Individual Project
Final Report

ID Number: **F223612**

Programme: **Robotics, Mechatronics and Control Engineering (RMCE)**

Module Code: **24WSC357**

Project Title: **Control system to balance a robot on a moving sphere**

Abstract:

This project aimed to develop a control system enabling balance in an existing Ball Balancing Robot (BBR). Through iterative troubleshooting and explorative software development, the robot achieved near stable performance, showcasing the potential for full autonomous balance. Key hardware limitations were noticed with the existing robot, mostly due to an underpowered microcontroller. The switch from Arduino Uno to ESP32 resolved all identified issues including lack of program memory, slow loop speed and inaccurate sensor data. Onboard Digital Motion Processing was used with an ICM20948 board for increased speed and reliability of orientation data. A 2D simulation environment was developed modelling the system in one plane which gave insight into the working dynamics and various control theories. While the final implementation used a basic PD controller, research was completed into how other methods like LQR, or reinforcement learning could be applied to this context. The project highlights the feasibility and complexity of Ball Balancing Robots and provides a clear foundation for future control development.

1 Table of Contents

2 Introduction

Ball balancing robots (BBRs) pose an interesting engineering challenge and have become increasingly common since their conception in 2005 [1]. They were originally designed to counter the difficulties faced by traditional statically stable mobile robots and offer new approaches to human-robot interactions. Wheeled statically stable robots must have large bases if they are to have any height comparable to a human, as otherwise they are at high risk of falling over when accelerating [2]. This makes them poor for tasks that involve working alongside people or in human environments. Inspiration was taken from humans and animals to create a dynamically stable system instead, capable of remaining upright at greater heights from constant micro adjustments. BBRs are one solution to this idea with the core concept involving a free rolling ball with a robot balanced on top. The single point of contact with the ground allows the robot to move omnidirectionally without first needing to turn. This offers more agility compared to other dynamically stable systems such as segway type robots [2]. However, this comes at the cost of increased control complexity, as the system must balance in all directions at all times.

The aim of this project was to build upon a robot inherited from a previous project, and design and implement the control system to enable the robot to balance on a bowling ball. While in theory the robot as inherited had all the necessary components to complete this task, many difficulties with the hardware were encountered throughout the project that needed to be overcome before any progress was made towards the final goal. As a result, a significant portion of this project involved troubleshooting and problem-solving rather than focusing on new control theory implementations. This report will begin by summarising some background information relevant to the project, alongside analysis of previous similar projects. Then, after a brief description of the provided hardware, it will detail a simulation environment that was created for testing, followed by the methodologies taken to iteratively improve the system. Finally, this report will outline some of the testing that was carried out before presenting results and evaluating.

3 Background

3.1 Control Theory

Ball balancing robots pose a significant control challenge due to their high instability and nonlinear dynamics. The controller is an integral component within the system and therefore finding the correct approach is a crucial step in making the robot functional. It is possible to model the system as an inverted pendulum in multiple planes [3]. A variety of techniques have been attempted by previous projects with some common examples including PID, LQR and RL. This section will analyse and criticise the potential options and their applications in the context of this project. Some crucial statistics to control systems are shown in Figure 1.

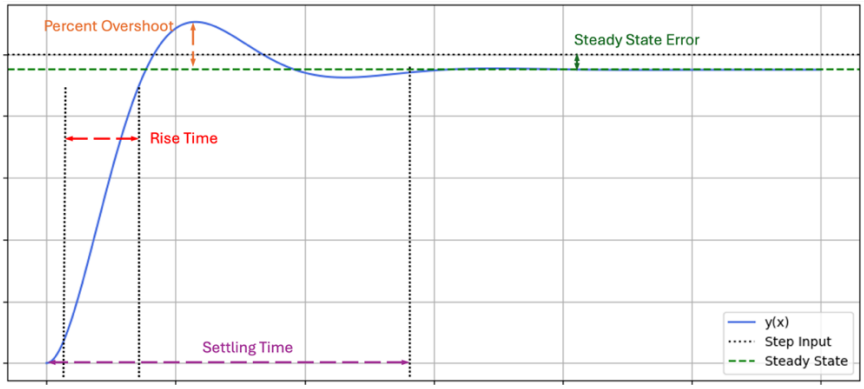


Figure 1 - Response of a typical closed loop control system

3.1.1 PID Control

PID (Proportional, Integral, Derivative) control is the most common form of control system. They are universally accepted in industry as the standard due to their robust performance and functional simplicity [4]. They operate within a closed loop system as illustrated by Figure 2, where an error value, *e* is continuously measured and calculated from a reference value, *r* – either fixed or variable. The integral and derivative by time are calculated from *e*, which put together with the error makes up three core signals. These are in turn multiplied by tuned K constants before being summed back together and fed into the plant. The resultant effect is then measured and used to calculate the next error values in a feedback loop.

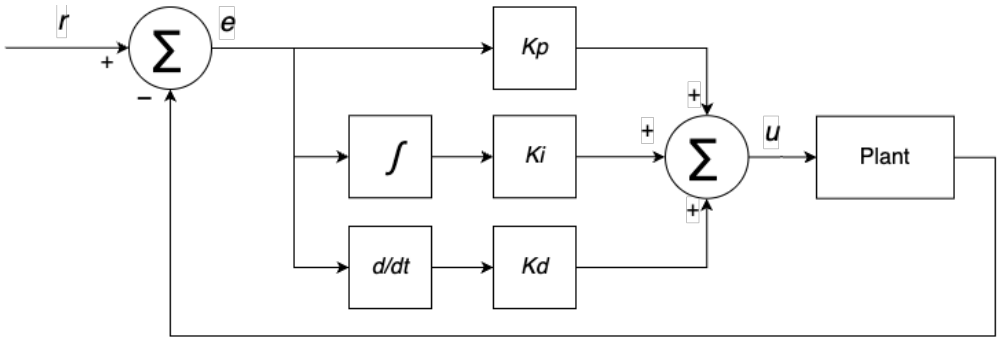


Figure 2 - Diagram of a PID controller

The three coefficients serve different purposes in the control of the system, and not all of them are always necessary. Increasing *Kp* will increase the speed of the response, but large values may introduce unwanted oscillations. The integral term sums errors over time and so removes any steady state error from the system. However, the rapid buildup of integral control can cause significant overshoot, leading to oscillation and instability when high gain values are used [5]. The derivative term compares error values over time and is useful in reducing overshoot and oscillations. However, it can significantly slow down response times if *Kd* is tuned too high. Table 1 shows the varying simplified effects of each parameter when they are increased, with the inverse being true for when they are decreased.

Table 1 - Effects of increasing *K* values [5]

Parameter Increased	Rise Time	Overshoot	Settling Time	Steady-State Error	Stability

Kp	Decrease	Increase	Small Change	Decrease	Degrade
Ki	Decrease	Increase	Increase	Decrease Significantly	Degrade
Kd	Minor Decrease	Minor Decrease	Minor Decrease	No Effect	Improve (for small *Kd*)

A notable example of PID control applied to a BBR is given by Wyrwał and Lindner [2] who designed a cascading system to drive their robot. They split the 3D problem into two planes and then used two PID regulators per plane. The first regulating loop was responsible for calculating the angle from vertical at which the robot would need to tilt to achieve a desired movement speed given a direction. While the secondary loop calculates motor speeds that would be needed to reach that previously calculated angle. The four total regulators were tuned manually and values found to make the robot functional.

3.1.2 LQR Control

LQR (Linear-Quadratic Regulator) is a popular control method for systems represented in state-space form. It works by minimising a cost function which is made up of weighted error values from the systems various states. State-space representation is a mathematical model used to describe physical systems with a set of first order differential equations. Instead of just focusing on inputs and outputs, it tracks a set of minimal state variables, often including position, velocity, temperature, etc, which are combined into a state vector *x*. A system in state space representation takes the form:

$$\begin{aligned} \dot{x} &= Ax + Bu \\ y &= Cx + Du \end{aligned} \tag{1}$$

Where x is the state vector, u is the control input, y is the system output and A, B, C, D are matrices describing the system and the relationships between states and outputs. The purpose of an LQR controller is to find the optimal control input that drives the system to a reference state, while minimising a quadratic cost function, defined below as:

$$J = \int_0^\infty (x^T Q x + u^T R u) dt \tag{2}$$

Where Q and R are matrices of weights for the states and inputs respectively, and J is the value that should be minimised. It can then be shown [6] that with a minimal value of J, a state gain matrix K, can be computed:

$$K = R^{-1} B^T P \tag{3}$$

Where P is found by solving

$$0 = A^T P + PA - PBR^{-1}B^T P + Q \tag{4}$$

K can then be used in the control loop to calculate what the input, u should be, given any state, x of the system:

$$u = -Kx \tag{5}$$

An example of a BBR using LQR control is Koos Van Der Blonk (2014) [7]. His work demonstrated that this form of control can be highly effective when implemented well, offering smooth and responsive balance performance. However, he commented that LQR controllers can be more difficult to tune because the parameters are often not related to the requirement characteristics, making finding optimal values to fit specifications more challenging.

3.1.3 Reinforcement Learning

A less common approach is to use reinforcement learning as a control method. It works by laying out a set of nodes that map an input vector to an output vector in layers as shown in Figure 3. Each node has a corresponding bias, and each line has a weight. The value of each node is calculated by summing the values of all other nodes that point towards it, each multiplied by the weight corresponding to the line. The bias is then added, to which point that node can now be used to calculate following nodes.

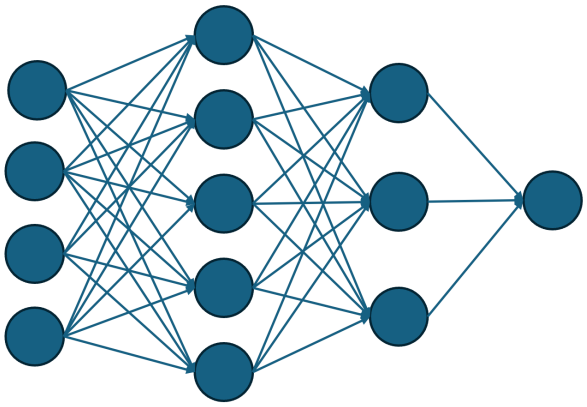


Figure 3 - An example neural network setup

The idea is that you feed the input state into the network and get a numerical response out of the other end. To begin with, the nodes are initialised with random values so the output will be meaningless but there are ways to make it "learn". If every time you test the network, you score its performance by some number of metrics, it can slowly change its bias and weight values to optimise the reward function that is defined. There are many different algorithms for making neural networks learn – a popular one being PPO (Proximal Policy Optimisation) which has strict rules regarding how much the internal weights can change each iteration.

Reinforcement learning can be a very powerful technique in systems that are difficult to model because it allows the agent to learn by interacting with the model itself. However it often takes a long time to train the network – and finding a reward function which adequately encourages learning can be challenging. It is also sometimes impractical to allow an agent control over hardware in order to learn. There are limited examples of RL being used for BBRs, however Yifan Zhou et al (2022) [8] managed to create a compound controller combining conventional feedback with deep learning to create a system that worked noticeably better than the feedback control alone.

3.1.4 Analysis

All control methods are viable for the context of BBRs, with the main limitations being complexity of implementation and tuning. LQR shows promise as a model-based solution, offering optimal control with mathematically guaranteed performance, provided an accurate system model is available. It can outperform PID in terms of stability and response time but requires additional linearisation steps and a full-state representation, which may not be easy to obtain.

