

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

1.	Introduction	6
1.1.	Background and Motivation	6
1.1.	. Aims and Objectives	7
1.2.	Scope and Contributions	7
1.3.	Main project Resources and Repository Access	8
2.	Technical Background and Literature Review	8
2.1.	Brief history and overview of what is a motion platform.	8
2.2.	Technologies used on motion platforms.	11
2.2.1.	Standard platform configuration for 6DOF	11
2.2.2.	Actuator control schemes: Closed Loop vs Open Loop	12
2.2.3.	Technology behind the previous MDX rollercoaster project	12
2.2.4.	Previous research solutions from other projects compared to the MDX rollercoaster's solution.	14
2.2.5.	Brief overview of Inverse kinematics for the flight motion platform and washout algorithms	16
2.3.	Project challenges and limitations	18
2.4.	Lit review	19
2.5.	Summary	20
3.	Proposed System: Design and Implementation	20
3.1.	System Overview	20
3.2.	Software Structure and Components	21
4.	High level design outline of control software	23
4.1.	Core functionality high level outline.	23
4.2.	Network diagram of software components (nodes):	25
4.3.	Integrating the system with X-Plane	28
5.	Improving and implementing flight motion platform virtual displays	29
5.1.	Initial outline of new display	29
5.2.	Importing CAD model into Unity	30
5.3.	Re-Tooling existing code and exporting built application.	31
6.	Design, testing and evaluation of fluidic muscle calibration procedures.	32
6.1.	Designing muscle calibration procedures	32
6.2.	Implementing and testing muscle calibration procedures	35
6.3.	Evaluating test result data and obtaining conclusions.	38

7.	Integration of muscle calibration data back into control software	41
7.1.	Converting calibration cycle data into cycle averaged data	41
7.2.	Converting averaged PtoD tables to DtoP tables in required format	43
7.3.	Merging averaged DtoP tables into a master flight motion platform DtoP table	46
7.4.	Evaluating accuracy of merged DtoP table	48
8.	Results and discussions	49
8.1.	Overall results	49
8.2.	Discussion from results	50
9.	Conclusions and future works	50
	References	52
	Appendix	54
9.1.	Generative AI Use Disclaimer Form	106

List of Figures

Figure 1.1-CAD image of initial MDX rollercoaster motion platform.....	6
Figure 1.2-Rollercoaster motion platform with blower fan showcased in New Scientist Live	6
Figure 2.1-Picture of a motion platform used to film POV scenes of a moving truck for a film (Day Shift 2022)	8
Figure 2.2-Side by side comparison of the Sanders teacher to the SIMONA research simulator..	9
Figure 2.3-Picture of a 1980s motion platform ride.	9
Figure 2.4-PS-6TL-1500 Motion platform by motion systems.	10
Figure 2.5-Diagram of all the possible movements an aircraft can perform while airborne.	10
Figure 2.6-DOF Reality illustration of different platforms they offer, ranging from 2DOF, 3DOF and 6DOF.....	11
Figure 2.7-Example diagram of a Stewart platform.	11
Figure 2.8-Example of linear actuator with feedback position control.	12
Figure 2.9-Picture of MDX rollercoaster used in outreach event like New Scientist Live	13
Figure 2.10-Difference between a Stewart platform and an inverted Stewart platform.	13
Figure 2.11-Illustration of load vs stroke graph at different pressures for a Festo fluidic muscle from source [13].....	14
Figure 2.12-Past examples from papers of Stewart motion platforms using fluidic muscles as actuators.....	14
Figure 2.13-Illustration from paper [17] of a planar robotic arm controller by two fluidic muscles	15
Figure 2.14-Illustration of results the program aims to provide with in the Week 8 experiments.	15
Figure 2.15-Inverse kinematics of a Stewart Platform illustration.	16
Figure 2.16-Plane Pose to muscle distances accounting for washout. This graph is from around project week 5.....	16
Figure 2.17-Washout filter illustration for roll and pitch.	17
Figure 2.18- Illustration of illusion made by motion platforms to create the sense of acceleration.....	17
Figure 2.19-Illustration of washout for low frequency transient acceleration.....	18

Figure 3.1- Render of motion platform to be made in this project.....	20
Figure 3.2-High level illustration of how all the components in flight motion platform are meant to interact together	21
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).....	22
Figure 4.1-State machine diagram of flight motion platform control software.....	23
Figure 4.2-Node diagram of entire software.....	25
Figure 4.3-Diagram showing how XPPython works.	28
Figure 5.1-Screen snip of Unity visualization application.....	29
Figure 5.2-Ouline of workflow taken to import the CAD models of the flight motion platform from SOLIDWORKS to Unity.....	30
Figure 5.3-Before and after optimizing model geometry in Blender to export to Unity.	30
Figure 5.4-Figure showcasing the rig used to simulate the moving platform.	31
Figure 6.1-Example chart of muscle hysteresis loop.	32
Figure 6.2-Illustration of calibration rig used to obtain muscle pressure-to-distance data.	33
Figure 6.3-Illustration of how calibration program and test rig will communicate.....	33
Figure 6.4-Flowchart outlining the working logic of the calibration software.....	34
Figure 6.5-Pressure (mbar) vs Distance (mm) plot for 13kg load	36
Figure 6.6-Pressure (mbar) vs Distance (mm) plot for 24kg load.	36
Figure 6.7-Pressure (mbar) vs Distance (mm) plot for 33kg load.	37
Figure 6.8-Pressure (mbar) vs Distance (mm) plot for 43kg load.	37
Figure 6.9-Front cover of previous student paper.....	38
Figure 6.10-Comparison between previous project distance reading range per pressure and this project's.....	39
Figure 6.11-Conclusion section of previous student's project on the rollercoaster chair.....	39
Figure 6.12-Standard deviation of sensor up readings for previous student MDX rollercoaster project.	40
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.	41
Figure 7.1-Snippet of table raw data from calibration results. This file is for the 43kg down curve.	42
Figure 7.2-Averaged load muscle calibration files from cycles 3 to 5.....	42
Figure 7.3-Data output for 43kg up data after running it through utility app.....	43
Figure 7.4-Snippet of distance to pressure table for a given load.	43
Figure 7.5-output DtoP files for all the calibrated loads.....	44
Figure 7.6-DtoP table for 43kg load.	44
Figure 7.7-PtoD plot of 43kg csv file.....	45
Figure 7.8-merge_d_to_p code snippet.	46
Figure 7.9-D_to_P_merger.py code snippet.	46
Figure 7.10-Master DtoP csv file after merging all the valid PtoD files.	46
Figure 7.11-Plots of error vs set distance plots of merged DtoP table.	48

List of tables

Table 1.1 List of objectives this project aims to achieve.	7
Table 2.1-Comparison of existing solutions	19
Table 4.1-Table explaining what each state in control software does.	24
Table 4.2- Table outlining function of each node in software network.	26
Table 4.3-Table outlining function of each network UDP port.	27
Table 7.1-Structure of final merged DtoP table	47
Table 8.1-Requirements table outlining which requirements were met at the end of the project.	49
Table 9.1-Table detailing future work objectives.	51

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].

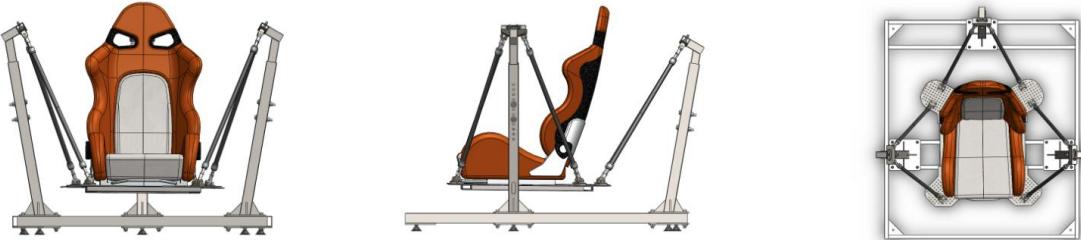


Figure 1.1-CAD image of initial MDX rollercoaster motion platform.

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 of 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

The 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

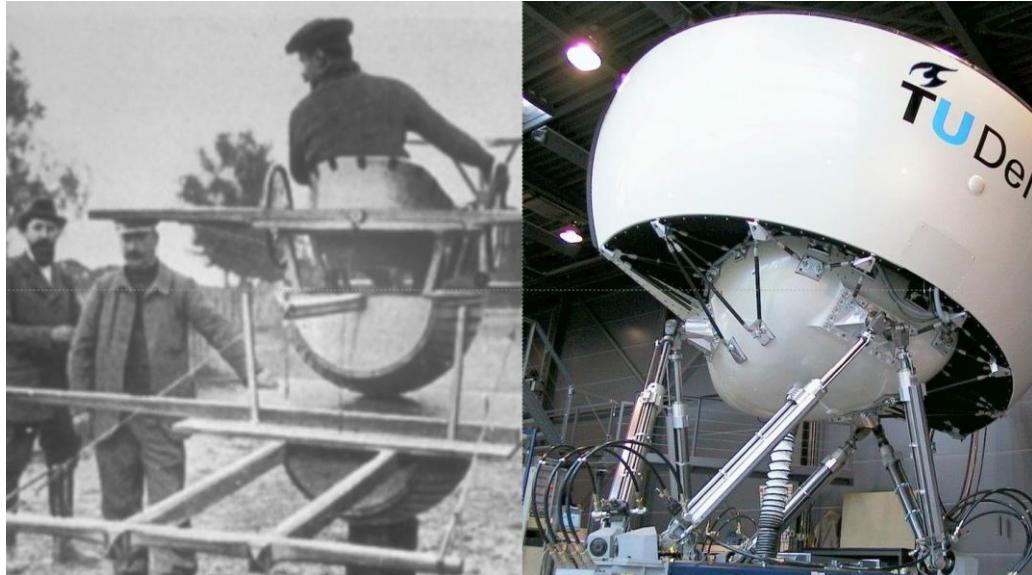


Figure 2.2-Side by side comparison of the Sanders teacher to the SIMONA research simulator.

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.

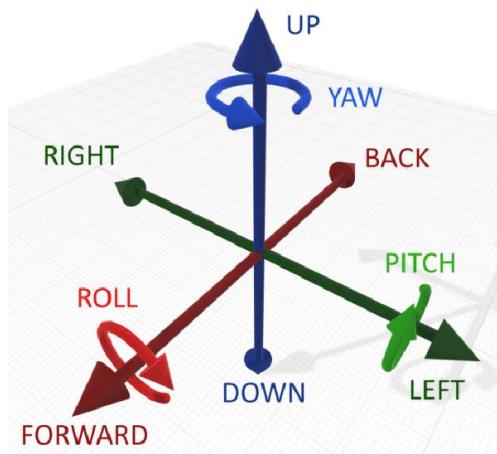


Figure 2.5-Diagram of all the possible movements an aircraft can perform while airborne.

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.

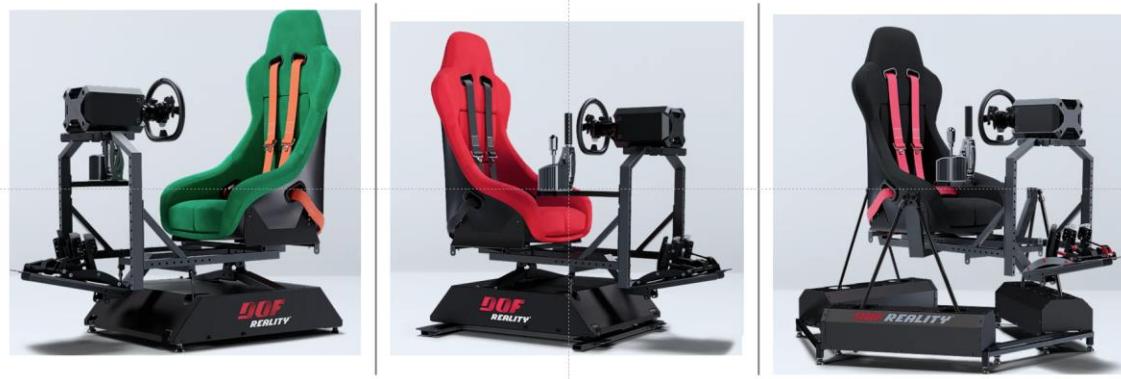


Figure 2.6-DOF Reality illustration of different platforms they offer, ranging from 2DOF, 3DOF and 6DOF

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 configuration 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.

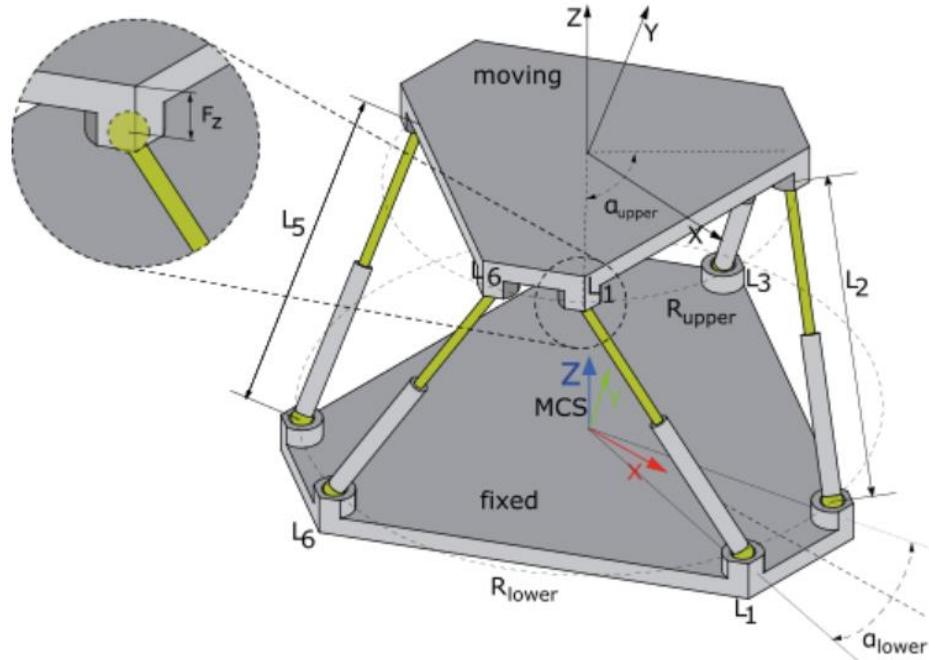


Figure 2.7-Example diagram of a Stewart platform.

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.

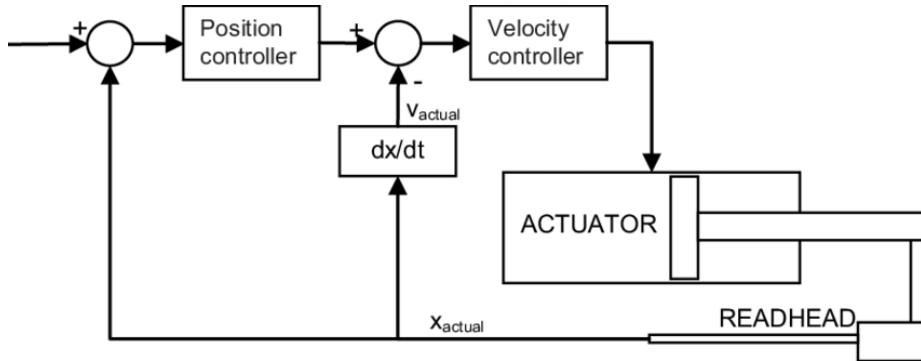


Figure 2.8-Example of linear actuator with feedback position control.

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

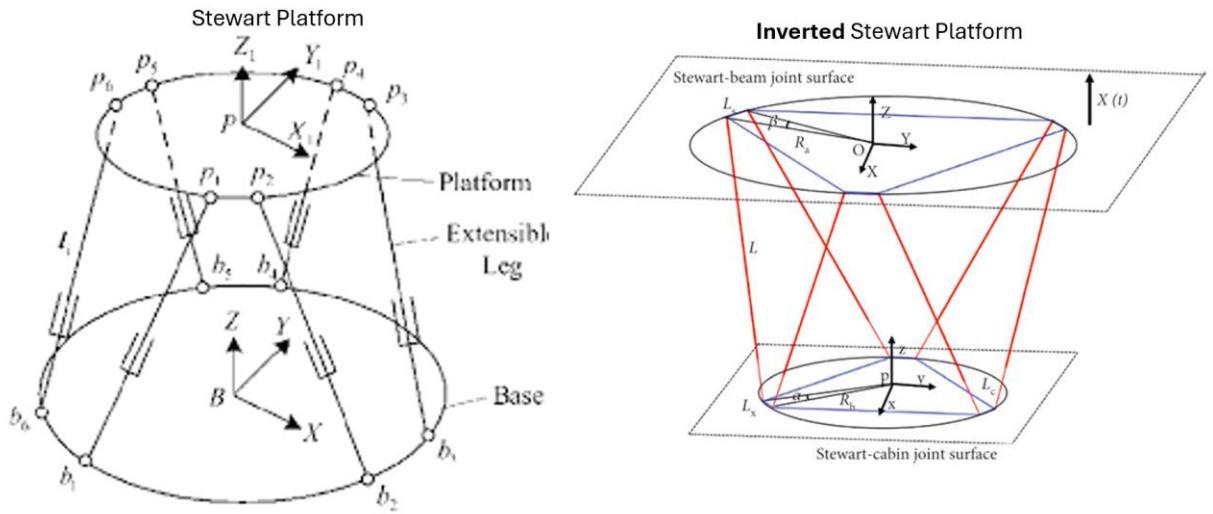


Figure 2.10-Difference between a Stewart platform and an inverted Stewart platform.

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.

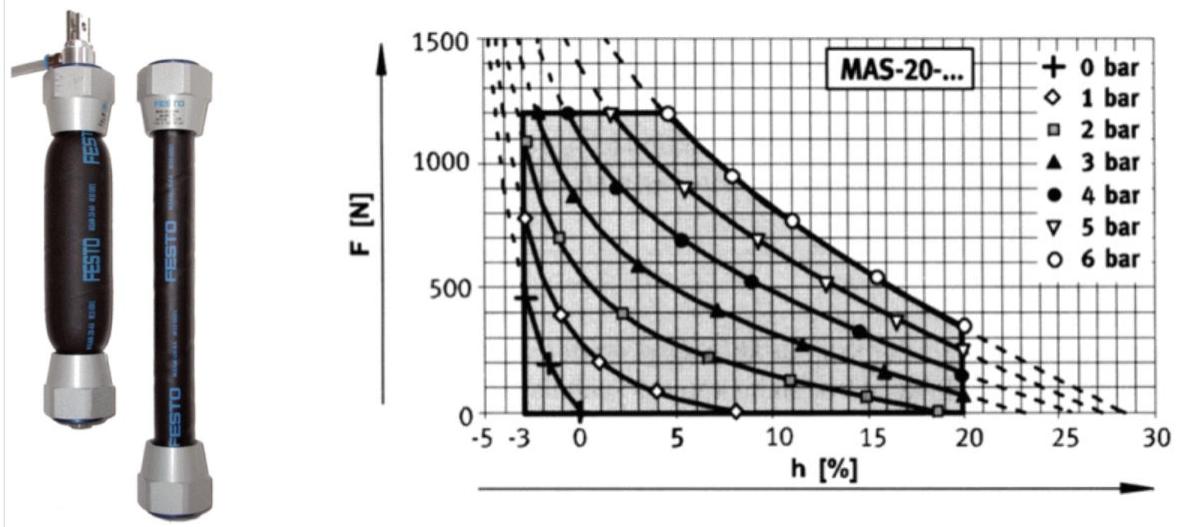


Figure 2.11-Illustration of load vs stroke graph at different pressures for a Festo fluidic muscle from source [13]

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.

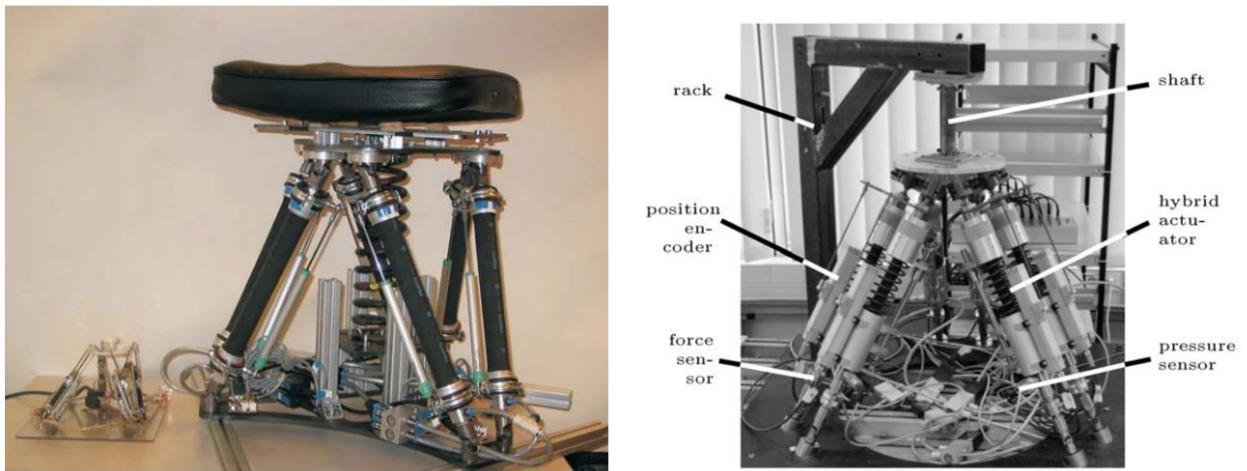


Figure 2.12-Past examples from papers of Stewart motion platforms using fluidic muscles as actuators.

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.

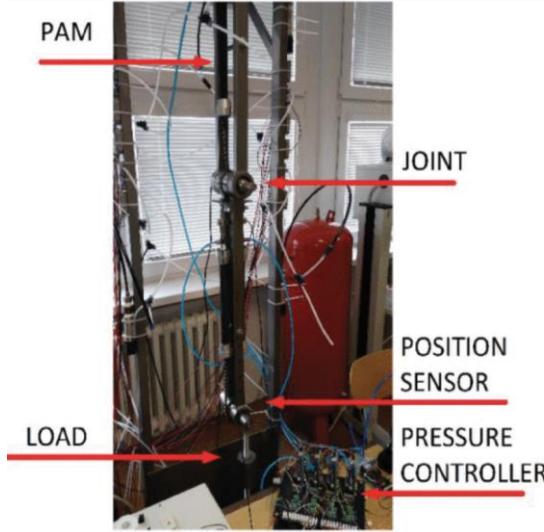


Figure 2.13-Illustration from paper [17] of a planar robotic arm controller by two fluidic muscles

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.

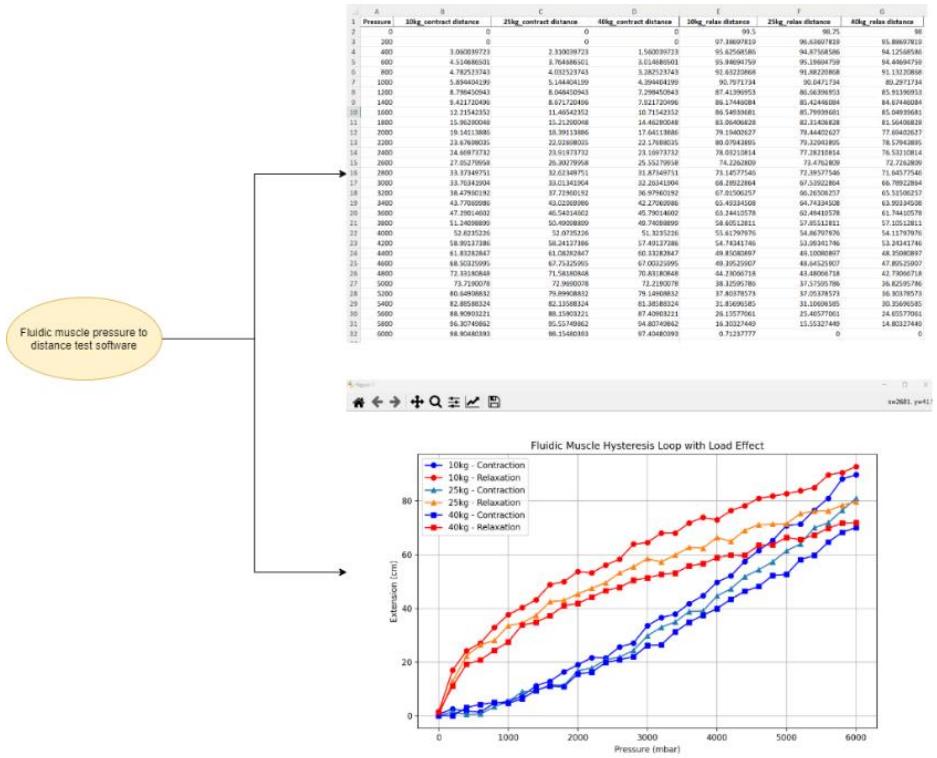


Figure 2.14-Illustration of results the program aims to provide with in the Week 8 experiments.

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.

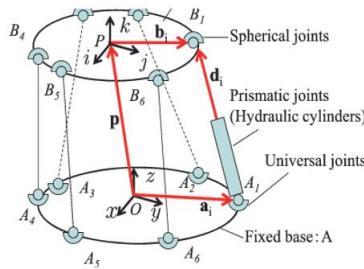


Figure 2.15-Inverse kinematics of a Stewart Platform illustration.

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.

