Leader-Follower Robot Movement Control

Author: Bradley Harris

Qualification: Final-Year Project – BSc in Computer Science

Supervisor: Prof. Dirk Pesch

Department: Department of Computer Science

Institution: University College Cork

Submission Date: 24th April 2024

Abstract

This project aims to emulate the Leader-Follower paradigm by developing and merging a variety of vision and movement control algorithms implemented on a group of mobile robots. The hardware used comprised two Waveshare Jetbots based on the NVIDIA Jetson Nano and a Raspberry Pi. For the vision system, Apriltags were utilised as real-world markers to track the Leader, whilst the Follower kept the tag at a set distance and in the centre of the camera's field of view. To achieve this an initial architecture based on the ROS robotics software framework was proposed but due to unforeseen issues, a pivot to a Python-based approach was needed. The produced system allowed the Follower to follow effectively without the need for any supporting data to be passed from the Leader. However, the system suffered from some mechanical limitations that caused poor performance on high-friction surfaces.

Declaration of Originality

In signing this declaration, you are confirming, in writing, that the submitted work is entirely your own original work, except where clearly attributed otherwise, and that it has not been submitted partly or wholly for any other educational award.
I hereby declare that:

- this is all my own work, unless clearly indicated otherwise, with full and proper accreditation;
- with respect to my own work: none of it has been submitted at any educational institution contributing in any way to an educational award;
- with respect to another's work: all text, diagrams, code, or ideas, whether verbatim, paraphrased or otherwise modified or adapted, have been duly attributed to the source in a scholarly manner, whether from books, papers, lecture notes or any other student's work, whether published or unpublished, electronically or in print.

Signed:
Date: April 23, 2024

Acknowledgements

I would like to thank Professor Dirk Pesch for providing resources and guidance throughout this undertaking.

Contents

Chapter 1

Introduction

The goal of this project is to demonstrate simple Leader-Follower movement control, with as little communication between the agents as possible by utilising common machine vision techniques and control theory.

Leader-Follower movement control has many applications, from truck convoys moving as a closely coordinated pack on the motorway, coordination on mine sites, agricultural applications such as planting, spraying, and harvesting, and robots in package fulfilment facilities. Being able to coordinate with surrounding vehicles is a key part of driving. As humans with only our eyes, we should be able to keep a constant distance, keep in lane, and not cut corners whilst on the road.

This video gives an example of Leader-Follower behaviour in a real-world application, in this case, truck platooning: `https://www.youtube.com/watch?v=lx9EFJ6qgZc`.

Similar behaviour can be emulated using a robotic platform by effectively implementing Leader-Follower movement, with an emphasis on tracking performance, algorithmic simplicity, and cost. The Leader leads the Follower along a line, using a pre-programmed course, or following another subject. The following robot then uses its camera to track the position, velocity, and course of the Leader in an attempt to emulate, or follow, its path accurately.

Chapter 2

Analysis

"Following" is vector mimicking or replicating the motion of the Leader. In a strict following regime, if the Leader has a velocity vector, the Follower should have the same velocity vector. The Follower should also replay the same steering vector when it has reached that point in the path. A more relaxed following strategy attempts to keep the Leader at a fixed distance ahead, similar to cruise control in a car, and the rear of the Leader in the centre of view. To achieve these approaches, the directional vector of the Leader over time is required at any moment in time. The Follower repeats the vector sequence with a delay.

2.1 User Requirements

2.1.1 Autonomous

For a Follower to be effective in a variety of situations, autonomy is crucial, necessitating no communication or data passed from the Leader. This autonomy enables the Follower to make decisions and adapt to dynamic situations independently, without relying on constant guidance from the Leader. By operating autonomously, the Follower interprets its environment on its own and can adjust its actions accordingly. This reduces the complexity of the system, network traffic, and potential points of failure. Thus, an autonomous Follower that operates independently from the Leader ensures efficiency, reliability, and resilience in the overall system.

2.1.2 Affordable

When designing an affordable system this often means optimising for cost wherever possible. Using one versatile sensor, like a camera, is desirable because it serves multiple purposes. A camera captures visual data which can be processed to gather various types of information such as object detection, tracking, and positional data. By relying on a single sensor, you simplify the system's design and can reduce production costs. With advancements in computer vision technology, a camera can provide versatile functionality without the need for additional sensors. Additionally, the emergence of smartphone camera sensors has driven down the cost, size, and power of these sensors. Using a camera can therefore maximize cost-effectiveness while still allowing the system to perform a wide range of tasks.

2.2 Subsystem Requirements

2.2.1 Movement

In this project Leader-Follower Movement is a 2 dimensional control problem. This simplifies the control domain down to yaw and velocity. For example, the Jetbot platform used in this project is a standard two-wheeled differential drive robot [3]. This means that each motor left and right are driven at different speeds to provide steering to the system. Control of similar systems is a well-documented process and similar systems are deployed in industry and in the home, like a common robotic vacuum cleaner.

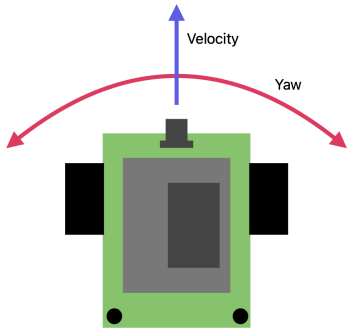

Figure 2.1: Yaw and Velocity

2.2.2 Leader tracking

Tracking the Leader is a critical element of this project. Obtaining the position and direction of the Leader accurately and consistently enables the Follower to maintain tracking and best emulate the path of the Leader. Various sensors can be used to achieve this such as an absolute positioning system, Radar, Lidar, or a vision-based system. Tracking the Leader through space allows the Follower to adjust its

trajectory to attempt to match the path taken. Creating an independent smooth tracking system can enhance the efficiency and safety of group movement in various applications such as autonomous vehicles and robotics.

2.2.3 Wireless Communication

The mobile nature of these robots eliminates the use of wired networking, therefore a wireless network should be utilised. Having a common gateway that hosts a wireless network for the agents to connect to is the easiest approach for developing across multiple systems, providing portability, network isolation, and a control path to each agent. This gateway can also be used to host services and collect performance data from the system.

Chapter 3

Design

This chapter outlines the approach taken to implement the solutions identified in the analysis. The technologies discussed in this chapter are the Raspberry Pi SOC, MQTT, the Jetbot platform, ROS, methods for object tracking, Apriltags and the motor controller.

3.1 Raspberry Pi gateway

To implement a wireless network and provide a simple interface to the agents, the Raspberry Pi[24] was identified as an ideal device to use as a gateway and access point.

The Raspberry Pi is a small, affordable, ARM-based SOC, developed by the Raspberry Pi Foundation. Originally developed to be an affordable system to teach computer science in schools. Running most Linux distributions the system is highly flexible and can be utilised for many applications such as a simple home server to host docker containers in, an IOT gateway, 3D printer control as well as a cheap embedded computer for robotics applications.

Through 5 different revisions, the Raspberry Pi IO has evolved but the standard 40-pin GPIO interface has become a standard across the industry. Alongside this, CSI, Ethernet, HDMI and USB are standard features across the Raspberry Pi generations.

The deep software support and large community that utilises them make the Raspberry Pi an ideal device for learning and tinkering with Linux and embedded computing.

