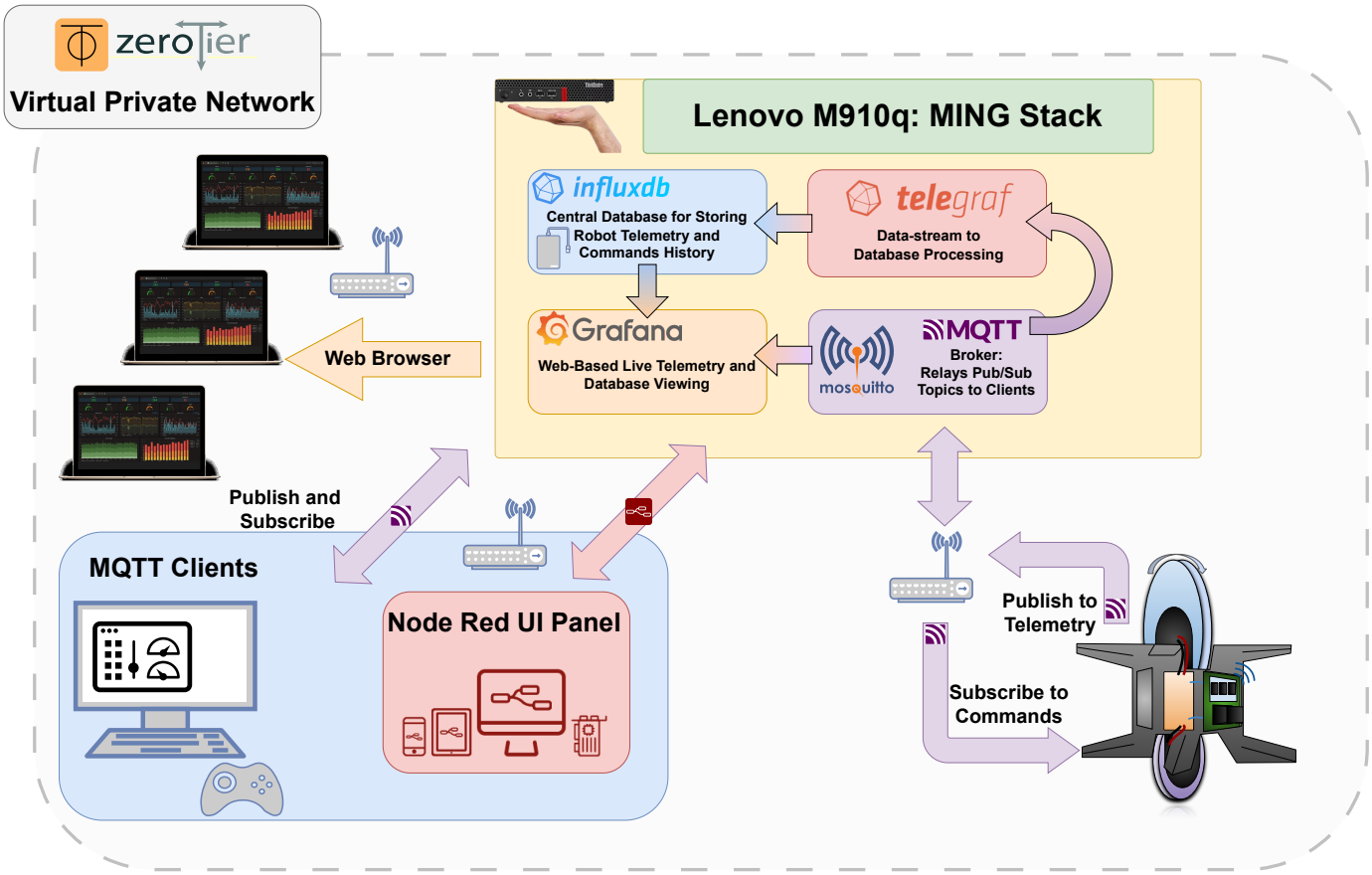
A controller can be connected through a PC to an MQTT topic using Paho-MQTT and Python. This opens up some interesting avenues for assisted control of the robot.

Conclusion

The setup of the Lenovo server, ZeroTier network, MQTT broker, Telegraf, InfluxDB, Grafana, and Node-RED provides a powerful platform for the development of a capstone project. These services are all open-source and free to use. They are well-documented and have large communities of users that can assist with any issues that may arise.

By following this guide, you should have a robust telemetry and database system for your robotics project.

At this point in the setup, if you have followed the guide, the system architecture is as follows:



Telemetry Overview

Best of luck with your robotics project!

Appendix B

Stand-up Calculations

A numerical analysis of stand-up torque requirements

standup_calculations

April 14, 2024

[27]:
```
import numpy as np
from scipy.integrate import odeint
import matplotlib
import matplotlib.pyplot as plt
```

[28]:
```
class robot:
  def __init__(self) -> None:

    # Mass Definitions
    self.m_reaction_wheel = 0.2
    self.m_motor = 0.18
    self.m_yaw_motor = 0.046
    self.m_yaw_reaction_wheel = 0.1
    self.m_chassis = 0.3 + 0.3
    self.m_batteries = 0.185
    self.m_microcontroller = 0.24

    self.total_mass = 2 * self.m_reaction_wheel + 2 * self.m_motor + self.
 ↪m_yaw_motor + self.m_yaw_reaction_wheel + self.m_chassis + 4 * self.
 ↪m_batteries + self.m_microcontroller - 0.07

    ### TESTING 1  vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
    self.total_mass = self.m_reaction_wheel + self.m_motor + 0.546 + .3

    ### END TESTING 1 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

    self.m_robot_body = self.total_mass - 2 * self.m_reaction_wheel # Note:␣
 ↪Including yaw wheel in robot body for now

    # Motor Parameters
    self.max_rpm = 3000 # rpm
    self.max_rpm_rad = self.max_rpm * (2 *np.pi) / 60 # rad/s
    self.stall_torque = 1.4 # Nm

    # Dimension Parameters
    self.wheel_diameter = 0.095 # 10cm
    self.wheel_radius = self.wheel_diameter / 2
```

```
    self.robot_height = 0.14 # Length of cylinder (assume COM in middle for now)

    ### TESTING 2  vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
    self.robot_height = 0.18

    ### END TESTING 2 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

    self.robot_diameter = 0.1
    self.robot_radius = self.robot_diameter / 2

    ## Note currently using wheelbots calculated wheel moments of inertia for␣
↪rxn wheels (can be replaced easily) http://hyperphysics.phy-astr.gsu.edu/
↪hbase/icyl.html
    # Rolling wheel inertias
    self.I1x = 0.000259 # pointing in drive direction
    self.I1y = 0.000503 # Rotational DOF inertia in kg * m2
    self.I1z = 0.000259

    # Reaction wheel Inertias
    self.I3x = 0.000503 # Rotational DOF Inertia
    self.I3y = 0.000259
    self.I3z = 0.000259 # Pointing'up'

    # Calculate robot body Intertia (about COM)
    self.Ibx = 1/4 * self.m_robot_body*self.robot_radius**2 + 1/12 * self.
↪m_robot_body * self.robot_height**2
    self.Iby = self.Ibx
    self.Ibz = 1/2 * self.m_robot_body * self.robot_radius**2

  def robot_ode(self, y, t, I_tot, I_rxn_wheel, d_to_cog, m_tot):
    theta, d_theta, omega = y

    T0 = self.stall_torque # Stall torque
    omega0 = self.max_rpm_rad # Max motor rate under load 282rad/s = 2700 RPM
    T=T0

    Mg = m_tot * 9.81 * d_to_cog * np.sin(theta)
    dd_theta = (Mg - T) / I_tot
    d_omega = T / I_rxn_wheel

    dydt = [d_theta, dd_theta, d_omega]
    return dydt

  def yaw_ode(self, y, t, torque_func, I_tot, I_rxn_wheel, m_tot, mu_static,␣
↪mu_dynamic, patch_radius):
      '''
```

```
    Differential equation for yaw motion considering friction.

    Inputs:
    y - current state [theta, d_theta, omega]
    t - current time
    torque_func - function to calculate torque as a function of time
    I_tot - total moment of inertia for robot
    I_rxn_wheel - reaction wheel moment of inertia
    m_tot - total mass of robot
    mu_static - static torsional friction coefficient
    mu_dynamic - dynamic friction coefficient
    patch_radius - estimated radius of contact patch with the ground
    '''

    theta, d_theta, omega = y  # Unpack the current state

    # Calculate normal force (approximation: weight of the robot)
    N = m_tot * 9.81

    # Static and dynamic frictional torques
    tau_static = mu_static * N * patch_radius
    tau_dynamic = mu_dynamic * N * patch_radius

    # Get torque input as a function of time
    T = torque_func(t)

    # Determine the direction of frictional force
    friction_direction = np.sign(d_theta)

    # Check if static friction is overcome
    if abs(d_theta) <= 0.002:
      if abs(T) < abs(tau_static):
        # Must overcome friction to move
        dd_theta = 0
      else:
        # Static friction not overcome, robot does not start rotating
        dd_theta = (T-tau_static*friction_direction)
    else:
      # Static friction overcome, robot starts rotating
      # Net torque on the robot (subtract dynamic frictional torque)
      net_torque = T - tau_dynamic * friction_direction
      # Angular acceleration of the robot
      dd_theta = net_torque / I_tot

    # Reaction wheel dynamics
    d_omega = T / I_rxn_wheel
```

```
    dydt = [d_theta, dd_theta, d_omega]
    return dydt
```

[29]:
```
r = robot()
print("Total Mass", r.total_mass)
print("Moments of Inertia", r.I1x, r.I3x, r.Ibx)
print("Robot Height", r.robot_height)
print("Stall Torque", r.stall_torque)
```

```
Total Mass 1.226
Moments of Inertia 0.000259 0.000503 0.00274645
Robot Height 0.18
Stall Torque 1.4
```