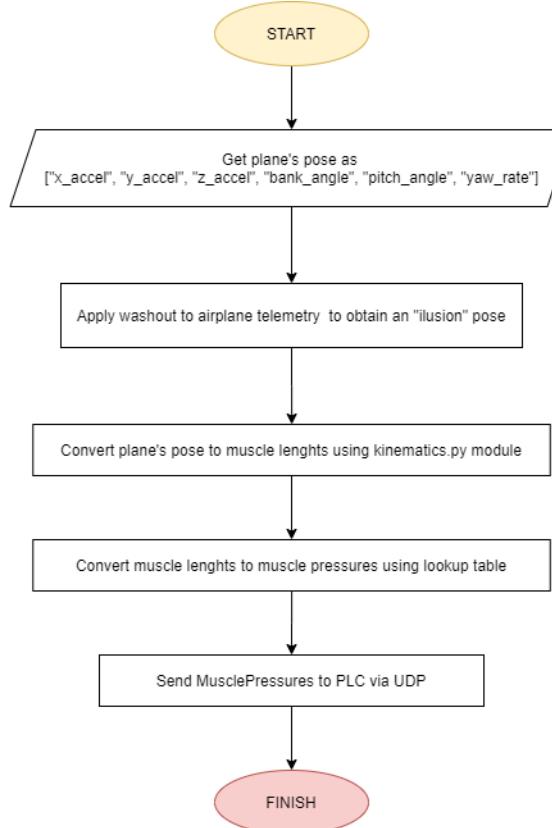


Figure 2.16-Plane Pose to muscle distances accounting for washout. This graph is from around project week 5.

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.

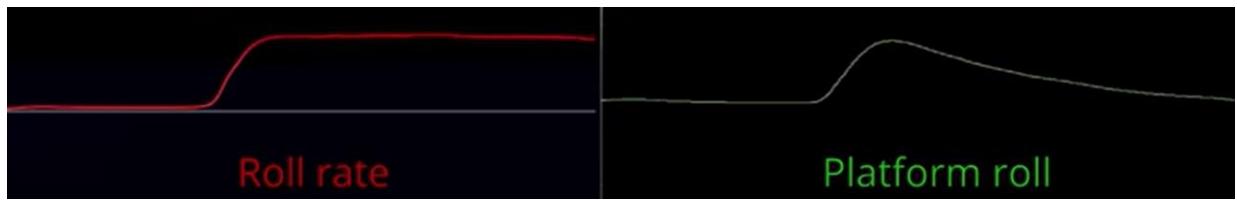
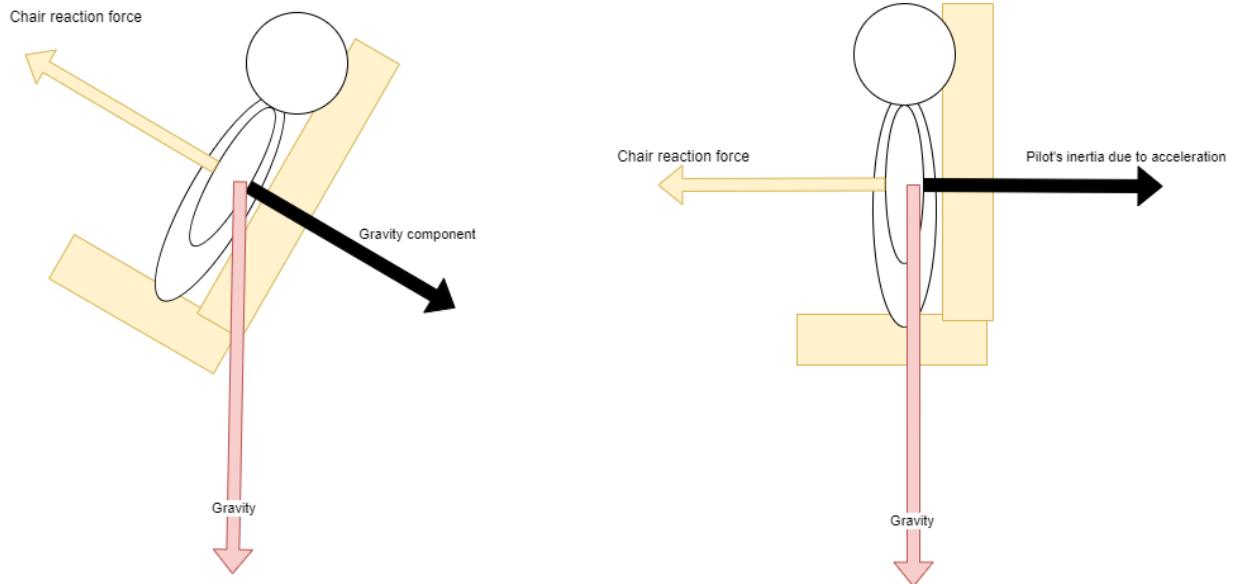


Figure 2.17-Washout filter illustration for roll and pitch.

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.



Motion Platform

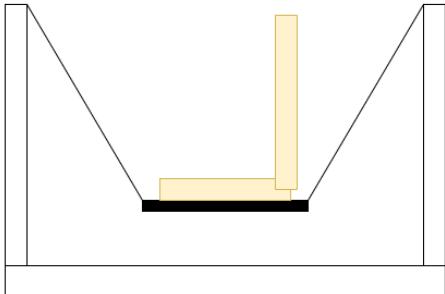
Airplane Accelerating Forwards

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

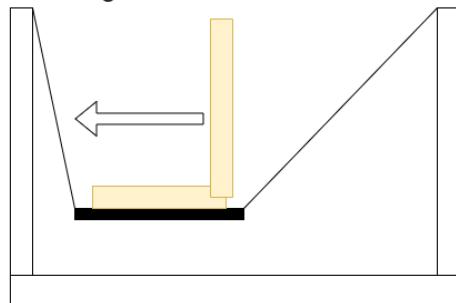
Figure 2.19 illustrates in better detail how the platform will create an illusion of acceleration during take-off by directly surging forwards for immediate acceleration and slowly tilting over time to create an illusion of constant acceleration. The user feels as if they are accelerating because gravity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

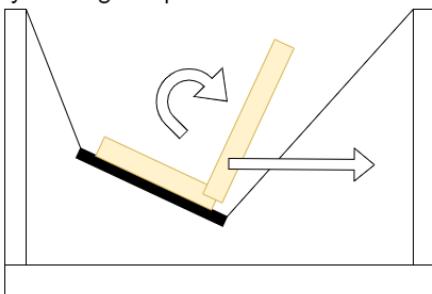
1. Platform is stationary



2. Pilot quickly increases throttle to takeoff and platform surges forward



3. Platform slowly turns back to home position whilst slowly turning the pilot on its back



4. Platform remains tilted back to provide illusion of acceleration

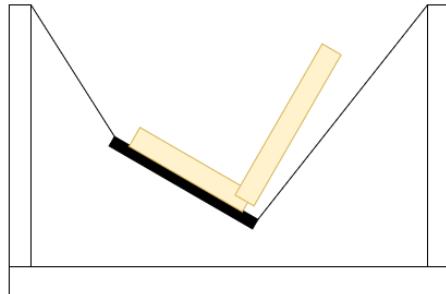


Figure 2.19-Illustration of washout for low frequency transient acceleration.

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.

- 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.

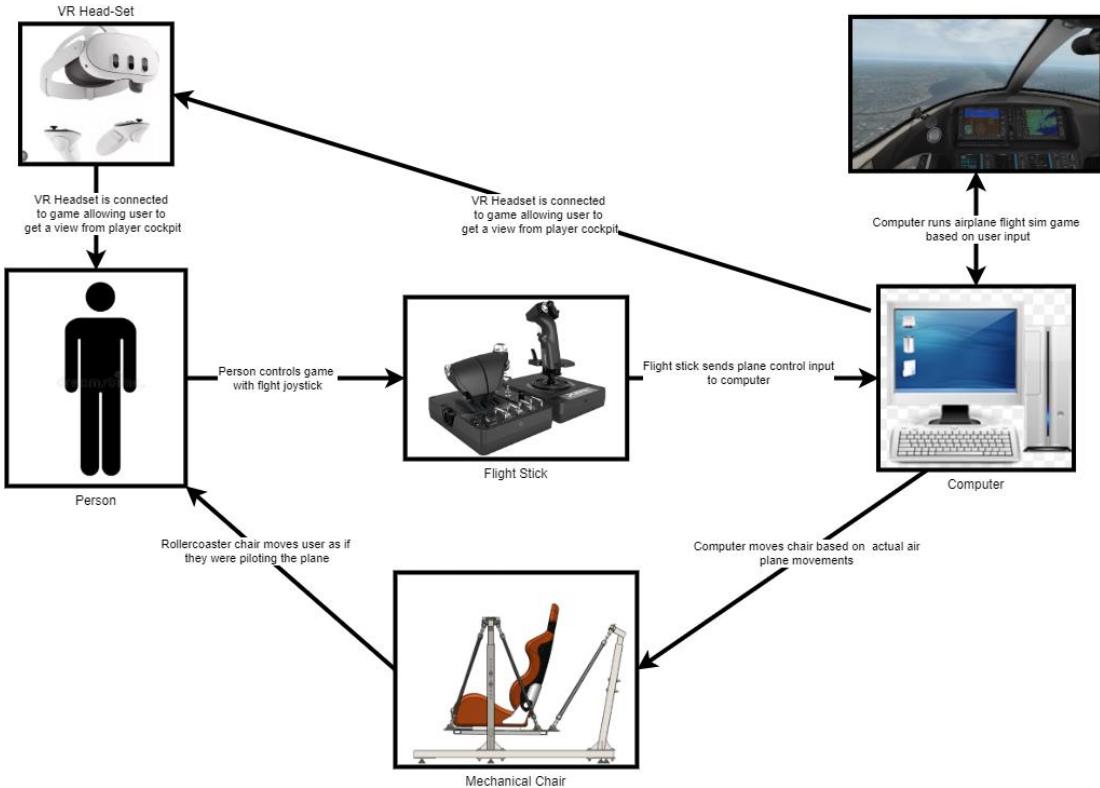


Figure 3.2-High level illustration of how all the components in flight motion platform are meant to interact together

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.

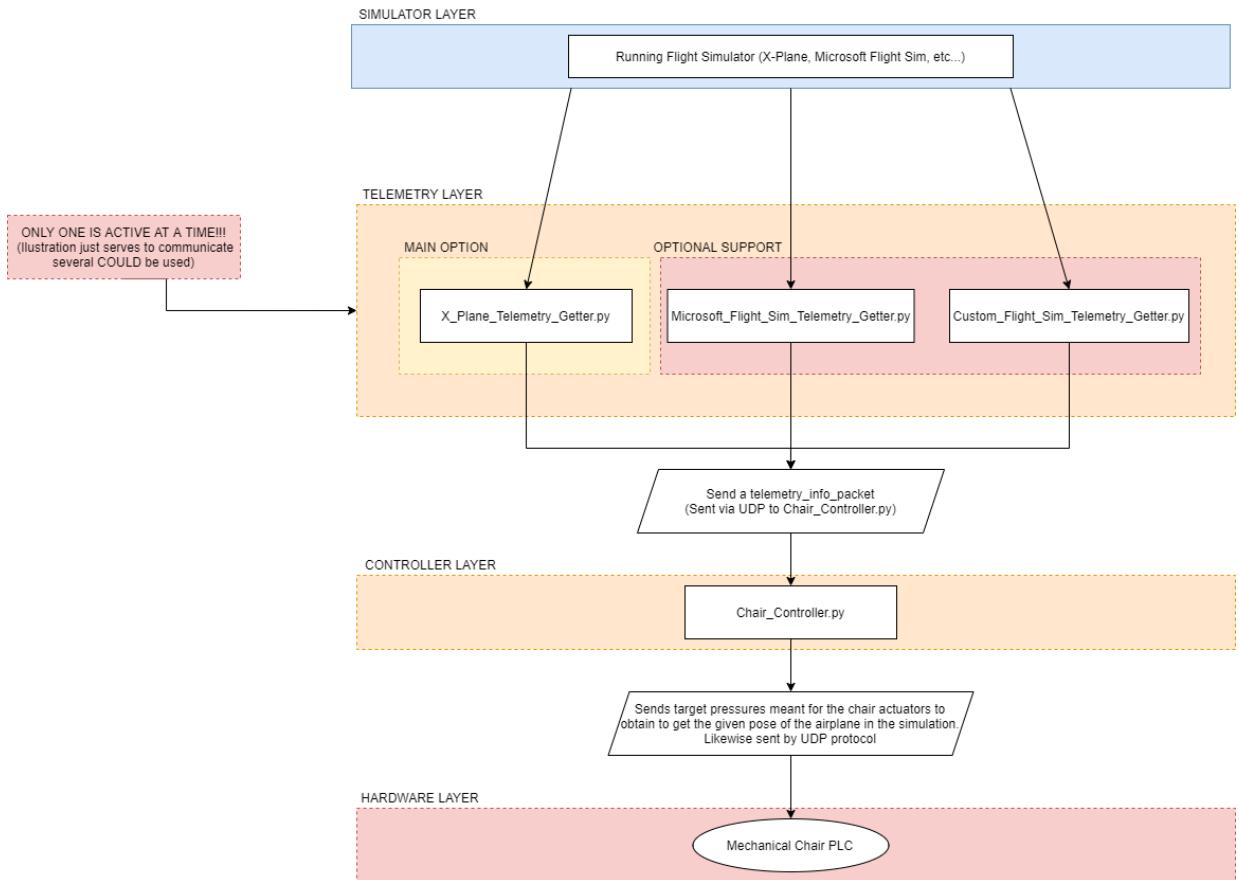


Figure 3.3-High level outline showcasing how the flight simulator is to interact with control software (python client) and thereby the mechanical chair(flight motion platform)

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.

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.

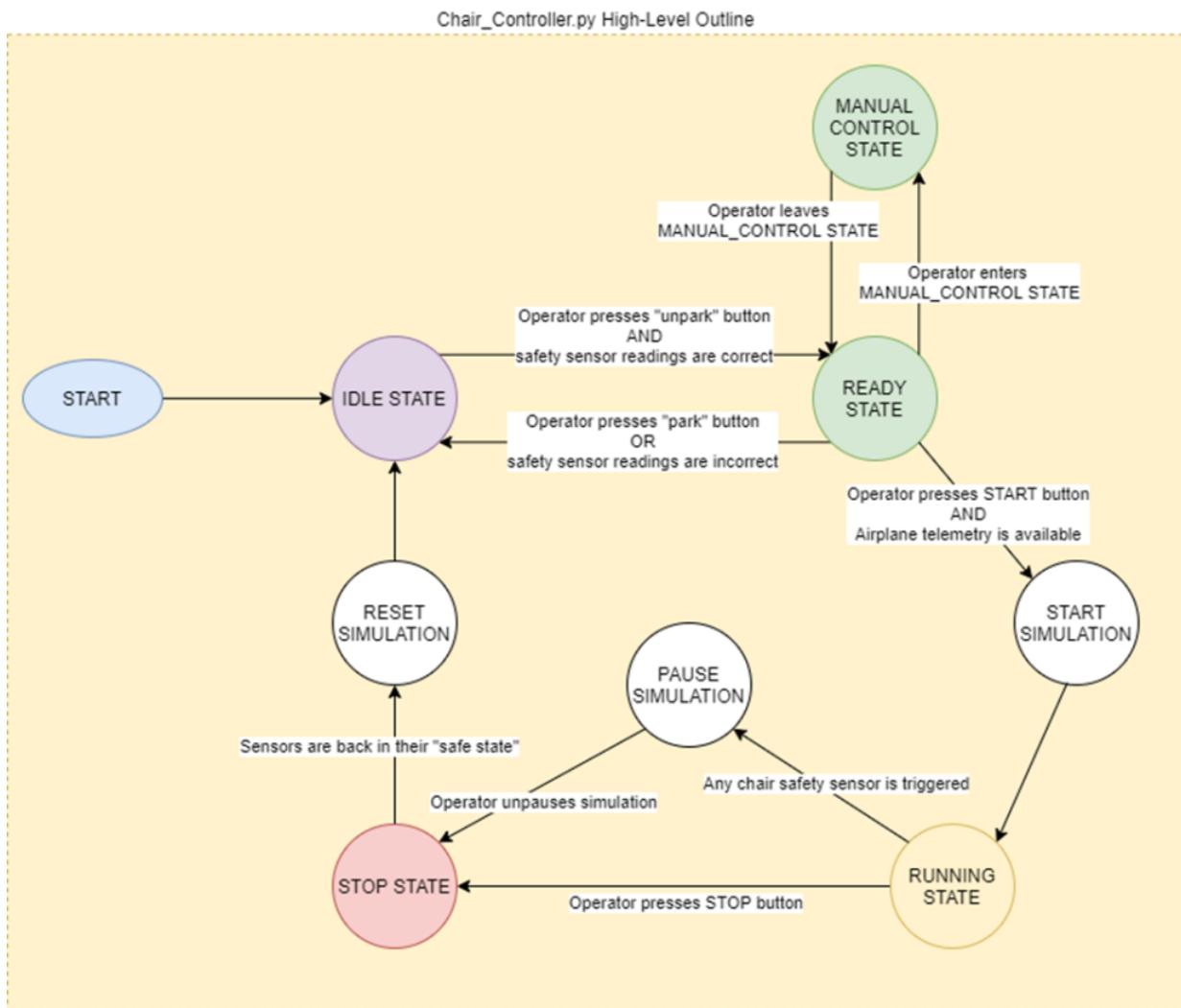


Figure 4.1-State machine diagram of flight motion platform control software.

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 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, pitch 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.

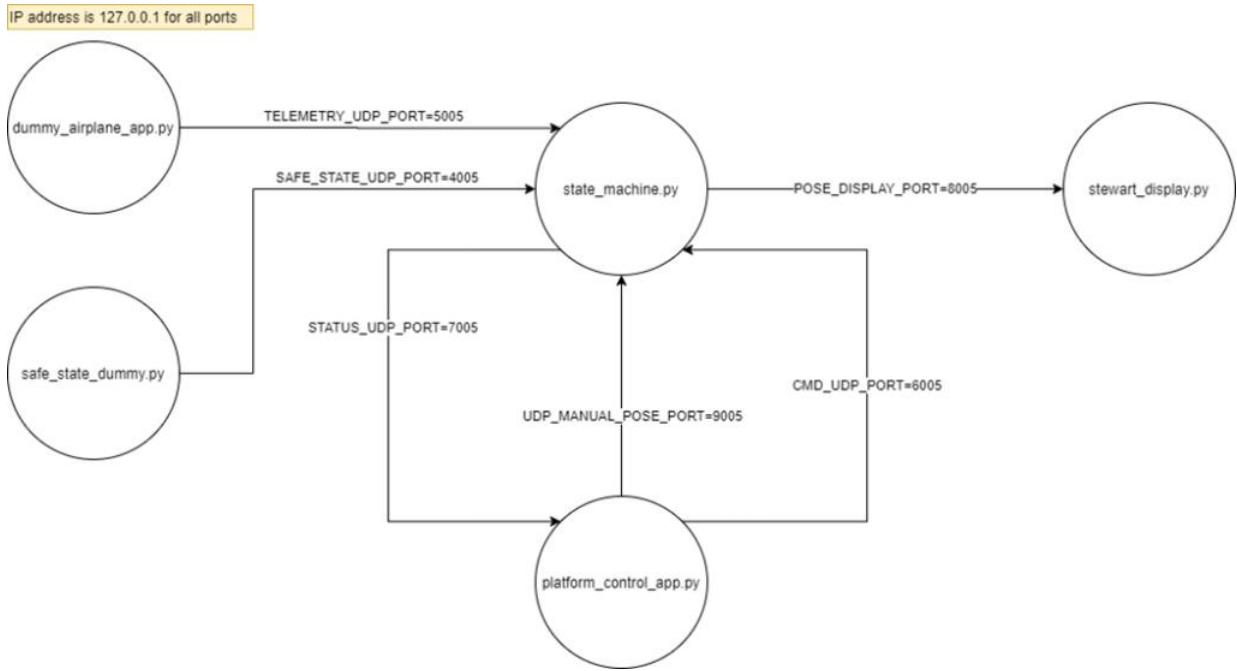


Figure 4.2-Node diagram of entire software.

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	<p>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.</p> <p>The data is sent as an array via “TELEMETRY_UDP_PORT” where the array is [xaccel, yacc, 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).</p>
safe_state_dummy.py	<p>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.</p>
state_machine.py	<p>This script implements the state machine described in chapter 4.1</p> <p>This state machine executes at a variable cycle rate set to 20ms at the moment.</p> <p>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).</p> <p>At the moment, the UDP command format is [“state_transition”, gain_xaccel, gain_yacc, gain_zacc, 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.</p> <p>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.</p> <p>Finally, this state machine node listens to the airplane's telemetry from the dummy_airplane_app.py node and uses it as previously described.</p>
platform_control_app.py	<p>This app is 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”.</p>

stewart_display.py	<p>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.</p> <p>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.</p>
---------------------------	--

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

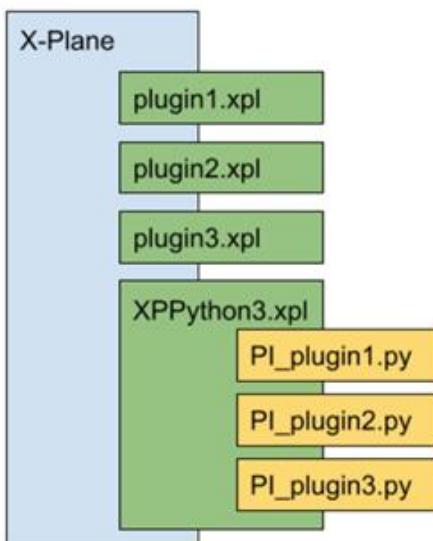
Once a basic skeletal system for the hardware was established, the next step was to replace the “dummy_airplane_app.py” with a program that sent X-Plane’s airplane telemetry to the same port in the same format as of now. In order to do this, a plugin for X-Plane was written. X-Plane supports plugins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

What is XPPython3

XPPython3 is a standard X-Plane compiled plugin, which loads and executes python plugins. Messages sent by X-Plane to plugins are received by XPPython3 which then forwards messages to each of the python plugins, translating native “C” language data types into relevant python datatypes.

Function calls made from python plugins are handled by XPPython3, which translates python data types to C and then makes the equivalent call into X-Plane using the X-Plane SDK. From X-Plane’s perspective, XPPython3 is a single plugin: Only XPPython3 knows about the individual python plugins.



The XPPython3 programming interface matches the X-Plane C SDK interface, using the same concepts. So if you’re familiar with the SDK you’ll find the python interface pretty easy. Similarly, you should be able to translate examples from C into python (and vice-versa) without much effort. [1]

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.

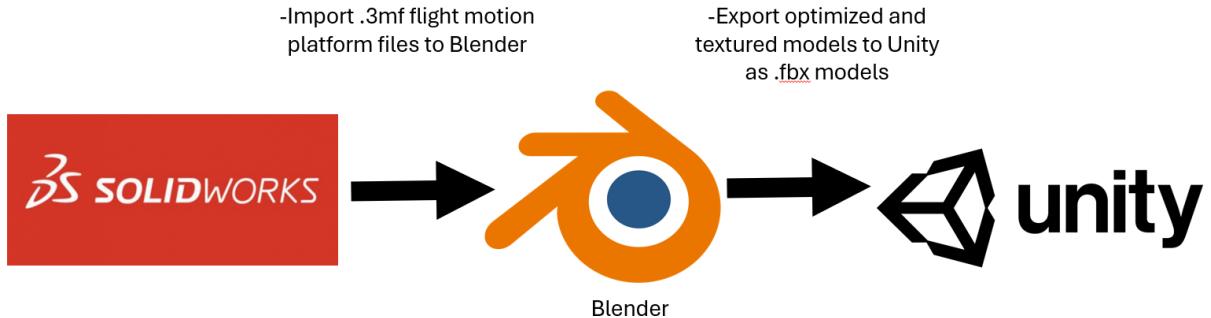
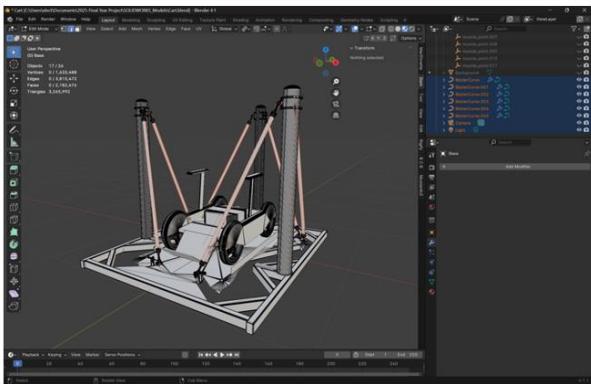


Figure 5.2-Outline of workflow taken to import the CAD models of the flight motion platform from SOLIDWORKS to Unity

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.

Before: +2M triangle count



After: Around 220k triangle count (over 93% reduction)

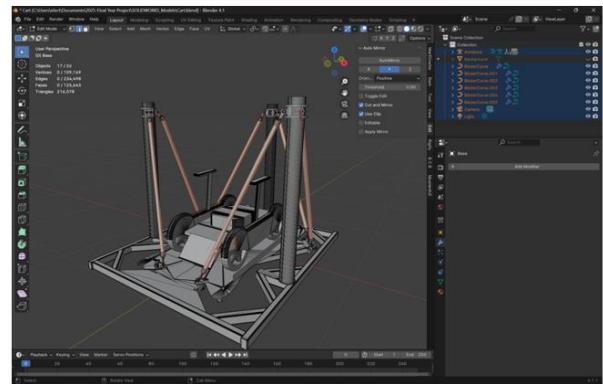


Figure 5.3-Before and after optimizing model geometry in Blender to export to Unity.