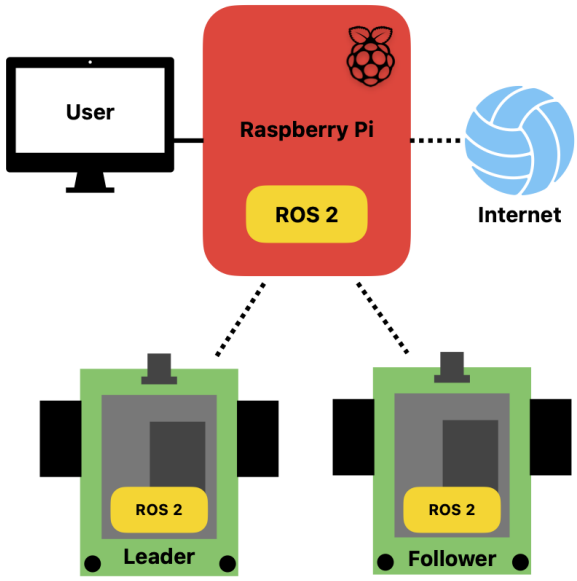

Figure 3.1: The initial architecture

3.2 MQTT

MQTT[14] is a lightweight asynchronous communications protocol and does not require very much bandwidth which is ideal over noisy, unreliable wireless connections. There are compatible libraries in many programming languages for easy adoption and it is non-blocking which is especially useful for its real-time applications. The individual quality of service levels enables flexibility depending on the requirements of the application. There are three quality of service levels:

- "at most once" which offers the lowest level of reliability but the best performance, sending one unacknowledged message;
- "At least once" which expects an acknowledgement back and re-sends the message if no acknowledgement is given, ensuring message delivery but at the cost of duplicate messages;
- "Exactly once" ensures message delivery and eliminates duplicates by requesting to send, receiving a confirmation, sending, and receiving an acknowledge-

ment, but this sequence requires more network overhead and time.

For these reasons, MQTT was identified as a suitable low-latency communications protocol to interface with the agents.

This system was especially helpful when tuning and debugging the PID Control3.7 loops for steering as the individual potential, differential and integral components could be compared and their contribution could be evaluated and altered during runtime. This allowed for specific behaviours to be attributed to the P, I or D value and that attribute could be adjusted accordingly.

3.3 Waveshare Jetbot

The robot platform provided for this project is the Waveshare Jetbot[33], based on the open source specification provided by NVIDIA to promote educational programs around their Jetson Nano system on chip 3.4. This two-wheeled robotics platform provides a cost-effective entry into the development of real-time systems, machine vision and AI. The Jetbot is comprised of a Jetson Nano controller, two DC motors, a small OLED display, a motor drive interface, a lithium-ion battery pack, and a camera. This minimal platform works well for basic educational experiments but has its limitations for more advanced applications.

The OS packaged with the Jetbot from Waveshare provides libraries for interfacing with the motor driver as well as the two Jetbot docker containers. One container is the OLED screen driver that displays useful information about the system. The other container starts a Juypter notebook that runs through a suite of examples, from basic drive control, to path following, and loading a model for object detection. The tutorials in this notebook attempt to teach the basics of neural nets and have a primary emphasis on supervised learning.

3.4 Jetson Nano

NVIDIA's Jetson family of products are a series of system-on-chip devices aimed at the embedded and robotics market for GPU-accelerated and AI applications. The Jetson Nano[17] is the entry-level device, discontinued in 2019, but still provides a good entry point into the NVIDIA ecosystem of embedded devices. NVIDIA supports its Jetson family of systems with the Jetpack releases, OS built for the custom hardware. These can be downloaded as ISO files or built and flashed to the device using the SDK manager.

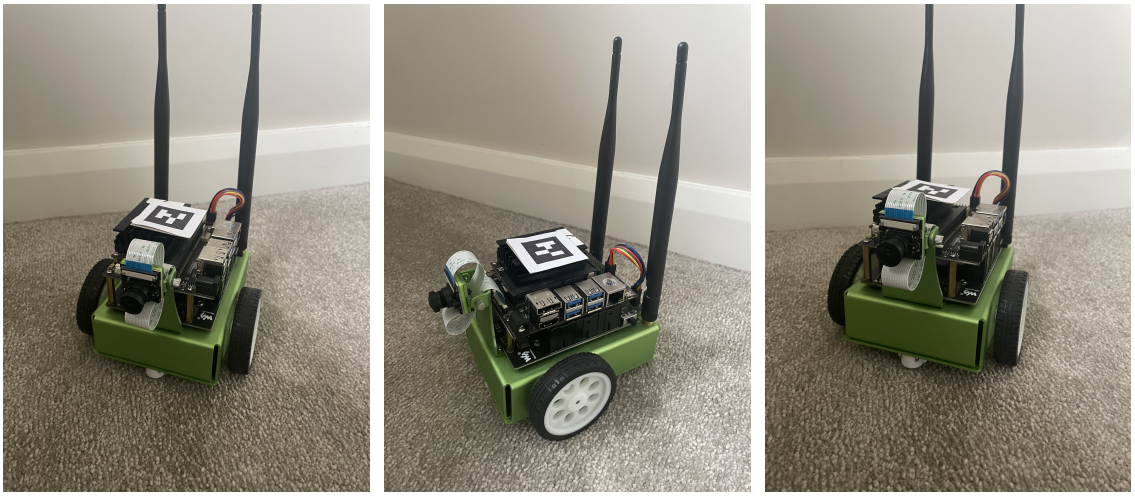

Figure 3.2: Pictures of the Waveshare Jetbot platform with an Apriltag on top

3.5 ROS

The Robotics Operating System[27], or ROS, is a common, widely-supported choice when choosing an architecture for robotics applications as these nodes are portable and are often plug-and-play with standardised message types. This allows for nodes to be treated like libraries or extensions to a system.

ROS is a publish-subscribe architecture, built to develop robotics systems in a modular fashion. ROS2 builds on the initial release of ROS by integrating cross-network topics with flexible protocol choice, a move away from a centralised system of the original ROS to a far more distributed system. ROS2 extends language support from C++ and Python to languages such as Rust and Javascript and incorporates better security features such as certificates and encrypted pipelines.

This node-based architecture encourages a modular design where different scripts handle the different parts of the robot. For example, the drive node may publish encoder values whilst also subscribing to a topic that dictates the wheel speeds. The camera script may publish frames from the camera with multiple machine vision nodes subscribing to it to provide functionality. The distributed nature of ROS2 allows these computationally intensive nodes, like machine vision or neural networks, to be run on a more powerful machine on the same network like a workstation with a power-hungry GPU.

There are many tools and simulation environments created to visualise and ease development. With packages like Nav2[28] to provide a navigation stack and specific packages for Intel and NVIDIA devices. NVIDIA provides a set of hardware-

accelerated ROS packages built for the Jetson family of SOCs. These packages include stereo depth estimation, SLAM, object detection and pose estimation nodes.

ROS is native to Ubuntu but can also be run on Windows and MacOS. It can even be installed on a Debian system like the Raspberry Pi with the correct dependencies installed. Getting ROS to run on the Jetson Nano requires the use of containers installed through Docker. NVIDIA provides a suite of containers called Isaac ROS[18] which are pre-configured to take advantage of the hardware acceleration in their GPUs. Isaac containers are the only way to run ROS on the Jetson family of SOCs and take advantage of the built-in GPU acceleration.

3.6 Object Tracking

Object tracking has many software options with 8 algorithms built into OpenCV[23] that provide coordinate positioning of a selected object. These algorithms require the selection of a body in the frame to track and follow that body through a sequence of frames. The coordinate position of the body can be obtained in every frame all in software. However, the pose of the body cannot be obtained through these means and the tracking can fail part way through. The eight models are of varying accuracy with different strengths and weaknesses from the ability to withstand occlusion of the object to being resistant to larger jumps in position between frames.