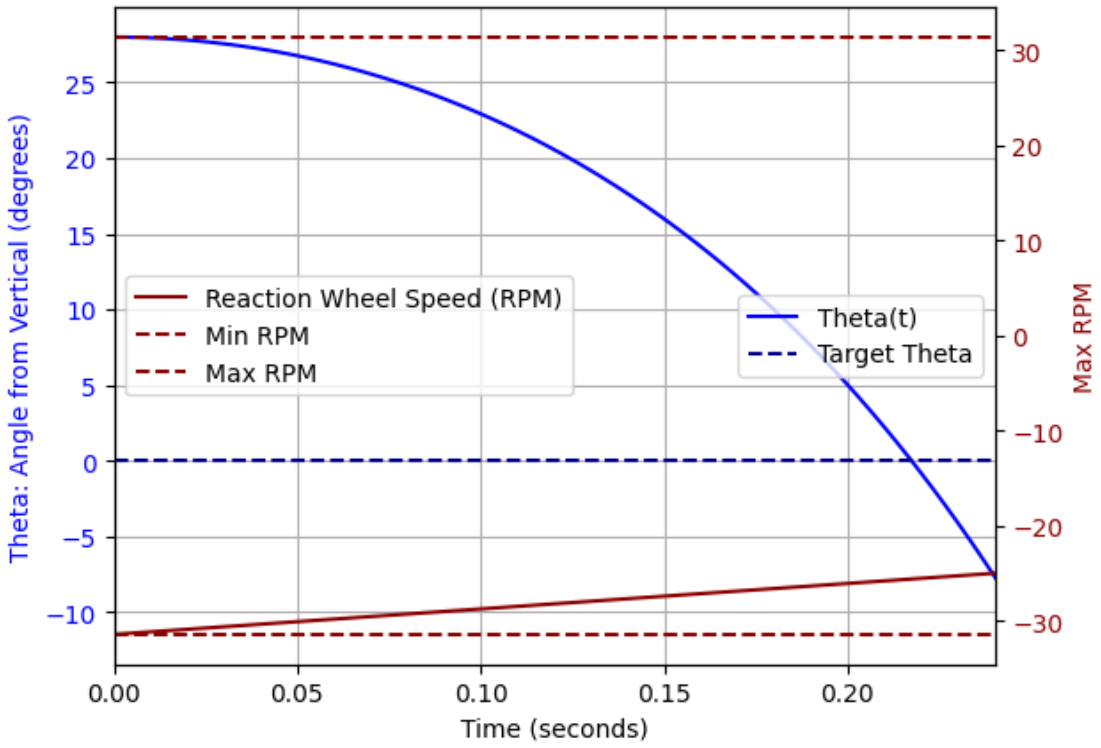
[30]:
```
# Cylinder standing on side, parallel axis down to contact point
# m * d**2 + i_0

# Also calculate I for wheel and shift down

# Standup w point of rotation edge of roll wheel

# parallel axis theorem
s_rollw = r.m_reaction_wheel * r.wheel_radius**2
s_rxnw = r.m_reaction_wheel * (r.robot_height + r.wheel_radius)**2
s_body = r.m_robot_body * (r.robot_height/2 + r.wheel_radius)**2

### TESTING 3  vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
r.I1x = 0.000501
r.I3x = 0.005224
r.Ibx = 0.000401

### END TESTING 3 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

I_total = r.I1x + r.I3x + r.Ibx + (s_rollw + s_rxnw + s_body)
```

[31]:
```
# Define initial parameters of ode
d_cog = r.robot_height / 3 + r.wheel_radius # Distance to contact point from cog

### TESTING 4  vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
d_cog = 0.15

### END TESTING 4 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

# Note: the angle is measured from vertical to z-axis of robot
initial_angle = np.deg2rad(28) # In radians
final_angle = 0
```

```
y0 = [initial_angle, 0, -r.max_rpm_rad]
t = np.linspace(0, 0.25, 100)
```

[32]:
```
ode_soln = odeint(r.robot_ode, y0=y0, t=t, args=(I_total, r.I3x, d_cog, r.
 ↪total_mass))
```

[33]:
```
plt.figure()

# Create first y-axis for theta
ax1 = plt.gca()
ax1.plot(t, np.rad2deg(ode_soln[:, 0]), 'b', label='Theta(t)')
ax1.axhline(y=0.0, color='darkblue', linestyle='--', label='Target Theta')
ax1.set_xlabel('Time (seconds)')
ax1.set_ylabel('Theta: Angle from Vertical (degrees)', color='b')
ax1.tick_params(axis='y', labelcolor='b')

# Create second y-axis for RPM
ax2 = ax1.twinx()
ax2.plot(t, 0.1*ode_soln[:, 2], 'darkred', label='Reaction Wheel Speed (RPM)')
ax2.axhline(y=0.1*-r.max_rpm_rad, color='darkred', linestyle='--', label='Min␣
 ↪RPM')
ax2.axhline(y=0.1*r.max_rpm_rad, color='darkred', linestyle='--', label='Max␣
 ↪RPM')
ax2.set_ylabel('Max RPM', color='darkred')
ax2.tick_params(axis='y', labelcolor='darkred')

# Enhance layout and grid
plt.margins(x=0)
ax1.grid(True)

# Set legends for each axis with positioning on the left side inside the plot
ax1.legend(loc='center right')
ax2.legend(loc='center left')

# Limiting the x-axis range
x_min, x_max = 0, 0.24  # Define your limits here
ax1.set_xlim(x_min, x_max)

# Show plot
plt.show()
```



The yaw wheel moment and the robot net moment are required to analyse the acceleration of each respective body under a torque.

Working with the 46g motor listed at https://store.tmotor.com/goods-1150-IP35+MN2806+Antigravity+Type+4-6S+UAV+Motor+400650KV.html

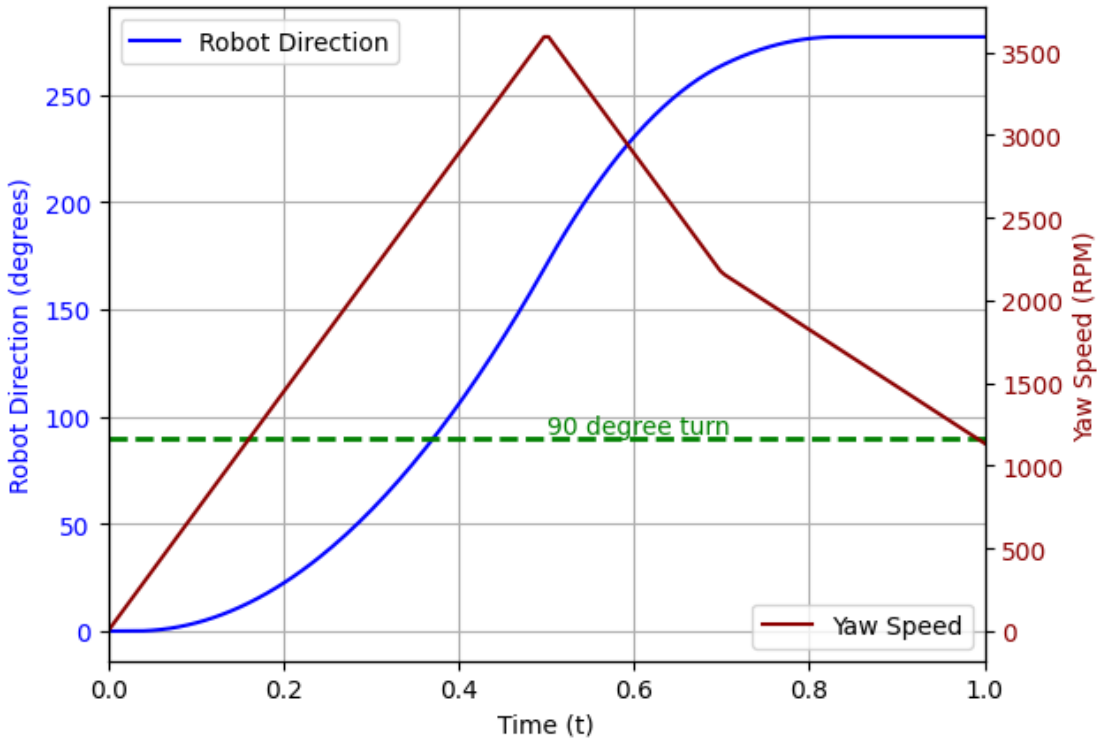
Dimensions

[34]:
```
# Calculations for Yaw wheel and requirements
# For simplicity we will assume a chunky cylinder.
M = r.total_mass - r.m_reaction_wheel # Remove the Yaw Wheel
I_tot = 1/2*M*(.075)**2               # This estimate uses robot diameter of␣
 ↪15cm

# Specify the motor mass and inertia
yaw_motor_inertia = .5*.025 * (.035/2)**2 # Estimate for the motor winding␣
 ↪inertia

# We mount ontop of the small motor, mass of ~50g. then 100g of material is␣
 ↪placed around outer ring
#Radius is ~ 20cm for the mass placement
```

```
yaw_radius = .030   # Reaction wheel radius
yaw_mass_addition = .15  # Reaction wheel mass (mass fixed at the outer radius)
yaw_inertial_mass = yaw_mass_addition*yaw_radius**2   # Yaw wheel inertia (mass␣
 ↪placed around edge)
Iyaw = yaw_motor_inertia + yaw_inertial_mass
r_patch = .005 #1cm diameter ground contact patch
mu_static = .65 # Guessing game
mu_dynamic = .45 # Guessing game

# robot angle, speed of spin, yaw wheel speed
y0 = [0, 0, 0]
t = np.linspace(0, 1, 200)

peak_torque = .105 # Peak torque is .22Nm

def torque_function(t, peak_torque=peak_torque, duration=0.5):
    """
    Returns a constant torque for a specified duration and then stops.

    :param t: Current time
    :param peak_torque: The peak torque value
    :param duration: Duration for which the torque is applied
    :return: Torque at time t
    """
    if t <= duration:
        return peak_torque
    elif t <= 1.4*duration:
        return -peak_torque
    else:
        return -.05

ode_soln = odeint(r.yaw_ode, y0=y0, t=t, args=(torque_function, I_tot, Iyaw, r.
 ↪total_mass, mu_static, mu_dynamic, r_patch))
```

[35]:
```
yaw_rpm = ode_soln[:, 2] * 60 / (2*np.pi)

# Create a figure and a set of subplots
fig, ax1 = plt.subplots()

# Plot the first data set (Robot Direction) on the first y-axis
ax1.plot(t, np.rad2deg(ode_soln[:, 0]), 'b', label='Robot Direction')
ax1.set_xlabel('Time (t)')
ax1.set_ylabel('Robot Direction (degrees)', color='b')
ax1.tick_params(axis='y', labelcolor='b')
ax1.margins(x=0)
ax1.grid()
```

```
# Create a second y-axis with the same x-axis
ax2 = ax1.twinx()

# Plot the second data set (Yaw Speed) on the second y-axis
ax2.plot(t, yaw_rpm, 'darkred', label='Yaw Speed')
ax2.set_ylabel('Yaw Speed (RPM)', color='darkred')
ax2.tick_params(axis='y', labelcolor='darkred')

# Adding legends
ax1.legend(loc='upper left')
ax2.legend(loc='lower right')

# Add a horizontal line at 90 degrees
ax1.axhline(y=90, color='green', linestyle='--', linewidth=2)

# Add a label for the turn degree line
x_position_for_label = max(t) * 0.5  # Adjust this as needed for your plot
ax1.text(x_position_for_label, 90, '90 degree turn',␣
 ↪verticalalignment='bottom', color='green')

# Show the plot
plt.show()

static_friction = M*9.81*mu_static
# Print the results
print(f"Mass of the robot (M): {M: .3f} kg")
print(f"Static Friction: {static_friction: .3f}N")  # N for Newtons
print("Mass of Motor:", 50, "g")
print("Mass of Reaction Wheel:", yaw_mass_addition*1000, "g")
print(f"Diameter of Reaction Wheel: {yaw_radius*2*100: .3f} cm")
print(f"Applied Torque: {.105: .3f}Nm")
```




```
Mass of the robot (M):  1.026 kg
Static Friction:  6.542N
Mass of Motor: 50 g
Mass of Reaction Wheel: 150.0 g
Diameter of Reaction Wheel:  6.000 cm
Applied Torque:  0.105Nm
```

Appendix C

Inverted Pendulum Dynamics

Deriving Broomy's system dynamics in the pitch axis

1-DOF Inverted Pendulum System Dynamics

Julian L

April 2024

1 Introduction

The following is a demonstration of the derivation for the equations of motion for a single degree of freedom classical inverted pendulum. The approach used is energy methods via the Lagrangian using classical mechanics.

