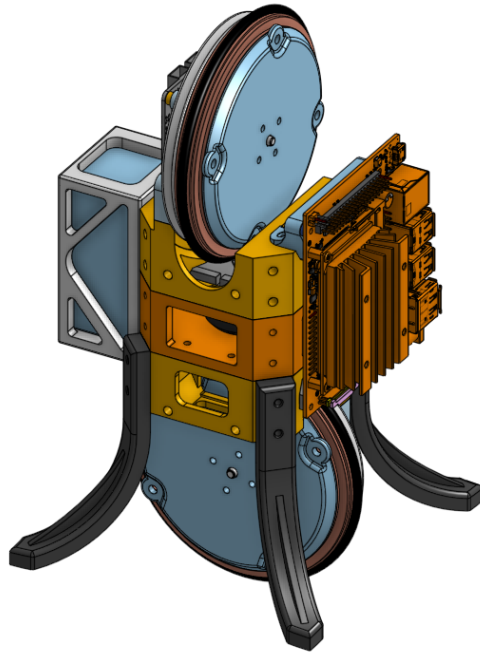


Learning to Balance

A Reaction-Wheel Unicycle Robot



Tristan Lee
Kyle Mackenzie
Simon Ghyselincks
Jackson Fraser
Julian Lapenna

ENPH 459 — Project #2411

The University of British Columbia

Vancouver, BC, Canada

April 14th 2021

Executive Summary

This two-year capstone project, sponsored by the UBC Engineering Physics Project Lab, combines robotics, control engineering, and machine learning to develop a self-balancing unicycle robot using a Reinforcement Learning (RL) algorithm.

Previous work includes capstones that built a self-balancing robot using classical control and a single-axis inverted pendulum managed with RL. Additionally, the Max Planck Institute's Wheelbot project [3], a compact self-balancing robot, has provided significant guidance and inspiration. Our goal is to enhance the Wheelbot by incorporating RL control and point-to-point navigation.

Currently, our efforts have produced a reaction wheel inverted pendulum (RWIP) to understand the unstable roll axis dynamics of the full robot. This RWIP is part of our prototyping pipeline, streamlining testing of classical and RL algorithms, along with system integration including actuators, sensors, and computational demands. We implemented both a traditional Proportional-Integral-Derivative controller (PID) as well as an RL controller and found the latter to be more effective, though aggressive in its response to disturbances.

Moving forward, we recommend promptly building a physical prototype and setting up comprehensive telemetry and data visualization to optimize testing and design iterations. In the project's second year, we will finalize the full robot's design, refine our control architecture with a state-space model for more sophisticated controllers, and develop the navigation system.

Contents

1	Introduction	5
1.1	Background	5
1.2	Previous Work	5
1.3	Requirements	5
2	System Overview	6
3	Project Development Overview	7
4	Robot Design	8
4.1	System Dynamics Overview	8
4.2	System Dynamics Design Motivation	10
4.3	Hardware Design	13
4.3.1	Self-Righting Capabilities	13
4.3.2	Powering The System	13
4.3.3	Actuators	13
4.3.3.1	Motors	14
4.3.3.2	Moteus Drivers	14
4.3.3.3	Flywheel	14
4.4	Computer and Software Systems	15
4.4.1	Actuator Control Interface	15
4.4.2	Inertial Measurement Unit	15
4.4.3	Server and Telemetry	16
4.5	Software Architecture	16
4.6	Control Algorithms	18
4.6.1	Classical Controls	18
4.6.2	RL Controls	19
4.6.3	RWIP Reinforcement Learning Control	20
5	Current Progress	21
6	Conclusions	22
7	Future Development and Recommendations	23
8	Deliverables	25
	Appendix A	26

Appendix B	36
Appendix C	44
Appendix D	51

List of Figures

1	Resting position of robot.	6
2	Labelled robot CAD design.	6
3	RWIP (Reaction Wheel Inverted Pendulum) prototype models the roll dynamics.	7
4	Flow chart of the RL Unicycle development process to date.	8
5	Diagram showing gyroscopic precession	9
6	Principal axes of control.	9
7	A reaction wheel pendulum roll axis and rolling inverted pendulum pitch axis model. . .	10
8	Center of mass shown as a design parameter.	11
9	Broom balancing problem.	11
10	Yaw axis simulation and design specifications.	12
11	A cutaway view of the yaw wheel compartment.	12
12	Side, angled and back view of the actuators.	14
13	Computational system map.	15
14	Project file structure tree diagram.	17
15	100Hz control cycle diagram.	17
16	Reaction wheel inverted pendulum diagram.	18
17	Detailed flow chart of robot state control loop.	19
18	Reinforcement Learning straining loop diagram. Source: Deep Reinforcement Learning for Constrained Field Development Optimization in Subsurface Two-phase Flow	19
19	Accelerated reinforcement learning training loop showing forward pass interactions and backward pass optimizations Source: https://blogs.nvidia.com/blog/deep-reinforcement- learning-gpus-robotics/	20
20	RWIP front, side and back view. Final robot draft chassis.	22
21	Project flowchart from proposal.	23
22	Project flowchart for 2024-2025	24

1 Introduction

1.1 Background

Robotics engineering and control theory are integral to modern automation, involving the design of robots and applying mathematical principles to their operations. Advances in machine learning have revolutionized these fields with the development of adaptive controllers that learn from their environment and interactions, enhancing robotic capabilities and applications.

Robotics often deals with complex dynamics that are difficult to model precisely. While traditional control systems offer reliability, they may struggle with unpredictable conditions. Reinforcement learning allows robots to adapt through trial and error, improving decision-making in dynamic environments and enabling complex tasks that might otherwise be considered too challenging or impractical, such as managing self-balancing unicycle robots.

1.2 Previous Work

Our capstone project aims to create a self-balancing unicycle robot, leveraging an RL control algorithm. This involves integrating control theory with both conventional and machine learning-based controllers, alongside hardware and software development, to achieve autonomous balance, automatic upright recovery, and point-to-point navigation. Our project builds upon capstone team 1868's classically controlled unicycle robot [1], team 2153's RL-controlled inverted pendulum [6], and the Max Planck Institute's Wheelbot project [3], which developed a more advanced self-balancing unicycle with self-righting capabilities.

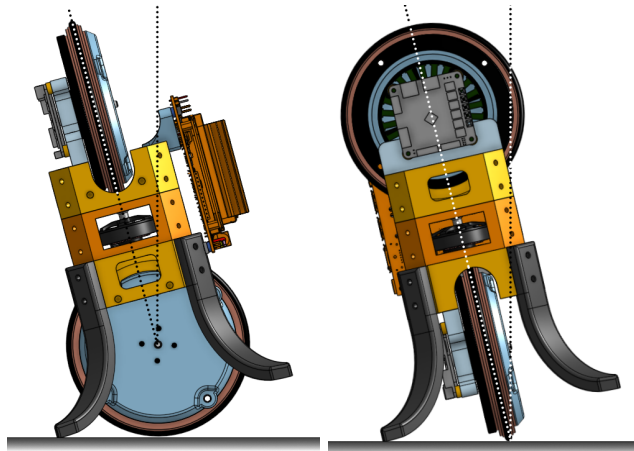
1.3 Requirements

The requirements for our robot are as follows:

1. Balance by staying upright within of vertical for 10 minutes continuously.
2. Stand up by entering balancing status from a rest position of 25° from vertical in 0.5 seconds.
3. Navigate freely inside a circle of radius 1m to within 2cm of the destination.

The angles are measured from vertical to the axis through the robot's center of mass (CoM) and the point of rotation. Figure 1 shows a front and side view of the robot in resting positions.

Additionally, a development goal made aside from the final deliverable was to achieve a working prototype robot that can balance with a single degree of freedom using both a classical and RL controller. Specifically, it should be an inverted pendulum so that the design will extend to the second phase of the project in year 2.



(a) Front view, the point of rotation is the joint of the bottom wheel. (b) Side view, the point of rotation is the contact point with the ground.

Figure 1: Resting position of robot.

2 System Overview

The robot is composed of two reaction wheels, a single drive wheel, a controller, and a battery, all mounted on a 3D printed PLA frame. It has a total height of 30cm and a weight of 1.25kg, incorporating a compact and efficient design intended to allow self-erection from a position resting on its resetting legs. The Jetson Nano acts as an autonomous controller that reads the sensors and reacts to the environment using the motors.

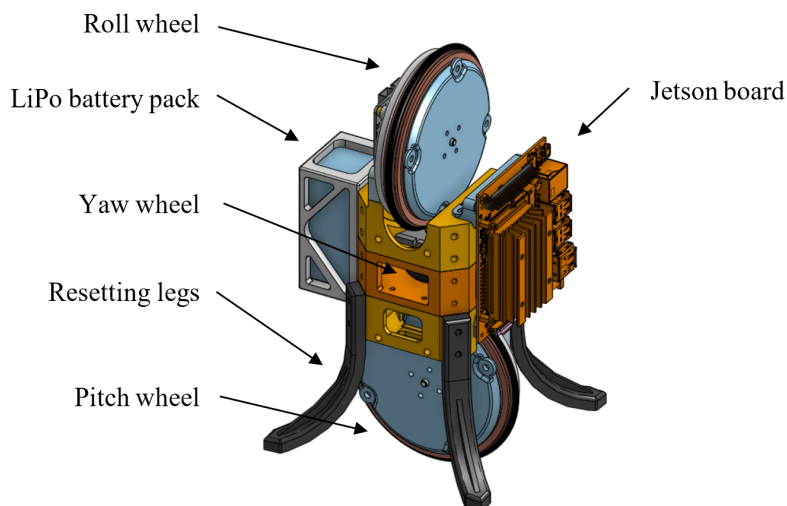


Figure 2: Labelled robot CAD design.

Much like a unicycle, it balances on one wheel, with side-to-side stability provided by the roll wheel and direction controlled by a yaw wheel. The mechanism of balancing and steering relies on a reaction torque produced by spinning the reaction wheels. When a motor applies torque to one of the flywheels, an equal and opposite torque acts on the robot's body, with the net effect altering the angular motion of both the wheel and the robot.

The unstable axes are roll and pitch, where tipping occurs, Section 4.1 provides a detailed breakdown of the physics.

3 Project Development Overview

The project's progression is divided into six iterative stages, all contributing to a roll wheel inverted pendulum (RWIP) prototype. This prototype serves as a benchmark to assess our roll dynamics model, and test control algorithms, sensors, and actuators. These stages are:

1. System dynamics research
2. Hardware design
3. Software architecture
4. Sensor and actuator interfacing
5. Control model construction
6. Integration and testing

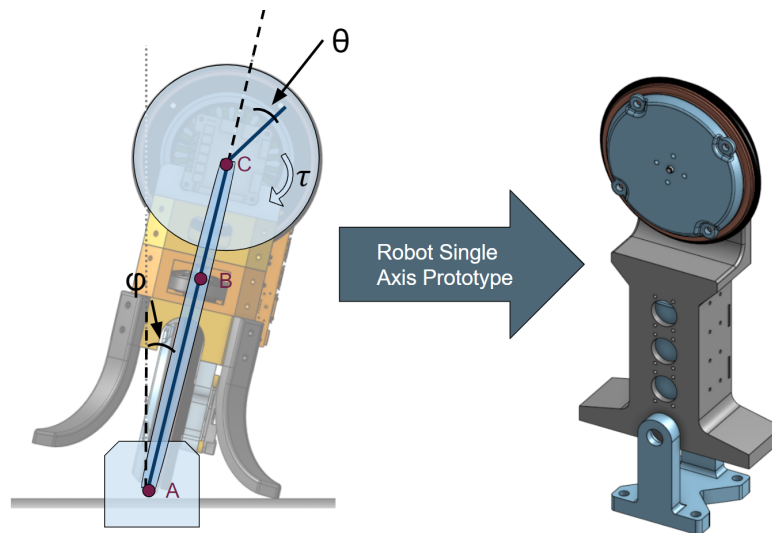


Figure 3: RWIP (Reaction Wheel Inverted Pendulum) prototype models the roll dynamics.