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 be used to make a brief proof of concept animation. This is 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.

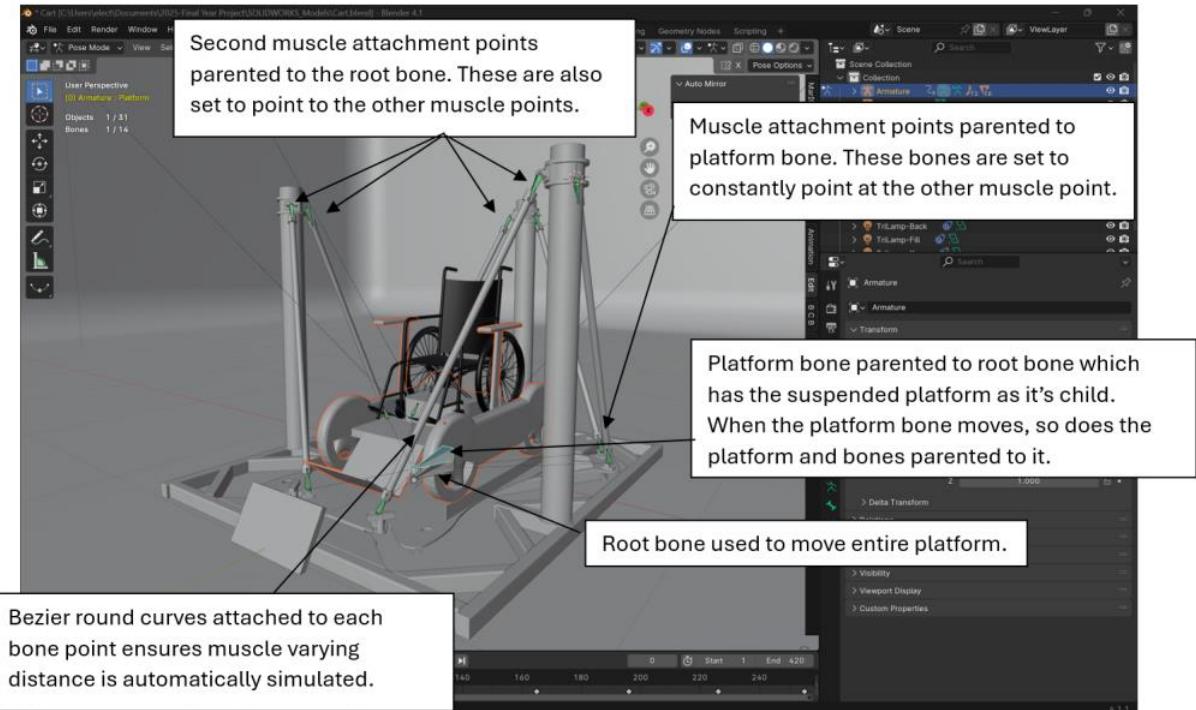


Figure 5.4-Figure showcasing the rig used to simulate the moving platform.

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.

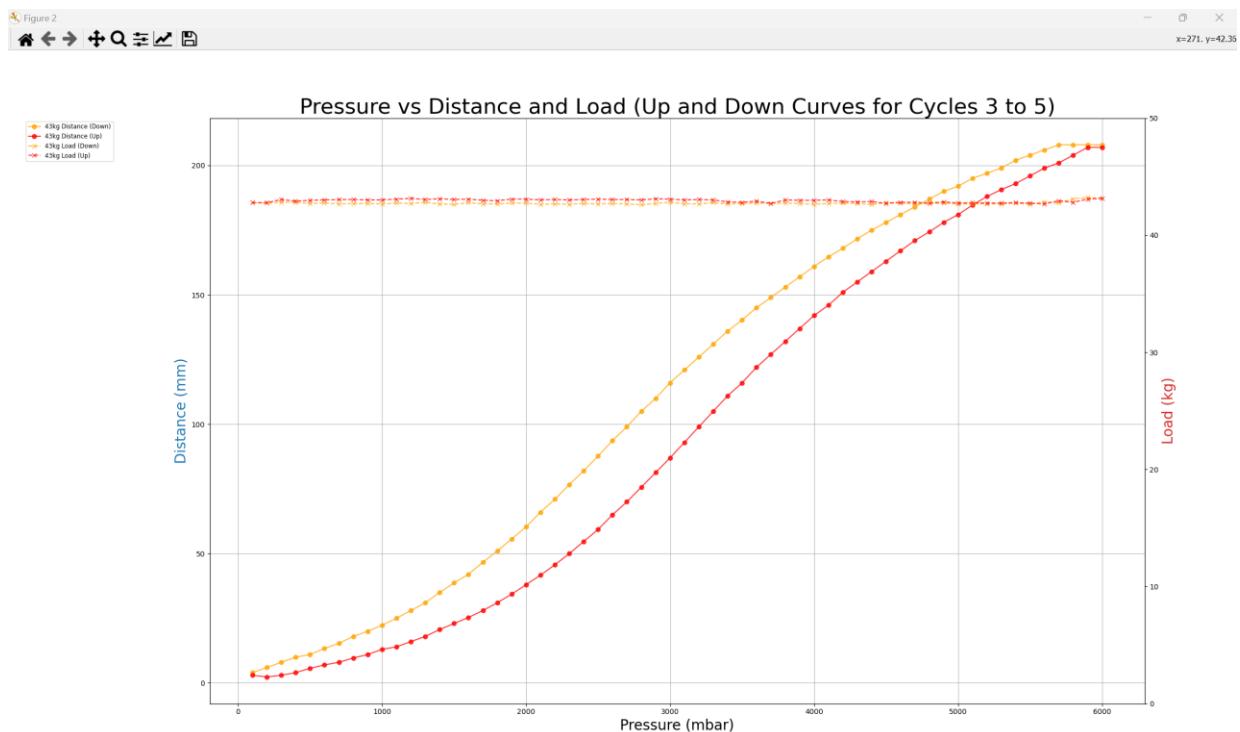


Figure 6.1-Example chart of muscle hysteresis loop.

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.

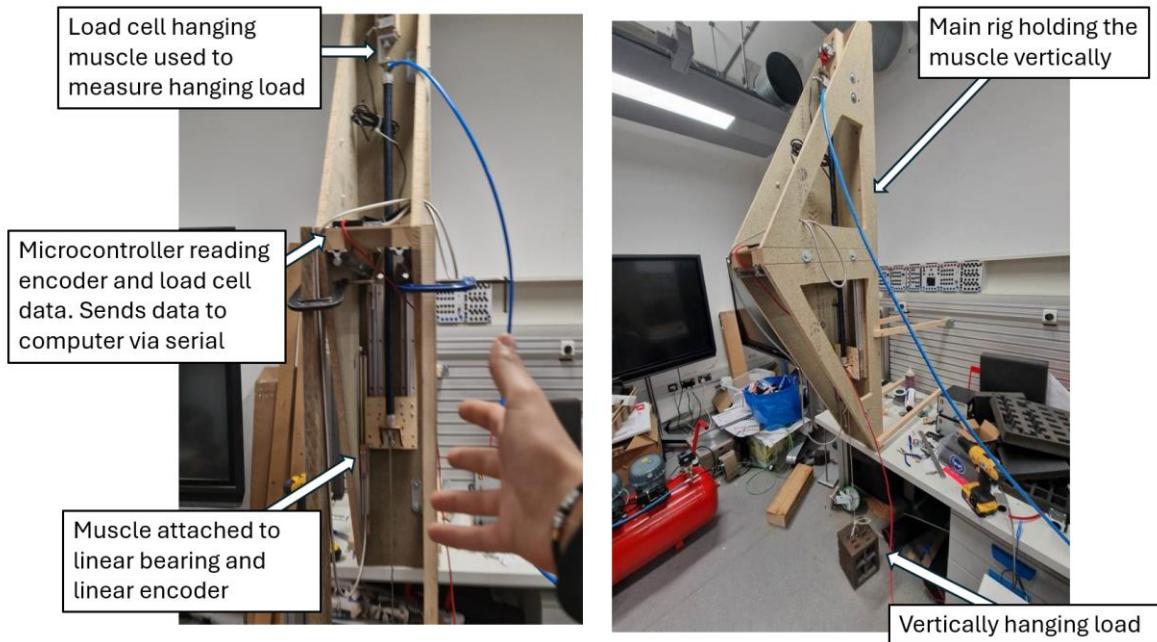


Figure 6.2-Illustration of calibration rig used to obtain muscle pressure-to-distance data.

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.

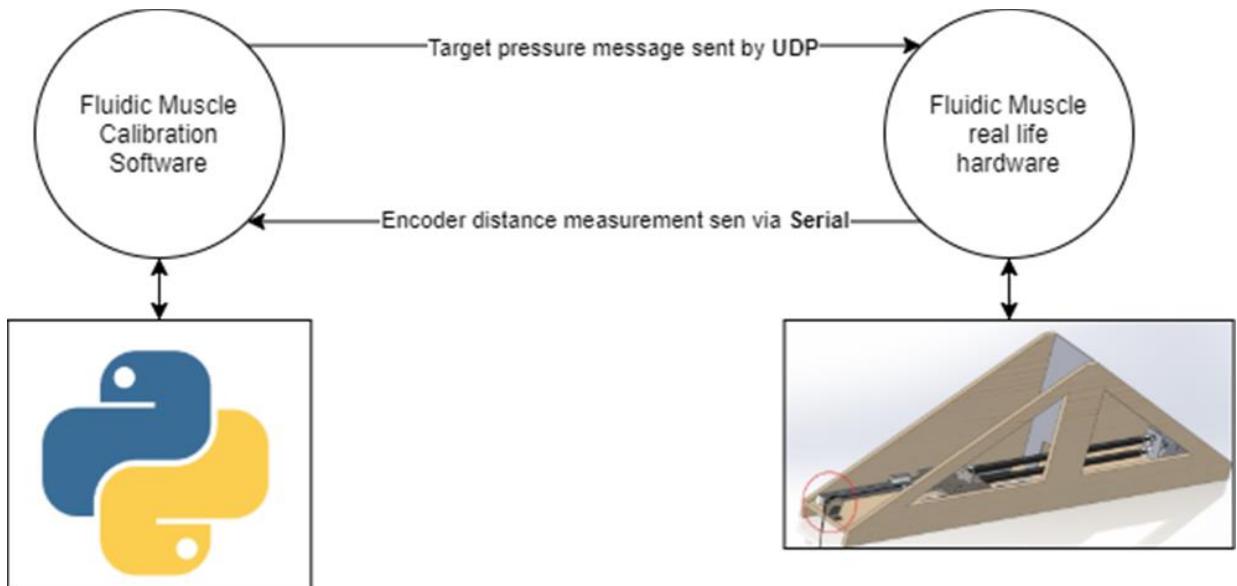


Figure 6.3-Illustration of how calibration program and test rig will communicate.

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.

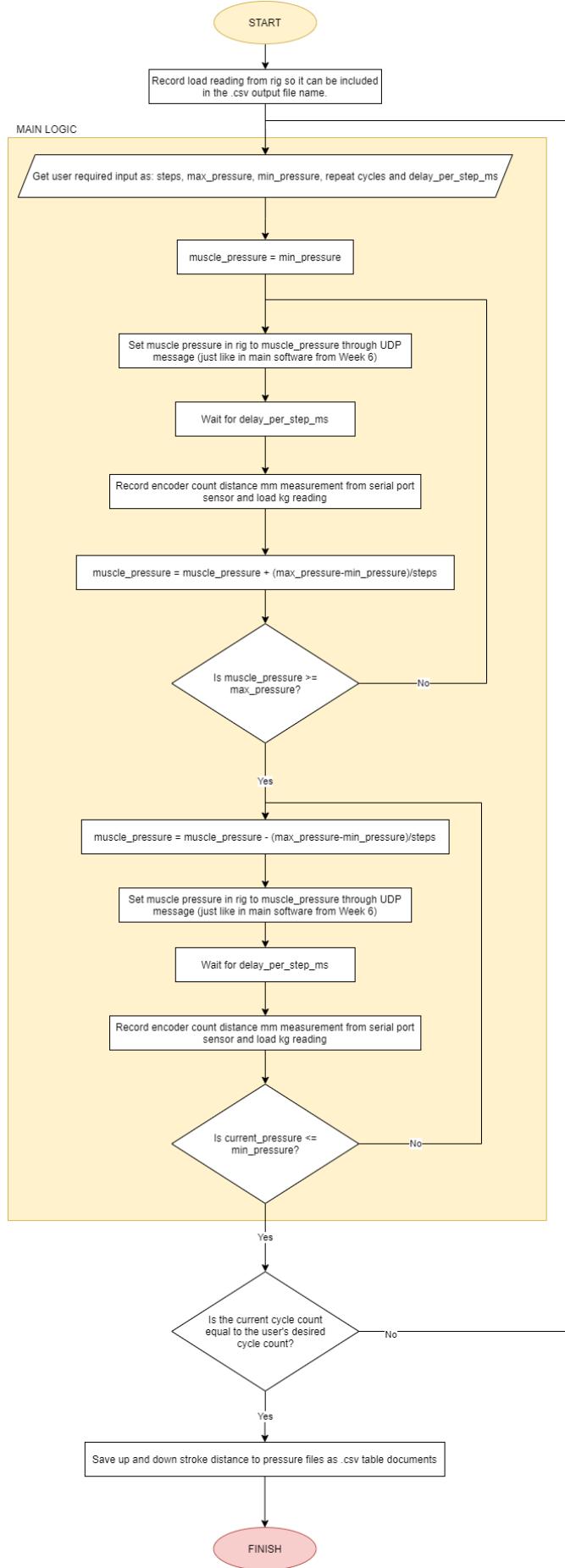


Figure 6.4-Flowchart outlining the working logic of the calibration software.

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

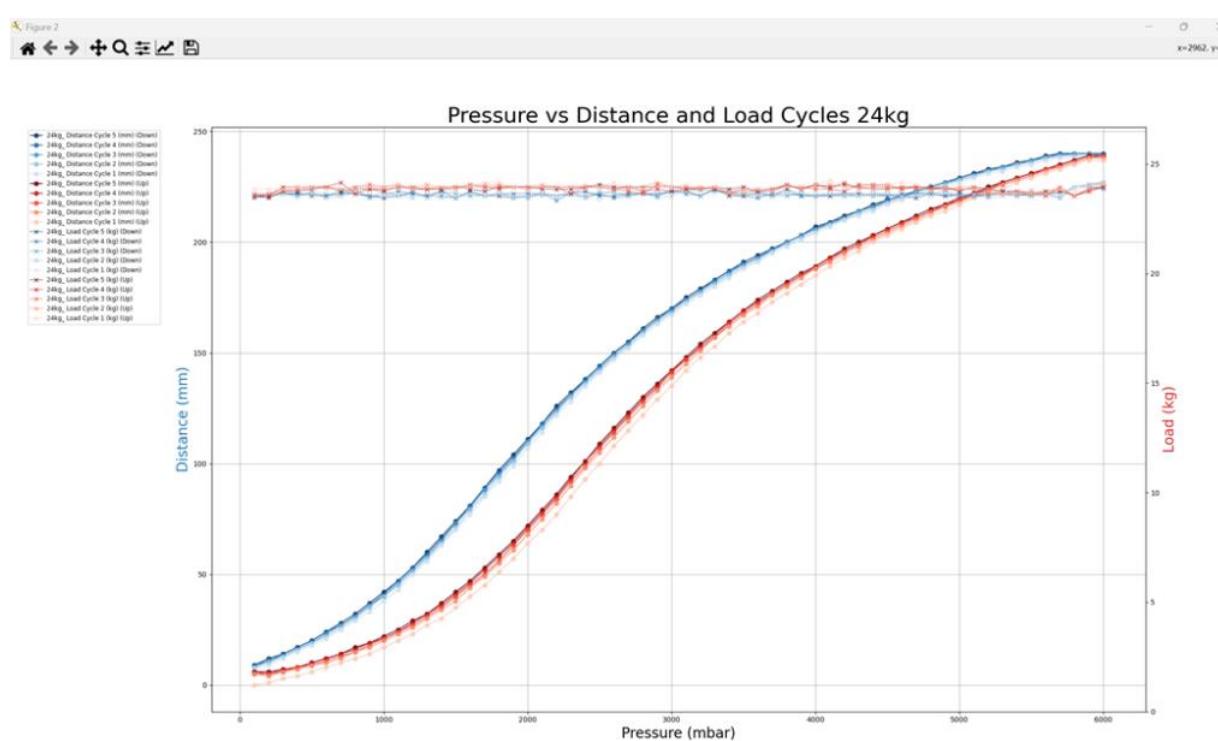
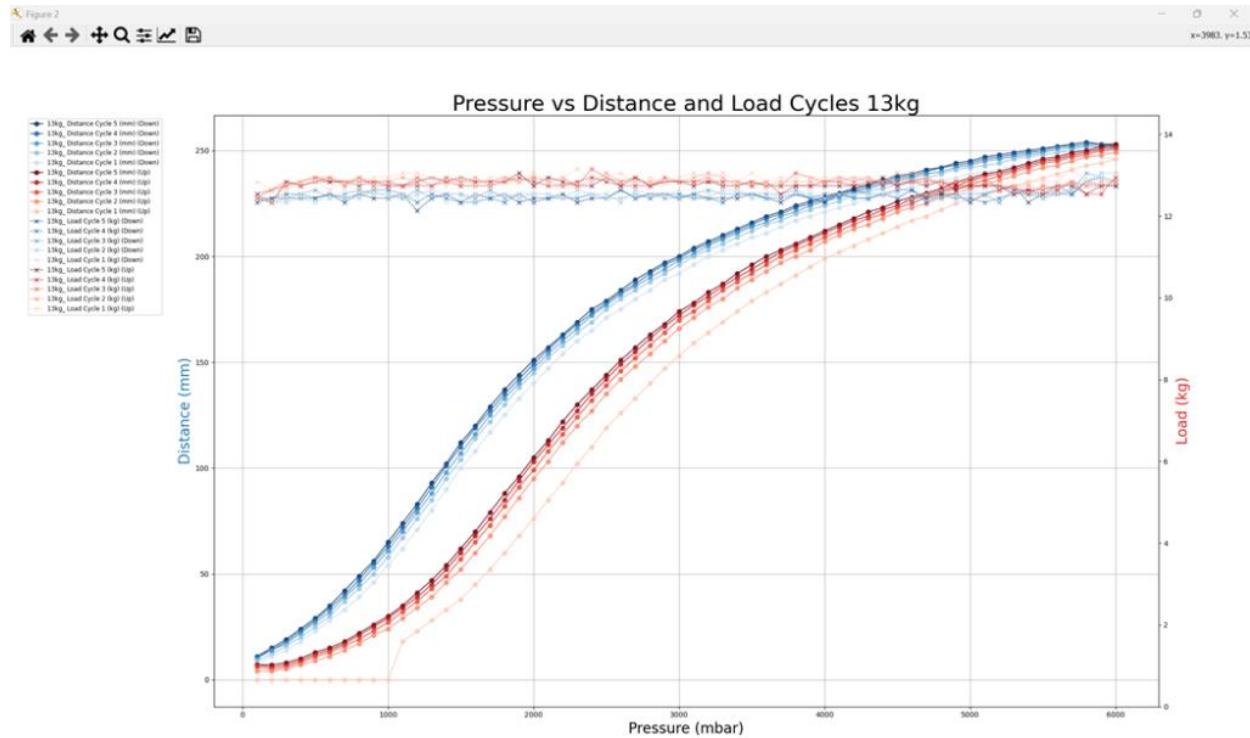
The script in charge of obtaining the pressure to distance calibration data of the muscle at different loads was called “calibration_data.py”. The GUI and code snippets for this software are included in Appendix 17 and Appendix 18 respectively. Likewise the GUI and code snippets for the “csv_muscle_plotter.py” 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.

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



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

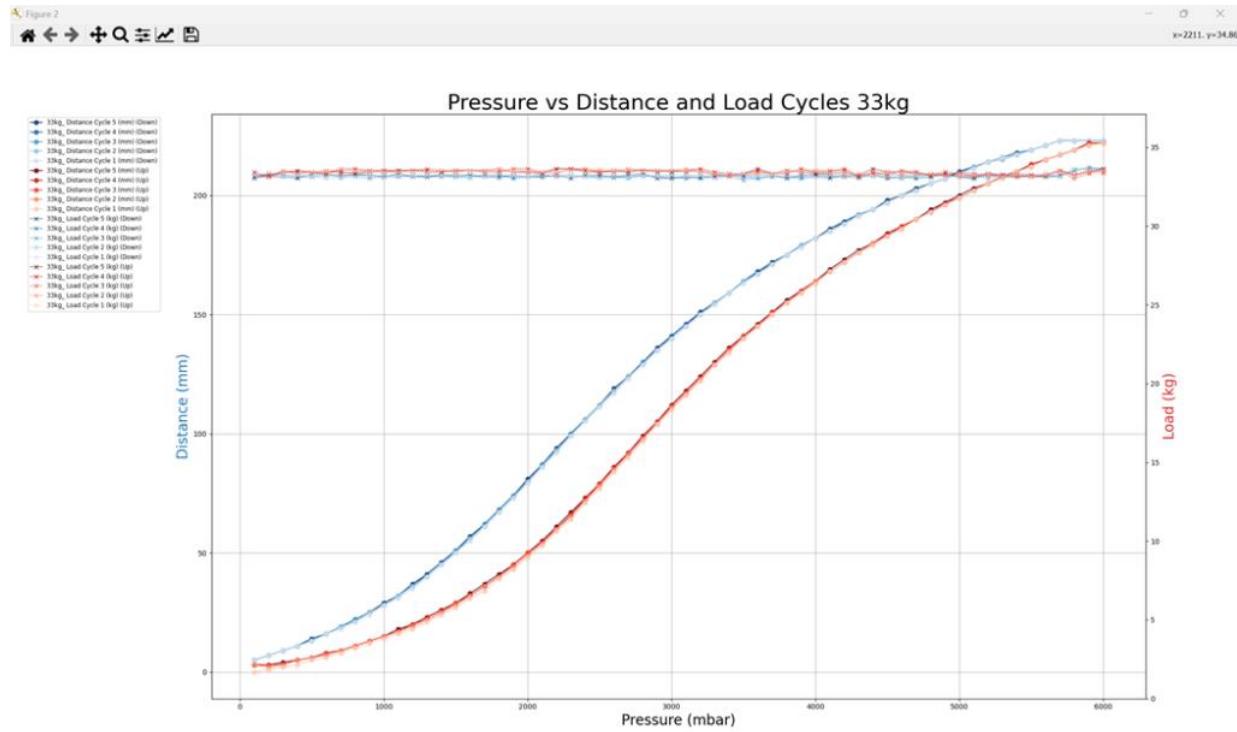


Figure 6.7-Pressure (mbar) vs Distance (mm) plot for 33kg load.

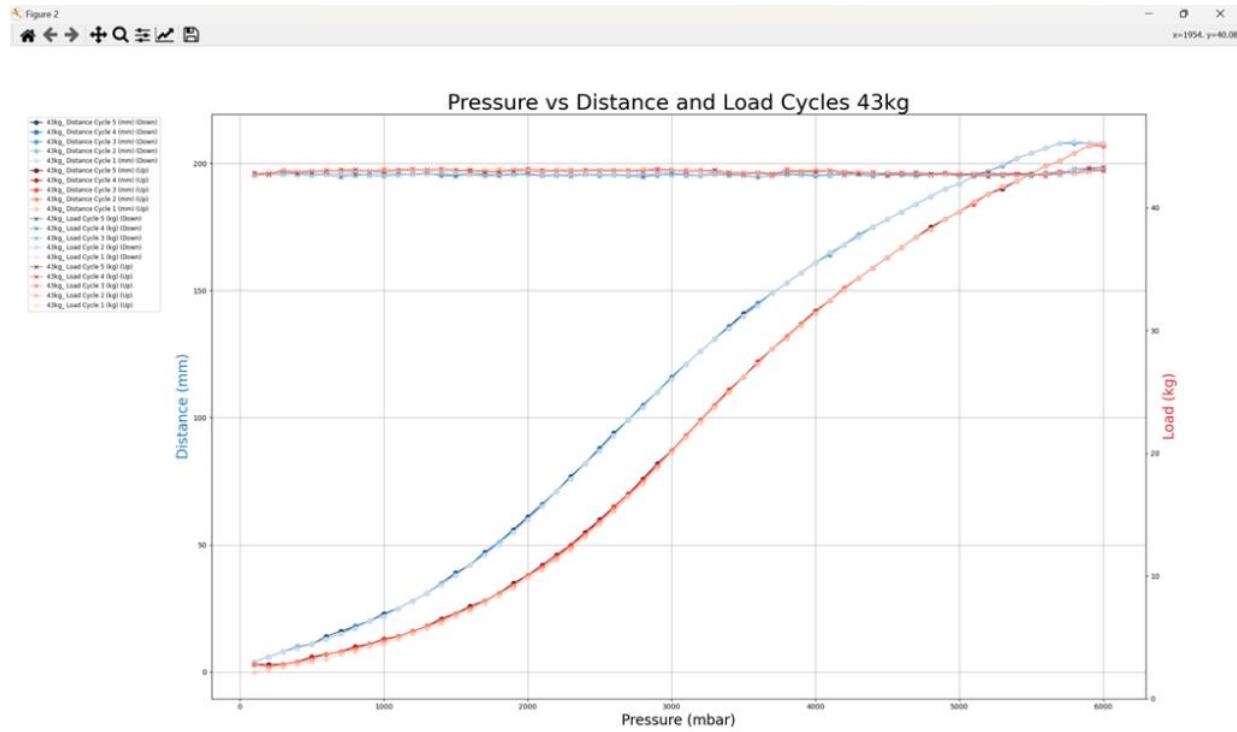


Figure 6.8-Pressure (mbar) vs Distance (mm) plot for 43kg load.