An automated derivation sequence using MATLAB is presented, which allows for parsing the equations of motion for an arbitrary system such as a 4-DOF unicycle robot. The code for the auto-derivation has been tested by hand against known solutions in the literature, as explored by Zwiers (2015).

2 Problem Description

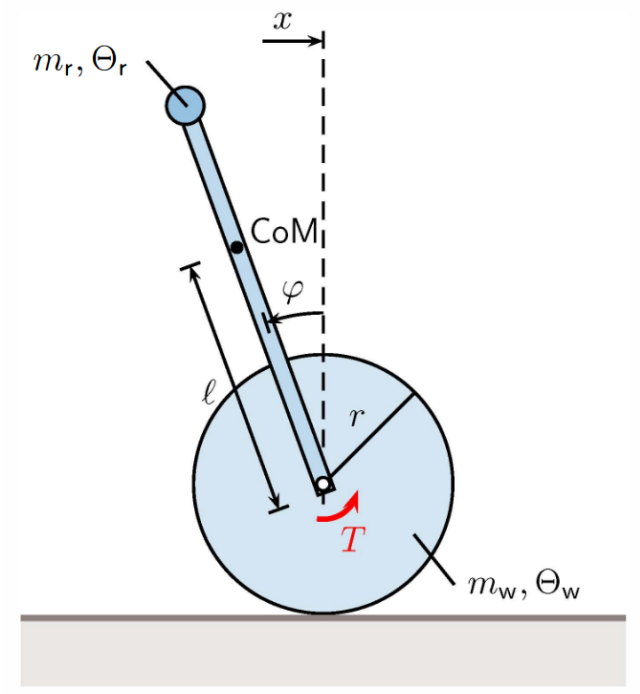


Figure 1: The dynamics of the classic inverted pendulum Zwiers (2015).

The inverted pendulum is attached at one end to a mass and at the other end to the center of the freely rotating wheel. The analysis of the problem involves two rigid bodies: the pendulum arm and the spinning drivewheel. The parameters of importance for each are

- l, the distance from the rotation center of the drivewheel to the center of mass of the body.
- I, the moment of inertia of the body about its centroid (Θ in Figure 1).
- m, the total mass of the body.

We use the subscript $_p$ for pendulum properties and $_w$ for wheel properties.

The properties in practice are determined experimentally or with the assistance of CAD.

Since we are restricted to the 2D plane in this single degree of freedom problem, there is only a single moment of inertia instead of a tensor like in the case of 3D.

We also specify our general coordinates for the problem. The angle of the drivewheel θ is an equally acceptable choice for a coordinate as the position x, since they are linearly correlated under $x = r\theta$. We will use θ for this analysis as the sensor readings in the final product use these coordinates. The angle ϕ is measured relative to the vertical, a configuration suitable for either an encoder or an IMU referencing gravity.

3 Parameter Measurement

3.1 Mass and Center of Mass Measurements

The mass and center of mass (CoM) were measured using a lab scale and a balancing method, respectively.

- **Drivewheel:** The wheel and rings mass (denoted as m_w) was measured to be 351g.
- **Pendulum and Motor:** The combined mass of the pendulum and motor with stator (denoted as m_p) was measured to be 1670g. The CoM of the pendulum with motor and stator (denoted as l_p) is 109mm. The pendulum CoM is found by balancing the apparatus with removed drivewheel overtop of a fulcrum and finding the stable resting point position.

3.2 Inertia Calculations

The moment of inertia for each component was calculated using the parallel axis theorem and the physical dimensions provided by CAD models and direct measurement.

3.2.1 Wheel Inertia

The wheel inertia (denoted as I_w) was found by comparing the CAD weight to the measured weight of the drivewheel to find agreement:

$$I_w = 768 \text{ kg} \cdot \text{mm}^2$$

In particular the metal rings were weighed and set to be the same weight in CAD which is the most influential part of the moment in question.

3.2.2 Pendulum Inertia

The pendulum moment of inertia (denoted as I_p) is a composite value derived from the inertia of individual components. Each was weighed, and it's respective part in CAD was set to have a uniform density corresponding to the measured weight.

The CAD model can then compute the inertia of the pendulum body about a given axis as the sum of the components:

$$I_p = 25403 \text{ kg} \cdot \text{mm}^2$$

4 Lagrangian Derivation

Our generalized coordinates are

$$\vec{q} = \begin{bmatrix} \varphi \\ \theta \end{bmatrix}, \quad \text{and} \quad \frac{d}{dt}\vec{q} = \dot{\vec{q}} = \begin{bmatrix} \dot{\varphi} \\ \dot{\theta} \end{bmatrix}$$

We derive the kinetic and potential energy of the system first:

4.1 Kinetic Energy

$$\begin{aligned}
T &= T_p + T_w \\
T_p &= \frac{1}{2}(\underbrace{I_p + m_p l_p^2}_{\text{Parallel Axis Theorem}})\dot{\varphi}^2 + \frac{1}{2}m_p(\underbrace{r^2\dot{\theta}^2 + l^2\dot{\varphi}^2 + 2rl\dot{\theta}\dot{\varphi}\cos\varphi}_{\text{Speed and rotation of CoM}}) \\
T_w &= \frac{1}{2}m_w(\underbrace{r\dot{\theta}}_{\text{CoM speed}})^2 + \frac{1}{2}I_w\dot{\theta}^2 \\
T_{net} &= \frac{1}{2}m_p\left(r^2\dot{\theta}^2 + l^2\dot{\varphi}^2 + 2rl\dot{\theta}\dot{\varphi}\cos\varphi\right) + \frac{1}{2}I_p\dot{\theta}^2 + \frac{1}{2}m_r r^2\dot{\theta}^2 + \frac{1}{2}I_w\dot{\theta}^2 \\
&= \frac{1}{2}\big(m_p\, r^2 + m_w r^2 + I_w\big)\dot{\theta}^2 + \frac{1}{2}\big(m_p l^2 + I_p\big)\dot{\varphi}^2 + m_p lr\cos\varphi\dot{\varphi}\,\dot{\theta} \\
T_{net} &= \frac{1}{2}[\dot{\varphi}, \dot{\theta}]\begin{bmatrix} m_p r^2 + m_w r^2 + I_w & m_p rl \\ m_p rl & m_p l^2 + I_p \end{bmatrix}\begin{bmatrix} \dot{\varphi} \\ \dot{\theta} \end{bmatrix}
\end{aligned}$$

This gives the form using the inertia matrix M, note the matrix is always symmetric.

4.2 Potential Energy

The potential energy is taken by projecting the position of the center of mass of the pendulum onto the vertical axis using $\cos(\varphi)$, noting that the drivewheel has no impact on the potential since the wheel is radially symmetric and always remains at the same height.

$$U = m_p lg\cos(\varphi)$$

This gives the complete Lagrangian

$$\mathcal{L}(\varphi, \theta, \dot{\varphi}, \dot{\theta}) = KE - PE = \frac{1}{2}\dot{\mathbf{q}}^T\mathbf{M}\dot{\mathbf{q}} - m_p lg\cos(\varphi)$$

5 Equations of Motion

The Euler-Lagrange equations for each coordinate will inform the equations of motion:

$$0 = \underbrace{\frac{\partial \mathcal{L}(q, \dot{q}, t)}{\partial q} - \frac{d}{dt}\left(\frac{\partial \mathcal{L}(q, \dot{q}, t)}{\partial \dot{q}}\right)}_{\text{Euler-Lagrange Equation}}$$

The non-conservative force of the torque is incorporated by having the equation not sum to zero, but instead the sum of non-conservative forces/torques. The details of this style of derivation can be found in Brevik (2017) Section 3.3.2.

These equations can be derived by hand, but all of the necessary information for the problem is already encoded in the starting Lagrangian. All equations of motion that follow are merely an algorithmic process, one that is prone to errors as well. For efficiency, it is preferable to devise a method to automatically differentiate.

5.1 Matlab Derivation

The required files to run this code are included at https://github.com/Team-2411-RL-Unicycle/pid-control The automated E-L solver uses a modified version of a file made by Veng (2023). It is incorporated into the RWIPpid_derivation.m file. The derivation technique is validated against the equations derived by Brevik (2017).

The first step is to define symbolic variables for all of the parameters, states, and inputs (note that subscript $_r$ is used for "robot" in place of $_p$ for pendulum)

Listing 1: MATLAB Code

```
%Robot variables
syms mr mw Iw Ir r l real
params = [mr, mw, Iw, Ir, r, l];
values = [1.670, ... mr [kg]
    0.351, ... mw [kg]
    0.000768, ... Iw [kg * m^2]
    0.113/2, ... r [m]
    0.109, ... l [m]
    0.025403 ... Ir [kg * m^2]
    ]; % full robot
g=9.81;

%State variables
syms phi dphi psi dpsi real
q = [phi, psi]; % phi = body angle, psi = wheel angle
dq = [dphi, dpsi];

% Input
syms tau real

% Mass matrix
M = [((mr + mw)*r^2 + Iw), r*l*mr*cos(phi);
    r*l*mr*cos(phi), mr*l^2 + Ir];
```

```
lagrangian = (1/2)*([dphi, dpsi])*M*([dphi, dpsi]') - mr*g*l * cos(phi);

% Non-conservative forces in each coordinate q (should only affect drive
% wheel)
Q = [0, tau];
```

The Lagrangian and its non-conservative forces are fully defined now. The equations are solved using the modified imported library and the solution equations for each second time derivative is solved giving $\frac{d}{dt}\dot{q}$, these solutions can be packed into a single array to form a matrix.