As for hardware-based solutions, the simplest way to obtain the current positional vector of the Leader would be with an electronic compass. These types of devices measure the direction or orientation relative to the Earth's magnetic field. Putting one compass on each agent and taking the difference between them, could allow the relative pose of the Leader to be estimated in this 2D scenario.

Alternatively, there are beacon-based options like the ultrasonic Super-Beacon's from Marvelmind[12]. These beacons provide positioning data relative to two stationary base stations. Each beacon can send and receive ultrasonic signals, and contains an inertial measurement unit and a built-in battery. This positional system provides the absolute coordinates of the beacons with the space, defined by the base stations. Boasting a precision of +/-2cm and a range of 50m these are ideal beacons for permanent installations in indoor settings.

However utilising these systems would require the Leader to broadcast the data obtained, directional or positional. The Beacons would also require setup, greatly limiting the portability of the system. Getting the Follower to estimate the Leader directional vector using vision techniques could be a faster and simpler option without any need for communication between the agents.

For this project, image-based pose estimation was selected due to existing camera support on the Jetbot and portability. To support pose estimation fiducial tags were added to the Leader robot.

3.6.1 Apriltags

A commonly used fiducial marker used in machine vision applications is Apriltags. These QR code-like tags are easy to print and deploy in any setting. The detection algorithm is easy to run on embedded hardware and the position and pose of a tag can be obtained. This is an easy way to obtain the directional vector of the Leader.

Developed by researchers at the University of Michigan and released as open-source back in 2010. These versatile tags can easily be printed out and come in a variety of family types that all convey their ID and other attributes in different shapes and sizes. Processing of these markers is lightweight and can be done in real-time, taking the same approach as QR codes. They encode 4 to 12 bits usually representing the ID that allow them to be identified more robustly from further away as less data needs to be extracted. This system can be deployed easily using ROS packages such as [2] or in Python with this library[4].

Apriltags are utilised in robotics, industrial vision and virtual reality applications. The detector provides the decoded ID, the decision margin, a homography matrix, and the centre and corner coordinates within the frame of the tags. By defining the size and camera parameters, detection will also return the rotation, translation, and error of the pose.

Using this information, it's easy to track an agent within a frame and gather useful positional data about the agent or the motion of the camera. The libraries are easily accessible with many exposed features and compatibility with other vision packages such as OpenCV.

3.7 Motor Controller

PID controllers were selected to convert the output of the vision system into values suitable for the motor drivers due to their adaptability and capacity to be adjusted for particular situations.

PID controllers are a common control loop mechanism that utilises feedback to ensure a system converges upon a set point. These controllers are often used in servo motors, heating elements and cruise control systems. The system calculates the error `e(t)` between the current value and the target value continuously. It scales the error by multiplying it by `K_p`, making up the proportional part. The I stands for the

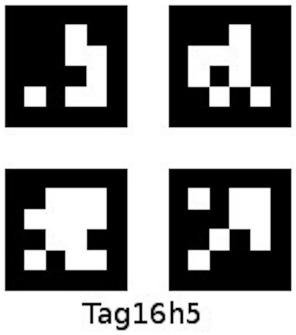

Figure 3.3: Apriltags in the Tag16h5 family with id's 0-3

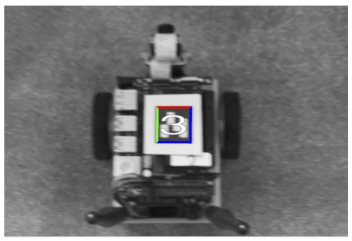

Figure 3.4: Apriltag on the top of a Jetbot

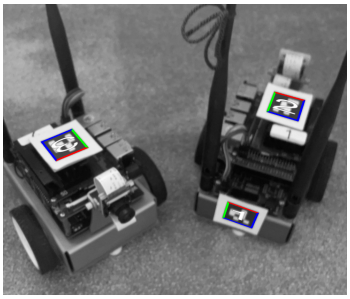

Figure 3.5: Apriltags on the Follower and Leader

integral, the error over time accumulated and scaled with `K_i`. D is the differential of the change in error at every iteration, scaled by `K_d`. All of these parts are added together to make up the output.

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt} \tag{3.1}$$

These controllers are ideal for linear systems but struggle in situations where the set point is continuously changing. When dealing with quadratic systems the PID controller estimates it to be a linear system. They are usually tuned to suit one situation. There are various tuning strategies for the k values from manual iterative tuning to software modeling.

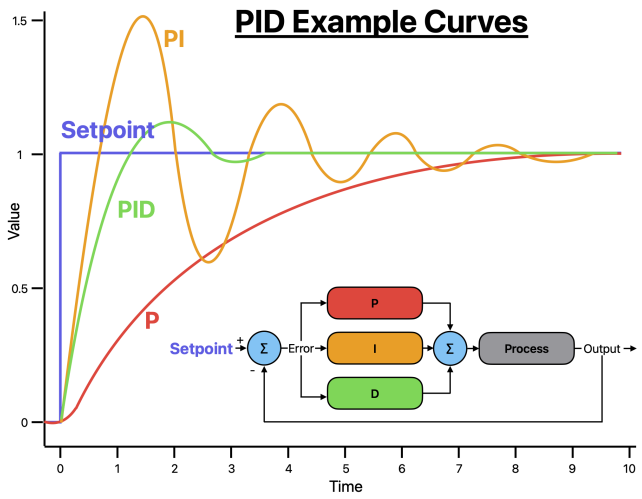

Figure 3.6: A PID graph illustration

To drive the motors the Jetbot library was utilised as it already comes pre-packaged with the robot class and contains all of the logic required to talk to the motor controller over the I2C bus. The robot object it defines utilises the Traitlets library to implement updates on the change of a value. This means that when you set the motor speed it is immediately sent to the motor controller. To encapsulate this class, a ROS node called Drive will be written. The drive node should subscribe to an appropriate topic and drive the motors accordingly.

Chapter 4

Implementation

This chapter discusses the approach and practical execution of the design proposed in the previous chapter. Starting with the initial experiments with the resources packaged with the Jetbots, the implementation of a Gateway using a Raspberry Pi, and the problems encountered when attempting to implement ROS. The pivot to a Python class-based implementation, camera calibration process, and the tuning of the PID controllers in the system.

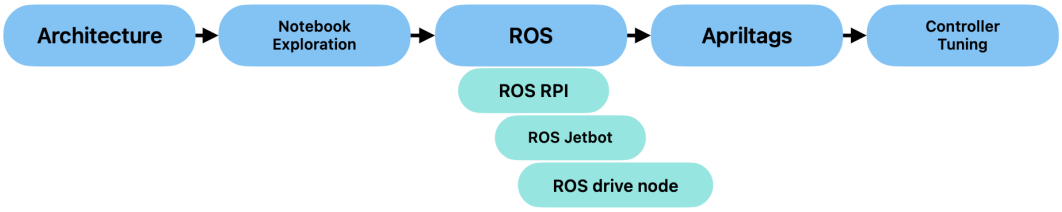

Figure 4.1: The initial road-map for the project

4.1 Initial Experiments

As mentioned previously the Jetbot comes with a Jupyter notebook with tutorials and examples. Choosing to work through them to understand the robot object that the Jetbot library provides. The notebook utilises the Traitlets library[32] to allow for function callbacks on a value change and widgets that allow for interactive sliders and inputs within the notebook.

The notebooks introduce topics such as line following and collision avoidance and prompt the user to gather images and label them to train a neural network. The notebooks do not teach how these networks are trained or how to improve the models, other than obtaining more examples. Typically hundreds of examples are required to get satisfactory results. The object detection notebook utilises a pre-trained model, mobilnet_v2, however, the link for the model is no longer active and attempting to load an alternative model led to crashes and incompatibilities with the hardware.