In contrast, PID is much simpler to implement and tune, making it highly attractive for prototypes and systems with limited computational resources. However, the performance can degrade under changing conditions due to the non-linear nature of the system.

RL introduces the potential for adaptive and highly optimised control through trial-and-error learning. It excels in performing well in systems that are non-linear as it does not rely on any underlying dependency on a model. Unfortunately, RL comes with difficult and long training times, higher computational requirements, and less predictability due to their black-box network nature.

Ultimately, the choice of controller is reliant on the hardware available, development time and desired robustness. In future developments, hybrid models of the different types are bound to give increased results.

3.2 Motor choice

An important hardware decision when designing a BBR is which motor type to use. The main contenders are stepper motors, brushed DC motors and brushless DC motors. This section will discuss the differences and outline what previous projects have used.

3.2.1 Stepper Motors

Stepper motors are discrete electric motors that rotate through a number of fixed “steps” instead of freely rotating. They function by having a toothed central rotor that is surrounded with a number of stator coils as shown in Figure 4.

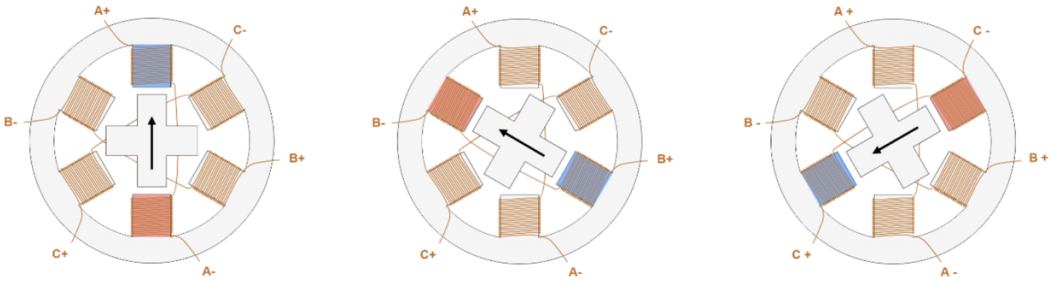


Figure 4 – Step routine of stepper motor [9]

When a specific coil is energised, it creates a magnetic field which the teeth on the rotor (made of either a permanent magnet or a variable reluctance iron core [9]) will align themselves with. By pulsing the coils in order around the circle, the rotor can be made to shift in a seemingly continuous manor. The stepped nature of these motors makes them ideal for precise position control, and the frequency at which they are pulsed allows for control over rotational velocity too. Because the mechanism is fixed into a number of steps (usually 200, with each increment being a rotation of 1.8° [10]), stepper motors can be operated with open loop because it can be assumed that the motor has turned exactly the amount of steps you have told it to. Because of the coils

holding the teeth in place, stepper motors have high holding torque at low speeds which can be useful to assume there is little slippage occurring. This backfires however if the load torque is too high, which can result in the motor skipping a step which negatively impacts the control [9]. At higher speeds steppers can become noisy with vibrations and signals and interfere with delicate electronics around them, and they also have a lower torque-to-inertia ratio compared to other motor types [9], which means in general they are slower to respond and worse in dynamic control applications – being prone to overshooting and oscillations. Another major drawback with steppers is that because they are position controlled, it is not easy to implement torque-control methodologies which can limit what control theories can be applied to the system later. It is possible, however it requires two additional controllers for current and magnetic flux, as well as encoders for closed loop feedback and some more complex runtime computations [11].

Stepper motors can also be micro stepped which is the process of energising multiple consecutive coils at once such that the teeth will align between them. By changing the patterns of coil energies, it is possible to get more than the 200 full steps. This allows for finer position control if used in that way, but more importantly it creates a smoother rotation when the motor is spinning at higher speeds reducing noise in the system. Micro-stepping does come with the downside in that it reduces the torque output of the motor in an exponential manner. William Miles et al (2017) [12] investigates the trade-offs between torque and noise, particularly in the context of BBRs – finalising that stepping at an $1/8^{th}$ of a step each pulse is the ideal equilibrium providing the motors are powerful enough to handle the torque downgrade.

Despite some drawbacks, there are a few previous examples of BBRs that have used stepper motors. Notably Kumagai and Ochiai [3] realised a successful robot using stepper motors, interestingly using acceleration as the control variable. Conversely, Wyrwał and Lindner [2] also used steppers in their implementation but commented on the vibrations having a negative effect on their other sensor readings giving them difficulty in testing stages.

3.2.2 DC Motors

DC motors are continuous actuators, split into two types – brushed and brushless. They both turn due to the interaction between permanent magnets and electromagnetic coils; with the main difference being where each are positioned as illustrated in Figure 5.

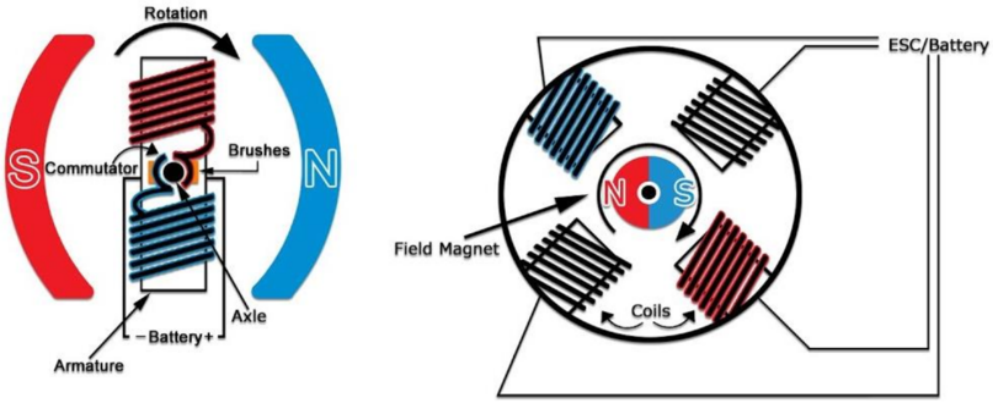


Figure 5 - Diagrams of DC motor types [13]

Brushed motors have coils on the rotor and permanent magnets as stators. The brushes are required to get electrical power to the spinning coils. It utilises a commutator which is a cylinder with alternatingly connected metal pads aligned with the magnets. They allow the stationary brushes to alternate the direction of power (positive or negative) so the motor will always spin the correct direction as instructed. Brushed DC motors are very simple and comparatively inexpensive, however the brushes can wear out over time. The control is also simple, with speed being proportional to the voltage supplied. This is usually regulated with a PWM signal to emulate changes in voltage.

Brushless DC motors have the electromagnetic coils on the stators and the permanent magnets on the rotor. The lack of spinning electrical parts means there are no moving contacts and nothing to wear over time. However, it is now required to control multiple coils with knowledge of precise spacings and timings to function as desired. This is usually handled by an external speed controller. Brushless motors are usually preferred for precision and reliability however they are much more expensive. All DC motors require feedback to be operated correctly, usually done with hall effect sensors or other position encoders.

Majority of the previous high functioning BBRs have used either brushed or brushless DC motors in their solutions. DC motors have the advantage of being able to be torque driven as torque is proportional to current. Jesperson (2019) [14], comments that brushless motors are the superior choice due to the high torque capabilities when gearing is used, as well as their continuous turning nature.

3.2.3 Comparison

Table 2 details the major differences between the three types of motors discussed, as well as how characteristics relevant to the context of BBRs.

FEATURE	STEPPER MOTOR	BRUSHED DC MOTOR	BRUSHLESS DC MOTOR (BLDC)
TORQUE AT LOW SPEED	High – good holding torque	Moderate	Moderate to high (varies by design)
TORQUE AT HIGH SPEED	Drops significantly	Moderate	Maintains torque well at high speeds
CONTROL CHARACTERISTICS	Open-loop position control is simple and precise at low speeds. Lacks direct torque control and suffers missed steps without feedback	Simple speed control via voltage or PWM. Torque is proportional to current enabling straightforward torque control. Positioning requires closed-loop control	Requires more complex control. Supports precise closed-loop velocity and torque control with proper feedback
EFFICIENCY	Low (draws current even when idle)	Moderate	High (less heat and friction)
SIGNAL INTERFERENCE	Moderate	High	Low to medium

RELIABILITY & MAINTENANCE	High (no brushes), but can overheat	Lower (brush wear requires maintenance)	High (no mechanical wear parts)
COMPLEXITY OF INTEGRATION	Medium: needs stepper driver and tuning	Low: simple motor driver	High: requires ESC and possibly sensors
COST	Low to medium	Low	Medium to high
STABILITY IN DYNAMIC SYSTEMS	Poor in open loop, may lose steps without feedback	Good with PID + encoder	Excellent with tuned control systems
RESPONSIVENESS (AGILITY)	Limited – slower acceleration	Fast response	Very responsive, ideal for fast control
POWER DENSITY	Low	Moderate	High
PREVIOUS BALLBOT USE	Rare – primarily in simple balancing demos or early prototypes	Common in research prototypes due to simplicity	Common in high performance designs

Table 2 - Comparison chart of different motor types

4 Hardware Overview

This section will describe the state of the system as it was provided at the beginning of the project, alongside some initial decisions that were made. However, any major details regarding troubleshooting, fixing and methodologies are detailed in section 6.

Figure 6 - Photograph of inherited robot

4.1 Microcontroller

The microcontroller included with the given hardware was an Arduino Uno – chosen for ease of use and reduced learning curve by the previous user [15]. While initially the Arduino seemed to be

fine in keeping up with requirements, it introduced several limitations including loop speed and program memory into the project that had to be addressed.

4.2 Drive System

The inherited hardware came with three NEMA 23 stepper motors, accompanied each with a TB6600 microstep driver [15]. The drivers take care of coil energising sequences and can even apply varying levels of micro stepping according to configuration switches on the side. A direction command is given to the driver and then it will step the attached motor once per rising edge of signal that is inputted to the PUL+ pin.

Despite some of the documented limitations of using stepper motors for the application of BBRs, including reduced torque control, signal disturbance and stability issues; the decision was made to keep the existing drive system configuration from the inherited hardware. This choice was influenced by both practical and strategic considerations. Replacing the motors would have required significant time investment in selecting new components, redesigning power electronics and modifying the robot almost in its entirety for new mounting points etc. The control theory accompanied with DC motors, while more robust in most cases, is also more complex. Additionally, the quality of motors desired would have far exceeded the budget of this project. Instead, the time was invested into other aspects of the control system development which had greater personal and project relevance. Furthermore, the challenge of working with steppers in this unconventional context presented an opportunity for additional learning and problem solving which greatly impacted the final decision. While this decision did impose certain constraints on control analysis further into the project, it provided a consistent hardware platform that supported experimentation and development in other areas.

4.3 Sensors

The system included two Sparkfun ICM-20948 9DOF IMUs. Each board housed an accelerometer, a gyroscope and a magnetometer for various orientation and movement-based measurements. The boards communicate to the host controller via I^2C, an address and bus-based data system. The boards had two configurable I^2C addresses allowing for both of them to be connected to the same bus to interface with the microcontroller.

Accelerometers function by measuring the force on a known internal mass to calculate the linear acceleration in three axes. The gyroscopes function with small vibrating core. Rotations in the environment have an effect on the vibration patterns via the Coriolis effect. And the magnetometer has a small magnetic core with copper coils wrapped around. Changes in the magnetic field around it induce currents in the coils that can be measured. Unfortunately, they are very sensitive to all magnetic fields, not just the Earth's and so they often require extensive calibration to be useable.