Listing 2: MATLAB Code

```
% Derive the equations of motion for each ddq
[eqs, ddq] = EulerLagrange(q,dq,lagrangian,Q);
% Explicit equations:
exp_eqs = ddq == eqs;
% Solve equations to isolate ddphi and ddtheta
ddqSolutions = solve(ddq == eqs, ddq);
% Convert solutions to cell array
ddqSolutionEquations = struct2cell(ddqSolutions) ;
ddqArray = [ddqSolutionEquations{:}].';
```

5.2 Derived Equations of Motion

Once we have n 2nd order ODEs for n general coordinates and their n general time derivatives we have enough to make a first order system of ODEs that characterize the system. The time-domain non-linearized equation from the derivation is given below, though the full expansion of G is not shown.

$$\frac{d}{dt}\vec{x} = \vec{G}(\vec{x}, t), \quad x = \begin{bmatrix} \varphi \\ \theta \\ \dot{\varphi} \\ \dot{\theta} \end{bmatrix}$$

Note that there is no explicit time dependence in the function G the inverted pendulum dynamics and rigid body characteristics are constant over time. From inspection of the solutions we see that θ, the angle of the wheel does not play a role in the function G and can be removed entirely if desired.

These system dynamics can be used to create a time-domain non-linear simulation using Euler's method to get numerical solutions. Friction can be added as a damping coefficient β such that we superimpose $\ddot{\varphi} = -\beta\dot{\varphi}$ onto the solution for example.

6 Controls Derivation

Now that the system dynamics are recovered we want to work in the Laplace domain for control. To do so, we need to get a linearized form of this non linear vector equation. This is similar to defining a first-order approximation to a single variable function: $f(x) \approx f(x_0) + f'(x_0) \cdot (x_0 - x)$. In this case the linearation happens about a vector in state-space and we use the Jacobian as the multivariate generlization of the first derivative.

For more information on this process refer to

Berkley Designing Information Devices and Systems II University of California, Berkeley (2021)

Caltech Jacobian Linearization California Institute of Technology (2002)

6.1 Linearization

We wish to convert

$$\vec{G}(\vec{x}, t) \approx Ax + Bu$$

via linearization about the operating point. We choose the upright position as the target and note that φ is the only variable present in G. $\hat{\vec{x}} = 0$ is the chosen linearization point:

$$\frac{d}{dt}\vec{x} \approx \hat{\vec{x}} + \text{ Jacobian}\{\vec{G}(\vec{x}, t)\}\Big|_{\vec{x}=\hat{\vec{x}}}(\vec{x} - \hat{\vec{x}}) = \Big(A\Big)\Big|_{\vec{x}=\hat{\vec{x}}}\vec{x}$$

We perform a similar linearization to get the effect of the system inputs by taking the Jacobian with respect to τ. The two combined give the cannonical $\frac{d}{dt}x = Ax + Bu$ of controls engineering. The final step is to take the Laplace transform of the entire equation and then solve for the transfer function between the system inputs u or in this case τ and the observables we want (mainly the system state x) but this generalizes to any observable that is a function of x and u

State Vector

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \end{bmatrix}$$

Input Vector

$$\mathbf{u} = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \end{bmatrix}$$

Output Vector

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \end{bmatrix}$$

State Equation

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \vdots \end{bmatrix} = \mathbf{Ax} + \mathbf{Bu}$$

Output Equation

$$\mathbf{y} = \mathbf{Cx} + \mathbf{Du}$$

State Transition Matrix

$$\mathbf{\Phi} = (s\mathbf{I} - \mathbf{A})^{-1}$$

Transfer Functions

$$\frac{\mathbf{y}}{\mathbf{u}} = \mathbf{C\Phi B} + \mathbf{D}$$

We solve for the transfer matrix $y = Gu$ at $x = 0$, noting that in our case $y = x$

6.2 MATLAB Derivation

Listing 3: MATLAB Code

```
% State vector of the system, note that Theta is not a state variable
% Phi, dPhi, dTheta
X = [q(1)' ; dq']
% The inputs are non-zero entries of Q (non-conservative forces)
U = Q(Q ~= 0);
% Vector functionn for the derivative of the state vector
dX = [dphi; ddqArray]

% Compute the Jacobian matrices to get nonlinear state matrices dX = Ax + Bu
A = jacobian(dX, X);
B = jacobian(dX, U);

% Substitute or linearize about an equilibrium point
% Define equilibrium point (for example, all zeros)
x0 = [0; 0; 0];
% Substitute equilibrium values x0 into A and B
Aeq = subs(A, X, x0)
Beq = subs(B, X, x0)
```

7 Controls Policy

There are actually two objectives that are necessary to keep the pendulum balanced in place.

1. The angle φ should be minimized to 0 degrees where possible.
2. The drivewheel velocity should be minimized to 0 degrees where possible.

To compute a control policy, the numerical matrices solved for this problem were combined with the solutions from the RWIP problem into a single system matrix containing the two axes of balance. The control policy solution was computed using LQR. Please refer to the LQR Appendix for more details.

References

Brevik, P. (2017). *Two-axis reaction wheel inverted pendulum* (Master's thesis, Norwegian University of Science and Technology). Retrieved from https://ntnuopen.ntnu.no/ntnu-xmlui/bitstream/handle/11250/2451060/12762_FULLTEXT.pdf?sequence=1 (Supervisor: Tor Engebret Onshus)

California Institute of Technology. (2002). *Jacobian linearization.* CDS Caltech. Retrieved from https://www.cds.caltech.edu/~murray/courses/cds101/fa02/caltech/pph02-ch19-23.pdf (Accessed: 2024-04-06)

University of California, Berkeley. (2021). *Designing information devices and systems ii.* EECS Berkeley. Retrieved from https://inst.eecs.berkeley.edu/~ee16b/sp21/notes/sp21/note15.pdf (Accessed: 2024-04-06)

Veng, M. (2023). *Euler-Lagrange Solver.* MATLAB Central File Exchange. Retrieved 2023-10-23, from https://www.mathworks.com/matlabcentral/fileexchange/93275-euler-lagrange-solver (Available online: https://www.mathworks.com/matlabcentral/fileexchange/93275-euler-lagrange-solver)

Zwiers, U. (2015, Nov). Modeling of wheeled inverted pendulum systems. *2015 16th International Conference on Research and Education in Mechatronics (REM)*, 273–277. doi: 10.1109/rem.2015.7380406

Appendix D

Reaction Wheel Inverted Pendulum Dynamics

Deriving Broomy's system dynamics in the roll axis

1-DOF System Dynamics and Classical Controls

Simon G

April 2024

1 Introduction

The following is a demonstration of the derivation for the equations of motion for a single degree of freedom reaction wheel inverted pendulum. The approach used is energy methods via the Lagrangian using classical mechanics.

An automated derivation sequence using MATLAB is presented, which allows for parsing the equations of motion for an arbitrary system such as a 4-DOF unicycle robot. The code for the auto-derivation has been tested by hand against known solutions in the literature, as explored by Brevik (2017), Montoya and Gil-González (2020).

2 Problem Description

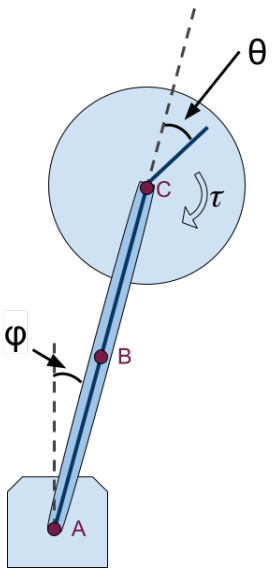


Figure 1: The dynamics of the reaction wheel pendulum.

The inverted pendulum is attached to a hinge point at A and has a flywheel mounted to a motor at C. The analysis of the problem involves two rigid bodies: the pendulum arm and the spinning flywheel. The parameters of importance for each are

- l, the distance from the rotation point A to the center of mass of the body.
- I, the moment of inertia of the body about its centroid.
- m, the total mass of the body.

We use the subscript $_p$ for pendulum properties and $_w$ for wheel properties. $l_p = \bar{AB}$, $l_m = \bar{AC}$

The properties in practice are determined experimentally or with the assistance of CAD.

Since we are restricted to the 2D plane in this single degree of freedom problem, there is only a single moment of inertia instead of a tensor like in the case of 3D.

We also specify our general coordinates for the problem. Importantly the angle θ is chosen relative to the pendulum arm since this is what is measurable by an encoder on the joint. The angle ϕ is measured relative to the vertical, a configuration suitable for either an encoder at A or an IMU referencing gravity.

3 Parameter Measurement

3.1 Mass and Center of Mass Measurements

The mass and center of mass (CM) were measured using a lab scale and a balancing method, respectively.

- **Flywheel:** The wheel and rings mass (denoted as m_w) was measured to be 346g. The CM of the wheel from the pendulum hinge (denoted as l_w) is 180mm. This was measured in CAD and also with a ruler.
- **Pendulum and Motor:** The combined mass of the pendulum and motor with stator (denoted as m_p) was measured to be 531g. The CM of the pendulum with motor and stator (denoted as l_p) is 100mm. The pendulum CM is found by balancing the apparatus with removed flywheel overtop of a fulcrum and finding the stable resting point position.

3.2 Inertia Calculations

The moment of inertia for each component was calculated using the parallel axis theorem and the physical dimensions provided by CAD models and direct measurement.

3.2.1 Wheel Inertia

The wheel inertia (denoted as I_w) was found by comparing the CAD weight to the measured weight of the flywheel to find agreement:

$$I_w = 725 \text{ kg} \cdot \text{mm}^2$$

In particular the metal rings were weighed and set to be the same weight in CAD which is the most influential part of the moment in question.