4.2 Architecture

To utilise a Raspberry Pi as a gateway network interfaces are required. It already has two, an Ethernet port (lan0) and a WI FI adapter (wlan0) built in. The wlan0 can be used to connect upstream to the broader internet, and lan0 can be used as a management port, with a static IP address, for low latency access. A USB wireless card was added to host the wireless access point for the Bots network with a DHCP server for assigning IP addresses. The routing tables were configured to allow for internet connectivity downstream to the Bots network and for SSH sessions to be opened from the management port down to the Bots network.

In addition to this, the Raspberry Pi is powerful enough to run services like ROS to manage and control the agents on the "Bots" network.

4.3 Design Repository

Git was used for version control and distribution when developing. Uploading the code to a GitHub repository `https://github.com/BradleyHarris19/FYP` allowed the code to be distributed to the multiple systems in use. Being able to write code on a laptop, then push it to a repository and pull on each device. This eased the development of programs on devices without a GUI where writing code must be done with command tools like Vim.

`rpi/` contains the scripts that are run on the Raspberry Pi,

4.4 Initial ROS Implementation

Intending on running ROS GUI nodes (like rqt graph) and managing the agents from the Raspberry Pi, installing ROS was necessary. ROS is naively built for Ubuntu-like OS's, so Raspberry Pi OS, which is Debian-based, is missing many dependencies.

The GitHub repository created by Ar-Ray-code[25] provides pre-built images and build tools to install ROS2 on Raspberry Pi. Download the correct script for the OS release, run the script and ROS will be installed.

For the Jetson Nano, it is recommended that ROS2 should be run inside a docker container. This is to abstract the custom hardware away from the software using NVIDIA's hardware-accelerated docker images. These can be downloaded or built using the scripts provided in the repository [8]. This toolkit provides setup and install scripts for a variety of machine learning and AI tools that can compiled into a Docker container. To install the ROS container on the Jetson Nano the following steps were taken:

1. Run `head -n 1 /etc/nv_tegra_release` on the Jetbot to ensure the LT4[1] version is correct.

2. The last Jetpack release for the Jetson Nano is 4.6.4[7] which can be flashed to the board using the SDK manager[31]. The SDK manager must be installed on an Ubuntu-based x86 system.

3. Booting up a Ubuntu VM that allows for direct USB pass-through, and installing the SDK manager, the latest version of Jetpack can be flashed to the EMMC module on the board.

4. Once the OS firmware is altered to allow the SD card slot on the carrier board to be seen by the module, the OS can be moved to the SD card. [11]

5. Now that a suitable OS is built, install the Jetson Containers tools and build the Docker container with ROS inside.

Once the ROS2 docker container is built using the Jetson-containers tools, ROS can be run inside the container.

Utilizing ROS's network capabilities was a key aspect of the project, enabling a more distributed system. When ROS2 nodes start up, they advertise their presence[26] on the network and continue to do so periodically to reach newly added nodes, also advertising when they go offline. While ROS on the Raspberry Pi successfully advertised and detected other nodes on the network, nodes within the Docker container on the Jetbots were unable to do so. Despite troubleshooting efforts such as using `ros2 multicast send` and `ros2 multicast recive` to verify the discovery process and checking network settings and routing tables for anomalies, the issue persisted.

[1] Lt4 is the board support package which includes the Linux Kernel, boot loader, NVIDIA drivers, flashing utilities, and sample file system based on Ubuntu 18.04

Since seamless network discovery was a critical feature of ROS for the project goals, the decision was taken to pivot away from ROS due to time constraints. Given more time a likely solution would be a discovery server, this changes the "mesh" topography to a discovery network with a Central authority as shown in Figure 4.4. A tool like Fastdds provides a discovery server for ROS2[5]

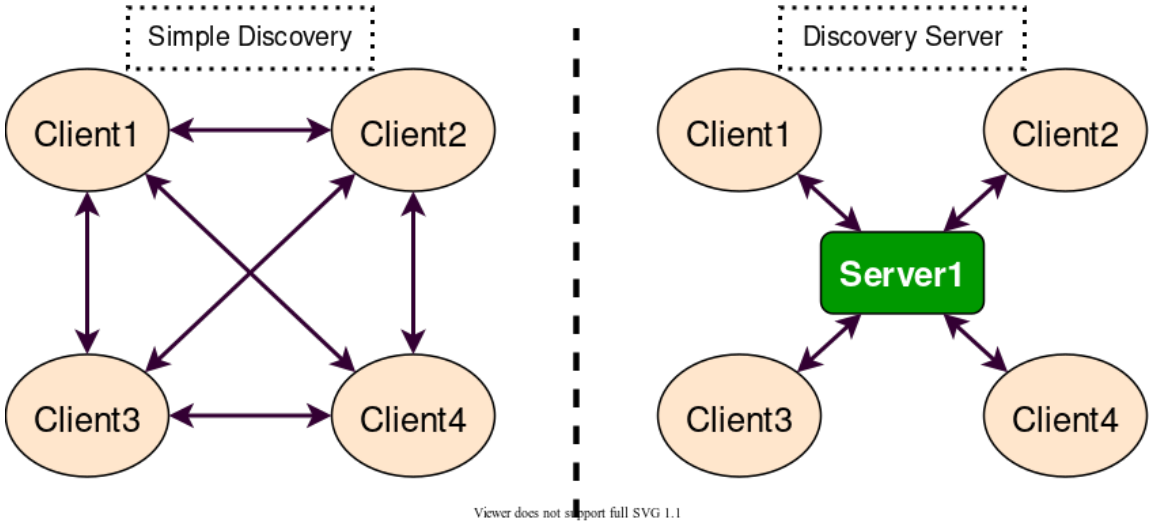

Figure 4.2: An illustration of how a discovery server works from[5]

4.5 Python Implementation

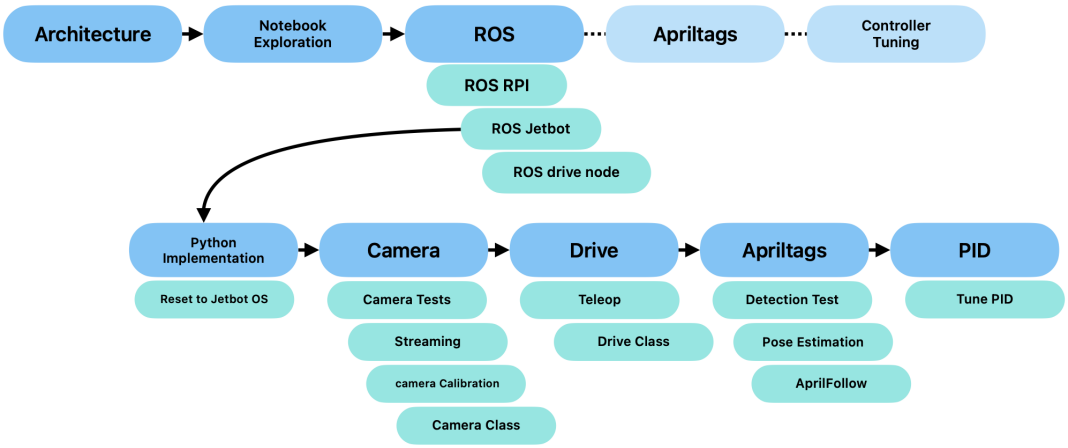

Figure 4.3: The new road map for the project

To move away from ROS, Python was chosen to implement some of the modules that ROS could have provided. Python would have been used regardless as ROS nodes can be written using the language. The extensive libraries such as OpenCV[19], NumPy[15], Eclipse Paho MQTT[21], and Duckytown Apriltags[4] provide excellent functionality that is useful in robotics applications and easy to integrate into programs. The object-orientated nature of Python allows for objects to be defined as a substitute for the nodes in ROS and data can be passed between objects for processing.