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.

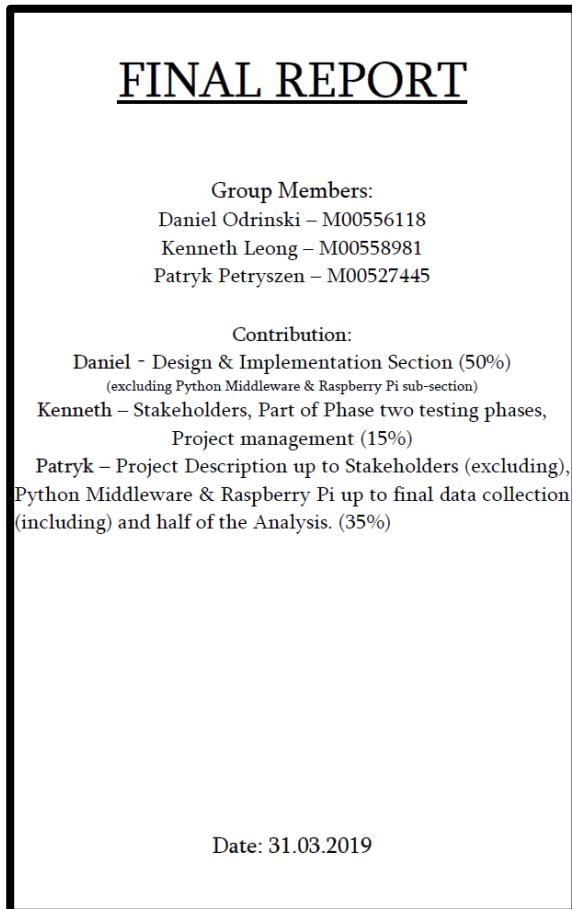


Figure 6.9-Front cover of previous student paper.

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.

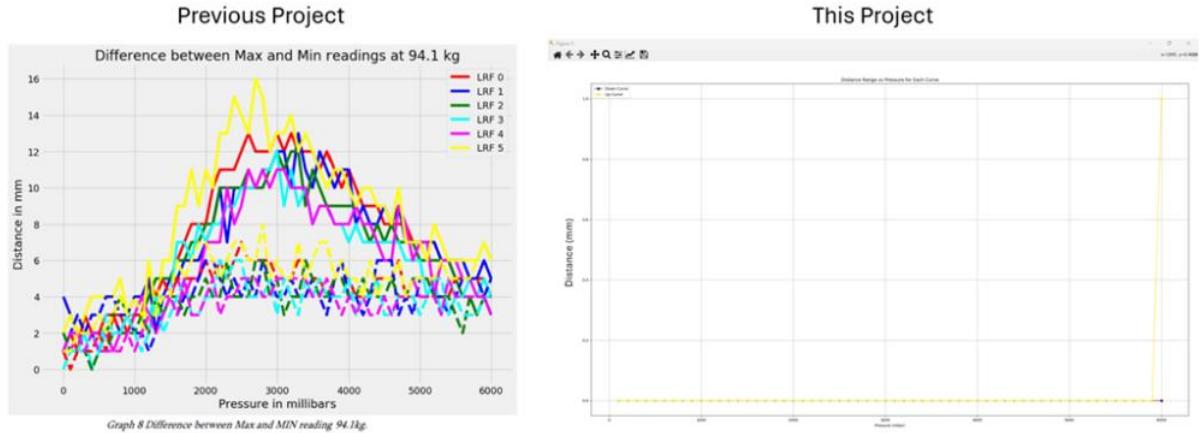


Figure 6.10-Comparison between previous project distance reading range per pressure and this project's.

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 an inclinometer is recommended 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 beneficial 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.

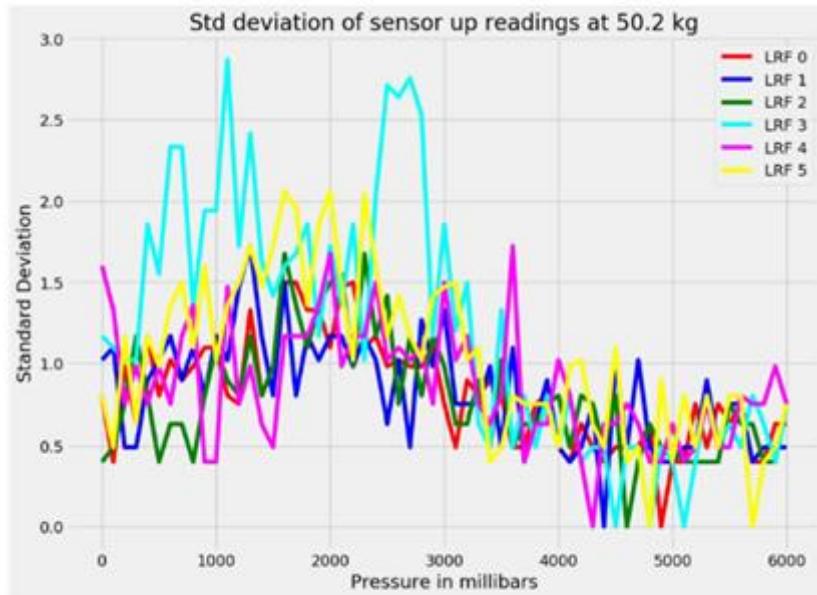
Figure 6.11-Conclusion section of previous student's project on the rollercoaster chair.

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).

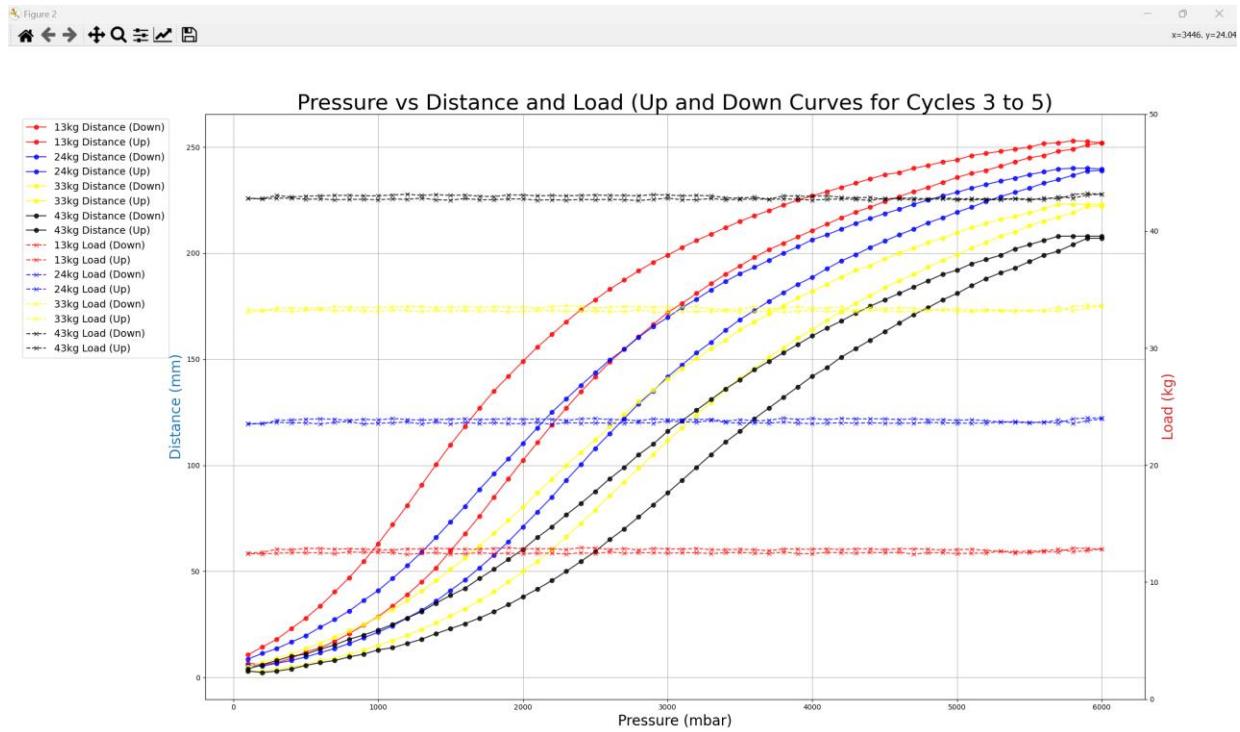


Figure 6.13-Final averaged distance and load cycle data plotted simultaneously on a graph. Cycles 3, 4 and 5 were counted and the first two cycles of each load calibration set was discarded.

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.

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

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	42.60958205912334	42.813455657492355
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	42.71151885830785
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.

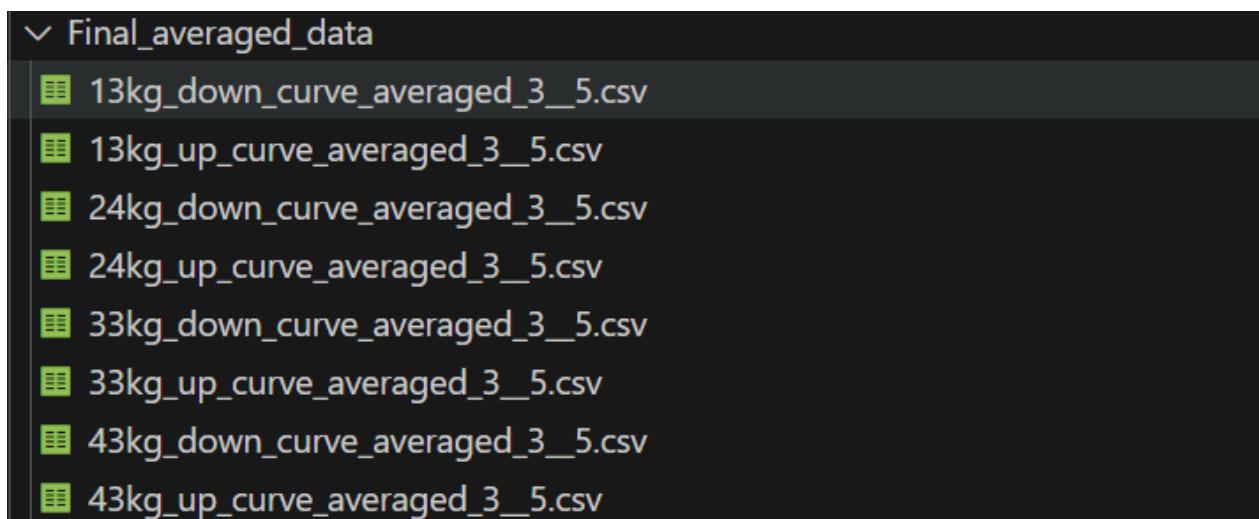


Figure 7.2-Averaged load muscle calibration files from cycles 3 to 5.

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	5.6666666666666667	42.98335032279986
600		7 43.01732925586136
700		8 43.05130818892286
800	9.6666666666666666	43.05130818892286
900		11 43.01732925586136
1000		13 43.01732925586136
1100		14 43.08528712198436
1200		16 43.15324498810738
1300		18 43.05130818892286
1400	20.6666666666666668	43.11926605504587
1500		23 43.05130818892286
1600	25.33333333333332	43.08528712198436
1700		28 42.98335032279985
1800		31 42.94937138973835
1900	34.33333333333336	43.08528712198436
2000		38 43.08528712198436
2100	41.6666666666666664	43.01732925586136
2200	45.6666666666666664	43.05130818892286
2300		50 43.01732925586136
2400	54.6666666666666664	43.05130818892286
2500	59.33333333333336	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 hysteresis.
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,
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.

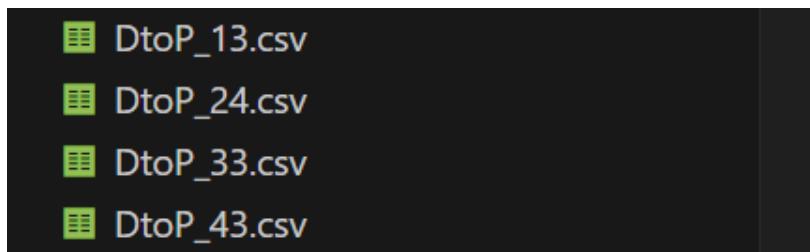


Figure 7.5-output DtoP files for all the calibrated loads.

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
```

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.

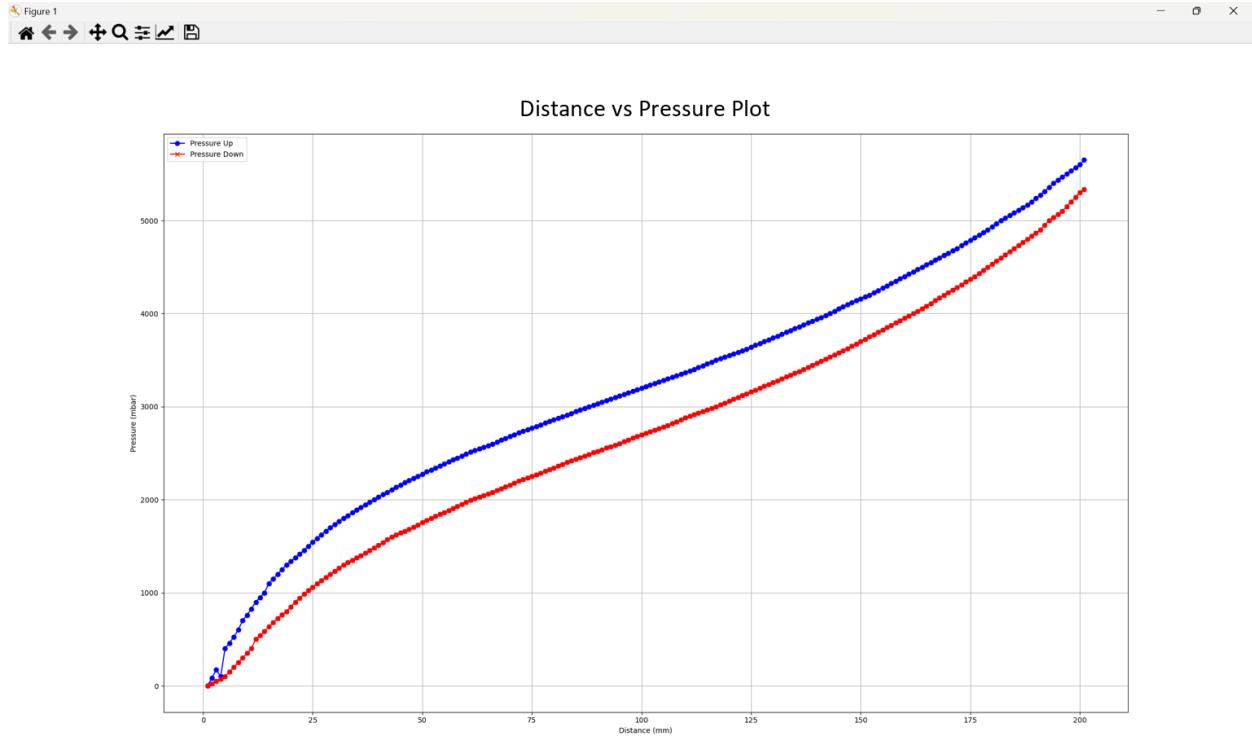


Figure 7.7-PtoD plot of 43kg csv file.

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.

```

256     def merge_d_to_p(self, infnames, outfname):
257         # creates distance to pressure curves file using values from infnames
258         # input file format:
259         # header as: weight=X where X is weight in kg
260         # row 1: comma separated list of 200 up-going pressures for mm distances from 0 to 199
261         # row 2: comma separated list of 200 down-going pressures for mm distances from 0 to 199
262         weights = []
263         up_d_to_p = []
264         down_d_to_p = []
265         for fname in infnames:
266             with open(fname) as fp:
267                 header = fp.readline()
268                 if 'weight=' in header:
269                     weights.append(int(header.split('=')[1]))
270                     up = fp.readline()
271                     values = [int(round(float(i))) for i in up.split(',')]
272                     up_d_to_p.append(values)
273                     down = fp.readline()
274                     values = [int(round(float(i))) for i in down.split(',')]
275                     down_d_to_p.append(values)
276
277         if len(weights) > 0:
278             header = '# weights,' + ','.join(str(n) for n in weights)
279             combined_d_to_p= []
280             for i in range (len(weights)):
281                 combined_d_to_p.append(up_d_to_p[i])
282             for i in range (len(weights)):
283                 combined_d_to_p.append(down_d_to_p[i])
284             with open(outfname, "w") as fp:
285                 fp.write(header + '\n')
286                 for i in range (len(weights)*2): # write up then down
287                     fp.write( ','.join(str(n) for n in combined_d_to_p[i]) + '\n')
288             else:
289                 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.

```

1  from calibrate.d_to_p_prep import D_to_P_prep
2
3  prepper = D_to_P_prep(200)
4  prepper.merge_d_to_p(["output\Final_averaged_data\DtoP_13.csv",
5                      "output\Final_averaged_data\DtoP_24.csv",
6                      "output\Final_averaged_data\DtoP_33.csv",
7                      "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.