3.2.2 Pendulum Inertia

The pendulum moment of inertia (denoted as I_p) is a composite value derived from the inertia of individual components:

1. **Battery:** The battery contributes an inertia of:

$$I_{\text{battery}} = \frac{1}{12} \cdot 0.185 \cdot (70^2 + 35^2) + 0.185 \cdot 50^2 = 446 \text{ kg} \cdot \text{mm}^2$$

2. **Pendulum Arm:** The corrected inertia for the pendulum arm is:

$$I_{\text{arm}} = 346 \text{ kg} \cdot \text{mm}^2 + 0.102 \cdot 45^2 = 552 \text{ kg} \cdot \text{mm}^2$$

3. **Motor and Mount:** The combined inertia for the motor and mount is:

$$I_{\text{motor}} = 0.5 \cdot 0.206 \cdot 30^2 + 0.206 \cdot 75^2 = 1251.75 \text{ kg} \cdot \text{mm}^2$$

The total pendulum inertia is then calculated as the sum of the components:

$$I_p = I_{\text{battery}} + I_{\text{arm}} + I_{\text{motor}} = 2250 \text{ kg} \cdot \text{mm}^2$$

4 Lagrangian Derivation

Our generalized coordinates are

$$\vec{q} = \begin{bmatrix} \varphi \\ \theta \end{bmatrix}, \quad \text{and} \quad \frac{d}{dt}\vec{q} = \dot{\vec{q}} = \begin{bmatrix} \dot{\varphi} \\ \dot{\theta} \end{bmatrix}$$

We derive the kinetic and potential energy of the system first:

4.1 Kinetic Energy

$$\begin{aligned}
T &= T_p + T_w \\
T_p &= \frac{1}{2}(\quad \underbrace{I_p + m_p l_p^2}_{\text{Parallel Axis Theorem}} \quad)\dot{\varphi}^2 \\
T_w &= \frac{1}{2} m_w (\quad \underbrace{l_w \dot{\varphi}}_{\text{Speed of CM}} \quad)^2 + \frac{1}{2} I_w (\quad \underbrace{\dot{\varphi} + \dot{\theta}}_{\text{net rotation earth frame}} \quad)^2 \\
T_{net} &= \frac{1}{2} \left(I_p + m_p l_p^2 + I_w + m_w l_w^2 \right) \dot{\varphi}^2 + \frac{1}{2} I_w (\dot{\varphi} + \dot{\theta})^2 \\
&= \frac{1}{2} \left(I_p + m_p l_p^2 \right) \dot{\varphi}^2 + \frac{1}{2} I_w \left(\dot{\varphi}^2 + 2\dot{\varphi}\dot{\theta} + \dot{\theta}^2 \right) \\
T_{net} &= \frac{1}{2} [\dot{\varphi}, \dot{\theta}] \begin{bmatrix} I_p + m_p l_p^2 + I_w + m_w l_w^2 & I_w \\ I_w & I_w \end{bmatrix} \begin{bmatrix} \dot{\varphi} \\ \dot{\theta} \end{bmatrix}
\end{aligned}$$

This gives the form using the inertia matrix M, note the matrix is always symmetric.

4.2 Potential Energy

The potential energy is taken by projecting the position of the center of masses onto the vertical axis using $\cos(\varphi)$, noting that the angle θ has no impact on the potential since the wheel is radially symmetric.

$$U = (m_p l_p + m_w l_w) g \cos(\varphi) = m_0 \cos(\varphi)$$

We can simplify future equations by assigning an equivalent variable $m_0 = (m_p l_p + m_w l_w) g$

This gives the complete Lagrangian

$$\mathcal{L}(\varphi, \theta, \dot{\varphi}, \dot{\theta}) = KE - PE = \frac{1}{2} \dot{\mathbf{q}}^T \mathbf{M} \dot{\mathbf{q}} - m_0 cos(\varphi)$$

5 Equations of Motion

The Euler-Lagrange equations for each coordinate will inform the equations of motion:

$$0 = \underbrace{\frac{\partial \mathcal{L}(q, \dot{q}, t)}{\partial q} - \frac{d}{dt}\left(\frac{\partial \mathcal{L}(q, \dot{q}, t)}{\partial \dot{q}}\right)}_{\text{Euler-Lagrange Equation}}$$

The non-conservative force of the torque is incorporated by having the equation not sum to zero, but instead the sum of non-conservative forces/torques. The details of this style of derivation can be found in Brevik (2017) Section 3.3.2.

These equations can be derived by hand, but all of the necessary information for the problem is already encoded in the starting Lagrangian. All equations of motion that follow are merely an algorithmic process, one that is prone to errors as well. For efficiency, it is preferable to devise a method to automatically differentiate.

5.1 Matlab Derivation

The required files to run this code are included at https://github.com/Team-2411-RL-Unicycle/pid-control The automated E-L solver uses a modified version of a file made by Veng (2023). It is incorporated into the RWIPpid_derivation.m file. The derivation technique is validated against the equations derived by Brevik (2017).

The first step is to define symbolic variables for all of the parameters, states, and inputs

Listing 1: MATLAB Code

```
% Robot variables
syms mp lp Ip mw lw Iw real
params = [mp, lp, Ip, mw, lw, Iw];
% Define numerical values for the parameters
values = [.531, 0.100, 0.002250, .346, 0.180, 0.000725];
g=9.81;
% State variables
syms phi theta dphi dtheta real
q = [phi, theta];
dq = [dphi, dtheta];
```

```
% Input
syms tau real

% Potential energy mass
m0 = (mp*lp + mw*lw)*g; % Effective U=mgh for combined parts
% Mass matrix
M = [(Ip + mp*lp^2 + Iw +mw*lw^2), Iw;
    Iw, Iw];
lagrangian = (1/2)*([dphi, dtheta])*M*([dphi, dtheta]') - m0 * cos(phi);
% Non-conservative forces in each coordinate q
Q = [0, tau];
```

The Lagrangian and its non-conservative forces are fully defined now. The equations are solved using the modified imported library and the solution equations for each second time derivative is solved giving $\frac{d}{dt}\dot{q}$, these solutions can be packed into a single array to form a matrix.

Listing 2: MATLAB Code

```
% Derive the equations of motion for each ddq
[eqs, ddq] = EulerLagrange(q,dq,lagrangian,Q);
% Explicit equations:
exp_eqs = ddq == eqs;
% Solve equations to isolate ddphi and ddtheta
ddqSolutions = solve(ddq == eqs, ddq);
% Convert solutions to cell array
ddqSolutionEquations = struct2cell(ddqSolutions) ;
ddqArray = [ddqSolutionEquations{:}].';
```

5.2 Derived Equations of Motion

Once we have n 2nd order ODEs for n general coordinates and their n general time derivatives we have enough to make a first order system of ODEs that characterize the system. The time-domain non-linearized result from the derivation is given below.

$$\frac{d}{dt}\vec{x} = \vec{G}(\vec{x},t) = \begin{bmatrix} d\varphi \\ d\theta \\ \frac{g_0 l_p m_p \sin(\phi) - \tau + g_0 l_w m_w \sin(\phi)}{m_p l_p^2 + m_w l_w^2 + I_p} \\ \frac{m_p \tau l_p^2 - I_w g_0 m_p \sin(\phi) l_p + m_w \tau l_w^2 - I_w g_0 m_w \sin(\phi) l_w + I_p \tau + I_w \tau}{I_w (m_p l_p^2 + m_w l_w^2 + I_p)} \end{bmatrix}, \quad x = \begin{bmatrix} \varphi \\ \theta \\ \dot{\varphi} \\ \dot{\theta} \end{bmatrix}$$

Note that there is no explicit time dependence in the function G the inverted pendulum dynamics and rigid body characteristics are constant over time. From inspection of the solutions we see that θ, the angle of the wheel does not play a role in the function G and can be removed entirely if desired.

These system dynamics can be used to create a time-domain non-linear simulation using Euler's method to get numerical solutions. Friction can be added as a damping coefficient β such that we superimpose $\ddot{\varphi} = -\beta\dot{\varphi}$ onto the solution for example.

6 Controls Derivation

Now that the system dynamics are recovered we want to work in the Laplace domain for control. To do so, we need to get a linearized form of this non linear vector equation. This is similar to defining a first-order approximation to a single variable function: $f(x) \approx f(x_0) + f'(x_0) \cdot (x_0 - x)$. In this case the linearation happens about a vector in state-space and we use the Jacobian as the multivariate generlization of the first derivative.

For more information on this process refer to

Berkley Designing Information Devices and Systems II University of California, Berkeley (2021)

Caltech Jacobian Linearization California Institute of Technology (2002)

6.1 Linearization

We wish to convert

$$\vec{G}(\vec{x}, t) \approx Ax + Bu$$

via linearization about the operating point. We choose the upright position as the target and note that φ is the only variable present in G. $\hat{\vec{x}} = 0$ is the chosen linearization point:

$$\frac{d}{dt}\vec{x} \approx \hat{\vec{x}} + \text{ Jacobian}\{\vec{G}(\vec{x}, t)\}\Big|_{\vec{x}=\hat{\vec{x}}} (\vec{x} - \hat{\vec{x}}) = \Big(A\Big)\Big|_{\vec{x}=\hat{\vec{x}}} \vec{x}$$

We perform a similar linearization to get the effect of the system inputs by taking the Jacobian with respect to τ. The two combined give the cannonical $\frac{d}{dt}x = Ax + Bu$ of controls engineering. The final step is to take the Laplace transform of the entire equation and then solve for the transfer function between the system inputs u or in this case τ and the observables we want (mainly the system state x) but this generalizes to any observable that is a function of x and u

State Vector

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \end{bmatrix}$$

Input Vector

$$\mathbf{u} = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \end{bmatrix}$$

Output Vector

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \end{bmatrix}$$

State Equation

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \vdots \end{bmatrix} = \mathbf{Ax} + \mathbf{Bu}$$

Output Equation

$$\mathbf{y} = \mathbf{Cx} + \mathbf{Du}$$

State Transition Matrix

$$\mathbf{\Phi} = (s\mathbf{I} - \mathbf{A})^{-1}$$

Transfer Functions

$$\frac{\mathbf{y}}{\mathbf{u}} = \mathbf{C\Phi B} + \mathbf{D}$$

We solve for the transfer matrix $y = Gu$ at $x = 0$, noting that in our case $y = x$

6.2 MATLAB Derivation

Listing 3: MATLAB Code

```
% State vector of the system, note that Theta is not a state variable
% Phi, dPhi, dTheta
X = [q(1)' ; dq']
% The inputs are non-zero entries of Q (non-conservative forces)
U = Q(Q ~= 0);
% Vector functionn for the derivative of the state vector
dX = [dphi; ddqArray]

% Compute the Jacobian matrices to get nonlinear state matrices dX = Ax + Bu
A = jacobian(dX, X);
B = jacobian(dX, U);

% Substitute or linearize about an equilibrium point
% Define equilibrium point (for example, all zeros)
x0 = [0; 0; 0];
% Substitute equilibrium values x0 into A and B
Aeq = subs(A, X, x0)
Beq = subs(B, X, x0)

% U to X transfer function
% dX = Ax + Bu implies  sX = Ax + Bu, solve for x = Gtf*u
syms s
Gtf = (s*eye(length(X)) - Aeq)^(-1)*Beq
```

6.3 System Transfer Function

$$\begin{pmatrix} \varphi(s) \\ \dot{\varphi}(s) \\ \dot{\theta}(s) \end{pmatrix} = \begin{pmatrix} -\frac{1}{m_p l_p^2 s^2 - g_0 m_p l_p + m_w l_w^2 s^2 - g_0 m_w l_w + I_p s^2} \\ -\frac{s}{m_p l_p^2 s^2 - g_0 m_p l_p + m_w l_w^2 s^2 - g_0 m_w l_w + I_p s^2} \\ \frac{m_p l_p^2 + m_w l_w^2 + I_p + I_w}{I_w s (m_p l_p^2 + m_w l_w^2 + I_p)} + \frac{g_0 l_p m_p + g_0 l_w m_w}{s(m_p l_p^2 + m_w l_w^2 + I_p)(m_p l_p^2 s^2 - g_0 m_p l_p + m_w l_w^2 s^2 - g_0 m_w l_w + I_p s^2)} \end{pmatrix} \tau(s)$$

We note that for our control problem we are trying to control the angle φ using torque, so the function of interest is the upper row equation:

$$\varphi(s) = \Big(-\frac{1}{m_p l_p^2 s^2 - g_0 m_p l_p + m_w l_w^2 s^2 - g_0 m_w l_w + I_p s^2} \Big) \tau(s)$$

Or rearranging we see that we have function of the form $\frac{1}{s^2+a^2}$:

$$\varphi(s) = \left(-\frac{1}{s^2(m_p l_p^2 + m_w l_w^2 + I_p) - m_0} \right) \tau(s)$$

This is a function with one pole in the RH plane making it unstable.