The first stage produced the project proposal, with subsequent stages focusing on development. The prototype streamlines the development pipeline for next year, facilitating faster iteration for the complete robot.

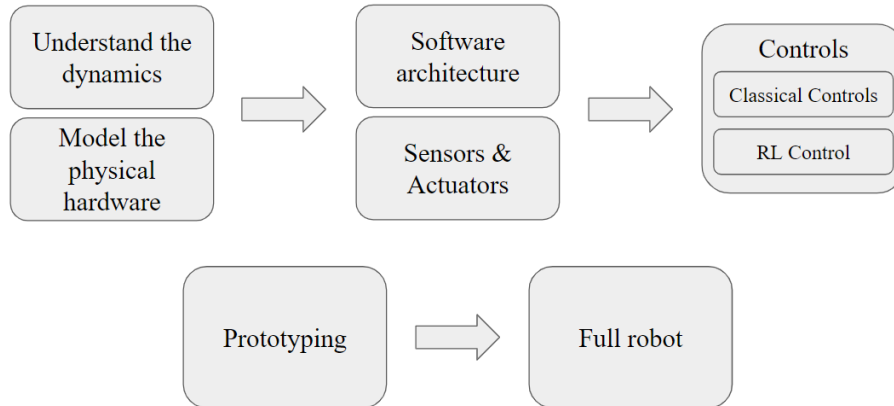


Figure 4: Flow chart of the RL Unicycle development process to date.

We achieved a completed RWIP robot and will cover the development, results and major considerations of the design in each stage. Our process flow chart is shown in Figure 4. We aim to follow a similar progression next year for the complete robot.

4 Robot Design

4.1 System Dynamics Overview

The design and control of our robot begin with an understanding of the fundamental physics and system dynamics governing its movement. The distribution of mass, total mass, and the interaction of the moving parts of the robot are important for a robust design.

The robot balances based on Newton's third law of motion, where for every action (torque), there is an equal and opposite reaction. This follows from the rotation form of Newton's second law $\tau = I\alpha$, indicating that angular acceleration is directly proportional to applied torque. Thus, when a motor applies torque to spin a flywheel, an equal and opposite torque acts on the robot's body. Calculating the appropriate torque for each axis allows the robot to maneuver into a stable state.

The reaction wheels and drive wheel of the robot create a motion that can be unintuitive due to gyroscopic precession, a consequence of the conservation of angular momentum. An illustration of the effect is shown in Figure 5.

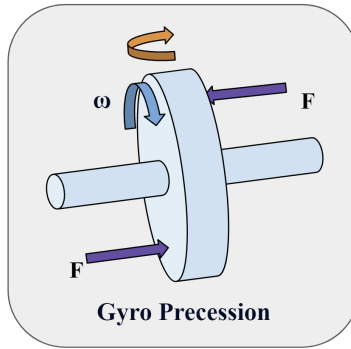


Figure 5: Diagram showing gyroscopic precession

When a wheel is spinning with a speed of ω , and a torque or force F is applied to it, the wheel will not only rotate in the expected direction of force, but will also produce a sideways twisting motion (shown in orange above), described in more detail in [2].

Motor inputs in one wheel can interact with another wheel's spin, disturbing the robot's intended motion. This interaction can be minimized by maintaining low wheel speeds and arranging the rotation axes orthogonally to each other, as shown in Figure 6.

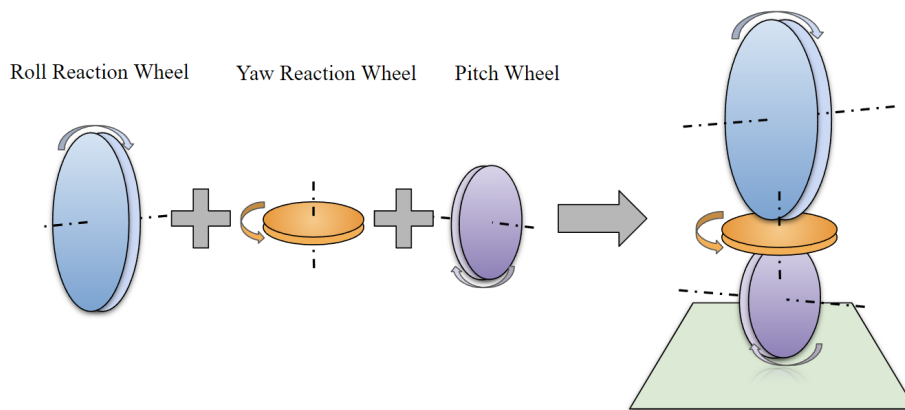


Figure 6: Principal axes of control.

In this setup, coupling occurs only through gyroscopic precession rather than direct control inputs. Additionally, by keeping the yaw wheel stationary except during turns, most gyroscopic effects arise from interactions between the pitch and roll wheels, creating yaw precession. In an upright state with no yaw wheel spin, the roll and pitch axes are fully decoupled from each other [3].

The yaw axis remains stable due to friction, whereas the roll and pitch axes are inherently unstable and pose greater control challenges. These axes can be treated as separate control problems, allowing for

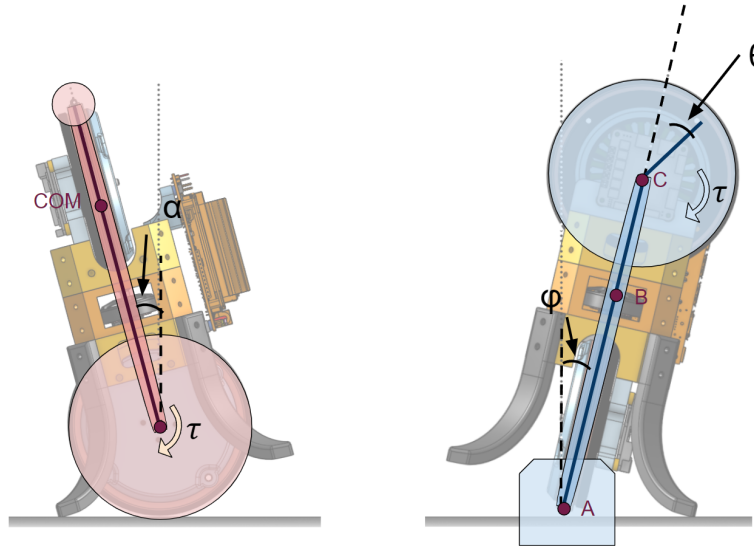


Figure 7: A reaction wheel pendulum roll axis and rolling inverted pendulum pitch axis model.

targeted solutions and effective decoupling, following strategies from previous projects [1].

4.2 System Dynamics Design Motivation

The moment of inertia informs the torque requirements for the robot to be able to self-right and balance. The total mass and its distribution are what determine the moment, with a greater cost associated with higher mass located further from the center of mass. The highest mounted item is the roll axis wheel and motor, where a high torque to mass ratio is desired to minimize this effect. The Wheelbot [3] identified self-righting as the limiting design factor so we used a numerical simulation of this procedure to de-risk our torque requirements and ensure that the final robot can self-right. The reasoning is that if the robot can accomplish this task then it will exceed the torque requirements for other system dynamics.

A center of mass that is 100-300mm in height optimizes controllability, based on early simulations of the model. The tradeoff is between a system that requires less effort to balance but that is less responsive (higher COM and inertia) versus a higher effort, more responsive system (lower COM and inertia), much like balancing a broom that is upright versus upside down.

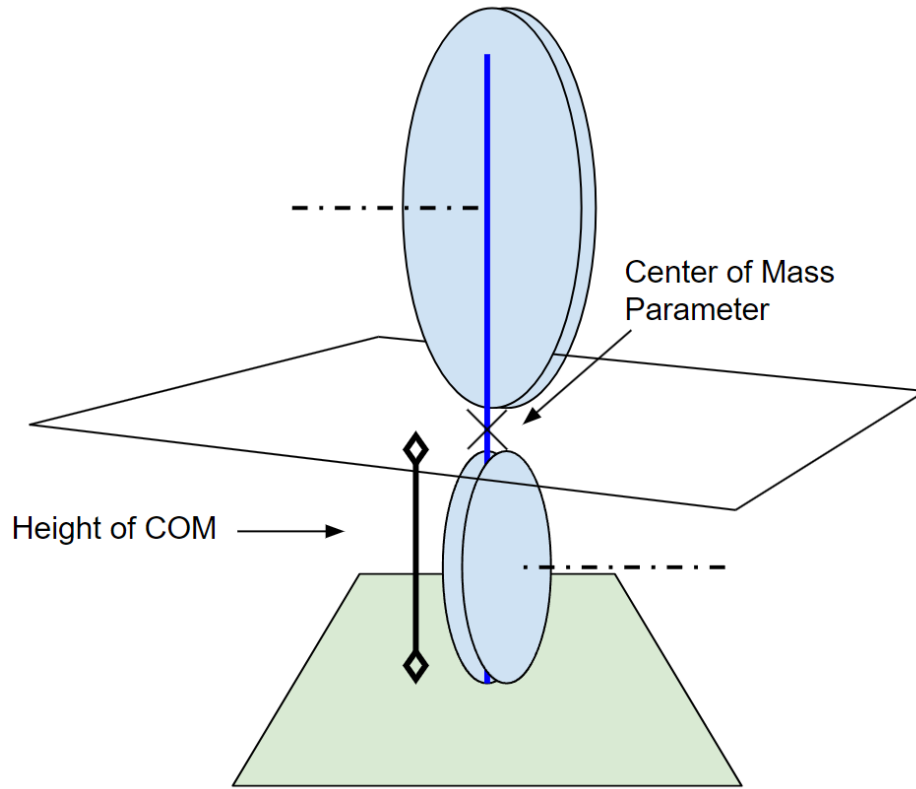


Figure 8: Center of mass shown as a design parameter.

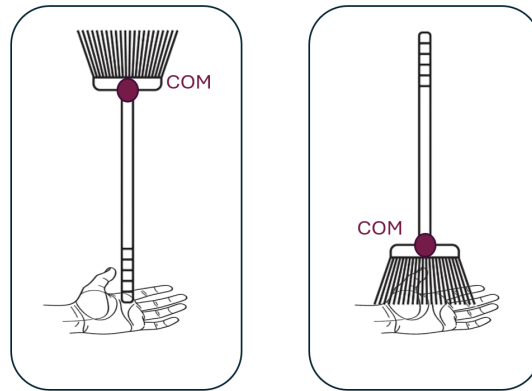


Figure 9: Broom balancing problem.

The yaw axis is controlled by a reaction wheel but the mechanism is entirely different since gravity is not a destabilizing force. To execute turns a yaw flywheel is spun to create a reaction torque spinning the robot about the ground contact point. The friction with the ground is used to stabilize the motion or any residual build up of speed in the wheel.

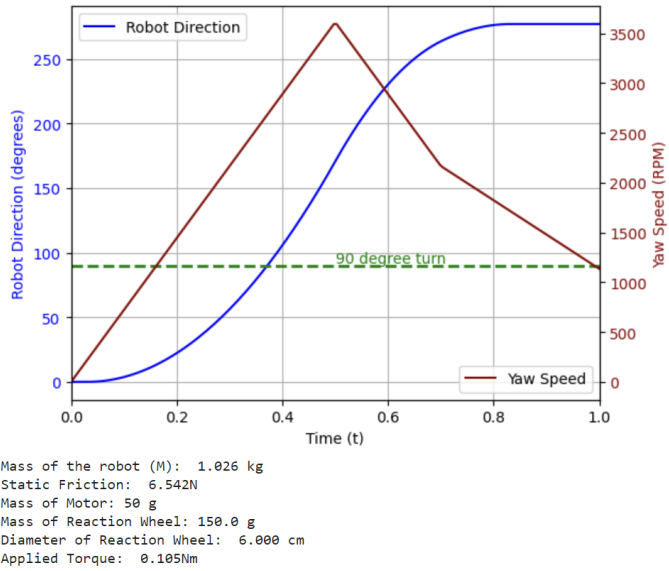


Figure 10: Yaw axis simulation and design specifications.