The IMU boards presented significant data reliability issues at the beginning of the project and were the main cause of delay of any major progress throughout the timespan of the project. With the setup as is and the codebase provided, the IMUs were incapable of outputting reliable data to the microcontroller, and the planned interpretation of the data did not function how the previous user planned. This made it infeasible to do any whole system testing on the hardware. Despite

this, it was decided to keep the sensors, as the issues appeared to come from configuration or implementation errors rather than hardware faults.

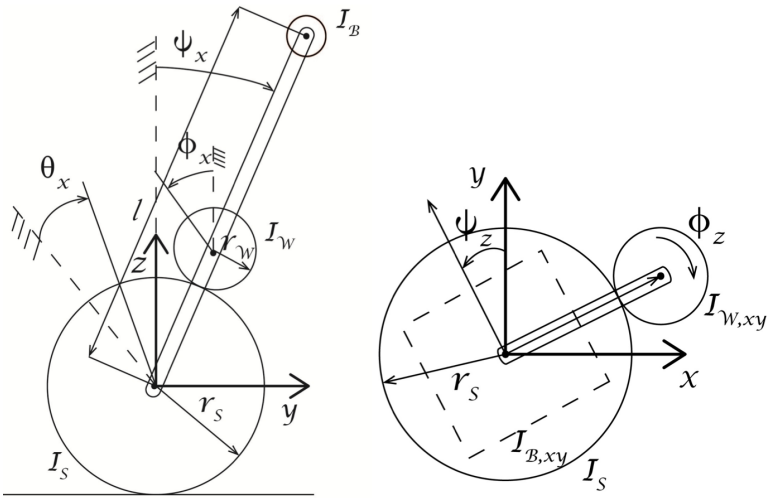
5 Modelling and Simulation

At an early stage, a simulation environment was developed to explore potential control strategies that could be applied to the system. The goal was to get a better understanding of dynamic behaviour and visualise results from varying control theories before the hardware was in a functional state and ready for real tests. It also acted as a way to continue work on the project over the holiday period where access to the robot was impossible.

5.1 2D Modelling

To model the dynamics of the robot, the method of Van der Blonk (2014) [7] was used as reference to understand and follow the underlying mathematics. The aim was to calculate the dynamic equation that describes the system to use in the simulation.

The system was broken down into the three orthogonal 2D planes for ease of dynamics, as shown in Figure 7. The xz-plane and yz-plane are identical so only one is shown.



a) Model for the yz-plane. b) Model for the xy-plane.

Figure 7 – Sketches of 2D models [7]

The system can be fully defined at any point in time by just the rotation of the ball and the rotation of the body from vertical. Therefore, a set of minimal coordinates of the system can be:

$$q_{yz} = \begin{bmatrix} \theta_x \\ \psi_x \end{bmatrix}, q_{xz} = \begin{bmatrix} \theta_y \\ \psi_y \end{bmatrix}, q_{xy} = [\psi_z] \tag{6}$$

The simulation only modelled the system in the yz-plane, a side view of the robot, so that's what this section will continue to describe.

The full derivation of the dynamic equations using the Euler–Lagrangian method was followed from Van der Blonk with no major alterations other than substituting dimensions of this project's robot instead. Assumptions included no-slip contact between the ball and ground, rigid body links,

and independent planar motion. For completeness, the approach involves forming a Lagrangian for the system:

$$L(q, \dot{q}) = T - V \tag{7}$$

Where T is the total kinetic energy and V is the total potential energy of all bodies involved. The equations of motion can then be extracted from the Euler-Lagrangian equation:

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{q}_i}\right) - \frac{\partial L}{\partial q_i} = \tau_{ext,i} \tag{8}$$

With $i = 1, \dots, n$ where n is the length of vector q, and τ_{ext} being the external torques on the system (the input from the motors). The final equations of motion obtained from the referenced derivation take the form of Equation (9). The full expanded form is shown in Appendix A.

$$M(q_{yz})\ddot{q}_{yz} + C(q_{yz}, \dot{q}_{yz})\dot{q}_{yz} + G(q_{yz}) = \tau_{ext} \tag{9}$$

5.2 Simulation Creation

Using the dynamic model, a simulation of the dynamics was created in Python utilising NumPy for matrix calculations and Matplotlib for visualisation. Equation (9) was rearranged to give Equation (10):

$$\ddot{q} = M^{-1}(\tau_{ext} - C\dot{q} - G) \tag{10}$$

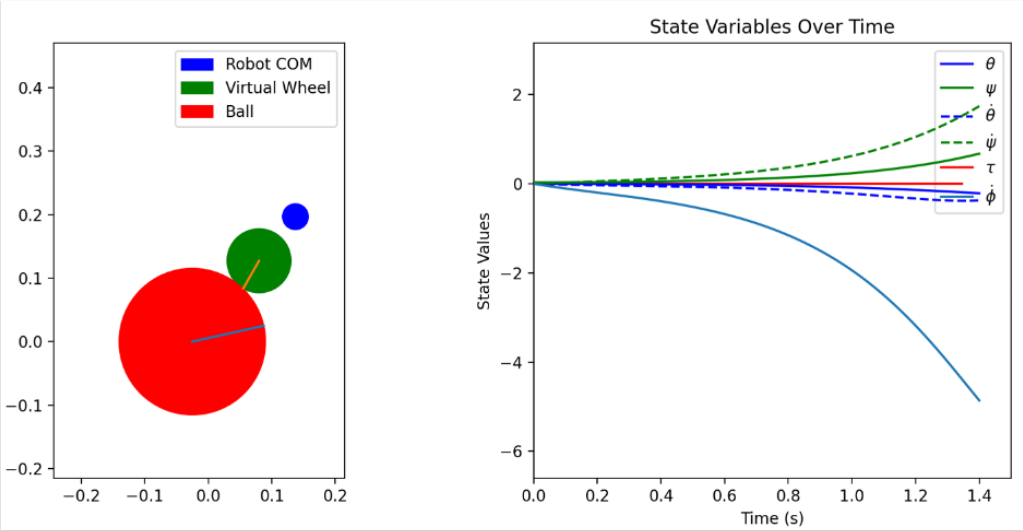
A dynamics function was written that takes in the state vector, q and $\dot{q}$, and calculates $\ddot{q}$ according to Equation (10). It first computes the matrices M, C and G as they are in turn expressions of q and $\dot{q}$. For initial testing, τ_{ext} was set to 0 which would be equivalent to the motors being unpowered and spinning freely, however the function was given the functionality to query any provided control function with the current states and receive back a value for τ_{ext} The full function can be found in Appendix B.

From SciPy, the function `solve_ivp` was used to compute the behaviour of the system according to the defined dynamics function over a period of time. IVP stands for Initial Value Problem, and the `solve_ivp` function "numerically integrates a system of ordinary differential equations given an initial value" [16]. It takes in the defined dynamics function as well as a state vector of initial conditions. At each time step of the model, the function records the current state of the system to memory. This data is then plotted on a graph using Matplotlib to show how the variables change over time.

The Matplotlib `funcAnimation` method was used to update frames in the graph, as well as create a visualisation made of plotted circles to represent bodies within the system. This was possible as the x and y coordinates of each body are able to be calculated from the minimal state vector at each time step. This created a great looking visualisation of the model and allowed the robot to be seen balancing (or falling off) the ball in real time. This provided instant feedback when running the simulation as to how it was reacting, instead of relying on just graphical data.

A snapshot of an example output is shown in Figure 8 where the system was let run for 1.5 seconds with the initial condition of the robot starting at a slight angle of 2° from vertical and the

motors remaining unpowered for the time being. The parameters remain consistent with the diagram in Figure 7.



a) Real time visualisation of 2D system

b) Graph output showing states over time

Figure 8 – Matplotlib output of Python Simulation

5.3 Simulation applications

Once the simulation appeared seemingly accurate to the expected dynamic behaviour, it was then used to learn about responses and varying control theories. Every call of the dynamics method would now also include a variable τ_{ext} that is calculated from a control method, usually wrapped up into a class.

5.3.1 PID control

As a first test, a basic PD controller was fitted to the simulation. A short time was spent tweaking values of Kp and Kd until the simulation successfully balanced itself for an extended period of time, given small random non-zero initial conditions. The simulation output can be seen in Figure 9. The control class code can be found in Appendix C.1

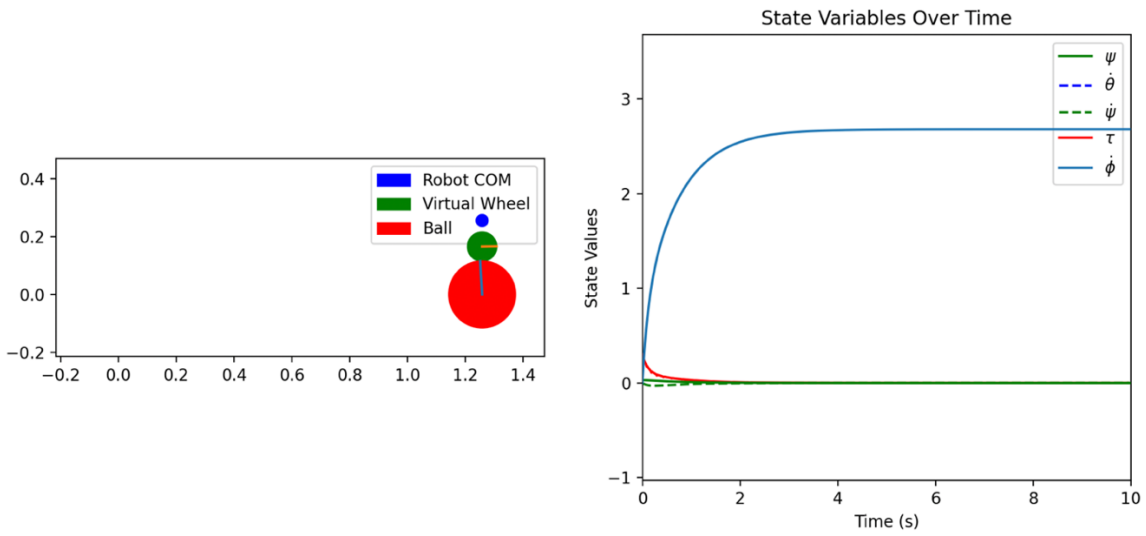


Figure 9 - Simulation response to simple PD control

In this state, the system was able to balance itself and remain on top of the ball very easily. However, the steady state of the system involved a constant rolling to one side which is not the desired outcome. This is because the PD controller was only regulating ψ (the angle from vertical) to be 0, and not the other states. This could have been fixed by also regulating $\dot{\theta}$ (the angular velocity of the ball) to be 0.

Alternatively, an additional damping matrix, D was added to the original dynamic equation to try and model some friction in the system. While it was impossible to model the friction values accurately, having a small damping force which opposed motion helped significantly in the stability of the system. It would now slowly reach a stationary point rather than continuous rolling like before. The results are shown below in Figure 10 where the exact same starting conditions and control coefficients were used as above in Figure 9.