To implement cross-network communication for simple messages and debugging, MQTT was chosen as a suitable low-latency alternative. It follows a similar publisher-subscriber architecture as ROS implemented through the use of a broker. The broker is a central node that all messages are piped through, in this project, the Eclipse Mosquitto broker[13] is deployed on the Raspberry Pi. The Jetbots publish their motor values and PID controller values to MQTT topics, to be subscribed to for tuning and debugging.

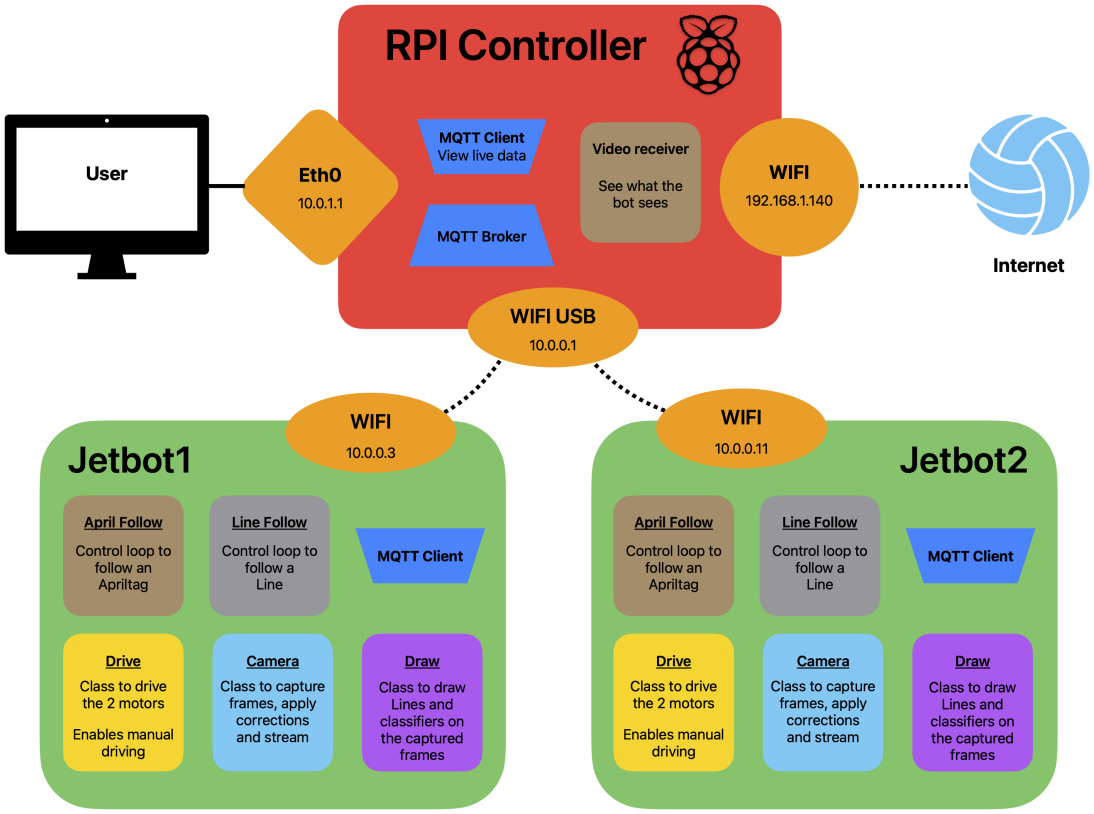

Figure 4.4: An architectural overview of the system

4.6 Camera

To build an interface for the camera, first, the script `testScripts/capture.py` was created to test and gain familiarity with the G-streamer pipeline. G-streamer[6] is a standard pipeline-based multimedia framework that supports a variety of camera interfaces and codecs. Each element in the pipeline performs a specific task, like reading files, decoding data, or displaying the video. An exclamation mark separates each element, and this pipeline command can be run directly in the terminal or passed to a video capture command as a string in a script.

This script opens a pipeline, captures a frame, saves it as a JPEG, and then sends the file to the Raspberry Pi, utilising the SCP command, for viewing and validation

in a desktop GUI. This image file can be utilized by other scripts for testing purposes at a later stage.

Streaming a live feed of what the agent can see is important when debugging vision issues or when attempting to interpret anomalous behaviours. To test streaming a live video feed from the robot, `testScripts/camtest.py` was developed to open a network port and make the frames from the capture pipeline available. The script `receive.py` in the `rpi` folder receives the video stream. This script aided in adjusting the resolution of the frames to optimize latency over the wireless connection. Another feature developed was a fallback port when the network stack does not clean unused ports fast enough once one stream ends and another begins.

All cameras are intrinsically different, despite being the same model with the same connector and adjusting the lens to the same focal length, there will be differences in the images they produce. This is especially true for cheap phone sensors such as the IMX219 Provided in the Jetbot kit. Fortunately, there are software corrections that can be applied, once the intrinsic parameters of the camera are obtained.

The calibration process is outlined in this article by Satya Mallick `https://learnopencv.com/camera-calibration-using-opencv/` and by using a version of his script at `calibration/calibration.py`, calibration was conducted on the Jetbot camera.

The script produces a matrix of intrinsic parameters for the camera, distance coefficients and vectors of both rotation and translation of the captured frame. The rotation and translation vectors are particular to the image calibrated but the distance coefficients and camera intrinsic parameters can be used to un-distort future images. Some calibration schemes, like shown in 4.6, utilise a series of images to average and verify the intrinsic parameters, These schemes encourage frames at multiple angles, tilt and roll with respect to the checkerboard. The matrix of intrinsic parameters is a matrix of shape 3,3 with only two pairs of values, the focal lengths expressed in pixel units, X and Y and the principal point that is usually at the image centre.

Calibration is typically performed on a checkerboard, of known dimension and square count, by obtaining the intersections of the squares and with the assumption that it is a flat plane in space the image can be corrected in software. This process is explained well in the navigation stack for ROS[29].

After producing these calibration matrices they are stored as a .npy file for future use in other scripts.

An object named `Camera` was developed to enable access to the camera for other scripts. Consolidating insights from the outlined scripts, the `src/camera.py` file was created. This file facilitates camera access and manages the transfer of image data.

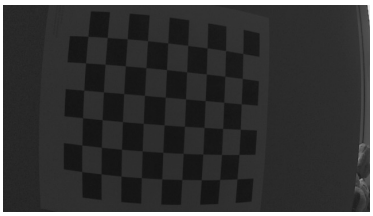

Figure 4.5: Checkerboard on a wall

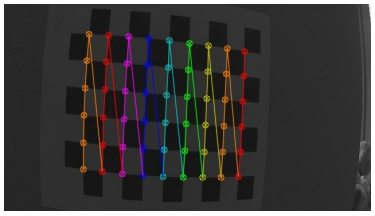

Figure 4.6: Checkerboard with calibration points

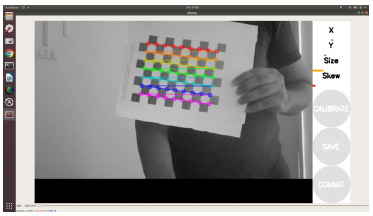

Figure 4.7: Checkerboard calibration with Nav2 tools

It defines a class for the camera object, establishing the pipeline to the camera and enabling configuration to open a web socket for video streaming. The class includes a `read` function for capturing and undistorting frames based on the calibration matrix. Additionally, the `stream` function is implemented to stream images over the socket connection. When executed independently, the file streams camera output over the opened web socket connection.

