Department of Design Engineering and Mathematics

Middlesex University

Development of Control and Interface Systems for a Flight Simulation Experience

Omar Maaouane Veiga

Supervisor: Michael Margolis

PDE3823 Major Project and Professional Practice

BEng Robotics and Mechatronics Engineering

2024 - 2025

Acknowledgements

I would like to sincerely thank my supervisor, Michael Margolis, for his invaluable support throughout this project. His previous work on the MDX rollercoaster motion platform provided a substantial codebase for controlling the platform, which served as the foundation for my own project. Were it not for his contributions, my project would not have been nearly as complete as it is. I am deeply grateful for his continuous support in all aspects of the project, including planning and determining the best approach.

Abstract

This report presents the design and development of a control and interface system for a flight simulation ride experience. The project is conducted in collaboration with other final-year students and the Royal Aeronautical Society as part of the Falcon 2 program [1]. Coventry University is also contributing by designing virtual flight experiences in X-Plane 11, the chosen flight simulator. The aim is to create an accessible ride for outreach events such as school visits and expositions.

While other students focus on designing the accessibility features of the simulator, this project focuses on developing the middleware between the simulator and physical platform. This middleware interprets motion data from the virtual aircraft into the appropriate platform actuator movements. This allows the software to deliver an immersive experience comparable to existing flight simulators.

A key aspect of this work is the platform's novel use of fluidic muscle actuators which operate without position feedback unlike most existing solutions. Moreover, there is limited software aimed at motion platform control via X-Plane which introduces unique technical challenges.

The system builds upon pre-existing control software from the MDX rollercoaster platform, adapted for a new wheelchair-accessible design. Extensive calibration tests were done on the muscles to account for the lack of muscle position feedback. The final deliverable is a piece of accessible software that connects X-Plane to the motion platform.

Table of contents

List of Figures

List of tables

1. Introduction

1.1. Background and Motivation

A motion platform in the context of virtual simulators, such as flight and racing simulators, is a mechanically actuated system designed to replicate real-world movements by tilting, rotating, or translating in response to simulated forces. Figure 1.1 is a CAD image of the original motion platform this project is based from [2].

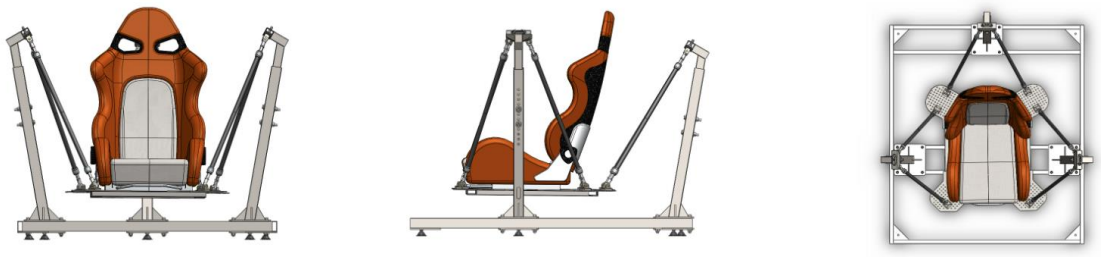


Figure 1.1-CAD image of initial MDX rollercoaster motion platform.

The platform made use of VR technology to provide an immersive yet fixed rollercoaster experience. This was a project originally led by Professor Brendan Walker and showcased at FutureFest 2015. This particular motion platform uses 6 fluidic muscles to translate and rotate the user in all 6 Degrees of Freedom (DOF). Fluidic muscles are hollow tubes made of fibre material which contract by a given amount when a given air pressure is set to them.

Even after the initial rollercoaster, there were further iterations such adding a fan in front of the user to simulate rushing air and further enhance the experience. Figure 1.2 shows the motion platform iteration with a blower fan showcased in New Scientist Live 2019.

Figure 1.2-Rollercoaster motion platform with blower fan showcased in New Scientist Live

In essence, this project aims to create a new flight motion platform based off the existing work done on the previous rollercoaster motion platform. The new flight motion platform gives the user control over the experience unlike the rollercoaster motion platform which was a fixed experience. This milestone is highly motivating because, if successful, it will be a meaningful project that continues to be used long after its completion and could even inspire others to innovate further just like the rollercoaster motion platform did for this project.

1.1 . Aims and Objectives

The main goal of this project is to develop a piece software that links X-Plane's aircraft telemetry to a wheelchair-accessible motion platform, enabling a broader range of users to experience and control flight simulations. The project's direction and challenges are largely shaped by the constraints of the pre-existing platform design of which the project focuses on.

List of key objectives this project aims to achieve is outlined in Table 1.1

Table 1.1 List of objectives this project aims to achieve.

Requirement number	Requirement Description
1	Software has a basic user interface that allows the operator to control if and how the flight motion platform moves according to the input telemetry.
2	Software provides START/PAUSE/STOP functionality to control the ride simulation as required.
3	Software is able to accurately set each muscle's length to a desired length without muscle length feedback. It will only achieve this by setting their pressure. The desired tolerance must be within +-25mm uncertainty range.
4	Software is able to set the platform's pose accurately enough for each motion to be noticeable to the naked eye. For example, the software can move the platform forwards or backwards in a visibly straight line. Target pose accuracy is very lenient in this project.
5	Software takes into account the "safe state" of the wheelchair through input dock sensors. If the chair comes loose itself from the platform, the platform will halt to a stop in a pre-determined "safe state" pose.
Advanced Objectives	
6	User interface is friendly and can be easily used by an inexperienced operator with minimal training.
7	User interface allows operator to manually control the chair's pose and see a side-by-side simulation of it.
8	Software allows support to connect chair to other flight sims aside from X-Plane.
9	Final ride platform has VR support for further immersion

1.2. Scope and Contributions

The scope within this project is strictly limited to working with the pre-designed flight motion platform which is being built into the final prototype independently of this project. This means that there will not be any focus on designing or building any part of the physical flight motion platform at any point. This project focuses on just developing the middleware between the simulator and the flight motion platform. This scope alone is enough to fill the limited 12-week project timespan.

The key contributions include adapting previously fragmented work and knowledge from various motion platforms. In addition, it incorporates independent research on the new, larger fluidic muscles used in the new, larger flight motion platform. The goal is to create a system that allows both wheelchair and non-wheelchair users to experience an immersive flight simulation.

1.3. Main project Resources and Repository Access

he GitHub repository [3] serves as the final submission repository for this project. In contrast, the GitLab repository [4] contains the complete commit history and all changes made throughout the development of the project. As a result, the GitLab repository is significantly larger, approximately 3GB, due to its inclusion of the entire project's history.

[5] references this project's blog page where this project's progress across the 12-week timespan is further detailed in a weekly basis. The blog includes further material such as weekly videos and Gantt charts over time.

[6]is the GitHub repo for the previous MDX rollercoaster motion platform control software. This repo only contains the necessary software to move the platform through code and nothing else. This project borrows existing parts of code solely to save time on developing the basic movement software from scratch.

2. Technical Background and Literature Review

2.1. Brief history and overview of what is a motion platform.

Motion platforms are mechanical systems that use hydraulic or pneumatic actuators to generate physical motion corresponding to a virtual environment or input. They enhance realism and immersion by simulating forces such as acceleration, braking, turbulence, and banking. Figure 2.1 shows an example of a motion platform being used to film point of view (POV) scenes of a moving truck for a film.

Figure 2.1-Picture of a motion platform used to film POV scenes of a moving truck for a film (Day Shift 2022)

Resource [7] links to a video showcasing motion platform's use in filming. Motion platforms are also used in professional training, research and amusement rides. Their flexibility to closely simulate any situation with movement involved whilst remaining stationary makes their applications highly diverse. A brief history and other examples where they are used will be mentioned in this subchapter.

Motion platforms have existed for over 100 years and have evolved so much since. For comparison, the earliest known motion platform known as the "Sanders teacher" [8] was made in 1910. It was an

entirely mechanical system meant to train WW1 airplane pilots. It didn't have any visual feedback elements like monitors or VR, nor it could actively move the user in a convincing manner like nowadays. In contrast, the SIMONA research simulator [9] from the Delft University of Technology was first mentioned in 1998. It uses large hydraulic cylinders and advanced projection systems to closely replicate the feeling of being in an aircraft. A comparison of earliest flight simulator (Sanders teacher 1910) to the SIMONA flight motion platform (1998) shown in Figure 2.2

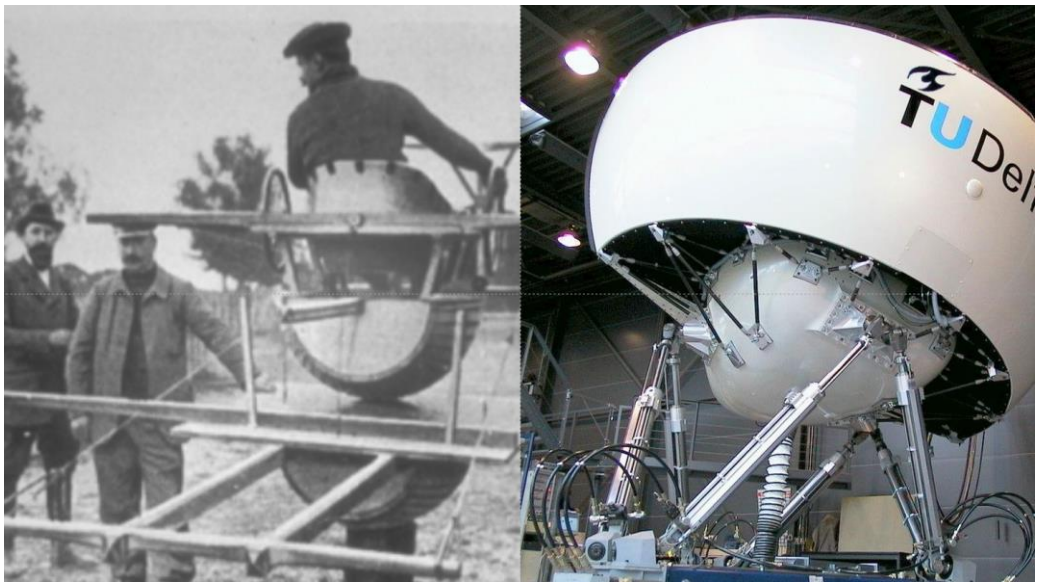


Figure 2.2-Side by side comparison of the Sanders teacher to the SIMONA research simulator.

As technology advanced from the 1910s, motion platforms began appearing more in the context of arcade videogames in the 1980s and now they are widely available as peripherals to enhance user immersion in flight simulators, racing sims and novelty ride experiences. Figure 2.3 shows an example of a motion platform specialised to be used as a funfair ride.

Figure 2.3-Picture of a 1980s motion platform ride.

In a similar scale to the SIMONA flight motion platform also the Motion systems' PS-6TL-1500 [10] which for a similar size uses rigid electric linear actuators to achieve its motion. This platform is mainly designed for professional applications, such as those in the entertainment industry. Just like in the example shown in Figure 2.1 can use the platform to film vehicle POV driving scenes without needing a real moving vehicle. Motion systems PS-6TL-1500 is shown in Figure 2.4.

Figure 2.4-PS-6TL-1500 Motion platform by motion systems.

With the evolution of technology and readily available simulation experiences in the form of videogames, motion platforms are also accessible by the average consumer. Using DOF Reality [11] as an example, platforms can come in 2DOF, 3DOF or 6DOF. Most motion platforms used in flight simulations utilize a 6DOF 'Stewart platform' configuration. This is the same configuration shown in Figure 2.1, Figure 2.2 and Figure 2.4 where 6 extend or contract to move the platform. This configuration allows movement in six directions: forward/backward, up/down, right/left, roll, pitch, and yaw, just like in a real aircraft. Figure 2.5 shows the 6 distinct motions a 6DOF motion platform is able to achieve.

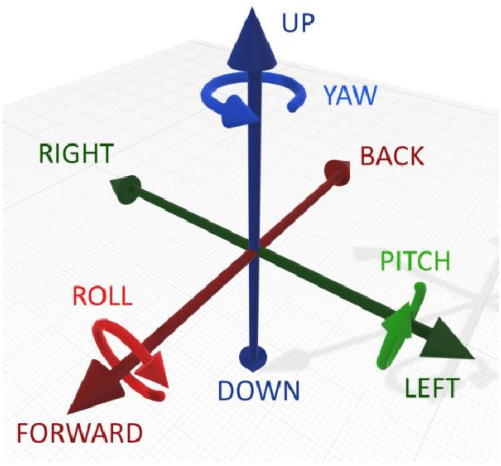


Figure 2.5-Diagram of all the possible movements an aircraft can perform while airborne.

The less advanced 2DOF and 3DOF platforms are aimed at car simulators, since a car can usually only move forwards and side to side. The 3DOF platform is a lower cost alternative to the 6DOF platform. 2DOF, 3DOF and 6DOF platforms are illustrated in Figure 2.6.

Figure 2.6-DOF Reality illustration of different platforms they offer, ranging from 2DOF, 3DOF and 6DOF

This project's platform has 6DOF and will be referred to as a flight motion platform because it will be focused on simulating a flight experience as accurately as possible. This distinction also helps to differentiate it from the rollercoaster motion platform or any other of the aforementioned motion platforms.

2.2. Technologies used on motion platforms.

2.2.1. Standard platform configuration for 6DOF

A common overall trend in industry is to use a stewart platform just like in Figure 2.7. The reason for this is because this confugration conveniently allows for the user standing on the platform to move and rotate in all possible directions through a compact platform with a fairly large working envelope. Refer to Figure 2.5 to see the different movement the platform is able to perform.

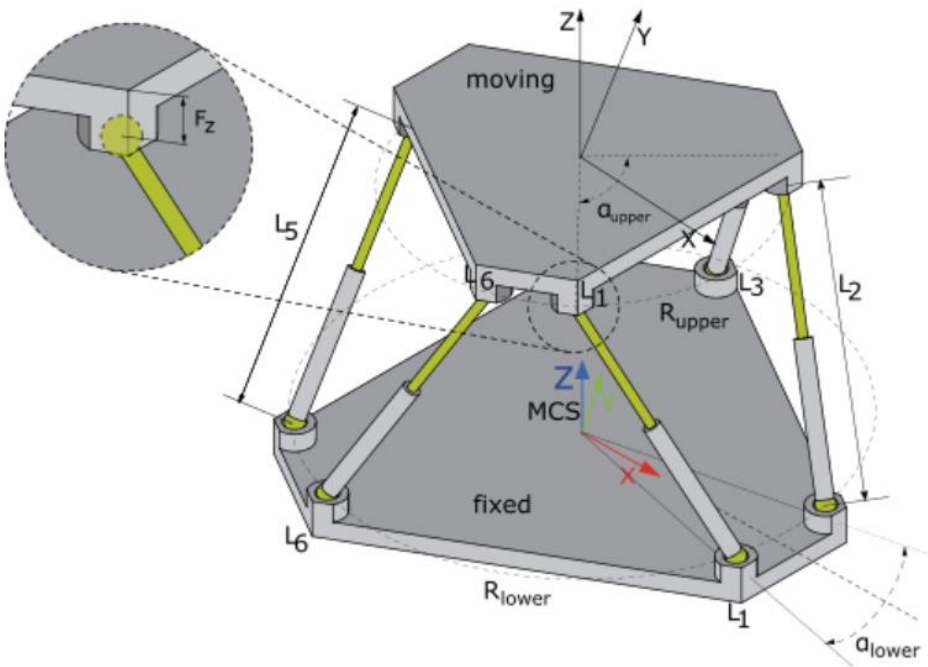


Figure 2.7-Example diagram of a Stewart platform.

2.2.2. Actuator control schemes: Closed Loop vs Open Loop

Another common trend in the industry is the use of electric, hydraulic, or pneumatic actuators with closed-loop position feedback control.

Closed-loop control is a scheme in which a computer system controls an actuator, such as a hydraulic cylinder or electric motor, while continuously receiving feedback on the actuator's actual position. Since the computer knows the actuator's position, it can determine whether to extend or retract it to achieve the desired length.

This, in turn, allows each of the six platform actuators to be set to a specific length, enabling the system to achieve a desired pose through inverse kinematics calculations. An example of an actuator with position feedback control is shown in Figure 2.8.

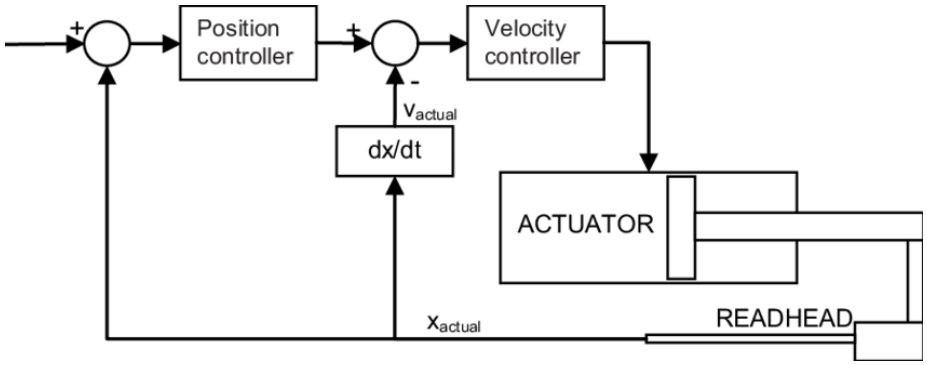


Figure 2.8-Example of linear actuator with feedback position control.

Position feedback control is generally a better option than open-loop control. In open-loop control, the system has no knowledge of the actuator's position; hence, it can only attempt to achieve the desired position or response by setting its control signal (pressure, voltage, etc.) to a predefined value. Open-loop control is very fragile and does not account for changes in the physical system, such as variations in load, actuator degradation, or friction.

In this project, the use of open-loop control for the flight motion platform's actuators was primarily a limitation imposed by the existing hardware, which had to be addressed, as will be shown later.

2.2.3. Technology behind the previous MDX rollercoaster project

Industry does not implement inverted Stewart platforms unlike the MDX rollercoaster [12]. This is likely due to the regular configuration offering ground clearance by default and thereby a better movement envelope. MDX rollercoaster picture is shown in Figure 2.9. Figure 2.10 shows the difference between a regular Stewart platform and an inverted Stewart platform.

Figure 2.9-Picture of MDX rollercoaster used in outreach event like New Scientist Live

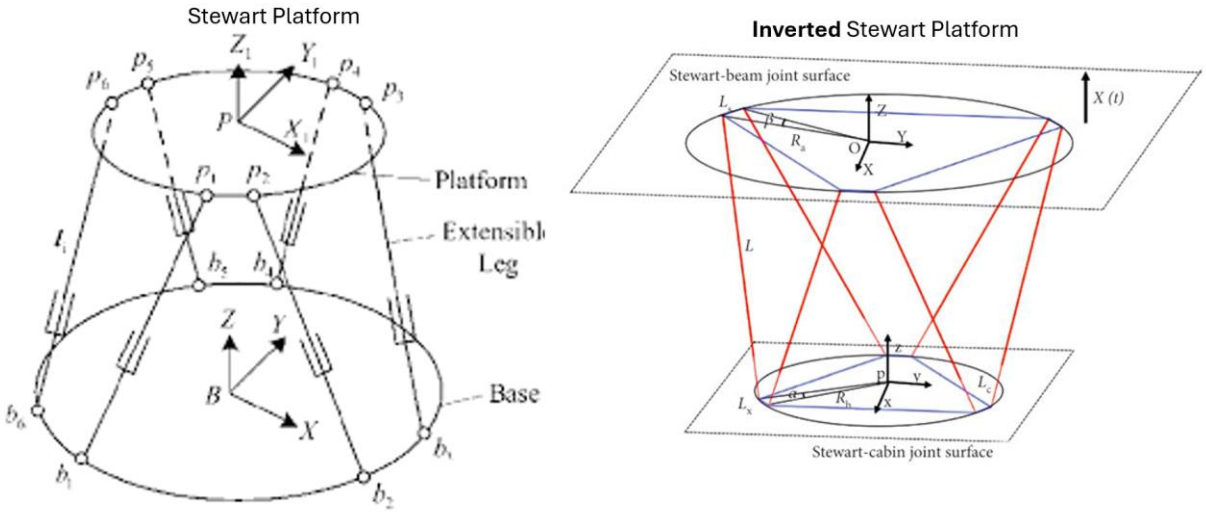


Figure 2.10-Difference between a Stewart platform and an inverted Stewart platform.

The actuators used for the platform are Festo fluidic muscles. Festo fluidic muscles are pneumatic (air pressure) linear actuators that function similarly to biological muscles. They are made of a flexible tube reinforced with a fibre mesh. When pressurised with air, the tube contracts along its length while expanding radially, generating tensile force (pulling motion). Source [13] evaluates the dynamic model of fluidic muscles in further detail. Figure 2.11 shows a contracted fluidic muscle (left) and relaxed fluidic muscle (right) alongside its response graph at different pressures.

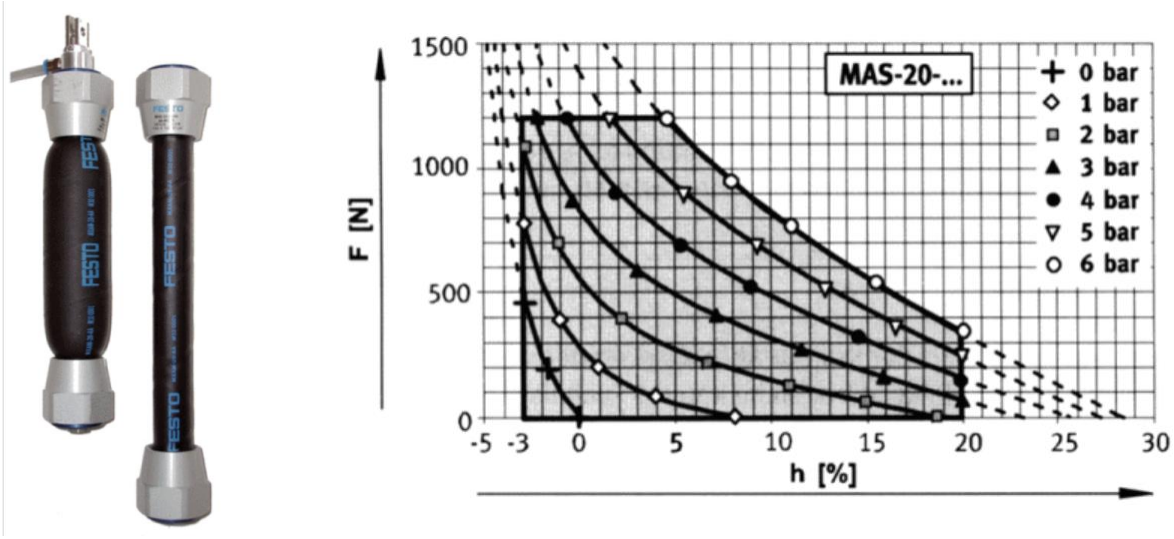


Figure 2.11-Ilustration of load vs stroke graph at different pressures for a Festo fluidic muscle from source [13]

2.2.4. Previous research solutions from other projects compared to the MDX rollercoaster's solution.

More closely related to this project, research has been conducted on connecting the X-Plane simulator to a motion platform [14] as well as using Festo fluidic muscles to control a Stewart platform [15] and [16] in separate papers, though not within the same project. Moreover, the paper on the Stewart platform using Festo muscles describes a setup where the muscles are connected to an upright pre-compressed spring and operate with closed-loop position control; unlike this project's motion platform, which lacks such control (open loop). Examples of motion platforms using fluidic muscles is shown in Figure 2.12.

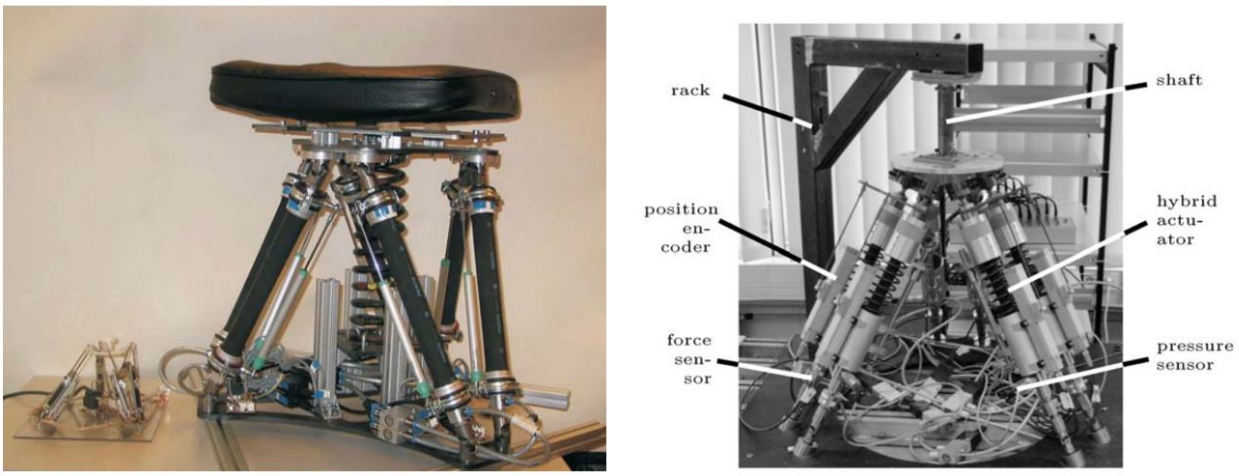


Figure 2.12-Past examples from papers of Stewart motion platforms using fluidic muscles as actuators.

Finally, a key detail worth noting is that the pressure-to-distance characteristics of the fluidic muscle are nonlinear due to the high 'dry friction' between the muscle fibres. One notable approach, presented in

paper [17], used a multi-layer perceptron (MLP) network trained on muscle pressure and angle data to develop a model with 84.66% accuracy, capable of controlling a planar 2DOF fluidic muscle robotic arm. This paper highlights the importance of different approaches in modelling nonlinear systems. Figure 2.13 shows an illustration from the paper of planar 3DOF robotic arm.

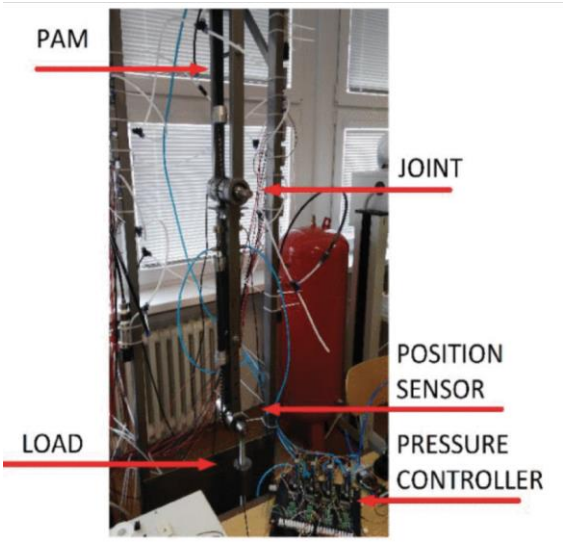


Figure 2.13-Illustration from paper [17] of a planar robotic arm controller by two fluidic muscles

A simpler yet still functional approach which was used for the MDX rollercoaster platform will be to map the muscle's pressures and distances to a table for a known load as illustrated in Figure 2.14.

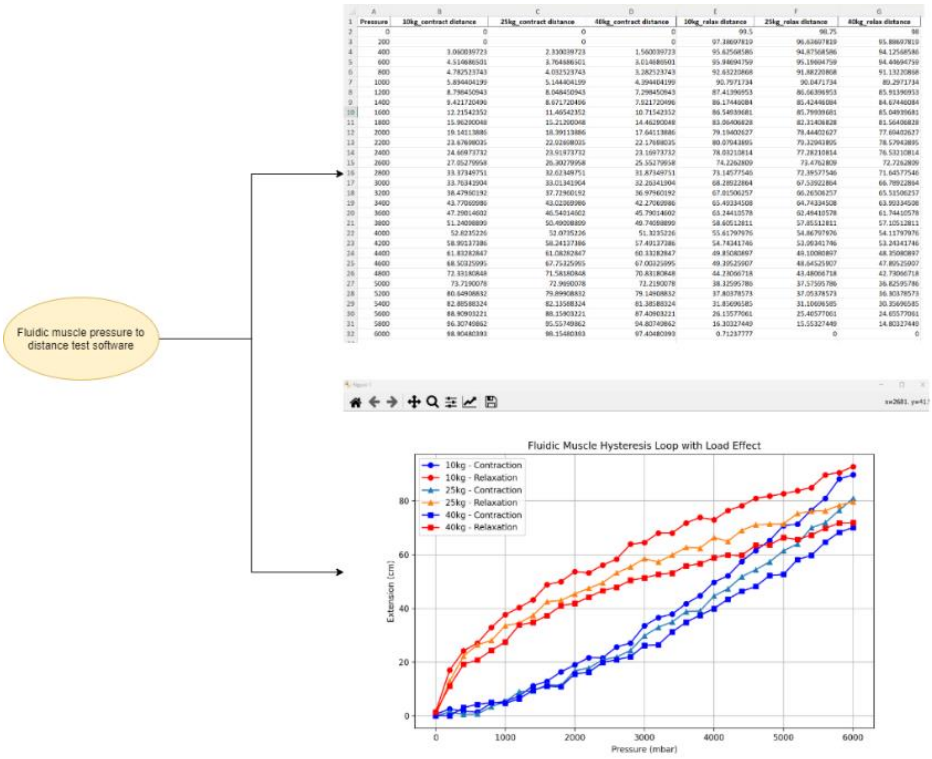


Figure 2.14-Illustration of results the program aims to provide with in the Week 8 experiments.

Using this lookup table approach will still ensure that the correct pressure is applied to each muscle for a given load, allowing the appropriate distance to be achieved and, consequently, the target pose.

2.2.5. Brief overview of Inverse kinematics for the flight motion platform and washout algorithms

Two fundamental concepts required for the platform to work are inverse kinematics [18] and washout filters [19]. Inverse kinematics are a set of mathematical formulas which allow the platform's controller to take in an input 6D pose (3D translation and 3D rotation) and output the required actuator lengths based on the platform's physical characteristics. In turn this allows the physical platform to achieve the desired pose which is easier to set than manually controlling every muscle's stroke individually. Figure 2.15 further illustrates the concept of inverse kinematics.

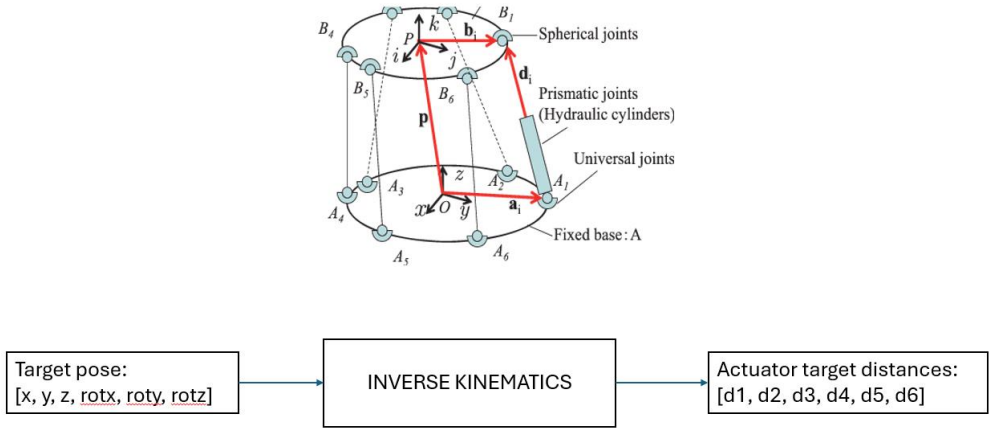


Figure 2.15-Inverse kinematics of a Stewart Platform illustration.

Another key concept in motion platforms which will become more relevant later on the project's development are washout filters. Washout filters are implemented between the target pose and inverse kinematics stage of the motion simulation as shown in Figure 2.16.

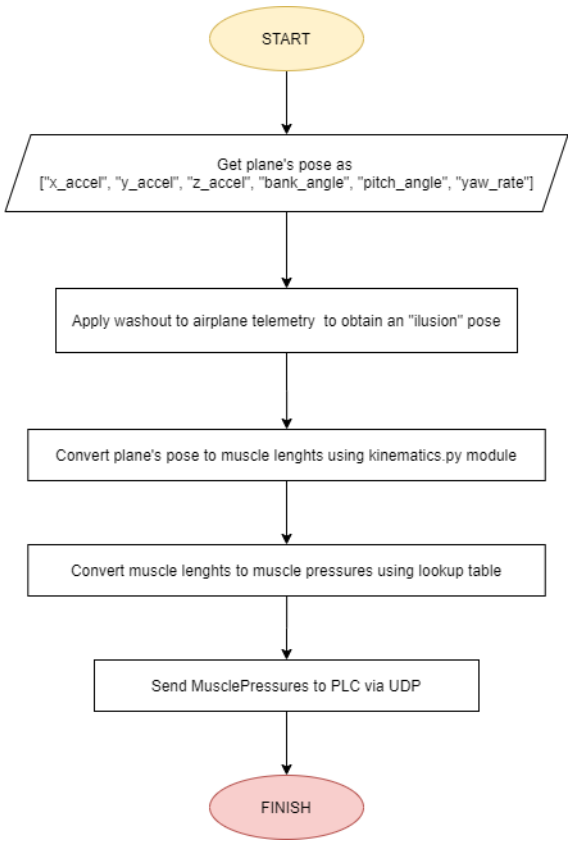


Figure 2.16-Plane Pose to muscle distances accounting for washout. This graph is from around project week 5.

Washout filters in motion platforms are motion cueing algorithms used to create realistic motion sensations while keeping the platform within its physical limits. They filter and scale the movement data from the simulator to prevent excessive displacement while still providing a convincing sense of acceleration and orientation changes. The paper [19] further explains the concept of washout in further mathematical detail. The video in resource [20] qualitatively explains how flight simulators trick the human brain into believing they are flying an aircraft while using the SIMONA research simulator [9] as an example.

Figure 2.17 demonstrates how a washout filter gradually moves the platform back to its resting roll position once the plane's angular roll acceleration ceases. This ensures the platform remains within its physical constraints while maintaining the illusion of rolling in an actual plane. It is worth noting that using widescreen monitors and covering as much of the user's field of view significantly enhances the illusion of movement, as visual and vestibular inputs are the key factors that make the brain perceive motion.

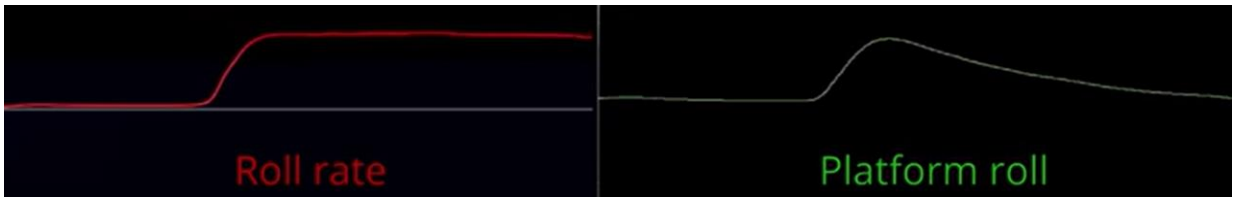


Figure 2.17-Washout filter illustration for roll and pitch.

Figure 2.18 illustrates another way washout filters enhance the illusion of movement. By slightly tilting the user and covering their field of view, a reaction force is created against the chair, which, over time, becomes indistinguishable from acceleration in an actual plane.

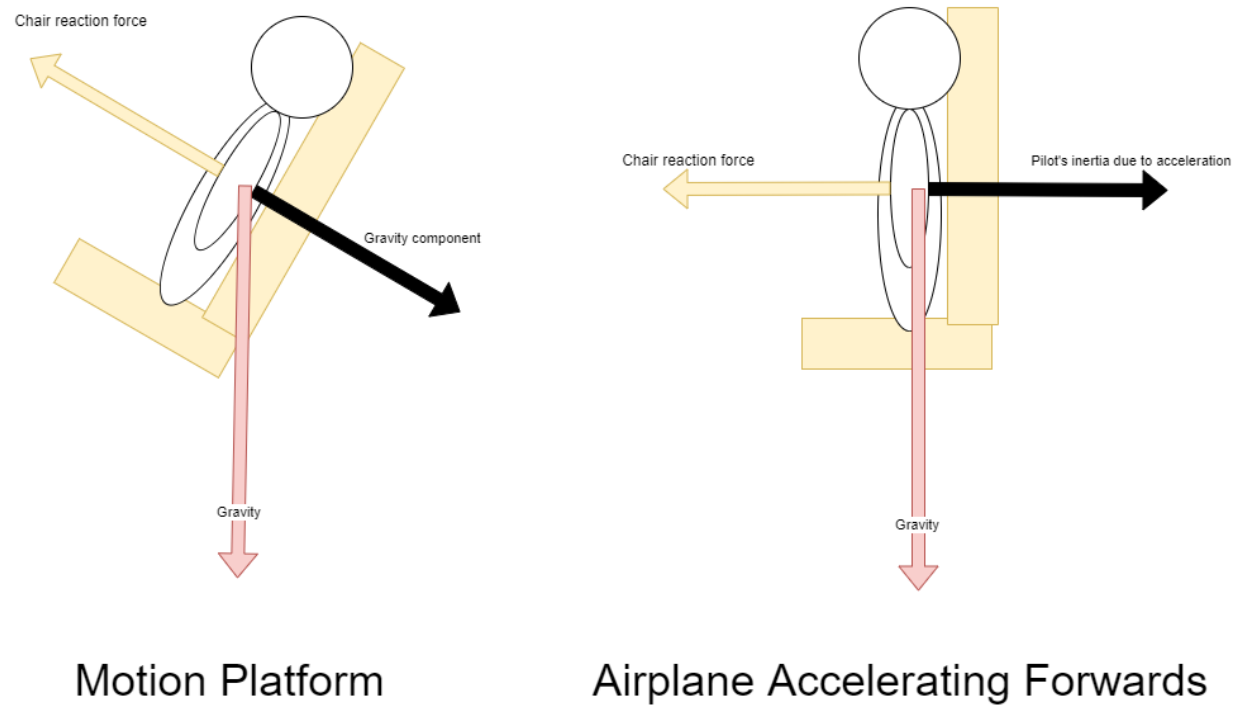


Figure 2.18- Illustration of illusion made by motion platforms to create the sense of acceleration.

Figure 2.19 illustrates in better detail how the platform will create an illusion of acceleration during take-off by directly surging forwards for immediate acceleration and slowly tilting over time to create an illusion of constant acceleration. The user feels as if they are accelerating because gravity is an

accelerating force. When combined with sufficient visual input suggesting motion in an airplane, it creates a convincing illusion of movement where none actually exists.

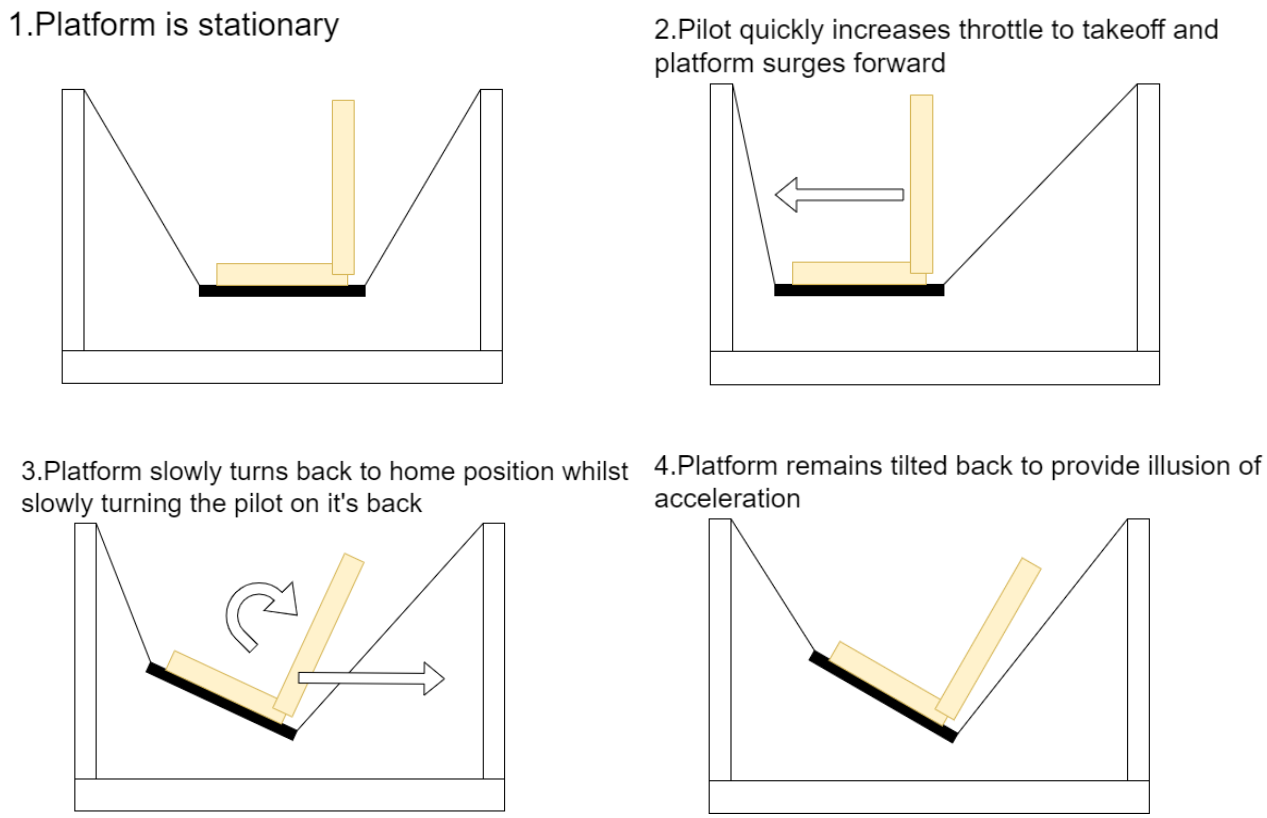


Figure 2.19-Ilustration of washout for low frequency transient acceleration.