We also consider the impact of the flywheel used to create a reaction torque. Once a flywheel reaches maximum speed, it can no longer provide any reaction torque. An ideal flywheel has a high moment of inertia to resist accelerations, but at the cost of increasing the overall mass of the robot.

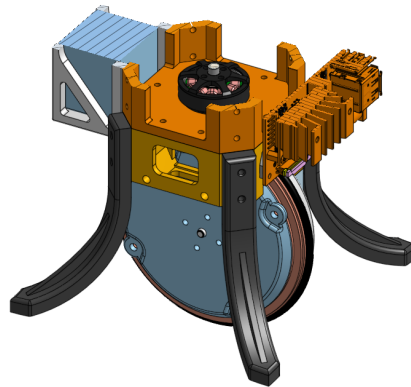


Figure 11: A cutaway view of the yaw wheel compartment.

Our design places the flywheel mass as far from the center of rotation as possible, with most of the robot height dedicated to flywheel diameter. To minimize design risk, our approach closely follows that of the Wheelbot with proven performance.

Finally, the full robot will have a fourth degree of control, translation through the environment which is managed by a combination of inputs from all the control axes. Once the robot is able to balance

upright, introducing a slight bias to lean forward or back will trigger acceleration in the pitch wheel, driving the robot forward and back.

4.3 Hardware Design

The mechanical design of our robot takes inspiration from the Wheelbot with some key changes to accommodate the addition of a yaw wheel and a larger processor for reinforcement learning. The main considerations are to minimize the total mass of the robot, while maintaining a low center of mass and orthogonal control axes.

The yaw wheel is added to allow for steering of the robot, which comes at the cost of extending the chassis vertically by 3cm.

4.3.1 Self-Righting Capabilities

The self-righting capabilities of the robot were the main factor in determining the overall torque requirements for the project. The Wheelbot paper calculated their required torque to stand up at 1.3 Nm. Following their calculations, we found our own stand up torque from our projected mass, moment of inertia and height of center of mass to be higher at 1.4 Nm. Details and calculations of the stand up maneuver can be found in Appendix A. Following this, we chose the MN6007II motors to achieve similar results as the Wheelbot.

4.3.2 Powering The System

The considerations for battery choice were maximum discharge current, capacity, and mass. The motor current required to overcome gravity in the rest position is approximately 24A. For this reason, we chose a battery with 50A maximum discharge current. Next, the battery should have enough capacity to support self-balancing and navigation for over 10 minutes of RL training. For this, we estimate 2A continuous average current, for 30 minutes continuous operation, which gives us a large factor of safety. Finally, the LiPo battery should be under 200g in order to keep overall robot mass to a minimum.

4.3.3 Actuators

Each of the robot's actuators is made up of a brushless DC (BLDC) motor, a moteus-n1 BLDC motor driver, an encoder magnet, and various 3D printed parts to provide the mechanical structure of the sub-system. Similar to other parts on the robot, the actuator design was strongly motivated by the Wheelbot. The final design of the roll and pitch axis actuators is a compact, self-contained and modular system that can be moved from various test stands to the chassis of the full robot.

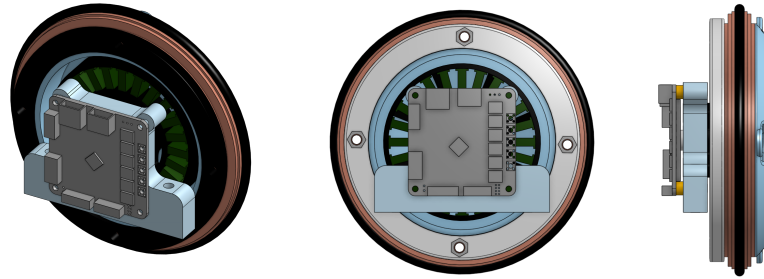


Figure 12: Side, angled and back view of the actuators.

4.3.3.1 Motors

Each motor must be able to perform rapidly-changing high torque manoeuvres in order to control the robot and maintain its upright position. Due to their high torque-to-mass ratio, brushless DC motors are most suitable for this application, while simultaneously meeting low mass and high torque constraints for self-righting. The T-Motor Antigravity MN6007 was used by Wheelbot for its high continuous torque, low mass, and thin outrunning pancake form factor [3]. Similarly, we chose its successor, the T-Motor Antigravity MN6007II as it makes improvements on both parameters for nearly the same price.

4.3.3.2 Moteus Drivers

The benefits of using brushless motors, comes at the cost of control complexity. Thus, we opted to buy three moteus-n1 motor drivers for their high power rating, on-board magnetic encoder, open-source firmware and control libraries, and field oriented based control. The high power rating is necessary for driving the MN6007II at sufficient voltages during high-torque maneuvers. The driver is mounted directly behind the motor with its encoder centered to the rotor shaft. The n1's STM32 runs a 15-30kHz control loop that implements field oriented control (FOC), an efficient and precise method of controlling the speed and torque of brushless motors which decouples the torque and magnetic fields. Because of this, the main 100Hz control loop can send torque requests directly to the driver, which handles the rest. Additionally, the Moteus can be queried for data such as the motor velocity, motor torque, phase currents, etc.

4.3.3.3 Flywheel

The design of the flywheel follows very closely to the Wheelbot. Steel rings machined from sheet metal, are positioned around the center of the MN6007II's rotor, such that they align with its CoM. These rings place the mass of the flywheel far from the axis of rotation to increase its moment of inertia. The housing of the rings is resin printed to avoid inaccuracies in concentricity of FDM 3D printers, which could cause oscillations in the wheel.

4.4 Computer and Software Systems

The robot uses a distributed network of microcomputers for signal processing that is used to control and respond to the system dynamics. The central controller, an NVIDIA Jetson Nano 4GB was selected for its GPU enhanced processor and integration with NVIDIA AI training tools. The Jetson comes with additional cost and weight, however simpler ESP32 or STM32 chip options are non-viable for RL control. A map of the computation network is shown in Figure 13, including the motor drivers from the previous section.

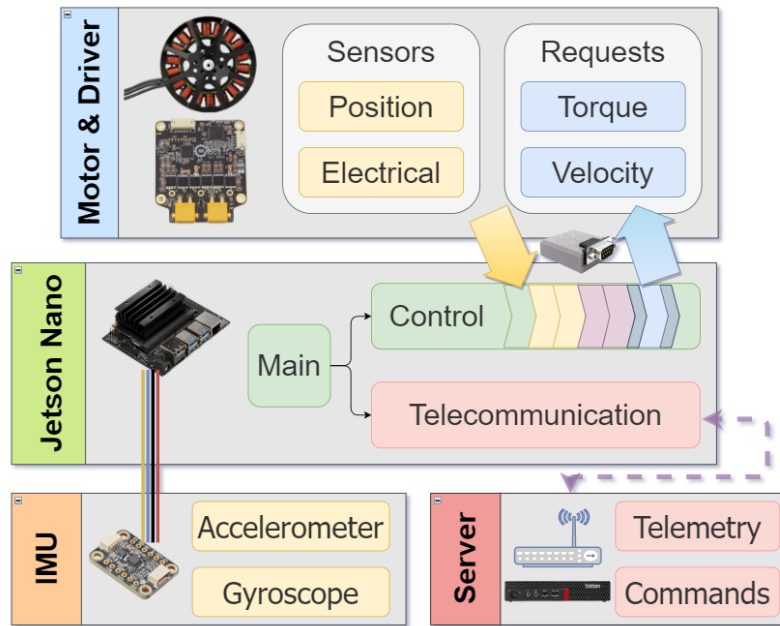


Figure 13: Computational system map.

4.4.1 Actuator Control Interface

A USB port on the Jetson connects to a CAN-FD bus which is used to interface with all of the motor drivers to receive motor data and send torque, position, or velocity requests.

4.4.2 Inertial Measurement Unit

An inertial measurement unit (IMU) connects to the Jetson via a 400kHz I2C connection. The data read from sensors is stored in the registers of an onboard processor then relayed back to the central controller to estimate the robot's position in space.

Our choice of IMU also follows from the Wheelbot. The ICM-20948 is a 9 DoF sensor with a 3-axis gyroscope, a 3-axis accelerometer, and a 3-axis compass. The data output from the IMU is processed by the Madgwick complementary filter, a sensor fusion algorithm. The gyroscope provides very accurate

orientation changes at a high frequency, but its data is prone to sensor bias and requires a reference to correct the drift. The accelerometer provides this reference by sensing the direction of gravity. The full algorithm can be found in the Madgwick publication [5]. Our robot uses a gain value of 0.5-1.0 in the algorithm.

4.4.3 Server and Telemetry


A wifi dongle connects the Jetson to the internet to send out telemetry data and receive live updates and commands for development purposes. While the robot communicates with our server via MQTT protocol, it is important to note that the robot can still operate fully autonomously.

For more information about our telemetry systems please see Appendix B or the web version [web version](#).

4.5 Software Architecture

Our robot software operates with a 100Hz control cycle, a rate which was selected to be reactive to environmental changes, but slow enough that the sequence of commands can be completed before the end of a cycle.

Consistent timing is maintained using a recompiled kernel from source, with the PREEMPT-RT patch enabled, allowing to set a soft 'real-time' priority to the control process. The procedure may be of interest to future projects by our sponsor, so we have included specific instructions in Appendix C, [web version](#).

The file structure of the project is shown in  for access to source files please view the repository hosted on [GitHub](#).

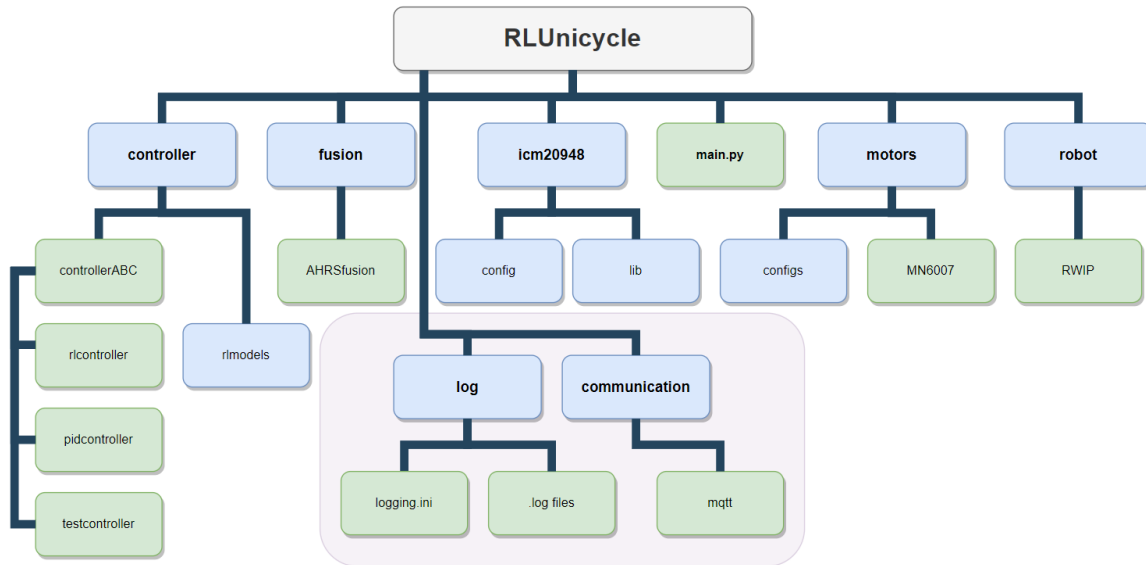


Figure 14: Project file structure tree diagram.