7 Controls Policy

There are actually two objectives that are necessary to keep the pendulum balanced upright.

1. The angle φ should be minimized to 0 degrees where possible.
2. The wheel velocity can not exceed the maximum for the motor configuration.

The wheel velocity in a balancing configuration can be controlled by biasing which side of the unstable equilibrium the pendulum is on. With a functioning and responsive control for φ, commanding the robot to hold position on one side of the equilibrium will cause a build-up of torque in one direction. This can be used to apply torque opposite to the direction of the spinning wheel to slow it down.

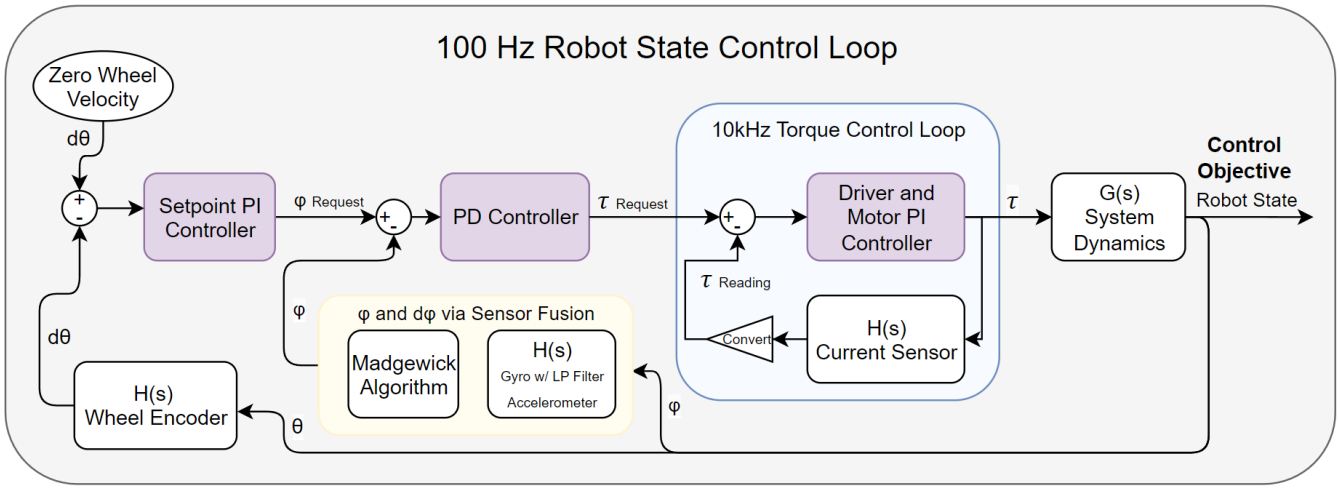


Figure 2: Control Flow Diagram

We read the wheel speed from the wheel encoder and get an error for how far off from zero velocity it is. This is fed through to a PI controller that updates the setpoint objective for the pendulum angle to counteract the velocity. This request is sent to a PD controller which functions off the error between the requested φ and the current φ as read by the the sensors. Finally this torque request is passed to the motor driver which has a high frequency feedback loop to apply the requested torque.

7.1 PD Controller for Pendulum Angle

The PD Controller for φ is tuned using the assumption that the torque requests have little delay before reaching the intended value. This is because the motor controller is running at 100 times faster than the

main control loop frequency of 100Hz. Thus we model the feedback loop of Controller -¿ $G(s)$ -¿ $H(s)$ Sensor Fusion. The sensor fusion and torque request mechanism are modeled as a delay of one $100Hz$ control cycle.

A PD controller is selected because of the dynamic setpoint that is being controlled by the cascade arrangement. If we were to include an I term then the controller would not be memoryless and would have undesirable response characteristics to the dynamic φ setpoint being requested by the higher level controller. The PD control model is a robust choice for a controller for this robot state parameter, Brevik (2017).

The MATLAB pid tuner is used to get feasible starting values based on this loop. The experimental parameters applied to the robot were found to closely match the predicted values.

7.2 PI Controller for Wheel Velocity

The PI controller is tuned heuristically once a good underlying PD controller for the angle is found. A starting value of around $K_p = 0.1$ was found to be helpful. Blending of integral term with a corresponding reduction of P is one approach to further tuning.

References

Brevik, P. (2017). *Two-axis reaction wheel inverted pendulum* (Master's thesis, Norwegian University of Science and Technology). Retrieved from `https://ntnuopen.ntnu.no/ntnu-xmlui/bitstream/handle/11250/2451060/12762_FULLTEXT.pdf?sequence=1` (Supervisor: Tor Engebret Onshus)

California Institute of Technology. (2002). *Jacobian linearization.* CDS Caltech. Retrieved from `https://www.cds.caltech.edu/~murray/courses/cds101/fa02/caltech/pph02-ch19-23.pdf` (Accessed: 2024-04-06)

Montoya, O. D., & Gil-González, W. (2020). Nonlinear analysis and control of a reaction wheel pendulum: Lyapunov-based approach. *Engineering Science and Technology, an International Journal*, *23*(1), 21–29. Retrieved from `https://doi.org/10.1016/j.jestch.2019.03.004` doi: 10.1016/j.jestch.2019.03.004

University of California, Berkeley. (2021). *Designing information devices and systems ii.* EECS Berkeley. Retrieved from `https://inst.eecs.berkeley.edu/~ee16b/sp21/notes/sp21/note15.pdf` (Accessed: 2024-04-06)

Veng, M. (2023). *Euler-Lagrange Solver.* MATLAB Central File Exchange. Retrieved 2023-10-23, from `https://www.mathworks.com/matlabcentral/fileexchange/93275-euler-lagrange-solver` (Available online: `https://www.mathworks.com/matlabcentral/fileexchange/93275-euler-lagrange-solver`)

Appendix E

Fast-MPC for Robotics

A design and implementation of fast model predictive control

FAST ONLINE MODEL PREDICTIVE CONTROL: OPTIMIZING FOR SPEED AND BALANCE

SIMON GHYSELINCKS*

Abstract. Model Predictive Control (MPC) offers a powerful framework for guiding robotic systems under constraints and uncertainty. However, real-time implementations often demand sub-millisecond solution times, pushing the limits of standard optimization approaches. This paper investigates "fast MPC," a variant leveraging specialized optimization methods to achieve high-frequency (100 Hz) online control for unstable, dynamically evolving systems. A reaction wheel inverted pendulum is used as a testbed, where balancing under torque constraints poses a significant challenge. An efficient interior point solver (Clarabel) demonstrates real-time feasibility of MPC and its potential for deployment on embedded hardware. Numerical experiments show that this approach outperforms simpler controllers, such as Proportional-Derivative (PD) and Linear Quadratic Regulator (LQR), while enforcing actuator limits and achieving rapid convergence. These results indicate that fast MPC can be effectively combined with data-driven modeling and real-time linearization, paving the way for more complex and adaptive robotic control applications in the future.

1. Motivation. Recent advancements in robotics, supported by improved control algorithms and computation, have enabled robots to perform increasingly complex tasks while interacting with dynamic environments. Such capabilities depend on robust control policies that process state estimates and respond rapidly to changes. These policies benefit from accurate models of robot motion—dynamical systems evolving over time under physical laws, control inputs, and disturbances. Increasingly precise, data-driven models improve predictions of future states, enabling more effective control strategies [4, 6, 8, 13].

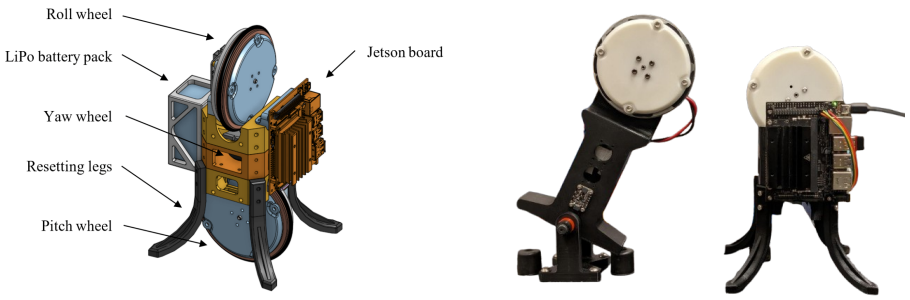
Model Predictive Control (MPC) is one such strategy: it repeatedly solves an optimization problem over a moving prediction horizon to determine control inputs that minimize a cost function while respecting physical constraints [2, 7, 12]. However, achieving high update frequencies (e.g., 100 Hz) for fast-moving robotic systems presents computational challenges. While MPC is well-established in slower industrial processes and automotive systems [13, 14, 15], its application to high-speed robotics requires new techniques. This area—referred to as "fast MPC"—focuses on reducing solver times to mere milliseconds [13, 16].

For simple systems, offline optimization can precompute control policies, but this becomes infeasible for high-dimensional or time-varying scenarios [16]. Instead, online optimization solves the quadratic program (QP) at each time step. This study focuses on the latter approach. For more details on QP, see Section 3 and Boyd and Vandenberghe [3, Ch. 4.4].