2.3. Project challenges and limitations

Based on previously shown research, this project's main challenges and limitations arise from:

- Most motion platforms use rigid actuators controlled hydraulically, pneumatically, or electrically. This project's platform will be controlled by six Festo pneumatic fluidic muscles (similar to traditional McKibben muscles) without closed-loop feedback (open-loop). This open-loop scheme means that the relationship between muscle pressure and displacement for a given load must be mapped, which formed a significant part of this project. This process will be referred to as "calibration".
- There are few examples of motion platforms prioritizing wheelchair accessibility with software designed to accommodate such users. Therefore, this project had to rely on the person's output and extensive analysis of the actual system to ensure proper implementation.
- The final chair will incorporate newer, longer fluidic muscles that require calibration to be set to a specific length without directly sensing their actual length. This meant that designing, testing, and validating calibration procedures was unavoidable.
- The platform configuration differs from conventional Stewart platforms, where actuators operate in compression. This project, however, uses an inverted design in which actuators function in tension. As a result, the project's development had to rely on an older existing codebase for the MDX rollercoaster chair, which required a significant amount of time to adapt to and fully understand.
- There is limited research and development on software that connects the X-Plane flight simulator to motion platforms or peripherals using Python. This added additional research and development tasks to the existing workload.

Control and Interface Systems for a Flight Simulation Experience Report. Veiga, March 2025

- Whilst this project aims to fulfil the requirements of the RAeS, it does not aim to be used in their actual program (at least within the given project development timespan). This is due to the limited 12-week development time frame which is completely needed to actually develop a functional project prototype. In order to submit it to the RAeS, more time would be needed to fully test the finish project and ensure it can be handed over to the RAeS with no technical issues.

Despite challenges and limitations, this project still has plenty of room to grow and be fully developed beyond the 12-week final year project timeframe. Given more time, areas for improvement, such as testing and refining the software for broader usability—are clearly identifiable.

2.4. Lit review

Table 2.1 below summarises and compares relevant industry solutions and research papers based on key criteria, highlighting their strengths and limitations in relation to this project.

Table 2.1-Comparison of existing solutions

Solution/Research	**DOF**	**Actuation Type**	**Control Method**	**Position Feedback**	**Application Relevance**
Sanders Teacher [8]	2DOF	Mechanical	Manual Control	No	Early flight training
SIMONA Simulator [9]	6DOF	Hydraulic	Closed loop	Yes	High-fidelity flight simulation
PS-6TL-1500 [10]	6DOF	Electric Linear Actuators	Closed loop	Yes	Professional defence and simulation
MDX Rollercoaster [12]	6DOF	Fluidic Muscles	Open loop	No	Amusement ride, outreach events
X-Plane Motion Platform from video [14]	6DOF	Electric Linear Actuators	Closed loop	Yes	Motion platform software integration
Paper on Fluidic Muscle Stewart Platform [16]	6DOF	Fluidic Muscles	Closed loop	Yes	Research on pneumatic actuators
Paper on MLP-based Fluidic Muscle Control [17]	2DOF	Fluidic Muscles	Machine Learning (MLP)	Yes	Nonlinear actuator modelling
This Project	**6DOF**	**Fluidic Muscles**	**Open loop (Lookup Table)**	**No**	**Accessible motion platform for simulation**

As seen from Table 2.1 this project's motion platform is extremely different because it combines different features from existing solutions into a project that has not been made before. It's an inverted Stewart platform which is rarely implemented in this context. It's controlled from X_Plane telemetry yet has no position feedback control. It uses larger muscles which have no available data on their pressure to distance characteristics (therefore must be calibrated) and so on.

This project will aim to fill the gaps between combining these features and overcome all the aforementioned challenges.

2.5. Summary

By the end of this project, a fully functional prototype of a flight motion platform will be completed. This platform will be designed to accommodate both wheelchair users and those without mobility impairments. Once the user enters the platform, they will be able to experience and control a flight simulation using peripherals such as a flight joystick for control and widescreen monitors for visual feedback. Throughout the simulation, the platform will move in real time based on the user's actions, creating an immersive and engaging experience. This system can be utilized in outreach events or project showcases.

Overall, this work is important because it explores an under-researched approach to motion platform control and expands accessibility in flight simulation experiences, building upon and evolving from previous existing work. The findings of this project have the potential to contribute to future research on alternative actuation methods and inclusive design in simulation-based training and entertainment, making it a valuable step toward improving both accessibility and interactivity. With this foundation, the next section will delve into the design and implementation process, outlining the technical aspects and innovations that drive the project forward.

3. Proposed System: Design and Implementation

3.1. System Overview

The motion platform in this project will be a modified version of the MDX rollercoaster ride which allows wheelchair users to ride the platform by mounting their wheelchair onto a cart that is pulled onto the platform. Render of final platform to be implemented is shown in Figure 3.1.

Figure 3.1- Render of motion platform to be made in this project.

Figure 3.2 showcases the initial high-level outline of how the flight motion platform system works as a whole. The VR headset can be replaced by any form of visual feedback such as a widescreen monitor. The reason VR is more preferable for a more immersive experience is because vision plays a major role alongside the vestibular system in a human's perception of motion. VR captures the human's entire vision and updates their view in real time which means the experience will be far more immersive.

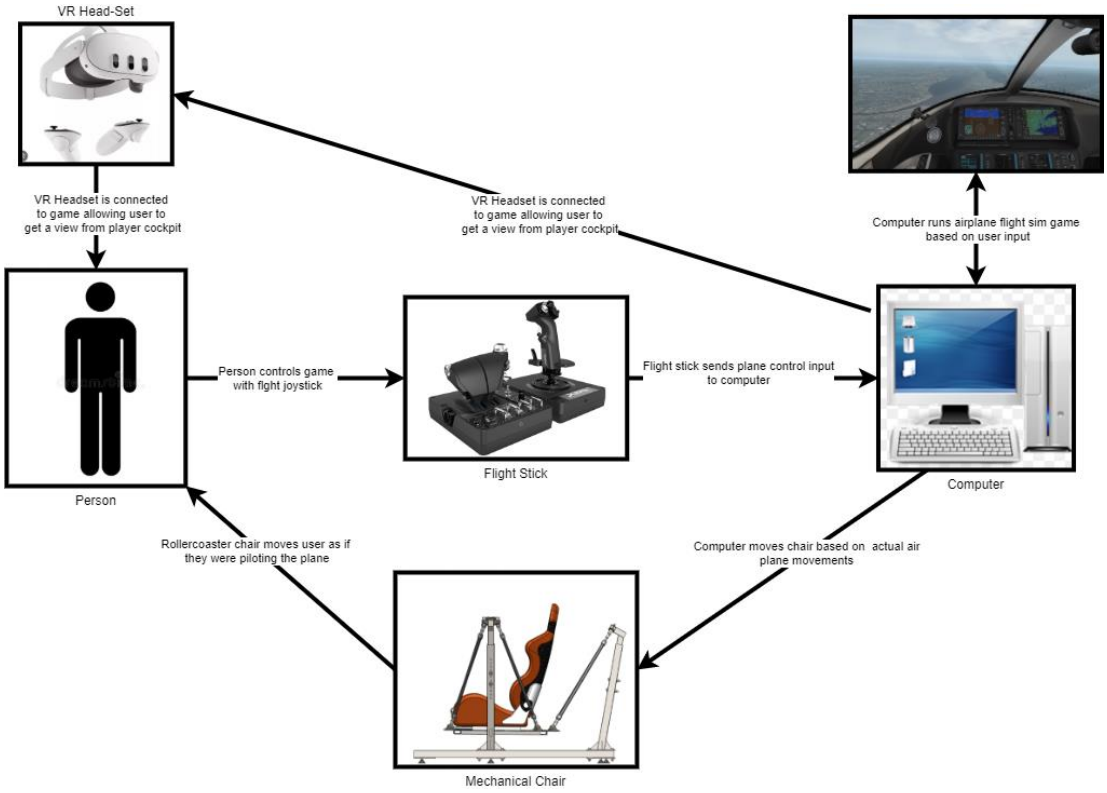


Figure 3.2-High level illustration of how all the components in flight motion platform are meant to interact together

It is worth noting that the flight motion platform is a completely new and separate build from the existing rollercoaster motion platform shown in Figure 2.9. The flight motion platform only adapts the rollercoaster's codebase and overall design, not its hardware. The muscles used in this platform are longer and wider in order to accommodate a bigger payload of up to 250kg. Just like mentioned before this will require calibrating the muscles in order to obtain their distance-to-pressure characteristics.

3.2. Software Structure and Components

Since this project focuses on the control software that interfaces between the simulator and the physical platform, the configuration and details of the physical system were established from the outset. However, due to the flight motion platform's unique design, several knowledge gaps needed to be addressed, including how to obtain telemetry data from X-Plane, how to convert this data into a valid platform output, and how to develop an intuitive GUI application that not only integrates these components but also provides the operator with control over the platform.

Figure 3.3 shows a high-level flowchart where the airplane telemetry data is sent into the main control software which controls the output to the flight motion platform based on the operator's inputs or its current state. This is an early-stage diagram; hence the Python client refers to the flight motion platform control software instead.

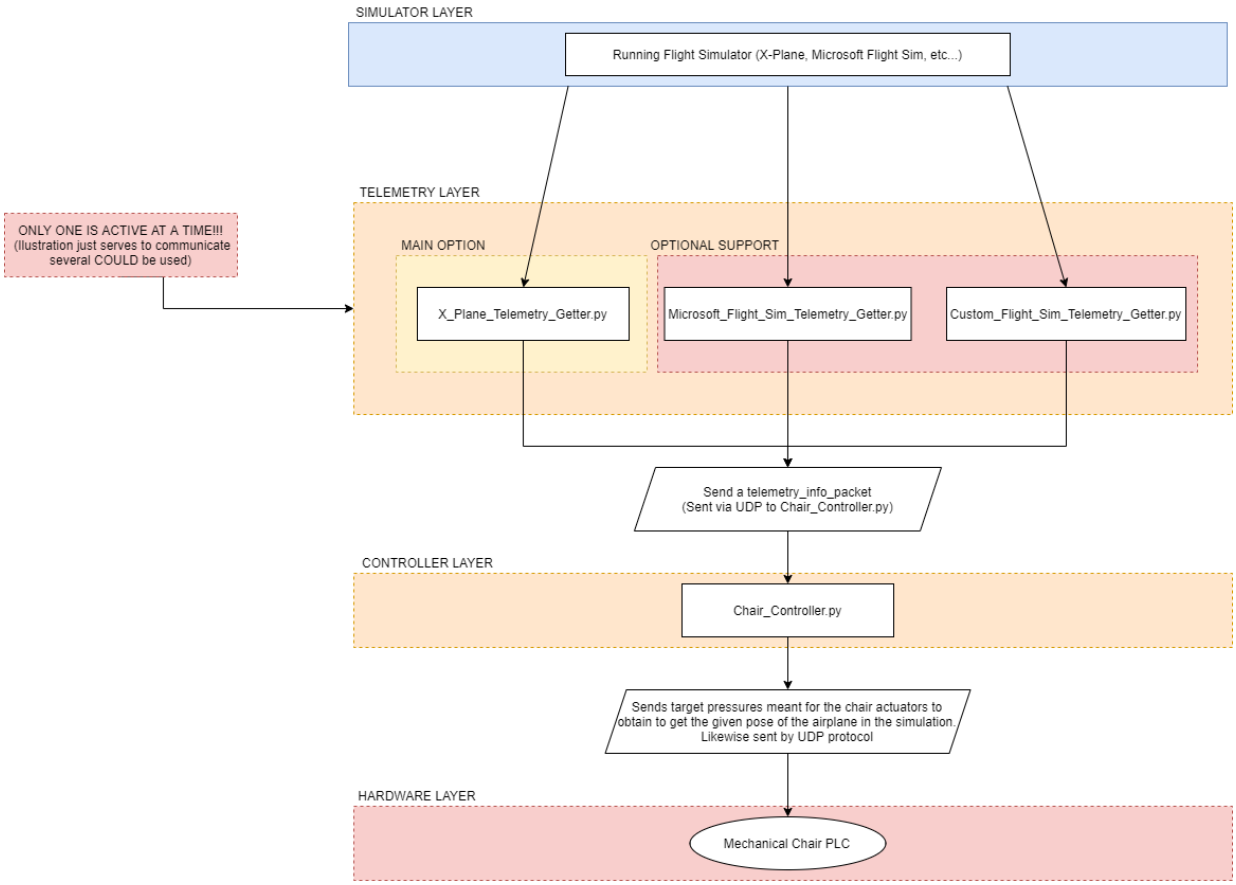


Figure 3.3-High level outline showcasing how the flight simulator is to interact with control software (python client) and thereby the mechanical chair(flight motion platform)

Note that Figure 3.3 is only a rough outline and script names will not be the same as the ones shown later. This will be explained accordingly.

Another key detail within this system is how the communication between programs or “layers” is done via User Datagram Protocol (UDP). UDP is a faster version of TCP which allows for flexible, fast and modular connectivity between software components and therefore better code maintainability. For example, by using UDP protocol to send a specific message to the software, virtually any other program that sends that message can be used to control the platform. This can be a manual control utility app or even another vehicle simulator (car racing sim, heli sim, drone sim, etc).

This use of UDP messaging also extends to the output to the physical motion platform. The output tells the platform which pressures to set to each muscle. As previously mentioned, there’s no feedback from the physical platform. It’s an open-loop system. As the report continues, the benefits and where UDP is used will become more apparent.

The project followed an agile development cycle, meaning there was continuous feedback from the RAes and Coventry University, which constantly shaped the project’s requirements. The agile approach introduced an additional challenge, requiring further adaptability on top of the inherent challenges of the project.

3.3. References to project repositories

4. High level design outline of control software

4.1. Core functionality high level outline.

The first step towards designing the control software after defining what it should do back in Table 1.1 was to further define its logic and how it'll specifically work. This section will focus on outlining the control software which was shown in the controller layer of Figure 3.3.

In order to better visualize what the software will do, a "state machine" architecture was chosen. In essence, the main program will act like a state machine whereby it can only be performing one set of logic (from many) at a time, and it can then transition between states when given conditions are met. The state machine diagram for software is shown in Figure 4.1.

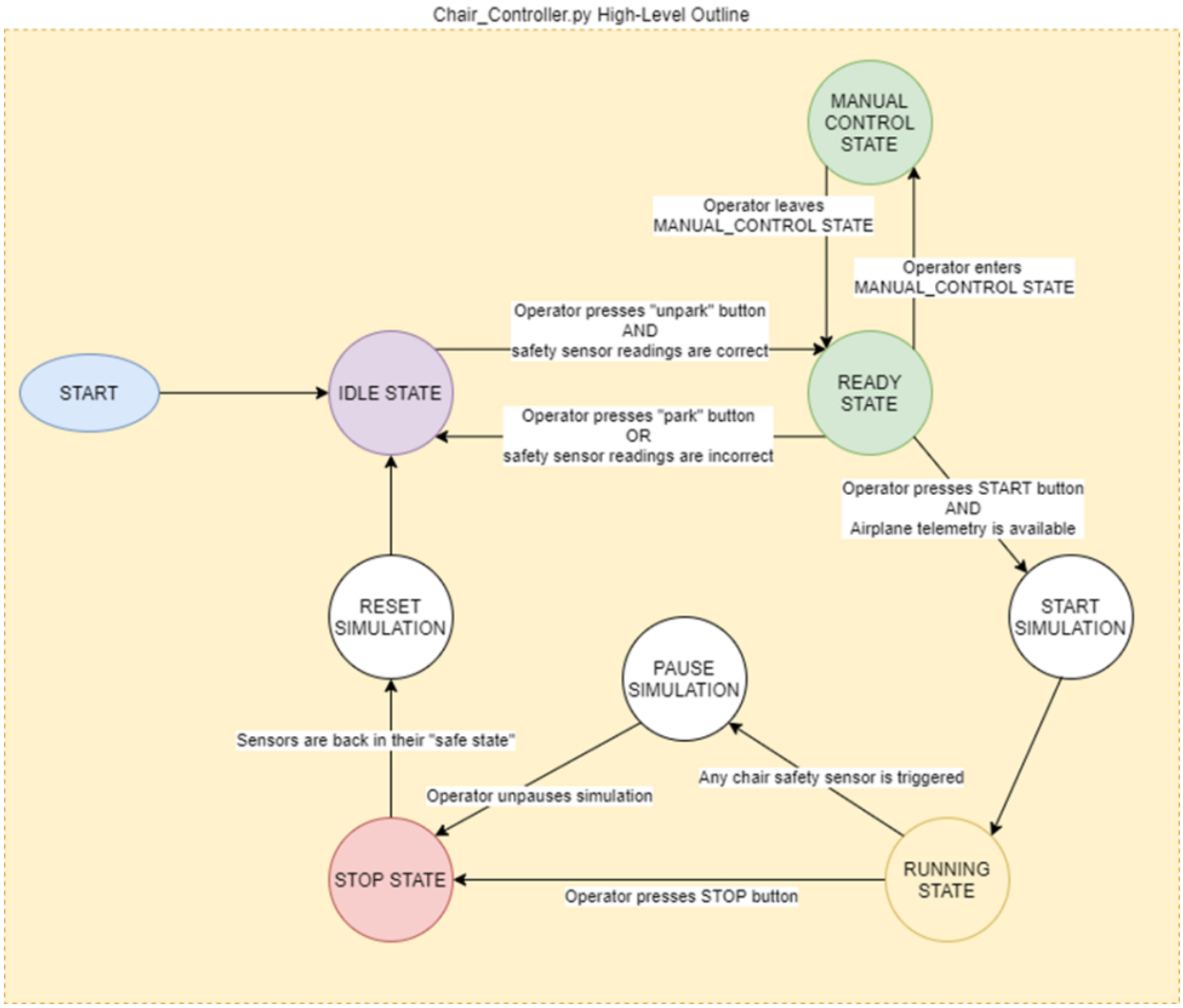


Figure 4.1-State machine diagram of flight motion platform control software.

As Figure 4.1 shows, the software can only be in one of multiple states and can only transition between states when certain actions occur, and certain conditions are met. For example, the operator controlling the software can only enter the running state if the software is receiving valid airplane telemetry and the operator presses the start button. The control software will be accessible through a python GUI application similar to existing programs anyone would expect to find in equipment like this.

What each state does is explained in Table 4.1.

Table 4.1-Table explaining what each state in control software does.

State	Description
IDLE STATE	This is simply a screen which waits until all the safety sensors are in the correct state and once it they do so, the user is notified that the sensors are in the correct state and is given the option to press the “un-park” button to enter the READY_STATE.
READY STATE	In this state, the software waits until the operator either enters the MANUAL CONTROL STATE or unparks the chair and consequently goes to the IDLE STATE
MANUAL CONTROL STATE	In this state, the operator is allowed to manually control the flight motion platform through the control software. The operator is able to control the platform’s: x position, y position, z position, roll, pith and yaw through the use of GUI sliders. The operator also has the capability of resetting the platform’s pose back to its normal state.
RUNNING STATE	During this state, the software takes in the plane telemetry data from the simulator (UDP message). The message contains the plane’s linear and angular acceleration (6DOF). Then the software applies the gains the operator set through the GUI, washout and any other post-processing before sending the output as actuator pressures through a UDP message.
STOP STATE	STOP_STATE is a state that waits until all the safety sensors are in their safe state and the user presses a GUI button before transferring back to the IDLE_STATE. During this state, the muscle pressures remain in as they last were in any of the previous states (or other “safe state”). When transition to IDLE_STATE happens, chair is put back on the parked position.

The flowcharts for each of the states are included in the appendix chapter of this report. These are Appendix 1, Appendix 2, Appendix 3, Appendix 4 and Appendix 5. These further outline the high-level logic of these states and what should occur when the user enters them.

A key detail to note how the safety sensors in this application are a requirement that has been taking into consideration for the software’s design. For example, the state of the safety sensors dictates whether the software can begin controlling the flight motion platform or not. If the sensors enter their unsafe state during the RUNNING state, this means that the char must’ve come undone and therefore the flight motion platform movement halts as the software transitions to the STOP state. The safe state signal through the code will be simplified as a Boolean input.

4.2. Network diagram of software components (nodes):

The software does not consist of just one python script running some logic. It consists of many python scripts each running separate from each other. Each concurrently running script will be referred to as a "node". These nodes are connected together by UDP protocol message streams which relay information to each other as needed to make the control software function as intended. The node diagram for the entire software is shown in Figure 4.2.

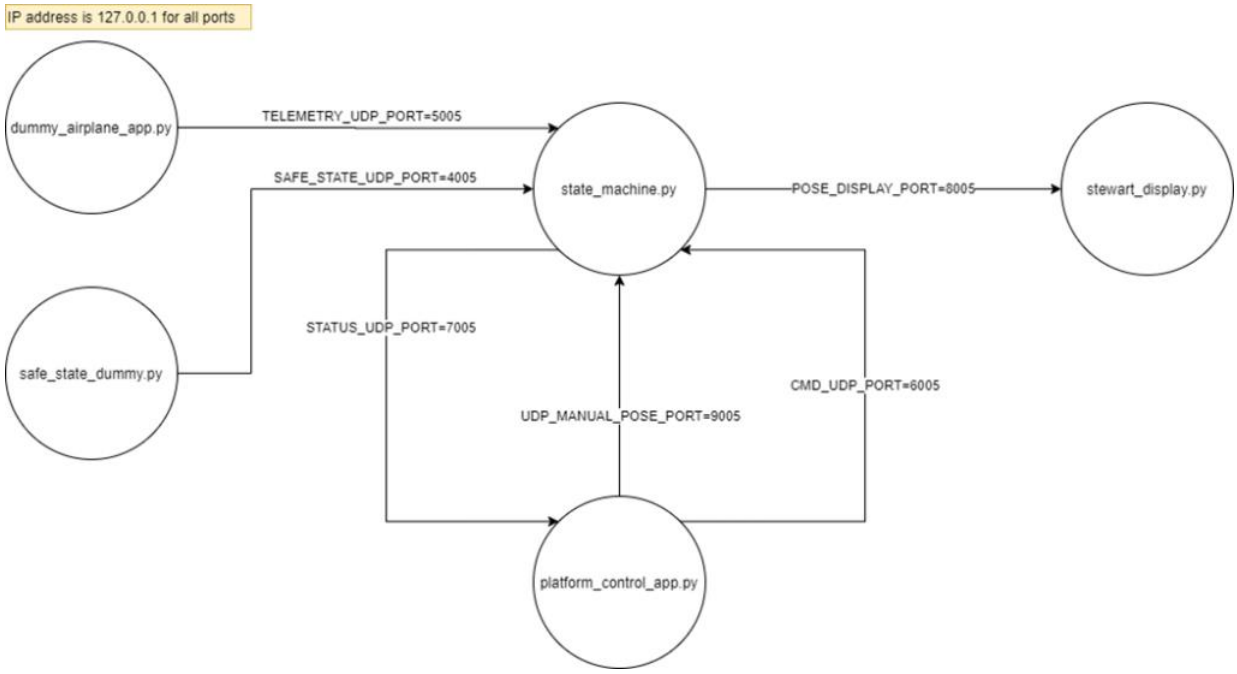


Figure 4.2-Node diagram of entire software.

This modular designed allowed for each script to maintain one single key purpose and made overall design and maintainability much better. The "state_machine.py" script is the one running all the state machine logic mentioned in the previous subchapter.

Meanwhile "platform_control_app.py" is the GUI interface through which the user controls "state_machine.py". "dummy_airplane_app.py" and "safe_state_dummy" are two nodes which serve as temporary placeholders for the airplane's input telemetry and flight motion platform's safe state. Making these two placeholders ensures the most can be done whilst fundamental components of the project are yet to be made.

"stewart_display.py" is a display application (python program) which takes in a pose message and displays a generic stewart platform display. This utility was reused from a GitHub repo [21]. Implementing this utility ensured there was a way to validate the incoming data and ensuring the output functioned correctly.

Table 4.2 and Table 4.3 further outline the function of each node in the network as well as what each connection within the network is meant to do.

Table 4.2- Table outlining function of each node in software network.

Node name	Function
dummy_airplane_app.py	This is dummy node which acts as a placeholder for the flight sim. Operator can modify the airplane's 3 axis acceleration (or position) and 3 axis rotation through a set of sliders. The data is sent as an array via "TELEMETRY_UDP_PORT" where the array is [xaccel, yaccel, zaccel roll_rate, pitch_rate, yaw_rate]. Each pose value is shown as a slider. Accel values only affect the chair's position (that's why it's labelled as xpos, ypos and zpos in GUI).
safe_state_dummy.py	Just like "dummy_airplane_app.py" this is a dummy placeholder application which sends a Boolean message through "SAFE_STATE_UDP_PORT". This Boolean message denotes whether the flight motion platform is in its safe state or not. This Boolean appears as a checkbox in the GUI.
state_machine.py	This script implements the state machine described in chapter 4.1 This state machine executes at a variable cycle rate set to 20ms at the moment. This machine can be fully controlled by sending commands through "CMD_UDP_PORT" hence it does not need to handle any of the GUI. Each state the machine can be in (IDLE, READY, RUNNING, MANUAL, and STOP). At the moment, the UDP command format is ["state_transition", gain_xaccel, gain_yaccel, gain_zaccel, gain_roll, gain_pitch, gain_yaw]. The state transition asks the state machine where to transition to and the rest are gain variables set by the operator or user to define how aggressive the ride should be. This machine also outputs its status through "STATUS_UDP_PORT" as [current_state, isSafe] where "current_state" denotes the machine's current state and "isSafe" is the safety sensor Boolean which determines whether any of the chair's safety sensors were tripped. This allows the GUI of "platform_control_app.py" to display the state machine's state accordingly and change any elements as required. Finally, this state machine node listens to the airplane's telemetry from the dummy_airplane_app.py node and uses it as previously described.
platform_control_app.py	This app in in charge of reading the state machine's status and allowing the user to send control commands to state_machine.py node through a GUI. The STATUS_UDP_PORT is used to display the machine's status and indicate whether the operator can do certain actions. The MANUAL_POSE_UDP_PORT is used to send the manual pose values from their respective sliders to "state_machine.py".

stewart_display.py	This is a display script from the "Stewart_Py" library [21]. Since the chair is an example of an inverted Stewart platform, this script provides an effective temporary way to visualize the chair's motion. This will allow to save time whilst gaining project progress while the chair is still under construction. Many aspects of the project were not finished as they were being worked with in parallel. This is the usual underside of agile methodology.

Table 4.3-Table outlining function of each network UDP port.

Node name	Port	Function
SAFE_STATE_UDP_PORT	4005	This port sends the safe state Boolean from the "safe_state_dummy.py". This port is listened by "state_machine.py".
TELEMETRY_UDP_PORT	5005	This port sends airplane telemetry from either flight sim or dummy as "[xaccel, yaccel, zaccel roll, pitch, yaw, isSafe]". The port is listened by "state_machine.py"
STATUS_UDP_PORT	7005	This port sends out state_machine.py's status to "platform_control_app.py" as "[current_state, isSafe]".
CMD_UDP_PORT	6005	This port sends control commands from "flight_control_app.py" to "state_machine.py" as [state_transition, gain_xaccel, gain_yaccel, gain_zaccel, gain_roll, gain_pitch, gain_yaw]. State transition is the state the operator wishes to transition to.
POSE_DISPLAY_PORT	8005	This port sends the output from "state_machine.py" RUNNING state. This output is a product of the plane telemetry's pose variables with each respective gain variable and the master gain. For example, if x_val is the x acceleration sent to "stewart_display.py", x_val = x_accel*x_accel_gain*master_gain. This allows the operator to control the sensitivity of the chair's movements and therefore the ride's sensitivity.
MANUAL_POSE_UDP_PORT	9005	This port sends out the pose values from the platform_control_app.py manual control sliders. These are sent as an array of [gain_xaccel, gain_yaccel, gain_zaccel, gain_roll, gain_pitch, gain_yaw].

Progress evidence of this entire chapter alongside videos is detailed in the project's blog [5] Week 2 and Week 3 post.

Final code snippets and GUI snippets (as of Week 11) of all the code nodes mentioned in this chapter are shown in the appendix section. These are all the appendixes from Appendix 6 to Appendix 14.

4.3. Integrating the system with X-Plane

Once a basic skeletal system for the hardware was established, the next step was to replace the "dummy_airplane_app.py" with a program that sent X-Plane's airplane telemetry to the same port in the same format as of now. In order to do this, a plugin for X-Plane was written. X-Plane supports plugins written in C and Delphi.

However, there's a popular third-party plugin called XPPython [22] which acts as an X-Plane plugin that executes several python plugins that can be easily inserted. For all X-Plane is concerned, XPPython is a plugin that runs sub-plugins. This is further illustrated in Figure 4.3 or resource [23]. This is a great solution since it allows to keep all the project code written in python.

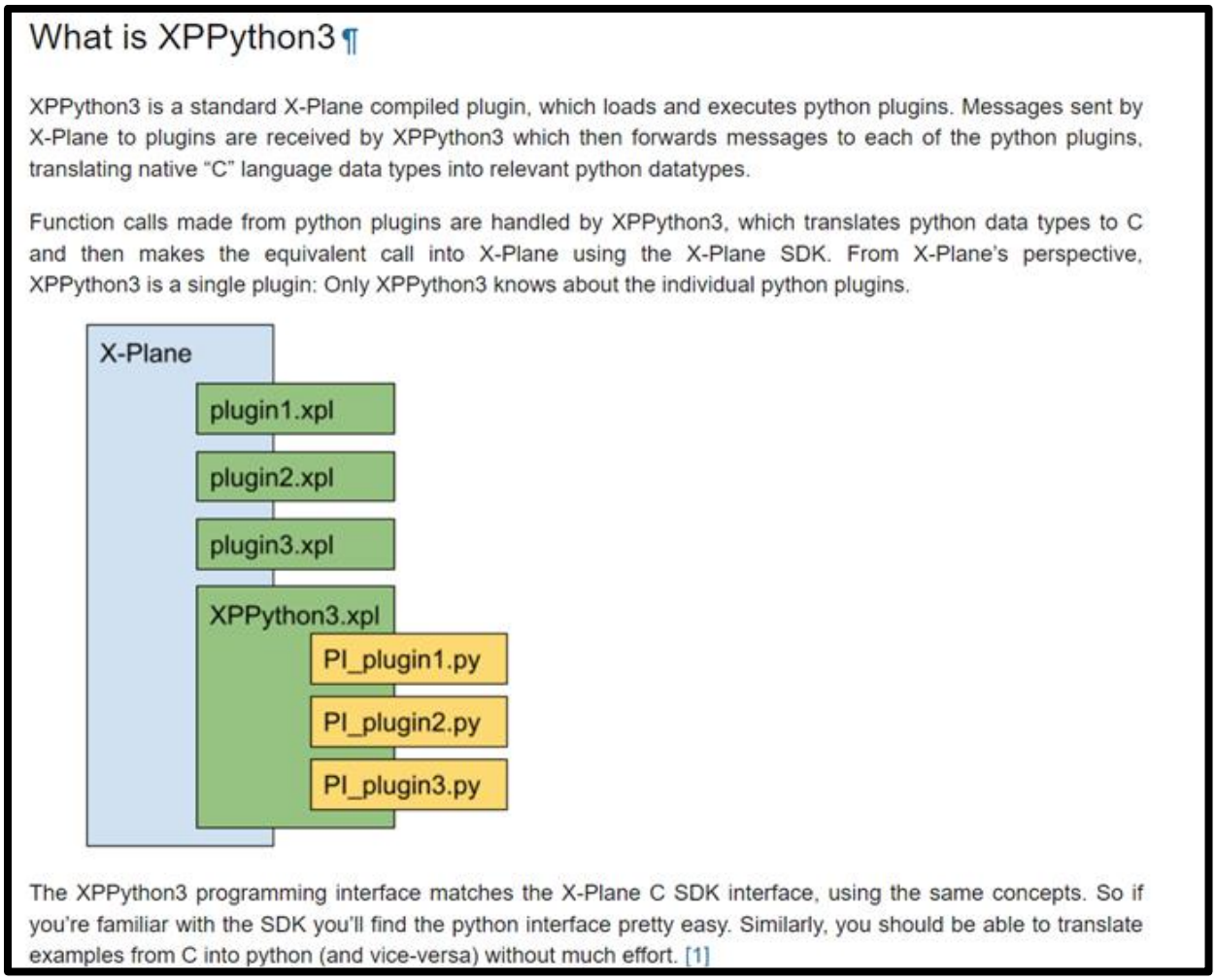


Figure 4.3-Diagram showing how XPPython works.

The code for the telemetry XPPython script that sends the airplane's telemetry when X-Plane runs is shown in Appendix 15. The name of the script is "PI_simple_telemetry.py".

Once this program was set up in X-Plane, the software defined in the previous subchapter was able to receive the plane's telemetry data and plot it through the stewart_display.py display. This progress is detailed in the project blog's [5] Week 4 post.

5. Improving and implementing flight motion platform virtual displays

5.1. Initial outline of new display

The next step, after developing a functional foundational software system capable of successfully receiving input data from X-Plane and outputting it to a useful destination, was to enhance the visualization tool. The existing “stewart.py” display was overly simplistic and did not accurately represent the final design.

The new display is a Unity-based application that renders a moving 3D CAD model, similar to the one shown in Figure 3.1. Functionally, it operates like the “stewart_display.py” application but offers a more advanced way to evaluate the software’s output beyond simply confirming that it works. This tool provides a clearer visualization of platform movement, helping to identify issues such as abrupt motion, potential collisions between platform sections, and interactions with the ground. A brief final screenshot of the application is shown in Figure 5.1.

Figure 5.1-Screen snip of Unity visualization application.

The remainder of the chapter will outline the process of developing this visualizer application as well as reference any appendix resources or blog posts. It’s also worth noting that the Unity visualiser application is a retooled version of a pre-existing Unity visualizer from a previous project.

5.2. Importing CAD model into Unity

The first step towards making the visualizer was to prepare the model for Unity by exporting it into the correct format as well as optimizing it. Figure 5.2 showcases the process taken to obtain the flight motion platform CAD models from SOLIDWORKS to Unity.

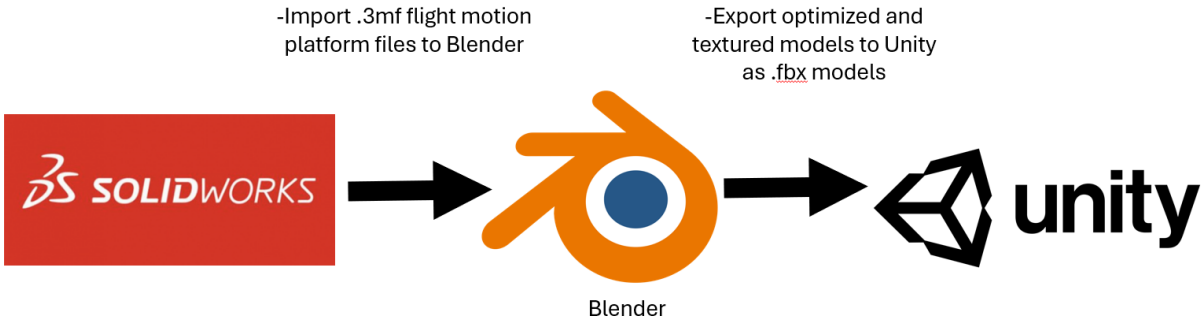


Figure 5.2-Ouline of workflow taken to import the CAD models of the flight motion platform from SOLIDWORKS to Unity

The models had to be imported and processed in Blender for three main reasons. First, the SOLIDWORKS models were in .3mf format, which is not natively supported by Unity. Second, the models lacked textures, resulting in plain grey surfaces instead of coloured ones. Most importantly, the SOLIDWORKS models had an extremely high triangle count of around 2 million, which required significant time and effort to optimize during the model processing stage. A before and after illustration is shown in Figure 5.3.

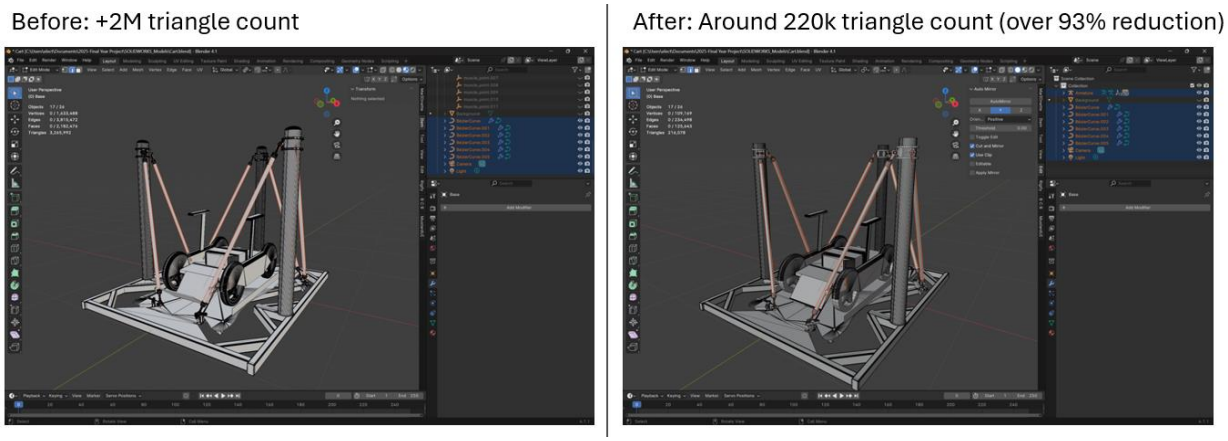


Figure 5.3-Before and after optimizing model geometry in Blender to export to Unity.

The model was optimized by either removing excessively detailed, high-density geometry or simplifying complex components, such as the wheels, with lower-polygon alternatives that maintained a visually identical appearance.

As an additional bonus (and to help overall visualization) a Blender rig was created which allows Blender to move the flight motion platform. Paired with Blender's effect animation and rendering capabilities this resulted in a great rig that could was used to make a brief proof of concept animation. This referenced in resource [24]. The rig is further outlined in Figure 5.4. Resource [25] has further information of Blender rigs and their general use outside this project.

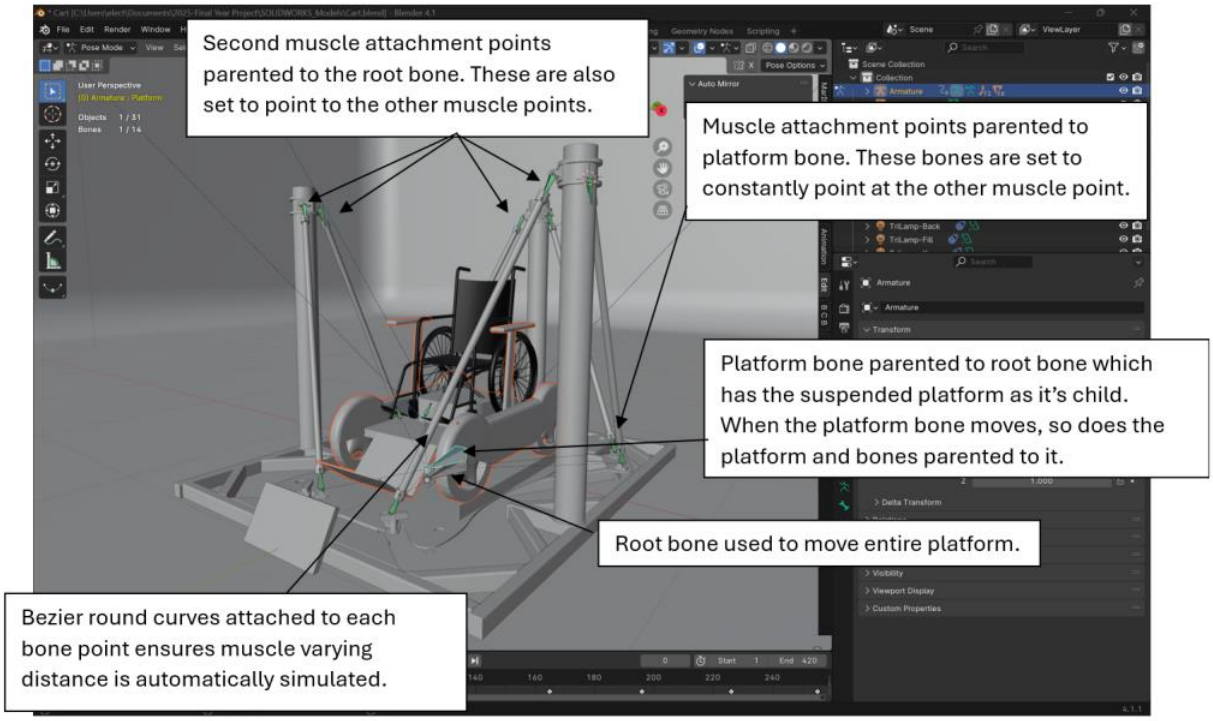


Figure 5.4-Figure showcasing the rig used to simulate the moving platform.