Our 100Hz control cycle executes the following six steps:

1. **Read IMUs:** A custom icm20948 library is used to read IMU data.
2. **Read Motors:** Moteus library reads the motor, driver, and flywheel position/velocity.
3. **State Estimate:** Madgwick sensor fusion algorithm (Madgwick, 2010) is used to correct for gyroscopic drift and provide robot orientation estimates.
4. **Controller Decision:** A selectable controller is used to translate the robot state into a torque command.
5. **New Request to Motors:** The controller waits until a fixed time to update the motor drivers with a new torque request.
6. **Offload Telemetry, Import Updates:** Data from the control cycle is offloaded to a parallel process and a receive queue is checked for any external commands.

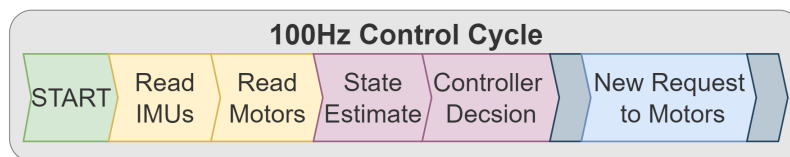


Figure 15: 100Hz control cycle diagram.

The control cycle process is managed from within the `robot.RWIP` class.

4.6 Control Algorithms

Classical control algorithms aim to model a system using equations that capture its physical behavior over time, and then adjust the system with inputs to achieve a desired outcome. We plan to evaluate PID, Linear Quadratic Regulator (LQR) and RL controllers, with PID and LQR being classical controllers that will set the standard for assessing RL controllers.

4.6.1 Classical Controls

In the RWIP we are using a PID controller and our system model was computed using Lagrangian classical mechanics see Appendix D Sections 1-6. To reach our goal of balancing, we specified the desirable outcome to have $\phi = 0$ so that the robot is upright in a typical balanced position, but also $\dot{\theta} = 0$ so that the wheel is not spinning.

The second condition is important, because once the wheel reaches its top speed, any additional torque in the same direction has no effect. Thus if the wheel is standing upright but spinning at top speed, a small perturbation can lead to it falling over without being able to recover.

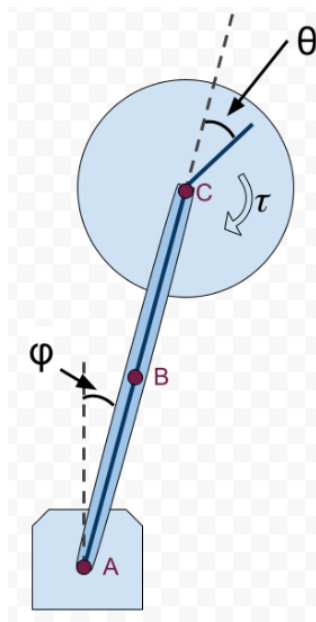


Figure 16: Reaction wheel inverted pendulum diagram.

Figure 17 describes a simplified overview of our control loop, for the full description see Appendix D. First, we read from our sensors to find the difference between our current angle ϕ and our goal (usually 0 for upright position, but not always).

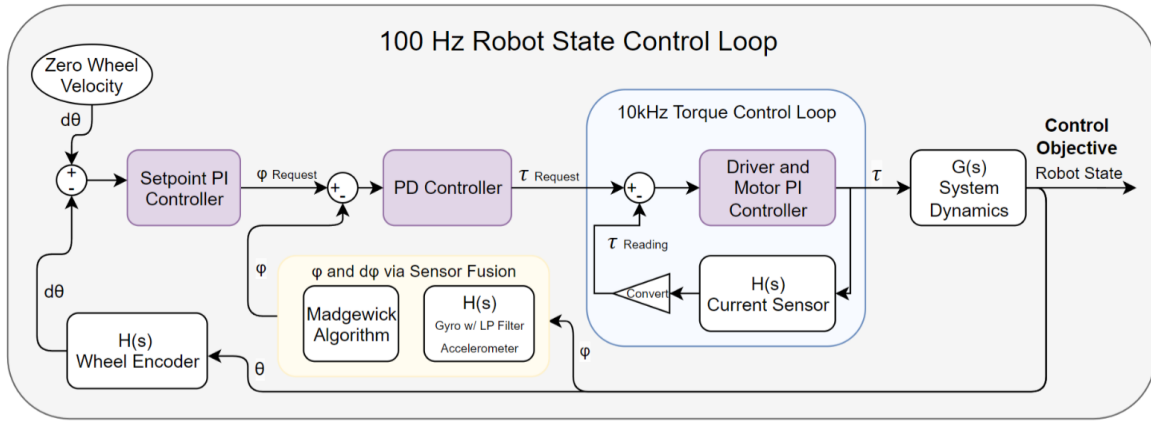


Figure 17: Detailed flow chart of robot state control loop.

Then the PID controller calculates the torque that is needed from the motor and sends this request to the Moteus driver. At the start of the next control loop cycle, we read from our sensors again and repeat the process, converging to our desired balancing state.

4.6.2 RL Controls

Reinforcement Learning (RL) at its core attempts to develop robust controllers using machine learning and the concept of learning through “trial and error.” More formally, an agent (our robot) in a state within some environment takes actions and observes their impact on both its state and the environment 479 Capstone Team 2166 [4]. The agent is controlled by a policy which attempts to maximize a reward function.

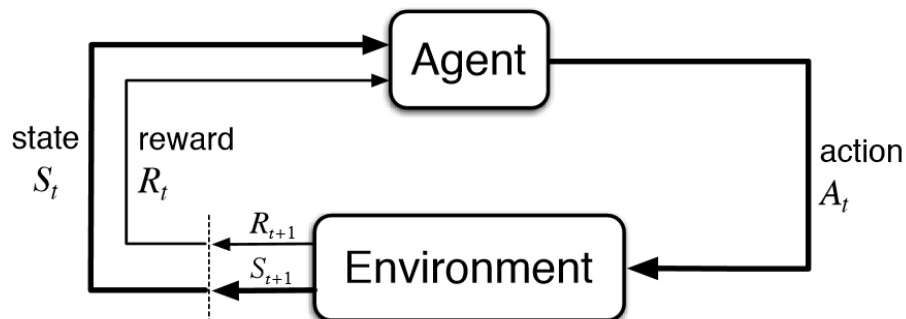


Figure 18: Reinforcement Learning training loop diagram. Source: Deep Reinforcement Learning for Constrained Field Development Optimization in Subsurface Two-phase Flow

While it is possible to train RL agents in the real world, there are a variety of physics simulation packages designed for efficiently training agents on hardware optimized for machine learning. The simulator and

RL framework used in our project is NVIDIA's Isaac Gym. Isaac Gym allows rapid iteration on different agents, reward functions, and policies, and the simulated environment allows exploring different options safely without the risk of damaging your hardware by training in real life. Isaac Gym also allows users to import CAD models directly into simulation which is very beneficial to our development pipeline

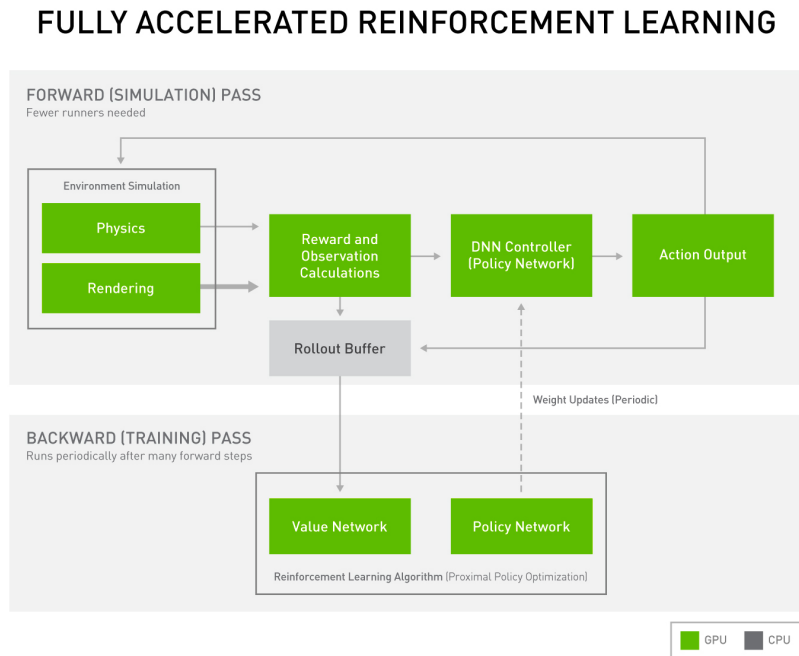


Figure 19: Accelerated reinforcement learning training loop showing forward pass interactions and backward pass optimizations Source: <https://blogs.nvidia.com/blog/deep-reinforcement-learning-gpus-robotics/>

While the concept of training thousands of agents all in parallel within a simulation sounds great, it is not guaranteed that the learned policy will be effective in the real world. One of the primary techniques for bridging the gap between simulation and real life (known as sim2real) is Domain Randomization (DR). DR accounts for randomly varying various parameters in simulation resulting in a policy that can better generalize in the wild [6].

4.6.3 RWIP Reinforcement Learning Control

As a proof of concept for our sim2real pipeline and RL control we designed a balancing policy for the RWIP. Our policy network is designed to output torque requests which integrate well with Moteus drivers. The state space of the model consisted of the pendulum angle and angular velocity as well as the angular velocity of the reaction wheel. After experimenting with several reward functions we settled

on:

$$r(t) = 1 - \theta^4 - 2 * \omega^2 - 0.5|\tau|$$

Where θ is the angle of the pendulum away from vertical, ω is the angular velocity of the wheel, and τ is the torque generated by the motor. The above function was successful as the quadratic penalty on velocity and linear penalty on torque lead to efficient balancing without spinning up the motor to high velocity or drawing high current with dramatic torques. The quartic penalty on angle allows the robot to have some small oscillations about upright which is preferable to the jerkier and less stable control which comes with harsher angle penalties.

In order to facilitate better performance in real life, DR was added to the input observations and output actions of the policy network. A scaling function was also added to the requested torques in simulation so that the policy can learn to account for the effect of angular velocity on achievable torque requests. Finally, reduced episode times prevented the RWIP from spending too much time in ideal situations during training that were not representative of real life (perfectly balanced upright).

This resulted in a policy which could balance effectively forever in real life while keeping wheel velocity below 2 rev/s.

5 Current Progress

The first four months of project development consisted of understanding the physics of the system, designing our robot, and planning our prototyping stages. Starting January 2024, we fabricated our test jigs and developed our software system. By the Project Fair in April, we integrated the sensors and actuators into a common control algorithm on the Jetson Nano, built a single-DOF prototype, and had a PID controller and a Reinforcement Learning controller working.

- Two working controllers, a PID controller and an RL controller
- Each controller can self-right from a resting angle of 27° autonomously
- Both controllers can balance indefinitely and withstand significant external disturbances
- The Jetson consistently holds a 100Hz cycle
- IMU is not drifting

Additionally, the first draft of the complete robot chassis is complete with a mount for the battery, Jetson and yaw wheel.