```

1  # weights,13,24,33,43,,,,,
2  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
3  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
4  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,
5  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
6  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,
7  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
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
9  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
10

```

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 columns 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.

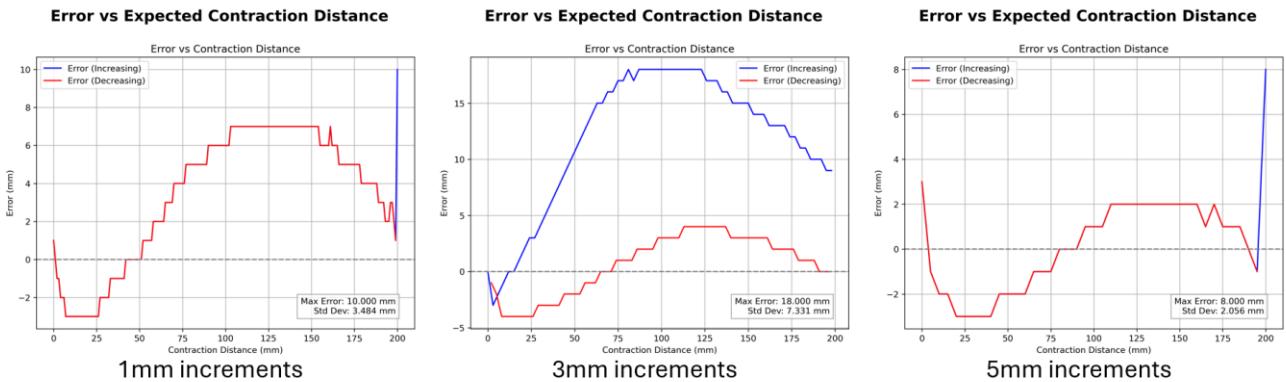


Figure 7.11-Plots of error vs set distance plots of merged DtoP table.

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 its 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 $\pm 25\text{mm}$. 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).

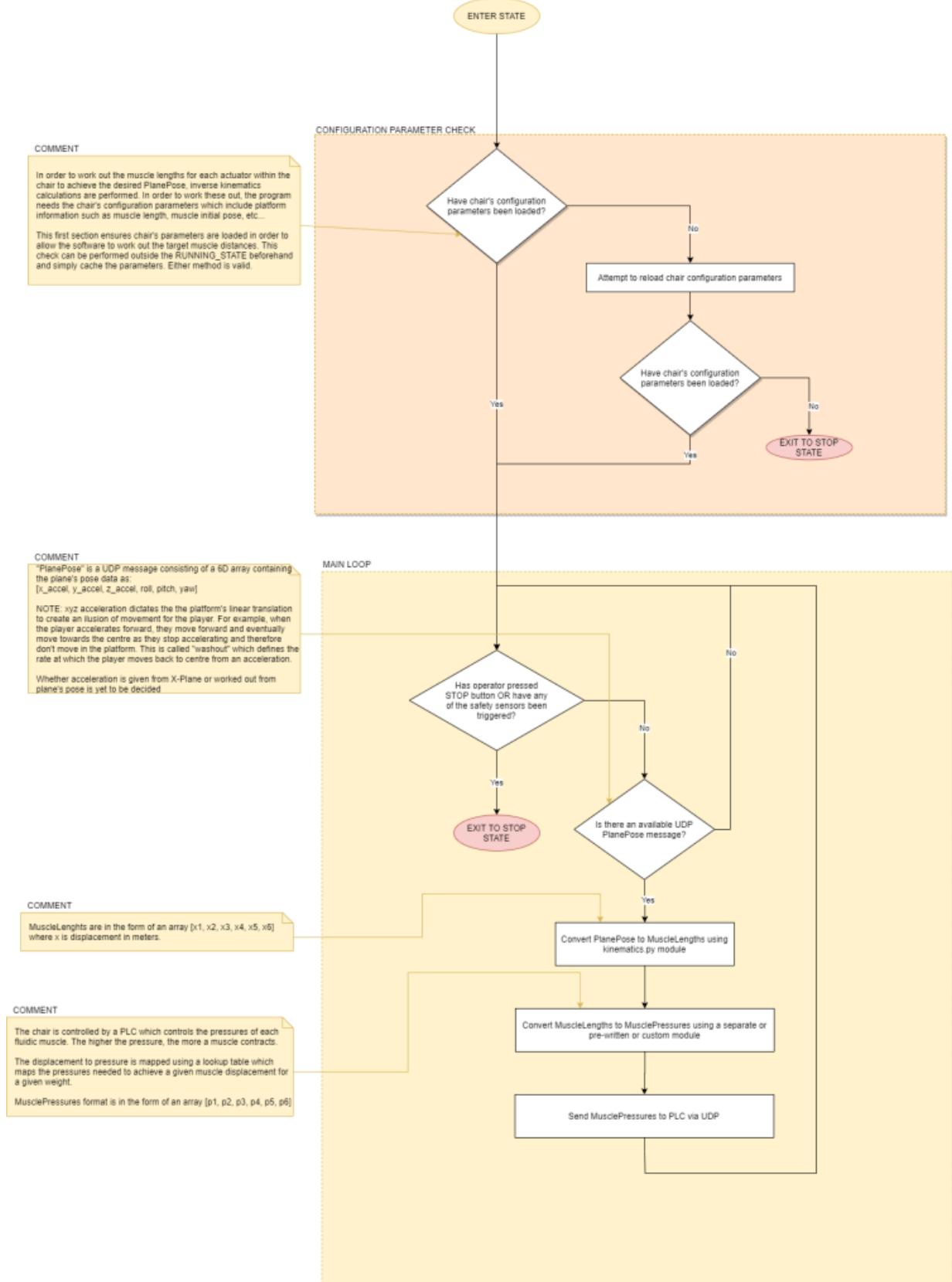
References

- [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=NSQDIUpArk>.
- [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.ln.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/>.

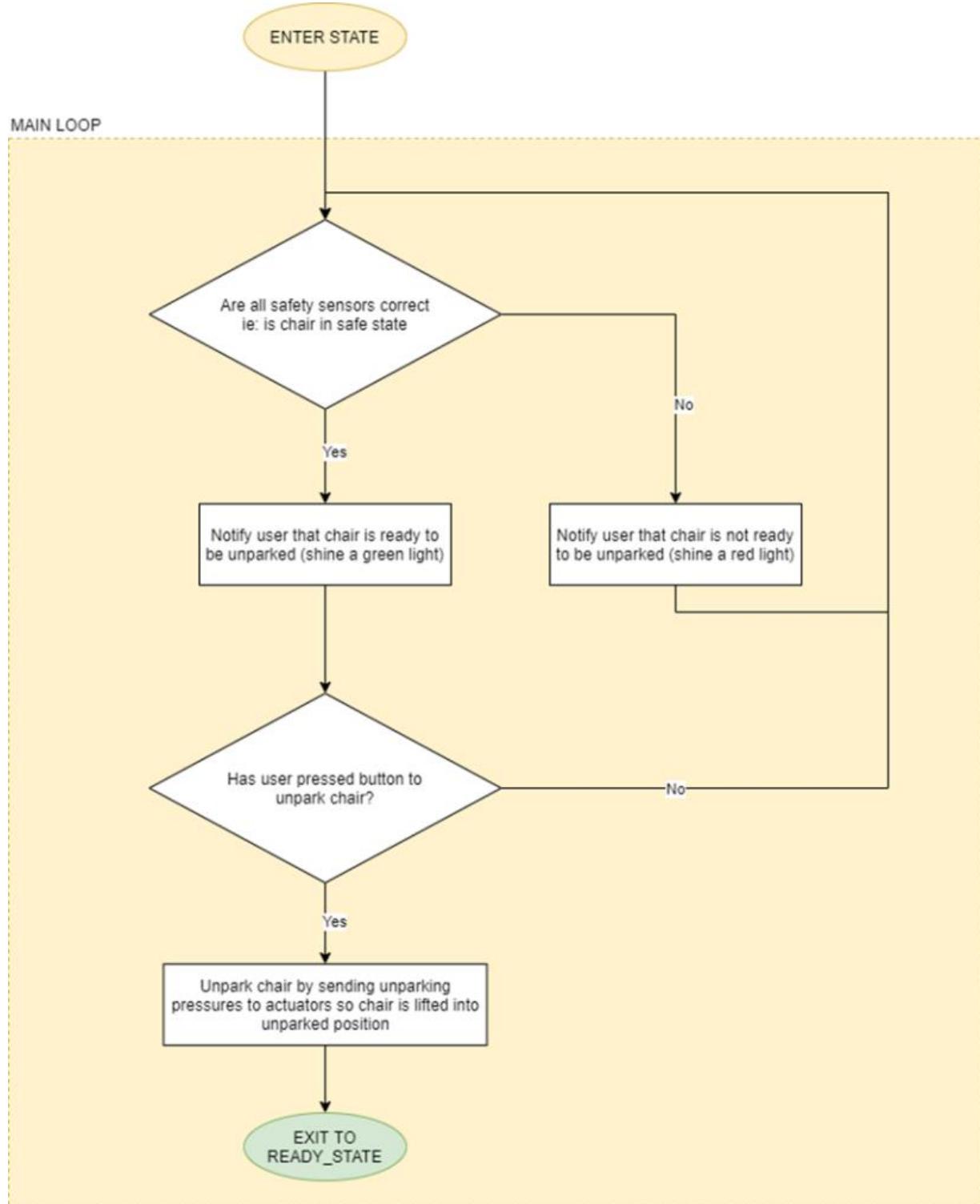
- [13] “Paper on evaluation of the dynamic model of fluidic muscles,” [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/1639161>.
- [14] “X-Plane 11 6DOF platform project (does not implement washout filters),” [Online]. Available: <https://www.youtube.com/watch?v=KMqd8JA2sos>.
- [15] “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://mememememememe.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

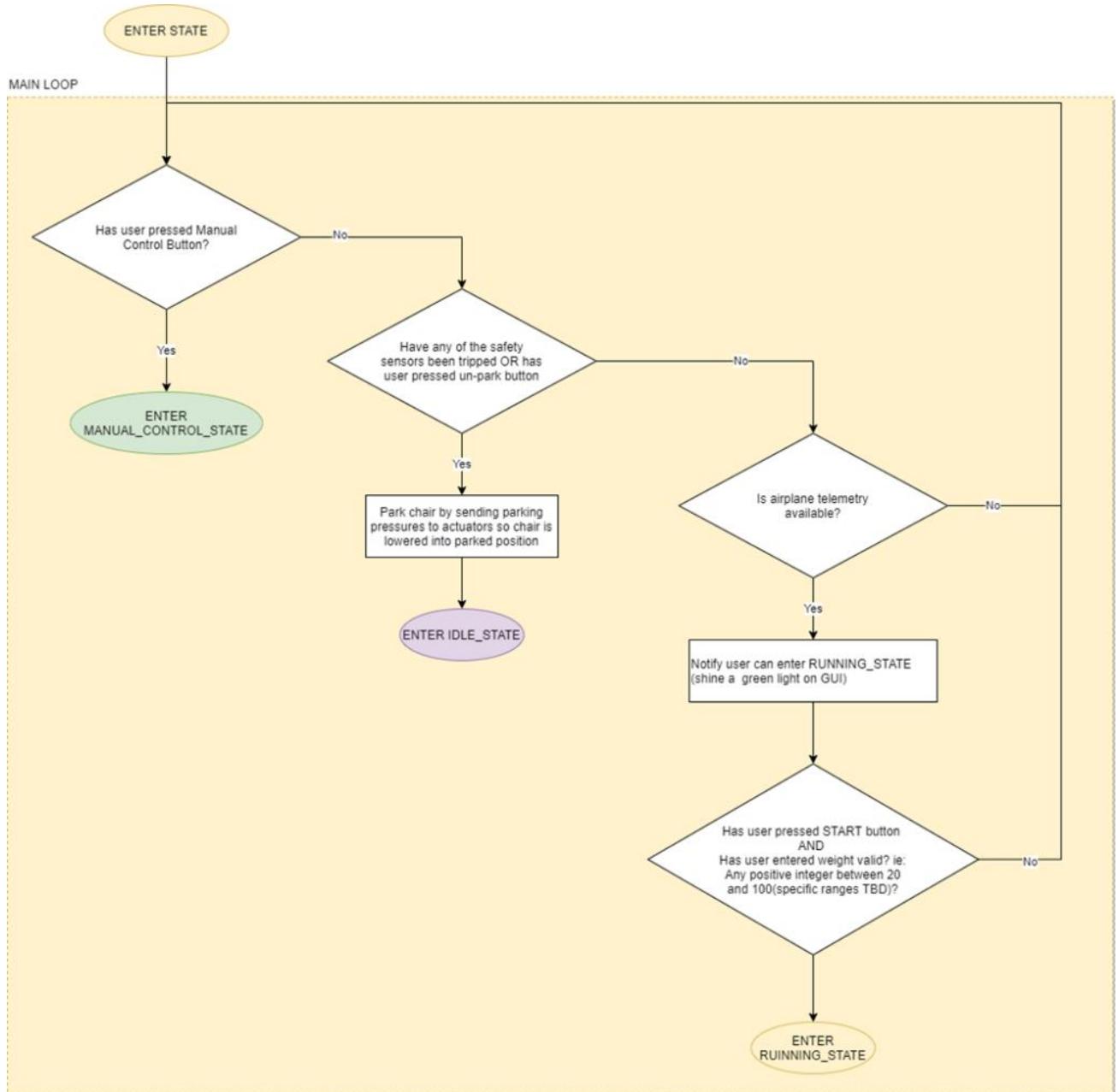
Appendix 1-RUNNING state logic.



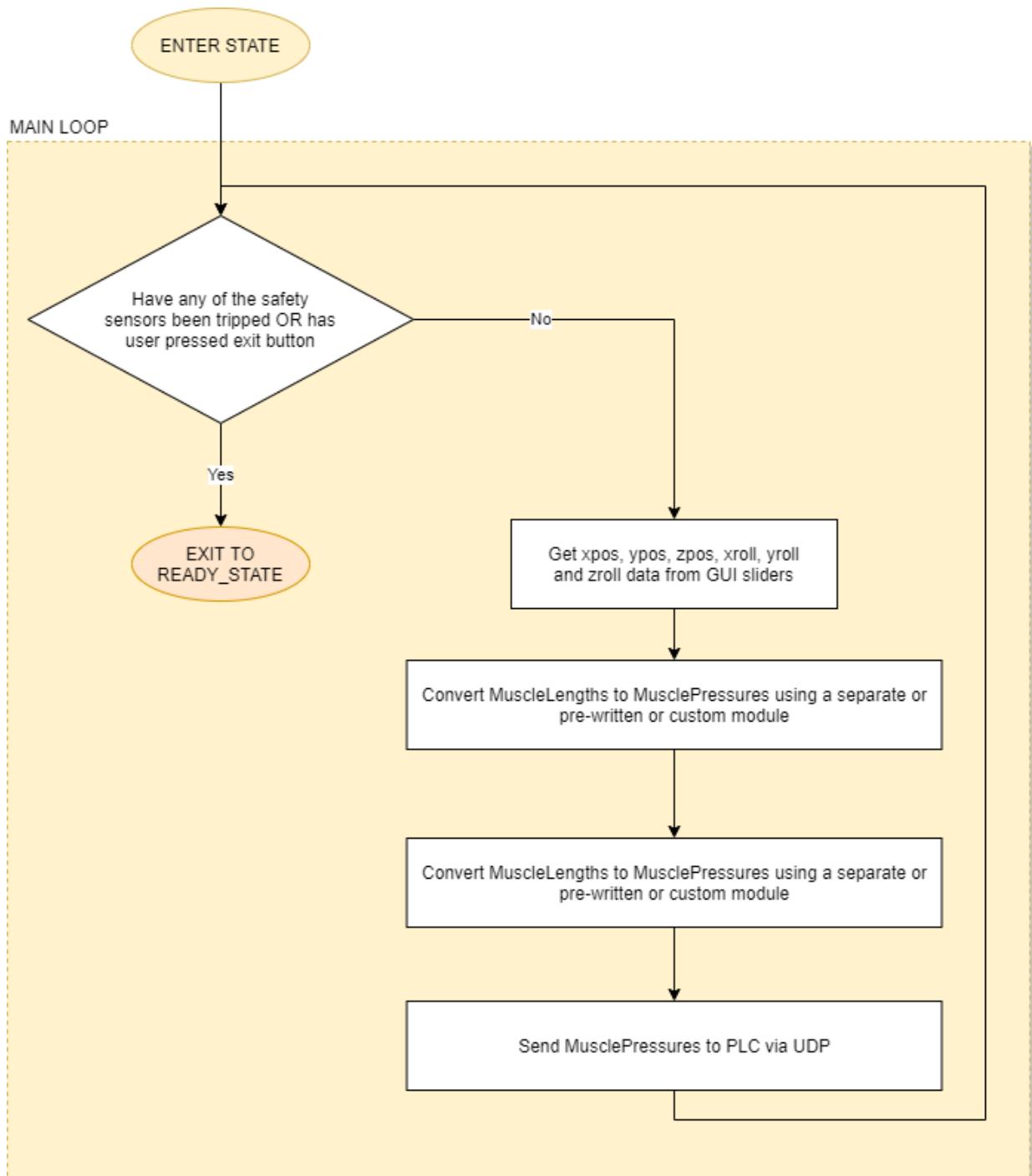
Appendix 2-IDLE state logic



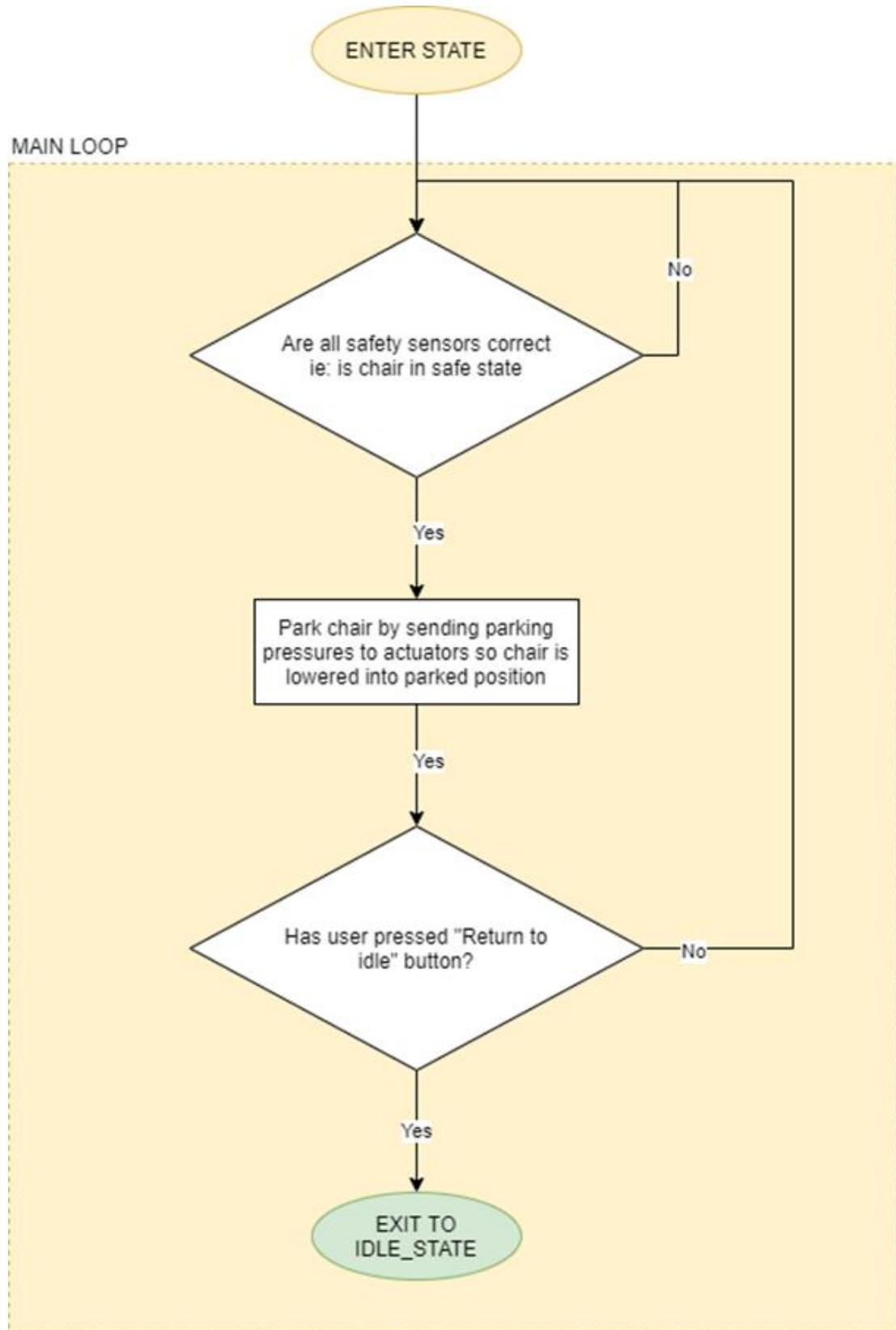
Appendix 3-READY state logic



Appendix 4-MANUAL state logic



Appendix 5-STOP state logic.



Appendix 6-state_machine.py code snippets

```

1 import keyboard
2 import time
3 import socket
4 import struct
5 import select
6 import numpy as np
7 from udp_tx_rx import UdpReceive, UdpSend
8 from platform_kinematics_module import PoseToDistances
9 from platform_pose import Platform
10
11 # This script handles the core logic of the software, managing the state machine
12 # responsible for receiving airplane telemetry (linear acceleration & rotational pose),
13 # computing inverse kinematics for the mechanical chair, and sending commands via UDP.
14
15 # Define IP address and port numbers for receiving and sending data
16 UDP_IP = "127.0.0.1" # Change if needed
17 SAFE_STATE_UDP_PORT = 4005
18 TELEMETRY_UDP_PORT = 10022
19 CMD_UDP_PORT = 6005
20 STATUS_UDP_PORT = 7005
21 POSE_DISPLAY_PORT = 10020
22 MANUAL_POSE_UDP_PORT = 9005
23
24 # Define update frequency (ensuring execution happens at 100Hz cycle time)
25 CYCLE_TIME = 1/100
26 |
27 # Initialize UDP receiver objects
28 safe_state_listener = UdpReceive(SAFE_STATE_UDP_PORT)
29 telemetry_listener = UdpReceive(TELEMETRY_UDP_PORT)
30 manual_telemetry_listener = UdpReceive(MANUAL_POSE_UDP_PORT)
31 command_listener = UdpReceive(CMD_UDP_PORT)
32
33 # Initialize UDP sender objects
34 display_sender = UdpSend()
35 status_sender = UdpSend()
36
37 # Global variables to retain values between cycles
38 global values
39 values = "0,0,0,0,0,0,0,0" # Default telemetry data
40
41 global user_command
42 user_command = "" # Command from user input
43
44 global user_gains
45 user_gains = [] # User-defined gain values
46
47 global safe_state
48 safe_state = False # Safe state of the system
49
50 # Base class for different states within the state machine
51 class State:
52     def __init__(self):
53         pass
54     def execute(self):
55         pass
56
57 # IDLE state: Waits until the chair is in a safe state
58 class IDLE(State):
59     def execute(self):
60         global safe_state
61         if safe_state:
62             return "ready"
63
64 # READY state: Waits for user command to transition to MANUAL or RUNNING state
65 class READY(State):
66     def execute(self):
67         global safe_state, user_command
68         if not safe_state:
69             return "idle"
70         elif user_command == "manual":
71             return "manual"
72         elif user_command == "running":
73             return "running"
74

```

```

75 # RUNNING state: Processes airplane telemetry, calculates inverse kinematics,
76 # and sends commands to the display. Transitions to STOP if chair is unsafe.
77 class RUNNING(State):
78     def __init__(self):
79         super().__init__()
80         self.pose_to_distances = PoseToDistances()
81         self.prev_values = values # Optimization to reduce redundant UDP messages
82         self.platform = Platform()
83
84     def execute(self):
85         global safe_state, user_command, values
86         if not safe_state or user_command == "stop":
87             return "stop"
88
89         # Only send messages if telemetry values have changed
90         elif values != self.prev_values:
91             xyzrpy = ",".join(values.split(",")[:3]) + "," + ",".join(values.split(",")[-3:])
92             xyzrpy_float = [float(num) for num in xyzrpy.split(",")]
93             new_values = "request" + "," + xyzrpy + "," + ",".join(map(str, self.pose_to_distances.move_platform(xyzrpy_float)))
94             print(new_values)
95             display_sender.send(new_values, (UDP_IP, POSE_DISPLAY_PORT))
96             self.prev_values = values
97             self.platform.set_pose(xyzrpy_float)
98
99     # STOP state: Waits for user command and safe state before transitioning back to IDLE
100 class STOP(State):
101     def execute(self):
102         global safe_state, user_command
103         if safe_state and user_command == "idle":
104             return "idle"
105
106     # MANUAL_CONTROL state: Allows manual control of the chair through UDP commands
107     class MANUAL_CONTROL(State):
108         def execute(self):
109             global user_command
110             if user_command == "ready":
111                 return "ready"
112
113     # State machine class: Manages state execution and transitions
114     class StateMachine:
115         def __init__(self):
116             self.state = None
117
118         def set_state(self, state: State):
119             self.state = state
120             print(f"State changed to {self.GetCurrentState()}")
121
122         def execute(self):
123             if self.state:
124                 transition = self.state.execute()
125                 if transition == "idle":
126                     self.set_state(IDLE())
127                 elif transition == "ready":
128                     self.set_state(READY())
129                 elif transition == "running":
130                     self.set_state(RUNNING())
131                 elif transition == "stop":
132                     self.set_state(STOP())
133                 elif transition == "manual":
134                     self.set_state(MANUAL_CONTROL())
135                 else:
136                     print("No state set")
137
138         def GetcurrentState(self):
139             return self.state.__class__.__name__
140

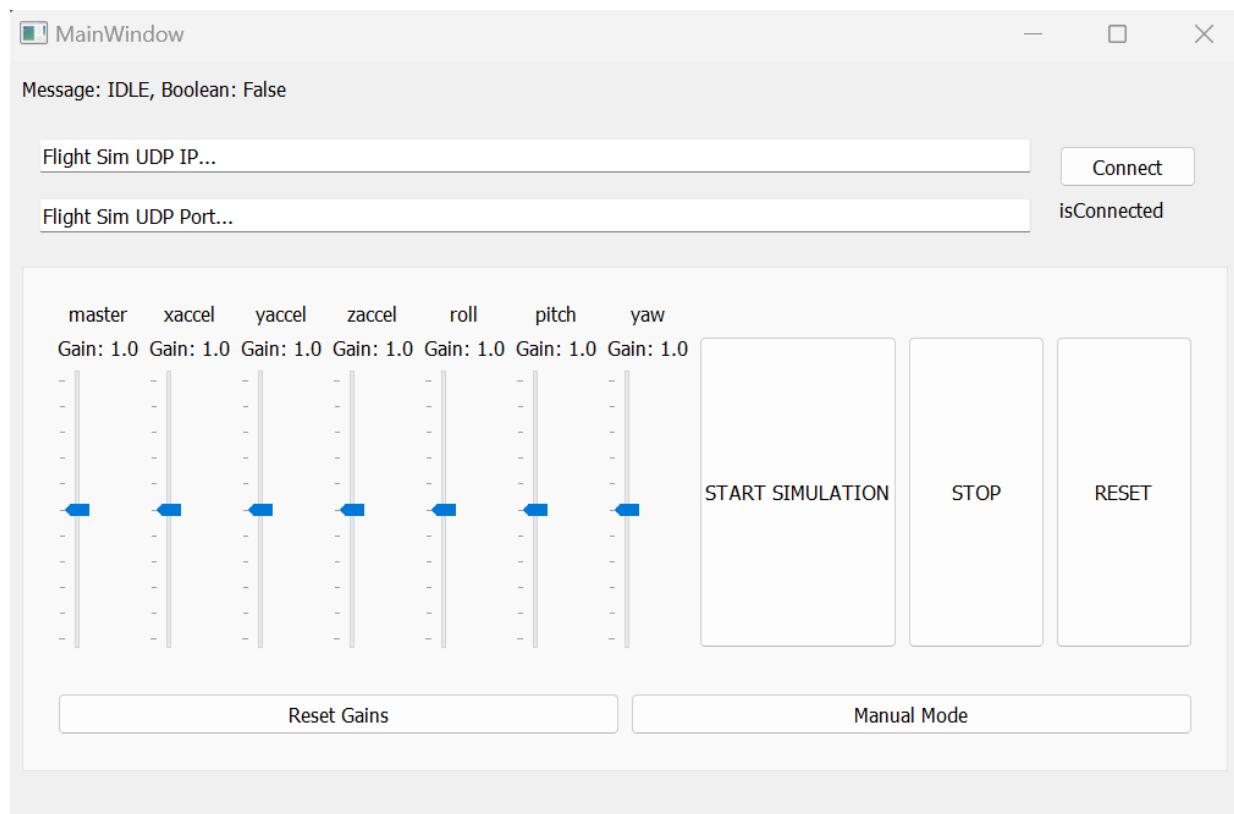
```

```

141 # Main loop: Runs state machine until 'q' is pressed
142 if __name__ == "__main__":
143     sm = StateMachine()
144     sm.set_state(IDLE())
145
146     target_time = time.time()
147
148     while not keyboard.is_pressed("q"):
149         current_time = time.time()
150
151         # Ensure state machine runs at defined CYCLE_TIME
152         if current_time >= target_time:
153
154             temp_values = None
155             while telemetry_listener.available() > 0:
156                 temp_values = telemetry_listener.get()
157
158             if temp_values:
159                 values = temp_values[1].split(",", 1)[1] # Remove "xplane_telemetry" prefix
160
161                 temp_safe_state = safe_state_listener.get()
162                 if temp_safe_state:
163                     safe_state = int(temp_safe_state[1])
164
165                 cmd_message = command_listener.get()
166                 if cmd_message:
167                     user_command, user_gains = cmd_message[1].split("|")
168                     user_gains = user_gains.split(",")
169                 else:
170                     user_command = None
171
172                 status = f"{sm.GetCurrentState()}|{safe_state}"
173                 status_sender.send(status, (UDP_IP, STATUS_UDP_PORT))
174
175                 sm.execute()
176                 target_time = current_time + CYCLE_TIME

```

Appendix 7-platform_control_app.py GUI snippets





Appendix 8-platform_control_app.py code snippets

```

1  from PyQt5 import QtWidgets
2  from gui import Ui_MainWindow # Import the generated UI class
3  from PyQt5.QtCore import QThread
4  import sys
5  import socket
6  import threading
7  import numpy as np
8  import struct
9  import time
10 from udp_tx_rx import UdpReceive, UdpSend
11
12 # Set up UDP ports for communication
13 UDP_IP = "127.0.0.1" # IP address for local communication
14 CMD_UDP_PORT = 6005 # Port for sending control commands
15 STATUS_UDP_PORT = 7005 # Port for receiving system status updates
16 MANUAL_POSE_UDP_PORT = 9005 # Port for sending manual control data
17
18 # Thread class to continuously listen for status updates from the state machine
19 class SM_Status_QThread(QThread):
20     def __init__(self, callback, udp_object):
21         super().__init__()
22         self.callback = callback # Store the callback function
23         self.isRunning = True # Control flag to manage thread execution
24         self.UdpListener = udp_object # UDP listener object
25
26     def run(self):
27         # Continuously check for new status messages while thread is running
28         while self.isRunning:
29             self.callback(self.UdpListener.get())
30
31     def stop(self):
32         """Stop the thread gracefully"""
33         self.isRunning = False
34         self.quit()
35         self.wait()
36

```

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

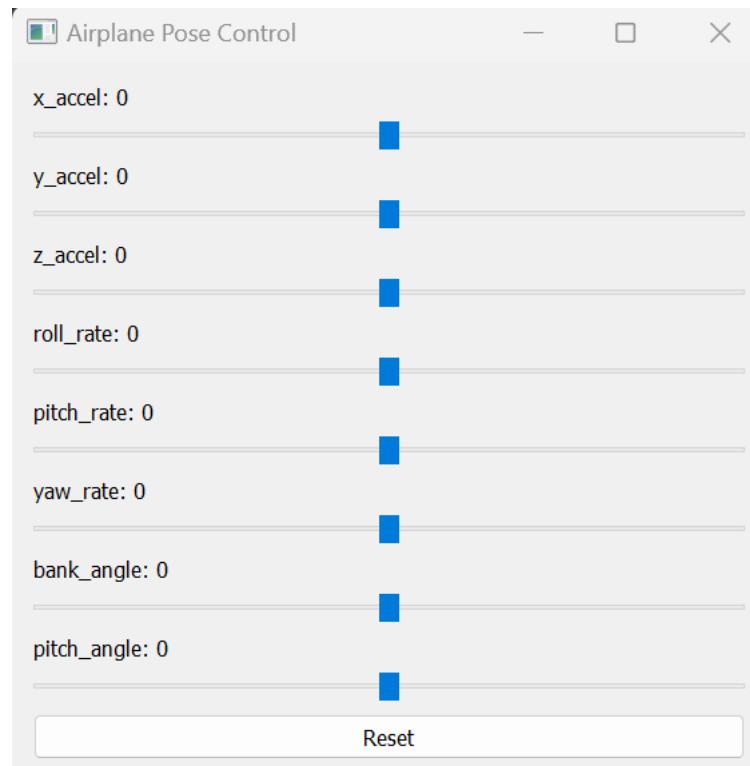
```