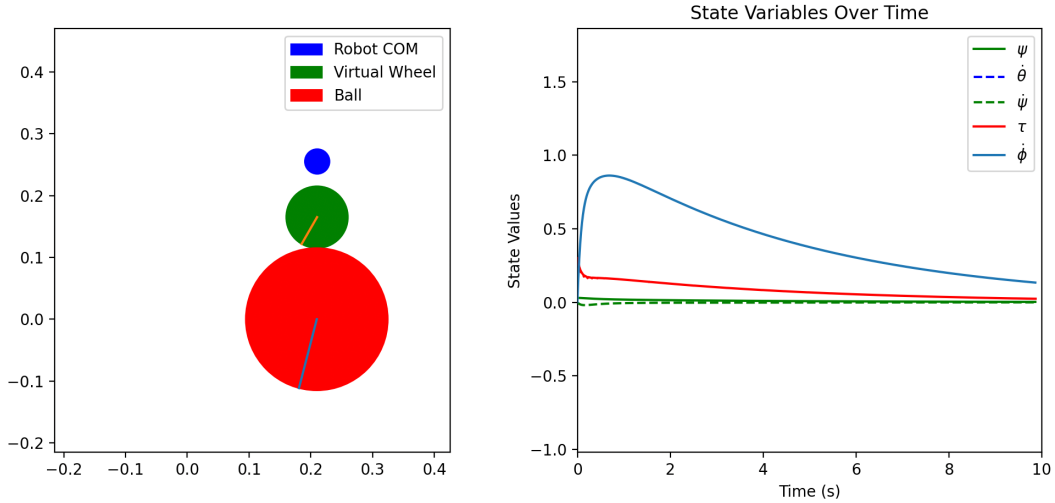


Figure 10 - Simulation response to added damping matrix

It is worth noting that this control approach was taking advantage of directly setting torque values for the virtual wheel, something which is not possible with the stepper motors that are on the real robot. This was noted but ignored at this stage as it was mostly a proof of concept.

5.3.2 LQR Control

An attempt at using LQR control on the simulation was made next. Before the simulation is run, the LQR class first calculates a linearised approximation of the dynamic equations to put into state space form. The A and B matrices are made from computing the Jacobians of the nonlinear dynamics, with respect to the state variables and control input, and then evaluated at some equivalent point (an upright and stationary position in this case). The Jacobians model the system's local behaviour around that point and so can be used to form the linearised state-space model that is required for LQR design. Arbitrary Q and R matrices were defined, prioritising the angular position of the robot and the velocity of the ball over the velocity of the robot and position of the ball. The K matrix was then computed using the `lqr` method from the Control Python library. This was then used within the dynamics function to utilise the LQR control. The full class code is shown in Appendix C.2 and the output of the simulation is shown in Figure 11.

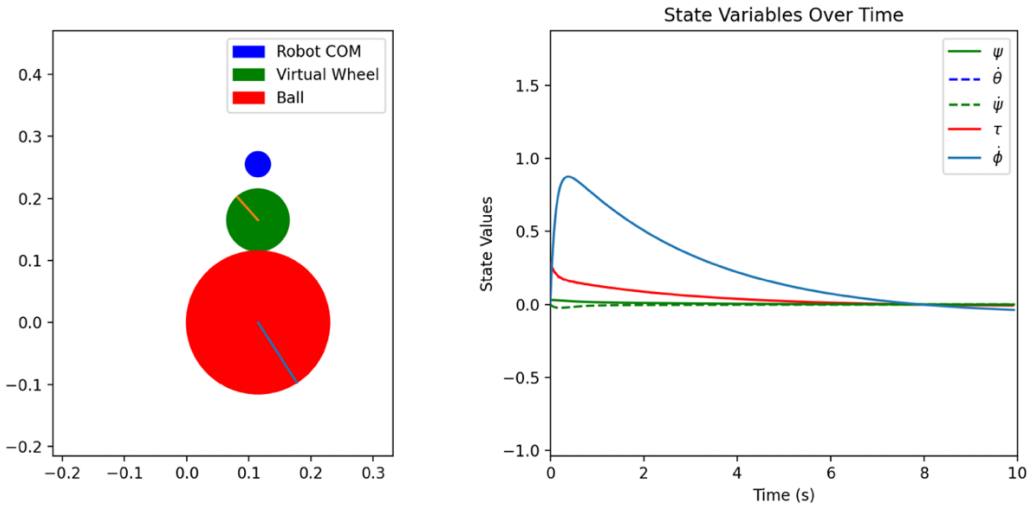


Figure 11 - Simulation response to LQR control

Within this approach, the torque input is being approximated as per Equation (11):

$$\tau_{ext} = I_W \times \ddot{\phi} \tag{11}$$

This is not necessarily an accurate assumption as it misses some key reaction torques and assumes there is perfect control over acceleration within the system. Although with stepper motors, this isn't too infeasible and hence it was deemed a suitable substitution in this example. The response is actually very similar to that of the PD control, stabilising very well, even before any proper tuning had taken place.

5.3.3 Reinforcement Learning

A reinforcement learning approach was also attempted to work with the simulation. This method started with creating a training environment for the system using the gymnasium python library. The environment provides the interface for learning setup such as step, reset functions. This also includes reward calculation and all other interactions between the training model and the simulation. A basic reward function was defined as shown in Figure 12 which encourages the system to stay upright and penalises it heavily for falling too far. It also includes other smaller penalties for moving too much and performing large actions to try and incentivise it to stay in one place and not output too unrealistic torque commands.

```
def compute_reward(self):
    theta, theta_dot, psi, psi_dot = self.state
    # Bonus for staying near upright
    upright_bonus = 10.0 if abs(psi) < 0.1 else 0.0

    # Quadratic penalty for tilting
    tilt_penalty = -10 * (psi ** 2)

    # Large penalty for falling over
    fall_penalty = -500 if abs(psi) > np.pi / 2 else 0.0

    velocity_penalty = -0.1 * (abs(theta_dot) + abs(psi_dot))

    # Small reward to incentivise survival
    survival_reward = 1.0

    return upright_bonus + tilt_penalty + fall_penalty + velocity_penalty + survival_reward

```

Figure 12 - RL reward function

This environment is then passed to a PPO algorithm (Proximal Policy Optimisation) from the Stable Baselines 3 library. The library includes useful functionality such as saving/loading models and training monitoring.

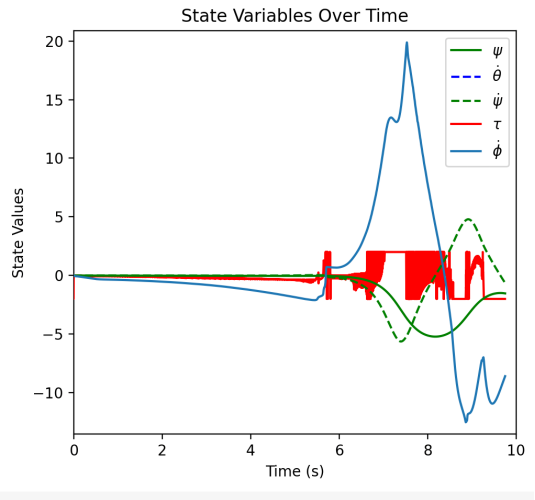


Figure 13 - Response graph of RL method

The model worked surprisingly well for the first 5 seconds, staying balanced and only slightly rolling to one side. But then it jolts and falls over with some strange torque commands. It is unknown why the torque output behaved like it did, but most likely the fault of the mapping between model output and torque input.

5.4 Simulation conclusions

The simulation gave some useful insights into the dynamics of the system, as well as being used as a tool to learn more about each of the control approaches by implementing them in code and also observing their responses to the system. While the simulation was not ultimately useful to the final goal, the development helped greatly with understanding the theory behind every aspect of the project.

6 Hardware Methodologies

This section will provide a detailed walkthrough of the methodologies involved with the actual hardware and discuss the challenges that were encountered when attempting to make the robot function in its entirety. It will discuss the limitations identified and the various solutions implemented to enhance the robot's functionality.

6.1 Microcontroller

6.1.1 Original Arduino Uno

As mentioned, the inherited robot came equipped with an Arduino Uno which, from initial testing, seemed to function perfectly fine for most areas of the project. It could run separate programs for each subsystem fine (the following detailed sections) and appeared to be enough. However, when initial integration tests were run, it was discovered that the loop time of the combined program

was far too long, and it was observed that the robot could not react quick enough to changes in orientation.

At this point in time, the program was using a modified version of an Arduino library: AccelStepper, to control the stepper motors. It worked by assigning motors to pins and taking in speed commands in steps per second. The library does not work in the background and instead relies on constantly calling a `runspeed` function as often as possible in the main loop. This would then poll the motors and step them if they were due a step according to the speed command calculated just before. While simple to implement and very fluid in changing speeds/accelerations in real time, it was heavily dependent on how frequently the `runspeed` function could be called. It is required that it is called at least as frequent as the maximum speed the motor would need to turn – ideally much faster than this, otherwise stepping becomes irregular and jittery.

The Arduino also had to handle all IMU data processing at this time, including using a Madgwick filter to combine inputs from two IMUs and calculate Euler orientation angles from the raw sensor data (more on this in Section 6.3.1) which happened to be more computationally expensive than hoped. When this part of the program was implemented together with the motor control, it was found that the loop time was significantly slower than the speeds the motors needed to be polled at, and so the system was unable to function in any meaningful way using this setup.

An alternative to using the polling approach implemented by the AccelStepper library is to use the Arduino's built in PWM control to generate step pulses for the drivers. This works in the background rather than being reliant on the main loop and so would eliminate the issue described above. It is available on 6 set pins, although only three timers are available – the exact number needed for this project. The outputs are usually controlled by using the `anaglogWrite` function however, this can only change the width of the pulses generated, not the frequency. The frequency can only be altered by modifying low level registers, specifically changing the pre-scaler values of the timers. The PWM outputs function by counting up to 256, N times, where N is the pre-scaler number set in the register. It increments the count once every clock cycle, and with the Arduino Uno's having a clock speed of 16MHz, this means the possible PWM outputs are derived from Equation (12):

$$f_{PWM} = \frac{16\text{MHz}}{N \times 256} \tag{12}$$

Or making N the subject:

$$N = \frac{16\text{MHz}}{f_{PWM} \times 256} \tag{13}$$

However, the ATmega328P hardware can only facilitate a pre-defined set of pre-scalers, namely [1, 8, 64, 256, 1024] on timers 0 and 1, with timer 2 having these and two additional values available of [32, 128]. This means there are a very limited number of frequencies available to use, being nearly impossible to get the exact frequency that is required, instead having to find the closest available one – which in a lot of cases is most likely too distant. Therefore, this option was disregarded quite quickly.

As well as loop time, there were some concerns over program memory capacity limitations imposed by the Arduino Uno. While in most cases the relatively small memory is perfectly fine, this project found some niche use cases that would require more than the 32KB available. The details of these scenarios are discussed in Section 6.3.

6.1.2 New Microcontroller

With the limitations discussed above, it was decided to upgrade the microcontroller in the system. The research of the previous student [15] was used to decide on purchasing an ESP32 microcontroller. The main features that influenced this decision was the 240MHz clock speed (15x as fast as the Arduino) and the increased useable program memory size (1.3MB up from 32KB of the Uno). It was thought that an improvement in both of these metrics would be enough to suitably control the system. The ability to program the ESP32 in the same language and structure as an Arduino based microcontroller also made transitioning between boards easier than expected.