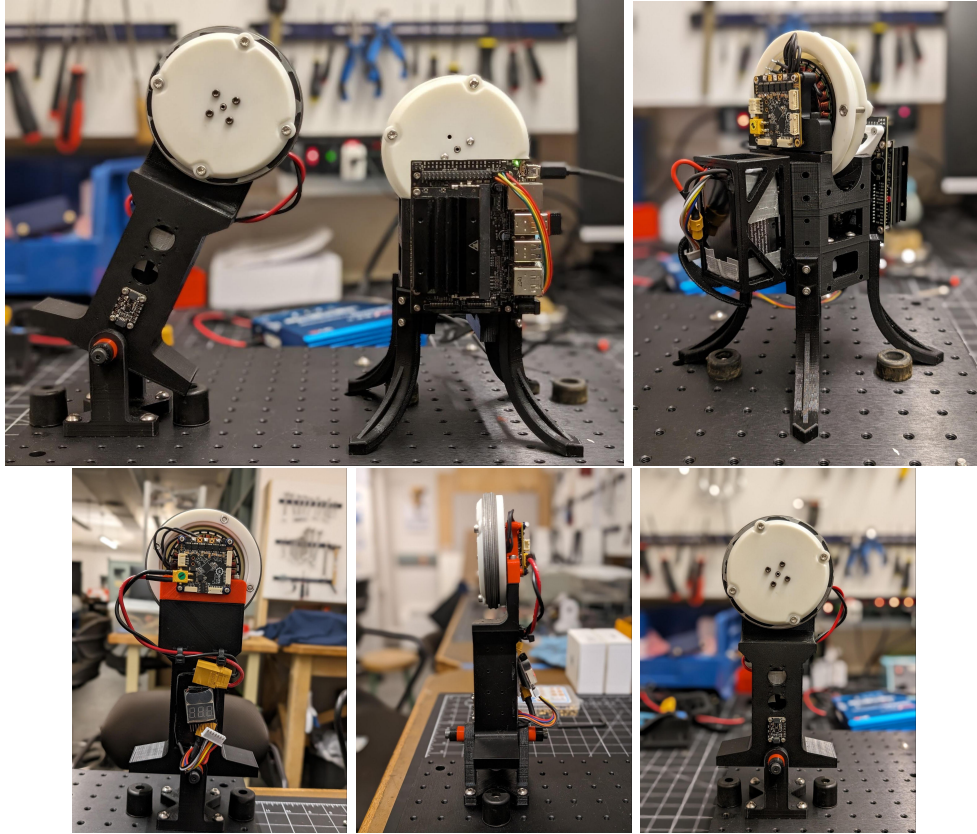


Figure 20: RWIP front, side and back view. Final robot draft chassis.

6 Conclusions

The main challenges in designing our prototype were removing noise and drift from our sensors, understanding the control of our motors for safe operation with lab equipment, and facilitating communication between our drivers, sensors, and microcontroller. The prototype has provided valuable insight and solutions to these challenges, and the lessons learned will carry over to next year's work. We have successfully met all objectives for the prototyping and development stage of the project, planned in the initial proposal:

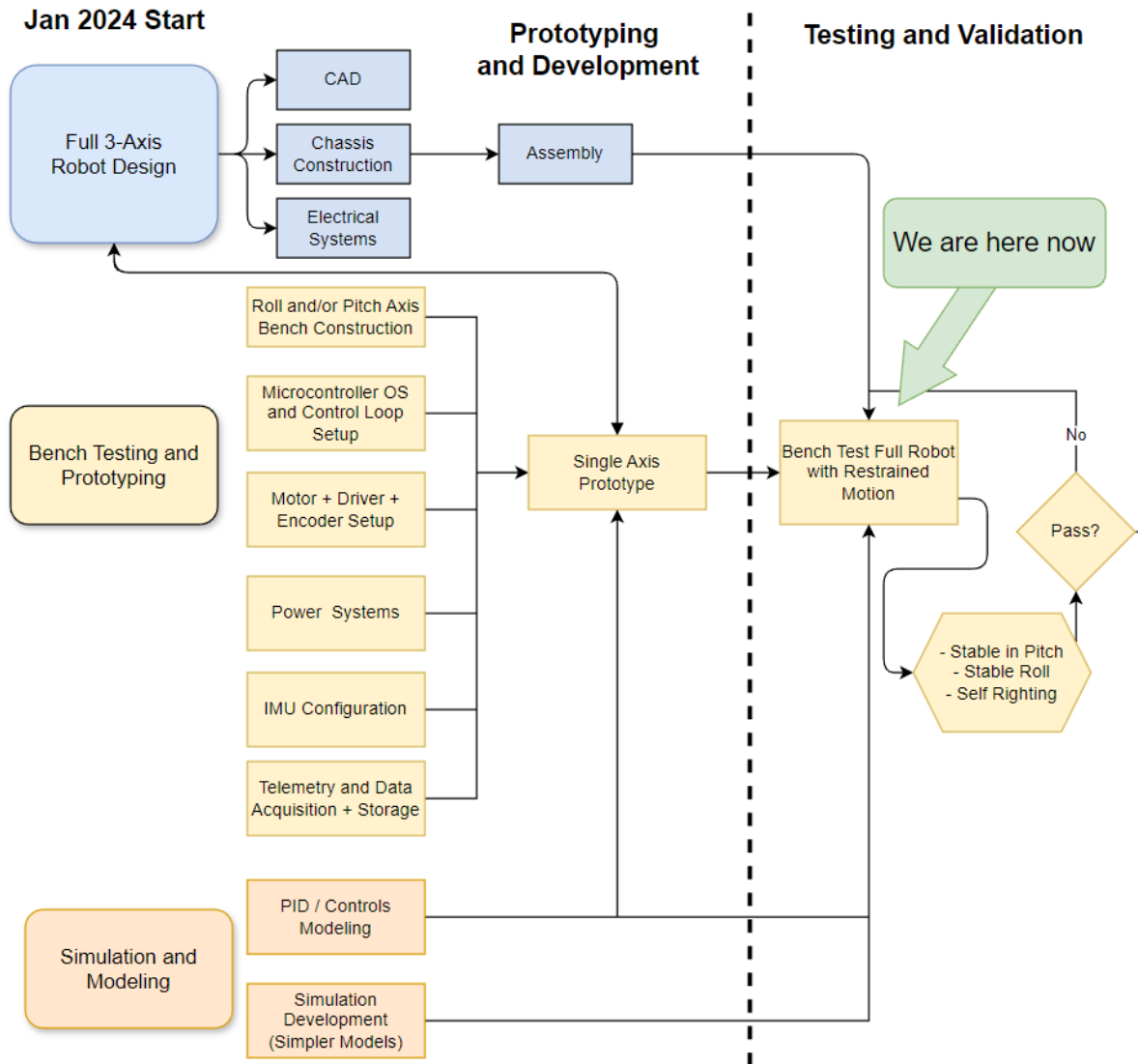


Figure 21: Project flowchart from proposal.

So far, we have observed our PID controller and Reinforcement Learning controller to have similar performance. We expect Reinforcement Learning to have an advantage over PID when we create our full robot next year because of its ability to learn non linear behaviours.

7 Future Development and Recommendations

Next year we will incorporate our findings from the RWIP prototype into the full robot design. We plan to explore other control methods such as LQR, which we expect to handle multiple independent inputs and outputs better than PID based on the findings of the Wheelbot [3]. This will be used to benchmark

the performance of a reinforcement learning controller which we hope will unlock some of the interesting nonlinear control dynamics inherent to angular motion. Finally, we plan to implement point-to-point navigation to extend the capability and compare performance across different control techniques.

We will test and validate the full robot design under restrained motion for the pitch and roll axis, after which we can pursue advanced development, including the use of ground truth external sensors. Our target is to have full RL capabilities on the robot by May 2025.

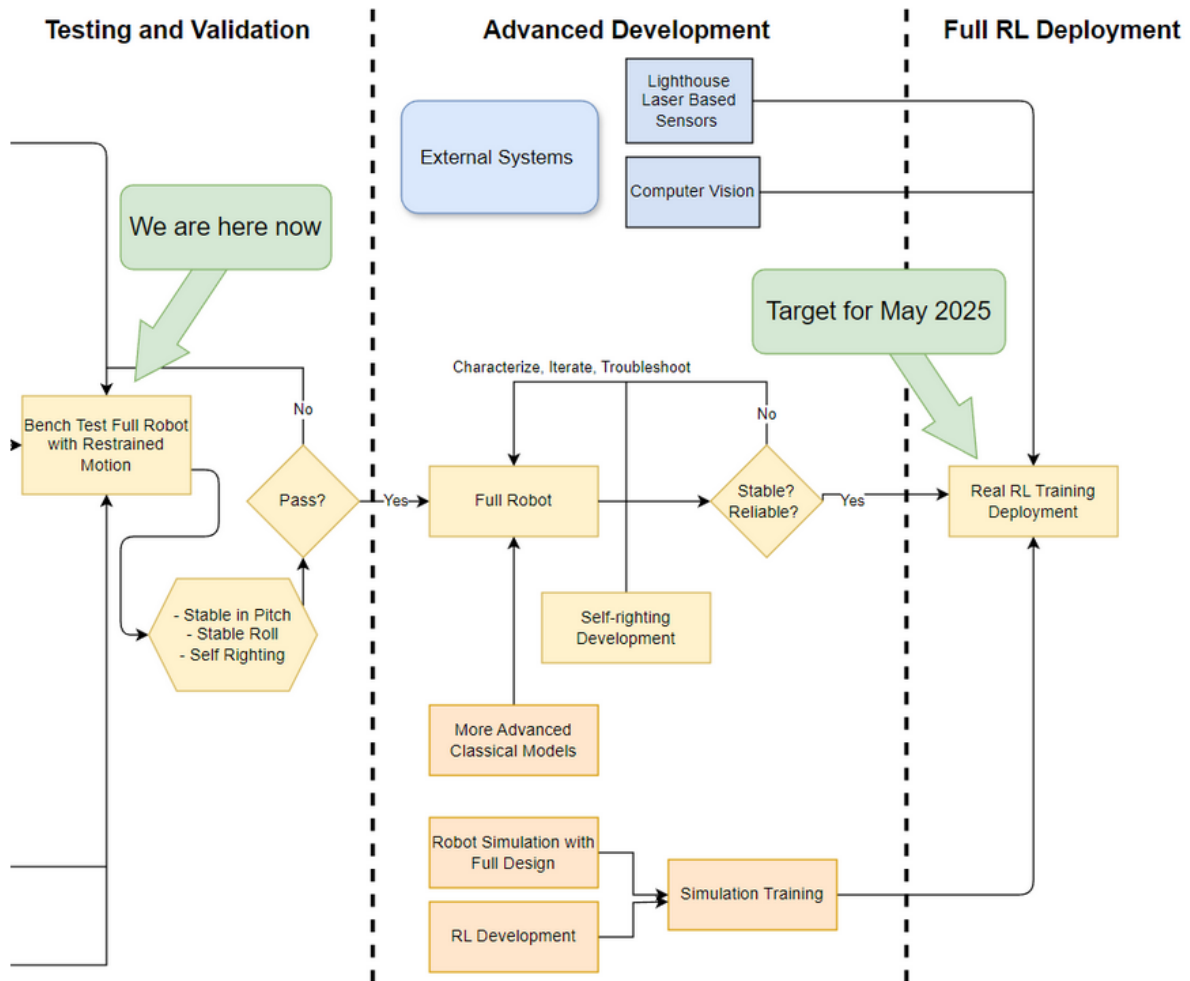


Figure 22: Project flowchart for 2024-2025

Regarding recommendations for prototype design, we recommend designing and fabricating the chassis of the robot early to prototype rapidly. Additionally, we recommend spending time early to implement an E-Stop for safe operation of LiPo batteries.

8 Deliverables

Our deliverables for the project so far are listed the following:

1. Source code: <https://github.com/Team-2411-RL-Unicycle/rl-unicycle.git>
2. CAD files: Main assembly
3. Omniverse Isaac Sim RL Environments Fork: <https://github.com/JacksonnF/OmnisaacGymEnvs>
4. Physical prototype

References

- [1] *AIUR ENPH 479 Capstone Team 1868-69*. Team Project.
- [2] 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.
- [3] A. René Geist et al. "The Wheelbot: A Jumping Reaction Wheel Unicycle". In: *arXiv* (July 2023). URL: <http://arxiv.org/abs/2207.06988>.
- [4] K. Gordon et al. *OpenSim2Real Monopod Platform*. 479 Capstone Team 2166. Team Project.
- [5] 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](https://doi.org/10.1109/ICORR.2011.5975346).
- [6] *TWIP ENPH 479 (Gym2Real) Capstone Team 2153*. Team Project.