The wheelchair model sitting on the cart that is meant to be wheeled into the flight motion platform was modelled by Maxence Rouillet who created this model [26]. The model is under the "CC attribution" license.

This rig did not affect anything within the model when exporting it to Unity. Once the geometry was optimized and some simple colour textures were assigned to the model, the model was able to be imported into the Unity application.

5.3. Re-Tooling existing code and exporting built application.

Re-tooling the existing code involved importing the flight motion platform model into the scene and then renaming some of its key components to the names the previous chair had. The reason for this was because the script in charge of listening for UDP messages refers to flight motion parts by name. Once that was done the application worked as intended and the flight motion platform could be properly visualized. The UDP output port in the main software was changed to 10020.

Something this visualizer tool shed light on was how the final motion platform would collide against the ground if not properly raised. At the moment the software does not account for this feature and will likely be left to future work due to the final flight motion platform with wheelchair access not being fully built by the end of the project's timespan.

The progress and working evidence of this display application is shown in the project blog's [5] Week 5 and Week 6 posts. Week 5 details just using the original rollercoaster visualization tool, but Week 6 showcases the visualization tool with the final flight motion platform and the wheelchair attached to it.

The project source files are referenced in both this project's git repository and a publicly accessible google drive folder [27]. The telemetry script code snippets in charge of moving the motion platform within the Unity visualizer/display app is shown in Appendix 16.

6. Design, testing and evaluation of fluidic muscle calibration procedures.

6.1. Designing muscle calibration procedures

Once a solid underlying control system was established for the project, the next major challenge was determining the pressure-to-distance characteristics of the new, longer muscles that the flight motion platform will use.

As a recap from the introduction and literature review section of this report, this project employs longer and wider fluidic muscles in an open-loop scheme. In this setup, the only way the system can control each muscle's contraction distance is by setting its pressure to a predefined value. To determine the appropriate pressures required to achieve a specific muscle length, the system must first "calibrate" the muscle it will use. This calibration process involves applying a range of pressures, from minimum to maximum, and measuring the contraction distance at each step.

A fluidic muscle's pressure to distance characteristics different when relaxing (down) compared to when contracting (up), which forms a hysteresis loop. This loop is shown in Figure 6.1.

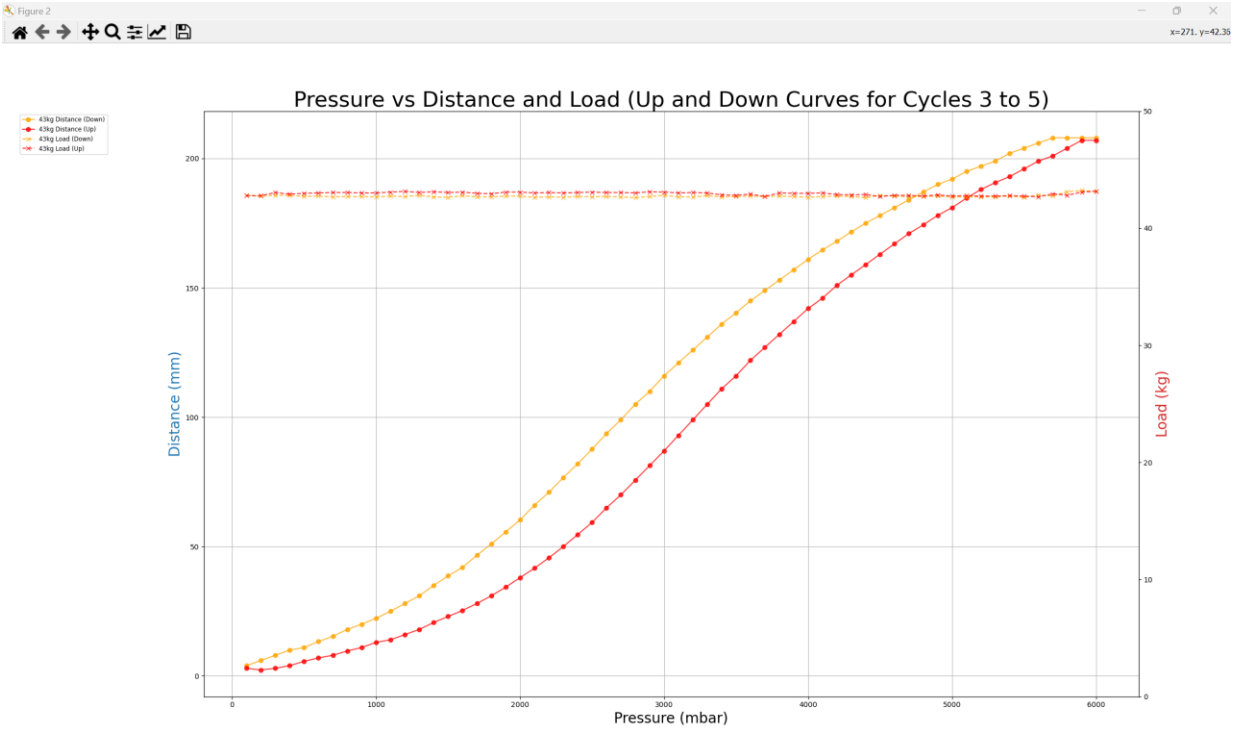


Figure 6.1-Example chart of muscle hysteresis loop.

As the figure shows, the muscle forms hysteresis loop (round full dots) which is formed when the muscle contracts from its maximum pressure (6000mbar) and relaxes to its minimum pressure (0mbar). The cross and dotted lines are simply the load across the muscle plots. These are included to validate that the load across the muscle is constant and does not vary too much. Obtaining this data will be referred to throughout the chapter as muscle calibration. Calibration data was obtained for loads weighting 13kg, 24kg, 33kg and 43kg.

The calibration data was obtained with the aid of a vertically mounted calibration rig just like the one shown in Figure 6.2.

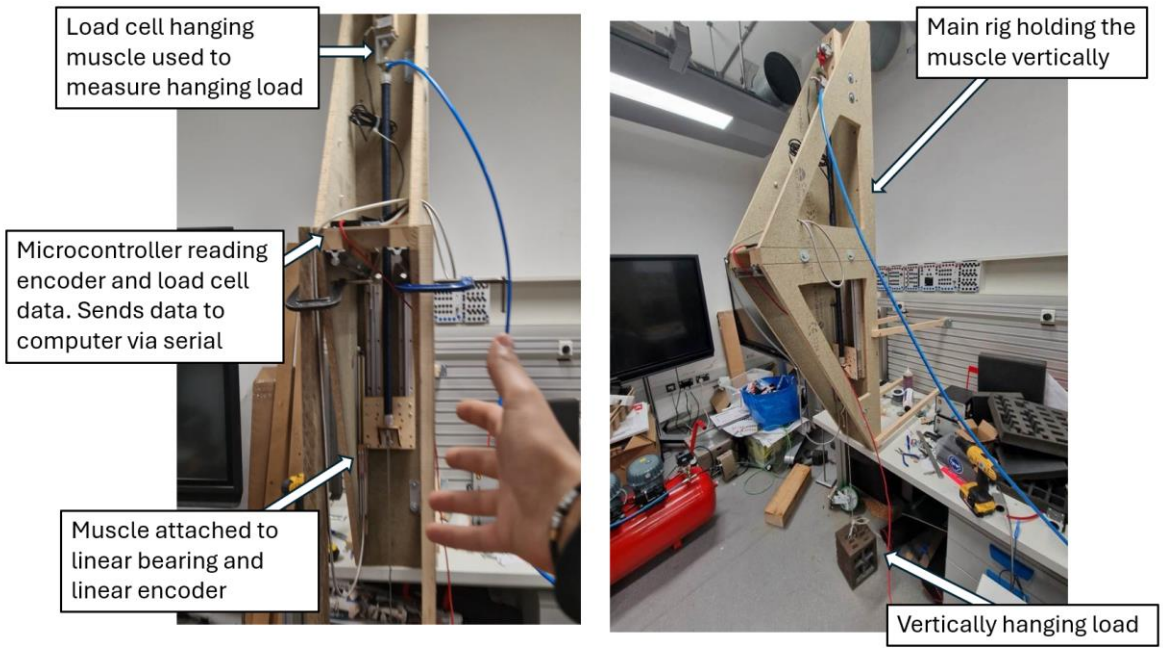


Figure 6.2-Ilustration of calibration rig used to obtain muscle pressure-to-distance data.

The core principle behind this rig was to hang a load acting collinearly with the muscle. The load's weight would be measured and validated across the calibration procedure. During the calibration procedure, a program running on a computer would send pressure commands to the muscle via UDP and record the distance and load values from the rig's serial messages at each pressure step. The high-level outline of the system is shown in Figure 6.3.

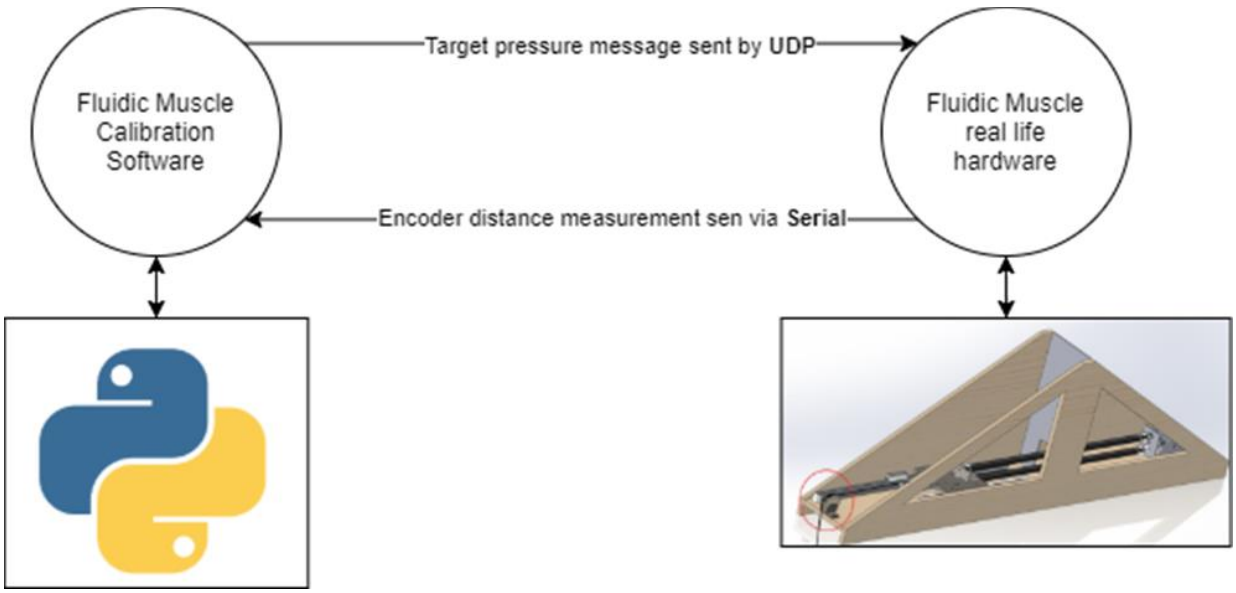


Figure 6.3-Ilustration of how calibration program and test rig will communicate.

Something worth noting is that the control software required to control the test muscle was already included in the rollercoaster software and worked as intended. Writing a python program to systematically log the muscle's pressure to distance characteristics for a given load allowed to obtain pressure distance data much more accurately and faster. The logic the calibration software executed is outlined in Figure 6.4.

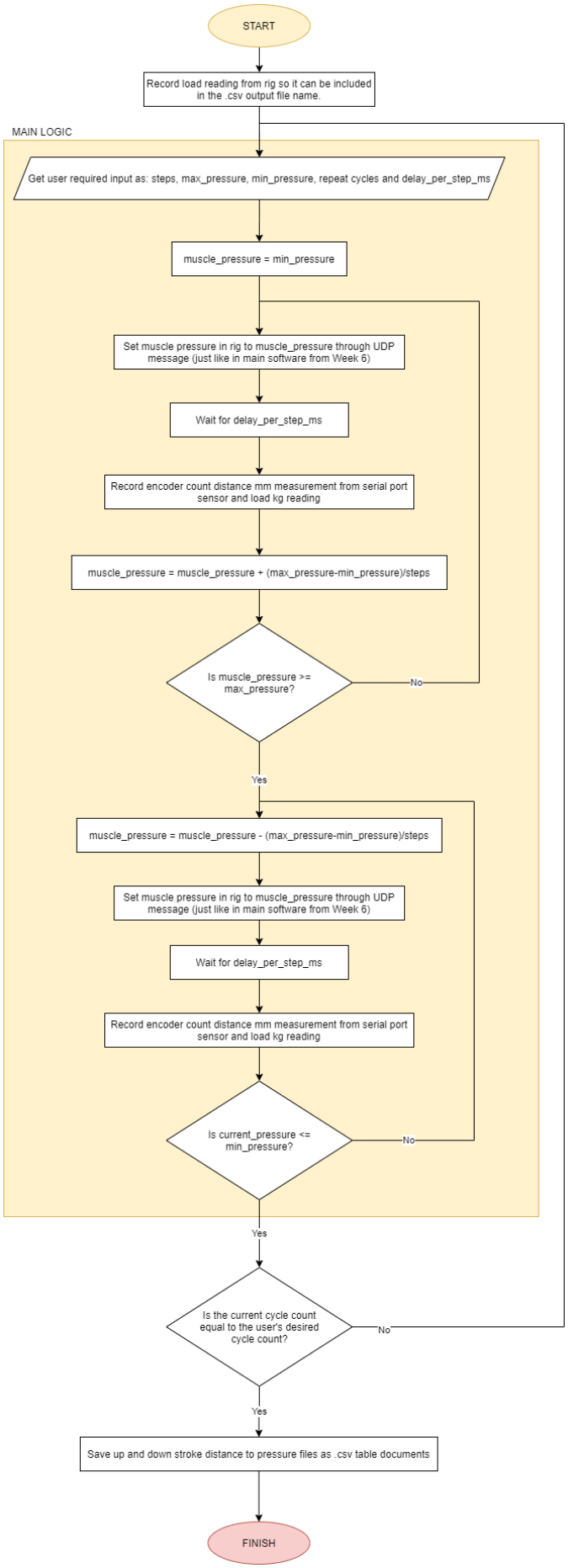


Figure 6.4-Flowchart outlining the working logic of the calibration software.

Before moving onto the implementation and testing of the rig, it's worth noting that development weeks 7 until the end of week 9 were spent trouble shooting iterations of the test rig as well as arduously evaluating output data to ensure that the most accurate results were being obtained. These iterations alongside project results were thoroughly outlined in the project blog [5] Week 7 to Week 9 posts.

This report will only cover and evaluate the procedure and test results of the final rig iteration.

6.2. Implementing and testing muscle calibration procedures

The script in charge of obtaining the pressure to distance calibration data of the muscle at different loads was called "calibration_data.py". The GUI and code snippets for this software are included in Appendix 17 and Appendix 18 respectively. Likewise the GUI and code snippets for the "csv_muscle_plotter.py" plotter utility which was used to make the plots shown in this report are included in Appendix 19 and Appendix 20.

The final calibration data tables were obtained when running a test with the following parameters:

-6 seconds delay per pressure step

-60 steps

-5 calibration cycles

The following figures are the final pressure (mbar), distance (mm) and load (kg) plots for 13kg, 24kg, 33kg and 43kg. In these plots, the different coloured lines represents either an up stroke or a down stroke where the muscle is either relaxing or contracting. The darker coloured lines represent later calibration cycles. For example, down cycle 1 is the lightest blue and down cycle 5 is the darkest blue.

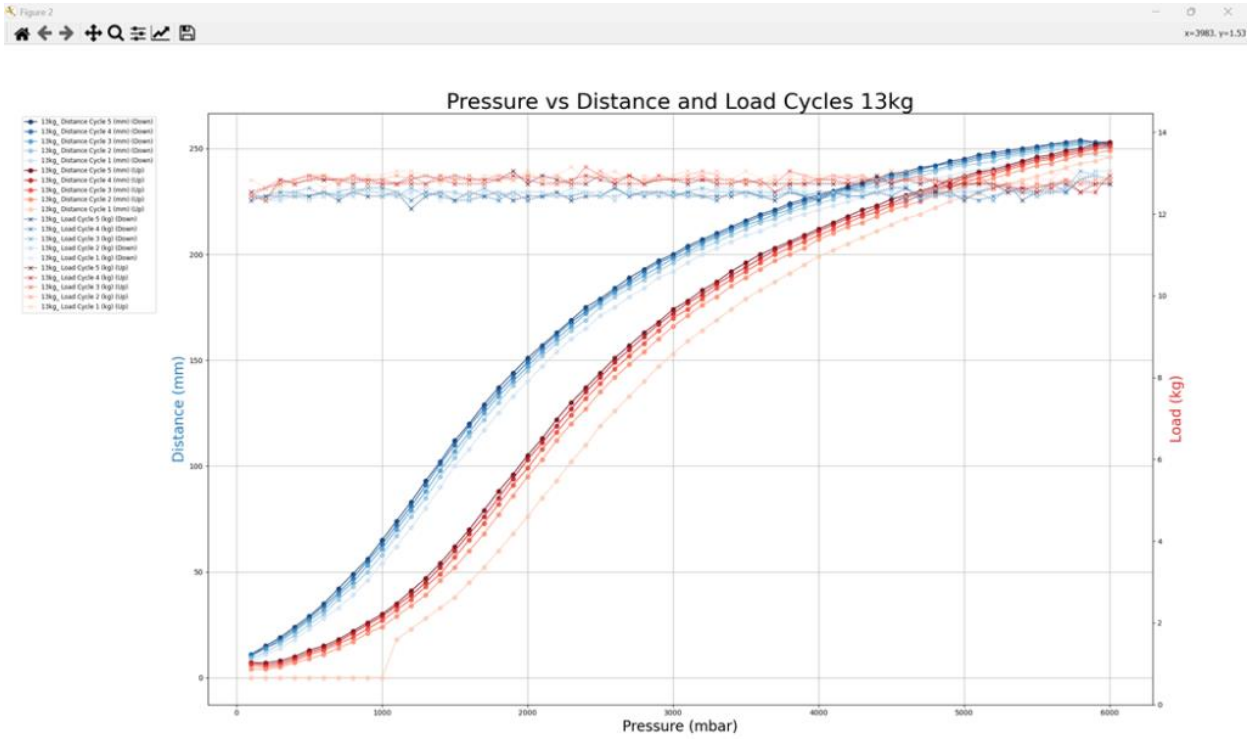


Figure 6.5-Pressure (mbar) vs Distance (mm) plot for 13kg load

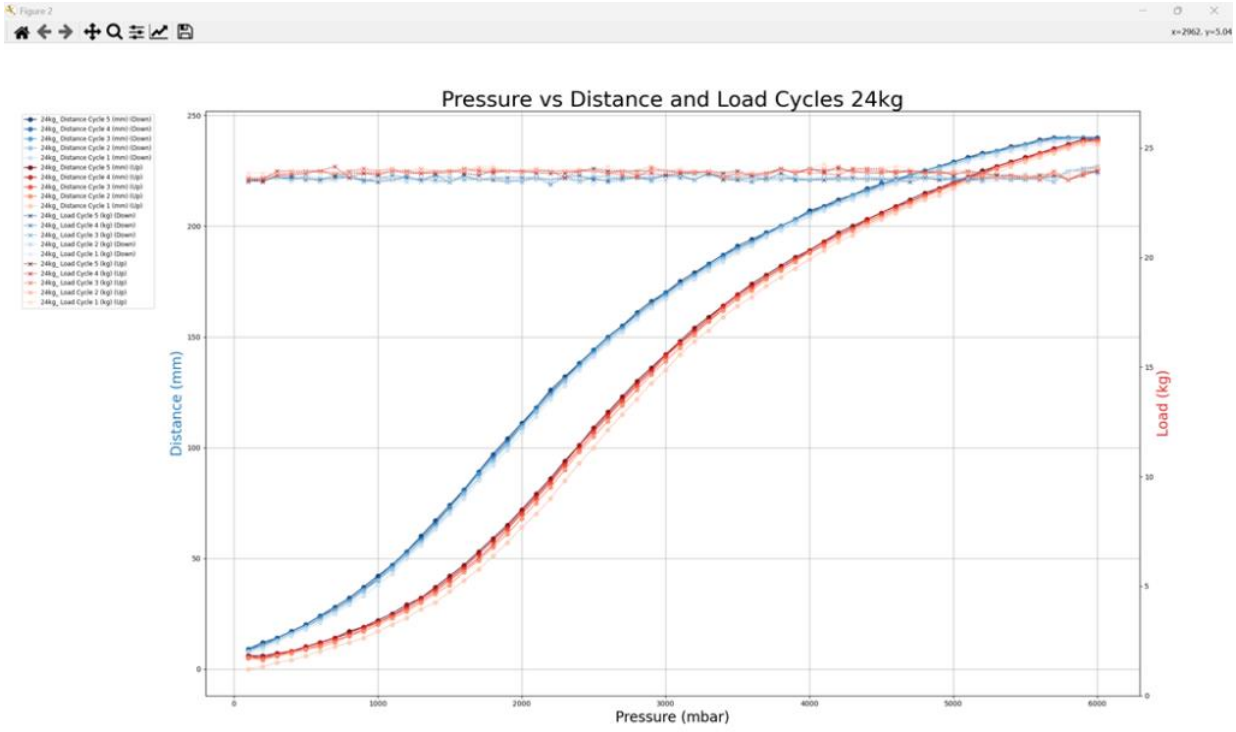


Figure 6.6-Pressure (mbar) vs Distance (mm) plot for 24kg load.

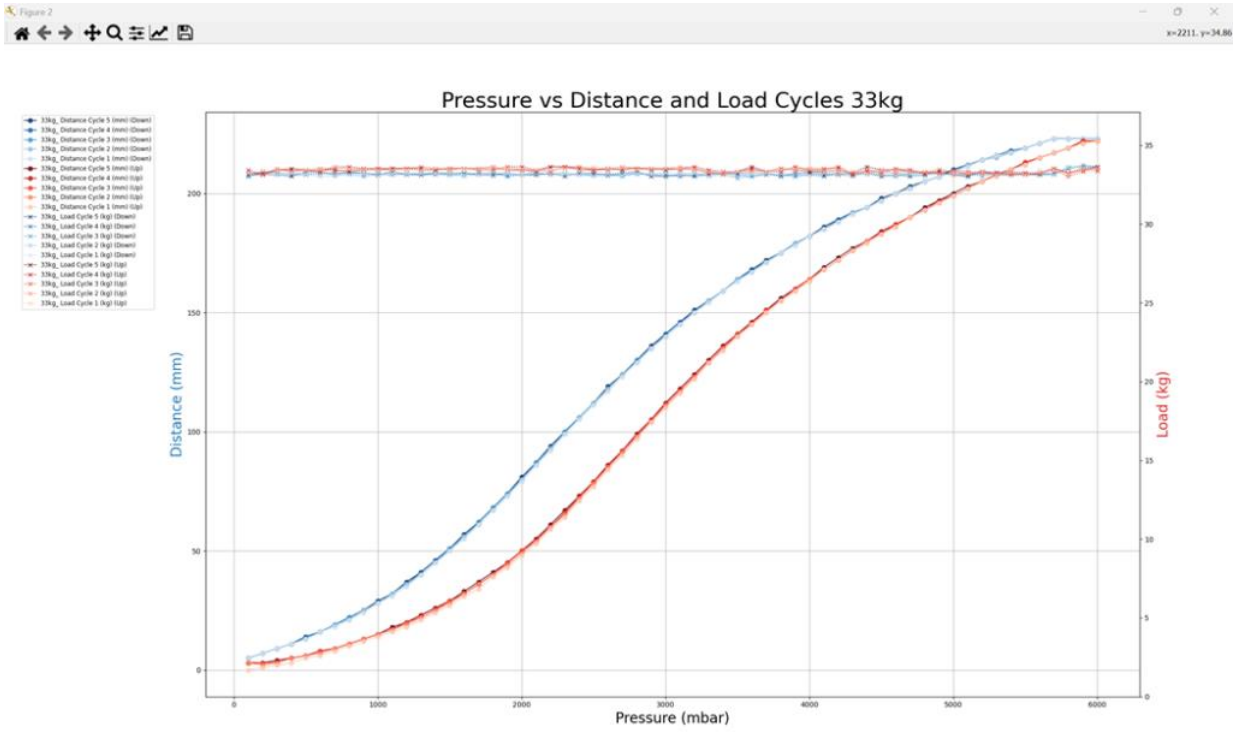


Figure 6.7-Pressure (mbar) vs Distance (mm) plot for 33kg load.

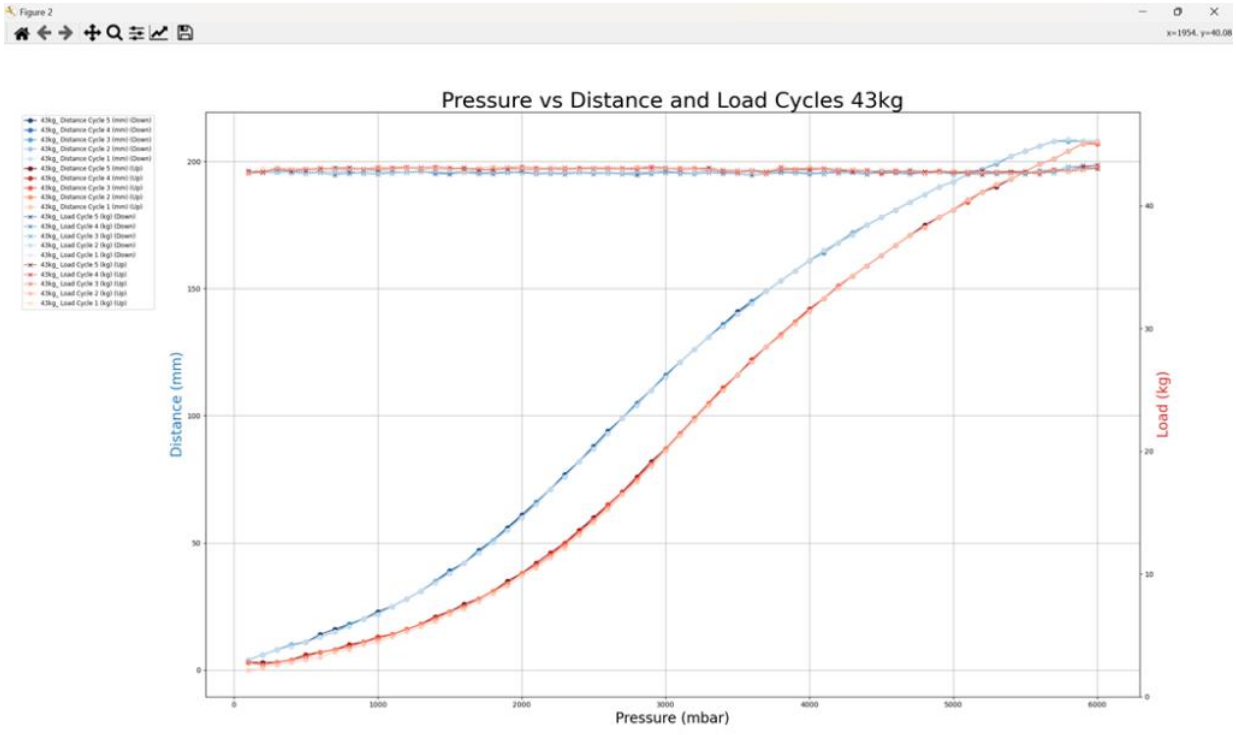
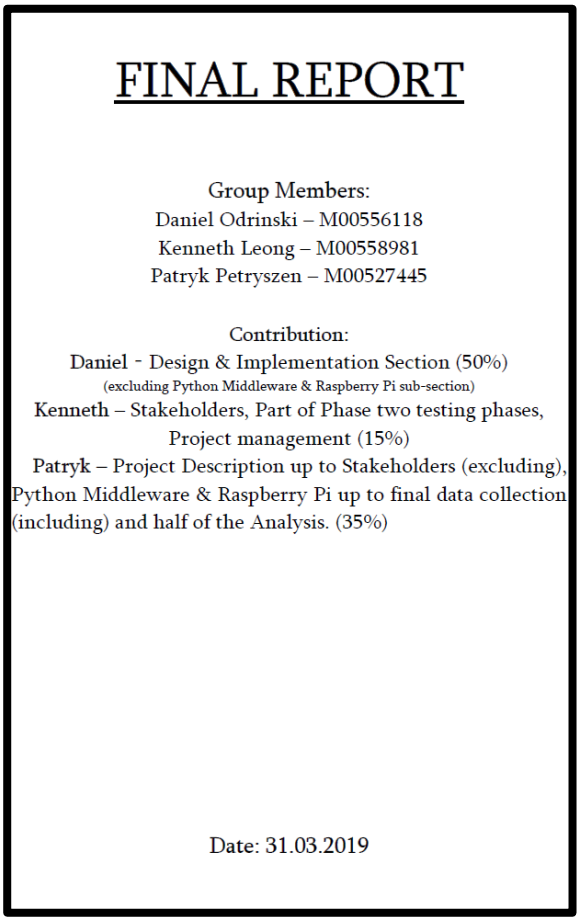


Figure 6.8-Pressure (mbar) vs Distance (mm) plot for 43kg load.

6.3. Evaluating test result data and obtaining conclusions.

When comparing this project's muscle calibration procedure to some student's previous work in the rollercoaster ride which used light sensor range finders on the actual rollercoaster (no rig) to calibrate the muscles took longer and used a far noisier source of distance measurement. The students in charge of the testing were *Daniel Odrinski, Kenneth Leong* and *Patryk Petryszen* who were in charge of designing the python middleware and calibration software for the rollercoaster ride. Front page of their report is shown in Figure 6.9.



FINAL REPORT

Group Members:
Daniel Odrinski – M00556118
Kenneth Leong – M00558981
Patryk Petryszen – M00527445

Contribution:
Daniel - Design & Implementation Section (50%)
(excluding Python Middleware & Raspberry Pi sub-section)
Kenneth – Stakeholders, Part of Phase two testing phases, Project management (15%)
Patryk – Project Description up to Stakeholders (excluding), Python Middleware & Raspberry Pi up to final data collection (including) and half of the Analysis. (35%)

Date: 31.03.2019

Figure 6.9-Front cover of previous student paper.

Figure 6.10 showcases the reading distance difference range between both the student laser finder rig and my rig. A higher range means higher reading noise and lower overall accuracy and precision. This project's software takes 100 distance reading snapshots per step.

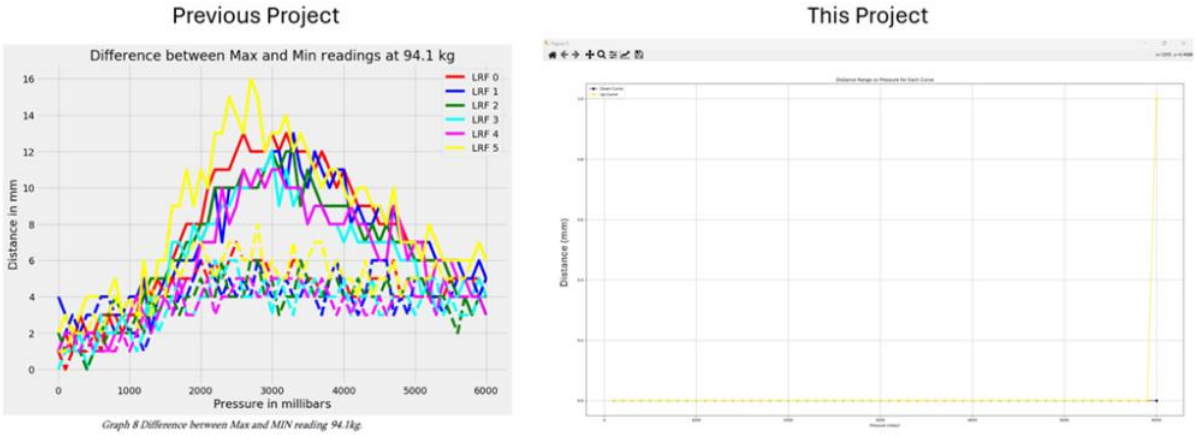


Figure 6.10-Comparison between previous project distance reading range per pressure and this project's.

As Figure 6.10 showcases, using an encoder offers practically no noise in distance measurements, therefore this project offers a less uncertain calibration procedure. Even though there's a 1mm outlier spike at 6000mbar, this still means that the encoder calibration method is far more accurate than the student's previous laser rangefinder method.

Moreover, in the described paper, the students were measuring each muscle's contraction distance directly on the motion platform itself instead of using a dedicated rig which is designed to keep the load against the muscle constant and thereby as accurate as possible. In the paper they also reported many problems with the sand used as a load shifting as the platform tilted and thereby causing load inconsistencies. That section of the paper is shown in Figure 6.11.

Conclusion

Overall the distribution of the weight on the chair affects the data measurements significantly, and and inclinometer is recommended use to minimise the risk of wrong weight distribution on the chair. Verification of the motion platform's actuators pressure capabilities and stretching ability is a very important aspect for the final project results. Lastly research more about the motion platforms functionality and increase the cycles if necessary. More data will be benefical for more accurate outcome of the project. During the project the team members have been using look up table to identify the distance for each pressure per weight.

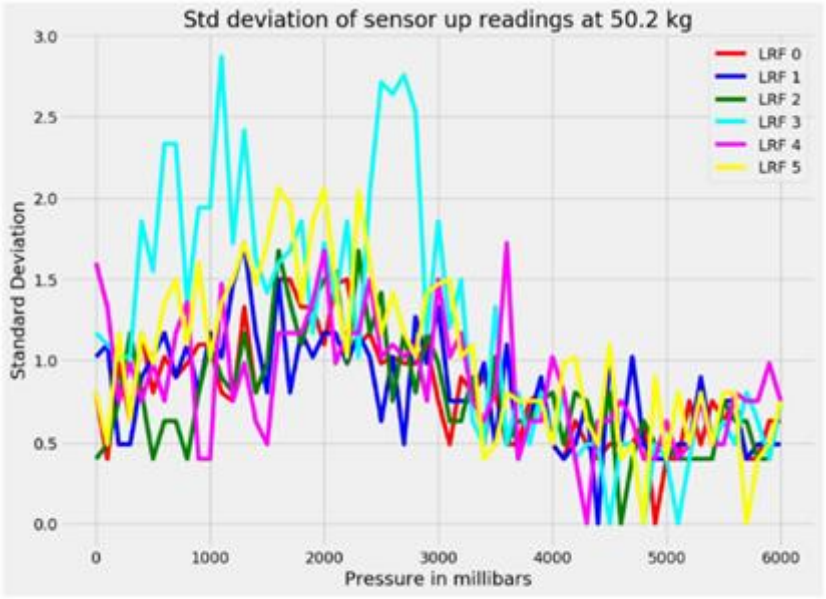
Figure 6.11-Conclusion section of previous student's project on the rollercoaster chair.

Another important point to raise directly from the calibration results in chapter 6.2 was how the first few calibration cycles appear to systematically "shift down" from the subsequent cycles. This effect became more apparent at lighter loads. A likely cause for the muscle at lighter loads converging towards an equilibrium curve is probably due to the light load not exhausting all the air pressure properly from the inner muscle tube. Therefore, when the subsequent cycles are performed, the muscle has slightly more air inside than before and therefore been able to gradually contract more each cycle, thereby gradually shifting the load curves up. Due to this shifting effect, the first 2 cycles for every calibration load were excluded from calculations.

The charts showing the standard deviation between 5 cycles are located from Appendix 21 to Appendix 24. In the worst-case scenario, which is the 13kg load, excluding the first two outlier cycles results in a 2.5mm standard deviation and 6mm range.

Just like in the previous figures, range and standard deviation appear to decrease as the load becomes heavier hence the motions platform muscles will become less accurate with lighter loads.

Even then, final calibration results prove to be much more accurate than and precise than the previous project's calibration results. The worst range and standard deviation for this set of calibration results are 2.5mm standard deviation and 6mm range. By comparison. The highest standard deviation from the previous rollercoaster project at 50.2kg is 2.9mm as shown in Figure 6.12. The worst standard deviation in this project at 43kg is only 0.5mm which is an over 82% decrease in test data standard deviation at a lighter load (which should be less accurate).



Graph 10 STD deviation sensor up reading 50.2kg.

Figure 6.12-Standard deviation of sensor up readings for previous student MDX rollercoaster project.

Therefore, the muscle's worst accuracy at 13kg is +-3mm or a 2.38% uncertainty range (when accounting for the 252mm contraction range at 13kg). Therefore, the project's final calibration results are not only accurate enough for the motion platform but also more accurate than previous work done in this area.

As a final showcase, Figure 6.13 serves as a rough final illustration of all the curves put together in the same graph. This allows to observe a noticeable trend where the heavier the load, the more "squished" the load loops become and therefore the higher the load, the lower the final extension. The highest the muscle can contract at a 43kg load is 208mm whereas at 13 kg it can contract to up to 252mm (21% increase in contraction distance).

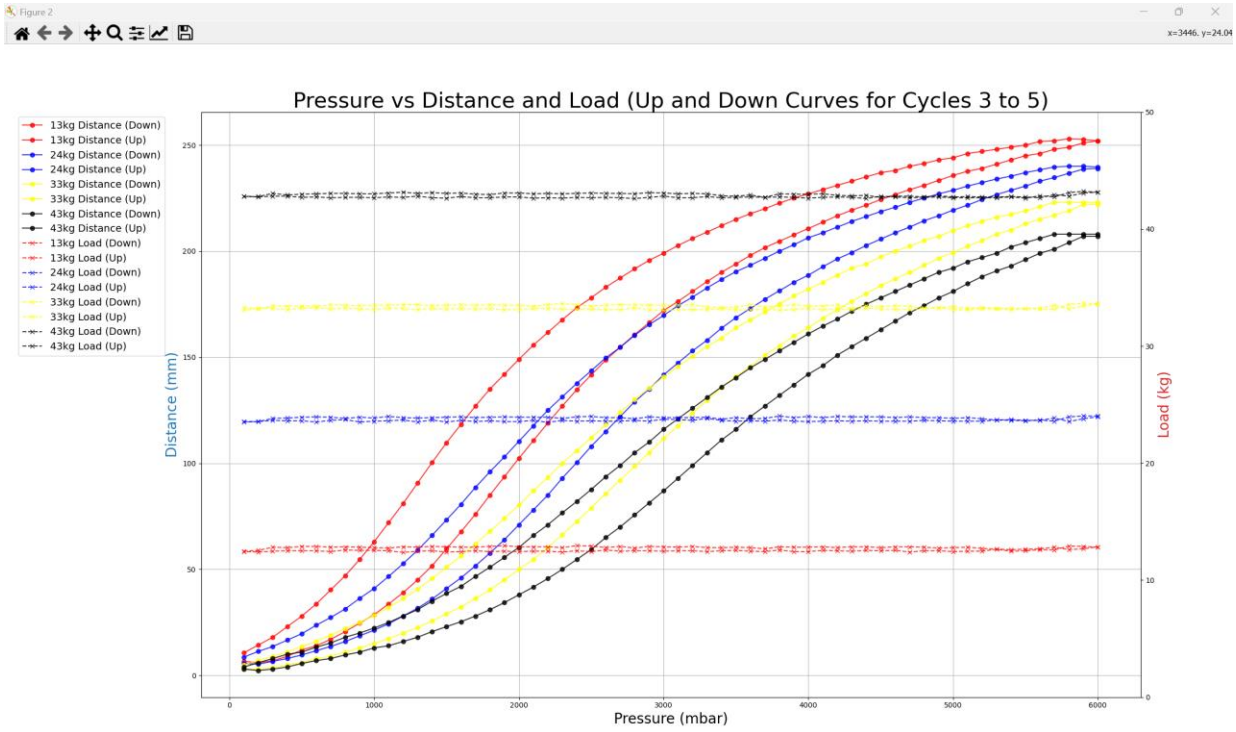


Figure 6.13-Final averaged distance and load cycle data plotted simultaneously on a graph. Cycles 3, 4 and 5 were counted and the first two cycles of each load calibration set was discarded.