Alongside purchasing an ESP32, three AD9833 boards were ordered. These are small programmable waveform generators which are communicated with via SPI (Serial Peripheral Interface) connection. They are capable of producing different shapes of wave (square, triangle and sinusoidal) across a large range of frequencies (0.1Hz to 12.5MHz) with a resolution of 0.1Hz across the entire range. They are controlled by sending 16-bit command words over the serial connection to specify shape and frequency. This allowed the microcontroller to calculate the required frequencies but then send those to the external boards to handle generation instead of using the main program loop. The boards were commanded to create square waves at varying frequencies and the signal outputs were observed on an oscilloscope. The boards were capable of generating the signals at accurate frequencies and were responsive to frequency change requests. However, when they were wired into the robot, they did not function with the stepper drivers. It is believed this is because the AD9833 boards output continuous analogue signals, and the TB6600 drivers expect cleaner faster edged digital signals. A curve and rise time were observed in the signals from the boards.

It is likely that the AD9833 could have been made to function in the system correctly with some additional external electronics attached to their outputs. For example, signal conditioning components such as level shifters and Schmitt triggers could have been used to shape the waveforms into acceptable signals for the motor drivers. However, this would greatly increase system complexity making it a less favourable solution to the approach that was ultimately adopted.

After having trouble with the signal generators, it was discovered that the PWM control on the ESP32 was far superior to that of the Arduino. It has 16 independent channels (as opposed to the three on the Uno), which can be mapped to almost any available GPIO pin, instead of the pre-defined pins on the Arduino. Additionally, there is no major limitations on the pre-scaler values, so it can output far more accurate frequencies without being constrained by defined values. The ESP32 also comes with a set of `ledc` functions as part of the ESP-SDK. This provides an easy way to interface with the PWM channels without having to manage the low-level registers. Figure 14 shows how a single PWM channel could be setup and then used. The setup includes choosing some parameters which ultimately are largely inconsequential for this project's use case. The duty

cycle (width of each pulse) does not matter for this application as the stepper drivers only rely on pulse edges – and so it was arbitrarily set at 50%. And the pulse width resolution allows higher degrees of accuracy when choosing the width of pulses – which in turn has no great effect on the system.

```
1. // Setup
2. ledcSetup(0, 500, 8);  // Start channel 0 with frequency 500Hz and 8-bit pulse width resolution
3. ledcAttachPin(18, 0);  // Attach PWM channel 0 to GPIO pin 18
4. ledcWrite(0, 127);   // Set 50% duty cycle (out of 2^n -1 = 255) on channel 0
5.
6. // Frequency Changes
7. ledcChangeFrequency(0, 1000, 8);    // Change the frequency of channel 0 to 1000Hz
8.
```

Figure 14 - Example PWM setup on ESP32

The `ledcChangeFrequency` function can be called as often as is needed to provide reactive speed responses to the motors – as long as the program does not try and change the frequency to the current outputting value. Overall, this solution provided precise, reliable and responsive signal generation which ultimately managed to successfully drive the stepper motors without any additional circuity and took pressure off of the main program loop speed.

During explorative testing using the ESP32, the V_N (power in) pin and the Ground pin were accidentally shorted which led to the microcontroller no longer accepting any new programs trying to be flashed onto it. This resulted in needing to purchase a second microcontroller. Fortunately, these boards were very inexpensive, costing £2.80 for the first one, and £10.33 for the second one. The second board was purchased with next day delivery and also came with a breakout board which made wiring and mounting a lot more accessible.

A new mounting bracket was designed in SolidWorks to house the microcontroller, with screw holes lined up with the ESP breakout board, and mounting holes measured to fit the existing holes on the robot from where the Arduino used to sit. A render of the bracket is shown in Figure 15.

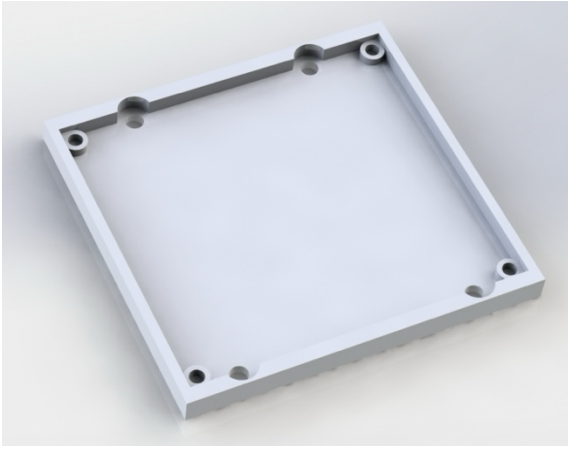


Figure 15 - 3D Render of microcontroller mounting bracket

6.2 Drive control

6.2.1 Kinematics

A major first milestone in the project involved understanding and tweaking the derivation set out by Soe Soe Htay et al. (2024) [17] for the kinematics of a three omni-wheeled mobile

robot. The kinematics of a system describe how changes in position/velocity of actuators will change the position/velocity of the end-effector – in this case the direction and rotation of drive. And conversely, the inverse-kinematics can be derived which calculates what motor speeds are required for a desired total movement effect. While the implementation from Soe Soe Htay described a system built for flat ground traversal, with wheels perpendicular to the ground – it was assumed that the kinematics would be similar enough for this project too. Testing discussed later in Section 7.1 proved this assumption to be correct.

The robot frame location in reference to the wheel positions was changed from the source derivation to better suit this projects hardware setup. This had a minor effect on the final outcome, requiring only small alterations in the process. The implemented frame positioning is shown in Figure 16 where $[u \quad v \quad r]$ are the velocities of the robot in its own coordinate system, while $[\dot{x} \quad \dot{y} \quad \dot{\theta}]$ are the velocities of the robot in a global reference frame. $[\omega_1 \quad \omega_2 \quad \omega_3]$ are the angular velocities of each wheel in rads^{-1}, with the nominal directions of the wheels indicated by red arrows.

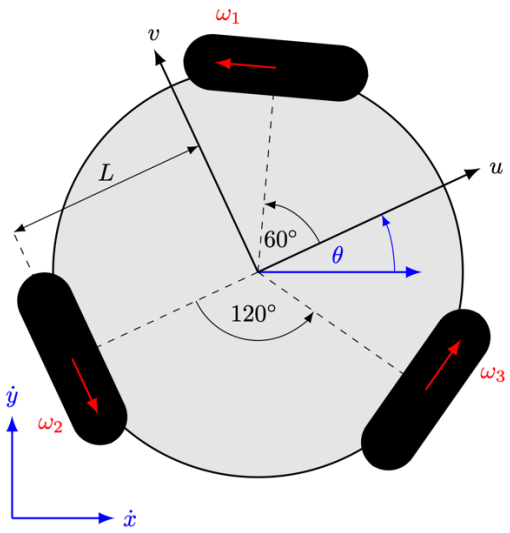


Figure 16 - Sketch of wheel configuration

Once completing the derivation, the inverse kinematic description of the system is shown in Equation (14) where a is the radius of the wheels and L is the distance of the wheels from the centre.

$$\begin{bmatrix}\omega_1\\ \omega_2\\ \omega_3\end{bmatrix} = \frac{1}{a}\begin{bmatrix}\frac{-\sqrt{3}}{2} & \frac{1}{2} & L\\ 0 & -1 & L\\ \frac{\sqrt{3}}{2} & \frac{1}{2} & L\end{bmatrix}\begin{bmatrix}u\\ v\\ r\end{bmatrix} \tag{14}$$

6.2.2 Frequency calculations

Now that the desired velocities of each wheel can be calculated, the required frequency needs to be sent to the stepper drivers. In theory the frequency is calculated using Equation (15).

$$f = \frac{|\omega|}{2\pi} \times (S \times M) \tag{15}$$

Where S is the number of steps per rotation of the motor (in the case of the NEMA 23, $S = 200$), and M is the factor of micro stepping used. However, in practice it was observed that the wheel rotational speeds were consistently slower by a factor of 4. The micro-stepping configuration was double-checked, as well as all calculations and frequency outputs but the source of the discrepancy was not found. So instead, Equation (16) was used to calculate frequencies. At this point, the robot was capable of moving in any commanded direction relative to its forward (u) axis, with any given velocity. It was tested on flat ground and also upside down with the ball on top to observe direction and speed.

$$f = \frac{|\omega|}{2\pi} \times (S \times M) \times 4 \tag{16}$$

Based on the findings from F. Sandahl et al (2017) [12], which evaluated the performance characteristics of stepper motors while varying the degree of micro-stepping – a factor of 8 was selected as the optimal level to ensure smooth rotation, without compromising the torque required for motion. Configuring this setup involved setting $M = 8$ in the program, as well as adjusting the TB6600 driver switches to operate in 1/8th stepping mode. The current limit settings on the drivers were initially configured to limit input to 1.5A rather than the maximum rating of 3.5A. This resulted in significantly reduced torque and produced unexpected performance issues such as skipping steps when very little load was applied. This caused considerable confusion during testing and delayed development for a few days until the cause was found.

6.3 IMU sensors

6.3.1 Arduino attempts

A major focus of attention within this project was attempting to retrieve useable and reliable data from the IMUs, as well as interpreting and using it in the most advantageous way. The initial setup consisted of two Sparkfun ICM-20948s, communicating over the same I^{2}C bus with the original Arduino Uno. The accompanying Arduino library [18] made getting initial raw values straightforward and these were fed into a Madgwick filter to compute Euler orientation angles (roll, pitch and yaw). The filter was implemented with another library [19] which takes in the raw data and computes a set of quaternions using the Madgwick algorithm from which the Euler angles are then calculated.

There was some trouble receiving data from the magnetometers as shown in Figure 17, but this was fixed by re-calling their setup functions after the rest of the sensor had been initialised.

```
11:09:10.692 -> Scaled. Acc (mg) [ -00021.48,  00007.32, -01019.53 ], Gyr (DPS) [ -00000.56, -00000.21,  00001.40 ], Mag (uT) [  00000.00,  00000.00,  00000.00 ], Tmp (C) [  00026.74 ]
11:09:10.758 -> Scaled. Acc (mg) [ -00020.02, -00000.98, -01017.09 ], Gyr (DPS) [  00000.27, -00001.71, -00001.31 ], Mag (uT) [  00000.00,  00000.00,  00000.00 ], Tmp (C) [  00026.30 ]
11:09:10.791 -> Scaled. Acc (mg) [ -00014.65,  00007.81, -01032.23 ], Gyr (DPS) [ -00002.15,  00003.88, -00002.70 ], Mag (uT) [  00000.00,  00000.00,  00000.00 ], Tmp (C) [  00026.50 ]
11:09:10.823 -> Scaled. Acc (mg) [ -00026.37,  00000.49, -01030.76 ], Gyr (DPS) [  00000.39, -00003.62, -00001.41 ], Mag (uT) [  00000.00,  00000.00,  00000.00 ], Tmp (C) [  00026.50 ]
11:09:10.855 -> Scaled. Acc (mg) [ -00018.07,  00000.00, -01006.35 ], Gyr (DPS) [ -00001.46,  00000.33,  00000.51 ], Mag (uT) [  00000.00,  00000.00,  00000.00 ], Tmp (C) [  00026.74 ]
11:09:10.923 -> Scaled. Acc (mg) [ -00021.48,  00007.81, -01022.95 ], Gyr (DPS) [ -00000.50,  00000.46, -00002.11 ], Mag (uT) [  00000.00,  00000.00,  00000.00 ], Tmp (C) [  00026.74 ]
11:09:10.956 -> Scaled. Acc (mg) [ -00012.70,  00007.32, -01031.25 ], Gyr (DPS) [ -00003.32,  00002.37,  00000.11 ], Mag (uT) [  00000.00,  00000.00,  00000.00 ], Tmp (C) [  00026.45 ]
```