Appendix A

Standup Calculations

A numerical analysis of standup 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 ~~~~~
        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 ~~~~~
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 ~~~~~
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 ~~~~~
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')

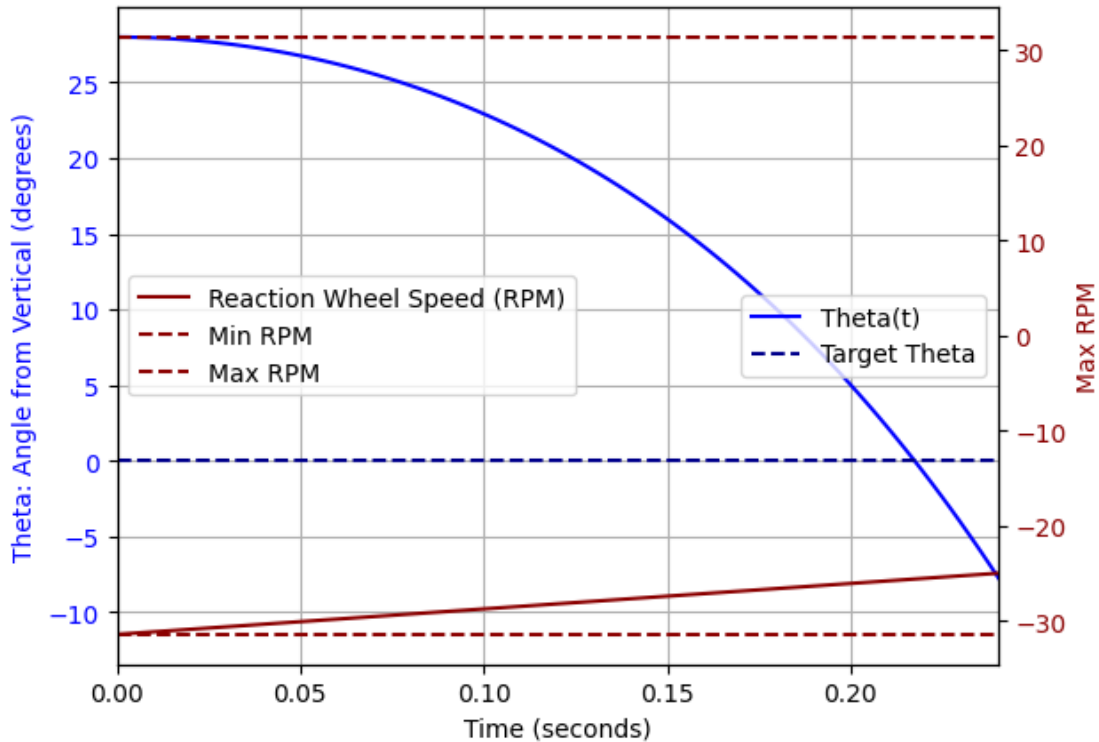
# Create second y-axis for RPM
ax2 = ax1.twinx()
ax2.plot(t, 0.1*ode_soln[:, 2], 'darkred', label='Reaction Wheel Speed (RPM)')
ax2.axhline(y=0.1*-r.max_rpm_rad, color='darkred', linestyle='--', label='Min_
↳RPM')
ax2.axhline(y=0.1*r.max_rpm_rad, color='darkred', linestyle='--', label='Max_
↳RPM')
ax2.set_ylabel('Max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

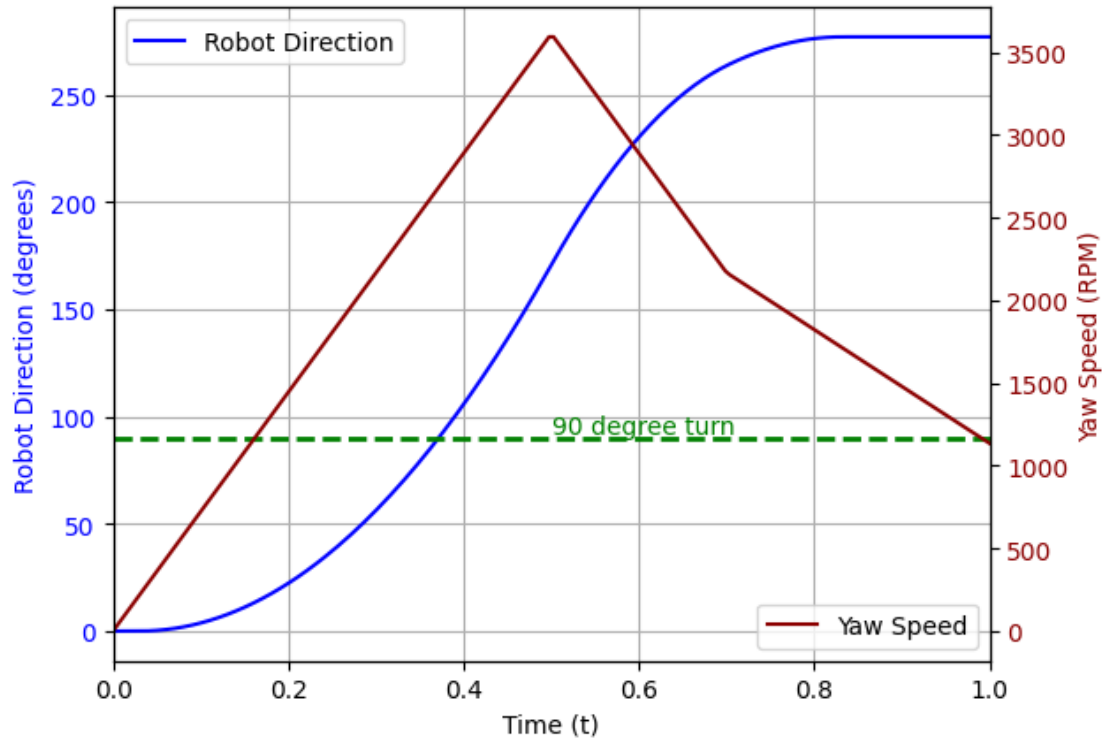
# Add a horizontal line at 90 degrees
ax1.axhline(y=90, color='green', linestyle='--', linewidth=2)

# 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 B

Telemetry and Database Systems

A description of a robotics telemetry server configuration.

Telemetry and Database System Documentation

Simon Ghyselincks, Team 2411

2024-04-15

Purpose

This is a user summary document for our capstone telemetry and database server. It is intended to provide an overview of the different services that are in use for the data pipeline. The capstone project leverages the MING stack as shown in the overview diagram below.

I recommend using a central server to manage all of these services through a Zerotier Virtual Network. This will allow you to access the services from anywhere via the internet without exposing the server to the public.

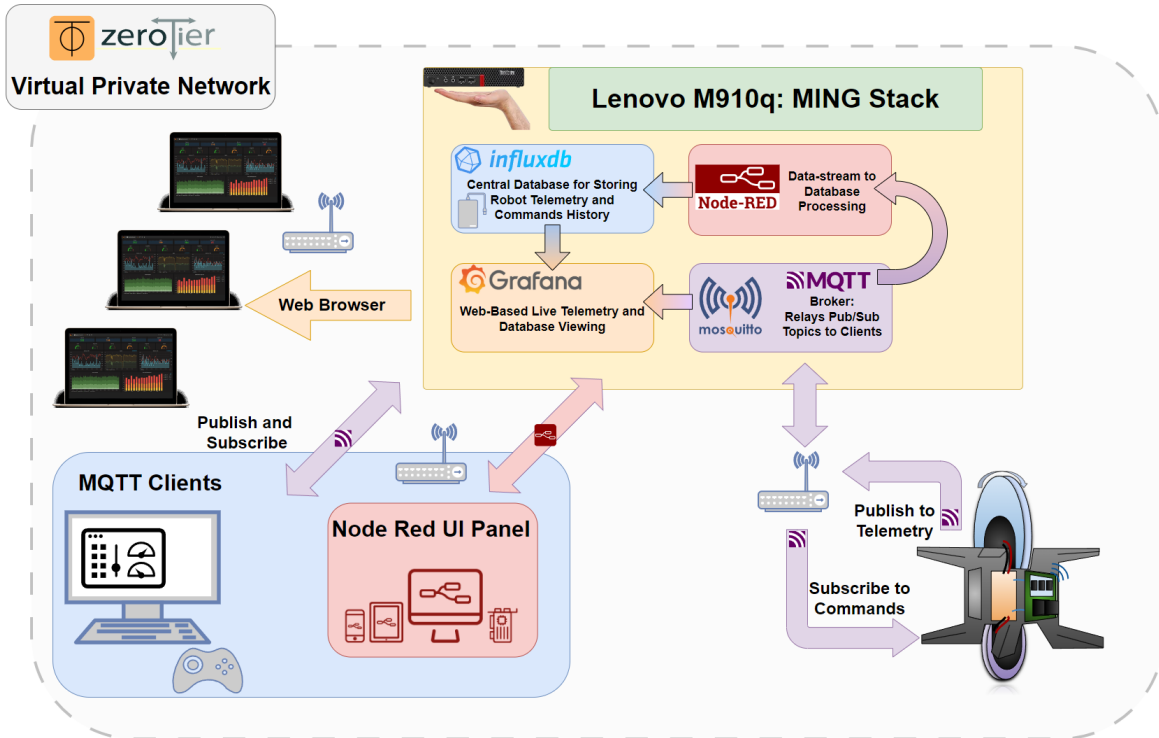
Hardware Recommendation

The Lenovo M900 series of refurbished tiny PCs are recommended as an affordable option that meets the compute needs for a server. The SSD of the device was set to dual boot into Linux Ubuntu 22.04 for the purposes of running a server.

The Raspberry Pi 4B 8GB with an external SSD was tested as a configuration but the requirements are at the limits of the processing power of the device.

Telemetry Services Overview

Telemetry and Control Command Communications



ZeroTier Virtual Network

Zerotier is a virtual network that allows for secure communication between devices over the internet. It is a VPN that allows for devices to be connected to a virtual network and communicate with each other as if they were on the same local network.

To setup a network you should first create a free account at <https://my.zerotier.com/>. Once you have an account you can create a network and add devices to it. The network ID is a 16 digit number that is used to identify the network.

Zerotier Client

The Zerotier client is a software that is installed on the devices that you want to connect to the network. Each device intended for the network including the server should have the client installed. Once it is installed, enter the network ID from the Zerotier website and then approve the device to the network. You may wish to set static IP addresses, especially for the server. This can all be done through the Zerotier website.

<https://www.zerotier.com/download/>

MQTT Overview

The robot on a network is using the MQTT protocol to implement live telemetry. Topics are used on a subscriber/publisher basis. All communication is routed through the Lenovo server that is acting as the broker. The Mosquitto MQTT server software is running on the Lenovo server which is the IP address used for routing messages. The default port for MQTT is 1883.

```
# Define the MQTT settings
broker_address = "172.22.1.1" #Lenovo's IP address
port = 1883
topic = "robot/telemetry"
```

The MQTT explorer offers comprehensive tools to explore available topics and more: <https://mqtt-explorer.com/> This can be a very useful tool for debugging and exploring the MQTT network to check if messages are being sent and received.

MQTT interfaces with Python, Node-Red, and Grafana to provide a comprehensive data pipeline. The MQTT broker is the central hub for all data that is being sent and received. The broker can be accessed by any device on the ZeroTier network that is subscribed to the topic.