4.7 Drive

The Jetbot Library provides a `Robot` class that is utilised within the Jupyter notebooks, abstracting the process of writing PWM values to the motor driver. This was achieved asynchronously using the traitlets library.

Initially, a basic drive script was created, `movetest.py`, to turn the motors on and drive forward for 1 second. This was followed by `teleop.py`, a script to control the Jetbot using a controller. The Gamepad library[22] facilitated connecting to and managing interrupts from the controller. The script then utilized inputs from the joystick to drive the two motors. These scripts initialized the `Robot` object from the Jetbot library, enabling the writing of speed values ranging from -1 to 1 to control the motors. This abstraction enabled the I2C bus communication with the motor driver to be hidden from the higher-level logic.

The Drive object within `src/drive.py` handles all interactions with the motors. It instantiates the `Robot` object and adapts the individual left and right motor control into a velocity and yaw component, which are combined to drive the motors as follows:

$$\text{left\_speed} = \begin{cases} \text{self.forward} + (\text{self.steering} \times 1.0) & \text{if self.steering} > 0 \\ 0 & \text{else} \end{cases}$$

$$\text{right\_speed} = \begin{cases} \text{self.forward} - (\text{self.steering} \times 1.0) & \text{if self.steering} < 0 \\ 0 & \text{else} \end{cases}$$

When run standalone the script allows for driving the Jetbot with the provided controller directly. Utilising the same code from the `teleop` script. It also broadcasts MQTT messages to topics `{jetbotid}/drive/{variable}` for the left (L) and right(R) motor speeds, as well as the overall speed scalar(S).

4.8 Apriltags

The Duckietown Apriltag library selected[4] is a Python wrapper around the April-Robotics library[1] which is written in c. The library provides a detector object that can be run on grey-scale images that will return a list of Apriltags identified. The detector object is initialized with the Apriltag family for detection, along with parameters like the number of threads to utilize, image sharpening and, other configuration settings. Each detection in this list contains information about the tag such as id, decision margin and the center and corner coordinates of the tag.

The script `apriltagTests/apriltest.py` was written to run the detector on JEPG images, this was used to gain a better understanding of the capabilities and limitations of the detector.

The `aprilimage.py` script captures a frame from the camera, applies the detector to identify tags, draws bounding boxes around the detected tags, labels them with their IDs, and marks their centres using OpenCV for visualization. Extending this `apriltagTests/aprilCam.py` incorporates camera streaming to do live detection. This was used to test decision margins and detection distance, tuning the detector configuration parameters to best fit the environment, camera and compute resources. Filtering out 'hallucinations' was achieved by utilising a decision margin threshold in conjunction with specifying a certain id. This worked well as for an id to be obtained, the tag needs to be decoded.

In conjunction with the parameters obtained through the camera calibration process, these scripts were modified to return the pose of the Apriltags detected. To obtain this from the detect function the `estimate_tag_pose` argument must be set to `true` and the camera intrinsic parameters and size of the Apriltag must be provided. Then an accurate homography[20], rotation and translation matrix of the tag is returned. From this data, the directional vector of any tag can be obtained.

The compromise of decoding the pose of a tag is the extra computing resources required to do so, this increases the loop time down from on average 30 milliseconds (30fps), the same frame timing as the camera, down to 80 milliseconds (12fps). The

2D data was enough to achieve accurate following so the decision was taken to move forward without the pose of the Apriltags.

`src/AprilFollow.py` is a control script that implements the Leader-Follower movement control by following an Apriltag. It utilises the previously mentioned Drive and Camera classes and incorporates the findings of the Apriltag test scripts to best detect Apriltags in the frame. It takes one Apriltag that is above a certain confidence threshold to filter out tags that are hallucinated. It takes the tag's centre point and calculates the horizontal position of the point relative to the centre of the frame, represented as a float between -1 and 1. This value is passed to the steering PID loop with a setpoint of 0. This system attempts to keep the Apriltag in the centre of the frame. To control speed, the length of the tag's right side is obtained. The further away the tag is the smaller that length becomes, and the greater the speed should become. The set point of the velocity PID loop is the length of that side at a distance of approximately 10cm.

Live PID components are broadcast over MQTT as they change for tuning. `{jetbotid}/{PID control name}/pid/{variable}` in, out, p_term, i_term, d_term, P, I, D

4.9 PID tuning

To tune the PID control loops a manual approach was taken, by individually tuning the P, I and D values testing the changes and iterating again. The values from the PID controllers were transmitted over MQTT in real-time, this helped correlate specific behaviours with the value that caused them.

Tuning the steering controller was the first step. Placing the bot on the floor putting an Apriltag on a stick and tuning to get the bot to always face the tag. The PID controller takes in a value from -1 to 1 with the setpoint at 0, the centre of the frame, as can be seen in figure 4.8. The controller outputs a yaw or steering value to the drive class to power the motors accordingly. Starting by scaling up `k_p` to gain the response needed to turn but without overshooting too badly. This worked well, however the motors lacked the response as the output value shrank, making it most of the way to facing the tag but stopping a few degrees short. This is where the integral comes in useful, as the error does not decrease, the I increases to gain the desired response. This has the effect of jerking the bot past its setpoint if `k_i` is too high. Here `k_d` is used to tame the sudden increase in `k_i`. Another side effect of utilising the Integral term is that if tracking is lost it can keep accumulating. The motors are not written to when tracking is lost but this means that when tracking is re-gained a large value is written to the motors. To solve this all that is required

is to reset the integral term when no tags are detected.

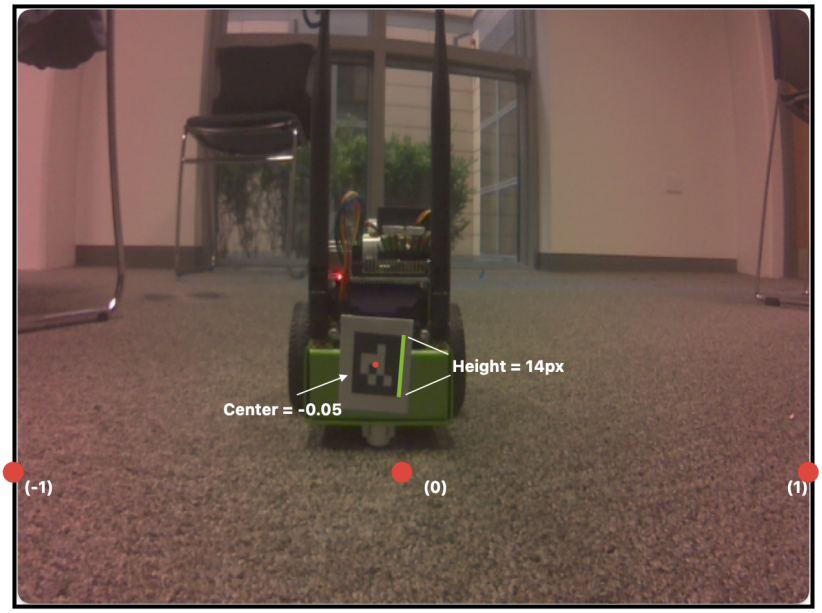

Figure 4.8: How the vision system utilises the image

Adding a constant velocity to both motors improved the steering control making small adjustments easier as the voltage was high enough to get finer response from the motors. To control the distance to the Apriltag in front we can take the vertical length of one side of the tag, shown the Figure 4.8 with the green line. Just using the proportional part of the PID controller, the setpoint can be defined in pixels, and this system can slow down the Follower as it gets closer to the rear of the Leader.