37 # Main application class that integrates the GUI and logic
38 class MainApp(QtWidgets.QMainWindow, Ui_MainWindow):
39     def __init__(self):
40         super().__init__()
41         self.setupUi(self) # Load the UI layout from the imported UI class
42
43         # Initialize UDP communication objects
44         self.cmd_sender = UdpSend()
45         self.manual_pose_sender = UdpSend()
46         self.status_receiver = UdpReceive(STATUS_UDP_PORT, blocking=False)
47
48         self.worker = None # Background thread for receiving system status
49
50         # Start background thread to continuously update GUI status
51         if self.worker is None or not self.worker.isRunning():
52             self.worker = SM_Status_QThread(self.update_status_label, self.status_receiver)
53             self.worker.start()
54
55         # Connect slider signals to labels for displaying gain values
56         self.master.valueChanged.connect(lambda value, lbl=self.master_label, mode="sim_gains": self.update_label(value, lbl, mode))
57         self.xaccel.valueChanged.connect(lambda value, lbl=self.xaccel_label, mode="sim_gains": self.update_label(value, lbl, mode))
58         self.yaccel.valueChanged.connect(lambda value, lbl=self.yaccel_label, mode="sim_gains": self.update_label(value, lbl, mode))
59         self.zaccel.valueChanged.connect(lambda value, lbl=self.zaccel_label, mode="sim_gains": self.update_label(value, lbl, mode))
60         self.roll.valueChanged.connect(lambda value, lbl=self.roll_label, mode="sim_gains": self.update_label(value, lbl, mode))
61         self.pitch.valueChanged.connect(lambda value, lbl=self.pitch_label, mode="sim_gains": self.update_label(value, lbl, mode))
62         self.yaw.valueChanged.connect(lambda value, lbl=self.yaw_label, mode="sim_gains": self.update_label(value, lbl, mode))
63
64         self.xaccel_manual.valueChanged.connect(lambda value, lbl=self.xaccel_label_2, mode="manual_gains": self.update_label(value, lbl, mode))
65         self.yaccel_manual.valueChanged.connect(lambda value, lbl=self.yaccel_label_2, mode="manual_gains": self.update_label(value, lbl, mode))
66         self.zaccel_manual.valueChanged.connect(lambda value, lbl=self.zaccel_label_2, mode="manual_gains": self.update_label(value, lbl, mode))
67         self.roll_manual.valueChanged.connect(lambda value, lbl=self.roll_label_2, mode="manual_gains": self.update_label(value, lbl, mode))
68         self.pitch_manual.valueChanged.connect(lambda value, lbl=self.pitch_label_2, mode="manual_gains": self.update_label(value, lbl, mode))
69         self.yaw_manual.valueChanged.connect(lambda value, lbl=self.yaw_label_2, mode="manual_gains": self.update_label(value, lbl, mode))
70
71         # Connect button clicks to sending control commands
72         self.start.clicked.connect(lambda value, command="running": self.send_control_command(command))
73         self.stop.clicked.connect(lambda value, command="stop": self.send_control_command(command))
74         self.reset.clicked.connect(lambda value, command="idle": self.send_control_command(command))
75         self.manual_mode_button.clicked.connect(lambda value, command="manual": self.send_control_command(command))
76         self.simulation_mode_button.clicked.connect(lambda value, command="ready": self.send_control_command(command))
77
78         # Reset sliders when reset buttons are clicked
79         self.reset_gains.clicked.connect(self.reset_sliders)
80         self.reset_gain_manual.clicked.connect(self.reset_sliders)
81
82         # Ensure the first tab (main control) is selected by default
83         self.tabWidget.setCurrentIndex(0)
84         self.tabWidget.tabBar().hide()
85
86         # Update label text with the current slider value based on the mode (simulation/manual)
87         def update_label(self, value, label, mode):
88             if mode == "sim_gains":
89                 label.setText(f"Gain: {value/5}")
90             elif mode == "manual_gains":
91                 label.setText(f"Gain: {(value*20)-100}")
92
93         # Send control command with the selected state and gain values
94         def send_control_command(self, state):
95             data_array = np.array([self.master.value(), self.xaccel.value(), self.yaccel.value(), self.zaccel.value(),
96                                  self.roll.value(), self.pitch.value(), self.yaw.value()])
97             data_array = data_array/5 # Normalize gain values
98
99             data_str = ",".join(map(str, data_array)) # Convert array to comma-separated string
100            message = f"{{state}}{data_str}" # Format message
101            print(message)
102            self.cmd_sender.send(message, (UDP_IP, CMD_UDP_PORT)) # Send command via UDP
103
104         # Update the GUI status label with the latest state information
105         def update_status_label(self, data):
106             if data is None:
107                 return
108
109             message, flag = data[1].split("|")
110             self.SM_STATUS.setText(f"Message: {message}, Boolean: {flag}")
111
112             # Update start button text based on safety status
113             if flag:
114                 self.start.setText("START SIMULATION")
115             else:
116                 self.start.setText("CHECK SAFETY SENSORS")
117
118             # Automatically switch to manual control tab if in MANUAL_CONTROL state
119             if message == "MANUAL_CONTROL":
120                 self.tabWidget.setCurrentIndex(1)
121             else:
122                 self.tabWidget.setCurrentIndex(0)

```

```
124     # Handle window close event to properly stop background threads
125     def closeEvent(self, event):
126         pass # Placeholder for cleanup logic if needed
127
128     # Reset all sliders to their default middle position
129     def reset_sliders(self):
130         sliders = [self.master, self.xaccel, self.yaccel, self.zaccel, self.roll, self.pitch, self.yaw,
131                    self.xaccel_manual, self.yaccel_manual, self.zaccel_manual, self.roll_manual, self.pitch_manual, self.yaw_manual]
132         for slider in sliders:
133             slider.setValue(5)
134
135     # Initialize and run the PyQt application
136     if __name__ == "__main__":
137         app = QtWidgets.QApplication(sys.argv)
138         window = MainApp()
139         window.show()
140         sys.exit(app.exec_())
```

Appendix 9-dummy_airplane_app.py GUI snippets



Appendix 10-dummy_airplane_app.py code snippets

```

1 import sys
2 import socket
3 import struct
4 from PyQt5.QtWidgets import QApplication, QWidget, QVBoxLayout, QSlider, QLabel, QCheckBox, QPushButton
5 from PyQt5.QtCore import Qt, QTimer
6 from udp_tx_rx import UdpSend, UdpReceive
7 import math
8
9 #This is a dummy airplane app where each part of the plane's pose is controlled using a dedicated slider
10 UDP_IP = "127.0.0.1"
11
12 #Define telemetry port for which to send out the messages containing status (any of the safety sensors tripper) and airplane pose
13 TELEMETRY_UDP_PORT = 5005
14
15 #NOTE: Again, its worth noting that in this project acceleration and translation are used somewhat arbitrarily as the airplane's linear accel
16 #dictates the chair's translatation to create an ilusion of movement.
17
18 #Define main window widget containing 6 sliders for the airplane's 6D pose (xaccel, yaccel, zaccel, roll, pitch, yaw)
19 class AirplaneControl(QWidget):
20     def __init__(self):
21         super().__init__()
22
23         #Set window title and size
24         self.setWindowTitle("Airplane Pose Control")
25         self.setGeometry(100, 100, 300, 400)
26
27         #Define layout to be vertical so that added components are on top of each other
28         self.layout = QVBoxLayout()
29
30         #Iteratively add label and slider widgets to the main window layout
31         self.sliders = {}
32         self.labels = {}
33         self.pose_keys = ["x_accel", "y_accel", "z_accel", "roll_rate", "pitch_rate", "yaw_rate", "bank_angle", "pitch_angle"]
34         self.pose_values = {key: 0 for key in self.pose_keys}
35
36
37         for key in self.pose_keys:
38             label = QLabel(f"{key}: 0")
39             slider = QSlider(Qt.Horizontal)
40             slider.setMinimum(-180)
41             slider.setMaximum(180)
42             slider.setValue(0)
43             slider.setSingleStep(1) # Increment/decrement by 1 on arrow click
44             slider.valueChanged.connect(lambda value, k=key: self.update_pose(k, value))
45
46             self.labels[key] = label
47             self.sliders[key] = slider
48
49             self.layout.addWidget(label)
50             self.layout.addWidget(slider)
51
52         #Add reset button widget to main window layout which resets all the slider values to their default value
53         self.reset_button = QPushButton("Reset")
54         self.reset_button.clicked.connect(self.reset_sliders)
55         self.layout.addWidget(self.reset_button)
56         self.setLayout(self.layout)
57
58         #UDP sender object
59         self.udp_sender = UdpSend()
60
61         #Define timer which sends out UDP messages every 100ms
62         self.timer = QTimer()
63         self.timer.timeout.connect(self.send_udp_data)
64         self.timer.start(100) # Send data every 100 ms
65
66         #Method which updates each element of the self.pose list to each respective slider value. This also sets the label text to the appropriate val
67     def update_pose(self, key, value):
68         self.pose_values[key] = value
69         self.labels[key].setText(f"[key]: {value}")

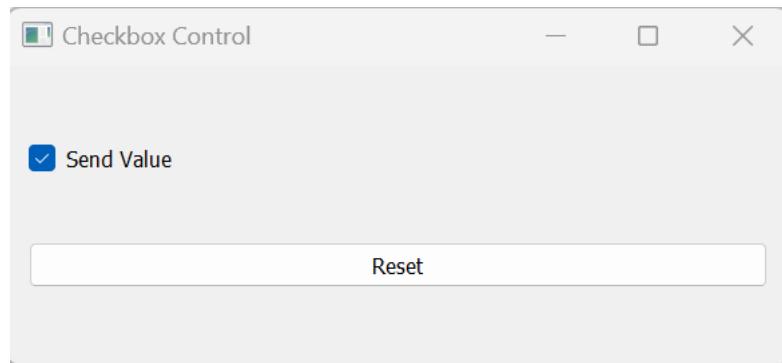
```

```

71     #This method resets all the sliders to 0
72     def reset_sliders(self):
73         """Resets all sliders to 0."""
74         for key in self.pose_keys:
75             self.sliders[key].setValue(0) # This will trigger valueChanged and update labels
76
77     #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
78     #states whether the chair or airplane iss in it's safe state
79     def send_udp_data(self):
80         data = list(self.pose_values.values()) # Convert dict_values to a list
81         data[-2:] = [math.radians(value) for value in data[-3:]] # Convert last two items to radians
82         data_str = ('xplane_telemetry, ' + ",".join(map(str, data)))
83         self.udp_sender.send(data_str, (UDP_IP, TELEMETRY_UDP_PORT))
84
85     #Start up the window once script is called
86     if __name__ == "__main__":
87         app = QApplication(sys.argv)
88         window = AirplaneControl()
89         window.show()
90         sys.exit(app.exec_())
91

```

Appendix 11-safe_state_dummy.py GUI snippet



Appendix 12-safe_state_dummy.py code snippets

```

1  import sys
2  import socket
3  import struct
4  from PyQt5.QtWidgets import QApplication, QWidget, QVBoxLayout, QCheckBox, QPushButton
5  from PyQt5.QtCore import QTimer
6  from udp_tx_rx import UdpSend
7
8  # Define UDP IP and port for sending telemetry data
9  UDP_IP = "127.0.0.1"
10 TELEMETRY_UDP_PORT = 4005
11
12 class CheckboxControl(QWidget):
13     def __init__(self):
14         super().__init__()
15
16         # Set up the window properties
17         self.setWindowTitle("Checkbox Control")
18         self.setGeometry(100, 100, 300, 200)
19
20         # Create a vertical layout
21         self.layout = QVBoxLayout()
22
23         # Define checkbox for sending data
24         self.checkbox = QCheckBox("Send Value")
25         self.checkbox.stateChanged.connect(self.send_udp_data)
26         self.layout.addWidget(self.checkbox)
27
28         # Define reset button to uncheck the checkbox
29         self.reset_button = QPushButton("Reset")
30         self.reset_button.clicked.connect(self.reset_checkbox)
31         self.layout.addWidget(self.reset_button)
32
33         # Set layout for the window
34         self.setLayout(self.layout)
35
36         # Initialize UDP sender object
37         self.udp_sender = UdpSend()
38
39         # Define a timer to send UDP messages every 100ms
40         self.timer = QTimer()

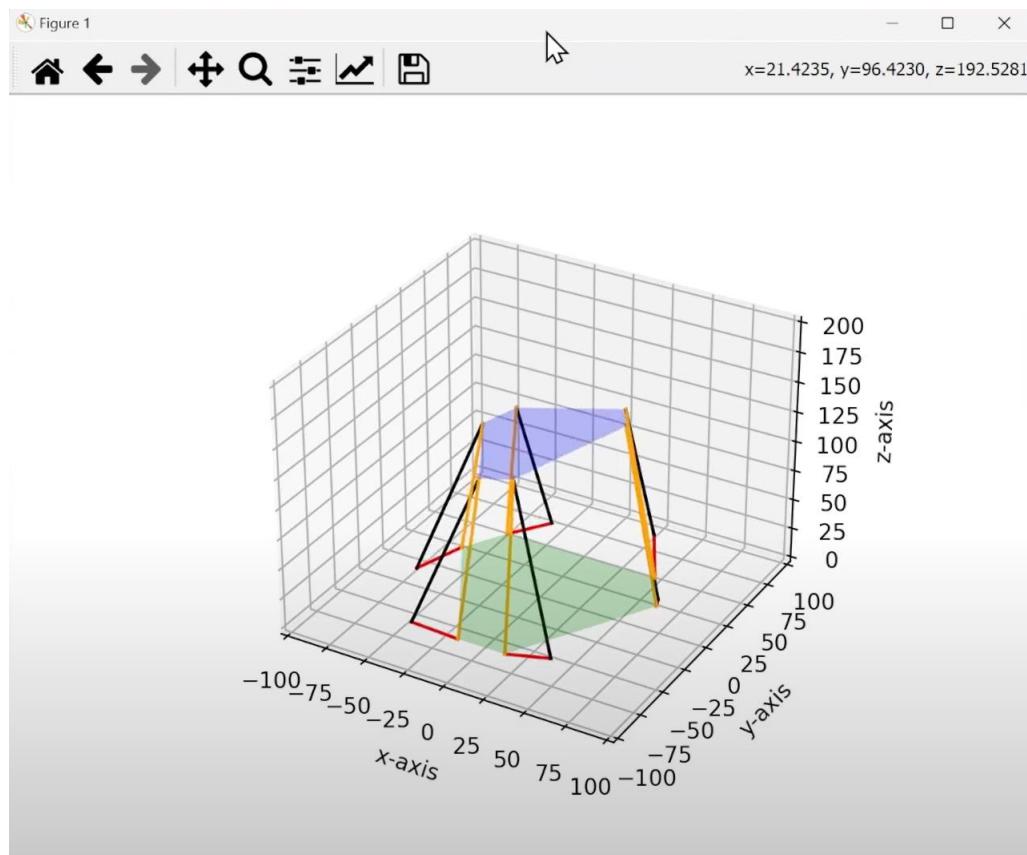
```

```

41     self.timer.timeout.connect(self.send_udp_data)
42     self.timer.start(100) # Send data every 100 ms
43
44     def reset_checkbox(self):
45         """Resets checkbox to unchecked state."""
46         self.checkbox.setChecked(False)
47
48     def send_udp_data(self):
49         """Sends UDP data based on checkbox state."""
50         data = int(self.checkbox.isChecked()) # Convert checkbox state to integer (0 or 1)
51         data_str = str(data) # Convert to string format for sending
52         self.udp_sender.send(data_str, (UDP_IP, TELEMETRY_UDP_PORT)) # Send data via UDP
53         #print("Sent data:", data_str) # Uncomment for debugging
54
55     # Run the application if this script is executed directly
56 if __name__ == "__main__":
57     app = QApplication(sys.argv)
58     window = CheckboxControl()
59     window.show()
60     sys.exit(app.exec_())

```

Appendix 13-stewart_displa.py GUI snippet



Appendix 14-stewart_display.py code snippets

```

1 import socket
2 import struct
3 import select
4 from src.stewart_controller import Stewart_Platform
5 import matplotlib.pyplot as plt
6 import numpy as np
7 import time
8 import matplotlib.animation as animation
9 from udp_tx_rx import UdpReceive
10
11 # Define UDP port to listen for pose data, which will be sent from the state_machine.py node
12 UDP_IP = "127.0.0.1"
13 TELEMETRY_UDP_PORT = 8005
14
15 # Set up UDP socket to listen for pose data (Non-blocking)
16 telemetry_listener = UdpReceive(TELEMETRY_UDP_PORT)
17
18 # Initialize Stewart Platform parameters (Hardcoded for now, will be configurable later)
19 # This setup is temporary and serves as a debug visualization of platform movement.
20 platform = Stewart_Platform(132/2, 100/2, 30, 130, 0.2269, 0.82, 5*np.pi/6)
21
22 # Function to listen for incoming airplane telemetry and update pose data.
23 # Inverse kinematics is handled using the "Stewart_Py" repository.
24 # Eventually, inverse kinematics calculations will be moved to state_machine.py,
25 # and the chair will only receive muscle pressure values.
26 def listen_for_telemetry():
27     """Reads the latest UDP message, discarding old ones."""
28     data = None
29
30     # Retrieve the most recent telemetry data from the UDP buffer
31     while telemetry_listener.available() > 0:
32         data = telemetry_listener.get()
33
34     if data is None:
35         return [], []
36
37     # Parse incoming pose data
38     pose_data = data[1].split(",")
39     pose_data = [int(float(r)) for r in pose_data] # Convert string values to integers
40     rotation = pose_data[6:9] # Extract rotation values from the data
41

```

```

42     # Flip roll and pitch for debugging purposes
43     rotation[0], rotation[1] = -rotation[1], rotation[0]
44
45     trans, rot = [0, 0, 0], rotation # Translation is set to zero (for now)
46     return trans, rot
47
48     # Initialize figure and axis for real-time 3D plotting
49     fig = plt.figure()
50     ax = fig.add_subplot(111, projection='3d')
51
52     # Function to update the Stewart platform's position and orientation based on telemetry data
53     def update(frame):
54         """Update function for real-time plotting."""
55         trans, rot = listen_for_telemetry()
56         if trans and rot: # Ensure valid data exists
57             servo_angles = platform.calculate(np.array(trans), np.array(rot) * (np.pi / 180)) # Convert degrees to radians
58             platform.plot_platform(ax=ax) # Update the 3D plot
59
60     # Start real-time animation (100ms update rate)
61     ani = animation.FuncAnimation(fig, update, interval=100)
62
63     # Display the 3D plot
64     plt.show()

```

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

```

1 """
2 Mdx_Telemetry
3 Written by Michael Margolis
4
5 Sends 6DoF Xplane telemetry over UDP.
6
7 # xplane uses OpenGL coordinates - x is right, y up, z back
8
9 Mdx coordinate frame follows ROS conventions, positive values: X is forward, Y is left, Z is up,
10 roll is right side down, pitch is nose down, yaw is CCW; all from perspective of person on platform.
11
12 datarefs:
13     xplane translation acc are in g
14         axil = surge = x (invert)
15         side = sway = y (invert)
16         nrml = heave = z
17     xplane roll and pitch are in degrees
18     rotation velocities are rad/sec
19         roll (deg to rad, invert)
20         pitch (deg to rad invert)
21         yaw (invert)
22
23
24     // Translation
25     double acc_axial;           // [G] backward +
26     double acc_lateral;         // [G] right +
27     double acc_normal;          // [G] up +, +1.0 normal g
28
29     // Rotation
30     double vel_roll;           // [rad/sec] clockwise + (Rechtskurve)
31     double vel_pitch;          // [rad/sec] nose up +
32     double vel_yaw;            // [rad/sec] clockwise + (Rechtskurve)
33     double roll;               // [deg] clockwise +
34     double pitch;              // [deg] nose up +
35
36
37 msg format:
38     surge accel, sway accel, heave accel, roll vel, pitch vel, yaw vel, roll, pitch
39 """
40
41
42 #"/D:\Program Files\X-Plane 12\X-Plane.exe" --load_rep="D:\MdxCessnaFlight.rep"
43

```

```

44 from XPPython3 import xp
45 from collections import namedtuple
46 from math import radians, pi, degrees, sqrt
47 from udp_tx_rx import UdpReceive
48 import XPLMDDataAccess
49
50 transform_refs = namedtuple('transform_refs', \
51     ('DR_g_axil', 'DR_g_side', 'DR_g_nrml', \
52     'DR_Prad', 'DR_Qrad', 'DR_Rrad', \
53     'DR_theta', 'DR_psi', 'DR_phi', \
54     'DR_groundspeed'))
55
56
57 UDP_IP = "127.0.0.1"
58 TARGET_PORT = 10022 # port of sim interface controller telemetry socket
59
60 xplm_key_pause = 0
61
62
63 class PythonInterface:
64     def XPluginStart(self):
65         self.Name = "PlatformItf v1.01"
66         self.Sig = "Mdx.Python.UdpTelemetry"
67         self.Desc = "A plug-in for the Mdx platform that sends telemetry data over UDP."
68         LISTEN_PORT = 10023
69         self.controller_addr = []
70         self.udp = UdpReceive(LISTEN_PORT)
71
72         self.init_drefs()
73
74         # Create our menu
75         Item = xp.appendMenuItem(xp.findPluginsMenu(), "Flight transforms", 0)
76         self.InputOutputMenuHandlerCB = self.InputOutputMenuHandler
77         self.Id = xp.createMenu("Platform Interface", xp.findPluginsMenu(), Item, self.InputOutputMenuHandlerCB, 0)
78         xp.appendMenuItem(self.Id, "View Transforms", 1)
79
80         self.IsWidgetVisible = 0 # Flag indicating if widget is being displayed.
81
82         self.OutputDataRef = []
83         for Item in range(len(self.xform_drefs)):
84             self.OutputDataRef.append(xp.findDataRef(self.xform_drefs[Item]))

```

```

85     # Register our FL callback with initial callback freq of 1 second
86     self.InputOutputLoopCB = self.InputOutputLoopCallback
87     xp.registerFlightLoopCallback(self.InputOutputLoopCB, 1.0, 0)
88
89     return self.Name, self.Sig, self.Desc
90
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_nrm1')
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)', 'Yaw rate (rad/s)', 'Roll angle (rad)', 'Pitch angle (rad)' )
134
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]

```

```

165     #print(msg,cmd)
166     if cmd == 'InitComs':
167         self.controller_addr.append(addr[0])
168         print("Added controller IP address " + addr[0])
169     elif cmd == 'Run':
170         xp.commandOnce(self.replay_play)
171     elif cmd == 'PauseToggle':
172         xp.commandOnce(self.pauseCmd)
173     elif cmd == 'Pause':
174         is_paused = xp.getDataf(self.pauseStateDR)
175         print('is_paused:{} {}'.format(is_paused, is_paused == 1))
176         if not is_paused:
177             xp.commandOnce(self.pauseCmd)
178     elif cmd == 'Reset':
179         xp.commandOnce(self.go_to_replay_begin)
180     elif cmd == 'Replay':
181         filepath = msg[1]
182         # XPLMDataAccess.XPLMLoadDataFile(XPLMDataAccess.xplm_DataFile_Replay, filepath)
183         ret = xp.loadDataFile(xp.DataFile_ReplayMovie, filepath)
184         print(ret, msg, filepath)
185         # XPLMDataAccess.XPLMSetDatai(self.replay_mode, 1)
186     elif cmd == 'Situation':
187         filepath = msg[1]
188         # XPLMDataAccess.XPLMLoadDataFile(XPLMDataAccess.xplm_DataFile_Replay, filepath)
189         ret = xp.loadDataFile(xp.DataFile_Situation, filepath)
190         print(ret, msg, filepath)
191         # XPLMDataAccess.XPLMSetDatai(self.replay_mode, 1)
192     else:
193         print(msg)
194
195     if self.IsWidgetVisible != 0: # Don't update dialog if widget not visible
196         for Item in range(len(telemetry_str)):
197             xp.setWidgetDescriptor(self.OutputEdit[Item], telemetry_str[Item])
198
199     return 0.025 # callback every 25ms.
200
201 def InputOutputMenuHandler(self, inMenuRef, inItemRef):
202     # If menu selected create our widget dialog
203     if inItemRef == 1:
204         if self.IsWidgetVisible == 0:
205             self.CreateInputOutputWidget(300, 550, 350, 350)
206             self.IsWidgetVisible = 1
207         else:
208             if not xp.isWidgetVisible(self.InputOutputWidget):
209                 xp.showWidget(self.InputOutputWidget)
210
211     def CreateInputOutputWidget(self, x, y, w, h):
212         x2 = x + w
213         y2 = y - h
214
215         # Create the Main Widget window
216         self.InputOutputWidget = xp.createWidget(x, y, x2, y2, 1, "Python - Mdx Platform Interface",
217                                                 1, 0, xp.WidgetClass_MainWindow)
218
219         # Add Close Box decorations to the Main Widget
220         xp.setWidgetProperty(self.InputOutputWidget, xp.Property_MainWindowHasCloseBoxes, 1)
221
222         # Create the Sub Widget window
223         InputOutputWindow = xp.createWidget(x + 50, y - 50, x2 - 50, y2 + 50, 1, "",
224                                             0, self.InputOutputWidget, xp.WidgetClass_SubWindow)
225
226         # Set the style to sub window
227         xp.setWidgetProperty(InputOutputWindow, xp.Property_SubWindowType, xp.SubWindowStyle_SubWindow)
228
229         CaptionText = []
230         self.InputEdit = []
231         self.OutputEdit = []
232         for Item in range(len(self.XformDesc)):
233             # Create a text widget
234
235             CaptionText.append(xp.createWidget(x + 60, y - (60 + (Item * 30)), x + 90, y - (82 + (Item * 30)), 1,
236                                              self.XformDesc[Item], 0, self.InputOutputWidget, xp.WidgetClass_Caption))
237             self.OutputEdit.append(xp.createWidget(x + 190, y - (60 + (Item * 30)), x + 270, y - (82 + (Item * 30)), 1,
238                                              "?", 0, self.InputOutputWidget, xp.WidgetClass_TextField))
239
240         # Register our widget handler
241         self.InputOutputHandlerCB = self.InputOutputHandler
242         xp.addWidgetCallback(self.InputOutputWidget, self.InputOutputHandlerCB)

```