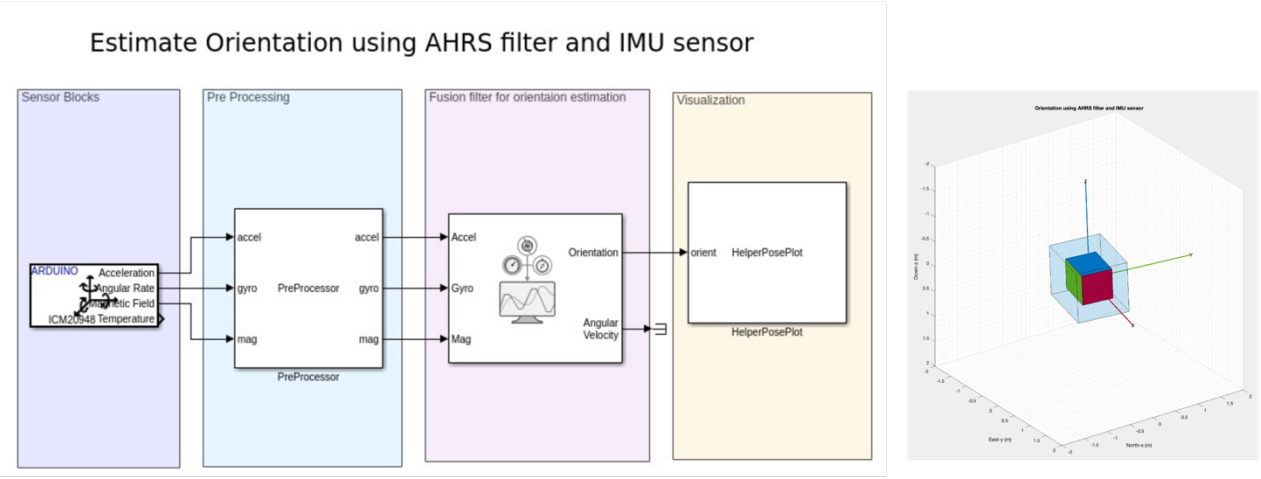
Figure 17 - Snapshot of incomplete magnetometer data

Once data was being received, it was studied and tested for accuracy and reliability. Unfortunately, the gyroscope values tended to drift very heavily, and the magnetometer values

seemed far off of reasonable numbers, leading to resultant Euler angles unstable and inaccurate. An attempt was made to calibrate the magnetometers to get more meaningful data out of them. Following the procedure set out by Nicolas Liaudat (2024) [20], large streams of raw magnetometer data was collected while the robot was moved around in every orientation possible with the motors switched on as they create magnetic fields strong enough to interfere. Then, using a Python script provided by Liaudat – this data, alongside an estimate of the Earth's magnetic field strength at this location, was used to calculate hard and soft iron correction matrices (A and B^{-1} respectively) to be applied to the raw data at runtime. The A matrix is subtracted from the raw values, and then the data is multiplied by B^{-1}.

$$A = \begin{bmatrix} 100.08 \\ 96.34 \\ 219.54 \end{bmatrix}$$
$$B^{-1} = \begin{bmatrix} 0.8444 & 0.0164 & 0.0110 \\ 0.0164 & 0.8186 & -0.0349 \\ 0.0110 & -0.0349 & 0.8331 \end{bmatrix} \quad (17)$$

While these corrections did help slightly, the resultant Euler angles from the filter still did not behave as precise as expected. It was at this point that alternative solutions were sought after. After having looked into using MATLAB and Simulink for control purposes, a Simulink package was found that had specific support for the ICM-20948 sensors and code generation for Arduino. Using an altered version of an example Simulink program, successful orientation outputs were constructed and observed as shown in Figure 18b.



a) Simulink program to compute orientation

b) Orientation visualisation output

Figure 18 - 3D orientation visualisation program and output

However, this program was compiled and run on an external laptop with sensor readings passing through the Arduino as the program file size was too large to be flashed into Arduino memory. An attempt was made to extract the pre-compiled code to observe the algorithms used, however this was unsuccessful as the Arduino-Simulink package does not save a copy of the generated C code after compiling. The program files of the installation of MATLAB being used were also corrupted at one stage which stopped the package from functioning correctly and hindered this line of development for an extended period of time.

6.3.2 Digital Motion Processing

With the new microcontroller some more possibilities opened up. An optional section of the Sparkfun library allows control and interfacing with the DMP (Digital Motion Processing). This is separate firmware that runs directly on the ICM-20948 which "offloads computation of motion processing algorithms from the host processor, improving system power performance" [18]. The DMP can handle background sensor calibration to keep the three sensors (accelerometer, gyroscope and magnetometer) maintaining optimal performance, and also compute sensor fusion to output quaternion data directly. It also allows for greater control over the individual sensors, for example allowing changes to be made to sampling frequencies. The firmware has to be flashed onto the host microcontroller to then be initialised on the ICM-20948 which takes up between 3-6KB. This additional memory constraint is what made this approach impossible while using the Arduino, however with the increased memory capacity of the ESP32, the DMP firmware could be included without issue, allowing full use of the sensor's onboard processing capabilities – and receiving orientation data directly in the form of quaternions.

Using this quaternion data, a vector $\vec{r}$ can be defined as upwards in reference to the robot's frame (perpendicular to the u and v axes shown in Figure 16.

Letting the quaternion $q = w + xi + yj + zk$

$$\vec{r} = \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} 2(xz + wy) \\ 2(yz - wx) \\ 1 - 2(x^2 + y^2) \end{bmatrix} \tag{18}$$

From this, the angle of which the robot is tilted from vertical can be calculated by taking the arccosine of the c component

$$\psi = \operatorname{acos}\left(1 - 2(x^2 + y^2)\right) \tag{19}$$

Similarly, the heading that the robot is tilted towards can be calculated from the a and b components

$$\alpha = \operatorname{atan2}(2(yz - wx), \quad 2(xz + wy)\ ) \tag{20}$$

6.3.3 Drawbacks

Some minor drawbacks were discovered with this approach including the incapability of using multiple sensors at once. While the main board's I^2C address could be selected between two options, the magnetometers on each board have fixed internal addresses which caused conflicts when initialising DMP on the second board. However, with the new DMP firmware, the output generated from just one sensor was deemed reliable enough to continue development. Another issue arose that at the beginning of the program, the DMP output would drift quite substantially for approximately 20 seconds before drifting back down to a stable origin level. This was likely due to the internal calibration that the DMP runs on the sensors and was accounted for by including time at the beginning of the program to allow for this before motors were initialised and the whole system started properly.

6.4 Final implementations

The final iteration of the electronic wiring with the new microcontroller and the removal of one IMU board can be found in Appendix D.

With each subsystem finally functioning correctly, it was possible to reconstruct a main program loop for the robot as shown in Figure 19.

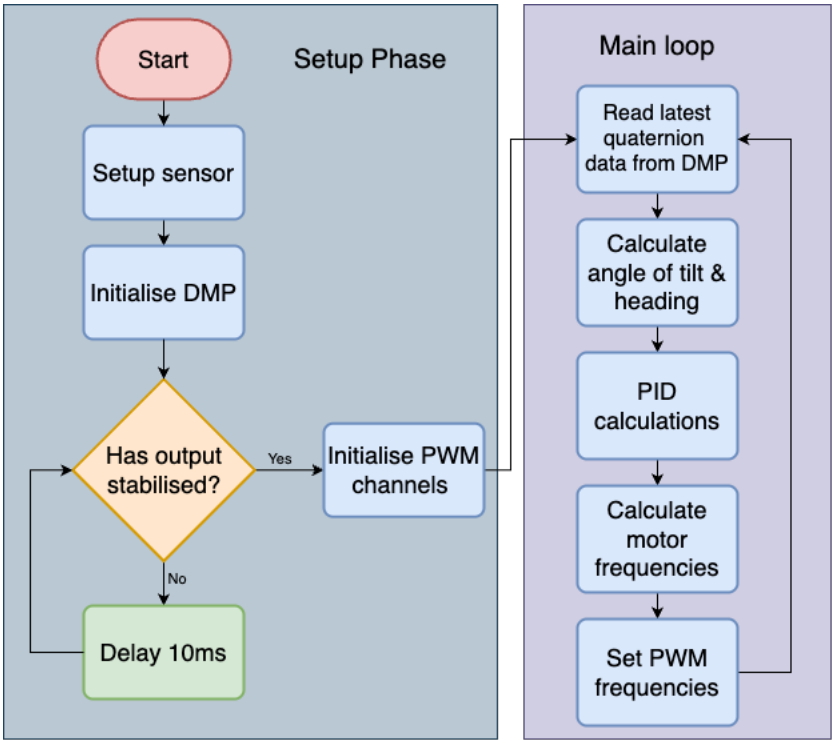


Figure 19 - Flowchart of main program

After system initialisations, the tilt angles are computed each loop from the latest available quaternion data. The ψ angle is then passed into a PD controller with a reference value of 0 to control towards maintaining an upright position. The controller is tuned to output a speed in ms^{-1} that the robot should drive at, which when combined with the α bearing, gives a desired velocity. The individual required motor speeds are then calculated from this velocity followed by computing the required PWM frequencies to reach those speeds. These frequencies are then set using the `ledcChangeFrequency` function, alongside the required motor directions which are controlled on separate pins.

7 System Testing and validation

Each major subsystem was tested independently before being put together and trialled at the end.

7.1 Motor drive accuracy

Objective: Ensure the robot is accurate with its drive system, hoping to be able to drive in a commanded direction with precision and little drift over time.

Method: The robot was inverted with the wheels facing upwards and the bowling ball placed on top. The robot was commanded to drive in a range of angles at a constant velocity for a minute each while a camera was mounted directly lined up over the centre of the ball. Markings on the

ball were used to identify in the recorded footage the drift (if any) and accuracy of the drive system.

Results: The footage was annotated with guidelines as shown in Figure 20 to better visualise if the ball was turning consistently and accurately.

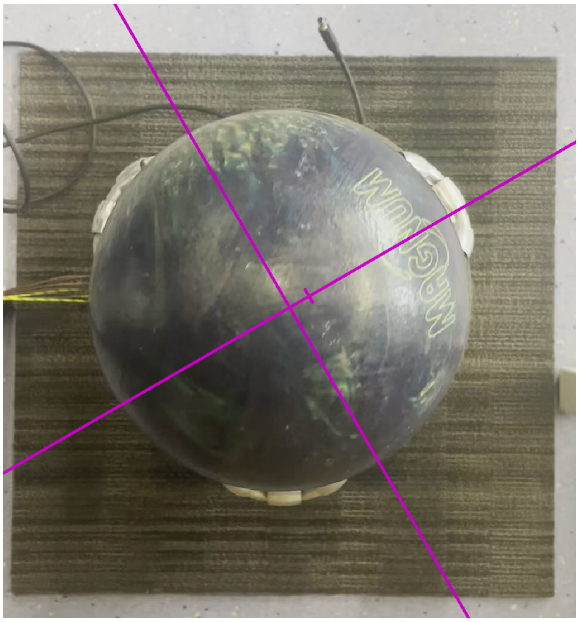


Figure 20 - Annotated test footage

Some small drifts were observed on the more complex headings and after the minute the markings had shifted slightly. The ball occasionally jolted a small amount during the process causing some disturbance in position.

Conclusions: The drive system is fit for purpose, managing to maintain accurate angles in short term and only occasionally driving off angle after extended periods of time which is not a concern for the robot at this stage. However, it is nice to know that if the robot did achieve navigation control, it would most likely be able to operate with a good amount of accuracy and reliability.