7. Integration of muscle calibration data back into control software

7.1. Converting calibration cycle data into cycle averaged data

Once the calibration data for the muscle was obtained and validated the next step was to convert the pressure to distance (PtoD) tables into distance to pressure (DtoP) tables that could be integrated into the main software. This involved writing several utility python tools to convert the tables into the specific desired format that the rollercoaster codebase needed.

Figure 7.1 shows a snippet of the structure each load curve calibration csv file had. Note that figures only show part of the data and not all of it.

Pressure (mbar)	Distance Cycle 1 (mm)	Distance Cycle 2 (mm)	Distance Cycle 3 (mm)	Distance Cycle 4 (mm)	Distance Cycle 5 (mm)	Load Cycle 1 (kg)	Load Cycle 2 (kg)	Load Cycle 3 (kg)	Load Cycle 4 (kg)	Load Cycle 5 (kg)
6000	208	208	208	208	208	43.01732925586136	43.11926605504587	43.01732925586136	43.32313965341488	43.11926605504587
5900	208	208	208	208	208	43.01732925586136	43.01732925586136	43.32313965341488	43.22120285423038	43.11926605504587
5800	209	208	208	208	208	43.11926605504587	43.22120285423038	43.01732925586136	43.22120285423038	43.11926605504587
5700	208	208	208	208	208	42.60958205912334	42.813455657492355	42.71151885830785	42.813455657492355	42.71151885830785
5600	206	206	206	206	206	42.813455657492355	42.60958205912334	42.813455657492355	42.915392456676855	42.813455657492355
5500	204	204	204	204	204	42.71151885830785	42.813455657492355	42.60958205912334	42.60958205912334	42.71151885830785
5400	202	202	202	202	202	42.60958205912334	42.71151885830785	42.813455657492355	42.60958205912334	42.813455657492355
5300	200	199	199	199	199	42.71151885830785	42.813455657492355	42.71151885830785	42.60958205912334	42.71151885830785
5200	197	197	197	197	197	42.813455657492355	42.71151885830785	42.50764525993883	42.71151885830785	42.813455657492355
5100	195	195	195	195	195	42.71151885830785	42.71151885830785	42.60958205912334	42.60958205912334	42.813455657492355
5000	192	192	192	192	192	42.60958205912334	42.50764525993883	42.71151885830785	42.71151885830785	42.60958205912334
4900	190	190	190	190	190	42.71151885830785	42.71151885830785	42.71151885830785	42.71151885830785	42.813455657492355
4800	187	187	187	187	187	42.60958205912334	42.71151885830785	42.813455657492355	42.60958205912334	42.71151885830785
4700	184	184	184	184	184	42.50764525993883	42.71151885830785	42.60958205912334	42.813455657492355	42.60958205912334
4600	181	181	181	181	181	42.60958205912334	42.71151885830785	42.71151885830785	42.813455657492355	42.71151885830785
4500	178	178	178	178	178	42.915392456676855	42.60958205912334	42.71151885830785	42.915392456676855	42.813455657492355
4400	175	175	175	175	175	42.813455657492355	42.71151885830785	42.60958205912334	42.60958205912334	42.60958205912334
4300	171	172	171	172	172	42.813455657492355	42.71151885830785	42.71151885830785	42.71151885830785	42.71151885830785
4200	168	168	168	168	168	42.60958205912334	42.71151885830785	42.60958205912334	42.915392456676855	42.71151885830785
4100	165	165	164	165	165	42.71151885830785	42.60958205912334	42.813455657492355	42.60958205912334	42.71151885830785
4000	161	161	161	161	161	42.71151885830785	42.813455657492355	42.60958205912334	42.71151885830785	42.60958205912334
3900	157	157	157	157	157	42.60958205912334	42.813455657492355	42.71151885830785	42.71151885830785	42.71151885830785
3800	153	153	153	153	153	42.813455657492355	42.813455657492355	42.71151885830785	42.813455657492355	42.71151885830785
3700	149	149	149	149	149	42.71151885830785	42.71151885830785	42.813455657492355	42.71151885830785	42.60958205912334
3600	144	144	145	145	145	42.813455657492355	42.71151885830785	42.915392456676855	42.813455657492355	42.50764525993883
3500	140	140	140	140	141	42.60958205912334	42.60958205912334	42.71151885830785	42.813455657492355	42.60958205912334
3400	135	135	136	136	136	42.813455657492355	42.71151885830785	42.60958205912334	42.71151885830785	42.71151885830785
3300	131	131	131	131	131	42.813455657492355	42.71151885830785	42.71151885830785	42.915392456676855	42.71151885830785
3200	126	126	126	126	126	42.71151885830785	42.60958205912334	42.71151885830785	42.71151885830785	42.60958205912334
3100	121	121	121	121	121	42.71151885830785	42.813455657492355	42.60958205912334	42.71151885830785	42.71151885830785
3000	115	115	116	116	116	42.60958205912334	42.813455657492355	42.915392456676855	42.813455657492355	42.71151885830785
2900	110	110	110	110	110	42.71151885830785	42.71151885830785	42.813455657492355	42.60958205912334	42.71151885830785
2800	104	104	105	105	105	42.813455657492355	42.813455657492355	42.60958205912334	42.71151885830785	42.50764525993883

Figure 7.1-Snippet of table raw data from calibration results. This file is for the 43kg down curve.

The first step was to write a utility tool called “csv_averager_utility.py” that was in charge of averaging a selected range of distance and load cycles into their respective averages, thereby producing a CSV file with columns for pressure (mbar), distance (avg mm), and load (avg kg). As discussed in chapter 6.3, it was decided to average only the last three cycles because the first two cycles were consistent outliers, especially under lighter loads. The application GUI and code snippets are appended in Appendix 25 and Appendix 26.

After running the all the up and dows csv files through the utility, the files shown in Figure 7.2 were produced.

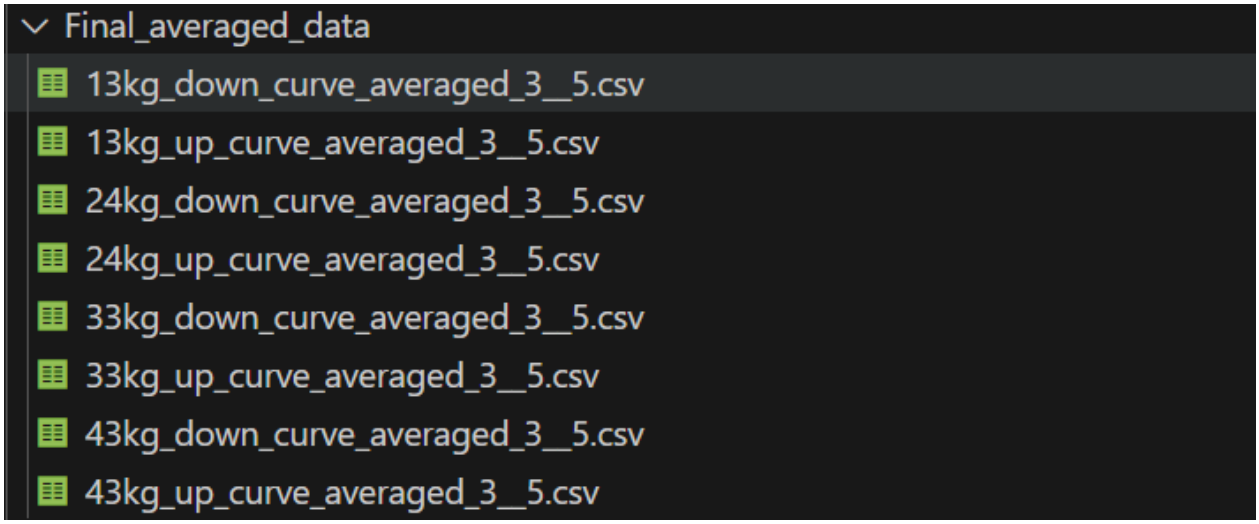


Figure 7.2-Averaged load muscle calibration files from cycles 3 to 5.

These new files had the structure shown in Figure 7.3 where now they only have pressure, distance and load columns and not multiple columns for each cycle.

Pressure (mbar)	Distance (mm)	Load(kg)
100	3	42.77947672443086
200	2.3333333333333335	42.77947672443086
300	3	43.05130818892286
400	4	42.91539245667686
500	5.666666666666667	42.98335032279986
600	7	43.01732925586136
700	8	43.05130818892286
800	9.666666666666666	43.05130818892286
900	11	43.01732925586136
1000	13	43.01732925586136
1100	14	43.08528712198436
1200	16	43.15324498810738
1300	18	43.05130818892286
1400	20.666666666666668	43.11926605504587
1500	23	43.05130818892286
1600	25.333333333333332	43.08528712198436
1700	28	42.98335032279985
1800	31	42.94937138973835
1900	34.333333333333336	43.08528712198436
2000	38	43.08528712198436
2100	41.666666666666664	43.01732925586136
2200	45.666666666666664	43.05130818892286
2300	50	43.01732925586136
2400	54.666666666666664	43.05130818892286
2500	59.333333333333336	43.08528712198436

Figure 7.3-Data output for 43kg up data after running it through utility app.

Averaging the data will better allow to prepare it into the correct D to P format required later on. As a refresher: DtoP=Distance to Pressure and PtoD=Pressure to Distance.

7.2. Converting averaged PtoD tables to DtoP tables in required format

The next step was to convert each load PtoD table into the format shown in Figure 7.4.

```
1   # weight=40. Up row is up curve and bottom row is down curve of hystheresis.
2   0.0,601.0,680.0,747.0,806.0,860.0,910.0,956.0,1000.0,1042.0,1081.0,1119.0,115
3   0.0,149.0,282.0,385.0,466.0,531.0,587.0,636.0,679.0,718.0,754.0,787.0,817.0,8
4
```

Figure 7.4-Snippet of distance to pressure table for a given load.

In this format, the 1st row is the weight header, the 2nd row is the up curve, and the 3rd row is the down curve. Each row is an array which has 200 elements corresponding to each individual mm distance stroke. Each value corresponds to the pressure required to obtain the given distance. For example, the

value located in element 5 in the 2nd row corresponds to the pressure needed to contract the muscle 5mm when lifting 40kg (860mbar).

These pressures are already pre-interpolated when transferred from the averaged PtoD tables. For example, if during calibration the muscle extends 5mm at 500mbar and 7mm at 800mbar, then the utility converting PtoD from DtoP interpolates (linear or cubic) for 6mm. For 6mm, the pressure would be 650mbar (if interpolating linearly).

The advantage of this method is that not only is it easier for the previously existing rollercoaster motion platform code to convert from distance to pressure needed to set a given actuator, but it also means it only needs to interpolate for weights instead of both weights and distances.

In order to transfer my calibration tables into the correct DtoP tables a utility app was made which could:

-Obtain the max length the user wants the muscles to contract (in this case 200mm)

-Group up/down calibration csv tables

-Work out interpolated distance to pressure arrays for each up/down curve for each calibrated load

-Put each up and down interpolated distance to pressure array into an output csv file

-Include load header in each output csv file

The name of the utility used to calibrate the muscles was “D_to_P_converter.py”. GUI snippets and code snippets of “D_To_P_converter.py” are appended in Appendix 27 and Appendix 28.

After running the averaged files shown in Figure 7.2, there are 4 resulting PtoD csv files shown in Figure 7.5.

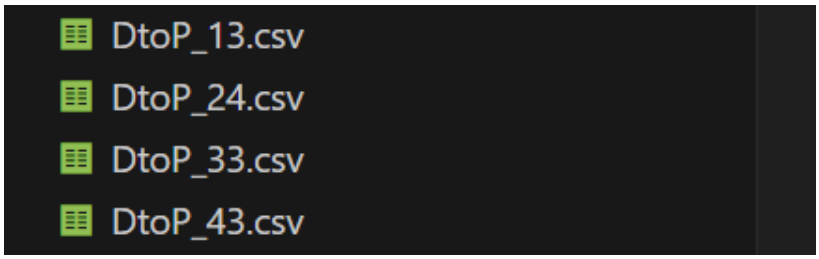


Figure 7.5-output DtoP files for all the calibrated loads.

Each DtoP load calibration file has the same structure as the DtoP table shown in Figure 7.4. DtoP_43.csv is shown in Figure 7.6 for the sake of cross-comparison.

```
1  # weight=43
2  0,86,171,100,400,460,525,600,700,760,825,900,950,1000,1100,1150,1200,1250,1300,1338,1375,1414,1457,1500,1543,1586,1625,1662,1700,1733,1
3  0,25,50,75,100,150,200,250,300,350,400,500,543,586,633,683,725,762,800,850,900,943,986,1025,1062,1100,1133,1167,1200,1233,1267,1300,132
4
```

Figure 7.6-DtoP table for 43kg load.

A quick plotter utility called “D_to_P_plotter.py” (GUI and code shown in Appendix 29 and Appendix 30) was made to visualize the output of each DtoP table. This is non-essential to the project and only serves to ensure the data looks correct. A plot of DtoP_43kg.csv is shown in Figure 7.7.

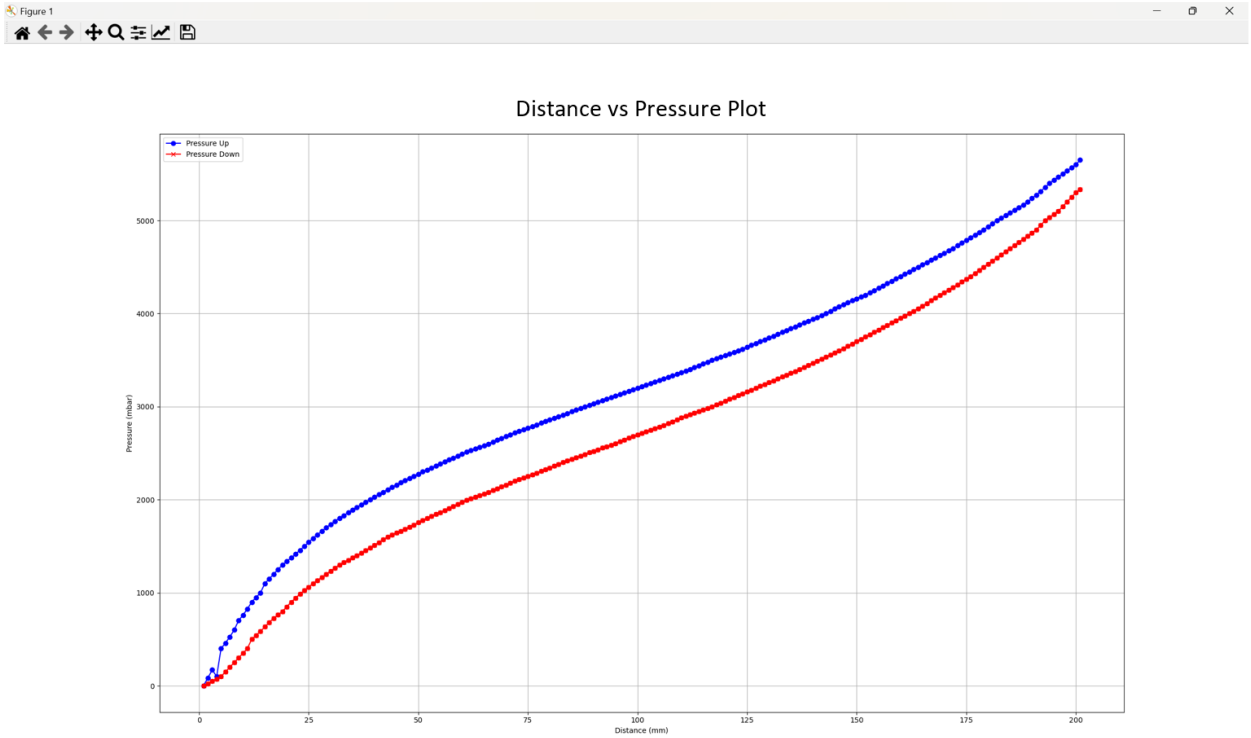


Figure 7.7-PtoD plot of 43kg csv file.

As expected, the shape of the DtoP table is the same as the original PtoD table but mirrored about y=x line.

7.3. Merging averaged DtoP tables into a master flight motion platform DtoP table

Now that all the PtoD tables were in the correct format, the final step was to merge them all together into a master DtoP table using a pre-existing method used in the previous rollercoaster project. Merging all the DtoP files into a master DtoP file allows the software to refer to just 1 table instead of multiple. The method is called “merge_d_to_p” and can be found within a script called “d_to_p_prep”. Figure 7.8 and Figure 7.9 showcase the code that was used to merge all the D_to_P tables into a single table.

```python
    def merge_d_to_p(self, infnames, outfname):
        # creates distance to pressure curves file using values from infiles
        # input file format:
        # header as:  weight=X  where X is weight in kg
        # row 1:  comma separated list of 200 up-going pressures for mm distances from 0 to 199
        # row 2:  comma separated list of 200 down-going pressures for mm distances from 0 to 199
        weights = []
        up_d_to_p = []
        down_d_to_p = []
        for fname in infnames:
            with open(fname) as fp:
                header = fp.readline()
                if 'weight=' in header:
                    weights.append(int(header.split('=')[1]))
                    up = fp.readline()
                    values = [int(round(float(i))) for i in up.split(',')]
                    up_d_to_p.append(values)
                    down = fp.readline()
                    values = [int(round(float(i))) for i in down.split(',')]
                    down_d_to_p.append(values)

        if len(weights) > 0:
            header = '# weights,' +  ','.join(str(n) for n in weights)
            combined_d_to_p= []
            for i in range (len(weights)):
                combined_d_to_p.append(up_d_to_p[i])
            for i in range (len(weights)):
                combined_d_to_p.append(down_d_to_p[i])
            with open(outfname, "w") as fp:
                fp.write(header + '\n')
                for i in range (len(weights)*2):  # write up then down
                    fp.write( ','.join(str(n) for n in combined_d_to_p[i] ) + '\n')
        else:
            print("no valid d to p files found")
```

Figure 7.8-merge_d_to_p code snippet.

A quick script which imports and calls this function (called D_to_P_merger.py) was also made as shown in the following code snippet.

```python
from calibrate.d_to_p_prep import D_to_P_prep

prepper = D_to_P_prep(200)
prepper.merge_d_to_p(["output\Final_averaged_data\DtoP_13.csv",
                      "output\Final_averaged_data\DtoP_24.csv",
                      "output\Final_averaged_data\DtoP_33.csv",
                      "output\Final_averaged_data\DtoP_43.csv",], "wheelchair_DtoP")
```

Figure 7.9-D_to_P_merger.py code snippet.

After running the utility, a csv file called wheelchair_DtoP was obtained. The structure of the csv file is shown in Figure 7.10.

```
# weights,13,24,33,43,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
0,33,67,100,133,167,200,300,350,400,433,467,500,550,600,633,667,700,727,755,782,808,833,858,883,908,933,958,983,1007,1027,1047,1067
0,38,75,112,150,188,167,325,400,460,517,567,617,667,714,757,800,838,875,912,950,988,1022,1056,1089,1118,1145,1173,1200,1227,1255,12
0,33,67,100,325,400,500,550,600,700,750,800,850,900,950,1000,1043,1086,1125,1162,1200,1238,1275,1311,1344,1378,1410,1440,1470,1500,
0,86,171,100,400,460,525,600,700,760,825,900,950,1000,1100,1150,1200,1250,1300,1338,1375,1414,1457,1500,1543,1586,1625,1662,1700,17
0,9,19,28,38,47,56,66,75,84,94,109,136,164,191,218,245,273,300,320,340,360,380,400,420,440,460,480,500,518,535,553,571,588,605,620,
0,12,23,35,46,58,69,81,92,112,150,187,229,271,311,344,378,411,444,478,508,533,558,583,609,636,664,691,717,742,767,792,813,833,853,8
0,20,40,60,80,100,150,200,250,300,350,400,438,475,514,557,600,633,667,700,733,767,800,833,867,900,930,960,990,1018,1045,1073,1100,1
0,25,50,75,100,150,200,250,300,350,400,500,543,586,633,683,725,762,800,850,900,943,986,1025,1062,1100,1133,1167,1200,1233,1267,1300
```

Figure 7.10-Master DtoP csv file after merging all the valid PtoD files.

The final DtoP table has 8 rows. The rows correspond to an up and download DtoP curve. The table structure is shown in Table 1.1. Note the second columns is meant to show the remaining 200 colums for each mm in 200mm.

Table 7.1-Structure of final merged DtoP table

Header	13, 24, 33, 43
13kg up DtoP	Pressure Val
24kg up DtoP	Pressure Val
33kg up DtoP	Pressure Val
43kg up DtoP	Pressure Val
13kg down DtoP	Pressure Val
24kg down DtoP	Pressure Val
33kg down DtoP	Pressure Val
43kg down DtoP	Pressure Val

Each DtoP row is treated as a 1D array. When the software runs, it'll automatically interpolate the output pressure between loads (if the configured load is somewhere in the middle) using the csv file shown in Figure 7.10.

After obtaining the master DtoP table with all the merged loads, a new configuration file was made and replaced off the existing configuration file for the rollercoaster. This new file references the aforementioned master DtoP table. After doing this, the test platform was able to move around noticeably different than before. The movement was less accurate and more jittery, but this was because the real-life muscles were different to the muscles the tables were calibrated for. The video showcasing this progress is shown in the Week 10 post of this project's blog [5].

7.4. Evaluating accuracy of merged DtoP table

As a final evaluation test on the merged DtoP table, a piece of software similar to the calibration software was made and tested on the calibration rig. This evaluation software steps in distance increments from 0mm to the maximum contraction distance (200mm). At each point it records the expected distance (distance) vs the recorded distance(measured_distance). It calculates the error at each distance point by subtracting the set distance from the measured distance so that:

error=distance-measured_dist

The scripts and flowcharts related to the DtoP evaluation tests are listed in the Appendix as follows:

-Appendix 31-Muscle DtoP test software flowchart

-Appendix 32-DtoP_Validator.py GUI snippet

-Appendix 33-DtoP_Validator.py code snippets

-Appendix 34-d_to_p_ver_2.py code snippets

-Appendix 35-DtoP_Validation_Plotter.py GUI snippet

-Appendix 36-DtoP_Validation_Plotter.py code snippets

The obtained results are shown in Figure 7.1. These are the graph results from "D_to_P_Validation_Plotter.py" program. The tested load was 24kg.

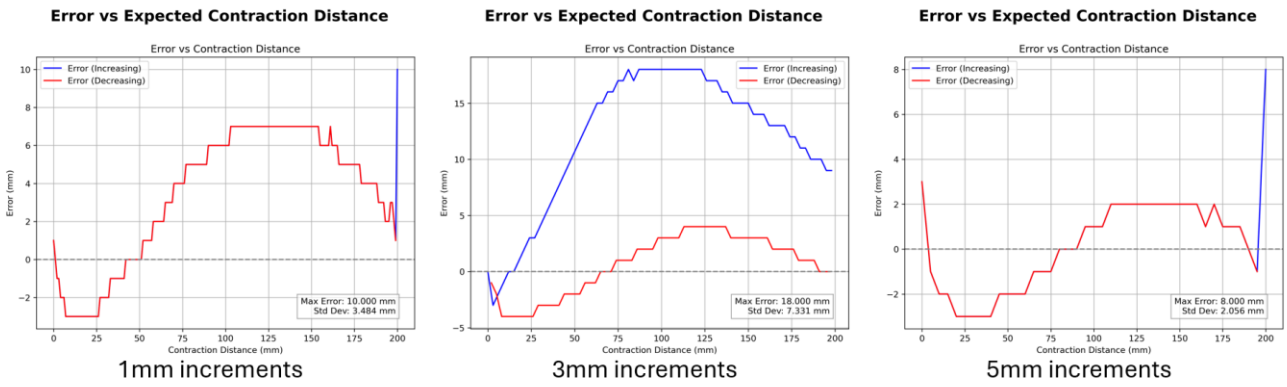


Figure 7.11-Plots of error vs set distance plots of merged DtoP table.

Due to time constraints, the weight interpolation feature could not be tested nor properly evaluated. The data obtained from the tests is limited due to only testing one load instead of multiple. Only the 24kg load was tested as this would be the usual load the muscles would be under.

What was surprisingly gathered from the results is how the distance increment has an effect on the error of the muscle's expected/set distance vs it's actual distance. A high increment of 5mm led to a more accurate response which was 8mm (least error). Meanwhile an increment of 3mm led to a much less accurate overall response which was 18mm (most error).

Some possible explanations for this could be the linear interpolation used to transfer PtoD tables to DtoP tables. Another could be due to some indexing error when obtaining a pressure for a given distance. The worst error is still within this projects target tolerance of +-25mm, therefore improving the accuracy of the DtoP merged table will be left for future work.

8. Results and discussions

8.1. Overall results

Table 8.1 highlight which requirements were able to be met at the end of the project. Green highlight means that the requirement was achieved. Red highlight means it wasn't achieved. Yellow highlight means that the requirement was only partially achieved.

Table 8.1-Requirements table outlining which requirements were met at the end of the project.

Requirement number	Requirement Description	Why was it not achieved? or why was it partially achieved?
1	Software has a basic user interface that allows the operator to control if and how the flight motion platform moves according to the input telemetry.	
2	Software provides START/PAUSE/STOP functionality to control the ride simulation as required.	
3	Software is able to accurately set each muscle's length to a desired length without muscle length feedback. It will only achieve this by setting their pressure. The desired tolerance must be within +-25mm uncertainty range.	
4	Software is able to set the platform's pose accurately enough for each motion to be noticeable to the naked eye. For example, the software can move the platform forwards or backwards in a visibly straight line. Target pose accuracy is very lenient in this project.	
5	Software takes into account the "safe state" of the wheelchair through input dock sensors. If the chair comes loose itself from the platform, the platform will halt to a stop in a pre-determined "safe state" pose.	Whilst the software does take into account a "safe state" signal as a Boolean signal, it does not interface with any real hardware to obtain such signal. The reason for this is due to the wheelchair docking system not being fully developed on time. Adapting this would not be too difficult and can be left for future work.
Advanced Objectives		
6	User interface is friendly and can be easily used by an inexperienced operator with minimal training.	Whilst the user interface is usable, it was not tested with other unexperienced users. There was little to no time to refine the user experience.
7	User interface allows operator to manually control the chair's pose and see a side-by-side simulation of it.	Manual mode and operator gains were deprecated as development evolved. Due to time constraints these features were left out but were shown to work fine in earlier stages of development.
8	Software allows support to connect chair to other flight sims aside from X-Plane.	So far, the only supported flight sim is X-Plane. However, because the software was written with a modular mindset, developing code for other sims is

		as easy as just writing a script that periodically sends a telemetry message via udp in a required format
9	Final ride platform has VR support for further immersion	This was not a focus on the main software. Adding VR support essentially comes down to setting up a VR headset on X-Plane which lies outside of the flight motion platform control software development which this project focuses on.

8.2. Discussion from results

Overall, most of the key objectives were successfully met. Although some were only partially met or not achieved at all, they were not as critical as those that were completed within the project's timeframe. Certain requirements also evolved over the course of the project. For example:

The original plan was to build a Unity-based GUI on top of the Python GUI. However, this was later changed in favour of developing the GUI using Python, while using Unity for a separate display application. The Python GUI proved sufficient for the project's needs, and Unity excelled at efficiently displaying the platform model, whereas the Python-based display was slow and unresponsive.

Some features, such as the washout filter, were not implemented. The washout functionality for the flight motion platform could not be completed within the project's timeline. The main reason for this was that the final flight motion platform was not fully constructed by the end of the development period. Since the physical platform was being developed independently from this project, certain elements and features that relied on access to the completed platform were deferred to future work. However, this had only a minimal impact on the project's overall deliverables.

9. Conclusions and future works

In conclusion, this project successfully achieved the following: it developed a control software system for operating a flight motion platform to enhance user immersion in the simulation, obtained calibration data for fluidic muscles through rigorous testing and data-driven improvements, and effectively integrated this calibration data back into the control software. Additionally, the calibration data met the project's target tolerance of +-25mm. The results were shown to be more accurate than those from previous work, contributing to further advancements in this area of research.

Although the project followed an agile methodology, which introduced challenges such as adapting to changing requirements and managing time effectively, it still achieved its core objectives. Some minor features, including washout filters, VR support, separating the control software and simulator onto different machines, manual control for the control software, and support for additional flight simulators, were deferred to future work. In particular, washout filters could not be fully tested due to the incomplete construction of the real flight platform.

For future work, the main focus should be on continuing from where this project left off and addressing the areas mentioned above. A further detailed summary for future work is shown in Table 9.1.

Table 9.1-Table detailing future work objectives.

Future work name	Cause of not being finished within project timeframe	Future work description
Washout filters	Final flight motion platform not being built in before the end of the project's timespan.	This will involve implementing the washout filters discussed in literature review. This will allow the platform to provide a more convincing motion to the user
Separating simulator and flight motion platform control software onto different machines	Not enough time to implement.	This involves running the airplane simulation on one computer while it communicates with another computer via UDP, which is responsible for receiving the UDP telemetry messages and outputting the motion cues for the flight motion platform, just as this project achieved. This setup allows the control computer, responsible for managing the flight motion platform, to offload some of the computational load associated with processing a graphically intensive application.
VR support	Final flight motion platform not being built in before the end of the project's timespan and low on priority list.	This is an additional feature which was not related with the control software. This is something that can be set in X-Plane completely separate from the flight motion platform control software. This can be implemented to improve ride immersion.
Improving distance to pressure (DtoP) accuracy	Not enough time to implement.	Further improving the accuracy of the DtoP control module. Some possible target causes are the linear interpolation used when converting PtoD to DtoP (could use cubic interpolation). Another cause could be an error or oversight on how the module retrieves table data which leads to erroneous pressure settings.
Manual control feature and gain settings	Not enough time to implement plus the final flight motion platform not being built in before the end of the project's timespan.	The manual control feature in control software would allow the operator to control and troubleshoot the flight motion platform pose through some sliders.
Support for other flight sims	Not enough time to implement and being low on priority list.	Some telemetry communication code could be written for simulators such as Microsoft Flight Sim or even custom unity games. This is meant to improve the flight motion platform's overall versatility.
UX/UI improvement and Quality Of Life (QoL) improvements to software	Not enough time to implement and being low on priority list.	Improving the UX experience would be helpful so that other operators can easily handle the platform with minimal difficulty. A crucial QoL improvement would be packaging my software as an executable binary that works just like any other app. At the moment manually running each relevant script via terminal is highly cumbersome.
Flight motion platform safety sensor integration into control software	Final flight motion platform not being built in before the end of the project's timespan.	Integrating the safety sensors is a key requirement to ensure the final wheelchair accessible flight motion platform is safe and does not cause any potential injuries to the user or others around the flight motion platform.

Make software raise platform to "parked position" to account for the final platform being flat against the ground. This is meant to prevent any collisions	Final flight motion platform was not built in time. Therefore, this feature became a much lower priority over getting the key milestones.	This is something the software has to account for when moving the platform. Otherwise, the flight motion platform will hit the floor when tilting by any small amount. The key is to simply raise the platform by around 5mm before it moves according to the simulation, so it does not hit the floor.
Using full 250mm range in tables instead of clipping them at 200mm	This was an oversight which was raised by my supervisor. Due to both time constraints and lack of final motion platform this was left for future work.	Using the full range of the muscles is crucial because about 5mm of the muscle are unusable due to the platform having to be raised by 5mm at all times. This leaves with 150mm range of motion per muscle as opposed to 200mm which provides a much more engaging experience (33% more movement range).

References

[1] "Falcon 2 programme homepage," [Online]. Available: https://www.aerosociety.com/careers-education/schools-outreach/the-falcon-2-programme/ .

[2] P. B. Walker, "Original MDX rollercoaster project in Future Fest 2015," [Online]. Available: https://loco.mdx.ac.uk/.

[3] "Project GitHub repository," [Online]. Available: https://github.com/electromaster0395/Flight-Motion-Platform-Control-Software.

[4] "Project GitLab repository," [Online]. Available: https://labcode.mdx.ac.uk/Electromaster039518/falcon2_flight_sim_software.

[5] "Project Blog Page," [Online]. Available: https://flightsimmotionplatformproject.wordpress.com/.

[6] "Motion platform Github page," [Online]. Available: https://github.com/michaelmargolis/MdxMotionPlatformV3 .

[7] "Video showcasing how motion platforms are used in filming," [Online]. Available: https://www.youtube.com/watch?app=desktop&v=NSQDlTupArk.

[8] "Brief history of the motion simulator webpage," [Online]. Available: https://www.wearetricycle.co.uk/pages/the-history-of-the-motion-simulator.

[9] "SIMONA research simulator," [Online]. Available: https://cs.lr.tudelft.nl/simona/facility/.

[10] "Motion systems PS-6TL-1500 homepage," [Online]. Available: https://motionsystems.eu/product/motion-platforms/ps-6tl-1500/.

[11] "DOF Reality Home Page," [Online]. Available: https://dofreality.com/.

[12] "MDX rollercoaster news page," [Online]. Available: https://www.mdx.ac.uk/news/2024/12/new-scientist/.

[13] “Paper on evaluation of the dynamic model of fluidic muscles,” [Online]. Available: https://ieeexplore.ieee.org/abstract/document/1639161.

[14] “X-Plane 11 6DOF platform project (does not implement washout filters),” [Online]. Available: https://www.youtube.com/watch?v=KMqd8JA2sos.

[15] “Paper on making stewart platform with fluidic muscles,” [Online]. Available: https://link.springer.com/article/10.1007/s11012-010-9407-8.

[16] R. N. a. H. B. Hubert Gattringer, “Paper on the modelling and control of a pneumatically driven stewart platform,” [Online]. Available: https://www.ferrobotics.com/app/uploads/2020/11/Publikation_movic2008_Gattringer_Naderer.pdf.

[17] “Dynamics Identification of Fluidic Muscle-actuated Planar Manipulator Using Two Nonlinear Models,” [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9108711&isnumber=9108708.

[18] “Articale about the inverse kinematics of a Stewart platform,” [Online]. Available: https://memememememememe.me/post/stewart-platform-math/.

[19] H. G. a. H. B. Klemens Springer, “Paper on washout filter concepts for motion simulators on the base of a stewart platform,” [Online]. Available: https://onlinelibrary.wiley.com/doi/pdf/10.1002/pamm.201110448.

[20] “Video qualitatively explaining basic concepts of washout in flight simulation,” [Online]. Available: https://www.youtube.com/watch?v=pfTdJD2OakA&t=31s.

[21] “Stewart_Py repository,” [Online]. Available: https://github.com/Yeok-c/Stewart_Py .

[22] “XPPython homepage,” [Online]. Available: https://xppython3.readthedocs.io/en/3.1.5/index.html .

[23] “XPPython plugin overview page,” [Online]. Available: https://xppython3.readthedocs.io/en/latest/development/index.html .

[24] “Final platform prototype render animation in Blender,” [Online]. Available: https://www.youtube.com/watch?v=-rCQpAGn2FM.

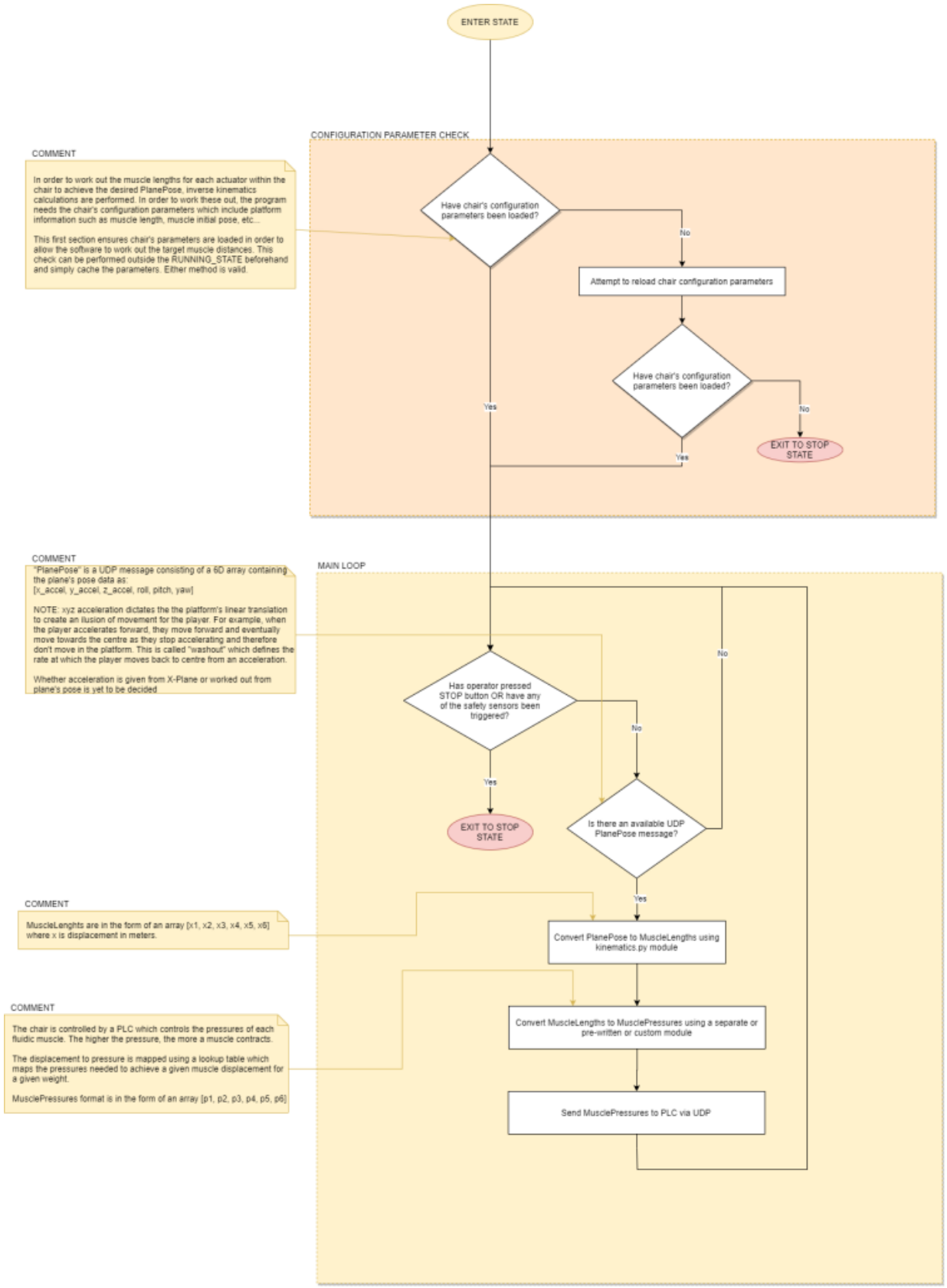
[25] “Official Blender character rig page,” [Online]. Available: https://studio.blender.org/characters/ .

[26] M. Rouillet, “Sketchfab wheelchair model,” [Online]. Available: https://sketchfab.com/3d-models/wheelchair-54911b7d81a44991804c87c6fc58c4df .