```

244     def InputOutputHandler(self, inMessage, inWidget, inParam1, inParam2):
245         if inMessage == xp.Message_CloseButtonPushed:
246             if self.IsWidgetVisible == 1:
247                 xp.hideWidget(self.InputOutputWidget)
248             return 1
249         return 0
250
251     def read_telemetry(self):
252         try:
253             datarefs = []
254             for Item in range(self.NumberOfDatarefs):
255                 datarefs.append( xp.getDataRef(self.OutputDataRef[Item]) )
256
257             raw_data = tuple(datarefs)
258             named_data = transform_refs._make(raw_data) # load namedtuple with values
259
260             # see https://developer.x-plane.com/code-sample/motionplatformdata/
261             telemetry = []
262             telemetry.append(named_data.DR_g_axil * -1) # surge_accel
263             telemetry.append(named_data.DR_g_side * -1) # sway_accel
264             telemetry.append(named_data.DR_g_nrm1) # heave_accel
265             telemetry.append(named_data.DR_Prad * -1) # roll_rate
266             telemetry.append(named_data.DR_Qrad * -1) # pitch_rate
267             #telemetry.append(named_data.DR_Rrad * -1) # yaw_rate
268             #telemetry.append(radians(named_data.DR_phi)) # bank angle
269             #telemetry.append(radians(named_data.DR_theta)) * -1) # pitch angle
270
271             telemetry.append(named_data.DR_phi) # bank angle
272             telemetry.append(named_data.DR_theta * -1) # pitch angle
273             telemetry.append(named_data.DR_Rrad * -1) # yaw_rate
274
275             return telemetry
276
277         except Exception as e:
278             xp.log(str(e) + " reading datarefs")
279             return [0,0,0,0,0,0,0,0]
280

```

Appendix 16-Telemetry code of Unity display app

```

1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using UnityEngine.UI;
5  using System;
6  using System.Text;
7  using System.Net;
8  using System.Net.Sockets;
9  using System.Threading;
10 using System.IO;
11
12 // Unity Script | 0 references
13 public class Telemetry : MonoBehaviour
14 {
15     private GameObject chair;
16     private GameObject prop;
17     //public Text transX, transY, transZ;
18     //public Text rotX, rotY, rotZ;
19     public Text telemetryLabel;
20
21     private Slider adjTransX, adjTransY, adjTransZ;
22     private Slider adjRotX, adjRotY, adjRotZ;
23     private Text[] xyzrpyValues;
24     private Text lblTelemetryIncoming, lblTelemetryOutgoing;
25     private Vector3 center;
26     private Vector3 packPos = new Vector3(0f, -15f, 0f);
27     private float scale = 180; // number of mm in each unity unit of measure
28     private float[] defaultLimits = new float[] { 100, 125, 140, 15, 15, 12 }; // mm and degrees for xyzrpy
29     private float[] limits = new float[6];
30
31     // Define IP Address and Port
32     static public string IP = "127.0.0.1";
33     static public int TxPort = 10009; // input messages to mdx platform
34     static public int RxFromPort = 10020; // output messages from mdx platform
35     IPEndPoint rxEndpoint; // for incoming render messages
36     IPEndPoint txEndpoint; // for outgoing orientation command messages
37
38     UdpClient client;
39     //Thread listener;
40
41     private bool isCommandMode = false; // this scene provides input to platform controller software
42     private bool isRenderMode = false; // this scene will display output from controller software
43     private bool isShuttingDown = false;
44
45     // Use this for initialization
46     // Unity Message | 0 references
47     void Start()
48     {
49         Console.WriteLine("Starting");
50         chair = GameObject.Find("ChairAndBase");
51         //prop = GameObject.Find("PropExtended");
52         //prop.SetActive(false);
53     }

```

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

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

```

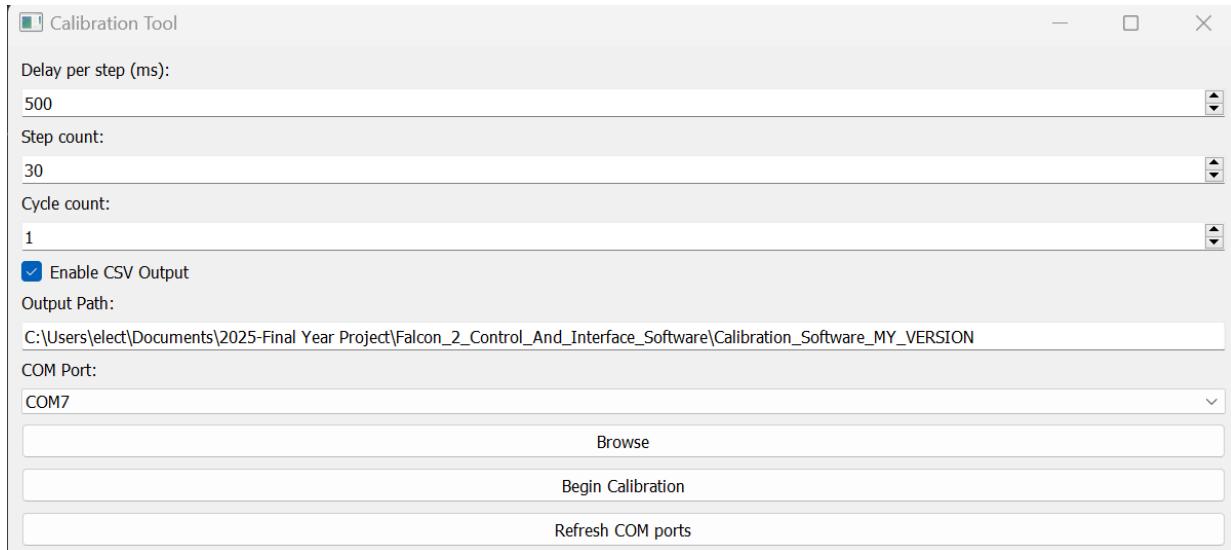
151     Vector3 localPos;
152     localPos.x = limits[1] * float.Parse(fields[2]) / scale;
153     localPos.y = limits[2] * float.Parse(fields[3]) / scale;
154     localPos.z = limits[0] * float.Parse(fields[1]) / scale;
155     Vector3 localRot;
156     localRot.x = limits[4] * float.Parse(fields[5]);
157     localRot.y = limits[5] * float.Parse(fields[6]);
158     localRot.z = limits[3] * float.Parse(fields[4]);
159     lblTelemetryIncoming.text = msg;
160     for (int i = 0; i < 6; i++)
161     {
162         xyzrpyValues[i].text = fields[i + 1];
163     }
164     UpdateOrientation(localPos, localRot);
165 }
166 catch (Exception e)
167 {
168     print(e);
169 }
170 }
171 else if (fields[0] == "request")
172 {
173     try
174     {
175         print(fields);
176         if (fields.Length > 6)
177         {
178             Vector3 localPos;
179             localPos.x = float.Parse(fields[2]) / scale;
180             localPos.y = float.Parse(fields[1]) / scale;
181             localPos.z = float.Parse(fields[0]) / scale;
182             Vector3 localRot;
183             localRot.x = float.Parse(fields[5]) * (float)(180.0 / Math.PI);
184             localRot.y = float.Parse(fields[4]) * (float)(180.0 / Math.PI);
185             localRot.z = float.Parse(fields[3]) * (float)(180.0 / Math.PI);
186             Debug.Log("Rotation: " + localRot.ToString());
187             lblTelemetryIncoming.text = localPos.ToString();
188             //if(fields.Length > 7)
189             //{
190             //    prop.SetActive(int.Parse(fields[7]) == 1);
191             //}
192
193             if (!isCommandMode && xyzrpyValues != null)
194             {
195                 // only display 6DOF values if sliders not active
196                 xyzrpyValues[0].text = localPos.z.ToString();
197                 xyzrpyValues[1].text = localPos.x.ToString();
198                 xyzrpyValues[2].text = localPos.y.ToString();
199
200                 xyzrpyValues[3].text = localRot.z.ToString();
201                 xyzrpyValues[4].text = localRot.x.ToString();
202                 xyzrpyValues[5].text = localRot.y.ToString();
203             }
204             lblTelemetryIncoming.text = msg;
205             UpdateOrientation(localPos, localRot);
206         }
207     }
208     catch (Exception e)
209     {
210         print(e);
211     }
212 }
213 }
214 }
215 }
216 }
217 }
218 }
219 public void SendMessage(byte[] message, IPEndPoint target)
220 {
221     if (isShuttingDown) return;
222     try
223     {
224         if (client != null)
225         {
226             client.Send(message, message.Length, target);
227         }
228     }
229     catch (Exception err)
230     {
231         Console.WriteLine("UDP send error:" + err.ToString());
232     }
233 }
234
235 // Unity Message|0 references
236 void OnApplicationQuit()
237 {
238     isShuttingDown = true;
239 }
240
241 /*
242  * functions for slider input control
243 */
244
245 // Unity Message|0 references
246 public void setIsActive(bool isActive)
247 {
248     if (isShuttingDown) return;
249     print(String.Format("isActive set to {0}", isActive));
250     byte[] message;

```

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

```
250
251     if (isActive)
252         message = Encoding.ASCII.GetBytes("<command,enable,\n");
253     else
254         message = Encoding.ASCII.GetBytes("<command,disable,\n");
255
256     if (client != null)
257     {
258         client.Send(message, message.Length, txEndpoint);
259     }
260 }
261
262 0 references
263 public void CenterPlatform()
264 {
265     adjTransX.value = adjTransY.value = adjTransZ.value = 0;
266     adjRotX.value = adjRotY.value = adjRotZ.value = 0;
267
268     if (!isRenderMode)
269     {
270         UpdateOrientation(centerPos, new Vector3(0f, 0f, 0f));
271     }
272
273     Debug.Log("Center");
274 }
275
276 0 references
277 public void ParkPlatform()
278 {
279     adjTransX.value = adjTransY.value = 0;
280     adjTransZ.value = parkPos.y * scale;
281     adjRotX.value = adjRotY.value = adjRotZ.value = 0;
282
283     if (!isRenderMode)
284     {
285         // only update from slider if output not active
286         UpdateOrientation(parkPos, new Vector3(0f, 0f, 0f));
287     }
288
289     Debug.Log("Park");
290 }
291
292 6 references
293 public void UpdateVal(float val, int index)
294 {
295     print(String.Format("update val {0} = {1}", index, val));
296
297     if (isShuttingDown) return;
298     if (xyzrpyValues == null && xyzrpyValues[index] != null)
299     {
300         xyzrpyValues[index].text = val.ToString();
301
302         if (isShuttingDown) return;
303         Vector3 localPos = chair.transform.localPosition;
304         Vector3 localRot = chair.transform.localRotation.eulerAngles;
305         localPos.x += adjTransX.value / scale;
306         localPos.y += adjTransZ.value / scale;
307
308         localPos.z = adjTransX.value / scale;
309         //Debug.Log("localPos" + localPos);
310         localPos += centerPos;
311
312         localRot.x = adjRotY.value; // positive roll is right side down (sitting in chair)
313         localRot.y = -adjRotZ.value; // positive pitch is nose down
314         localRot.z = -adjRotX.value; // positive yaw is ccm
315
316         Debug.Log(localRot);
317
318         if (!isRenderMode) // only update from slider if output not active
319         {
320             UpdateOrientation(localPos, localRot);
321         }
322         SendPositionMessage();
323     }
324 }
325
326 1 reference
327 private void SendPositionMessage()
328 {
329     if (isShuttingDown) return;
330     String msgStr = String.Format("{0},{1},{2},{3},{4},{5},\n",
331         adjTransX.value, adjTransY.value, adjTransZ.value, adjRotX.value, adjRotY.value, adjRotZ.value);
332
333     Debug.Log(msgStr);
334     byte[] message = Encoding.ASCII.GetBytes(msgStr);
335     if (client != null)
336     {
337         client.Send(message, message.Length, txEndpoint);
338     }
339 }
340
341 0 references
342 public void TransXSliderChanged(float sliderValue) { UpdateVal(sliderValue, 0); }
343
344 0 references
345 public void TransYSliderChanged(float sliderValue) { UpdateVal(sliderValue, 1); }
346
347 0 references
348 public void TransZSliderChanged(float sliderValue) { UpdateVal(sliderValue, 2); }
349
350 0 references
351 public void RotXSliderChanged(float sliderValue) { UpdateVal(sliderValue, 3); }
352
353 0 references
354 public void RotYSliderChanged(float sliderValue) { UpdateVal(sliderValue, 4); }
355
356 0 references
357 public void RotZSliderChanged(float sliderValue) { UpdateVal(sliderValue, 5); }
```

Appendix 17-calibration_software.py GUI snippet



Appendix 18-calibration_software.py code snippets

```

1  from common.serialProcess import SerialProcess # Import SerialProcess for handling serial communication
2  from fluidic_muscle import FluidicMuscle # Import FluidicMuscle module
3  import csv # Import CSV for file handling
4  import pandas as pd # Import pandas for data manipulation
5  import sys # Import sys for system-specific parameters and functions
6  from PyQt5.QtWidgets import QApplication # Import PyQt5 for GUI functionality
7  from GUI import CalibrationApp # Import the base CalibrationApp class from GUI module
8  import os # Import OS for file path handling
9  from platform_pose import Platform # Import Platform for controlling the platform
10 import time # Import time for delays and timing operations
11
12 # Define a new class that extends CalibrationApp
13 class NewCalibrationApp(CalibrationApp):
14     def __init__(self):
15         # Call the parent class constructor to inherit properties and UI components
16         super().__init__()
17
18         # Define calibration parameters
19         self.LOAD = 10 # Default load in kg
20         self.STEP_COUNT = 30 # Number of pressure steps
21         self.CYCLE_COUNT = 1 # Number of calibration cycles
22         self.CURRENT_CYCLE = 0 # Current cycle index
23         self.DELAY_PER_STEP_MS = 500 # Delay per step in milliseconds
24
25         # Define pressure settings
26         self.MAX_PRESSURE_MBAR = 6000 # Maximum pressure in mbar
27         self.PRESSURE_INCREMENT = self.MAX_PRESSURE_MBAR / self.STEP_COUNT # Pressure increment per step
28         self.CURRENT_PRESSURE = self.PRESSURE_INCREMENT # Start pressure
29
30         # Initialize serial encoder and platform objects
31         self.encoder = SerialProcess()
32         self.platform = Platform()
33
34         # Create empty DataFrames for calibration curves
35         self.P_to_D_up_curve_table = pd.DataFrame(columns=["Pressure (mbar)", "Distance (mm)", "Load(kg)"])
36         self.P_to_D_down_curve_table = pd.DataFrame(columns=["Pressure (mbar)", "Distance (mm)", "Load(kg)"])
37
38         # Modify UI button functionality
39         self.pushButton_start.setText("Begin Calibration") # Change button text
40         self.pushButton_start.clicked.connect(self.on_start_button_clicked) # Connect button to custom method
41

```

```

42     # Refresh available COM ports
43     self.refresh_COM_ports()
44     self.pushButton_refresh.clicked.connect(self.refresh_COM_ports)
45
46     # Set default values for UI elements
47     self.spinBox_step_count.setValue(self.STEP_COUNT)
48     self.spinBox_cycle_count.setValue(self.CYCLE_COUNT)
49     self.spinBox_delay.setValue(self.DELAY_PER_STEP_MS)
50
51     # Connect spinbox changes to update methods
52     self.spinBox_step_count.valueChanged.connect(self.update_step_count)
53     self.spinBox_cycle_count.valueChanged.connect(self.update_cycle_count)
54     self.spinBox_delay.valueChanged.connect(self.update_delay)
55
56     # Set default output path
57     self.lineEdit_output_path.setText(os.getcwd())
58
59     # Update functions for UI inputs
60     def update_step_count(self):
61         """Updates the step count and recalculates pressure increment."""
62         self.STEP_COUNT = self.spinBox_step_count.value()
63         self.PRESSURE_INCREMENT = self.MAX_PRESSURE_MBAR / self.STEP_COUNT
64         print(self.PRESSURE_INCREMENT)
65
66     def update_cycle_count(self):
67         """Updates the cycle count when changed in UI."""
68         self.CYCLE_COUNT = self.spinBox_cycle_count.value()
69
70     def update_delay(self):
71         """Updates the delay per step when changed in UI."""
72         self.DELAY_PER_STEP_MS = self.spinBox_delay.value()
73
74     def on_start_button_clicked(self):
75         """Handles the start button click event."""
76         print("Start button pressed in new app!")
77         print(f"Load: {self.LOAD} kg")
78         print(f"Delay: {self.DELAY_PER_STEP_MS} ms")
79         print(f"Step Count: {self.STEP_COUNT}")
80         print(f"Cycle Count: {self.CYCLE_COUNT}")

81
82     self.label_output_path.setText("Processing...")
83
84     self.platform.muscle.send_pressures([0,0,0,0,0,0])
85     time.sleep(self.DELAY_PER_STEP_MS / 1000)
86
87     # Open the selected COM port and retrieve initial load value
88     port = self.comboBox_com_port.currentText()
89     if port:
90         print(port)
91         self.encoder.open_port(port, 115200)
92         time.sleep(0.1) # Wait for queue to fill up
93         self.encoder.write("R".encode())
94         _, load = self.GetDistanceAndLoad()
95         self.LOAD = round(load)
96
97     # Start calibration process
98     self.Calibrate_Cycles()
99
100    def refresh_COM_ports(self):
101        """Refresh the list of available COM ports."""
102        self.comboBox_com_port.clear()
103        ports = self.encoder.list_ports()
104        for p in ports:
105            self.comboBox_com_port.addItem(str(p.device))
106

```

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

```

107     def Calibrate(self):
108         #Reset muscle pressure
109         """Performs the calibration process over multiple cycles."""
110         for cycle in range(self.CYCLE_COUNT):
111             self.CURRENT_PRESSURE = self.PRESSURE_INCREMENT
112             print(f"Cycle {cycle + 1}/{self.CYCLE_COUNT}")
113
114             # Increasing pressure phase
115             while self.CURRENT_PRESSURE <= self.MAX_PRESSURE_MBAR:
116                 time.sleep(self.DELAY_PER_STEP_MS / 2000)
117                 extension, load = self.GetDistanceAndLoad()
118                 time.sleep(self.DELAY_PER_STEP_MS / 2000)
119                 self.P_to_D_up_curve_table = pd.concat(
120                     [self.P_to_D_up_curve_table, pd.DataFrame([{ "Pressure (mbar)": self.CURRENT_PRESSURE, "Distance (mm)": extension,
121                                         "Load(kg)": load }])], ignore_index=True)
122                 self.platform.muscle.send_pressures([int(self.CURRENT_PRESSURE), 0, 0, 0, 0, 0])
123                 self.CURRENT_PRESSURE += self.PRESSURE_INCREMENT
124
125             # Decreasing pressure phase
126             self.CURRENT_PRESSURE -= self.PRESSURE_INCREMENT # Avoid duplicate max pressure reading
127             while self.CURRENT_PRESSURE >= self.PPRESSURE_INCREMENT:
128                 time.sleep(self.DELAY_PER_STEP_MS / 2000)
129                 extension, load = self.GetDistanceAndLoad()
130                 time.sleep(self.DELAY_PER_STEP_MS / 2000)
131                 self.P_to_D_down_curve_table = pd.concat(
132                     [self.P_to_D_down_curve_table, pd.DataFrame([{ "Pressure (mbar)": self.CURRENT_PRESSURE, "Distance (mm)": extension,
133                                         "Load(kg)": load }])], ignore_index=True)
134                 self.platform.muscle.send_pressures([int(self.CURRENT_PRESSURE), 0, 0, 0, 0, 0])
135                 self.CURRENT_PRESSURE -= self.PPRESSURE_INCREMENT
136
137             # Save averaged results to CSV
138             P_to_D_up_curve_table_avg = self.P_to_D_up_curve_table.groupby("Pressure (mbar)").mean().reset_index()
139             P_to_D_down_curve_table_avg = self.P_to_D_down_curve_table.groupby("Pressure (mbar)").mean().reset_index()
140
141             if(self.checkbox_csv_output.isChecked()):
142                 P_to_D_up_curve_table_avg.to_csv(f"{self.lineEdit_output_path.text()}/{self.LOAD}kg_up_curve.csv", index=False)
143                 P_to_D_down_curve_table_avg.to_csv(f"{self.lineEdit_output_path.text()}/{self.LOAD}kg_down_curve.csv", index=False)
144                 print("SAVED CSV FILES TO OUTPUT FOLDER")
145
146     def Calibrate_Cycles(self):
147         """Performs the calibration process over multiple cycles and stores distinct distance/load data for each cycle."""
148
149         # Reset muscle pressure
150         column_names = ["Pressure (mbar)"] + [
151             f"Distance Cycle {i+1} (mm)" for i in range(self.CYCLE_COUNT)
152         ] + [
153             f"Load Cycle {i+1} (kg)" for i in range(self.CYCLE_COUNT)
154         ]
155
156         # Initialize DataFrames
157         self.P_to_D_up_curve_table = pd.DataFrame(columns=column_names)
158         self.P_to_D_down_curve_table = pd.DataFrame(columns=column_names)
159
160         for cycle in range(self.CYCLE_COUNT):
161             self.CURRENT_PRESSURE = self.PPRESSURE_INCREMENT
162             print(f"Cycle {cycle + 1}/{self.CYCLE_COUNT}")
163
164             # Increasing pressure phase
165             while self.CURRENT_PRESSURE <= self.MAX_PRESSURE_MBAR:
166                 time.sleep(self.DELAY_PER_STEP_MS / 2000)
167                 extension, load = self.GetDistanceAndLoad()
168                 time.sleep(self.DELAY_PER_STEP_MS / 2000)
169
170                 if self.CURRENT_PRESSURE in self.P_to_D_up_curve_table["Pressure (mbar)"].values:
171                     row_index = self.P_to_D_up_curve_table.index[self.P_to_D_up_curve_table["Pressure (mbar)"] == self.CURRENT_PRESSURE][0]
172                     self.P_to_D_up_curve_table.at[row_index, f"Distance Cycle {cycle+1} (mm)"] = extension
173                     self.P_to_D_up_curve_table.at[row_index, f"Load Cycle {cycle+1} (kg)"] = load
174                 else:
175                     new_row = {col: None for col in column_names}
176                     new_row["Pressure (mbar)"] = self.CURRENT_PRESSURE
177                     new_row[f"Distance Cycle {cycle+1} (mm)"] = extension
178                     new_row[f"Load Cycle {cycle+1} (kg)"] = load
179                     self.P_to_D_up_curve_table = pd.concat(
180                         [self.P_to_D_up_curve_table, pd.DataFrame([new_row])], ignore_index=True
181                     )
182

```

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

```

183     self.platform.muscle.send_pressures([int(self.CURRENT_PRESSURE), 0, 0, 0, 0])
184     self.CURRENT_PRESSURE += self.PRESSURE_INCREMENT
185
186     # Decreasing pressure phase
187     self.CURRENT_PRESSURE -= self.PRESSURE_INCREMENT # Avoid duplicate max pressure reading
188     while self.CURRENT_PRESSURE >= self.PRESSURE_INCREMENT:
189         time.sleep(self.DELAY_PER_STEP_MS / 2000)
190         extension, load = self.GetDistanceAndLoad()
191         time.sleep(self.DELAY_PER_STEP_MS / 2000)
192
193         if self.CURRENT_PRESSURE in self.P_to_D_down_curve_table["Pressure (mbar)"].values:
194             row_index = self.P_to_D_down_curve_table.index[self.P_to_D_down_curve_table["Pressure (mbar)"] == self.CURRENT_PRESSURE][0]
195             self.P_to_D_down_curve_table.at[row_index, f"Distance Cycle {cycle+1} (mm)"] = extension
196             self.P_to_D_down_curve_table.at[row_index, f"Load Cycle {cycle+1} (kg)"] = load
197
198             new_row = {col: None for col in column_names}
199             new_row["Pressure (mbar)"] = self.CURRENT_PRESSURE
200             new_row[f"Distance Cycle {cycle+1} (mm)"] = extension
201             new_row[f"Load Cycle {cycle+1} (kg)"] = load
202             self.P_to_D_down_curve_table = pd.concat(
203                 [self.P_to_D_down_curve_table, pd.DataFrame([new_row])], ignore_index=True
204             )
205
206             self.platform.muscle.send_pressures([int(self.CURRENT_PRESSURE), 0, 0, 0, 0])
207             self.CURRENT_PRESSURE -= self.PRESSURE_INCREMENT
208
209     # Compute the averaged results across cycles
210     #P_to_D_up_curve_table_avg = self.P_to_D_up_curve_table.groupby("Pressure (mbar)").mean().reset_index()
211     #P_to_D_down_curve_table_avg = self.P_to_D_down_curve_table.groupby("Pressure (mbar)").mean().reset_index()
212
213     # Save CSV files if checkbox is checked
214     if self.checkbox_csv_output.isChecked():
215         self.P_to_D_up_curve_table.to_csv(f"{self.lineEdit_output_path.text()}/{self.LOAD}kg_up_curve.csv", index=False)
216         self.P_to_D_down_curve_table.to_csv(f"{self.lineEdit_output_path.text()}/{self.LOAD}kg_down_curve.csv", index=False)
217         print("SAVED CSV FILES TO OUTPUT FOLDER")
218
219     def GetDistanceAndLoad(self):
220         """Retrieves extension and load values from the encoder."""
221         data = self.encoder.read()
222         print(data)
223         if data is None:
224             print("function does not function!")
225             return (0,0,0,0)
226         array = data.split(",")
227         return (-float(array[1]), float(array[3]) / 9.81) #extension (mm) and load (kg)
228
229     def Calibrate_With_Noise(self):
230         """Performs calibration with multiple snapshots for noise analysis."""
231         SNAPSHOT_COUNT = 100 # Hardcoded number of snapshots per step
232
233         for cycle in range(self.CYCLE_COUNT):
234             self.CURRENT_PRESSURE = self.PRESSURE_INCREMENT
235             print(f"Cycle {cycle + 1}/{self.CYCLE_COUNT}")
236
237             # Increasing pressure phase
238             while self.CURRENT_PRESSURE <= self.MAX_PRESSURE_MBAR:
239                 time.sleep(self.DELAY_PER_STEP_MS / 1000)
240
241                 # Capture multiple snapshots
242                 snapshots = [self.GetDistanceAndLoad() for _ in range(SNAPSHOT_COUNT)]
243                 distances, loads = zip(*snapshots)
244                 avg_distance = sum(distances) / SNAPSHOT_COUNT
245                 avg_load = sum(loads) / SNAPSHOT_COUNT
246
247                 # Calculate snapshot ranges
248                 distance_range = max(distances) - min(distances)
249                 load_range = max(loads) - min(loads)
250
251                 # Store data in DataFrame
252                 row_data = {"Pressure (mbar)": self.CURRENT_PRESSURE,
253                             "Distance (mm)": avg_distance,
254                             "Load(kg)": avg_load,
255                             "Distance Range (mm)": distance_range, # Add snapshot range for Distance
256                             "Load Range (kg)": load_range} # Add snapshot range for Load