7.2 Orientation Accuracy

Objective: Determine the accuracy of orientation calculations stemmed from sensor readings.

Method: The robot was orientated at a range of known angles from horizontal and held stationary. The orientation angles were continuously calculated using quaternion data from the DMP firmware on the IMU and then logged for comparison. 330 samples per measured angle were taken.

Results: The actual angle was subtracted from each measured/calculated angle and plotted on the graph shown in Figure 21.

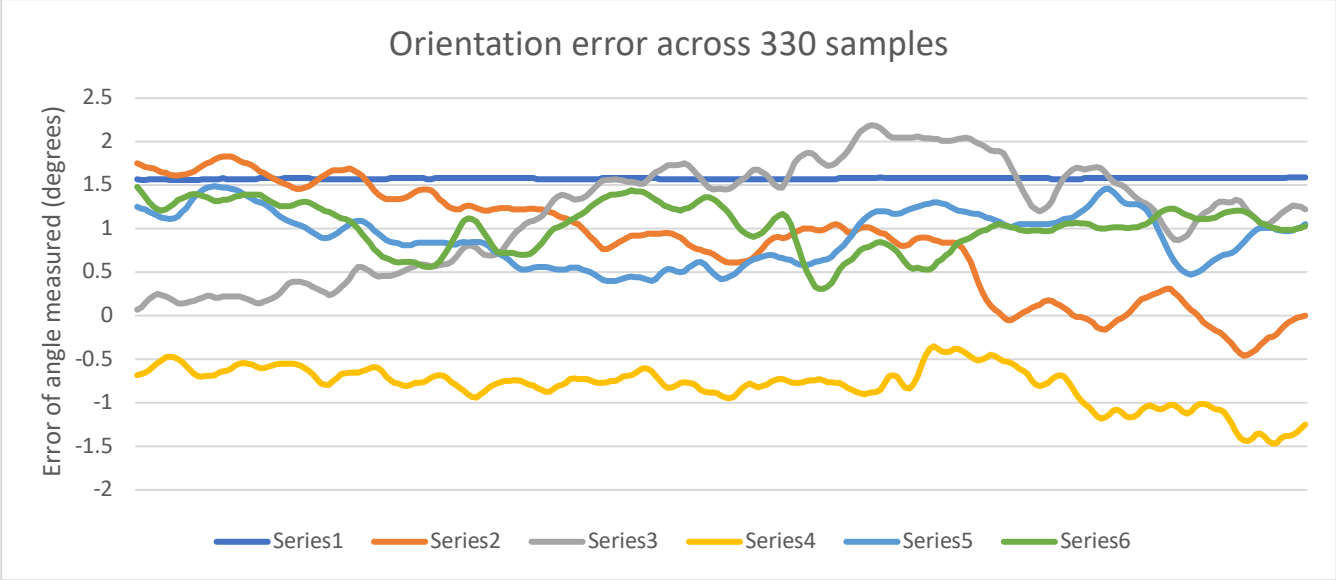


Figure 21 - Graph showing orientation measurement error

Conclusions: The data samples have large variations mostly due to unideal setup of the test rather than to the detriment of the system. Despite this, the system manages to stay within ±2.5° at each angle. A stable offset of just over 1.5° is observed for when the robot is held horizontal which might account for errors at the other measured angles too. In this case there is the possibility of the system being able to be improved by introducing a negative bias to the output.

7.3 Control loop time

Objective: Measure the total program loop time

Method: The robot was programmed with its main code and run as normal for 1000 loop cycles. An addition was made in the code to make the robot print to serial at the beginning of each loop the amount of time in microseconds had passed since the last loop iteration.

Results: The loop times were tallied and plotted on a bar chart as the data was discrete. Shown in Figure 22.

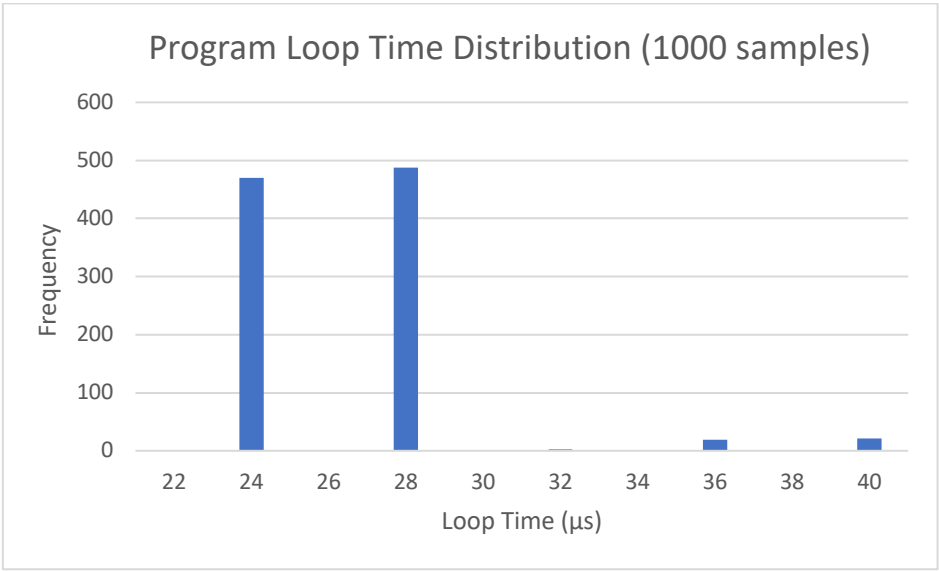


Figure 22 - Frequency distribution of program loop times

Conclusions: The average loop time observed was 26.5µs. This is suspiciously fast which indicates that there might have been an error in sampling during this test. However, despite this, the robot has been observed to respond to all changes sufficiently fast with no issues noticed.

7.4 Full system testing

Objective: Observe the robot balancing on the bowling ball

Method: The robot was powered on and left for initialisations/calibrations to finish. It was then placed carefully on top of the ball and held lightly, ready to catch the robot if it fell. With mild guidance the robot was observed attempting to balance.

Results: The robot was able to hold balance to some degree. At small deviations from vertical it managed to correct itself, however once it began tilting slightly too far it would rush to try and get the ball back underneath it which pushed the ball further and it would eventually try to drive too fast and lose balance.

Conclusions: The system is definitely fully functional but requires further control tuning for the robot to perform to a higher standard. It was also noted that the basic PD controller used for this test was only regulating the angle from vertical, which led to the case of the robot accelerating with the ball forwards to prioritise staying vertical. A more detailed controller could use other variables such as the translational velocity to refine the behaviour further.

8 Evaluation

During the implementation and testing phases, several additional insights were gained which, while out of the scope of this project, could be focuses for future improvements of the system.

8.1 Moment of inertia

The robot with its current mechanical build is quite short compared to other attempts at BBR style robots discussed earlier. Having a taller body would raise the centre of mass and greatly increase the moment of inertia of the robot around the ball's contact point. This increase in inertia would resist fast angular accelerations, effectively making the robot less prone to falling and better at remaining balanced under disturbance. Additionally, the greater inertia could allow for smoother control responses and increase margins for tuning.

8.2 Unstable wheels

When running the robot for extended periods of time or at faster speeds, the omni-wheels have a habit of dislodging themselves from the motor shafts and falling off. They are currently held in place with grub screws that grip into the shafts however these are notorious for coming loose in systems with high vibrations like this one. This loosening of the wheels probably has a negative effect on the performance of the robot before they fall off too as it could allow the wheel to skip steps around the shaft – so fixing this should be a priority. The motor shafts already have a D profile machined into them, so a good solution could be to machine D shaped hubs for the wheels to match and be held more securely in place.

8.3 Power system

When upgrading the microcontroller, the original power distribution board that was made became redundant. The ESP32 requires an input voltage of 3.3V – 5V and the current board was stepping down the 24V power supply to 12V for the Arduino. Adding a new power circuit would allow the robot to run untethered from a laptop. Eventually the robot should house its own battery so it can operate truly wirelessly and without restrictions which would be beneficial for larger scale testing.

8.4 Control theory

The most obvious improvement to the system as it stands is to implement a sufficiently more advanced control system – or at least a properly tuned PID controller. While they were not tested on the hardware, the simulation testing of other control models seemed promising and could be attempted. Kumagai and Ochiai [3] described a novel approach, using angular accelerations directly in their control loop in place of torque, making a similar assumption that was made with this project's simulation implementations. They commented that "using ball acceleration…instead of torque to drive the ball made the robot rather robust against change of inertia". The motor drive function of this project's robot could be tweaked to accept accelerations instead of velocities and control the motors that way instead. As mentioned above, a more advanced controller could observe more states of the robot to determine its control output such as the falling velocity or velocity of the ball (can be calculated from known wheel speeds). This would make the robot behave in a more controlled, stable manner.

9 Conclusion

The purpose of this project was to design a control system to enable balance in an existing robot. Through iterative development and problem-solving, the project reached a promising stage with the final implementation demonstrating real potential for true balance. However, the setbacks from various initial hardware limitations – only discovered as the project progressed – meant the robot did not reach optimal performance. Despite this, the inclusion of the upgraded microcontroller enabled full system functionality, aside from a final, more advanced control theory implementation. A major milestone was the retrieval and interpretation of reliable and accurate orientation data which opened up the possibility for further development. Additionally, the discovery of improved PWM channel control on the ESP32 allowed for precise and responsive motor driving which was crucial for the system.

While future work on this project could just focus on implementing a more advanced and robust control theory, some mechanical modifications such as increasing the height of the robot would also greatly benefit performance. As well as the addition of an onboard battery to power both the motors and the microcontroller as this would enable untethered operation – a crucial step for real world deployment. Once this is achieved, the robot could be programmed to not only balance in place, but to actively navigate its environment. This level of autonomous mobility has applications in wider society, particularly in service and assistance roles in public. Ultimately this project has laid the groundwork for a capable Ball Balancing Robot, highlighting the challenge and promise in combining low cost-hardware with a smart control system.

10 Works Cited

[1] T. Lauwers, G. Kantor, and R. Hollis, “One is Enough!,” Oct. 2005. Available: http://www.msl.ri.cmu.edu/publications/pdfs/isrr05.pdf

[2] D. Wyrwał and T. Lindner, “Control algorithm for holonomic robot that balances on single spherical wheel,” *MATEC Web of Conferences*, vol. 252, no. 02005, 2019, doi: https://doi.org/10.1051/matecconf/201925202005.

[3] M. Kumagai and T. Ochiai, “Development of a Robot Balanced on a Ball - First Report, Implementation of the Robot and Basic Control -,” *Journal of Robotics and Mechatronics*, vol. 22, no. 3, pp. 348–355, Jun. 2010, doi: https://doi.org/10.20965/jrm.2010.p0348.

[4] National Instruments, “The PID Controller & Theory Explained,” *www.ni.com*, Mar. 07, 2025. https://www.ni.com/en/shop/labview/pid-theory-explained.html

[5] Thorlabs, “Driver PID Settings,” *www.thorlabs.com*. https://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=9013#

[6] K. Ogata, *Modern Control Engineering*, 5th ed., no. 1. Upper Saddle River, NJ, USA: Prentice Hall, 2010, pp. 794–795. doi: https://doi.org/10.1115/1.3426465.