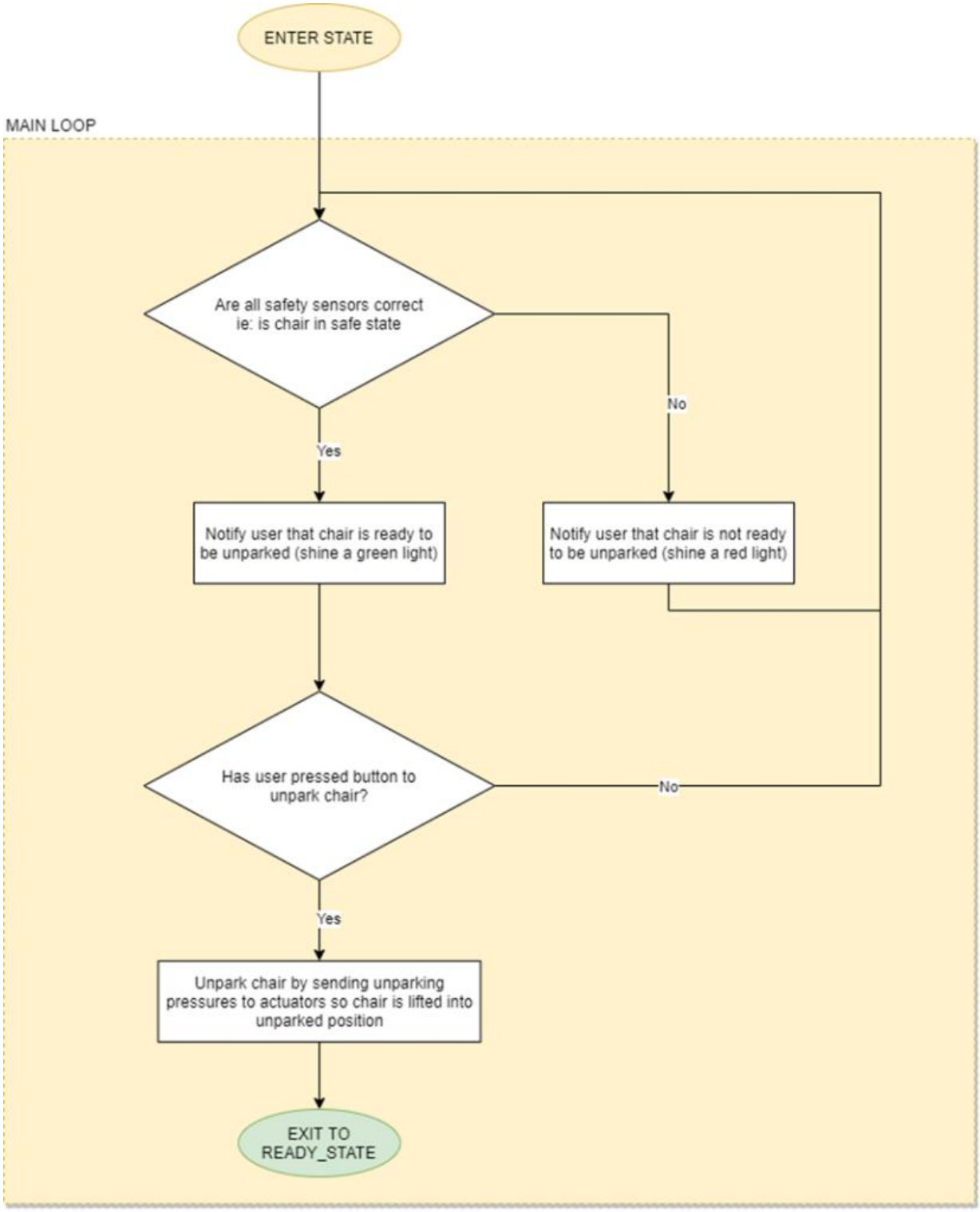
[27] “Unity display app source code repo,” [Online]. Available: https://drive.google.com/drive/folders/1-X01xMI0iL_UhBGNBjc6xrkrSI3d4_Iv?usp=sharing .

Appendix

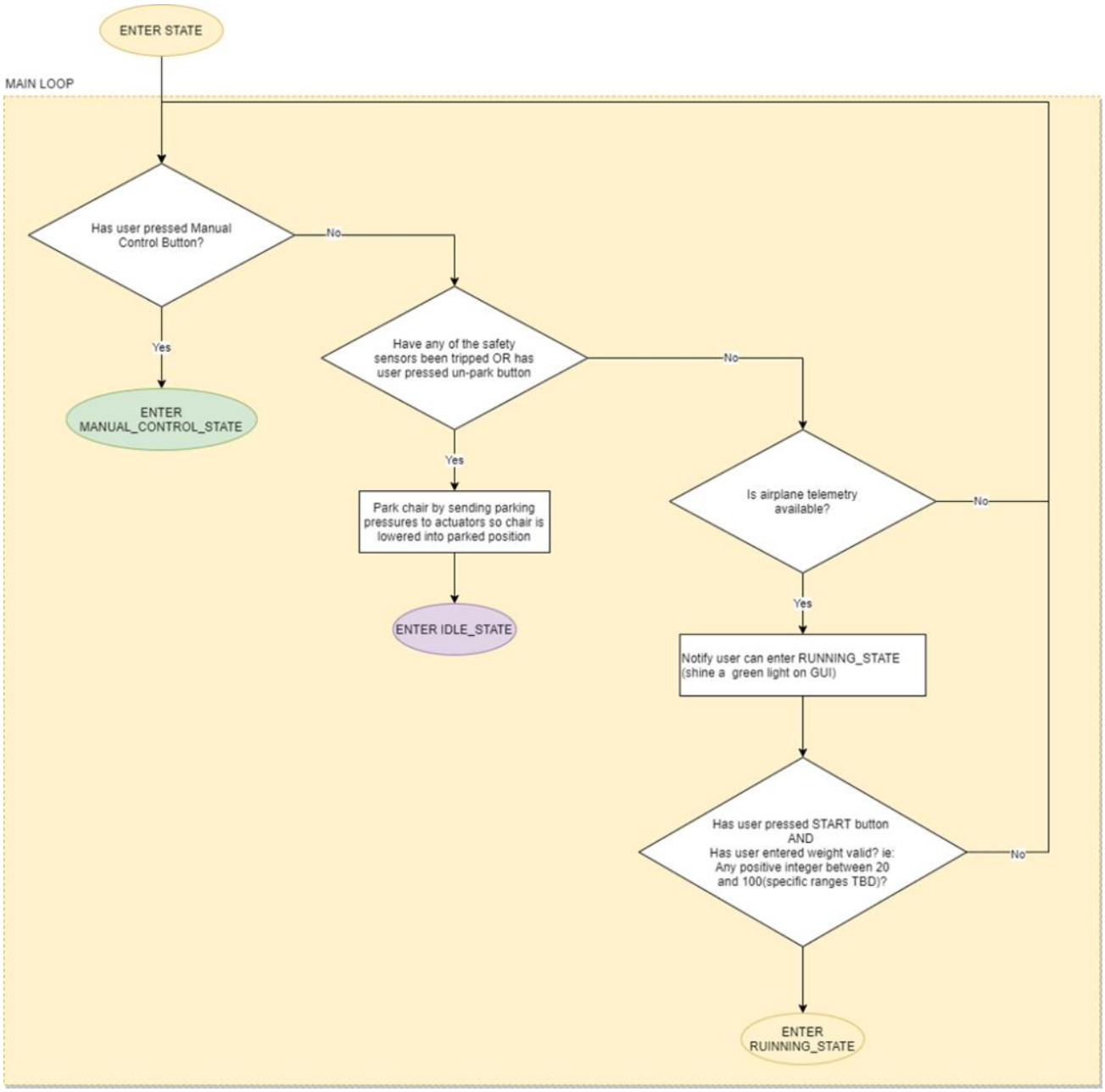
Appendix 1-RUNNING state logic.



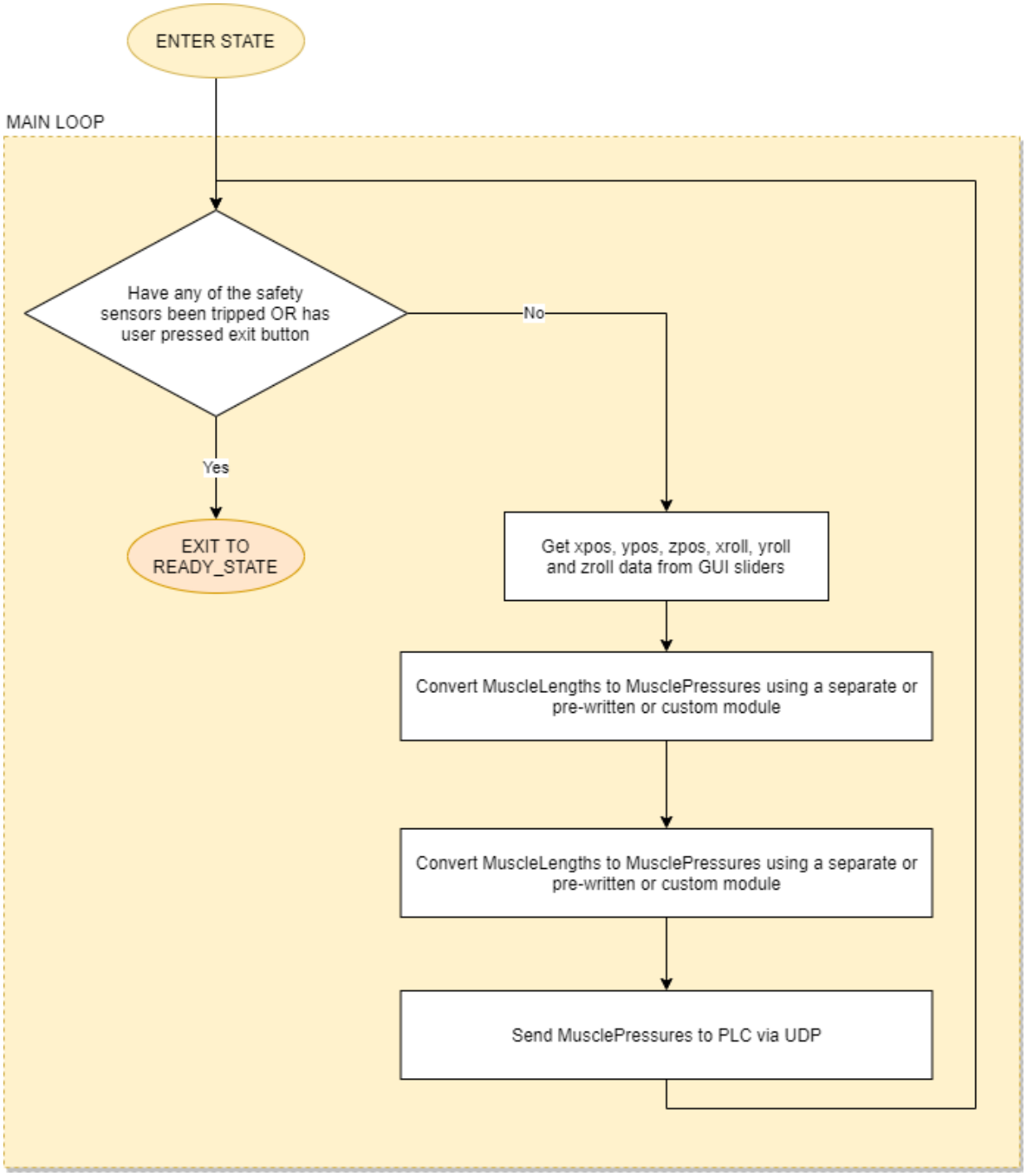
Appendix 2-IDLE state logic


ENTER STATE
MAIN LOOP
Are all safety sensors correct ie: is chair in safe state
No
Yes
Notify user that chair is ready to be unparked (shine a green light)
Notify user that chair is not ready to be unparked (shine a red light)
Has user pressed button to unpark chair?
No
Yes
Unpark chair by sending unparking pressures to actuators so chair is lifted into unparked position
EXIT TO READY_STATE

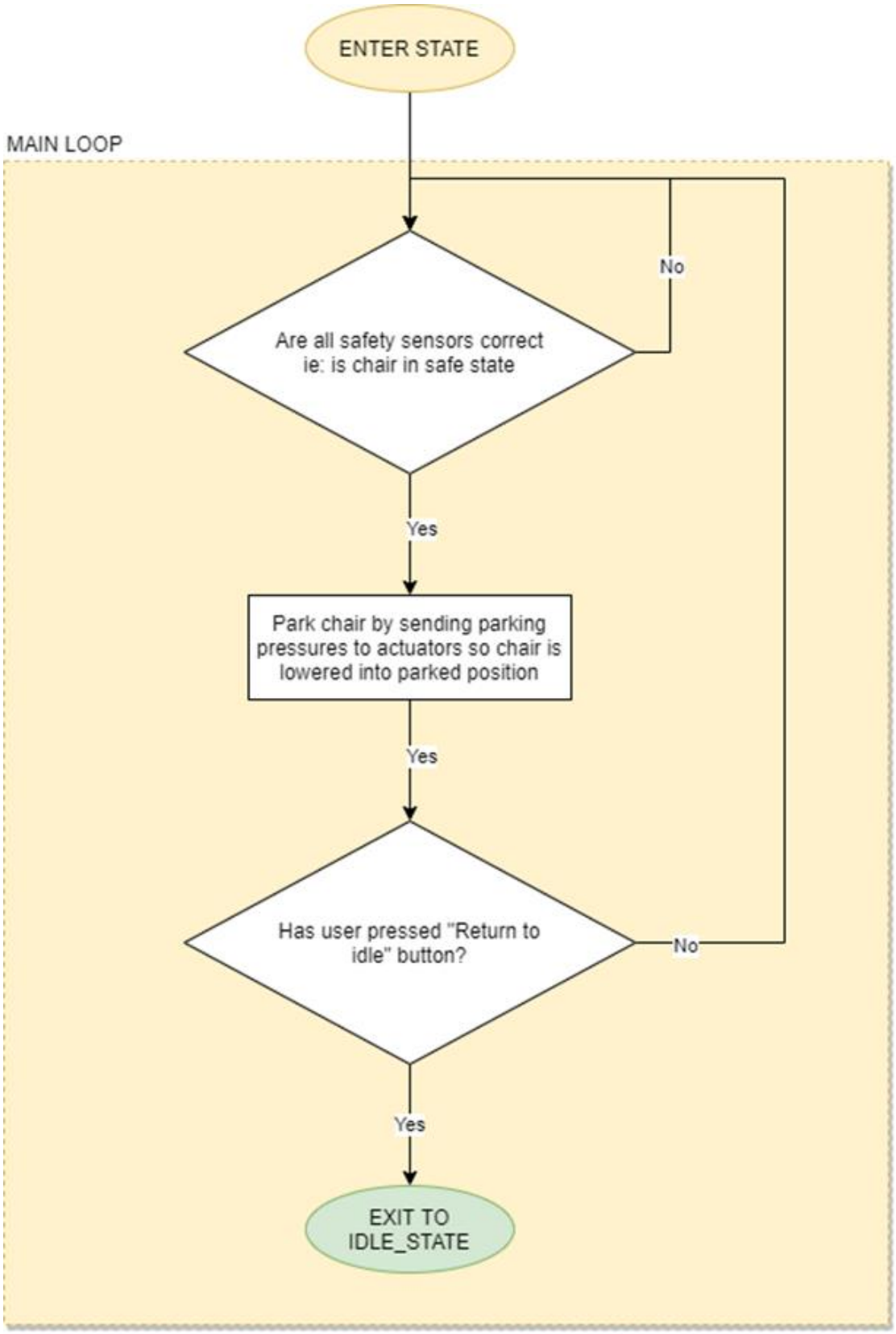
Appendix 3-READY state logic



Appendix 4-MANUAL state logic



Appendix 5-STOP state logic.



Appendix 6-state_machine.py code snippets

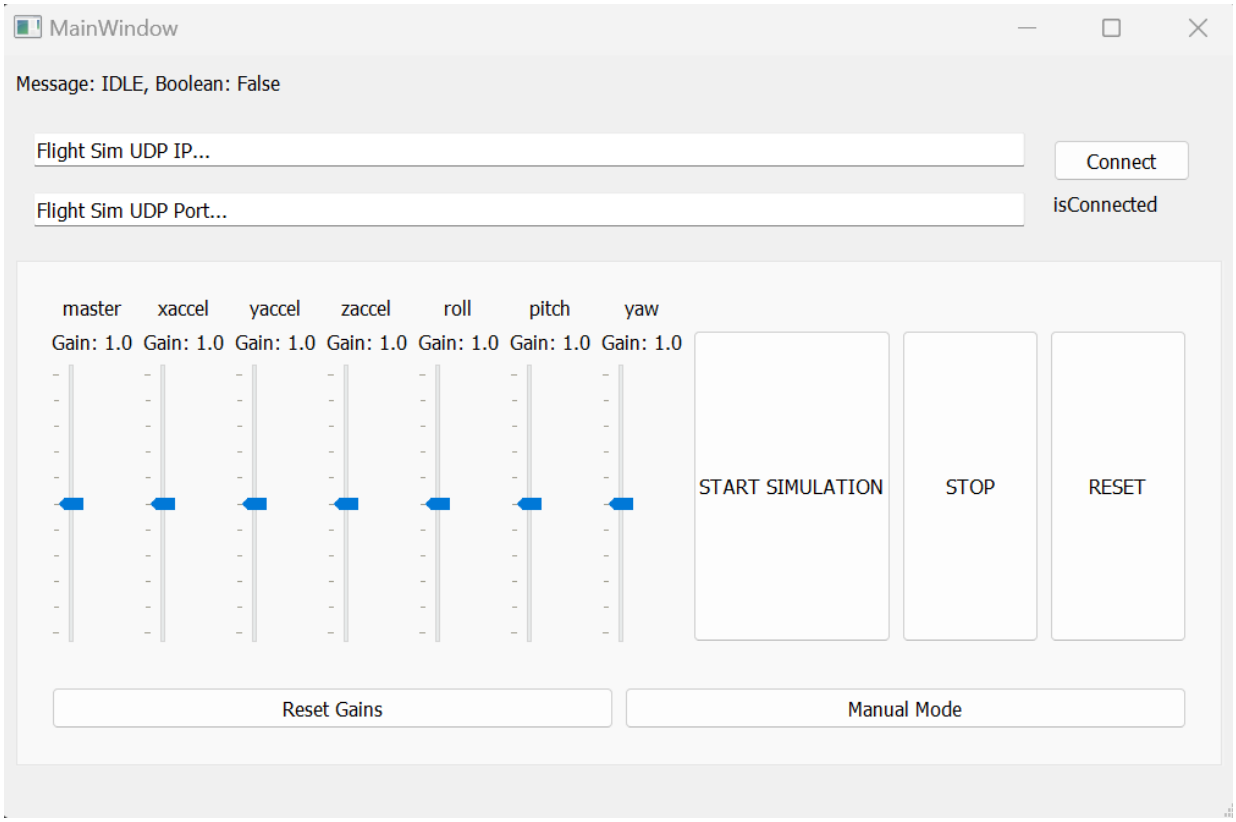
```python
import keyboard
import time
import socket
import struct
import select
import numpy as np
from udp_tx_rx import UdpReceive, UdpSend
from platform_kinematics_module import PoseToDistances
from platform_pose import Platform

# This script handles the core logic of the software, managing the state machine
# responsible for receiving airplane telemetry (linear acceleration & rotational pose),
# computing inverse kinematics for the mechanical chair, and sending commands via UDP.

# Define IP address and port numbers for receiving and sending data
UDP_IP = "127.0.0.1"  # Change if needed
SAFE_STATE_UDP_PORT = 4005
TELEMETRY_UDP_PORT = 10022
CMD_UDP_PORT = 6005
STATUS_UDP_PORT = 7005
POSE_DISPLAY_PORT = 10020
MANUAL_POSE_UDP_PORT = 9005

# Define update frequency (ensuring execution happens at 100Hz cycle time)
CYCLE_TIME = 1/100

# Initialize UDP receiver objects
safe_state_listener = UdpReceive(SAFE_STATE_UDP_PORT)
telemetry_listener = UdpReceive(TELEMETRY_UDP_PORT)
manual_telemetry_listener = UdpReceive(MANUAL_POSE_UDP_PORT)
command_listener = UdpReceive(CMD_UDP_PORT)

# Initialize UDP sender objects
display_sender = UdpSend()
status_sender = UdpSend()

# Global variables to retain values between cycles
global values
values = "0,0,0,0,0,0,0,0"  # Default telemetry data

global user_command
user_command = ""  # Command from user input

global user_gains
user_gains = []  # User-defined gain values

global safe_state
safe_state = False  # Safe state of the system

# Base class for different states within the state machine
class State:
    def __init__(self):
        pass
    def execute(self):
        pass

# IDLE state: Waits until the chair is in a safe state
class IDLE(State):
    def execute(self):
        global safe_state
        if safe_state:
            return "ready"

# READY state: Waits for user command to transition to MANUAL or RUNNING state
class READY(State):
    def execute(self):
        global safe_state, user_command
        if not safe_state:
            return "idle"
        elif user_command == "manual":
            return "manual"
        elif user_command == "running":
            return "running"

```

```python
# RUNNING state: Processes airplane telemetry, calculates inverse kinematics,
# and sends commands to the display. Transitions to STOP if chair is unsafe.
class RUNNING(State):
    def __init__(self):
        super().__init__()
        self.pose_to_distances = PoseToDistances()
        self.prev_values = values  # Optimization to reduce redundant UDP messages
        self.platform = Platform()

    def execute(self):
        global safe_state, user_command, values
        if not safe_state or user_command == "stop":
            return "stop"

        # Only send messages if telemetry values have changed
        elif values != self.prev_values:
            xyzrpy = ",".join(values.split(",")[:3]) + "," + ",".join(values.split(",")[-3:])
            xyzrpy_float = [float(num) for num in xyzrpy.split(",")]
            new_values = "request" + "," + xyzrpy + "," + ",".join(map(str, self.pose_to_distances.move_platform(xyzrpy_float)))
            print(new_values)
            display_sender.send(new_values, (UDP_IP, POSE_DISPLAY_PORT))
            self.prev_values = values
            self.platform.set_pose(xyzrpy_float)

# STOP state: Waits for user command and safe state before transitioning back to IDLE
class STOP(State):
    def execute(self):
        global safe_state, user_command
        if safe_state and user_command == "idle":
            return "idle"

# MANUAL_CONTROL state: Allows manual control of the chair through UDP commands
class MANUAL_CONTROL(State):
    def execute(self):
        global user_command
        if user_command == "ready":
            return "ready"

```

```python
# State machine class: Manages state execution and transitions
class StateMachine:
    def __init__(self):
        self.state = None

    def set_state(self, state: State):
        self.state = state
        print(f"State changed to {self.GetCurrentState()}")

    def execute(self):
        if self.state:
            transition = self.state.execute()
            if transition == "idle":
                self.set_state(IDLE())
            elif transition == "ready":
                self.set_state(READY())
            elif transition == "running":
                self.set_state(RUNNING())
            elif transition == "stop":
                self.set_state(STOP())
            elif transition == "manual":
                self.set_state(MANUAL_CONTROL())
        else:
            print("No state set")

    def GetCurrentState(self):
        return self.state.__class__.__name__

```

```python
# Main loop: Runs state machine until 'q' is pressed
if __name__ == "__main__":
    sm = StateMachine()
    sm.set_state(IDLE())

    target_time = time.time()

    while not keyboard.is_pressed("q"):
        current_time = time.time()

        # Ensure state machine runs at defined CYCLE_TIME
        if current_time >= target_time:

            temp_values = None
            while telemetry_listener.available() > 0:
                temp_values = telemetry_listener.get()

            if temp_values:
                values = temp_values[1].split(",", 1)[1]  # Remove "xplane_telemetry" prefix

            temp_safe_state = safe_state_listener.get()
            if temp_safe_state:
                safe_state = int(temp_safe_state[1])

            cmd_message = command_listener.get()
            if cmd_message:
                user_command, user_gains = cmd_message[1].split("|")
                user_gains = user_gains.split(",")
            else:
                user_command = None

            status = f"{sm.GetCurrentState()}|{safe_state}"
            status_sender.send(status, (UDP_IP, STATUS_UDP_PORT))

            sm.execute()
            target_time = current_time + CYCLE_TIME
```

Appendix 7-platform_control_app.py GUI snippets



Appendix 8-platform_control_app.py code snippets

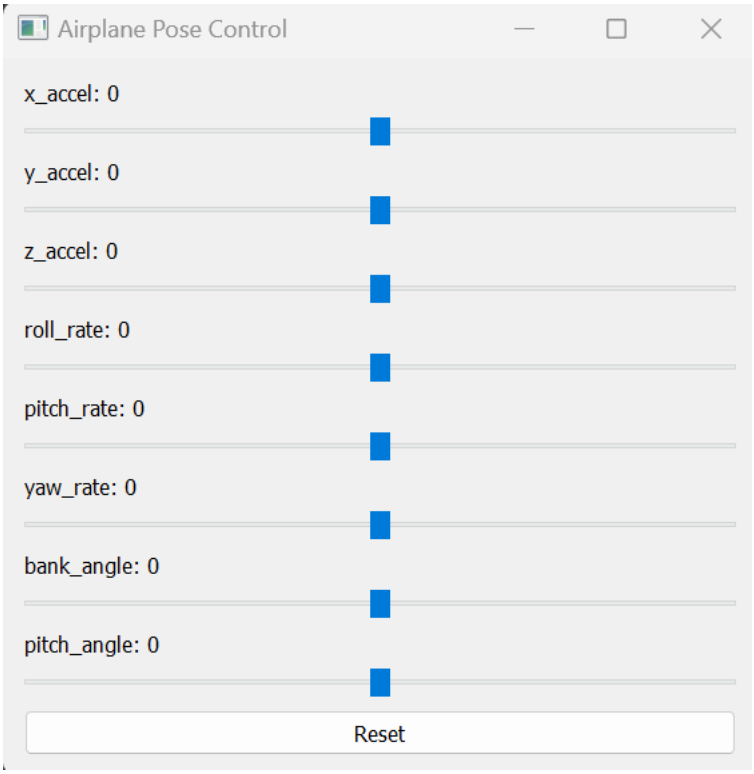
```python
from PyQt5 import QtWidgets
from gui import Ui_MainWindow  # Import the generated UI class
from PyQt5.QtCore import QThread
import sys
import socket
import threading
import numpy as np
import struct
import time
from udp_tx_rx import UdpReceive, UdpSend

# Set up UDP ports for communication
UDP_IP = "127.0.0.1"  # IP address for local communication
CMD_UDP_PORT = 6005  # Port for sending control commands
STATUS_UDP_PORT = 7005  # Port for receiving system status updates
MANUAL_POSE_UDP_PORT = 9005  # Port for sending manual control data

# Thread class to continuously listen for status updates from the state machine
class SM_Status_QThread(QThread):
    def __init__(self, callback, udp_object):
        super().__init__()
        self.callback = callback  # Store the callback function
        self.isRunning = True  # Control flag to manage thread execution
        self.UdpListener = udp_object  # UDP listener object

    def run(self):
        # Continuously check for new status messages while thread is running
        while self.isRunning:
            self.callback(self.UdpListener.get())

    def stop(self):
        """Stop the thread gracefully"""
        self.isRunning = False
        self.quit()
        self.wait()

```

```python
# Main application class that integrates the GUI and logic
class MainApp(QtWidgets.QMainWindow, Ui_MainWindow):
    def __init__(self):
        super().__init__()
        self.setupUi(self)  # Load the UI layout from the imported UI class

        # Initialize UDP communication objects
        self.cmd_sender = UdpSend()
        self.manual_pose_sender = UdpSend()
        self.status_receiver = UdpReceive(STATUS_UDP_PORT, blocking=False)

        self.worker = None  # Background thread for receiving system status

        # Start background thread to continuously update GUI status
        if self.worker is None or not self.worker.isRunning:
            self.worker = SM_Status_QThread(self.update_status_label, self.status_receiver)
            self.worker.start()

        # Connect slider signals to labels for displaying gain values
        self.master.valueChanged.connect(lambda value, lbl=self.master_label, mode="sim_gains": self.update_label(value, lbl, mode))
        self.xaccel.valueChanged.connect(lambda value, lbl=self.xaccel_label, mode="sim_gains": self.update_label(value, lbl, mode))
        self.yaccel.valueChanged.connect(lambda value, lbl=self.yaccel_label, mode="sim_gains": self.update_label(value, lbl, mode))
        self.zaccel.valueChanged.connect(lambda value, lbl=self.zaccel_label, mode="sim_gains": self.update_label(value, lbl, mode))
        self.roll.valueChanged.connect(lambda value, lbl=self.roll_label, mode="sim_gains": self.update_label(value, lbl, mode))
        self.pitch.valueChanged.connect(lambda value, lbl=self.pitch_label, mode="sim_gains": self.update_label(value, lbl, mode))
        self.yaw.valueChanged.connect(lambda value, lbl=self.yaw_label, mode="sim_gains": self.update_label(value, lbl, mode))

        self.xaccel_manual.valueChanged.connect(lambda value, lbl=self.xaccel_label_2, mode="manual_gains": self.update_label(value, lbl, mode))
        self.yaccel_manual.valueChanged.connect(lambda value, lbl=self.yaccel_label_2, mode="manual_gains": self.update_label(value, lbl, mode))
        self.zaccel_manual.valueChanged.connect(lambda value, lbl=self.zaccel_label_2, mode="manual_gains": self.update_label(value, lbl, mode))
        self.roll_manual.valueChanged.connect(lambda value, lbl=self.roll_label_2, mode="manual_gains": self.update_label(value, lbl, mode))
        self.pitch_manual.valueChanged.connect(lambda value, lbl=self.pitch_label_2, mode="manual_gains": self.update_label(value, lbl, mode))
        self.yaw_manual.valueChanged.connect(lambda value, lbl=self.yaw_label_2, mode="manual_gains": self.update_label(value, lbl, mode))

        # Connect button clicks to sending control commands
        self.start.clicked.connect(lambda value, command="running": self.send_control_command(command))
        self.stop.clicked.connect(lambda value, command="stop": self.send_control_command(command))
        self.reset.clicked.connect(lambda value, command="idle": self.send_control_command(command))
        self.manual_mode_button.clicked.connect(lambda value, command="manual": self.send_control_command(command))
        self.simulation_mode_button.clicked.connect(lambda value, command="ready": self.send_control_command(command))

        # Reset sliders when reset buttons are clicked
        self.reset_gains.clicked.connect(self.reset_sliders)
        self.reset_gain_manual.clicked.connect(self.reset_sliders)
```

```python
        # Ensure the first tab (main control) is selected by default
        self.tabWidget.setCurrentIndex(0)
        self.tabWidget.tabBar().hide()

    # Update label text with the current slider value based on the mode (simulation/manual)
    def update_label(self, value, label, mode):
        if mode == "sim_gains":
            label.setText(f"Gain: {value/5}")
        elif mode == "manual_gains":
            label.setText(f"Gain: {(value*20)-100}")

    # Send control command with the selected state and gain values
    def send_control_command(self, state):
        data_array = np.array([self.master.value(), self.xaccel.value(), self.yaccel.value(), self.zaccel.value(),
                               self.roll.value(), self.pitch.value(), self.yaw.value()])
        data_array = data_array/5  # Normalize gain values

        data_str = ",".join(map(str, data_array))  # Convert array to comma-separated string
        message = f"{state}|{data_str}"  # Format message
        print(message)
        self.cmd_sender.send(message, (UDP_IP, CMD_UDP_PORT))  # Send command via UDP

    # Update the GUI status label with the latest state information
    def update_status_label(self, data):
        if data is None:
            return

        message, flag = data[1].split("|")
        self.SM_STATUS.setText(f"Message: {message}, Boolean: {flag}")

        # Update start button text based on safety status
        if flag:
            self.start.setText("START SIMULATION")
        else:
            self.start.setText("CHECK SAFETY SENSORS")

        # Automatically switch to manual control tab if in MANUAL_CONTROL state
        if message == "MANUAL_CONTROL":
            self.tabWidget.setCurrentIndex(1)
        else:
            self.tabWidget.setCurrentIndex(0)
```

```
    # Handle window close event to properly stop background threads
    def closeEvent(self, event):
        pass  # Placeholder for cleanup logic if needed

    # Reset all sliders to their default middle position
    def reset_sliders(self):
        sliders = [self.master, self.xaccel, self.yaccel, self.zaccel, self.roll, self.pitch, self.yaw,
                   self.xaccel_manual, self.yaccel_manual, self.zaccel_manual, self.roll_manual, self.pitch_manual, self.yaw_manual]
        for slider in sliders:
            slider.setValue(5)

# Initialize and run the PyQt application
if __name__ == "__main__":
    app = QtWidgets.QApplication(sys.argv)
    window = MainApp()
    window.show()
    sys.exit(app.exec_())
```

Appendix 9-dummy_airplane_app.py GUI snippets

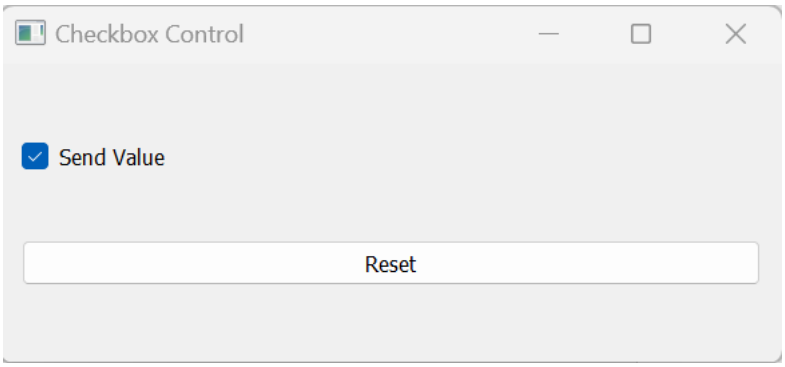


Appendix 10-dummy_airplane_app.py code snippets

```python
import sys
import socket
import struct
from PyQt5.QtWidgets import QApplication, QWidget, QVBoxLayout, QSlider, QLabel, QCheckBox, QPushButton
from PyQt5.QtCore import Qt, QTimer
from udp_tx_rx import UdpSend, UdpReceive
import math

#This is a dummy airplane app where each part of the plane's pose is controlled using a dedicated slider

UDP_IP = "127.0.0.1"

#Define telemetry port for which to send out the messages containing status (any of the safety sensors tripper) and airplane pose
TELEMETRY_UDP_PORT = 5005

#NOTE: Again, its worth noting that in this project acceleration and translation are used somewhat arbitrarily as the airplane's linear accel
#dictates the chair's translatation to create an ilusion of movement.

#Define main window widget containing 6 sliders for the airplane's 6D pose (xaccel, yaccel, zaccell, roll, pitch, yaw)
class AirplaneControl(QWidget):
    def __init__(self):
        super().__init__()

        #Set window title and size
        self.setWindowTitle("Airplane Pose Control")
        self.setGeometry(100, 100, 300, 400)

        #Define layout to be vertical so that added components are on top of each other
        self.layout = QVBoxLayout()

        #Iteratively add label and slider widgets to the main window layout
        self.sliders = {}
        self.labels = {}
        self.pose_keys = ["x_accel", "y_accel", "z_accel", "roll_rate", "pitch_rate", "yaw_rate", "bank_angle", "pitch_angle"]
        self.pose_values = {key: 0 for key in self.pose_keys}

        for key in self.pose_keys:
            label = QLabel(f"{key}: 0")
            slider = QSlider(Qt.Horizontal)
            slider.setMinimum(-180)
            slider.setMaximum(180)
            slider.setValue(0)
            slider.setSingleStep(1)  # Increment/decrement by 1 on arrow click
            slider.valueChanged.connect(lambda value, k=key: self.update_pose(k, value))

            self.labels[key] = label
            self.sliders[key] = slider

            self.layout.addWidget(label)
            self.layout.addWidget(slider)

        #Add reset button widget to main window layout which resets all the slider values to their default value
        self.reset_button = QPushButton("Reset")
        self.reset_button.clicked.connect(self.reset_sliders)
        self.layout.addWidget(self.reset_button)
        self.setLayout(self.layout)

        #UDP sender object
        self.udp_sender = UdpSend()

        #Define timer which sends out UDP messages every 100ms
        self.timer = QTimer()
        self.timer.timeout.connect(self.send_udp_data)
        self.timer.start(100)  # Send data every 100 ms

    #Method which updates each element of the self.pose list to each respective slider value. This also sets the label text to the appropiate val
    def update_pose(self, key, value):
        self.pose_values[key] = value
        self.labels[key].setText(f"{key}: {value}")
```

```
    #This method resets all the sliders to 0
    def reset_sliders(self):
        """Resets all sliders to 0."""
        for key in self.pose_keys:
            self.sliders[key].setValue(0)  # This will trigger valueChanged and update labels

    #This method sends the required telemetry data as a list of 6 floats and one boolean. The 6 floats are the airplane's pose and the boolean
    #states whether the chair or airplane iss in it's safe state
    def send_udp_data(self):
        data = list(self.pose_values.values())  # Convert dict_values to a list
        data[-2:] = [math.radians(value) for value in data[-3:]]  # Convert last two items to radians
        data_str = ("xplane_telemetry, " + ",".join(map(str, data)))
        self.udp_sender.send(data_str, (UDP_IP, TELEMETRY_UDP_PORT))

#Start up the window once script is called
if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = AirplaneControl()
    window.show()
    sys.exit(app.exec_())

```

Appendix 11-safe_state_dummy.py GUI snippet

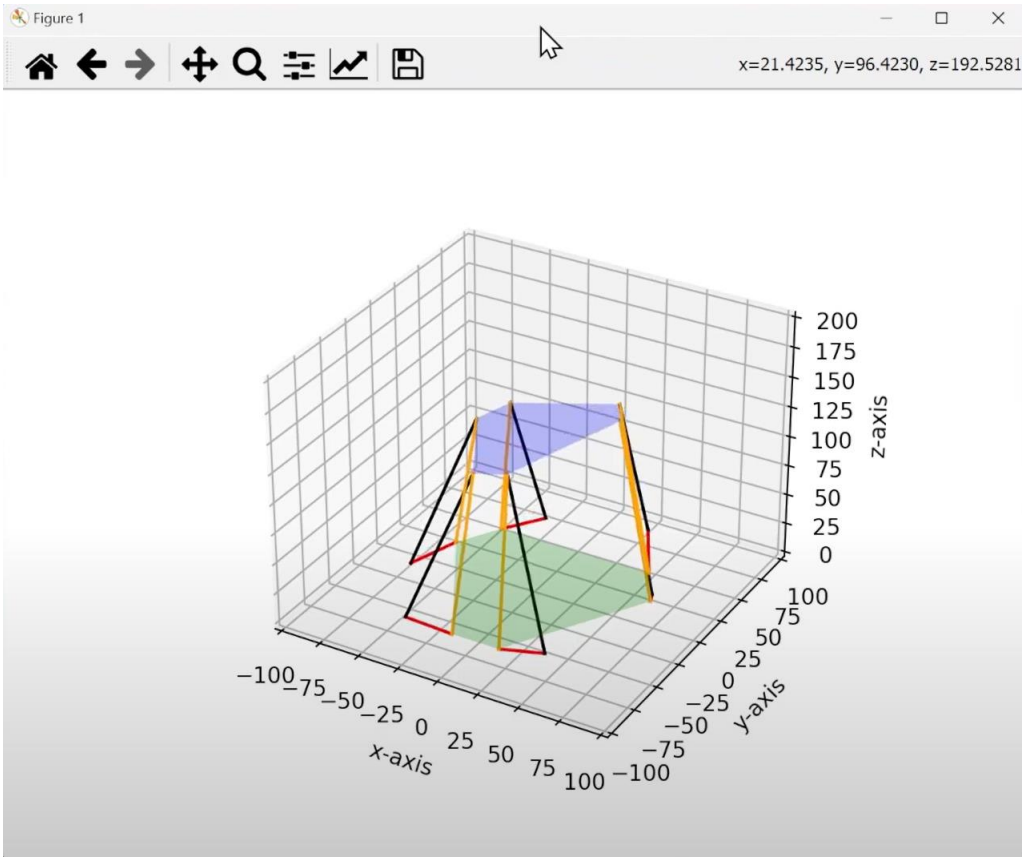


Appendix 12-safe_state_dummy.py code snippets

```
import sys
import socket
import struct
from PyQt5.QtWidgets import QApplication, QWidget, QVBoxLayout, QCheckBox, QPushButton
from PyQt5.QtCore import QTimer
from udp_tx_rx import UdpSend

# Define UDP IP and port for sending telemetry data
UDP_IP = "127.0.0.1"
TELEMETRY_UDP_PORT = 4005

class CheckboxControl(QWidget):
    def __init__(self):
        super().__init__()

        # Set up the window properties
        self.setWindowTitle("Checkbox Control")
        self.setGeometry(100, 100, 300, 200)

        # Create a vertical layout
        self.layout = QVBoxLayout()

        # Define checkbox for sending data
        self.checkbox = QCheckBox("Send Value")
        self.checkbox.stateChanged.connect(self.send_udp_data)
        self.layout.addWidget(self.checkbox)

        # Define reset button to uncheck the checkbox
        self.reset_button = QPushButton("Reset")
        self.reset_button.clicked.connect(self.reset_checkbox)
        self.layout.addWidget(self.reset_button)

        # Set layout for the window
        self.setLayout(self.layout)

        # Initialize UDP sender object
        self.udp_sender = UdpSend()

        # Define a timer to send UDP messages every 100ms
        self.timer = QTimer()
```

```
        self.timer.timeout.connect(self.send_udp_data)
        self.timer.start(100)  # Send data every 100 ms

    def reset_checkbox(self):
        """Resets checkbox to unchecked state."""
        self.checkbox.setChecked(False)

    def send_udp_data(self):
        """Sends UDP data based on checkbox state."""
        data = int(self.checkbox.isChecked())  # Convert checkbox state to integer (0 or 1)
        data_str = str(data)  # Convert to string format for sending
        self.udp_sender.send(data_str, (UDP_IP, TELEMETRY_UDP_PORT))  # Send data via UDP
        #print("Sent data:", data_str)  # Uncomment for debugging

# Run the application if this script is executed directly
if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = CheckboxControl()
    window.show()
    sys.exit(app.exec_())
```