MQTT Summary

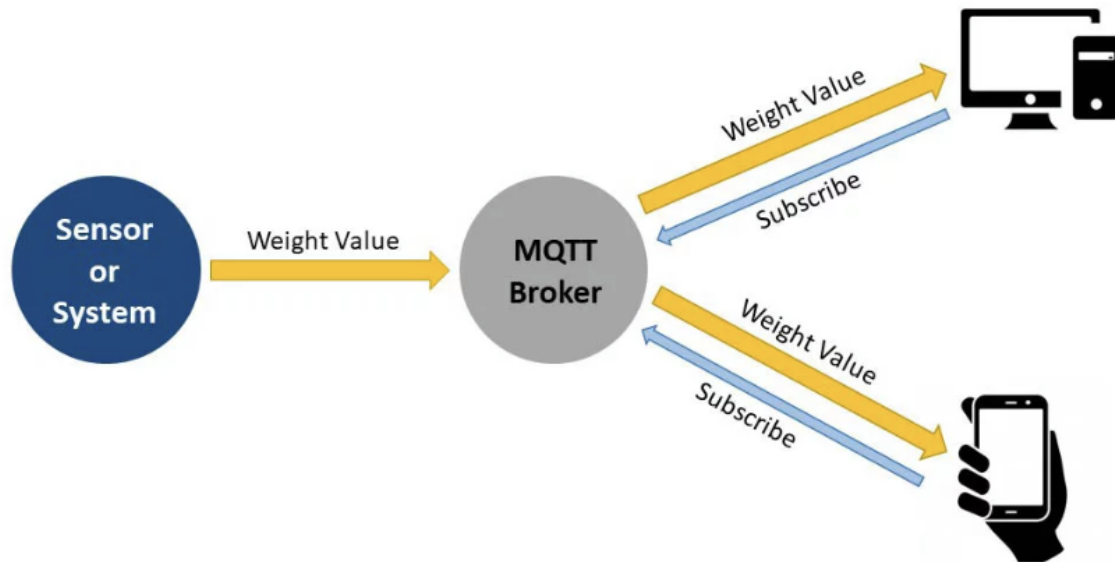
MQTT is a lightweight messaging protocol that provides an efficient and cost-effective method of carrying out telemetry-based communication between devices. It's especially popular in Internet of Things (IoT) applications due to its minimal bandwidth requirements and ease of implementation on hardware with limited processing capabilities.

Key Features of MQTT:

- **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.
- **Retained Messages:** Supports retaining the last message sent on a topic, making it available immediately to new subscribers.
- **Last Will and Testament:** Provides a means for a client to notify other clients about an abnormal disconnection.

Application for our Robot

The Lenovo acts as a broker for all the data that is streaming out of the robot over a Wi-Fi connection to the internet. This offloads the databasing and broadcasting duty from the robot to the broker which can dedicate more resources to data management. The robot can publish data to a topic, which can be picked up by various subscribers such as the Lenovo's Grafana server or other laptops, phones, etc that are connected to the broker and subscribed to the topic.



More Detailed System Diagram needed

The broker can duplicate the published data to many devices in real-time. Another hidden stream for the data is through Node-Red to InfluxDB where the aggregate data can be store more permanently for access to testing records at a later date. Additionally, an MQTT bridge is connected to the Pi's Grafana server which offers an advanced dashboard service for viewing live telemetry. Note that databased telemetry can also be viewed through Grafana which is connected to InfluxDB, but it is a separate data stream from MQTT and should have a separate dashboard.

Publishing Messages

A sample script for publishing messages with **Paho-MQTT client** [Documentation](#) to the Lenovo server/broker while connected to Zero-Tier is provided, see the mqtt.py file. The script allows for publishing a controller value or a cpu_usage data point.

Prerequisites

- **MQTT Broker Setup:** Ensure that the MQTT broker, in this case, the Lenovo server, is up and running.
- **Network Connection:** Connect your device to the Zero-Tier network to ensure visibility and access to the broker.
- **Python Environment:** Make sure Python is installed on your device along with necessary libraries: paho-mqtt, json
- **Broker Details:** In the script, set the broker_address to the Lenovo server's IP address and port to 1883 (default MQTT port).

A client is formed with a ClientID that should be descriptive, we then connect to the MQTT broker A dictionary of values can be converted to JSON format using JSON dump, method. The broker is designed to work with either raw values or JSON The JSON values can be a collection of data types with a Key and Value

The publish method publishes the data to the desired stream where other clients can subscribe to the topic.

Subscribing to messages:

- A Grafana server has been setup where some useful dashboards can be maintained. The **IoT MQTT Panel** for Android also offers some nice services including the ability to send messages.
- More details on subscribing through the **Paho-MQTT client** can be found at <https://www.emqx.com/en/blog/how-to-use-mqtt-in-python> ### QoS The protocol allows for fast transmission of messages and a similar PUB/SUB model as ROS topics. Telemetry read data can be sent using QoS level 0 which only attempts a single delivery of the data and does not wait for confirmation of receipt, this is perfectly acceptable for fast streaming data that is not mission critical.

For mission critical commands such as parameter adjustments and messages to the robot, a QoS level of 2 can be tied to the message which ensures that it is delivered to the robot exactly once. This avoids duplicate commands or lost commands through the network. It is a slower communication protocol but it is not an issue for lower bandwidth messages that are originating from control devices to the robot.

Grafana Live Telemetry and Database Dashboards

Grafana is a powerful open-source platform for creating dashboards and visualizing time-series data. It is particularly well-suited for monitoring and analyzing real-time data. Grafana supports a wide range of data sources and can be used to display both live and historical data in a variety of formats, including graphs, tables, and gauges. Think of it as graph nirvana.

When it comes to viewing the telemetry data, a plugin can be installed to function as a bridge between the MQTT broker and the Grafana server. <https://grafana.com/grafana/plugins/grafana-mqtt-datasource/>

To setup Grafana, install the software on the Lenovo server first. The default port for Grafana is 3000. The program operates through a web browser and can be accessed by navigating to the IP address of the Lenovo server on port 3000.

Once you have logged in, you can add a data source by selecting MQTT from the list of available data sources if you have correctly installed the plugin. The panel will listen to all messages on a particular topic and display them in a graph or table format.

InfluxDB Databasing

The database can be accessed through Python API, through Grafana, or even a direct viewer. The database is also to be installed on the Lenovo server. The default port for InfluxDB is 8086. The database can be accessed through a web browser by navigating to the IP address of the Lenovo server on port 8086.

MQTT gives livestream data but if we want data storage and permanence between runs it needs to be databased. InfluxDB offers this service along with data manipulation services and a special query language. It also includes a data explorer through the web interface.

Node Red

Node Red 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 the wide range of nodes in the palette that can be deployed to its runtime in a single-click.

For this application Node Red is used to bridge the MQTT broker to the InfluxDB database. This allows for the data to be stored in a database for later access. The data can be manipulated and stored in a more permanent format.

The Node Red server is installed on the Lenovo server. The default port for Node Red is 1880. The program operates through a web browser and can be accessed by navigating to the IP address of the Lenovo server on port 1880.

Node Red Dashboard

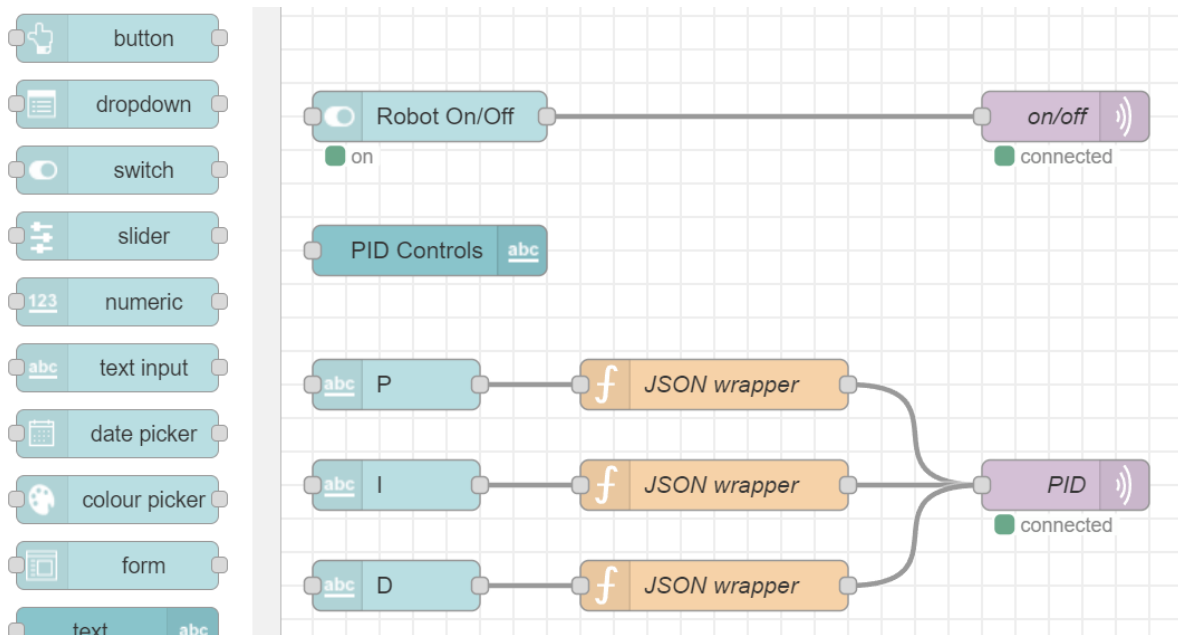


Figure 1: Node Red Dashboard

The Node Red dashboard is an additional feature that is accessed through the Node Red server. It allows for the creation of custom dashboards that can be used to provide a GUI for a robotics project. The GUI can be used to connect directly to incoming signals and also produce outgoing command signals that can be sent to the robot.

Conclusion

These services combined together have proven to be a powerful tool for capstone development. In addition the Lenovo server functions as a valuable workstation for the team. The services are all open source and free to use. The services are also very well documented and have a large community of users that can help with any issues that may arise. Best of luck with your capstone project!

Appendix C

RL Kernel Linux on Jetson Nano

Instructions for enabling the PREEMPT_RT flag for real time control on the NVIDIA Jetson Nano.

RT Kernel Linux on Jetson Nano

Simon Ghyselincks, Team 2411

2024-04-15

Linux RT Kernel Compile Guide

The following guide is intended to provide step-by-step instructions on how to compile a real-time (RT) Linux kernel for the NVIDIA Jetson Nano. The RT kernel is based on the PREEMPT_RT patch, which adds real-time capabilities to the Linux kernel by making it fully preemptible and reducing the latency of the kernel's interrupt handling.

This guide has been modified from some valuable instructions found at: <https://forums.developer.nvidia.com/t/applying-a-preempt-rt-patch-to-jetpack-4-5-on-jetson-nano/168428/4>

Download Source Files and Install Packages

First download the BSP from the NVIDIA website. The BSP contains the kernel source code, device tree files, and other necessary files for building the kernel. The BSP also contains the sample root filesystem, which is used to create the final image for the Jetson Nano. You may wish to look up the most recent version of the Tegra for Linux, in this case we are using R32.7.4.

You can download all of these files onto a Linux machine specifically running Ubuntu 18.04. Another option that has been tested is compiling on the Jetson Nano itself which is running the correct version of Linux by default. For our project we installed 18.04 on a laptop and compiled the kernel there.

i Note

Source Files:

<https://developer.nvidia.com/embedded/linux-tegra-r3274>

Download:

- Driver Package (BSP)

- Sample Root File System
- Driver Package (BSP) Sources
- GCC Tool Chain can also be obtained via the command line:

```
wget http://releases.linaro.org/components/toolchain/binaries/7.3-2018.05/aarch64-linux-
```

Pile all the files into a single directory and install packages

```
sudo apt-get update
sudo apt-get install libncurses5-dev
sudo apt-get install build-essential
sudo apt-get install bc
sudo apt-get install lbzip2
sudo apt-get install qemu-user-static
sudo apt-get install python

mkdir $HOME/jetson_nano
cd $HOME/jetson_nano
```

Extract all of the files

```
sudo tar xpf jetson-210_linux_r32.7.4_aarch64.tbz2
cd Linux_for_Tegra/rootfs/
sudo tar xpf ../../tegra_linux_sample-root-filesystem_r32.7.4_aarch64.tbz2
cd ../../
tar -xvf gcc-linaro-7.3.1-2018.05-x86_64_aarch64-linux-gnu.tar.xz
sudo tar -xjf public_sources.tbz2
tar -xjf Linux_for_Tegra/source/public/kernel_src.tbz2
```

Apply RT Patch

Go into extracted kernel source and apply RT patch

```
cd kernel/kernel-4.9/
./scripts/rt-patch.sh apply-patches
```

Configure and compile:

```

TEGRA_KERNEL_OUT=jetson_nano_kernel
mkdir $TEGRA_KERNEL_OUT
export CROSS_COMPILE=$HOME/jetson_nano/gcc-linaro-7.3.1-2018.05-x86_64_aarch64-linux-gnu/bin/
make ARCH=arm64 O=$TEGRA_KERNEL_OUT tegra_defconfig
make ARCH=arm64 O=$TEGRA_KERNEL_OUT menuconfig

```

The menu config opens an old school BIOS menu. Set the proper settings for the RT kernel:

General setup → *Timer subsystem* → *Timer tick handling* → *Full dynticks system (tickless)*
Kernel Features → *Preemption Model: Fully Preemptible Kernel (RT)*
Kernel Features → *Timer frequency: 1000 HZ*

At this point you can go tamper with device tree files (.dtsi) or other things, next step is the compile stage!