Chapter 5

Evaluation

5.1 User Requirements

5.1.1 Autonomous

The Follower performed its task independently with no communication with the Leader. All of the data obtained from the leader was obtained with the camera through the use of Apriltags as fiducial markers. Steering to keep the tag in the centre of the frame and controlling the speed to maintain the size of the tag. This implementation is flexible and deployable in a variety of environments, allowing for effective autonomous following in any location.

5.1.2 Affordable

The Jetbot platforms used in this project cost approximately two hundred and fifty euros each. The critical components of one of the agents are the Camera, SOC, battery and drive train. All of these components are readily available and an equivalent system can be constructed for under one hundred euro. A large component of the cost of the Jetbot is the Jetson Nano computer which could be substituted with a Raspberry Pi Zero to reduce the cost. Many software solutions have arisen because hardware expenses are significant, as hardware inherently requires initial financial investment. For example, inferring the distance to an object by measuring the size of a line or the method of simultaneous location and mapping (SLAM) for mapping out a space is a replacement for Lidar sensors. These methods can be implemented completely in software, with only camera sensors, but can be quite computationally intensive and often not as effective as a real sensor can be.

5.2 System Requirements

5.2.1 Movement

Creating an appropriate movement control scheme was easy as the differential drive control scheme is common, simple and well-understood. The `Drive` class handled the motor interface effectively, abstracting away the individual motors to the velocity and yaw controls of the domain.

The PID controllers used to control the velocity and yaw were difficult to tune but provided fantastic results, enabling the Follower to follow the leader effectively. PID controllers perform best in closed-loop systems with direct feedback from the actuator they are controlling. This system took in a value and setpoint from the current camera frame and controlled the motors to get the value to converge to the setpoint, this setup is not ideal for PID controllers

Another disadvantage that PID controllers had in this system was that the setpoint was contiguously changing from frame to frame. This prevented the system from truly converging, the error very rarely reached zero. This created a situation where, if tracking of the Leader was lost, the integral term continuously increased and introduced erratic motion when re-obtaining the tag. This was solved by resetting the steering values if tracking was lost.

Tuning these controllers was difficult and time-consuming due to the need to physically move the Jetbots back into position to try again. Modelling the system or running software tools is not an option due to the lack of encoders and support. With more experience, this process becomes easier and over time, with the live values, the signs of an incorrect K_p, K_i or K_d can be correlated as the system reacts too aggressively, ramps up too fast, or overshoots often.

5.2.2 Leader Tracking

The tracking of the Leader is a critical part of the system. This was achieved through the use of Apriltags. Apriltags were an effective solution to the problem of obtaining positional and pose estimation of the Leader. The detection algorithm effectively provides the coordinate positions of the centre and corners of every tag in the frame. The confidence of each detection is provided so the less certain detections can be filtered out and hallucinations reduced. Once the camera intrinsic matrix is obtained from the calibration process and the size of the tag is provided, the detector provides the pose of the tags detected. The Apriltag-based system chosen has a lightweight detection algorithm that could be effectively run on the hardware of the Jetson Nano without the need for hardware acceleration. The algorithm could be configured to

utilise all 4 cores on the CPU, with other options such as image refining that can be minimised to reduce the compute load. This system could keep up with the frame rate of the camera without affecting the frame pacing. As mentioned above the detection algorithm is prone to hallucinating as its sole job is to find Apriltags, luckily it provides a confidence value per detection so unconfident detections can be filtered out. Motion blur in the frames caused the majority of the issues when tracking a moving target. Jerky movements in the Leader or fast turns when close can cause the tag to not be detected. The detection system's domain was defined by the field of view of the camera, if the field of view was larger, the tracking around sharp corners would improve as the Leader would remain in view for longer. This larger FOV comes with the trade-off that the distance calculations become more angular.

5.2.3 Wireless communication

Setting up and utilising the Raspberry Pi as a gateway and access point was made significantly easier with the abundance of tutorials, blogs and form discussions surrounding development on the device.

The USB WiFi adapter that hosted the "Bots" network, required drivers to be installed and occasionally would not mount correctly on boot. This was avoided through the use of a script that would rebuild the driver and mount the dongle, restarting the wireless network.

Running the MQTT broker on the Raspberry Pi was ideal as the data from both agents could be accessed from the Gateway. Debugging the systems that published messages could be performed easily as data points could be correlated with the movement behaviour of the robots.

5.3 Platform

The Jetbot platform was adequate for this project. The following sections examine the impact of the different components of the platform.

5.3.1 Jetson Nano

The Jetson Nano SOC has reached the end of support with NVIDIA. The result of this is that it has been left behind the latest OS and software updates that make it comparable with some software packages.

Having to run almost every application inside a container to gain access to hardware acceleration on the custom hardware

The last Jetpack release for it was 4.6.4 which utilises Ubuntu 18 in 2019. Moving onto the Jetson Orion and other newer family members which are more advanced and starting at over twice the cost of the Nano. With no price equivalent replacement for entry into the NVIDIA embedded ecosystem, this unsupported SOC is the only option for teachers or hobbyists. Development with it was aided by the community tools such as the JetsonHacks GitHub repositories[9], blog posts[11] and historical form posts that solved issues that arose[16].

5.3.2 Jetbot

Camera

The camera provided with the Jetbot kit worked well when run at default settings. The G-streamer pipeline was fully featured and easy to learn, however some configurations that the camera advertised did not function. The camera suffered from quite a lot of motion blur and when attempting to alter the exposure to mitigate this issue, the camera often outputs static or no signal.

Figure 5.1: Loss of tracking over 4 frames due to motion blur

Battery

The three 18650 lithium batteries could sustain the Jetbot for about six hours of development. Continuously running the Jetson Nano and occasionally driving the motors, the battery performed sufficiently and was never a bottleneck.

Motors

The motors that drive the wheels of the Jetbot are TT motors that are simply driven with DC from 3-6 volts which makes them very common in schools and affordable robotics platforms. They have plastic shells and gearboxes, come in a single or dual-shaft variant, and occasionally come with optical encoders. The wide voltage range and slip clutch inside make them resilient to mistakes in electronics or mechanical design. The Jetbot has two of the single-shaft variants with no encoders.

A side effect of the plastic construction of these motors is a loose gearbox, which can worsen with wear and tear over time or be due to manufacturing defects. When driving the two motors in the Jetbot at the same voltage, the expected behaviour would be for the agent to move forward in a straight line, however, the motors in each bot were not matched in this way. Driving both motors at the same value caused a drift to the left or right as the bot travelled forward. The difference was very small, but noticeable as illustrated in figure 5.3. The PID controllers were always fighting this drift.

Another Side effect of how these motors were driven is that the PWM signals passed to the motor driver do not correlate linearly to the actual speed of the motors. This caused more issues for the PID controllers as they expected a system that could be estimated to be linear. In deployment, this causes poor reactions on surfaces with high friction like carpets where when trying to drive the motors at lower speeds means that not enough voltage is provided to the motors to move them. Performance is best on smooth surfaces such as tiles or wooden boards as there is less friction for the wheels and castors. A good solution to this would be a PID controller for each motor that controls the velocity of each wheel. This would require installing encoders on each wheel for feedback. This way the setpoint can be defined and the real velocity value can be read off of the encoders ensuring the wheel spins when on higher friction surfaces. A PID loop can still be utilised as an overarching controller setting set-points for the motor PID controllers. This tiered approach creates a more abstract view of the hardware platform, enabling more confidence in the drive system for the programmer. In conjunction with the addition of encoders, upgrading the motors to ones with metal gearboxes or better voltage response would also improve the drive train significantly.