Appendix 13-stewart_displa.py GUI snippet



Appendix 14-stewart_display.py code snippets

```python
import socket
import struct
import select
from src.stewart_controller import Stewart_Platform
import matplotlib.pyplot as plt
import numpy as np
import time
import matplotlib.animation as animation
from udp_tx_rx import UdpReceive

# Define UDP port to listen for pose data, which will be sent from the state_machine.py node
UDP_IP = "127.0.0.1"
TELEMETRY_UDP_PORT = 8005

# Set up UDP socket to listen for pose data (Non-blocking)
telemetry_listener = UdpReceive(TELEMETRY_UDP_PORT)

# Initialize Stewart Platform parameters (Hardcoded for now, will be configurable later)
# This setup is temporary and serves as a debug visualization of platform movement.
platform = Stewart_Platform(132/2, 100/2, 30, 130, 0.2269, 0.82, 5*np.pi/6)

# Function to listen for incoming airplane telemetry and update pose data.
# Inverse kinematics is handled using the "Stewart_Py" repository.
# Eventually, inverse kinematics calculations will be moved to state_machine.py,
# and the chair will only receive muscle pressure values.
def listen_for_telemetry():
    """Reads the latest UDP message, discarding old ones."""
    data = None

    # Retrieve the most recent telemetry data from the UDP buffer
    while telemetry_listener.available() > 0:
        data = telemetry_listener.get()

    if data is None:
        return [], []  # Return empty lists if no data is received

    # Parse incoming pose data
    pose_data = data[1].split(",")
    pose_data = [int(float(r)) for r in pose_data]  # Convert string values to integers
    rotation = pose_data[5:9]  # Extract rotation values from the data

```

```python
    # Flip roll and pitch for debugging purposes
    rotation[0], rotation[1] = -rotation[1], rotation[0]

    trans, rot = [0, 0, 0], rotation  # Translation is set to zero (for now)
    return trans, rot

# Initialize figure and axis for real-time 3D plotting
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Function to update the Stewart platform's position and orientation based on telemetry data
def update(frame):
    """Update function for real-time plotting."""
    trans, rot = listen_for_telemetry()
    if trans and rot:  # Ensure valid data exists
        servo_angles = platform.calculate(np.array(trans), np.array(rot) * (np.pi / 180))  # Convert degrees to radians
        platform.plot_platform(ax=ax)  # Update the 3D plot

# Start real-time animation (100ms update rate)
ani = animation.FuncAnimation(fig, update, interval=100)

# Display the 3D plot
plt.show()
```

Appendix 15-PI_simple_telemetry.py written by Michael Margolis (retooled by Omar Maaouane)

```
"""
Mdx_Telemetry
Written by Michael Margolis

Sends 6DoF Xplane telemetry over UDP.

# xplane uses OpenGL coordinates - x is right, y up, z back

Mdx coordinate frame follows ROS conventions, positive values: X is forward, Y is left, Z is up,
roll is right side down, pitch is nose down, yaw is CCW; all from perspective of person on platform.

datarefs:
    xplane translation acc are in g
        axil = surge = x  (invert)
        side = sway = y   (invert)
        nrml = heave = z
    xplane roll and pitch are in degrees
    rotation velocities are rad/sec
        roll              (deg to rad, invert)
        pitch             (deg to rad   invert
        yaw               (invert)


        // Translation
        double acc_axial;              // [G] backward +
        double acc_lateral;            // [G] right +
        double acc_normal;             // [G] up +, +1.0 normal g

        // Rotation
        double vel_roll;               // [rad/sec] clockwise + (Rechtskurve)
        double vel_pitch;              // [rad/sec] nose up +
        double vel_yaw;                // [rad/sec] clockwise + (Rechtskurve)
        double roll;                   // [deg] clockwise +
        double pitch;                  // [deg] nose up +


   msg format:
        surge accel, sway accel, heave accel, roll vel, pitch vel, yaw vel, roll, pitch
"""


#"D:\Program Files\X-Plane 12\X-Plane.exe" --load_rep="D:\MdxCessnaFlight.rep"

```

```
from XPPython3 import xp
from collections import namedtuple
from math import radians #, pi, degrees, sqrt
from udp_tx_rx import UdpReceive
import XPLMDataAccess

transform_refs = namedtuple('transform_refs', \
                  ('DR_g_axil', 'DR_g_side', 'DR_g_nrml', \
                   'DR_Prad', 'DR_Qrad',  'DR_Rrad', \
                   'DR_theta', 'DR_psi', 'DR_phi', \
                   'DR_groundspeed'))


UDP_IP = "127.0.0.1"
TARGET_PORT = 10022 # port of sim interface controller telemetry socket

xplm_key_pause = 0


class PythonInterface:
    def XPluginStart(self):
        self.Name = "PlatformItf v1.01"
        self.Sig = "Mdx.Python.UdpTelemetry"
        self.Desc = "A plug-in for the Mdx platform that sends telemetry data over UDP."
        LISTEN_PORT = 10023
        self.controller_addr = []
        self.udp = UdpReceive(LISTEN_PORT)

        self.init_drefs()

        # Create our menu
        Item = xp.appendMenuItem(xp.findPluginsMenu(), "Flight transforms", 0)
        self.InputOutputMenuHandlerCB = self.InputOutputMenuHandler
        self.Id = xp.createMenu("Platform Interface", xp.findPluginsMenu(), Item, self.InputOutputMenuHandlerCB, 0)
        xp.appendMenuItem(self.Id, "View Transforms", 1)

        self.IsWidgetVisible = 0  # Flag indicating if widget is being displayed.

        self.OutputDataRef = []
        for Item in range(len(self.xform_drefs)):
            self.OutputDataRef.append(xp.findDataRef(self.xform_drefs[Item]))
```

```
85
86          # Register our FL callbadk with initial callback freq of 1 second
87          self.InputOutputLoopCB = self.InputOutputLoopCallback
88          xp.registerFlightLoopCallback(self.InputOutputLoopCB, 1.0, 0)
89
90          return self.Name, self.Sig, self.Desc
91
92      def XPluginStop(self):
93          # Unregister the callback
94          xp.unregisterFlightLoopCallback(self.InputOutputLoopCB, 0)
95
96          if self.IsWidgetVisible == 1:
97              xp.destroyWidget(self.InputOutputWidget, 1)
98              self.IsWidgetVisible = 0
99
100         xp.destroyMenu(self.Id)
101         self.udp.close()
102
103     def XPluginEnable(self):
104         return 1
105
106     def XPluginDisable(self):
107         pass
108
109     def XPluginReceiveMessage(self, inFromWho, inMessage, inParam):
110         pass
111
112
113     def init_drefs(self):
114         self.xform_drefs = []
115         # the following must match transform_refs namedtuple
116         self.xform_drefs.append('sim/flightmodel/forces/g_axil')
117         self.xform_drefs.append('sim/flightmodel/forces/g_side')
118         self.xform_drefs.append('sim/flightmodel/forces/g_nrml')
119         self.xform_drefs.append("sim/flightmodel/position/Prad") # roll rate rad/sec
120         self.xform_drefs.append("sim/flightmodel/position/Qrad") # pitch rate rad/sec
121         self.xform_drefs.append("sim/flightmodel/position/Rrad") # yaw rate  rad/sec
122         self.xform_drefs.append("sim/flightmodel/position/theta") # pitch degrees
123         self.xform_drefs.append("sim/flightmodel/position/psi") # heading degrees
124         self.xform_drefs.append("sim/flightmodel/position/phi") # bank angle degrees
125         self.xform_drefs.append("sim/flightmodel/position/groundspeed") # not yet used
126
127
128         self.NumberOfDatarefs = len(self.xform_drefs)
129         if self.NumberOfDatarefs != len(transform_refs._fields) :
130             xp.log("invalid nbr drefs {} != {}".format(len(self.NumberOfDatarefs), len(transform_refs)))
131
132         # self.XformDesc =  ('X (surge)', 'Y (sway)', 'Z (heave)', 'Roll', 'Pitch', 'Yaw')
133         self.XformDesc =  ('Surge (g)', 'Sway (g)', 'Heave (g)', 'Roll rate (rad/s)', 'Pitch rate (rad/s)',
134                            'Yaw rate (rad/s)', 'Roll angle (rad)', 'Pitch angle (rad)' )
135
136         self.toggle_replay =  xp.findCommand('sim/replay/replay_toggle') # toggle replay mode on/off.
137         self.replay_off = xp.findCommand('sim/replay/replay_off')  # Replay mode off.
138         self.go_to_replay_begin = xp.findCommand('sim/replay/rep_begin') # Replay mode: go to beginning.
139         self.go_to_replay_end = xp.findCommand('sim/replay/rep_end') # Replay mode: go to end.
140         self.replay_pause = xp.findCommand('sim/replay/rep_pause') # Replay mode: pause.
141         self.replay_play = xp.findCommand('sim/replay/rep_play_rf') # Replay mode: play forward.
142         self.pauseCmd =  xp.findCommand('sim/operation/pause_toggle')
143         self.pauseStateDR =  xp.findDataRef('sim/time/paused')    # boolean int, 1 when paused
144
145         # self.norm_factors = [2.0, 2.0, .5, -.02, .15, .15]
146         self.norm_factors = [1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
147
148         self.replay_mode = xp.findDataRef("sim/time/replay_mode");
149
150     def InputOutputLoopCallback(self, elapsedMe, elapsedSim, counter, refcon):
151         telemetry = self.read_telemetry()
152         telemetry_str = ['{:.3f}'.format(x) for x in telemetry]
153         msg = "xplane_telemetry," + ",".join(s for s in telemetry_str)
154         msg =   msg + '\n'
155         #print(self.controller_addr)
156         try:
157             self.udp.send(msg, (UDP_IP, TARGET_PORT))
158         except:
159             pass
160
161         while self.udp.available() > 0:
162             addr, payload = self.udp.get()
163             msg = payload.split(',')
164             cmd = msg[0]
```

```
            #print(msg,cmd)
            if cmd == 'InitComs':
                self.controller_addr.append(addr[0])
                print("Added controller IP address " + addr[0])
            elif cmd == 'Run':
                xp.commandOnce(self.replay_play)
            elif cmd == 'PauseToggle':
                xp.commandOnce(self.pauseCmd)
            elif cmd == 'Pause':
                is_paused = xp.getDataf(self.pauseStateDR)
                print("is_paused={} {}".format(is_paused, is_paused == 1))
                if not is_paused:
                    xp.commandOnce(self.pauseCmd)
            elif cmd == 'Reset':
                 xp.commandOnce(self.go_to_replay_begin)
            elif cmd == 'Replay':
                filepath = msg[1]
                # XPLMDataAccess.XPLMLoadDataFile(XPLMDataAccess.xplm_DataFile_Replay, filepath)
                ret = xp.loadDataFile(xp.DataFile_ReplayMovie, filepath)
                print(ret, msg, filepath)
                # XPLMDataAccess.XPLMSetDatai(self.replay_mode, 1)
            elif cmd == 'Situation':
                filepath = msg[1]
                # XPLMDataAccess.XPLMLoadDataFile(XPLMDataAccess.xplm_DataFile_Replay, filepath)
                ret = xp.loadDataFile(xp.DataFile_Situation, filepath)
                print(ret, msg, filepath)
                # XPLMDataAccess.XPLMSetDatai(self.replay_mode, 1)
            else:
                print(msg)

        if self.IsWidgetVisible != 0:  # Don't update dialog if widget not visible
            for Item in range(len(telemetry_str)):
                xp.setWidgetDescriptor(self.OutputEdit[Item], telemetry_str[Item])

        return 0.025 # callback every 25ms.

    def InputOutputMenuHandler(self, inMenuRef, inItemRef):
        # If menu selected create our widget dialog
        if inItemRef == 1:
```

```
            if self.IsWidgetVisible == 0:
                self.CreateInputOutputWidget(300, 550, 350, 350)
                self.IsWidgetVisible = 1
            else:
                if not xp.isWidgetVisible(self.InputOutputWidget):
                    xp.showWidget(self.InputOutputWidget)

    def CreateInputOutputWidget(self, x, y, w, h):
        x2 = x + w
        y2 = y - h

        # Create the Main Widget window
        self.InputOutputWidget = xp.createWidget(x, y, x2, y2, 1, "Python - Mdx Platform Interface",
                                                 1, 0, xp.WidgetClass_MainWindow)

        # Add Close Box decorations to the Main Widget
        xp.setWidgetProperty(self.InputOutputWidget, xp.Property_MainWindowHasCloseBoxes, 1)

        # Create the Sub Widget window
        InputOutputWindow = xp.createWidget(x + 50, y - 50, x2 - 50, y2 + 50, 1, "",
                                            0, self.InputOutputWidget, xp.WidgetClass_SubWindow)

        # Set the style to sub window
        xp.setWidgetProperty(InputOutputWindow, xp.Property_SubWindowType, xp.SubWindowStyle_SubWindow)

        CaptionText = []
        self.InputEdit = []
        self.OutputEdit = []
        for Item in range(len(self.XformDesc)):
            # Create a text widget

            CaptionText.append(xp.createWidget(x + 60, y - (60 + (Item * 30)), x + 90, y - (82 + (Item * 30)), 1,
                       self.XformDesc[Item], 0, self.InputOutputWidget, xp.WidgetClass_Caption))
            self.OutputEdit.append(xp.createWidget(x + 190, y - (60 + (Item * 30)), x + 270, y - (82 + (Item * 30)), 1,
                       "?", 0, self.InputOutputWidget, xp.WidgetClass_TextField))

        # Register our widget handler
        self.InputOutputHandlerCB = self.InputOutputHandler
        xp.addWidgetCallback(self.InputOutputWidget, self.InputOutputHandlerCB)

```

```python
    def InputOutputHandler(self, inMessage, inWidget, inParam1, inParam2):
        if inMessage == xp.Message_CloseButtonPushed:
            if self.IsWidgetVisible == 1:
                xp.hideWidget(self.InputOutputWidget)
            return 1
        return 0

    def read_telemetry(self):
        try:
            datarefs = []
            for Item in range(self.NumberOfDatarefs):
                datarefs.append( xp.getDataf(self.OutputDataRef[Item]))

            raw_data = tuple(datarefs)
            named_data = transform_refs._make(raw_data) # load namedtuple with values

            # see https://developer.x-plane.com/code-sample/motionplatformdata/
            telemetry = []
            telemetry.append(named_data.DR_g_axil * -1) # surge_accel
            telemetry.append(named_data.DR_g_side * -1) # sway_accel
            telemetry.append(named_data.DR_g_nrml-1) # heave_accel
            telemetry.append(named_data.DR_Prad * -1) # roll_rate
            telemetry.append(named_data.DR_Qrad * -1) # pitch_rate
            #telemetry.append(named_data.DR_Rrad * -1)  # yaw_rate
            #telemetry.append(radians(named_data.DR_phi)) # bank angle
            #telemetry.append(radians(named_data.DR_theta) * -1) # pitch angle

            telemetry.append(named_data.DR_phi) # bank angle
            telemetry.append(named_data.DR_theta * -1) # pitch angle
            telemetry.append(named_data.DR_Rrad * -1)  # yaw_rate

            return telemetry

        except Exception as e:
            xp.log(str(e) + " reading datarefs")
            return [0,0,0,0,0,0,0,0]

```

Appendix 16-Telemetry code of Unity display app

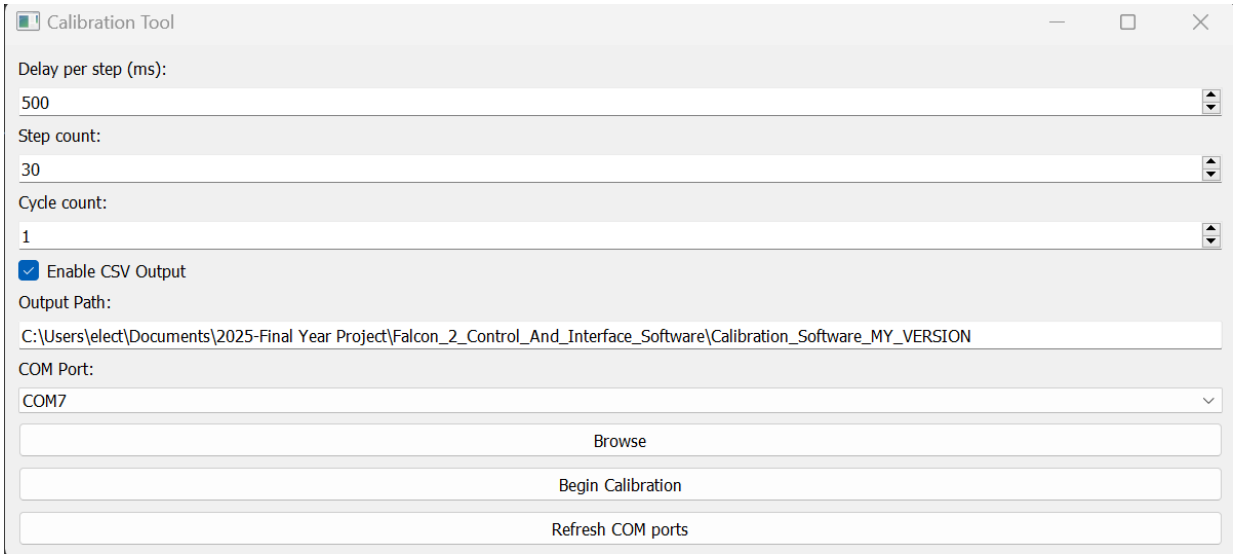
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

using System;
using System.Text;
using System.Net;
using System.Net.Sockets;
using System.Threading;

Unity Script | 0 references
public class Telemetry : MonoBehaviour
{
    private GameObject chair;
    private GameObject prop;

    //public Text transX, transY, transZ;
    //public Text rotX, rotY, rotZ;
    public Text telemetryLabel;

    private Slider adjTransX, adjTransY, adjTransZ;
    private Slider adjRotX, adjRotY, adjRotZ;
    private Text[] xyzrpyValues;
    private Text lblTelemetryIncoming, lblTelemetryOutgoing;
    private Vector3 centerPos = new Vector3(0f, -.15f, .0f);
    private Vector3 parkPos = new Vector3(0f, -1.8f, 0f);
    private float scale = 100; // number of mm in each unity unit of measure
    private float[] defaultLimits = new float[] { 100, 125, 140, 15, 15, 12 }; // mm and degrees for xyzrpy
    private float[] limits = new float[6];

    // Define IP Address and Port
    static public string IP = "127.0.0.1";
    static public int TxToPort   = 10009; // input messages to mdx platform
    static public int RxFromPort = 10020; // output messages from mdx platform
    IPEndPoint rxEndpoint; // for incoming render messages
    IPEndPoint txEndpoint; // for outgoing orientation command messages

    UdpClient client;
    //Thread listener;

    private bool isCommandMode = false;  // this scene provides input to platform controller software
    private bool isRenderMode = false; // this scene will display output from controller software
    private bool isShuttingDown = false;

    // Use this for initialization
    Unity Message | 0 references
    void Start()
    {
        Console.WriteLine("Starting");
        chair = GameObject.Find("ChairAndBase");
        //prop = GameObject.Find("PropExtended");
        //prop.SetActive(false);
```

```
        xyzrpyValues = new Text[6];
        xyzrpyValues[0] = GameObject.Find("transX").GetComponent<Text>();
        xyzrpyValues[1] = GameObject.Find("transY").GetComponent<Text>();
        xyzrpyValues[2] = GameObject.Find("transZ").GetComponent<Text>();
        xyzrpyValues[3] = GameObject.Find("rotX").GetComponent<Text>();
        xyzrpyValues[4] = GameObject.Find("rotY").GetComponent<Text>();
        xyzrpyValues[5] = GameObject.Find("rotZ").GetComponent<Text>();

        lblTelemetryIncoming = GameObject.Find("lblTelemetryIncoming").GetComponent<Text>();
        if (lblTelemetryIncoming != null && lblTelemetryIncoming.text.Length > 0) isRenderMode = true;
        lblTelemetryOutgoing = GameObject.Find("lblTelemetryOutgoing").GetComponent<Text>();
        if (lblTelemetryOutgoing != null && lblTelemetryOutgoing.text.Length > 0) isCommandMode = true;
        print(string.Format("is render mode={0}, isOutput = {1}", isRenderMode, isCommandMode));

        if (isCommandMode)
        {
            adjTransX = GameObject.Find("AdjTransX").GetComponent<Slider>();
            adjTransY = GameObject.Find("AdjTransY").GetComponent<Slider>();
            adjTransZ = GameObject.Find("AdjTransZ").GetComponent<Slider>();
            adjRotX = GameObject.Find("AdjRotX").GetComponent<Slider>();
            adjRotY = GameObject.Find("AdjRotY").GetComponent<Slider>();
            adjRotZ = GameObject.Find("AdjRotZ").GetComponent<Slider>();
            lblTelemetryOutgoing.text = "Telemetry output on port " + TxToPort.ToString();

        }

        if (isRenderMode)
        {
            lblTelemetryIncoming.text = "Telemetry input on port " + RxFromPort.ToString();
        }
        rxEndpoint = new IPEndPoint(IPAddress.Any, RxFromPort);
        txEndpoint = new IPEndPoint(IPAddress.Parse(IP), TxToPort);
        client = new UdpClient(rxEndpoint);
        UpdateOrientation(centerPos, new Vector3(0, 0, 0));
        Array.Copy(defaultLimits, limits,6);
    }

    // Update is called once per frame
    Unity Message | 0 references
    void Update()
    {
        CheckMessages();
    }

    1 reference
    private void CheckMessages()
    {
        Byte[] data = new byte[0];
        try
        {
            if (client != null && client.Available > 0)
            {
                data = client.Receive(ref rxEndpoint);
                string[] msgs = Encoding.ASCII.GetString(data).Split('\n');
                foreach (string msg in msgs)
                {
                    if (msg.Length > 0 && msg[0] != 0)
                    {
                        ProcessMsg(msg);
                        print("Got:" + msg);
                        //Byte[] sendBytes = Encoding.ASCII.GetBytes("Got:" + msg);
                        //SendMessage(sendBytes, rxEndpoint);
                    }
                }
            }
        }
        catch (Exception err)
        {
            Console.WriteLine("UDP recv error:" + err.ToString());
            //client.Close();
            //return;
        }
    }

    6 references
    private void UpdateOrientation(Vector3 localPos, Vector3 localRot)
    {
        chair.transform.localPosition = localPos + centerPos;
        chair.transform.eulerAngles = localRot + new Vector3(180, 0, 0); //Had to add offset for my model (DELETE IF NEEDED!)
        //chair.transform.eulerAngles = new Vector3(180,0,0);
    }

    1 reference
    void ProcessMsg(String msg)
    {
        if (isShuttingDown || !isRenderMode) return;

        char[] sep = { ',', '\n' };
        String[] fields = msg.Split(sep);
        print(msg);
        print(fields.Length);
        if(fields.Length > 6) {
            if (fields[0] == "monitor")
            {

            }
            else if (fields[0] == "normalized")
            {
                try
                {
                    print(fields);
                    if (fields.Length > 6)
                    {
```

```
                    Vector3 localPos;
                    localPos.x = limits[1] * float.Parse(fields[2]) / scale;
                    localPos.y = limits[2] * float.Parse(fields[3]) / scale;
                    localPos.z = limits[0] * float.Parse(fields[1]) / scale;
                    Vector3 localRot;
                    localRot.x = limits[4] * float.Parse(fields[5]);
                    localRot.y = limits[5] * float.Parse(fields[6]);
                    localRot.z = limits[3] * float.Parse(fields[4]);
                    lblTelemetryIncoming.text = msg;
                    for (int i = 0; i < 6; i++)
                    {
                        xyzrpyValues[i].text = fields[i + 1];
                    }
                    UpdateOrientation(localPos, localRot);
                }
            }
            catch (Exception e)
            {
                print(e);
            }
        }
        else if (fields[0] == "request")
        {
            try
            {
                print(fields);
                if (fields.Length > 6)
                {
                    Vector3 localPos;
                    localPos.x = float.Parse(fields[2]) / scale;
                    localPos.y = float.Parse(fields[3]) / scale;
                    localPos.z = float.Parse(fields[1]) / scale;
                    Vector3 localRot;
                    localRot.x = float.Parse(fields[5]) * (float)(180.0 / Math.PI);
                    localRot.y = float.Parse(fields[6]) * (float)(180.0 / Math.PI);
                    localRot.z = float.Parse(fields[4]) * (float)(180.0 / Math.PI);
                    Debug.Log("Rotation: " + localRot.ToString());
                    lblTelemetryIncoming.text = localPos.ToString();

                    //if(fields.Length > 7)
                    //{
                    //    prop.SetActive(int.Parse(fields[7]) == 1);
                    //}


                    if (!isCommandMode && xyzrpyValues != null)
                    {
                        // only display 6DoF values if sliders not active
                        xyzrpyValues[0].text = localPos.z.ToString();
                        xyzrpyValues[1].text = localPos.x.ToString();
                        xyzrpyValues[2].text = localPos.y.ToString();
                        xyzrpyValues[3].text = localRot.z.ToString();
                        xyzrpyValues[4].text = localRot.x.ToString();
                        xyzrpyValues[5].text = localRot.y.ToString();
                    }
                    lblTelemetryIncoming.text = msg;
                    UpdateOrientation(localPos, localRot);
                }
            }
            catch (Exception e)
            {
                print(e);
            }
        }

    }
}

0 references
public void SendMessage(byte[] message, IPEndPoint target)
{
    if (isShuttingDown) return;
    try
    {
        if(client != null)
        {
            client.Send(message, message.Length, target);
        }
    }
    catch (Exception err)
    {
        Console.WriteLine("UDP send error:" + err.ToString());
    }
}

@ Unity Message | 0 references
void OnApplicationQuit()
{
    isShuttingDown = true;
}


/*
 *   functions for slider input control
 */

0 references
public void setIsActive(bool isActive)
{
    if (isShuttingDown) return;
    print(String.Format("isActive set to {0}", isActive));
    byte[] message;
```

```
        if (isActive)
            message = Encoding.ASCII.GetBytes("command,enable,\n");
        else
            message = Encoding.ASCII.GetBytes("command,disable,\n");

        if (client != null)
        {
            client.Send(message, message.Length, txEndpoint);
        }
    }

    0 references
    public void CenterPlatform()
    {
        adjTransX.value = adjTransY.value = adjTransZ.value = 0;
        adjRotX.value = adjRotY.value = adjRotZ.value = 0;
        if (!isRenderMode)
        {
            UpdateOrientation(centerPos, new Vector3(0f, 0f, 0f));
        }

        Debug.Log("Center");
    }

    0 references
    public void ParkPlatform()
    {
        adjTransX.value = adjTransY.value = 0;
        adjTransZ.value = parkPos.y * scale;
        adjRotX.value = adjRotY.value = adjRotZ.value = 0;
        if (!isRenderMode)
        { // only update from slider if output not active
            UpdateOrientation(parkPos, new Vector3(0f, 0f, 0f));
        }

        Debug.Log("Park");
    }

    6 references
    public void UpdateVal(float val, int index)
    {
        print(String.Format("update val {0} = {1}", index, val));
        if (isShuttingDown) return;
        if (xyzrpyValues != null && xyzrpyValues[index] != null)
        {
            xyzrpyValues[index].text = val.ToString(); ;

            if (isShuttingDown) return;
            Vector3 localPos = chair.transform.localPosition;
            Vector3 localRot = chair.transform.localRotation.eulerAngles;
            localPos.x = adjTransY.value / scale;
            localPos.y = adjTransZ.value / scale;
            localPos.z = adjTransX.value / scale;
            //Debug.Log("localPos" + localPos);
            localPos += centerPos;

            localRot.x = adjRotY.value; // positive roll is right side down (sitting in chair)
            localRot.y = -adjRotZ.value;  // positive pitch is nose down
            localRot.z = -adjRotX.value; // positive yaw is ccw

            Debug.Log(localRot);

            if (!isRenderMode)  // only update from slider if output not active
            {
                UpdateOrientation(localPos, localRot);
            }
            SendPositionMessage();
        }
    }

    1 reference
    private void SendPositionMessage()
    {
        if (isShuttingDown) return;
        String msgStr = (string.Format("xyzrpy,{0}, {1}, {2}, {3}, {4}, {5},\n",
                            adjTransX.value, adjTransY.value, adjTransZ.value, adjRotX.value, adjRotY.value, adjRotZ.value));
        Debug.Log(msgStr);
        Byte[] message = Encoding.ASCII.GetBytes(msgStr);
        if (client != null)
        {
            client.Send(message, message.Length, txEndpoint);
        }
    }

    0 references
    public void TransXSliderChanged(float sliderValue) { UpdateVal(sliderValue, 0); }

    0 references
    public void TransYSliderChanged(float sliderValue) { UpdateVal(sliderValue, 1); }

    0 references
    public void TransZSliderChanged(float sliderValue) { UpdateVal(sliderValue, 2); }

    0 references
    public void RotXSliderChanged(float sliderValue)   { UpdateVal(sliderValue, 3); }

    0 references
    public void RotYSliderChanged(float sliderValue)   { UpdateVal(sliderValue, 4); }

    0 references
    public void RotZSliderChanged(float sliderValue)   { UpdateVal(sliderValue, 5);  }
}
```

Appendix 17-calibration_software.py GUI snippet



Appendix 18-calibration_software.py code snippets

```python
from common.serialProcess import SerialProcess  # Import SerialProcess for handling serial communication
from fluidic_muscle import FluidicMuscle  # Import FluidicMuscle module
import csv  # Import CSV for file handling
import pandas as pd  # Import pandas for data manipulation
import sys  # Import sys for system-specific parameters and functions
from PyQt5.QtWidgets import QApplication  # Import PyQt5 for GUI functionality
from GUI import CalibrationApp  # Import the base CalibrationApp class from GUI module
import os  # Import OS for file path handling
from platform_pose import Platform  # Import Platform for controlling the platform
import time  # Import time for delays and timing operations

# Define a new class that extends CalibrationApp
class NewCalibrationApp(CalibrationApp):
    def __init__(self):
        # Call the parent class constructor to inherit properties and UI components
        super().__init__()

        # Define calibration parameters
        self.LOAD = 10  # Default load in kg
        self.STEP_COUNT = 30  # Number of pressure steps
        self.CYCLE_COUNT = 1  # Number of calibration cycles
        self.CURRENT_CYCLE = 0  # Current cycle index
        self.DELAY_PER_STEP_MS = 500  # Delay per step in milliseconds

        # Define pressure settings
        self.MAX_PRESSURE_MBAR = 6000  # Maximum pressure in mbar
        self.PRESSURE_INCREMENT = self.MAX_PRESSURE_MBAR / self.STEP_COUNT  # Pressure increment per step
        self.CURRENT_PRESSURE = self.PRESSURE_INCREMENT  # Start pressure

        # Initialize serial encoder and platform objects
        self.encoder = SerialProcess()
        self.platform = Platform()

        # Create empty DataFrames for calibration curves
        self.P_to_D_up_curve_table = pd.DataFrame(columns=["Pressure (mbar)", "Distance (mm)", "Load(kg)"])
        self.P_to_D_down_curve_table = pd.DataFrame(columns=["Pressure (mbar)", "Distance (mm)", "Load(kg)"])

        # Modify UI button functionality
        self.pushButton_start.setText("Begin Calibration")  # Change button text
        self.pushButton_start.clicked.connect(self.on_start_button_clicked)  # Connect button to custom method

```

```
        # Refresh available COM ports
        self.refresh_COM_ports()
        self.pushButton_refresh.clicked.connect(self.refresh_COM_ports)

        # Set default values for UI elements
        self.spinBox_step_count.setValue(self.STEP_COUNT)
        self.spinBox_cycle_count.setValue(self.CYCLE_COUNT)
        self.spinBox_delay.setValue(self.DELAY_PER_STEP_MS)

        # Connect spinbox changes to update methods
        self.spinBox_step_count.valueChanged.connect(self.update_step_count)
        self.spinBox_cycle_count.valueChanged.connect(self.update_cycle_count)
        self.spinBox_delay.valueChanged.connect(self.update_delay)

        # Set default output path
        self.lineEdit_output_path.setText(os.getcwd())

    # Update functions for UI inputs
    def update_step_count(self):
        """Updates the step count and recalculates pressure increment."""
        self.STEP_COUNT = self.spinBox_step_count.value()
        self.PRESSURE_INCREMENT = self.MAX_PRESSURE_MBAR / self.STEP_COUNT
        print(self.PRESSURE_INCREMENT)

    def update_cycle_count(self):
        """Updates the cycle count when changed in UI."""
        self.CYCLE_COUNT = self.spinBox_cycle_count.value()

    def update_delay(self):
        """Updates the delay per step when changed in UI."""
        self.DELAY_PER_STEP_MS = self.spinBox_delay.value()

    def on_start_button_clicked(self):
        """Handles the start button click event."""
        print("Start button pressed in new app!")
        print(f"Load: {self.LOAD} kg")
        print(f"Delay: {self.DELAY_PER_STEP_MS} ms")
        print(f"Step Count: {self.STEP_COUNT}")
        print(f"Cycle Count: {self.CYCLE_COUNT}")
```

```

        self.label_output_path.setText("Processing...")

        self.platform.muscle.send_pressures([0,0,0,0,0,0])
        time.sleep(self.DELAY_PER_STEP_MS / 1000)

        # Open the selected COM port and retrieve initial load value
        port = self.comboBox_com_port.currentText()
        if port:
            print(port)
            self.encoder.open_port(port, 115200)
            time.sleep(0.1)  # Wait for queue to fill up
            self.encoder.write("R".encode())
            _, load = self.GetDistanceAndLoad()
            self.LOAD = round(load)

        # Start calibration process
        self.Calibrate_Cycles()

    def refresh_COM_ports(self):
        """Refresh the list of available COM ports."""
        self.comboBox_com_port.clear()
        ports = self.encoder.list_ports()
        for p in ports:
            self.comboBox_com_port.addItem(str(p.device))

```

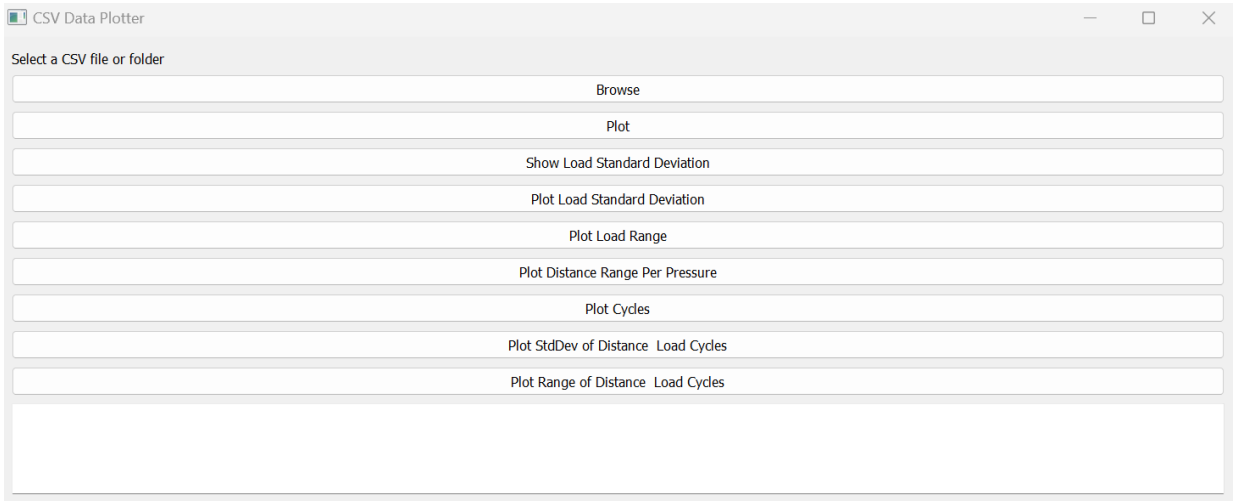
```
    def Calibrate(self):
        #Reset muscle pressure
        """Performs the calibration process over multiple cycles."""
        for cycle in range(self.CYCLE_COUNT):
            self.CURRENT_PRESSURE = self.PRESSURE_INCREMENT
            print(f"Cycle {cycle + 1}/{self.CYCLE_COUNT}")

            # Increasing pressure phase
            while self.CURRENT_PRESSURE <= self.MAX_PRESSURE_MBAR:
                time.sleep(self.DELAY_PER_STEP_MS / 2000)
                extension, load = self.GetDistanceAndLoad()
                time.sleep(self.DELAY_PER_STEP_MS / 2000)
                self.P_to_D_up_curve_table = pd.concat(
                    [self.P_to_D_up_curve_table, pd.DataFrame([{ "Pressure (mbar)": self.CURRENT_PRESSURE, "Distance (mm)": extension,
                                                                 "Load(kg)": load }])], ignore_index=True)
                self.platform.muscle.send_pressures([int(self.CURRENT_PRESSURE), 0, 0, 0, 0, 0])
                self.CURRENT_PRESSURE += self.PRESSURE_INCREMENT

            # Decreasing pressure phase
            self.CURRENT_PRESSURE -= self.PRESSURE_INCREMENT  # Avoid duplicate max pressure reading
            while self.CURRENT_PRESSURE >= self.PRESSURE_INCREMENT:
                time.sleep(self.DELAY_PER_STEP_MS / 2000)
                extension, load = self.GetDistanceAndLoad()
                time.sleep(self.DELAY_PER_STEP_MS / 2000)
                self.P_to_D_down_curve_table = pd.concat(
                    [self.P_to_D_down_curve_table, pd.DataFrame([{ "Pressure (mbar)": self.CURRENT_PRESSURE, "Distance (mm)": extension,
                                                                   "Load(kg)": load }])], ignore_index=True)
                self.platform.muscle.send_pressures([int(self.CURRENT_PRESSURE), 0, 0, 0, 0, 0])
                self.CURRENT_PRESSURE -= self.PRESSURE_INCREMENT

        # Save averaged results to CSV
        P_to_D_up_curve_table_avg = self.P_to_D_up_curve_table.groupby("Pressure (mbar)").mean().reset_index()
        P_to_D_down_curve_table_avg = self.P_to_D_down_curve_table.groupby("Pressure (mbar)").mean().reset_index()

        if(self.checkbox_csv_output.isChecked()):
            P_to_D_up_curve_table_avg.to_csv(f"{self.lineEdit_output_path.text()}/{self.LOAD}kg_up_curve.csv", index=False)
            P_to_D_down_curve_table_avg.to_csv(f"{self.lineEdit_output_path.text()}/{self.LOAD}kg_down_curve.csv", index=False)
            print("SAVED CSV FILES TO OUTPUT FOLDER")
```

```
    def Calibrate_Cycles(self):
        """Performs the calibration process over multiple cycles and stores distinct distance/load data for each cycle."""