1.1. Contributons. This work builds upon the foundations of Wang and Boyd by evaluating online MPC implementations in Python on modern embedded hardware, leveraging recent optimization tools like CVXPY [1, 5] and Clarabel [11]. Testing is performed on a reaction wheel inverted pendulum, a simplified model for balancing tasks in robotics. Future extensions aim to apply these methods to a unicycle robot prototype, see Figure 1.1, that is inspired by previous work [9]. The results of this study will be used to build an implementation of fast online MPC for a highly unstable and dynamic robot.

Notation is summarized in Table 1.1. The paper proceeds as follows: Section 2 details the control problem and system dynamics; Section 3 outlines the optimization

*Engineering Physics, University of British Columbia (sghyseli@student.ubc.ca).



(a) Unicycle reaction wheel robot. (b) Inverted pendulum prototype with robot.

Fig. 1.1: Balancing robot prototypes.

methods; and Section 4 presents a case study. I will conclude by discussing results and future directions.

Table 1.1: Notation used throughout the paper.

Symbol	Description
$x_t \in \mathbb{R}^n$	State vector at time t.
$u_t \in \mathbb{R}^m$	Control input vector at time t.
$\dot{v} \in \mathbb{R}^n$	Time derivative of arbitrary vector $\mathbf{v}$.
$f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n$	Vector field describing the system dynamics.
$A \in \mathbb{R}^{n \times n}$	State transition matrix.
$B \in \mathbb{R}^{n \times m}$	Control input matrix.
$Q \in \mathbb{S}^n_+$	State cost matrix (positive semidefinite).
$R \in \mathbb{S}^m_{++}$	Control input cost matrix (positive definite).

2. Control Problem Description.

2.1. System Dynamics. MPC begins with a model of the system's time evolution. For a robot, motion is determined by the laws of physics and formalized with a state vector $x_t \in \mathbb{R}^n$ describing the motion and position. Control state vector $u_t \in \mathbb{R}^m$ specifies inputs to actuators that convert stored energy into mechanical motion. The state dynamics are a first order system of ordinary differential equations (ODEs),

$$\dot{x}_t = f(x_t, u_t), \tag{2.1}$$

where f is a nonlinear vector field.

The inverted pendulum and reaction wheel robot in Figure 1.1 are examples of systems that can be modeled using Eq. (2.1). The system dynamics are commonly linearized about point x_0 and u_0 to obtain a linear time invariant (LTI) system of the form,

$$\dot{x}_t = Ax_t + Bu_t + w(t), \tag{2.2}$$

where $A \in \mathbb{R}^{n \times n}$ and $B \in \mathbb{R}^{n \times m}$ are the state transition and control input matrices and w is a stochastic random variable representing potential disturbances to the system.

To implement MPC in discrete time, we discretize the LTI system with a time step Δt to get a new state transition matrix $\bar{A}$ and control input matrix $\bar{B}$,

$$\begin{aligned} x_{t+1} &= (I_n + \Delta t A)x_t + \Delta t\,(Bu_t + \bar{w}) \\ x_{t+1} &= \bar{A}x_t + \bar{B}u_t + \bar{w}, \end{aligned} \tag{2.3}$$

which can be iterated forward in time to predict the future states of the system. For simplicity, we use A and B without the bar notation to refer to the discrete time matrices hereafter.

The linearization of the dynamics offers many advantages for practical control designs outside of MPC [4, Ch. 8]. However in the case of linear MPC, the linearized dynamics in Eq. 2.3 are essential to ensure the control problem is a Quadratic Program (QP) that can be solved efficiently.

2.2. Constraints and Control Objective. At each discrete time step, a control input u_t is chosen and held until the next step. Over a finite horizon T, we consider a sequence $u_0, u_1, \ldots, u_{T-1}$ that influences the states $x_1, x_2, \ldots, x_T$. Physical limitations (e.g., actuator saturation) impose linear constraints on states and inputs:

$$F_x x_t + F_u u_t \leq h_c. \tag{2.4}$$

For simplicity, we often use simple box constraints on x_t and u_t. For example the motors have a maximum torque output, constraining the control input to the convex set $\{u \mid \|u\|_\infty \leq u_{\max}\}$.

The MPC control problem is to find a sequence of control inputs u_t that minimizes a cost function J over a finite time horizon of T steps, subject to the constraints given by the system dynamics (2.3) and control constraints (2.4), defined as,

$$\begin{aligned} \text{minimize} \quad & J = \ell_f(x_T) + \sum_{t=0}^{T-1} \ell(x_t, u_t) \\ \text{subject to} \quad & x_{t+1} = Ax_t + Bu_t + \bar{w}, \quad t = 0, 1, \ldots, T-1 \\ \text{and} \quad & x_{\min} \leq x_t \leq x_{\max}, \quad u_{\min} \leq u_t \leq u_{\max}. \end{aligned} \tag{2.5}$$

The loss J is defined as the sum of each stage cost function ℓ with an additional terminal stage cost ℓ_f. The cost functions are constructed to be quadratic of the state and control input vectors. Note that the constraint functions are affine, so a quadratic loss J is sufficient to make this a QP problem.

2.2.1. Cost Function. The cost function for MPC inherits theory from Linear-Quadratic Regulator (LQR) control, described in detail by Brunton and Kutz [4, Ch. 8.5]. Let Q be the positive semi-definite $\mathbb{S}^n_+$ state cost matrix and R be the positive definite $\mathbb{S}^m_{++}$ control input cost matrix. Wang and Boyd [16] include a matrix S which is a cross term between the state and control input vectors, often set to zero for simplicity since it can be difficult to tune its parameters in practice. The stage cost function is defined as,

$$\ell(x_t, u_t) = x_t^\top Q x_t + u_t^\top R u_t + 2x_t^\top S u_t + q^\top x_t + r^\top u_t, \tag{2.6}$$

The terminal cost function is defined as,

$$\ell_f(x_T) = x_T^\top Q_f x_T + q_f^\top x_T. \tag{2.7}$$

By construction, this applies a weighted penalty to an orthonormal set of basis vectors in the state and control input space. Q and R are often tuned by hand to achieve a desired balance between control effort and state tracking, and the terminal cost Q_f is often set to Q or some positive multiple of it.

For a single time step the cost function has the form of a single affine matrix operation on the combined state and control input vectors,

$$\begin{bmatrix} x_t \\ u_t \end{bmatrix}^\top \begin{bmatrix} Q & S \\ S^\top & R \end{bmatrix} \begin{bmatrix} x_t \\ u_t \end{bmatrix} + \begin{bmatrix} q \\ r \end{bmatrix}^\top \begin{bmatrix} x_t \\ u_t \end{bmatrix}.$$

2.3. Quadratic Program Formulation. The quadratic cost J in Eq. 2.5 is expanded over all time steps using a concatenation of the state and control input vectors, omitting x_0 and u_T since they are fixed by the initial and terminal conditions,

$$z = [u_0, x_1, u_1, \ldots, x_T] \in \mathbb{R}^{T(m+n)}.$$

The cost function and constraints in standard QP form are,

$$\begin{aligned} &\text{minimize} && J = z^\top H z + g^\top z \\ &\text{subject to} && Pz \leq h, \quad Cz = b, \end{aligned} \tag{2.8}$$

with a breakdown of the matrices H, g, P, h, C, and b given below.

2.3.1. Cost Function Formulation. H is block tridiagonal with Q and R on the diagonal and S on the off-diagonal blocks and g is the affine term,

$$H := \begin{bmatrix} R & 0 & 0 & \ldots & 0 & 0 & 0 \\ 0 & Q & S & \ldots & 0 & 0 & 0 \\ 0 & S^\top & R & \ldots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & Q & S & 0 \\ 0 & 0 & 0 & \ldots & S^\top & R & 0 \\ 0 & 0 & 0 & \ldots & 0 & 0 & Q_f \end{bmatrix} \qquad g := \begin{bmatrix} r + 2S^\top x_0 \\ q \\ r \\ \vdots \\ q \\ r \\ q_f \end{bmatrix}.$$

For the class of control problems considered here, S is set to 0 making H a block diagonal matrix.

2.3.2. Problem Constraints Formulation. The motion and control box consraints from Eq. 2.5 can be written as a set of linear inequalities, the control limitations as $Pz \leq h$, defined as,

$$P := \begin{bmatrix} F_u & 0 & 0 & \ldots & 0 & 0 & 0 \\ 0 & F_x & F_u & \ldots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & F_x & F_u & 0 \\ 0 & 0 & 0 & \ldots & 0 & 0 & F_x \end{bmatrix} \qquad h := \begin{bmatrix} h_c - F_x x_0 \\ h_c \\ \vdots \\ h_c \\ h_f \end{bmatrix}.$$

Note that for box constraints, each element of x and u have two constraints applied via the F_x and F_u matrices.

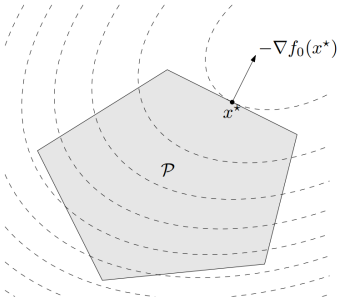


Fig. 2.1: Geometric interpretation of the QP problem, taken from Boyd and Vandenberghe [3, pg. 153].

2.3.3. Dynamic Constraints Formulation. The system dynamics are enforced by the equality constraint $Cz = b$, ensuring the state evolves according to the system dynamics,

$$C := \begin{bmatrix} -B & I & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & -A & -B & I & \dots & 0 & 0 & 0 \\ 0 & 0 & 0 & -A & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & I & 0 & 0 \\ 0 & 0 & 0 & 0 & \dots & -A & -B & I \end{bmatrix} \qquad b := \begin{bmatrix} Ax_0 + \bar{w} \\ \bar{w} \\ \vdots \\ \bar{w} \\ \bar{w} \end{bmatrix}.$$

The QP is a quadratic function constrained to a convex set that is formed by the intersection of the half-spaces defined by the linear inequalities $Pz \leq h$ and the hyper-planes defined by the equality constraints $Cz = b$, as shown in Figure 2.1.