[7] K. van der Blonk, “Modeling and Control of a Ball-Balancing Robot Internship & Master thesis at ALTEN Mechatronics,” University of Twente, 2014. Available: https://essay.utwente.nl/65559/1/vanderBlonk_MSc_EEMCS.pdf

[8] Y. Zhou, J. Lin, S. Wang, and C. Zhang, “Learning Ball-balancing Robot Through Deep Reinforcement Learning,” Aug. 2022. Available: https://arxiv.org/pdf/2208.10142

[9] C. Fiore, “Stepper Motors Basics: Types, Uses, and Working Principles,” *MPS*, vol. A-0040, no. 1.0, Jun. 2022, Available: https://media.monolithicpower.com/mps_cms_document/2/0/2020-stepper-motors-basics-types-uses-and-working-principles_r1.0_1.pdf?_ga=2.111113349.1298969933.1746115151-911651124.1746115151

[10] Omega, “What is a stepper motor? - Principles, types and crontollers,” *Omega.co.uk*, 2019. https://www.omega.co.uk/prodinfo/stepper_motors.html

[11] Analog Devices, “Field Oriented Control (FOC) as a Hardware Building Block,” *Analog.com*, 2024. https://www.analog.com/en/lp/001/field-oriented-control.html

[12] F. Sandahl and W. Miles, “Ball ballancing robot,” Bachelor thesis, KTH Royal Institute of Technology, 2017. Available: https://www.diva-portal.org/smash/get/diva2:1201175/FULLTEXT01.pdf

[13] O. Gur and J. Feldman, “Engine-Propeller Matching,” presented at the IACAS 2023, Mar. 2023. Available: https://www.researchgate.net/publication/369362366_Engine-Propeller_Matching

[14] T. Jespersen, “Kugle - Modelling and Control of a Ball-balancing Robot,” MSc Thesis, Aalborg University, 2019. Available: http://doi.org/10.13140/RG.2.2.31490.73928

[15] “Development of a Balancing Robot to remain stable on a moving sphere,” Loughborough University, 2024.

[16] The SciPy community, “scipy.integrate.solve_ivp — SciPy v1.6.3 Reference Guide,” *docs.scipy.org*. https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.solve_ivp.html

[17] M. Soe Soe Htay, K. Win Phyu, S. Yadanar, and W. Yi Win, “Kinematics and Control A Three-wheeled Mobile Robot with Omni-directional Wheels,” presented at the ICCR , Sep. 2024. Accessed: May 05, 2025. [Online]. Available: https://www.researchgate.net/publication/383780115_Kinematics_and_Control_A_Three-wheeled_Mobile_Robot_with_Omni-directional_Wheels

[18] Sparkfun, “Arduino support for ICM_20948 w/ portable C backbone,” *GitHub*, Nov. 11, 2024. https://github.com/sparkfun/SparkFun_ICM-20948_ArduinoLibrary

[19] Arduino, “Arduino implementation of the MadgwickAHRS algorithm,” *GitHub*, Mar. 22, 2016. https://github.com/arduino-libraries/MadgwickAHRS?tab=readme-ov-file

[20] N. Liaudat, “Magnetometer calibration,” *Github.com*, 2024. https://github.com/nliaudat/magnetometer_calibration

11 Appendix A

This appendix provides the full set of dynamic equations derived using the Euler-Lagrange method, following the modelling approach from Van der Blonk [7] described in Section 5.1 and according to Figure 7.

The equation takes the form:

$$M(q_{yz})\ddot{q}_{yz} + C(q_{yz}, \dot{q}_{yz})\dot{q}_{yz} + G(q_{yz}) = \tau_{ext}$$

With the Matrices $M(q_{yz})$, $C(q_{yz}, \dot{q}_{yz})$ and $G(q_{yz})$ defined as:

$$M(q_{yz}) = \begin{bmatrix} I_S + r_S^2 m_{tot} + \frac{r_S^2}{r_W^2} I_W & r_S \lambda \cos(\psi_x) - \frac{r_S^2}{r_W^2} I_W \\ r_S \lambda \cos(\psi_x) - \frac{r_S^2}{r_W^2} I_W & r_{tot}^2 m_W + \frac{r_S^2}{r_W^2} I_W + I_B' \end{bmatrix}$$

$$C(q_{yz}, \dot{q}_{yz}) = \begin{bmatrix} 0 & -r_S \lambda \dot{\psi}_x \sin(\psi_x) \\ 0 & 0 \end{bmatrix}$$

$$G(q_{yz}) = \begin{bmatrix} 0 \\ -\lambda g \sin(\psi_x) \end{bmatrix}$$

$$m_{tot} = m_S + m_W + m_B$$

$$r_{tot} = r_S + r_B$$

$$\lambda = m_W(r_S + r_W) + m_B l$$

$$I_B' = I_B + m_B l$$

Where parameters are shown below:

Parameter	Description
r_S	Radius of the ball (m)
r_W	Radius of virtual actuating wheel (m)
I_S	Moment of inertia of the ball (kg·m^2)
I_W	Moment of inertia of the virtual actuating wheel in the *yz*-plane (kg·m^2)
I_B	Moment of inertia of the body of the robot in the *yz*-plane (kg·m^2)
l	Distance between COM of the ball and COM of the body of the robot (m)
m_S	Mass of the ball (kg)
m_W	Mass of the virtual actuating wheel (kg)
m_B	Mass of the body of the robot (kg)

12 Appendix B

This appendix shows the dynamics function that was fed into SciPy's `solve_ivp` method to calculate dynamic behaviour over time.

```
1. def dynamics(s, t, state):
 2.     '''
 3.     Computes q_dot and q_ddot given current state and control input.
 4.
 5.     Parameters:
 6.         t (float): Current time
 7.         state (array): [q, q_dot] where q = generalized coordinates, q_dot = velocities
 8.         control_input (array): Torques/forces (tau) (size: nx1)
 9.
10.     Returns:
11.         dstate_dt (array): Time derivatives [q_dot, q_ddot]
12.     '''
13.     q = state[:2]
14.     q_dot = state[2:]
15.
16.     # stationary ball
17.     # q[0] = 0
18.     # q_dot[0] = 0
19.
```

```
    torque_in = s.controllerFunc(s, state, t=t)
    if s.control == "Accel":
        torque_in *= s.Iw

    translated_torques = np.array([torque_in*s.Rs/s.Rw, torque_in*-s.Rs/s.Rw])
    s.control_torques.append((t, torque_in))
    if s.friction:
        q_ddot = np.linalg.inv(s.M(q)) @ (translated_torques - s.C(q, q_dot) @ q_dot -
s.G(q) s.D(q_dot))
    else:
        q_ddot = np.linalg.inv(s.M(q)) @ (translated_torques - s.C(q, q_dot) @ q_dot -
s.G(q))
    return np.concatenate((q_dot, q_ddot))

```

```
    def M(s, q, lib=np):
        theta, psi = q
        if lib == np:
            m = np.array([
                [s.Is+(s.Rs**2)*s.Mtot+((s.Rs**2)/s.Rw**2)*s.Iw, s.Rs*s.Lam*np.cos(psi)-
((s.Rs**2)/s.Rw**2)*s.Iw],
                [s.Rs*s.Lam*np.cos(psi) - ((s.Rs**2)/s.Rw**2)*s.Iw, (s.Rtot**2)*s.Mw +
((s.Rs**2)/s.Rw**2)*s.Iw + s.Idb]
            ])
        elif lib == sym:
            m = sym.Matrix([
                [s.Is+(s.Rs**2)*s.Mtot+((s.Rs**2)/s.Rw**2)*s.Iw, s.Rs*s.Lam*sym.cos(psi)-
((s.Rs**2)/s.Rw**2)*s.Iw],
                [s.Rs*s.Lam*sym.cos(psi) - ((s.Rs**2)/s.Rw**2)*s.Iw, (s.Rtot**2)*s.Mw +
((s.Rs**2)/s.Rw**2)*s.Iw + s.Idb]
            ])
        return m

    def C(s, q, q_dot, lib=np):
        ''' Returns the Coriolis matrix C(q, q_dot)'''
        theta, psi = q
        d_theta, d_psi = q_dot
        if lib is np:
            c = np.array([
                [0, -1*s.Rs*s.Lam*d_psi*np.sin(psi)],
                [0, 0]
            ])
        elif lib is sym:
            c = sym.Matrix([
                [0, -1*s.Rs*s.Lam*d_psi*sym.sin(psi)],
                [0, 0]
            ])
        return c

    def G(s, q, lib=np):
        ''' Returns the Gravity vector G(q)'''
        theta, psi = q
        if lib is np:
            g = np.array([0, -s.Lam*s.grav*np.sin(psi)])
        elif lib is sym:
            g = sym.Matrix([0, -s.Lam*s.grav*sym.sin(psi)])
        return g

    def D(s, q_dot):
        u_theta = 1
        u_psi = 1
        theta_dot, psi_dot = q_dot
        d = np.array([u_theta*theta_dot, u_psi*psi_dot])
        return d

```

13 Appendix C

13.1 PID Controller class

```
class PIDController():
```

```
    def __init__(self, Kp, Kd, Ki):
        self.Kp = Kp
        self.Kd = Kd
        self.Ki = Ki
        self.count = 0
        self.saved = []
        self.integ_add = 0

    def torque_control(self, plant, state, desired_state=[0,0,0,0], t=None):
        q = state[:2]
        q_dot = state[2:]

        q_desired = desired_state[:2]
        q_dot_desired = desired_state[2:]

        psi_desired = q_desired[1]
        psi = q[1]
        error = psi-psi_desired

        psi_dot_desired = q_dot_desired[1]
        psi_dot = q_dot[1]

        self.integ_add += error
        return self.Kp*error + self.Kd*(psi_dot-psi_dot_desired) + self.Ki*self.integ_add

```

13.2 LQR Controller class

```
class LQRControl():
    def __init__(self, plant):
        self.calcSS(plant)

    def control(s, plant, state, desired_state=[0,0,0,0], t=None):
        u = -s.K @ state
        return u[0]

    def calcSS(self, plant: Plant):
        theta, psi, theta_dot, psi_dot, ddphi = sym.symbols('theta psi theta_dot psi_dot ddphi')
        M = plant.M([theta, psi], lib=sym)
        C = plant.C([theta, psi], [theta_dot, psi_dot], lib=sym)
        G = plant.G([theta, psi], lib=sym)
        q_ddot = M.inv() * (sym.Matrix([[ddphi*plant.Iw*plant.Rs/plant.Rw], [ddphi*plant.Iw*-
plant.Rs/plant.Rw]]) - C * sym.Matrix([[theta_dot], [psi_dot]]) - G)

        x = sym.Matrix([theta, psi, theta_dot, psi_dot])
        f = sym.Matrix([theta_dot, psi_dot, q_ddot[0], q_ddot[1]])

        A = f.jacobian(x)
        B = f.jacobian(sym.Matrix([ddphi]))

        # Evaluate at equilibrium (theta=0, psi=0, theta_dot=0, psi_dot=0, tau=0)
        A_lin = A.subs({theta: 0, psi: 0, theta_dot: 0, psi_dot: 0, ddphi: 0})
        B_lin = B.subs({theta: 0, psi: 0, theta_dot: 0, psi_dot: 0, ddphi: 0})

        A = sym.matrix2numpy(A_lin)
        B = sym.matrix2numpy(B_lin)

        Q = np.diag([5, 500, 500, 5])
        R = np.array([[2]])

        K, _, _ = ctrl.lqr(A, B, Q, R)
        # print("LQR Gain Matrix K:", K)
        self.K = K

```

14 Appendix D

Below shows the wiring schematic of the whole system with the new ESP32 microcontroller.