        # Reset muscle pressure
        column_names = ["Pressure (mbar)"] + [
            f"Distance Cycle {i+1} (mm)" for i in range(self.CYCLE_COUNT)
        ] + [
            f"Load Cycle {i+1} (kg)" for i in range(self.CYCLE_COUNT)
        ]

        # Initialize DataFrames
        self.P_to_D_up_curve_table = pd.DataFrame(columns=column_names)
        self.P_to_D_down_curve_table = pd.DataFrame(columns=column_names)

        for cycle in range(self.CYCLE_COUNT):
            self.CURRENT_PRESSURE = self.PRESSURE_INCREMENT
            print(f"Cycle {cycle + 1}/{self.CYCLE_COUNT}")

            # Increasing pressure phase
            while self.CURRENT_PRESSURE <= self.MAX_PRESSURE_MBAR:
                time.sleep(self.DELAY_PER_STEP_MS / 2000)
                extension, load = self.GetDistanceAndLoad()
                time.sleep(self.DELAY_PER_STEP_MS / 2000)

                if self.CURRENT_PRESSURE in self.P_to_D_up_curve_table["Pressure (mbar)"].values:
                    row_index = self.P_to_D_up_curve_table.index[self.P_to_D_up_curve_table["Pressure (mbar)"] == self.CURRENT_PRESSURE][0]
                    self.P_to_D_up_curve_table.at[row_index, f"Distance Cycle {cycle+1} (mm)"] = extension
                    self.P_to_D_up_curve_table.at[row_index, f"Load Cycle {cycle+1} (kg)"] = load
                else:
                    new_row = {col: None for col in column_names}
                    new_row["Pressure (mbar)"] = self.CURRENT_PRESSURE
                    new_row[f"Distance Cycle {cycle+1} (mm)"] = extension
                    new_row[f"Load Cycle {cycle+1} (kg)"] = load
                    self.P_to_D_up_curve_table = pd.concat(
                        [self.P_to_D_up_curve_table, pd.DataFrame([new_row])], ignore_index=True
                    )
```

```
                self.platform.muscle.send_pressures([int(self.CURRENT_PRESSURE), 0, 0, 0, 0, 0])
                self.CURRENT_PRESSURE += self.PRESSURE_INCREMENT

            # Decreasing pressure phase
            self.CURRENT_PRESSURE -= self.PRESSURE_INCREMENT  # Avoid duplicate max pressure reading
            while self.CURRENT_PRESSURE >= self.PRESSURE_INCREMENT:
                time.sleep(self.DELAY_PER_STEP_MS / 2000)
                extension, load = self.GetDistanceAndLoad()
                time.sleep(self.DELAY_PER_STEP_MS / 2000)

                if self.CURRENT_PRESSURE in self.P_to_D_down_curve_table["Pressure (mbar)"].values:
                    row_index = self.P_to_D_down_curve_table.index[self.P_to_D_down_curve_table["Pressure (mbar)"] == self.CURRENT_PRESSURE][0]
                    self.P_to_D_down_curve_table.at[row_index, f"Distance Cycle {cycle+1} (mm)"] = extension
                    self.P_to_D_down_curve_table.at[row_index, f"Load Cycle {cycle+1} (kg)"] = load
                else:
                    new_row = {col: None for col in column_names}
                    new_row["Pressure (mbar)"] = self.CURRENT_PRESSURE
                    new_row[f"Distance Cycle {cycle+1} (mm)"] = extension
                    new_row[f"Load Cycle {cycle+1} (kg)"] = load
                    self.P_to_D_down_curve_table = pd.concat(
                        [self.P_to_D_down_curve_table, pd.DataFrame([new_row])], ignore_index=True
                    )

                self.platform.muscle.send_pressures([int(self.CURRENT_PRESSURE), 0, 0, 0, 0, 0])
                self.CURRENT_PRESSURE -= self.PRESSURE_INCREMENT

        # Compute the averaged results across cycles
        #P_to_D_up_curve_table_avg = self.P_to_D_up_curve_table.groupby("Pressure (mbar)").mean().reset_index()
        #P_to_D_down_curve_table_avg = self.P_to_D_down_curve_table.groupby("Pressure (mbar)").mean().reset_index()

        # Save CSV files if checkbox is checked
        if self.checkbox_csv_output.isChecked():
            self.P_to_D_up_curve_table.to_csv(f"{self.lineEdit_output_path.text()}/{self.LOAD}kg_up_curve.csv", index=False)
            self.P_to_D_down_curve_table.to_csv(f"{self.lineEdit_output_path.text()}/{self.LOAD}kg_down_curve.csv", index=False)
            print("SAVED CSV FILES TO OUTPUT FOLDER")

    def GetDistanceAndLoad(self):
        """Retrieves extension and load values from the encoder."""
        data = self.encoder.read()
        print(data)
        if data is None:
            print("function does not function!")
            return (0.0,0.0)
        array = data.split(",")
        return (-float(array[1]), float(array[3]) / 9.81) #extension (mm) and load (kg)

    def Calibrate_With_Noise(self):
        """Performs calibration with multiple snapshots for noise analysis."""
        SNAPSHOT_COUNT = 100  # Hardcoded number of snapshots per step

        for cycle in range(self.CYCLE_COUNT):
            self.CURRENT_PRESSURE = self.PRESSURE_INCREMENT
            print(f"Cycle {cycle + 1}/{self.CYCLE_COUNT}")

            # Increasing pressure phase
            while self.CURRENT_PRESSURE <= self.MAX_PRESSURE_MBAR:
                time.sleep(self.DELAY_PER_STEP_MS / 1000)

                # Capture multiple snapshots
                snapshots = [self.GetDistanceAndLoad() for _ in range(SNAPSHOT_COUNT)]
                distances, loads = zip(*snapshots)
                avg_distance = sum(distances) / SNAPSHOT_COUNT
                avg_load = sum(loads) / SNAPSHOT_COUNT

                # Calculate snapshot ranges
                distance_range = max(distances) - min(distances)
                load_range = max(loads) - min(loads)

                # Store data in DataFrame
                row_data = {"Pressure (mbar)": self.CURRENT_PRESSURE,
                            "Distance (mm)": avg_distance,
                            "Load(kg)": avg_load,
                            "Distance Range (mm)": distance_range,  # Add snapshot range for Distance
                            "Load Range (kg)": load_range}  # Add snapshot range for Load
```

```
                # Adding individual snapshot values to row data
                for i in range(SNAPSHOT_COUNT):
                    row_data[f"Distance Snapshot {i+1}"] = distances[i]
                    row_data[f"Load Snapshot {i+1}"] = loads[i]

                self.P_to_D_up_curve_table = pd.concat([self.P_to_D_up_curve_table, pd.DataFrame([row_data])], ignore_index=True)

                self.platform.muscle.send_pressures([int(self.CURRENT_PRESSURE), 0, 0, 0, 0, 0])
                self.CURRENT_PRESSURE += self.PRESSURE_INCREMENT

            # Decreasing pressure phase
            self.CURRENT_PRESSURE -= self.PRESSURE_INCREMENT  # Avoid duplicate max pressure reading
            while self.CURRENT_PRESSURE >= self.PRESSURE_INCREMENT:
                time.sleep(self.DELAY_PER_STEP_MS / 1000)

                snapshots = [self.GetDistanceAndLoad() for _ in range(SNAPSHOT_COUNT)]
                distances, loads = zip(*snapshots)
                avg_distance = sum(distances) / SNAPSHOT_COUNT
                avg_load = sum(loads) / SNAPSHOT_COUNT

                # Calculate snapshot ranges
                distance_range = max(distances) - min(distances)
                load_range = max(loads) - min(loads)

                row_data = {"Pressure (mbar)": self.CURRENT_PRESSURE,
                            "Distance (mm)": avg_distance,
                            "Load(kg)": avg_load,
                            "Distance Range (mm)": distance_range,  # Add snapshot range for Distance
                            "Load Range (kg)": load_range}  # Add snapshot range for Load

                # Adding individual snapshot values to row data
                for i in range(SNAPSHOT_COUNT):
                    row_data[f"Distance Snapshot {i+1}"] = distances[i]
                    row_data[f"Load Snapshot {i+1}"] = loads[i]

                self.P_to_D_down_curve_table = pd.concat([self.P_to_D_down_curve_table, pd.DataFrame([row_data])], ignore_index=True)

                self.platform.muscle.send_pressures([int(self.CURRENT_PRESSURE), 0, 0, 0, 0, 0])
                self.CURRENT_PRESSURE -= self.PRESSURE_INCREMENT

        # Save averaged results to CSV
        P_to_D_up_curve_table_avg = self.P_to_D_up_curve_table.groupby("Pressure (mbar)").mean().reset_index()
        P_to_D_down_curve_table_avg = self.P_to_D_down_curve_table.groupby("Pressure (mbar)").mean().reset_index()

        if self.checkbox_csv_output.isChecked():
            P_to_D_up_curve_table_avg.to_csv(f"{self.lineEdit_output_path.text()}/{self.LOAD}kg_up_curve_with_noise.csv", index=False)
            P_to_D_down_curve_table_avg.to_csv(f"{self.lineEdit_output_path.text()}/{self.LOAD}kg_down_curve_with_noise.csv", index=False)
            print("SAVED CSV FILES WITH NOISE DATA TO OUTPUT FOLDER")

if __name__ == "__main__":
    app = QApplication([])
    window = NewCalibrationApp()
    window.show()
    app.exec_()
```

Appendix 19-csv_muscle_plotter.py GUI



Appendix 20-csv_muscle_plotter.py code snippets

```python
import sys  # System-specific parameters and functions
import os  # Provides functions for interacting with the operating system
import pandas as pd  # Library for data manipulation and analysis
import matplotlib.pyplot as plt  # Library for creating static, animated, and interactive visualizations
import matplotlib.cm as cm
from PyQt5.QtWidgets import QApplication, QWidget, QVBoxLayout, QPushButton, QFileDialog, QLabel, QTextEdit  # PyQt5 modules for GUI development
import re

class CSVPlotter(QWidget):  # Defines a class that inherits from QWidget for creating a GUI application
    def __init__(self):
        super().__init__()  # Calls the parent class constructor
        self.initUI()  # Initializes the user interface

    def initUI(self):
        self.CMAP = "viridis"
        layout = QVBoxLayout()  # Creates a vertical box layout

        self.label = QLabel("Select a CSV file or folder")  # Label to display selected file/folder
        layout.addWidget(self.label)  # Adds label to layout

        self.btn_browse = QPushButton("Browse", self)  # Button for browsing files/folders
        self.btn_browse.clicked.connect(self.browse)  # Connects button click event to browse method
        layout.addWidget(self.btn_browse)  # Adds button to layout

        self.btn_plot = QPushButton("Plot", self)  # Button for plotting data
        self.btn_plot.clicked.connect(self.plot_data)  # Connects button click event to plot_data method
        layout.addWidget(self.btn_plot)  # Adds button to layout

        self.btn_stddev = QPushButton("Show Load Standard Deviation", self)  # Button to show standard deviation for Load
        self.btn_stddev.clicked.connect(self.show_stddev)  # Connects button click event to show_stddev method
        layout.addWidget(self.btn_stddev)  # Adds button to layout

        self.btn_plot_stddev = QPushButton("Plot Load Standard Deviation", self)  # Button to plot standard deviation for Load
        self.btn_plot_stddev.clicked.connect(self.plot_stddev)  # Connects button click event to plot_stddev method
        layout.addWidget(self.btn_plot_stddev)  # Adds button to layout

        self.btn_plot_range = QPushButton("Plot Load Range", self)  # Button to plot load range
        self.btn_plot_range.clicked.connect(self.plot_load_range)  # Connect to method
        layout.addWidget(self.btn_plot_range)  # Add button to layout

        self.btn_plot_noise = QPushButton("Plot Distance Range Per Pressure", self)  # Button to plot noise data
        self.btn_plot_noise.clicked.connect(self.plot_noise)  # Connect to method
        layout.addWidget(self.btn_plot_noise)  # Add button to layout

        self.btn_plot_noise = QPushButton("Plot Cycles", self)  # Button to plot noise data
        self.btn_plot_noise.clicked.connect(self.plot_cycle_data)  # Connect to method
        layout.addWidget(self.btn_plot_noise)  # Add button to layout

        self.btn_plot_stddev_cycles = QPushButton("Plot StdDev of Distance & Load Cycles", self)
        self.btn_plot_stddev_cycles.clicked.connect(self.plot_standard_deviation_cycles)
        layout.addWidget(self.btn_plot_stddev_cycles)

        self.btn_plot_range_cycles = QPushButton("Plot Range of Distance & Load Cycles", self)
        self.btn_plot_range_cycles.clicked.connect(self.plot_range_cycles)
        layout.addWidget(self.btn_plot_range_cycles)

        self.text_output = QTextEdit(self)  # Text box to display standard deviation results
        self.text_output.setReadOnly(True)
        layout.addWidget(self.text_output)

        self.setLayout(layout)  # Sets the layout for the window
        self.setWindowTitle("CSV Data Plotter")  # Sets the window title
        self.setGeometry(200, 200, 500, 300)  # Defines the size and position of the window

        self.file_paths = []  # Initializes an empty list to store selected file(s)

    def browse(self):
        options = QFileDialog.Options()  # File dialog options
        path = QFileDialog.getExistingDirectory(self, "Select Folder", "", options=options)  # Opens a folder selection dialog

        if path:  # If a folder is selected
            self.file_paths = self.find_csv_files(path)  # Finds all CSV files in the selected folder
            self.label.setText(f"Selected Folder: {path}")  # Updates label text
        else:  # If a file is selected
            file_path, _ = QFileDialog.getOpenFileName(self, "Select CSV File", "", "CSV Files (*.csv)", options=options)  # Opens a file selection dialog
            if file_path:
                self.file_paths = self.find_csv_pairs(file_path)  # Finds corresponding up/down curve pairs
                self.label.setText(f"Selected File: {file_path}")  # Updates label text

    def find_csv_files(self, folder_path):
        """Find all valid CSV files in the folder."""
        csv_files = [os.path.join(folder_path, f) for f in os.listdir(folder_path) if f.endswith(".csv")]  # Gets all CSV files in the folder
        return csv_files  # Returns the list of CSV file paths

    def find_csv_pairs(self, file_path):
        """Find the corresponding up/down curve pair if a single file is selected."""
        base_name = os.path.basename(file_path)  # Extracts the file name
        folder = os.path.dirname(file_path)  # Extracts the folder path
```

```
        if "up_curve" in base_name:  # Checks if the file is an up curve file
            paired_file = base_name.replace("up_curve", "down_curve")  # Finds corresponding down curve file
        elif "down_curve" in base_name:  # Checks if the file is a down curve file
            paired_file = base_name.replace("down_curve", "up_curve")  # Finds corresponding up curve file
        else:
            return [file_path]  # Returns the selected file if no pair is found

        paired_path = os.path.join(folder, paired_file)  # Constructs the full path of the paired file
        return [file_path, paired_path] if os.path.exists(paired_path) else [file_path]  # Returns both files if the pair exists

    def show_stddev(self):
        if not self.file_paths:
            self.text_output.setText("No valid CSV files selected!")
            return

        result_text = "Standard Deviation of Load Column:\n"

        for file in self.file_paths:
            try:
                df = pd.read_csv(file)
                if "Load(kg)" in df.columns:
                    stddev = df["Load(kg)"].std()
                    result_text += f"\nFile: {os.path.basename(file)}\nLoad Standard Deviation: {stddev:.4f}\n"
                else:
                    result_text += f"\nFile: {os.path.basename(file)}\nLoad column not found!\n"
            except Exception as e:
                result_text += f"\nError reading {file}: {e}\n"

        self.text_output.setText(result_text)

    def plot_stddev(self):
        if not self.file_paths:
            self.text_output.setText("No valid CSV files selected!")
            return

        file_names = []
        stddev_values = []
        base_names = []  # Stores unique load identifiers
        color_map = {}  # Maps base names to colors

        # Extract base names and compute standard deviations
        for file in self.file_paths:
            try:
                df = pd.read_csv(file)
                if "Load(kg)" in df.columns:
                    stddev = df["Load(kg)"].std()
                    file_name = os.path.basename(file)

                    # Extract base name (removing "up_curve" or "down_curve")
                    base_name = file_name.replace("up_curve", "").replace("down_curve", "").replace(".csv", "")

                    file_names.append(file_name)
                    stddev_values.append(stddev)
                    base_names.append(base_name)
            except Exception as e:
                print(f"Error reading {file}: {e}")

        if not file_names:
            self.text_output.setText("No valid CSV files found!")
            return

        # Generate unique colors for each base name
        unique_base_names = list(set(base_names))
        cmap = cm.get_cmap(self.CMAP, len(unique_base_names))  # Use a colormap
        colors = {name: cmap(i) for i, name in enumerate(unique_base_names)}  # Assign colors

        # Plot bars with shared colors for each base name
        plt.figure(figsize=(10, 6))
        bars = [plt.bar(file_names[i], stddev_values[i], color=colors[base_names[i]]) for i in range(len(file_names))]

        # Create a legend mapping base names to colors
        legend_handles = [plt.Rectangle((0, 0), 1, 1, color=colors[name]) for name in unique_base_names]
        plt.legend(legend_handles, unique_base_names, title="Load Groups", loc="upper left")

        plt.xlabel("CSV Files")
        plt.ylabel("Standard Deviation of Load (kg)")
        plt.title("Load Standard Deviation for Each CSV File")
        plt.xticks(rotation=45, ha="right")
        plt.tight_layout()
        plt.show()

```

```python
    def plot_cycle_data(self):
        """Plots all Distance and Load Cycles vs Pressure for multiple CSV files."""

        if not self.file_paths:
            self.label.setText("No valid CSV files selected!")
            return

        plt.figure()

        # Create figure and primary axis for Pressure vs Distance Cycles
        fig, ax1 = plt.subplots(figsize=(10, 6))
        ax2 = ax1.twinx()  # Secondary y-axis for Load Cycles

        base_names = []
        data = {}

        # Extract base names and group files
        for file in self.file_paths:
            try:
                file_name = os.path.basename(file)
                base_name = file_name.replace("up_curve", "").replace("down_curve", "").replace(".csv", "").strip()

                if base_name not in data:
                    data[base_name] = []
                data[base_name].append(file)

                if base_name not in base_names:
                    base_names.append(base_name)
            except Exception as e:
                print(f"Error processing {file}: {e}")

        # Define explicit contrasting colors for up/down curves
        color_pairs = {
            "12kg_": ("red", "blue"),
            "13kg_": ("green", "purple"),
            "24kg_": ("orange", "cyan"),
            "33kg_": ("brown", "magenta"),
            "43kg_": ("yellow", "purple")
        }

        max_load_value = 0  # Track max load value for scaling

        name = ""
        # Loop through grouped data
        for base_name, files in data.items():
            name = base_name
            if base_name not in color_pairs:
                print(f"Warning: No predefined color pair for {base_name}, skipping.")
                continue

            up_color, down_color = color_pairs[base_name]

            for file in files:
                try:
                    df = pd.read_csv(file)
                    label_suffix = "Up" if "up_curve" in file else "Down"
                    base_color = up_color if "up_curve" in file else down_color

                    # Extract Distance and Load cycle columns dynamically
                    distance_cols = sorted(
                        [col for col in df.columns if "Distance Cycle" in col],
                        key=lambda x: int(x.split("Cycle ")[-1].split(" ")[0]),
                        reverse=True  # Ensure Cycle 5 is first (darkest)
                    )
                    load_cols = sorted(
                        [col for col in df.columns if "Load Cycle" in col],
                        key=lambda x: int(x.split("Cycle ")[-1].split(" ")[0]),
                        reverse=True
                    )

                    # Update max load value
                    max_load_value = max(max_load_value, df[load_cols].values.max())

                    # Generate shades of base color for cycles (lightest to darkest)
                    distance_shades = [plt.cm.Reds(i / len(distance_cols) + 0.2) if "up_curve" in file else plt.cm.Blues(i / len(distance_cols) + 0.2)
                                       for i in range(len(distance_cols))]
                    load_shades = [plt.cm.Reds(i / len(load_cols) + 0.2) if "up_curve" in file else plt.cm.Blues(i / len(load_cols) + 0.2)
                                   for i in range(len(load_cols))]

                    # Plot each Distance cycle on the primary y-axis
                    for i, cycle_col in enumerate(distance_cols):
                        ax1.plot(df["Pressure (mbar)"], df[cycle_col],
                                 label=f"{base_name} {cycle_col} ({label_suffix})",
                                 marker="o", linestyle="-", color=distance_shades[len(distance_shades)-i-1], alpha=0.8)
```

```
                    # Plot each Load cycle on the secondary y-axis
                    for i, cycle_col in enumerate(load_cols):
                        ax2.plot(df["Pressure (mbar)"], df[cycle_col],
                                 label=f"{base_name} {cycle_col} ({label_suffix})",
                                 marker="x", linestyle=(0, (1, 1)), color=load_shades[len(load_shades)-i-1], alpha=0.8)  # Fine dotted line

                except Exception as e:
                    print(f"Error reading {file}: {e}")

        name = name.replace("_", "")
        # Configure axis labels and title
        ax1.set_xlabel("Pressure (mbar)", fontsize=20)
        ax1.set_ylabel("Distance (mm)", color="tab:blue", fontsize=20)
        ax2.set_ylabel("Load (kg)", color="tab:red", fontsize=20)
        ax1.set_title(f"Pressure vs Distance and Load Cycles {name}", fontsize=30)

        # Set load axis limit with a margin
        ax2.set_ylim(0, max_load_value * 1.1)  # 10% margin above max load

        # Combine legends from both axes
        ax1_handles, ax1_labels = ax1.get_legend_handles_labels()
        ax2_handles, ax2_labels = ax2.get_legend_handles_labels()
        ax1.legend(ax1_handles + ax2_handles, ax1_labels + ax2_labels, loc="upper left", fontsize="small", bbox_to_anchor=(-0.2, 1))

        # Show grid for better readability
        ax1.grid(True)

        # Show the plot
        plt.tight_layout()
        plt.show()



    def plot_load_range(self):
        if not self.file_paths:
            self.text_output.setText("No valid CSV files selected!")
            return

        file_names = []
        load_ranges = []
        file_base_names = []  # List to track base names for each file

        data = {}

        # Extract base names and group files
        for file in self.file_paths:
            try:
                file_name = os.path.basename(file)
                base_name = file_name.replace("up_curve", "").replace("down_curve", "").replace(".csv", "")

                if base_name not in data:
                    data[base_name] = []
                data[base_name].append(file)
            except Exception as e:
                print(f"Error processing {file}: {e}")

        # Generate unique colors for each base name
        base_names = list(data.keys())  # Get all unique base names
        cmap = cm.get_cmap(self.CMAP, len(base_names))  # Assign unique colors
        colors = {name: cmap(i) for i, name in enumerate(base_names)}

        # Extract load ranges
        for base_name, files in data.items():
            for file in files:
                try:
                    df = pd.read_csv(file)
                    if "Load(kg)" in df.columns:
                        load_range = df["Load(kg)"].max() - df["Load(kg)"].min()
                        file_names.append(os.path.basename(file))  # Store filename
                        load_ranges.append(load_range)  # Store load range
                        file_base_names.append(base_name)  # Track base name for color mapping
                except Exception as e:
                    print(f"Error reading {file}: {e}")

        if not file_names:
            self.text_output.setText("No valid CSV files found!")
            return

        # Plot bars with shared colors for each base name
        plt.figure(figsize=(10, 6))
        bars = [plt.bar(file_names[i], load_ranges[i], color=colors[file_base_names[i]]) for i in range(len(file_names))]

        # Create a legend mapping base names to colors
        legend_handles = [plt.Rectangle((0, 0), 1, 1, color=colors[name]) for name in base_names]
        plt.legend(legend_handles, base_names, title="Load Groups", loc="upper left")

        plt.xlabel("CSV Files")
        plt.ylabel("Load Range (kg)")
        plt.title("Load Range for Each CSV File")
        plt.xticks(rotation=45, ha="right")
```

```
        plt.tight_layout()
        plt.show()

    def plot_data(self):
        if not self.file_paths:
            self.label.setText("No valid CSV files selected!")
            return

        plt.figure()

        # Create figure and primary axis for Pressure vs Distance
        fig, ax1 = plt.subplots(figsize=(10, 6))
        ax2 = ax1.twinx()  # Secondary y-axis for Load

        base_names = []
        data = {}

        # Extract base names and group files
        for file in self.file_paths:
            try:
                file_name = os.path.basename(file)

                # Remove "_up_curve_averaged_x__y" and "_down_curve_averaged_x__y" portions
                base_name = re.sub(r"_(up_curve|down_curve)_averaged_\d+__\d+", "", file_name)
                base_name = base_name.replace(".csv", "").strip()

                if base_name not in data:
                    data[base_name] = []
                data[base_name].append(file)

                if base_name not in base_names:
                    base_names.append(base_name)
            except Exception as e:
                print(f"Error processing {file}: {e}")

        # Define explicit contrasting colors for up/down curves
        color_pairs = {
            "13kg": ("red", "orange"),
            "24kg": ("blue", "cyan"),
            "33kg": ("yellow", "gold"),
            "43kg": ("black", "red")
        }
```

```
        # Loop through grouped data
        for base_name, files in data.items():
            if base_name not in color_pairs:
                print(f"Warning: No predefined color pair for {base_name}, skipping.")
                continue

            up_color, down_color = color_pairs[base_name]

            for file in files:
                try:
                    df = pd.read_csv(file)
                    is_up_curve = "up_curve" in file
                    label_suffix = "Up" if is_up_curve else "Down"
                    base_color = up_color if is_up_curve else down_color

                    # Plot Pressure vs Distance on primary y-axis (solid line)
                    ax1.plot(df["Pressure (mbar)"], df["Distance (mm)"],
                             label=f"{base_name} Distance ({label_suffix})",
                             marker="o", linestyle="-", color=base_color, alpha=0.8)

                    # Plot Pressure vs Load on secondary y-axis (dashed line with x markers)
                    ax2.plot(df["Pressure (mbar)"], df["Load(kg)"],
                             label=f"{base_name} Load ({label_suffix})",
                             marker="x", linestyle="--", color=base_color, alpha=0.8)

                except Exception as e:
                    print(f"Error reading {file}: {e}")

        # Configure axis labels and title
        ax1.set_xlabel("Pressure (mbar)", fontsize=20)
        ax1.set_ylabel("Distance (mm)", color="tab:blue", fontsize=20)
        ax2.set_ylabel("Load (kg)", color="tab:red", fontsize=20)
        ax1.set_title("Pressure vs Distance and Load (Up and Down Curves for Cycles 3 to 5)", fontsize=30)

        # Set load axis limit with a margin
        ax2.set_ylim(0, 50)

        # Combine legends from both axes
        ax1_handles, ax1_labels = ax1.get_legend_handles_labels()
        ax2_handles, ax2_labels = ax2.get_legend_handles_labels()
        ax1.legend(ax1_handles + ax2_handles, ax1_labels + ax2_labels, loc="upper left", fontsize="small", bbox_to_anchor=(-0.2, 1))

        # Show grid for better readability
        ax1.grid(True)
```

```
        # Show the plot
        plt.tight_layout()
        plt.show()

    def plot_noise(self):
        if not self.file_paths:
            self.text_output.setText("No valid CSV files selected!")
            return

        plt.figure(figsize=(10, 6))  # Create a new figure for plotting
        colors = cm.get_cmap(self.CMAP, len(self.file_paths))  # Generate colors per file
        file_colors = {}  # Dictionary to track colors assigned per file

        for i, file in enumerate(self.file_paths):
            try:
                df = pd.read_csv(file)

                # Ensure that the necessary columns are present
                if "Pressure (mbar)" in df.columns and "Distance Range (mm)" in df.columns:
                    pressure = df["Pressure (mbar)"].values
                    distance_range = df["Distance Range (mm)"].values

                    # Assign a unique color per file
                    color = colors(i)
                    file_name = os.path.basename(file)

                    # Determine if it's an "Up Curve" or "Down Curve"
                    curve_type = "Up Curve" if "up_curve" in file_name else "Down Curve"

                    # Avoid duplicate legend labels (one per file)
                    if file_name not in file_colors:
                        plt.plot(pressure, distance_range, 'o-', color=color, label=curve_type)
                        file_colors[file_name] = color  # Store assigned color
                    else:
                        plt.plot(pressure, distance_range, 'o-', color=color)  # No label to avoid duplicate

            except Exception as e:
                print(f"Error reading {file}: {e}")

        # Set plot labels and title
        plt.xlabel("Pressure (mbar)")
        plt.ylabel("Distance Range (mm)")
        plt.title("Distance Range vs Pressure for Each Curve")
```

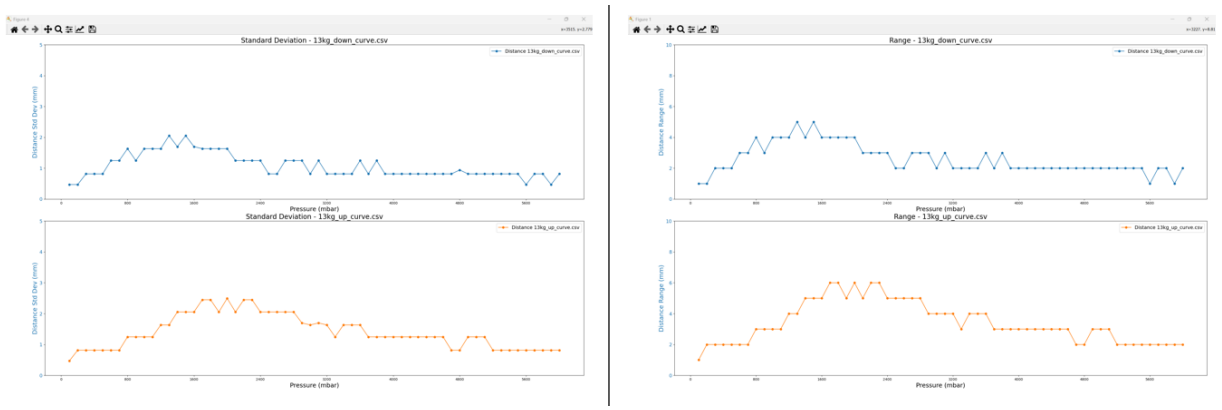
```
        # Display the legend with only "Up Curve" and "Down Curve" once
        plt.legend(loc="upper left")

        # Add grid for better readability
        plt.grid(True)

        # Adjust layout to avoid clipping
        plt.tight_layout()

        # Show the plot
        plt.show()

    def plot_range_cycles(self):
        """Plots the standard deviation of Distance and Load across 5 cycles for each pressure setting."""
        if not self.file_paths:
            self.text_output.setText("No valid CSV files selected!")
            return

        plt.figure(figsize=(10, 6))

        fig, ax1 = plt.subplots(figsize=(10, 6))
        ax2 = ax1.twinx()  # Secondary y-axis for Load Standard Deviation


        # Extract unique base filenames
        base_names = [os.path.basename(file) for file in self.file_paths]

        # Generate unique colors for each base name using a colormap
        cmap = cm.get_cmap(self.CMAP, len(base_names))  # Use the specified colormap
        colors = {name: cmap(i) for i, name in enumerate(base_names)}

        for file in self.file_paths:
            try:
                df = pd.read_csv(file)
                file_name = os.path.basename(file)

                # Identify distance and load cycle columns
                distance_cols = [col for col in df.columns if "Distance Cycle" in col]
                load_cols = [col for col in df.columns if "Load Cycle" in col]

                # Ensure there are exactly 5 cycles
                if len(distance_cols) != 5 or len(load_cols) != 5:
                    print(f"Skipping {file_name}: Expected 5 cycles, found {len(distance_cols)} distance and {len(load_cols)} load cycles.")
                    continue
```

```
                # Compute range for each pressure setting
                distance_range = df[distance_cols].max(axis=1) - df[distance_cols].min(axis=1)
                load_range = df[load_cols].max(axis=1) - df[load_cols].min(axis=1)

                # Extract Pressure column
                pressure = df["Pressure (mbar)"]

                # Assign a unique color to this file
                color = colors[file_name]

                # Plot Range of Distance
                ax1.plot(pressure, distance_range, marker="o", linestyle="-", label=f"{file_name} Distance Range", color=color)

                # Plot Range of Load
                ax2.plot(pressure, load_range, marker="x", linestyle="--", label=f"{file_name} Load Range", color=color)

            except Exception as e:
                print(f"Error processing {file_name}: {e}")

        # Configure axes labels and title
        ax1.set_xlabel("Pressure (mbar)")
        ax1.set_ylabel("Distance Range (mm)", color="blue")
        ax2.set_ylabel("Load Range (kg)", color="red")
        ax1.set_title("Range of Distance & Load across 5 Cycles")

        # Combine legends from both axes
        ax1_handles, ax1_labels = ax1.get_legend_handles_labels()
        ax2_handles, ax2_labels = ax2.get_legend_handles_labels()
        ax1.legend(ax1_handles + ax2_handles, ax1_labels + ax2_labels, loc="upper left")

        ax1.grid(True)
        plt.tight_layout()
        plt.show()

    def plot_standard_deviation_cycles(self):
        """Plots the standard deviation of Distance and Load across 5 cycles for each pressure setting."""
        if not self.file_paths:
            self.text_output.setText("No valid CSV files selected!")
            return

        plt.figure(figsize=(10, 6))

        fig, ax1 = plt.subplots(figsize=(10, 6))
        ax2 = ax1.twinx()  # Secondary y-axis for Load Standard Deviation
```

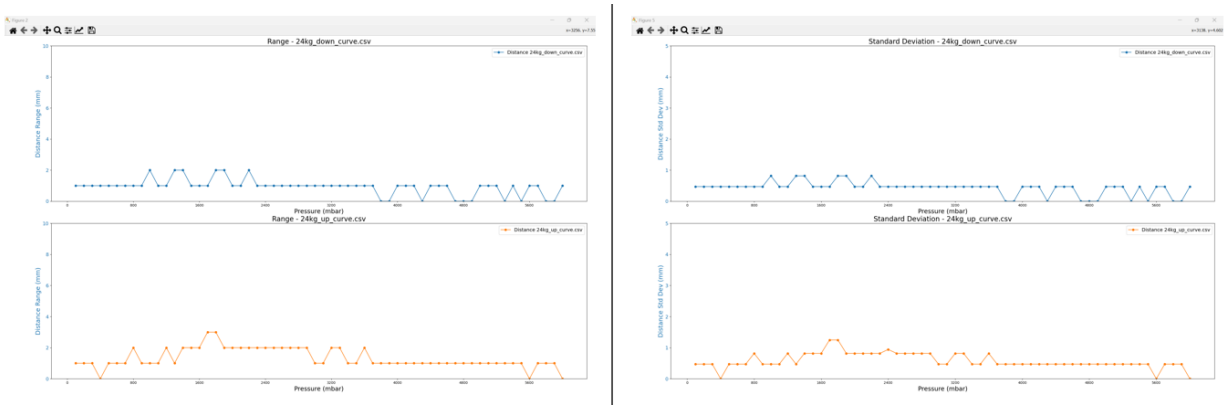
```
        # Extract unique base filenames
        base_names = [os.path.basename(file) for file in self.file_paths]

        # Generate unique colors for each base name using a colormap
        cmap = cm.get_cmap(self.CMAP, len(base_names))  # Use the specified colormap
        colors = {name: cmap(i) for i, name in enumerate(base_names)}

        for file in self.file_paths:
            try:
                df = pd.read_csv(file)
                file_name = os.path.basename(file)

                # Identify distance and load cycle columns
                distance_cols = [col for col in df.columns if "Distance Cycle" in col]
                load_cols = [col for col in df.columns if "Load Cycle" in col]

                # Ensure there are exactly 5 cycles
                if len(distance_cols) != 5 or len(load_cols) != 5:
                    print(f"Skipping {file_name}: Expected 5 cycles, found {len(distance_cols)} distance and {len(load_cols)} load cycles.")
                    continue

                # Compute standard deviation for each pressure setting
                distance_std = df[distance_cols].std(axis=1)
                load_std = df[load_cols].std(axis=1)

                # Extract Pressure column
                pressure = df["Pressure (mbar)"]

                # Assign a unique color to this file
                color = colors[file_name]

                # Plot Standard Deviation of Distance
                ax1.plot(pressure, distance_std, marker="o", linestyle="-", label=f"{file_name} Distance StdDev", color=color)

                # Plot Standard Deviation of Load
                ax2.plot(pressure, load_std, marker="x", linestyle="--", label=f"{file_name} Load StdDev", color=color)

            except Exception as e:
                print(f"Error processing {file}: {e}")

        # Configure axes labels and title
        ax1.set_xlabel("Pressure (mbar)")
        ax1.set_ylabel("Distance Standard Deviation (mm)", color="blue")
        ax2.set_ylabel("Load Standard Deviation (kg)", color="red")
```

```
        ax1.set_title("Standard Deviation of Distance & Load across 5 Cycles")

        # Combine legends from both axes
        ax1_handles, ax1_labels = ax1.get_legend_handles_labels()
        ax2_handles, ax2_labels = ax2.get_legend_handles_labels()
        ax1.legend(ax1_handles + ax2_handles, ax1_labels + ax2_labels, loc="upper left")

        ax1.grid(True)
        plt.tight_layout()
        plt.show()


if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = CSVPlotter()
    window.show()
    sys.exit(app.exec_())
```

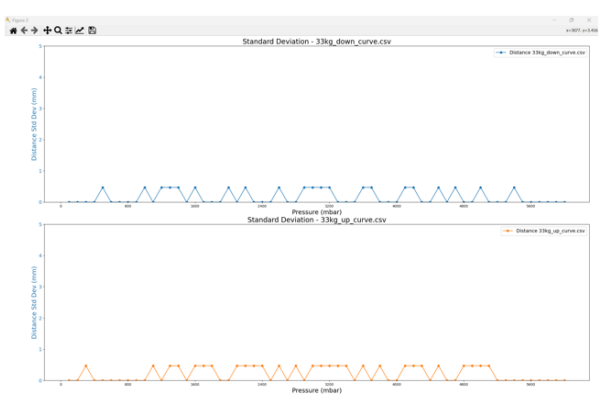
Appendix 21-stdev and range plot of final 13kg plot

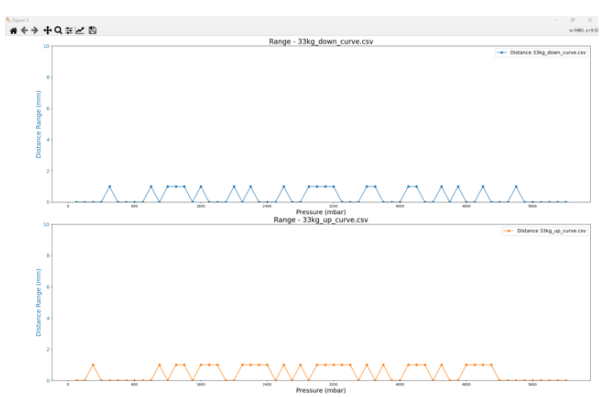


Appendix 22-stdev and range plot of final 24kg plot

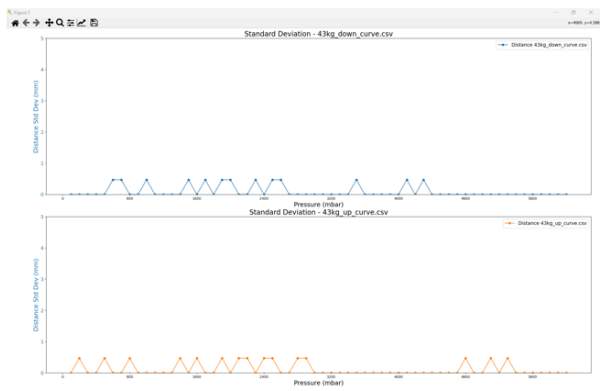


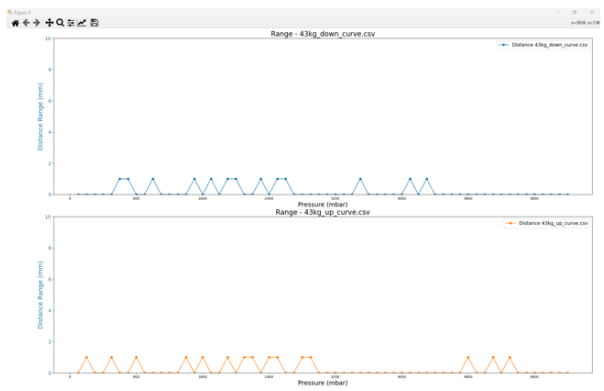
Appendix 23-stdev and range plot of final 33kg plot



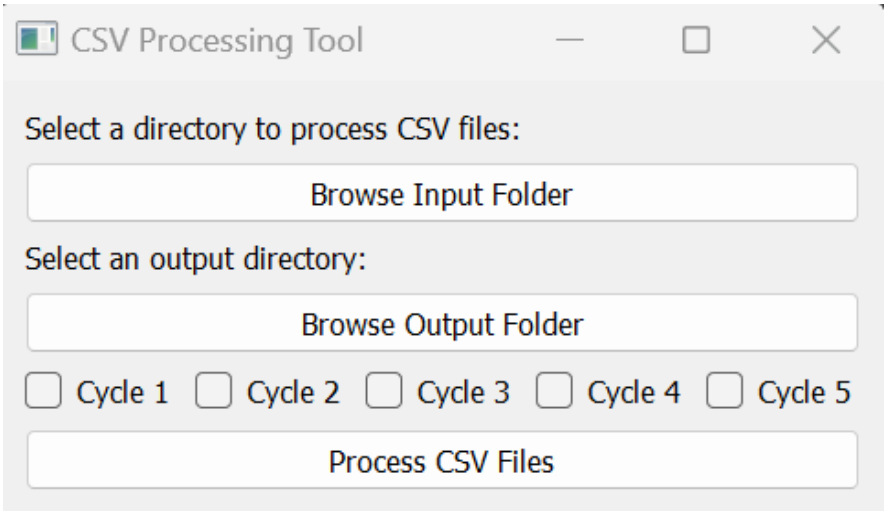


Appendix 24-stdev and range plot of final 43kg plot





Appendix 25-csv_averager_utility.py GUI

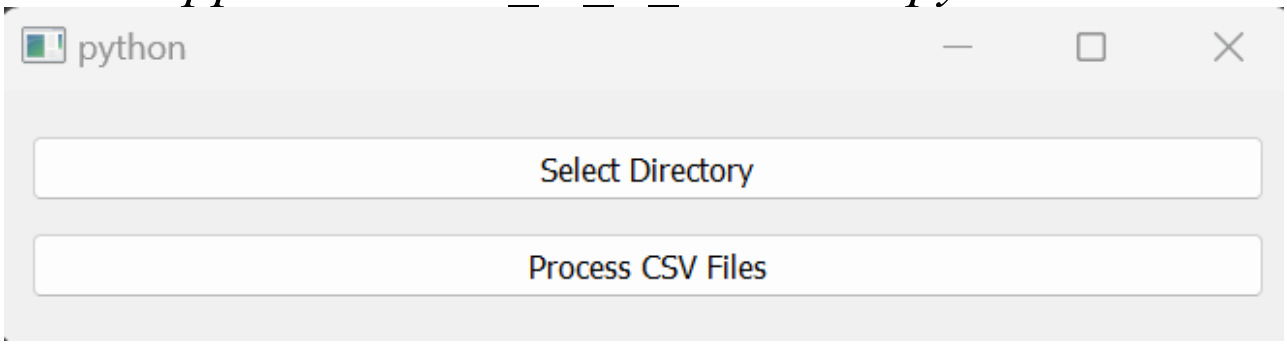


Appendix 26-csv_averager_utility.py code snippets