Figure 5.2: The TT motors driving the wheels

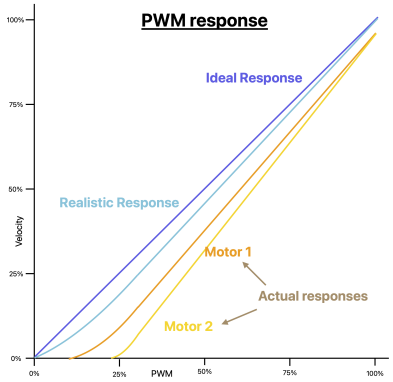

Figure 5.3: An illustration of the motor responses

Wheels

The wheels on the Jetbot were rubbery but thin. They lost grip instantly when they collected debris on the floor which further impacted the effectiveness of the motors. The small screw that held these wheels on was ineffective and this led to the thin wheel wobble and tilt.

Castors

The castors on the front and rear of the bot that keep it balanced on the two drive wheels are small ball bearings in plastic housings. These also became defective when small particles found their way into them and friction between the plastic and steel ball increased. These small wheels also regularly got caught on the grout between tiles and slightly raised obstacles such as tape on the floor. When this happened the wheels could not get the bot unstuck and intervention was often necessary.

Chapter 6

Conclusions

In this project, Leader-Follower movement control was achieved using machine vision with Apriltag fiducial markers and PID controllers using the Waveshare Jetbot platform.

6.1 Work Conducted

Implementing this project required a portable network that was implemented with the Raspberry Pi gateway and the "Bots" network it hosted. The ROS software stack was a path that did not pan out as foreseen and the project was pivoted to use a class-based approach in Python. The vision technique utilised used Apriltags as real-world markers that could be used to track the Leader accurately. PID controllers are used to control the speed and steering of the Follower to make the system reactive to changes in the leader's position.

6.2 Learning's

Ideal hardware platform characteristics include a rigid body and motors with a linear response. It is impractical to expect a perfect hardware platform, therefore software solutions like PID controllers and encoders are available to ensure that the hardware behaves as expected. A model of the system can be constructed which can be worked on to produce an effective system.

Delving into the machine vision side of this project, working through concepts like motion blur, resolution, frame rate, and exposure led to a familiarity with the limitations and advantages of using cameras. Using Apriltags in this project was a

suitable option for tracking the Leader, as they portray a lot of data, and can be used to support a much wider set of applications.

Using PID controllers for motor control was an appropriate choice, and learning how to work with and tune them provided valuable experience, particularly learning a strategy for control theory and how important live data and response analysis can be in a dynamic system.

6.3 Future Work

Future work would involve improving the existing Jetbot platform with better motors and encoders. This would enable a tiered approach for the PID controllers, allowing better control over the platform. Implementing the same control scheme using ROS and extending the capabilities of the agents through the use of packages such as Nav2 would allow the system to conduct mapping, localization, autonomous navigation, obstacle avoidance, and enable fleet movement.

In addition, Apriltags could be utilised to gain the heading of the Leader relative to the Follower, allowing a consistent orientation to be maintained or the Leader's path to be predicted.

Bibliography

[1] AprilRobotics. The original AprilTag Library. Retrieved from https://github.com/AprilRobotics/Apriltag

[2] AprilRobotics. AprilTag ROS Package. Retrieved from https://github.com/AprilRobotics/apriltag_ros

[3] "Differential wheeled robot" Wikipedia, The Free Encyclopedia. Available: https://en.wikipedia.org/wiki/Differential_wheeled_robot.

[4] Duckietown. Duckietown AprilTags Library. Retrieved from https://github.com/duckietown/lib-dt-Apriltags

[5] eProsima Fast DDS Documentation. ROS 2 Discovery Server. Retrieved from https://fast-dds.docs.eprosima.com/en/latest/fastdds/ros2/discovery_server/ros2_discovery_server.html

[6] GStreamer Documentation, "GStreamer Python Bindings," [Online]. Available: https://gstreamer.freedesktop.org/documentation/index.html?gi-language=python. [Accessed: April 22, 2024].

[7] NVIDIA. Jetpack SDK version 4.6.4. Retrieved from https://developer.nvidia.com/Jetpack-sdk-464

[8] Dusty Robotics. Jetson Containers Repository. Retrieved from https://github.com/dusty-nv/Jetson-containers

[9] JetsonHacks. JetsonHacksNano Repository. Retrieved from https://github.com/JetsonHacksNano

[10] NVIDIA. Jetson Nano Developer Kit. Retrieved from https://developer.nvidia.com/embedded/jetson-nano-developer-kit

[11] Youyeetoo Forum. Jetson Nano Recognizes the SD Card and Starts with SD. Retrieved from https://forum.youyeetoo.com/t/jetson-nano-recognizes-the-sd-card-and-starts-with-sd/186

[12] Marvelmind Robotics. Marvelmind Starter Set Super-mp-3D. Retrieved from https://marvelmind.com/product/starter-set-super-mp-3d/

[13] Mosquitto. Mosquitto MQTT Broker. Retrieved from https://mosquitto.org/

[14] MQTT.org. Retrieved from https://mqtt.org/

[15] NumPy. NumPy Documentation. Retrieved from https://numpy.org/doc/stable/user/index.html

[16] NVIDIA Developer Forums. Solution: dpkg error processing package nvidia-l4t-bootloader-configure. Retrieved from https://forums.developer.nvidia.com/t/solution-dpkg-error-processing-package-nvidia-l4t-bootloader-configure

[17] NVIDIA. Jetson Nano Developer Kit. Retrieved from https://developer.nvidia.com/embedded/jetson-nano-developer-kit

[18] NVIDIA. NVIDIA Isaac ROS Package. Retrieved from https://developer.nvidia.com/isaac-ros

[19] OpenCV. OpenCV Website. Retrieved from https://opencv.org/

[20] OpenCV. Homography Tutorial. Retrieved from https://docs.opencv.org/4.x/d9/dab/tutorial_homography.html

[21] Eclipse Paho. Paho MQTT Python Client Documentation. Retrieved from https://eclipse.dev/paho/index.php?page=clients/python/index.php

[22] Piborg. Piborg Gamepad Repository. Retrieved from https://github.com/piborg/Gamepad/tree/master

[23] PyImageSearch. OpenCV Object Tracking. Retrieved from https://pyimagesearch.com/2018/07/30/opencv-object-tracking/

[24] Raspberry Pi Foundation. Raspberry Pi 5. Retrieved from https://www.raspberrypi.com/products/raspberry-pi-5/

[25] Ar-Ray-code. Raspberry Pi Bullseye ROS2 Repository. Retrieved from https://github.com/Ar-Ray-code/rpi-bullseye-ros2

[26] ROS Wiki. ROS Discovery Documentation. Retrieved from https://docs.ros.org/en/iron/Concepts/Basic/About-Discovery.html

[27] ROS Wiki. ROS 2 Rolling documentation. Retrieved from https://docs.ros.org/en/rolling/index.html

[28] ROS Wiki. ROS Navigation Stack documentation. Retrieved from https://navigation.ros.org/index.html

[29] ROS Wiki. ROS Navigation Stack Tutorial: Camera Calibration. Retrieved from https://navigation.ros.org/tutorials/docs/camera_calibration.html

[30] Youyeetoo Forum. Jetson Nano Recognizes the SD Card and Starts with SD. Retrieved from https://forum.youyeetoo.com/t/jetson-nano-recognizes-the-sd-card-and-starts-with-sd/186

[31] NVIDIA. NVIDIA SDK Manager Documentation. Retrieved from https://docs.nvidia.com/sdk-manager/introduction/index.html

[32] IPython. Traitlets Library. Retrieved from https://github.com/ipython/traitlets

[33] Waveshare JetBot AI Kit. Retrieved from https://www.waveshare.com/wiki/JetBot_AI_Kit