3. Optimization Method.

3.1. Exact Primal Barrier Method. The goal is to solve the QP problem in Eq. 2.8. Let $\mathcal{F} := \{z \mid Pz \leq h\}$, then the inequality constraints can be absorbed into the minimization objective using the indicator function $\delta_{\mathcal{F}}$,

$$\text{minimize} \quad J = z^\top H z + g^\top z + \delta_{\mathcal{F}}(z).$$

However, the indicator funtion is non-differentiable, so we apply a homotopy between the indicator function and the logarithmic barrirer function to recover a differentiable objective. The homotopy broadcasts or smooths the indicator function's infinite gradient at the boundary toward the interior points of $\mathcal{F}$ as a relaxed problem, see Figure 3.1. The QP stated using the barrier function is given by,

$$\begin{aligned} &\text{minimize} \quad J = z^\top H z + g^\top z - \mu \sum_{i=1}^{2T(m+n)} \log(h_i - p_i^\top z) \\ &\text{subject to} \quad Cz = b, \end{aligned} \tag{3.1}$$

where $\mu > 0$ is a barrier parameter. As $\mu \to 0$, the solution approaches that of the original QP. This method, known as the primal barrier method [3, Ch. 11], can be solved using Newton's method. For the special case considered here—no cross-terms ($S = 0$), diagonal Q and R, and box constraints—Wang and Boyd [16] show that complexity scales as $T(n^3 + n^2 m)$.

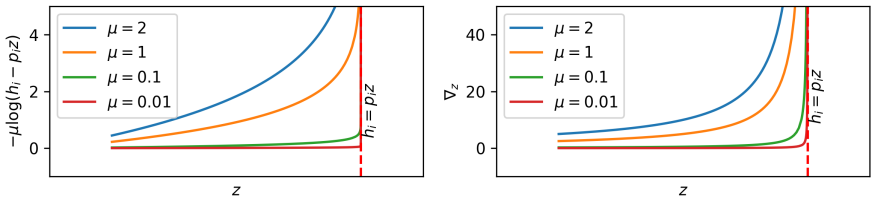


Fig. 3.1: Single constraint logarithmic barrier function (right), and its gradient (left).

3.2. Approximate Primal Barrier Method. To further reduce computational time, the barrier parameter μ can be held fixed rather than approaching zero. Although this only approximates the exact solution, it is often sufficient for good control performance. Additionally, "warm starts" use the previous solution as an initial guess, reducing the required Newton steps. Limiting the maximum Newton iterations per solve (e.g., to 3–10 steps) further ensures real-time feasibility. While this deviates from theoretical guarantees, numerical experiments by Wang and Boyd [16] demonstrate its effectiveness for fast MPC. In the context of online optimal control, a fixed time limit on the computation is more effective than a fixed iteration limit.

4. Case Study: Inverted Reaction Wheel Pendulum. To illustrate the fast MPC approach, we simulate an inverted reaction wheel pendulum. The system's nonlinear dynamics, shown in Figure 1.1, model a pendulum with a reaction wheel acting as the single control input. The state has three elements, and the control input is a single motor torque. We discretize the system at 100Hz and run a 5-second simulation, starting 1 radian from upright and subject to a 1.0 Nm motor torque limit. The goal is to stabilize the pendulum while preventing the wheel from spinning to excessive speeds.

Three controllers are compared:

1. **PD Controller:** A simple proportional-derivative controller using only angle error and its derivative. Although fast and simple, it cannot manage multiple objectives or constraints.
2. **LQR:** A linear-quadratic regulator using the same Q and R as MPC. It balances the pendulum but is unaware of actuator limits, applying infeasible torques.
3. **Fast Online MPC:** At each time step, the system is linearized, and the MPC QP is solved using the approximate primal barrier method. Torque constraints and a 30-step horizon are enforced, and a time limit (e.g., 1 ms) replaces iterative convergence checks. Warm starts reuse the previous solution to accelerate computation.

4.1. Results. The experimental code [10] is available on Github for download.

The three controllers run in a simulation environment consisting of the nonlinear system dynamics advanced forward with a numerical ODE solver. At each step of the numerical solution, a new control input is calculated using the respective controller. The results of the simulation are shown in Figure 4.1. The PD controller is able to perform well, but is only able to control a single objective, the pendulum angle, which will eventually result in the reaction wheel spinning up to its maximum speed. The LQR controller is able to balance the pendulum, but is not aware of the motor torque limits, resulting in torque commands that are not feasible and poor performance. The

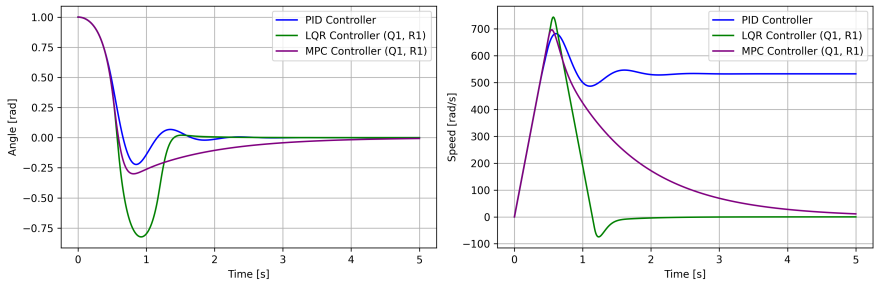


Fig. 4.1: PID, LQR, MPC control comparison for pendulum angle (left) and reaction wheel speed (right). MPC and LQR use the same Q and R matrices, but LQR is not aware of the motor torque limit of 1 Nm, resulting in overshoot behavior. Computation of MPC is restricted to 1 ms per time step.

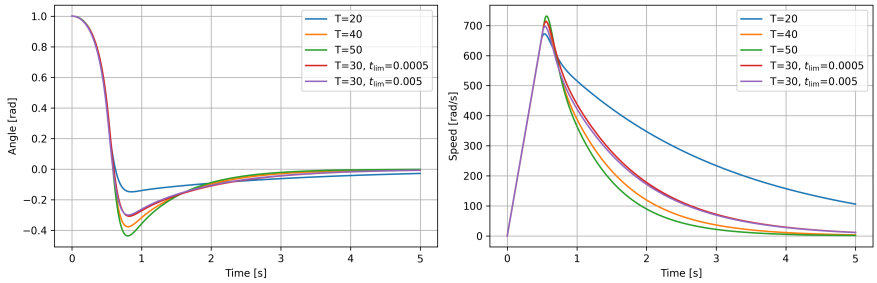


Fig. 4.2: A comparison of algorithm settings for MPC for pendulum angle (left) and reaction wheel speed (right). The number of time horizon steps is varied from 20 to 50 and a demonstration of an extreme computational time limits of 0.5 ms versus a complete solution at 5 ms is shown.

MPC controller is able to balance the pendulum and reduce the reaction wheel speed to zero, while respecting the motor torque limits.

The fast MPC allows for heuristic tuning of acceptable time limits for computation, as well as time horizon, shown in Figure 4.2. When MPC is constrained to just a single step of the interior point method, the control policy still holds well, likely due to successive optimal solutions being close in proximity and using the warm start to intialize close to the optimal solution. The time horizon has a larger impact on the problem complexity and performance, with a horizon of 50 steps resulting in a more optimal solution within the same 1 ms time limit.

5. Conclusion. This study demonstrates that fast MPC, implemented via Python and tools like CVXPY and Clarabel, can achieve millisecond-scale QP solves suitable for real-time robotic control. By applying approximate barrier methods, warm starts, and strict time limits, one can trade off optimality for speed while maintaining acceptable performance. The approach readily extends to data-driven models, making it promising for complex, unstable systems. Future work will apply this method to a unicycle reaction wheel robot prototype and evaluate its performance on hardware.

References.

[1] A. Agrawal, R. Verschueren, S. Diamond, and S. Boyd. A rewriting system for convex optimization problems. *Journal of Control and Decision*, 5(1):42–60, 2018.

[2] A. Aswani, H. Gonzalez, S. S. Sastry, and C. Tomlin. Provably safe and robust learning-based model predictive control. *Automatica*, 49(5):1216–1226, 2013. ISSN 0005-1098. doi: https://doi.org/10.1016/j.automatica.2013.02.003. URL https://www.sciencedirect.com/science/article/pii/S0005109813000678.

[3] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, March 2004. ISBN 0521833787.

[4] S. L. Brunton and J. N. Kutz. *Data-driven science and engineering*. Cambridge University Press, Cambridge, England, 2 edition, May 2022.

[5] S. Diamond and S. Boyd. CVXPY: A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research*, 17(83):1–5, 2016.

[6] F. Djeumou, C. Neary, and U. Topcu. How to learn and generalize from three minutes of data: Physics-constrained and uncertainty-aware neural stochastic differential equations, 2023. URL https://arxiv.org/abs/2306.06335.

[7] C. E. García, D. M. Prett, and M. Morari. Model predictive control: Theory and practice—a survey. *Automatica*, 25(3):335–348, 1989. ISSN 0005-1098. doi: https://doi.org/10.1016/0005-1098(89)90002-2. URL https://www.sciencedirect.com/science/article/pii/0005109889900022.

[8] A. R. Geist and S. Trimpe. Structured learning of rigid-body dynamics: A survey and unified view from a robotics perspective. *GAMM-Mitteilungen*, 44(2): e202100009, 2021.

[9] A. R. Geist, J. Fiene, N. Tashiro, Z. Jia, and S. Trimpe. The Wheelbot: A Jumping Reaction Wheel Unicycle, July 2023. URL http://arxiv.org/abs/2207.06988. arXiv:2207.06988 [cs].

[10] S. Ghyselincks. Fast online mpc for reaction wheel inverted pendulum. https://github.com/chipnbits/fastMPC-simul, 2024. Available at https://github.com/chipnbits/fastMPC-simul.

[11] P. J. Goulart and Y. Chen. Clarabel: An interior-point solver for conic programs with quadratic objectives, 2024. URL https://arxiv.org/abs/2405.12762.

[12] K. Holkar and L. M. Waghmare. An overview of model predictive control. *International Journal of control and automation*, 3(4):47–63, 2010.

[13] D. Nguyen-Tuong and J. Peters. Model learning for robot control: a survey. *Cognitive Processing*, 12(4):319–340, Apr. 2011. ISSN 1612-4790. doi: 10.1007/s10339-011-0404-1. URL http://dx.doi.org/10.1007/s10339-011-0404-1.

[14] S. J. Qin and T. A. Badgwell. An overview of industrial model predictive control technology. In *AIche symposium series*, volume 93, pages 232–256. New York, NY: American Institute of Chemical Engineers, 1971-c2002., 1997.

[15] J. Richalet, A. Rault, J. Testud, and J. Papon. Model predictive heuristic control: Applications to industrial processes. *Automatica*, 14(5):413–428, 1978. ISSN 0005-1098. doi: https://doi.org/10.1016/0005-1098(78)90001-8. URL https://www.sciencedirect.com/science/article/pii/0005109878900018.

[16] Y. Wang and S. Boyd. Fast Model Predictive Control Using Online Optimization. *IEEE Transactions on Control Systems Technology*, 18(2):267–278, Mar. 2010. ISSN 1063-6536, 1558-0865. doi: 10.1109/TCST.2009.2017934. URL http://ieeexplore.ieee.org/document/5153127/.