```python
import sys
import os
import pandas as pd
from PyQt5.QtWidgets import (
    QApplication, QWidget, QPushButton, QFileDialog, QVBoxLayout, QLabel, QCheckBox, QHBoxLayout, QMessageBox
)

class CSVProcessor(QWidget):
    def __init__(self):
        super().__init__()
        self.initUI()
        self.selected_cycles = []        # List to store selected cycle numbers
        self.output_directory = ""       # Path for output directory

    def initUI(self):
        layout = QVBoxLayout()

        # Label and button to select input directory
        self.label = QLabel("Select a directory to process CSV files:")
        layout.addWidget(self.label)

        self.btnBrowse = QPushButton("Browse Input Folder")
        self.btnBrowse.clicked.connect(self.browseFolder)
        layout.addWidget(self.btnBrowse)

        # Label and button to select output directory
        self.output_label = QLabel("Select an output directory:")
        layout.addWidget(self.output_label)

        self.btnOutputBrowse = QPushButton("Browse Output Folder")
        self.btnOutputBrowse.clicked.connect(self.browseOutputFolder)
        layout.addWidget(self.btnOutputBrowse)

        # Create checkboxes for selecting cycles 1 to 5
        self.checkboxes = []
        self.checkbox_layout = QHBoxLayout()
        for i in range(1, 6):
            checkbox = QCheckBox(f"Cycle {i}")
            checkbox.stateChanged.connect(self.updateSelectedCycles)
            self.checkboxes.append(checkbox)
            self.checkbox_layout.addWidget(checkbox)
        layout.addLayout(self.checkbox_layout)

        # Button to process CSV files
        self.btnProcess = QPushButton("Process CSV Files")
        self.btnProcess.clicked.connect(self.processFiles)
        layout.addWidget(self.btnProcess)
```

```
        self.setLayout(layout)
        self.setWindowTitle("CSV Processing Tool")

    def updateSelectedCycles(self):
        # Update the list of selected cycles based on checkbox states
        self.selected_cycles = [i+1 for i, cb in enumerate(self.checkboxes) if cb.isChecked()]

    def browseFolder(self):
        # Prompt user to select input folder
        folder = QFileDialog.getExistingDirectory(self, "Select Input Directory")
        if folder:
            self.label.setText(f"Selected Input Directory: {folder}")
            self.directory = folder

    def browseOutputFolder(self):
        # Prompt user to select output folder
        folder = QFileDialog.getExistingDirectory(self, "Select Output Directory")
        if folder:
            self.output_label.setText(f"Selected Output Directory: {folder}")
            self.output_directory = folder

    def processFiles(self):
        # Validate input and output folder selections
        if not hasattr(self, 'directory') or not self.directory:
            QMessageBox.warning(self, "Warning", "Please select an input directory first.")
            return

        if not self.output_directory:
            QMessageBox.warning(self, "Warning", "Please select an output directory.")
            return

        if not self.selected_cycles:
            QMessageBox.warning(self, "Warning", "Please select at least one cycle to average.")
            return

        # Walk through all CSV files in input directory and process them
        for root, _, files in os.walk(self.directory):
            for file in files:
                if file.endswith(".csv"):
                    file_path = os.path.join(root, file)
                    self.processCSV(file_path)

        QMessageBox.information(self, "Success", "CSV files have been processed and saved to the output folder.")

    def processCSV(self, file_path):
        df = pd.read_csv(file_path)
        pressure_col = "Pressure (mbar)"

        # Build column names for selected cycles
        distance_cols = [f"Distance Cycle {i} (mm)" for i in self.selected_cycles]
        load_cols = [f"Load Cycle {i} (kg)" for i in self.selected_cycles]

        # Calculate averages across selected cycles
        df["Distance (mm)"] = df[distance_cols].mean(axis=1)
        df["Load(kg)"] = df[load_cols].mean(axis=1)

        # Keep only pressure, averaged distance and averaged load columns
        df = df[[pressure_col, "Distance (mm)", "Load(kg)"]]

        # Construct new filename with selected cycle range
        start_cycle = min(self.selected_cycles)
        end_cycle = max(self.selected_cycles)

        new_file_name = os.path.basename(file_path).replace(".csv", f"_averaged_{start_cycle}__{end_cycle}.csv")
        new_file_path = os.path.join(self.output_directory, new_file_name)
        df.to_csv(new_file_path, index=False)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = CSVProcessor()
    window.show()
    sys.exit(app.exec_())
```

Appendix 27-D_to_P_converter.py GUI

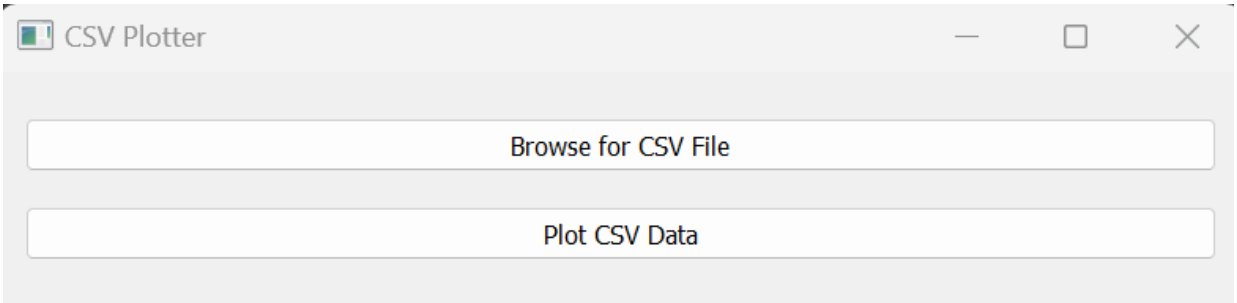


Appendix 28-D_to_P_converter code snippets

```python
import os
import pandas as pd
import numpy as np
from PyQt5 import QtWidgets
from PyQt5.QtWidgets import QFileDialog, QMessageBox
import scipy.interpolate as interp

# Interpolation method used for generating evenly spaced pressure values
INTERPOLATION_TYPE = 'linear'  # Change to 'cubic' if needed

class CSVProcessor:
    def __init__(self, directory):
        self.directory = directory
        self.file_pairs = self.find_csv_pairs()  # Find matching up/down file pairs by weight

    def find_csv_pairs(self):
        # List all CSV files in the directory
        files = [f for f in os.listdir(self.directory) if f.endswith('.csv')]
        weights = {}

        # Group files by weight and direction (up/down)
        for file in files:
            parts = file.split('_')
            try:
                weight = int(parts[0][:-2])  # Extract weight from filename (e.g., "13kg_up_..." → 13)
                direction = 'up' if 'up' in file else 'down' if 'down' in file else None
                if direction:
                    weights.setdefault(weight, {})[direction] = file
            except ValueError:
                continue  # Skip files that don't match expected naming format

        # Return only complete up/down pairs
        return {w: p for w, p in weights.items() if 'up' in p and 'down' in p}

    def process_and_save(self):
        # Process each up/down pair
        for weight, pair in self.file_pairs.items():
            up_file = os.path.join(self.directory, pair['up'])
            down_file = os.path.join(self.directory, pair['down'])

            # Read CSV files
            df_up = pd.read_csv(up_file)
            df_down = pd.read_csv(down_file)

            # Extract distance and pressure columns (assumed to be the first two columns)
            distance_up, pressure_up = df_up.iloc[:, 1], df_up.iloc[:, 0]
            distance_down, pressure_down = df_down.iloc[:, 1], df_down.iloc[:, 0]
```

```
48
49              max_distance = 200  # Maximum distance to interpolate over
50              interpolated_distances = np.arange(0, max_distance + 1, 1)  # Step of 1 mm
51              print(interpolated_distances)  # Debug print
52
53              # Create interpolation functions
54              f_up = interp.interp1d(distance_up, pressure_up, kind=INTERPOLATION_TYPE, fill_value='extrapolate')
55              f_down = interp.interp1d(distance_down, pressure_down, kind=INTERPOLATION_TYPE, fill_value='extrapolate')
56
57              # Interpolate pressure values over the new distance grid
58              interpolated_up = f_up(interpolated_distances)
59              interpolated_down = f_down(interpolated_distances)
60
61              # Stack both interpolated arrays for saving
62              output_data = np.vstack([interpolated_up, interpolated_down])
63
64              # Create output filename with weight info
65              output_filename = os.path.join(self.directory, f'DtoP_{weight}.csv')
66
67              # Round and convert to integer
68              output_data = np.round(output_data).astype(int)
69
70              # Save to CSV with a header showing weight
71              np.savetxt(output_filename, output_data, delimiter=',', header=f'# weight={weight}', comments='', fmt='%d')
72
73              print(f'Saved: {output_filename}')  # Confirmation message
74
75  class CSVApp(QtWidgets.QWidget):
76      def __init__(self):
77          super().__init__()
78          self.initUI()  # Setup the UI
79
80      def initUI(self):
81          self.layout = QtWidgets.QVBoxLayout()
82
83          # Button to select input directory
84          self.select_dir_btn = QtWidgets.QPushButton('Select Directory')
85          self.select_dir_btn.clicked.connect(self.select_directory)
86
87          # Button to start processing
88          self.process_btn = QtWidgets.QPushButton('Process CSV Files')
89          self.process_btn.clicked.connect(self.process_csv_files)
90
91          # Add buttons to layout
92          self.layout.addWidget(self.select_dir_btn)
93          self.layout.addWidget(self.process_btn)
94          self.setLayout(self.layout)
95
96      def select_directory(self):
97          # Open folder selection dialog
98          dir_name = QFileDialog.getExistingDirectory(self, 'Select Directory')
99          if dir_name:
100             self.processor = CSVProcessor(dir_name)  # Create processor instance
101             QMessageBox.information(self, 'Success', f'Found {len(self.processor.file_pairs)} valid pairs.')
102
103     def process_csv_files(self):
104         # Trigger processing if a directory was selected
105         if hasattr(self, 'processor'):
106             self.processor.process_and_save()
107             QMessageBox.information(self, 'Success', 'Processing completed!')
108         else:
109             QMessageBox.warning(self, 'Error', 'No directory selected.')
110
111 # Entry point for the PyQt application
112 if __name__ == '__main__':
113     app = QtWidgets.QApplication([])
114     window = CSVApp()
115     window.show()
116     app.exec_()
```

Appendix 29-D_To_P_plotter.py GUI snippet

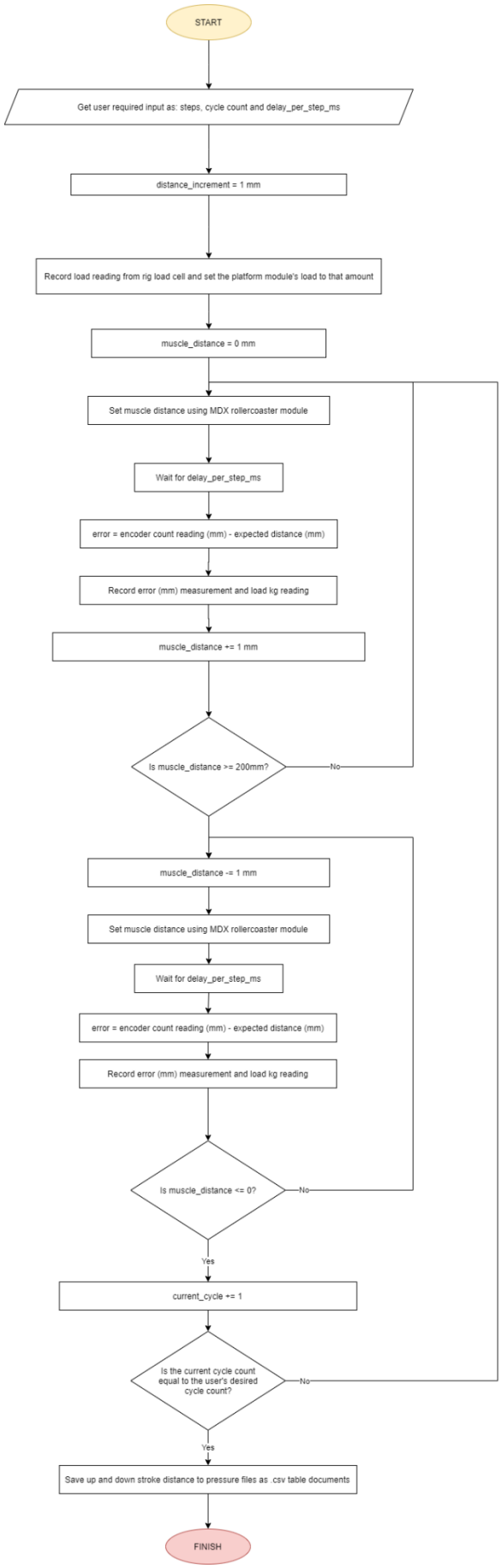


Appendix 30-D_To_P_plotter code snippets

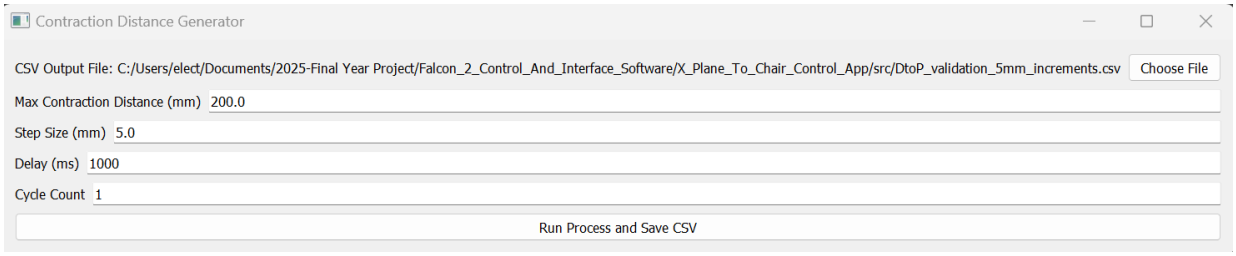
```python
import sys
import pandas as pd
import matplotlib.pyplot as plt
from PyQt5 import QtWidgets
from PyQt5.QtWidgets import QFileDialog, QMessageBox

class PlotterApp(QtWidgets.QWidget):
    def __init__(self):
        super().__init__()
        self.initUI()

    def initUI(self):
        self.layout = QtWidgets.QVBoxLayout()

        # Button for selecting CSV file
        self.browse_file_btn = QtWidgets.QPushButton('Browse for CSV File')
        self.browse_file_btn.clicked.connect(self.browse_file)

        # Button for plotting the data
        self.plot_btn = QtWidgets.QPushButton('Plot CSV Data')
        self.plot_btn.clicked.connect(self.plot_data)

        # Add buttons to the layout
        self.layout.addWidget(self.browse_file_btn)
        self.layout.addWidget(self.plot_btn)

        # Set layout
        self.setLayout(self.layout)

        self.selected_file = None

    def browse_file(self):
        # Open file dialog to select a CSV file
        file_name, _ = QFileDialog.getOpenFileName(self, 'Select CSV File', '', 'CSV Files (*.csv)')
        if file_name:
            self.selected_file = file_name
            QMessageBox.information(self, 'File Selected', f'Selected file: {file_name}')

    def plot_data(self):
        # Check if file is selected
        if self.selected_file:
            try:
                # Read CSV file (skip the first row which contains weight info)
                df = pd.read_csv(self.selected_file, skiprows=1, header=None)

```

```
                # Ensure we have at least two rows
                if df.shape[0] < 2:
                    raise ValueError("CSV file doesn't contain enough data for plotting.")

                # Extract the second and third rows for plotting (pressure values)
                pressure_up = df.iloc[0, :].values  # First row of pressures
                pressure_down = df.iloc[1, :].values  # Second row of pressures

                print(pressure_up)

                # The x-axis (distance) is just the index of the values, starting from 1
                distance = list(range(1, len(pressure_up) + 1))

                # Plot the data
                plt.figure(figsize=(8, 6))

                plt.plot(distance, pressure_up, label="Pressure Up",
                         color='b', linestyle='-', marker='o')  # Line with points for Pressure Up

                plt.plot(distance, pressure_down, label="Pressure Down",
                         color='r', linestyle='-', marker='x')  # Line with points for Pressure Down

                # Add scatter for individual points (optional, as we already added markers to the plot)
                plt.scatter(distance, pressure_up, color='b', zorder=5)
                plt.scatter(distance, pressure_down, color='r', zorder=5)

                plt.xlabel('Distance (mm)')
                plt.ylabel('Pressure (mbar)')
                plt.title('Pressure vs Distance Plot')
                plt.legend()
                plt.grid(True)
                plt.show()

            except Exception as e:
                QMessageBox.warning(self, 'Error', f'Failed to read and plot the CSV file.\n{str(e)}')
        else:
            QMessageBox.warning(self, 'Error', 'No file selected. Please select a CSV file first.')

if __name__ == '__main__':
    app = QtWidgets.QApplication(sys.argv)
    window = PlotterApp()
    window.setWindowTitle("CSV Plotter")
    window.setGeometry(100, 100, 300, 150)  # Set window size and position
    window.show()
    sys.exit(app.exec_())
```

Appendix 31-Muscle DtoP test software flowchart



Appendix 32-DtoP_Validator.py GUI snippet



Appendix 33-DtoP_Validator.py code snippets

```python
import sys
import time
import csv
from platform_pose import Platform
from common.serialProcess import SerialProcess
from PyQt5.QtWidgets import (
    QApplication, QWidget, QLabel, QLineEdit, QPushButton,
    QVBoxLayout, QFileDialog, QHBoxLayout, QMessageBox
)

from d_to_p_ver_2 import DistanceToPressure

class ContractionApp(QWidget):
    def __init__(self):
        super().__init__()
        self.initUI()  # Initialize the user interface
        self.output_file = ""  # Output file path for saving data
        self.platform = Platform()  # Platform instance for motion control
        self.encoder = SerialProcess()  # Serial connection for encoder
        self.d_to_p = DistanceToPressure("output\\wheelchair_DtoP.csv")  # Distance-to-pressure conversion
        self.d_to_p.set_load(24)  # Set load factor for the conversion

    def initUI(self):
        """Set up the user interface components"""
        self.setWindowTitle("Contraction Distance Generator")

        layout = QVBoxLayout()

        # Output file selection section
        file_layout = QHBoxLayout()
        self.file_label = QLabel("CSV Output File: Not selected")  # Label to show selected file
        file_btn = QPushButton("Choose File")  # Button to select file
        file_btn.clicked.connect(self.choose_file)  # Connect button to choose_file function
        file_layout.addWidget(self.file_label)
        file_layout.addWidget(file_btn)
        layout.addLayout(file_layout)

        # Input fields with default values
        defaults = {
            "Max Contraction Distance (mm)": "200.0",  # Default max contraction distance
            "Step Size (mm)": "5.0",  # Default step size
            "Delay (ms)": "1000",  # Default delay in milliseconds
            "Cycle Count": "1"  # Default cycle count
```

```
        }

        self.inputs = {}  # Dictionary to store input fields
        for label, default in defaults.items():
            row = QHBoxLayout()
            lbl = QLabel(label)  # Label for input field
            inp = QLineEdit()  # Input field
            inp.setText(default)  # Set default text
            row.addWidget(lbl)
            row.addWidget(inp)
            layout.addLayout(row)
            self.inputs[label] = inp  # Store input field in dictionary

        # Process Button
        run_btn = QPushButton("Run Process and Save CSV")  # Button to run process
        run_btn.clicked.connect(self.run_process)  # Connect button to run_process function
        layout.addWidget(run_btn)

        self.setLayout(layout)  # Set layout for the window

    def choose_file(self):
        """Open a file dialog to choose output CSV file"""
        file_name, _ = QFileDialog.getSaveFileName(self, "Save CSV", "", "CSV Files (*.csv)")  # File dialog
        if file_name:
            self.output_file = file_name  # Set the selected file path
            self.file_label.setText(f"CSV Output File: {file_name}")  # Update label to show selected file

    def GetDistanceAndLoad(self):
        """Retrieve extension and load values from the encoder"""
        data = self.encoder.read()  # Read data from encoder
        print(data)
        if data is None:
            print("function does not function!")
            return (0.0, 0.0)  # Return default values if no data
        array = data.split(",")  # Split the data into components
        return (-float(array[1]), float(array[3]) / 9.81)  # Return extension (mm) and load (kg)

    def run_process(self):
        """Run the contraction process and save the results to CSV"""
        if not self.output_file:
            QMessageBox.warning(self, "No File", "Please select an output CSV file.")  # Warning if no file is selected
            return

        try:
            # Get values from input fields and convert to appropriate types
            max_dist = float(self.inputs["Max Contraction Distance (mm)"].text())
            step_size = float(self.inputs["Step Size (mm)"].text())
            delay_ms = int(self.inputs["Delay (ms)"].text())
            cycles = int(self.inputs["Cycle Count"].text())
        except ValueError:
            QMessageBox.critical(self, "Invalid Input", "Please enter valid numeric values.")  # Error if input is invalid
            return

        port = "COM10"  # Serial port for encoder connection
        print(port)
        self.encoder.open_port(port, 115200)  # Open serial connection to the encoder
        time.sleep(0.1)  # Wait for the queue to fill up
        self.encoder.write("R".encode())  # Send "R" command to the encoder
        _, load = self.GetDistanceAndLoad()  # Get initial load value
        self.LOAD = round(load)  # Round the load value
        print("THE LOAD IS ", self.LOAD)

        # Build the contraction profile (one rising + falling pass)
        distances = []  # List to store contraction distances
        d = 0.0
        while d <= max_dist:
            distances.append(round(d, 4))  # Add each distance to the list
            d += step_size
        d = max_dist - step_size
        while d >= 0:
            distances.append(round(d, 4))  # Add each distance for the falling pass
            d -= step_size

        # Create data structure to hold results
        results = {dist: {} for dist in distances}

        # Run the cycles
        for cycle in range(1, cycles + 1):
            for dist in distances:
                # Get pressure value based on the contraction distance
                pressure = self.d_to_p.get_pressure(max(int(dist - 2), 0))

```

```
                print(f"Pressure at {dist}mm is {pressure}mbar")

                # Send pressure values to the platform
                self.platform.muscle.send_pressures([pressure, 0, 0, 0, 0, 0])
                time.sleep(delay_ms / 1000.0)  # Wait for the delay

                print("Distance: ", dist, "  Cycle: ", cycle)

                # Measure the distance and load after each step
                measured_dist, load = self.GetDistanceAndLoad()

                print(measured_dist, " ", dist)

                # Store the results (error and load) for each distance and cycle
                results[dist][f"error mm cycle {cycle}"] = measured_dist - dist
                results[dist][f"load kg cycle {cycle}"] = load

        # Build headers for the CSV
        headers = ["contraction_distance mm"]
        for cycle in range(1, cycles + 1):
            headers.append(f"error mm cycle {cycle}")
            headers.append(f"load kg cycle {cycle}")

        # Write the results to the CSV file
        with open(self.output_file, mode='w', newline='') as file:
            writer = csv.writer(file)
            writer.writerow(headers)  # Write headers
            for dist in distances:
                row = [dist]
                for cycle in range(1, cycles + 1):
                    row.append(results[dist].get(f"error mm cycle {cycle}", ""))  # Add error values
                    row.append(results[dist].get(f"load kg cycle {cycle}", ""))  # Add load values
                writer.writerow(row)  # Write row to CSV

        # Show a message box to indicate the process is complete
        QMessageBox.information(self, "Done", f"Data saved to {self.output_file}")

if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = ContractionApp()  # Create an instance of the app
    window.show()  # Show the window
    sys.exit(app.exec_())  # Start the Qt application loop
```

Appendix 34-d_to_p_ver_2.py code snippets

```
import numpy as np
import pandas as pd

class DistanceToPressure:
    def __init__(self, csv_path):
        # Load the CSV file using pandas
        self.data = pd.read_csv(csv_path)

        rows = self.data.values.tolist()  # Convert the data into a list of rows

        # Ensure I am getting the row data (print all values in the second column for verification)
        for i in rows:
            print(i[1])

        # Store the rows corresponding to weight-direction pairs in self.weight_direction_rows
        self.weight_direction_rows = rows
        # Define a list of known weights
        self.weights = [13, 24, 33, 43]
        self.is_going_up = True  # Initial direction of movement (default: True, going up)
        self.last_distance = 0  # Track the last distance

    def set_load(self, load):
        """Set the load and find the closest weights to interpolate between."""
        # If the load is not in the predefined weights, calculate the closest lower and upper weights
        if load not in self.weights:
            lower_weight = max([w for w in self.weights if w < load], default=None)
            upper_weight = min([w for w in self.weights if w > load], default=None)

            # If no valid lower or upper weight, raise an error
            if lower_weight is None and upper_weight is None:
                raise ValueError(f"No weights available for load {load}.")
            # If only the lower weight is available, use it and set interpolation factor to 0
            elif lower_weight is None:
                self.load = (upper_weight, upper_weight)
                self.interpolation_factor = 0
```

```
            # If only the upper weight is available, use it and set interpolation factor to 0
            elif upper_weight is None:
                self.load = (lower_weight, lower_weight)
                self.interpolation_factor = 0
            # Otherwise, use both lower and upper weights for interpolation
            else:
                self.load = (lower_weight, upper_weight)
                self.interpolation_factor = (load - lower_weight) / (upper_weight - lower_weight)
        else:
            # If the load matches one of the predefined weights, no interpolation is needed
            self.load = (load, load)
            self.interpolation_factor = 0

    def get_pressure(self, distance):
        """
        Calculate the pressure based on the given distance.
        The pressure is calculated by interpolating between pressure values corresponding
        to the lower and upper weights at the specified distance.
        """
        # NOTE FOR FUTURE ME: Change curves to be set only once when setting load for better efficiency!

        lower_weight, upper_weight = self.load

        # Find the index of the lower and upper weights in the list of weights
        lower_row_index = self.weights.index(lower_weight)
        upper_row_index = self.weights.index(upper_weight)

        lower_weight_rows = []
        upper_weight_rows = []

        # Find the rows corresponding to the lower and upper weights (for both directions)
        lower_weight_rows.append(self.weight_direction_rows[lower_row_index])
        lower_weight_rows.append(self.weight_direction_rows[lower_row_index + 4])

        upper_weight_rows.append(self.weight_direction_rows[upper_row_index])
        upper_weight_rows.append(self.weight_direction_rows[upper_row_index + 4])

        # Print values from the "up" and "down" curves to check if correct rows are selected
        print(upper_weight_rows[0][4], upper_weight_rows[1][4])

```

```
        # Determine the direction of movement based on whether the distance has increased or decreased
        self.is_going_up = distance - self.last_distance > 0

        # Initialize the pressure variable
        lerp_pressure = -1

        # Interpolate the pressure based on the direction (up or down)
        if self.is_going_up:
            # Interpolate for the "up" direction using the rows for the lower and upper weights
            lerp_pressure = lower_weight_rows[0][distance] + (self.interpolation_factor * (upper_weight_rows[0][distance] - lower_weight_rows[0][distance]))
        else:
            # Interpolate for the "down" direction using the rows for the lower and upper weights
            lerp_pressure = lower_weight_rows[1][distance] + (self.interpolation_factor * (upper_weight_rows[1][distance] - lower_weight_rows[1][distance]))

        # Update the last distance for future calculations
        self.last_distance = distance

        # Return the pressure, rounded to the nearest integer
        return round(lerp_pressure)

# Example usage:
# Create an instance of the DistanceToPressure class by providing the path to your CSV file
#muscle = DistanceToPressure("output\\wheelchair_DtoP.csv")

# Set the load (for example, 40kg)
#muscle.set_load(24)

# Retrieve the pressure for a given distance (e.g., distance = 10, up direction)
#try:
#    for i in range(0, 10):
#        pressure = muscle.get_pressure(i)
#        print(f"Pressure at {i}mm distance (up direction): {pressure} mbar")
#
#    for i in range(10, 0, -1):
#        pressure = muscle.get_pressure(i)
#        print(f"Pressure at {i}mm distance (down direction): {pressure} mbar")
#
#except IndexError as e:
#    print(f"Error: {e}")

```

```
        # Determine the direction of movement based on whether the distance has increased or decreased
        self.is_going_up = distance - self.last_distance > 0

        # Initialize the pressure variable
        lerp_pressure = -1

        # Interpolate the pressure based on the direction (up or down)
        if self.is_going_up:
            # Interpolate for the "up" direction using the rows for the lower and upper weights
            lerp_pressure = lower_weight_rows[0][distance] + (self.interpolation_factor * (upper_weight_rows[0][distance]
                                                                                           - lower_weight_rows[0][distance]))
        else:
            # Interpolate for the "down" direction using the rows for the lower and upper weights
            lerp_pressure = lower_weight_rows[1][distance] + (self.interpolation_factor * (upper_weight_rows[1][distance]
                                                                                           - lower_weight_rows[1][distance]))

        # Update the last distance for future calculations
        self.last_distance = distance

        # Return the pressure, rounded to the nearest integer
        return round(lerp_pressure)

# Example usage:
# Create an instance of the DistanceToPressure class by providing the path to your CSV file
#muscle = DistanceToPressure("output\\wheelchair_DtoP.csv")

# Set the load (for example, 40kg)
#muscle.set_load(24)

# Retrieve the pressure for a given distance (e.g., distance = 10, up direction)
#try:
#    for i in range(0, 10):
#        pressure = muscle.get_pressure(i)
#        print(f"Pressure at {i}mm distance (up direction): {pressure} mbar")
#
#    for i in range(10, 0, -1):
#        pressure = muscle.get_pressure(i)
#        print(f"Pressure at {i}mm distance (down direction): {pressure} mbar")
#
#except IndexError as e:
#    print(f"Error: {e}")
```

Appendix 35-DtoP_Validation_Plotter.py GUI snippet

Appendix 36-DtoP_Validation_Plotter.py code snippets

```python
import sys
import pandas as pd
from PyQt5.QtWidgets import (
    QApplication, QWidget, QVBoxLayout, QPushButton, QFileDialog,
    QLabel, QHBoxLayout
)
from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as FigureCanvas
from matplotlib.figure import Figure


class PlotWindow(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Distance Plotting Utility")  # Set window title
        self.setGeometry(100, 100, 900, 700)  # Set window size and position

        self.layout = QVBoxLayout()  # Main vertical layout for the window
        self.setLayout(self.layout)

        button_layout = QHBoxLayout()  # Layout for buttons

        # Button for selecting CSV file
        self.load_button = QPushButton("Select CSV File")
        self.load_button.clicked.connect(self.load_csv)  # Connect button to load_csv method
        button_layout.addWidget(self.load_button)

        # Button for switching between distance and error plots
        self.toggle_button = QPushButton("Switch to Error Plot")
        self.toggle_button.clicked.connect(self.toggle_plot_mode)  # Connect button to toggle_plot_mode method
        self.toggle_button.setEnabled(False)  # Initially disabled
        button_layout.addWidget(self.toggle_button)

        # Button for exporting the plot as an image
        self.export_button = QPushButton("Export Plot as Image")
        self.export_button.clicked.connect(self.export_plot)  # Connect button to export_plot method
        self.export_button.setEnabled(False)  # Initially disabled
        button_layout.addWidget(self.export_button)

        self.layout.addLayout(button_layout)  # Add button layout to main layout

        # Set up the plot canvas
        self.canvas = FigureCanvas(Figure())  # Create an empty figure canvas
        self.ax = self.canvas.figure.add_subplot(111)  # Add subplot to figure
```

```
        self.layout.addWidget(self.canvas)  # Add the canvas to the layout

        # Labels for displaying the max error and standard deviation
        self.error_label = QLabel("Max Error: N/A")
        self.std_label = QLabel("Standard Deviation of Error: N/A")
        self.layout.addWidget(self.error_label)
        self.layout.addWidget(self.std_label)

        # Initialize data attributes
        self.df = None  # Dataframe to hold CSV data
        self.max_error = None  # Max error value
        self.std_dev = None  # Standard deviation of error
        self.plot_mode = 'distance'  # Default plot mode is 'distance'

    def load_csv(self):
        """Load the selected CSV file and prepare data for plotting"""
        filepath, _ = QFileDialog.getOpenFileName(self, "Open CSV File", "", "CSV Files (*.csv)")  # Open file dialog
        if not filepath:
            return  # If no file is selected, return

        df = pd.read_csv(filepath)  # Read CSV into dataframe
        df.columns = [col.strip() for col in df.columns]  # Clean column names (remove any extra spaces)

        # Add a new column for the actual measured distance
        df['actual_distance'] = df['contraction_distance mm'] + df['error mm cycle 1']

        # Store the dataframe and calculate error statistics
        self.df = df
        self.max_error = df['error mm cycle 1'].abs().max()  # Calculate max error
        self.std_dev = df['error mm cycle 1'].std()  # Calculate standard deviation of error

        # Update the labels with error statistics
        self.error_label.setText(f"Max Error: {self.max_error:.3f} mm")
        self.std_label.setText(f"Standard Deviation of Error: {self.std_dev:.3f} mm")

        # Enable buttons after CSV is loaded
        self.export_button.setEnabled(True)
        self.toggle_button.setEnabled(True)
        self.plot_mode = 'distance'  # Default to distance plot
        self.toggle_button.setText("Switch to Error Plot")  # Update button text

        self.plot_distance()  # Plot the distance graph initially

    def toggle_plot_mode(self):
        """Switch between distance plot and error plot"""
        if self.plot_mode == 'distance':
            self.plot_mode = 'error'  # Switch to error plot mode
            self.toggle_button.setText("Switch to Distance Plot")  # Update button text
            self.plot_error()  # Plot the error graph
        else:
            self.plot_mode = 'distance'  # Switch back to distance plot mode
            self.toggle_button.setText("Switch to Error Plot")  # Update button text
            self.plot_distance()  # Plot the distance graph

    def plot_distance(self):
        """Plot the contraction distance vs actual distance graph"""
        if self.df is None:
            return  # If no data is available, return

        df = self.df
        max_idx = df['contraction_distance mm'].idxmax()  # Find index of max contraction distance

        self.ax.clear()  # Clear the current plot
        self.canvas.figure.suptitle("Contraction vs Actual Distance", fontsize=18, fontweight='bold')  # Set plot title

        # Plot expected contraction distance (dashed line)
        self.ax.plot(df['contraction_distance mm'],
                     df['contraction_distance mm'],
                     linestyle='--', color='black', label='Expected Distance')

        # Plot measured distance (increasing)
        self.ax.plot(df['contraction_distance mm'][:max_idx + 1],
                     df['actual_distance'][:max_idx + 1],
                     color='blue', label='Measured Distance (Increasing)')

        # Plot measured distance (decreasing)
        self.ax.plot(df['contraction_distance mm'][max_idx + 1:],
                     df['actual_distance'][max_idx + 1:],
                     color='red', label='Measured Distance (Decreasing)')

        # Set labels and legend
        self.ax.set_title("Contraction Distance vs Actual Distance", fontsize=12)
```

```
        self.ax.set_xlabel("Contraction Distance (mm)")
        self.ax.set_ylabel("Distance (mm)")
        self.ax.legend()
        self.ax.grid(True)  # Add grid for better readability
        self.canvas.draw()  # Redraw the canvas

    def plot_error(self):
        """Plot the error vs expected contraction distance graph"""
        if self.df is None:
            return  # If no data is available, return

        df = self.df
        max_idx = df['contraction_distance mm'].idxmax()  # Find index of max contraction distance

        self.ax.clear()  # Clear the current plot
        self.canvas.figure.suptitle("Error vs Expected Contraction Distance", fontsize=18, fontweight='bold')  # Set plot title

        # Plot error for increasing distance
        self.ax.plot(df['contraction_distance mm'][:max_idx + 1],
                     df['error mm cycle 1'][:max_idx + 1],
                     color='blue', label='Error (Increasing)')

        # Plot error for decreasing distance
        self.ax.plot(df['contraction_distance mm'][max_idx + 1:],
                     df['error mm cycle 1'][max_idx + 1:],
                     color='red', label='Error (Decreasing)')

        self.ax.axhline(0, linestyle='--', color='gray')  # Add horizontal line at y=0 for reference
        self.ax.set_title("Error vs Contraction Distance", fontsize=12)
        self.ax.set_xlabel("Contraction Distance (mm)")
        self.ax.set_ylabel("Error (mm)")
        self.ax.legend()
        self.ax.grid(True)  # Add grid for better readability
        self.canvas.draw()  # Redraw the canvas

    def export_plot(self):
        """Export the current plot as an image"""
        if self.df is None:
            return  # If no data is available, return

        filepath, _ = QFileDialog.getSaveFileName(self, "Save Plot As Image", "", "PNG Files (*.png)")  # Open save dialog
        if not filepath:
            return  # If no file path is selected, return

        fig = Figure(figsize=(8, 6))  # Create a new figure
        fig.subplots_adjust(top=0.85)  # Make space for big title
        ax = fig.add_subplot(111)  # Add subplot to the figure
        df = self.df
        max_idx = df['contraction_distance mm'].idxmax()  # Find index of max contraction distance

        # Plot either the distance or error graph based on current mode
        if self.plot_mode == 'distance':
            fig.suptitle("Contraction vs Actual Distance", fontsize=18, fontweight='bold')

            ax.plot(df['contraction_distance mm'],
                    df['contraction_distance mm'],
                    linestyle='--', color='black', label='Expected Distance')

            ax.plot(df['contraction_distance mm'][:max_idx + 1],
                    df['actual_distance'][:max_idx + 1],
                    color='blue', label='Measured Distance (Increasing)')

            ax.plot(df['contraction_distance mm'][max_idx + 1:],
                    df['actual_distance'][max_idx + 1:],
                    color='red', label='Measured Distance (Decreasing)')

            ax.set_title("Contraction Distance vs Actual Distance", fontsize=12)
            ax.set_xlabel("Contraction Distance (mm)")
            ax.set_ylabel("Distance (mm)")
        else:
            fig.suptitle("Error vs Expected Contraction Distance", fontsize=18, fontweight='bold')

            ax.plot(df['contraction_distance mm'][:max_idx + 1],
                    df['error mm cycle 1'][:max_idx + 1],
                    color='blue', label='Error (Increasing)')

            ax.plot(df['contraction_distance mm'][max_idx + 1:],
                    df['error mm cycle 1'][max_idx + 1:],
                    color='red', label='Error (Decreasing)')

            ax.axhline(0, linestyle='--', color='gray')
            ax.set_title("Error vs Contraction Distance", fontsize=12)
```

```
            ax.set_xlabel("Contraction Distance (mm)")
            ax.set_ylabel("Error (mm)")

        ax.legend()  # Add legend
        ax.grid(True)  # Add grid

        # Add error statistics to the plot
        stat_text = f"Max Error: {self.max_error:.3f} mm\nStd Dev: {self.std_dev:.3f} mm"
        ax.text(0.95, 0.05, stat_text, transform=ax.transAxes,
                fontsize=10, verticalalignment='bottom', horizontalalignment='right',
                bbox=dict(facecolor='white', alpha=0.7, edgecolor='gray'))

        fig.savefig(filepath, dpi=300, bbox_inches='tight')  # Save plot to the selected file path
        print(f"Plot exported to: {filepath}")


if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = PlotWindow()
    window.show()  # Show the plot window
    sys.exit(app.exec_())  # Execute the application

```

9.1. Generative AI Use Disclaimer Form

This form is intended for PDE3823 students submitting their final-year project reports. Please complete the checklist below to provide transparency regarding the use of Generative AI tools during the project work. Submit this form alongside your final report.

Student Details:

- **Name:** Omar Maaouane Veiga
- **Student ID:** M00853972
- **Project Title:** Control and Interface Systems for a Flight Simulation Experience
- **Supervisor:** Michael Margolis

Generative AI Usage Declaration

Please indicate how generative AI tools were used during your project by ticking the relevant boxes below:

☐ No Generative AI tools were used during the completion of this project.

☐ AI tools were used for brainstorming and idea generation.

☒ AI tools were used for text drafting and report writing assistance.

☒ AI tools were used for code generation, debugging, or suggestions.

☐ AI tools were used for data analysis or visualisation support.

☐ AI tools were used for design simulation, modelling, or testing.

☐ Other (Please specify): _____

Acknowledgment

I hereby confirm that:

- All AI-generated content used in this report was critically evaluated, edited, and verified for accuracy and appropriateness.
- All intellectual contributions remain my own, and the final report reflects my independent work and understanding.
- Proper acknowledgment has been given where applicable.

Student Signature:  **Date:** 30/03/2025