Optional Mods

I tried to modify

```
tegra210-porg-gpio-p3448-0000-b00.dtsi
```

the source file, found using a find file function in terminal. It did not fix things. In general the P3450 model requires the p3448-0000-3449-b00 series of files. This was confirmed by looking at all the source configs and scripts.

Compile

```

make ARCH=arm64 O=$TEGRA_KERNEL_OUT -j4

sudo cp jetson_nano_kernel/arch/arm64/boot/Image $HOME/jetson_nano/Linux_for_Tegra/kernel/Image
sudo cp -r jetson_nano_kernel/arch/arm64/boot/dts/* $HOME/jetson_nano/Linux_for_Tegra/kernel/dts
sudo make ARCH=arm64 O=$TEGRA_KERNEL_OUT modules_install INSTALL_MOD_PATH=$HOME/jetson_nano/Linux_for_Tegra/kernel

cd $HOME/jetson_nano/Linux_for_Tegra/rootfs/
sudo tar --owner root --group root -cjf kernel_supplements.tbz2 lib/modules
sudo mv kernel_supplements.tbz2 ../kernel/

cd ..
sudo ./apply_binaries.sh

```


The image creator requires the device model. For the 4GB Jetson nano it is `-r 300`. This will select the correct dtb:

```
cd tools
sudo ./jetson-disk-image-creator.sh -o jetson_nano.img -b jetson-nano -r 300
```

It is crucial to select the correct device tree since it will not boot otherwise. If you are unsure of which to select, follow through the source code in the `jetson-disk-image-creator.sh` to find what the different flags do. Or try the NVIDIA forums but good luck over there!

Use Balena etcher to put image in `$HOME/jetson_nano/Linux_for_Tegra/tools/jetson_nano.img` onto the SD card

Python in RT

The python script needs to be run with the priority changed from (20) to highest level (99) for it to be RT enabled.

Warning

Running a task at this level could lock the CPU or cause system instability.

Setting Python Scheduling Privileges

Note that for this description our team is using Python 3.8 in a virtual environment, the instructions path files may change slightly if using a different version.

The scheduling priority is a top-level system command and is usually locked behind ‘sudo’. This is problematic when running a Python script because we don’t want to run it as sudo allowing it full access to wreak havoc on the OS. The solution is to grant only the scheduling part of ‘sudo’ to the Python interpreter:

This command only needs to be set once after Python 3.8 is installed (the same in use in our venv): `sudo setcap 'cap_sys_nice=eip' /usr/bin/python3.8`

1. **setcap:** This is a utility that sets or changes the capabilities of a file/executable. Capabilities are a Linux feature that allow for more fine-grained access control; they provide a way to grant specific privileges to executables that normally only the root user would have.
2. **'cap_sys_nice=eip':** This argument specifies the capabilities to be set on the file, in this case, `/usr/bin/python3.8`. It’s composed of three parts:

- **cap_sys_nice**: This is the specific capability being set. **cap_sys_nice** allows the program to raise process nice values (which can deprioritize processes) and change real-time scheduling priorities and policies, without requiring full root privileges.
- **e**: This stands for “effective” and means the capability is “activated” and can be used by the executable.
- **i**: This stands for “inheritable”, meaning this capability can be inherited by child processes created by the executable.
- **p**: This stands for “permitted”, which means the capability is allowed for the executable. It’s a part of the set of capabilities that the executable is permitted to use.

3. **/usr/bin/python3.8**: This is the path to the Python 3.8 executable. The command sets the specified capabilities on this specific file.

Setting Script Specific RT

The ‘RT’ scheduling priority is code 99. Some imported C implementation allows for resetting the scheduling for the process. The function is wrapped in try/except block to ensure it activates.

```
# Define constants for the scheduling policy
SCHED_FIFO = 1 # FIFO real-time policy

class SchedParam(ctypes.Structure):
    _fields_ = [('sched_priority', ctypes.c_int)]

def set_realtime_priority(priority=99):
    libc = ctypes.CDLL('libc.so.6')
    param = SchedParam(priority)
    # Set the scheduling policy to FIFO and priority for the entire process (0 refers to the
    if libc.sched_setscheduler(0, SCHED_FIFO, ctypes.byref(param)) != 0:
        raise ValueError("Failed to set real-time priority. Check permissions.")
```

We run this function at the start of the script which will reassign the scheduling priority to the highest level. This can be verified to work by opening the system monitor and checking the priority of the script such as with htop.

```

1  [|||]          7.9%]   Tasks: 150, 372 thr; 1 running
2  [||]          3.8%]   Load average: 0.23 0.46 0.45
3  [|||||]       15.7%]  Uptime: 11:13:39
4  [||]          2.6%]
Mem[|||||||||] |2.02G/3.86 ]
Swp[|]          0K/1.93G]

```

PID	USER	PR	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
21934	jetson	RT	0	249M	34288	14388	S	15.5	0.8	3:34.00	python main_
21925	jetson	20	0	249M	34288	14388	S	15.5	0.8	3:34.80	python main_
20943	jetson	20	0	8072	4232	3000	R	3.2	0.1	1:10.78	htop
8199	zerotier-	20	0	43448	10348	5028	S	1.9	0.3	1:35.93	/usr/sbin/ze

Figure 1: RT Priority Enabled

Appendix D

1-DOF Equations of Motion and Control Dynamics

A derivation of the equations of motion, transfer functions, and PID controller for a reaction wheel pendulum prototype.

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

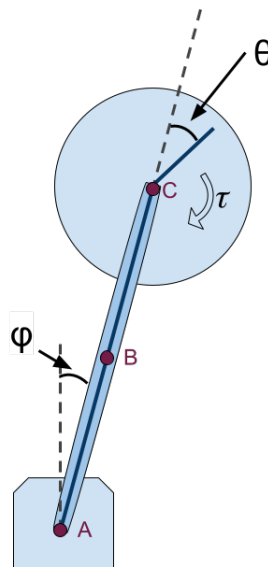


Figure 1: The dynamics of the reaction wheel pendulum.

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

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

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

```
1 % Robot variables
2 syms mp lp Ip mw lw lw real
3 params = [mp, lp, Ip, mw, lw, lw];
4 % Define numerical values for the parameters
5 values = [.531, 0.100, 0.002250, .346, 0.180, 0.000725];
6 g=9.81;
7 % State variables
8 syms phi theta dphi dtheta real
9 q = [phi, theta];
10 dq = [dphi, dtheta];
```



```

11 % Input
12 syms tau real
13
14 % Potential energy mass
15 m0 = (mp*lp + mw*lw)*g; % Effective U=mgh for combined parts
16 % Mass matrix
17 M = [(Ip + mp*lp^2 + Iw +mw*lw^2), Iw;
18      Iw, Iw];
19 lagrangian = (1/2)*([dphi, dtheta])*M*([dphi, dtheta]') - m0 * cos(phi);
20 % Non-conservative forces in each coordinate q
21 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

```

1 % Derive the equations of motion for each ddq
2 [eqs, ddq] = EulerLagrange(q,dq,lagrangian,Q);
3 % Explicit equations:
4 exp_eqs = ddq == eqs;
5 % Solve equations to isolate ddphi and ddtheta
6 ddqSolutions = solve(ddq == eqs, ddq);
7 % Convert solutions to cell array
8 ddqSolutionEquations = struct2cell(ddqSolutions) ;
9 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 generalization 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 . $\hat{x} = 0$ is the chosen linearization point:

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

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 canonical $\frac{d}{dt}x = Ax + Bu$ 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

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \vdots \end{bmatrix} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u}$$

Output Equation

$$\mathbf{y} = \mathbf{C}\mathbf{x} + \mathbf{D}u$$

State Transition Matrix

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

Transfer Functions

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

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

```
1 % State vector of the system, note that Theta is not a state variable
2 % Phi, dPhi, dTheta
3 X = [q(1)' ; dq']
4 % The inputs are non-zero entries of Q (non-conservative forces)
5 U = Q(Q ~= 0);
6 % Vector function for the derivative of the state vector
7 dX = [dphi; ddqArray]
8
9 % Compute the Jacobian matrices to get nonlinear state matrices dX = Ax + Bu
10 A = jacobian(dX, X);
11 B = jacobian(dX, U);
12
13 % Substitute or linearize about an equilibrium point
14 % Define equilibrium point (for example, all zeros)
15 x0 = [0; 0; 0];
16 % Substitute equilibrium values x0 into A and B
17 Aeq = subs(A, X, x0)
18 Beq = subs(B, X, x0)
19
20 % U to X transfer function
21 % dX = Ax + Bu implies sX = Ax + Bu, solve for x = Gtf*u
22 syms s
23 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:

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

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.

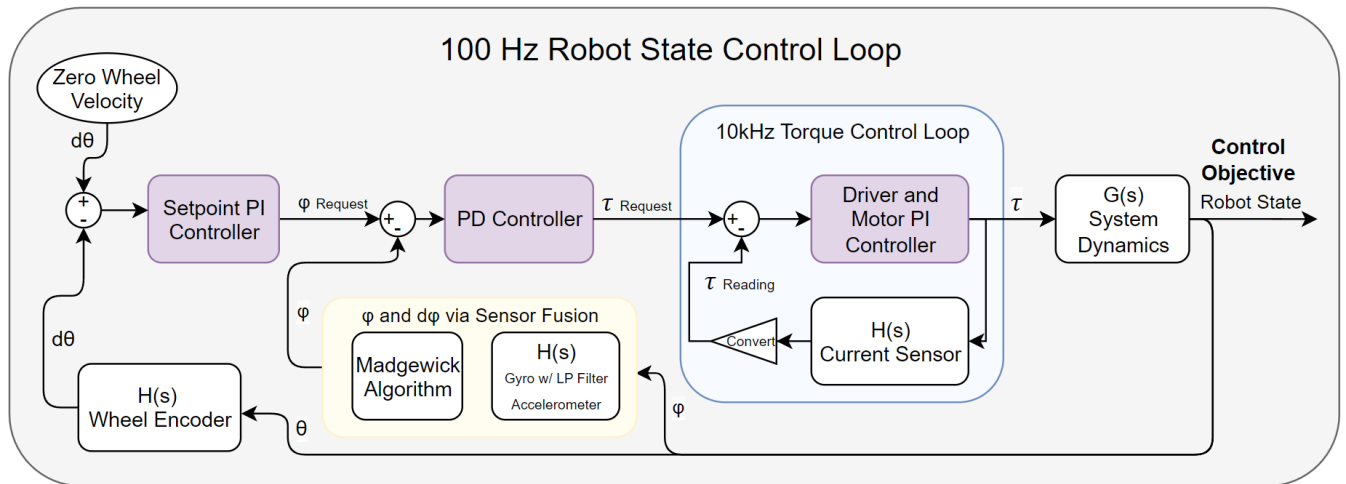


Figure 2: Control Flow Diagram

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 - j $G(s)$ - j $H(s)$ 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>)