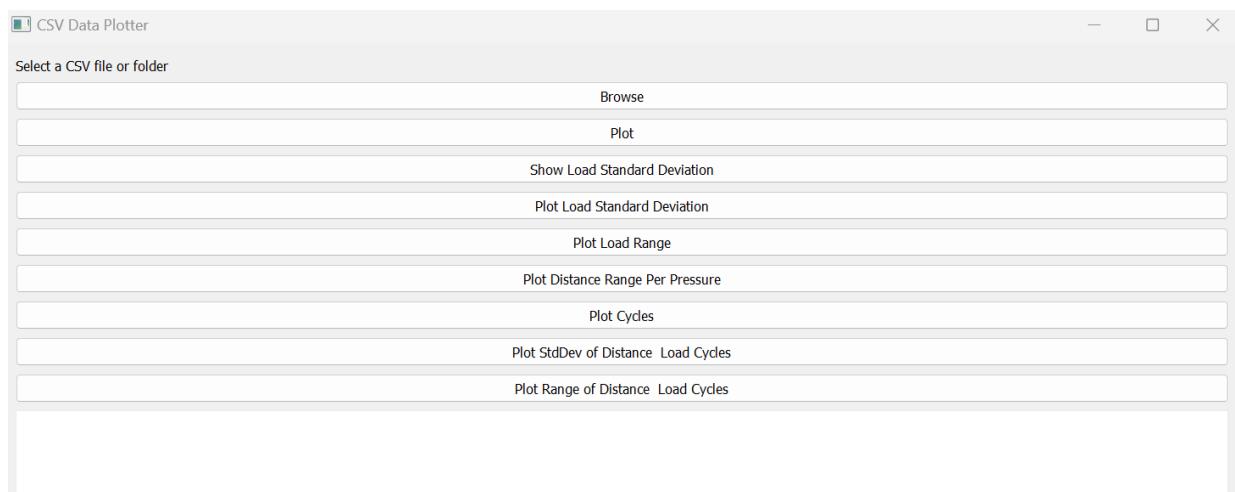
```

```

257
258         # Adding individual snapshot values to row data
259         for i in range(SNAPSHOT_COUNT):
260             row_data[f"Distance Snapshot {i+1}"] = distances[i]
261             row_data[f"Load Snapshot {i+1}"] = loads[i]
262
263             self.P_to_D_up_curve_table = pd.concat([self.P_to_D_up_curve_table, pd.DataFrame([row_data])], ignore_index=True)
264
265             self.platform.muscle.send_pressures([int(self.CURRENT_PRESSURE), 0, 0, 0, 0, 0])
266             self.CURRENT_PRESSURE += self.PRESSURE_INCREMENT
267
268             # Decreasing pressure phase
269             self.CURRENT_PRESSURE -= self.PRESSURE_INCREMENT # Avoid duplicate max pressure reading
270             while self.CURRENT_PRESSURE >= self.PRESSURE_INCREMENT:
271                 time.sleep(self.DELAY_PER_STEP_MS / 1000)
272
273             snapshots = [self.GetDistanceAndLoad() for _ in range(SNAPSHOT_COUNT)]
274             distances, loads = zip(*snapshots)
275             avg_distance = sum(distances) / SNAPSHOT_COUNT
276             avg_load = sum(loads) / SNAPSHOT_COUNT
277
278             # Calculate snapshot ranges
279             distance_range = max(distances) - min(distances)
280             load_range = max(loads) - min(loads)
281
282             row_data = {"Pressure (mbar)": self.CURRENT_PRESSURE,
283                         "Distance (mm)": avg_distance,
284                         "Load(kg)": avg_load,
285                         "Distance Range (mm)": distance_range, # Add snapshot range for Distance
286                         "Load Range (kg)": load_range} # Add snapshot range for Load
287
288             # Adding individual snapshot values to row data
289             for i in range(SNAPSHOT_COUNT):
290                 row_data[f"Distance Snapshot {i+1}"] = distances[i]
291                 row_data[f"Load Snapshot {i+1}"] = loads[i]
292
293             self.P_to_D_down_curve_table = pd.concat([self.P_to_D_down_curve_table, pd.DataFrame([row_data])], ignore_index=True)
294
295             self.platform.muscle.send_pressures([int(self.CURRENT_PRESSURE), 0, 0, 0, 0, 0])
296             self.CURRENT_PRESSURE -= self.PRESSURE_INCREMENT
297
298             # Save averaged results to CSV
299             P_to_D_up_curve_table_avg = self.P_to_D_up_curve_table.groupby("Pressure (mbar)").mean().reset_index()
300             P_to_D_down_curve_table_avg = self.P_to_D_down_curve_table.groupby("Pressure (mbar)").mean().reset_index()
301
302             if self.checkbox_csv_output.isChecked():
303                 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)
304                 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)
305                 print("SAVED CSV FILES WITH NOISE DATA TO OUTPUT FOLDER")
306
307             if __name__ == "__main__":
308                 app = QApplication([])
309                 window = NewCalibrationApp()
310                 window.show()
311                 app.exec_()

```

Appendix 19-csv_muscle_plotter.py GUI



Appendix 20-csv_muscle_plotter.py code snippets

```

1 import sys # System-specific parameters and functions
2 import os # Provides functions for interacting with the operating system
3 import pandas as pd # Library for data manipulation and analysis
4 import matplotlib.pyplot as plt # Library for creating static, animated, and interactive visualizations
5 import matplotlib.cm as cm
6 from PyQt5.QtWidgets import QApplication, QWidget, QVBoxLayout, QPushButton, QLabel, QTextEdit # PyQt5 modules for GUI development
7 import re
8
9 class CSVPlotter(QWidget): # Defines a class that inherits from QWidget for creating a GUI application
10     def __init__(self):
11         super().__init__() # Calls the parent class constructor
12         self.initUI() # Initializes the user interface
13
14     def initUI(self):
15         self.CMAP = "viridis"
16         layout = QVBoxLayout() # Creates a vertical box layout
17
18         self.label = QLabel("Select a CSV file or folder") # Label to display selected file/folder
19         layout.addWidget(self.label) # Adds label to layout
20
21         self.btn_browse = QPushButton("Browse", self) # Button for browsing files/folders
22         self.btn_browse.clicked.connect(self.browse) # Connects button click event to browse method
23         layout.addWidget(self.btn_browse) # Adds button to layout
24
25         self.btn_plot = QPushButton("Plot", self) # Button for plotting data
26         self.btn_plot.clicked.connect(self.plot_data) # Connects button click event to plot_data method
27         layout.addWidget(self.btn_plot) # Adds button to layout
28
29         self.btn_stddev = QPushButton("Show Load Standard Deviation", self) # Button to show standard deviation for Load
30         self.btn_stddev.clicked.connect(self.show_stddev) # Connects button click event to show_stddev method
31         layout.addWidget(self.btn_stddev) # Adds button to layout
32
33         self.btn_plot_stddev = QPushButton("Plot Load Standard Deviation", self) # Button to plot standard deviation for Load
34         self.btn_plot_stddev.clicked.connect(self.plot_stddev) # Connects button click event to plot_stddev method
35         layout.addWidget(self.btn_plot_stddev) # Adds button to layout
36
37         self.btn_plot_range = QPushButton("Plot Load Range", self) # Button to plot load range
38         self.btn_plot_range.clicked.connect(self.plot_load_range) # Connect to method
39         layout.addWidget(self.btn_plot_range) # Add button to layout
40
41         self.btn_plot_noise = QPushButton("Plot Distance Range Per Pressure", self) # Button to plot noise data
42         self.btn_plot_noise.clicked.connect(self.plot_noise) # Connect to method
43         layout.addWidget(self.btn_plot_noise) # Add button to layout
44
45         self.btn_plot_noisy = QPushButton("Plot Cycles", self) # Button to plot noise data
46         self.btn_plot_noisy.clicked.connect(self.plot_cycle_data) # Connect to method
47         layout.addWidget(self.btn_plot_noisy) # Add button to layout
48
49         self.btn_plot_stddev_cycles = QPushButton("Plot StdDev of Distance & Load Cycles", self)
50         self.btn_plot_stddev_cycles.clicked.connect(self.plot_standard_deviation_cycles)
51         layout.addWidget(self.btn_plot_stddev_cycles)
52
53         self.btn_plot_range_cycles = QPushButton("Plot Range of Distance & Load Cycles", self)
54         self.btn_plot_range_cycles.clicked.connect(self.plot_range_cycles)
55         layout.addWidget(self.btn_plot_range_cycles)
56
57         self.text_output = QTextEdit(self) # Text box to display standard deviation results
58         self.text_output.setReadOnly(True)
59         layout.addWidget(self.text_output)
60
61         self.setLayout(layout) # Sets the layout for the window
62         self.setWindowTitle("CSV Data Plotter") # Sets the window title
63         self.setGeometry(200, 200, 500, 300) # Defines the size and position of the window
64
65         self.file_paths = [] # Initializes an empty list to store selected file(s)
66
67     def browse(self):
68         options = QFileDialog.Options()
69         path = QFileDialog.getExistingDirectory(self, "Select Folder", "", options=options) # Opens a folder selection dialog
70
71         if path: # If a folder is selected
72             self.file_paths = self.find_csv_files(path) # Finds all CSV files in the selected folder
73             self.label.setText(f"Selected Folder: {path}") # Updates label text
74         else: # If a file is selected
75             file_path, _ = QFileDialog.getOpenFileName(self, "Select CSV File", "", "CSV Files (*.csv)", options=options) # Opens a file selection dialog
76             if file_path:
77                 self.file_paths = self.find_csv_pairs(file_path) # Finds corresponding up/down curve pairs
78                 self.label.setText(f"Selected File: {file_path}") # Updates label text
79
80     def find_csv_files(self, folder_path):
81         """Find all valid CSV files in the folder."""
82         csv_files = [os.path.join(folder_path, f) for f in os.listdir(folder_path) if f.endswith(".csv")]
83         return csv_files # Returns the list of CSV file paths
84
85     def find_csv_pairs(self, file_path):
86         """Find the corresponding up/down curve pair if a single file is selected."""
87         base_name = os.path.basename(file_path) # Extracts the file name
88         folder = os.path.dirname(file_path) # Extracts the folder path
89

```

```

90     if "up_curve" in base_name: # Checks if the file is an up curve file
91         paired_file = base_name.replace("up_curve", "down_curve") # Finds corresponding down curve file
92     elif "down_curve" in base_name: # Checks if the file is a down curve file
93         paired_file = base_name.replace("down_curve", "up_curve") # Finds corresponding up curve file
94     else:
95         return [file_path] # Returns the selected file if no pair is found
96
97     paired_path = os.path.join(folder, paired_file) # Constructs the full path of the paired file
98     return [file_path, paired_path] if os.path.exists(paired_path) else [file_path] # Returns both files if the pair exists
99
100    def show_stddev(self):
101        if not self.file_paths:
102            self.text_output.setText("No valid CSV files selected!")
103            return
104
105        result_text = "Standard Deviation of Load Column:\n"
106
107        for file in self.file_paths:
108            try:
109                df = pd.read_csv(file)
110                if "Load(kg)" in df.columns:
111                    stddev = df["Load(kg)"].std()
112                    result_text += f"\nFile: {os.path.basename(file)}\nLoad Standard Deviation: {stddev:.4f}\n"
113                else:
114                    result_text += f"\nFile: {os.path.basename(file)}\nLoad column not found!\n"
115            except Exception as e:
116                result_text += f"\nError reading {file}: {e}\n"
117
118        self.text_output.setText(result_text)
119
120    def plot_stddev(self):
121        if not self.file_paths:
122            self.text_output.setText("No valid CSV files selected!")
123            return
124
125        file_names = []
126        stddev_values = []
127        base_names = [] # Stores unique load identifiers
128        color_map = {} # Maps base names to colors
129
130        # Extract base names and compute standard deviations
131        for file in self.file_paths:
132            try:
133                df = pd.read_csv(file)
134                if "Load(kg)" in df.columns:
135                    stddev = df["Load(kg)"].std()
136                    file_name = os.path.basename(file)
137
138                    # Extract base name (removing "up_curve" or "down_curve")
139                    base_name = file_name.replace("up_curve", "").replace("down_curve", "").replace(".csv", "")
140
141                    file_names.append(file_name)
142                    stddev_values.append(stddev)
143                    base_names.append(base_name)
144            except Exception as e:
145                print(f"Error reading {file}: {e}")
146
147        if not file_names:
148            self.text_output.setText("No valid CSV files found!")
149            return
150
151        # Generate unique colors for each base name
152        unique_base_names = list(set(base_names))
153        cmap = cm.get_cmap(self.CMAP, len(unique_base_names)) # Use a colormap
154        colors = {name: cmap(i) for i, name in enumerate(unique_base_names)} # Assign colors
155
156        # Plot bars with shared colors for each base name
157        plt.figure(figsize=(10, 6))
158        bars = [plt.bar(file_names[i], stddev_values[i], color=colors[base_names[i]]) for i in range(len(file_names))]
159
160        # Create a legend mapping base names to colors
161        legend_handles = [plt.Rectangle((0, 0), 1, 1, color=colors[name]) for name in unique_base_names]
162        plt.legend(legend_handles, unique_base_names, title="Load Groups", loc="upper left")
163
164        plt.xlabel("CSV Files")
165        plt.ylabel("Standard Deviation of Load (kg)")
166        plt.title("Load Standard Deviation for Each CSV File")
167        plt.xticks(rotation=45, ha="right")
168        plt.tight_layout()
169        plt.show()
170

```

```

171     def plot_cycle_data(self):
172         """Plots all Distance and Load Cycles vs Pressure for multiple CSV files."""
173
174         if not self.file_paths:
175             self.label.setText("No valid CSV files selected!")
176             return
177
178         plt.figure()
179
180         # Create figure and primary axis for Pressure vs Distance Cycles
181         fig, ax1 = plt.subplots(figsize=(10, 6))
182         ax2 = ax1.twinx() # Secondary y-axis for Load Cycles
183
184         base_names = []
185         data = {}
186
187         # Extract base names and group files
188         for file in self.file_paths:
189             try:
190                 file_name = os.path.basename(file)
191                 base_name = file_name.replace("up_curve", "").replace("down_curve", "").replace(".csv", "").strip()
192
193                 if base_name not in data:
194                     data[base_name] = []
195                     data[base_name].append(file)
196
197                 if base_name not in base_names:
198                     base_names.append(base_name)
199             except Exception as e:
200                 print(f"Error processing {file}: {e}")
201
202         # Define explicit contrasting colors for up/down curves
203         color_pairs = {
204             "12kg_": ("red", "blue"),
205             "13kg_": ("green", "purple"),
206             "24kg_": ("orange", "cyan"),
207             "33kg_": ("brown", "magenta"),
208             "43kg_": ("yellow", "purple")
209         }
210
211         max_load_value = 0 # Track max load value for scaling
212
213         name = ""
214         # Loop through grouped data
215         for base_name, files in data.items():
216             name = base_name
217             if base_name not in color_pairs:
218                 print(f"Warning: No predefined color pair for {base_name}, skipping.")
219                 continue
220
221             up_color, down_color = color_pairs[base_name]
222
223             for file in files:
224                 try:
225                     df = pd.read_csv(file)
226                     label_suffix = "Up" if "up_curve" in file else "Down"
227                     base_color = up_color if "up_curve" in file else down_color
228
229                     # Extract Distance and Load cycle columns dynamically
230                     distance_cols = sorted(
231                         [col for col in df.columns if "Distance Cycle" in col],
232                         key=lambda x: int(x.split("Cycle ")[-1].split(" ")[0]),
233                         reverse=True # Ensure Cycle 5 is first (darkest)
234                     )
235                     load_cols = sorted(
236                         [col for col in df.columns if "Load Cycle" in col],
237                         key=lambda x: int(x.split("Cycle ")[-1].split(" ")[0]),
238                         reverse=True
239                     )
240
241                     # Update max load value
242                     max_load_value = max(max_load_value, df[load_cols].values.max())
243
244                     # Generate shades of base color for cycles (lightest to darkest)
245                     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)
246                                     for i in range(len(distance_cols))]
247                     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)
248                                     for i in range(len(load_cols))]
249
250                     # Plot each Distance cycle on the primary y-axis
251                     for i, cycle_col in enumerate(distance_cols):
252                         ax1.plot(df["Pressure (mbar)"], df[cycle_col],
253                                 label=f"{base_name} {cycle_col} ({label_suffix})",
254                                 marker="o", linestyle="-", color=distance_shades[len(distance_shades)-i-1], alpha=0.8)

```

```

256     # Plot each load cycle on the secondary y-axis
257     for i, cycle_col in enumerate(load_cols):
258         ax2.plot(df["Pressure (mbar)"], df[cycle_col],
259                  label=f"{base_name} {cycle_col} ({label_suffix})",
260                  marker="x", linestyle=(0, (1, 1)), color=load_shades[len(load_shades)-i-1], alpha=0.8) # Fine dotted line
261
262     except Exception as e:
263         print(f"Error reading {file}: {e}")
264
265     name = name.replace("_", "")
266
267     # Configure axis labels and title
268     ax1.set_xlabel("Pressure (mbar)", fontsize=20)
269     ax1.set_ylabel("Distance (mm)", color="tab:blue", fontsize=20)
270     ax2.set_ylabel("Load (kg)", color="tab:red", fontsize=20)
271     ax1.set_title(f"Pressure vs Distance and Load Cycles {name}", fontsize=30)
272
273     # Set load axis limit with a margin
274     ax2.set_ylim(0, max_load_value * 1.1) # 10% margin above max load
275
276     # Combine legends from both axes
277     ax1_handles, ax1_labels = ax1.get_legend_handles_labels()
278     ax2_handles, ax2_labels = ax2.get_legend_handles_labels()
279     ax1.legend(ax1_handles + ax2_handles, ax1_labels + ax2_labels, loc="upper left", fontsize="small", bbox_to_anchor=(-0.2, 1))
280
281     # Show grid for better readability
282     ax1.grid(True)
283
284     # Show the plot
285     plt.tight_layout()
286     plt.show()
287
288
289 def plot_load_range(self):
290     if not self.file_paths:
291         self.text_output.setText("No valid CSV files selected!")
292         return
293
294     file_names = []
295     load_ranges = []
296     file_base_names = [] # List to track base names for each file
297
298     data = {}
299
300     # Extract base names and group files
301     for file in self.file_paths:
302         try:
303             file_name = os.path.basename(file)
304             base_name = file_name.replace("up_curve", "").replace("down_curve", "").replace(".csv", "")
305
306             if base_name not in data:
307                 data[base_name] = []
308                 data[base_name].append(file)
309             except Exception as e:
310                 print(f"Error processing {file}: {e}")
311
312             # Generate unique colors for each base name
313             base_names = list(data.keys()) # Get all unique base names
314             cmap = cm.get_cmap(self.CMAP, len(base_names)) # Assign unique colors
315             colors = {name: cmap(i) for i, name in enumerate(base_names)}
316
317             # Extract load ranges
318             for base_name, files in data.items():
319                 for file in files:
320                     try:
321                         df = pd.read_csv(file)
322                         if "Load(kg)" in df.columns:
323                             load_range = df["Load(kg)"].max() - df["Load(kg)"].min()
324                             file_names.append(os.path.basename(file)) # Store filename
325                             load_ranges.append(load_range) # Store load range
326                             file_base_names.append(base_name) # Track base name for color mapping
327                         except Exception as e:
328                             print(f"Error reading {file}: {e}")
329
330             if not file_names:
331                 self.text_output.setText("No valid CSV files found!")
332                 return
333
334             # Plot bars with shared colors for each base name
335             plt.figure(figsize=(10, 6))
336             bars = [plt.bar(file_names[i], load_ranges[i], color=colors[file_base_names[i]]) for i in range(len(file_names))]
337
338             # Create a legend mapping base names to colors
339             legend_handles = [plt.Rectangle((0, 0), 1, 1, color=colors[name]) for name in base_names]
340             plt.legend(legend_handles, base_names, title="Load Groups", loc="upper left")
341
342             plt.xlabel("CSV Files")
343             plt.ylabel("Load Range (kg)")
344             plt.title("Load Range for Each CSV File")
345             plt.xticks(rotation=45, ha="right")

```

```

346     plt.tight_layout()
347     plt.show()
348
349     def plot_data(self):
350         if not self.file_paths:
351             self.label.setText("No valid CSV files selected!")
352             return
353
354         plt.figure()
355
356         # Create figure and primary axis for Pressure vs Distance
357         fig, ax1 = plt.subplots(figsize=(10, 6))
358         ax2 = ax1.twinx() # Secondary y-axis for Load
359
360         base_names = []
361         data = {}
362
363         # Extract base names and group files
364         for file in self.file_paths:
365             try:
366                 file_name = os.path.basename(file)
367
368                 # Remove "_up_curve_averaged_x_y" and "_down_curve_averaged_x_y" portions
369                 base_name = re.sub(r"_(up_curve|down_curve)_averaged_\d+_\d+", "", file_name)
370                 base_name = base_name.replace(".csv", "").strip()
371
372                 if base_name not in data:
373                     data[base_name] = []
374                 data[base_name].append(file)
375
376                 if base_name not in base_names:
377                     base_names.append(base_name)
378             except Exception as e:
379                 print(f"Error processing {file}: {e}")
380
381         # Define explicit contrasting colors for up/down curves
382         color_pairs = {
383             "13kg": ("red", "orange"),
384             "24kg": ("blue", "cyan"),
385             "33kg": ("yellow", "gold"),
386             "43kg": ("black", "red")
387         }
388
389         # Loop through grouped data
390         for base_name, files in data.items():
391             if base_name not in color_pairs:
392                 print(f"Warning: No predefined color pair for {base_name}, skipping.")
393                 continue
394
395             up_color, down_color = color_pairs[base_name]
396
397             for file in files:
398                 try:
399                     df = pd.read_csv(file)
400                     is_up_curve = "up_curve" in file
401                     label_suffix = "Up" if is_up_curve else "Down"
402                     base_color = up_color if is_up_curve else down_color
403
404                     # Plot Pressure vs Distance on primary y-axis (solid line)
405                     ax1.plot(df["Pressure (mbar)"], df["Distance (mm)"],
406                             label=f"{base_name} Distance ({label_suffix})",
407                             marker="o", linestyle="-", color=base_color, alpha=0.8)
408
409                     # Plot Pressure vs Load on secondary y-axis (dashed line with x markers)
410                     ax2.plot(df["Pressure (mbar)"], df["Load(kg)"],
411                             label=f"{base_name} Load ({label_suffix})",
412                             marker="x", linestyle="--", color=base_color, alpha=0.8)
413
414                 except Exception as e:
415                     print(f"Error reading {file}: {e}")
416
417             # Configure axis labels and title
418             ax1.set_xlabel("Pressure (mbar)", fontsize=20)
419             ax1.set_ylabel("Distance (mm)", color="tab:blue", fontsize=20)
420             ax2.set_ylabel("Load (kg)", color="tab:red", fontsize=20)
421             ax1.set_title("Pressure vs Distance and Load (Up and Down Curves for Cycles 3 to 5)", fontsize=30)
422
423             # Set load axis limit with a margin
424             ax2.set_ylim(0, 50)
425
426             # Combine legends from both axes
427             ax1_handles, ax1_labels = ax1.get_legend_handles_labels()
428             ax2_handles, ax2_labels = ax2.get_legend_handles_labels()
429             ax1.legend(ax1_handles + ax2_handles, ax1_labels + ax2_labels, loc="upper left", fontsize="small", bbox_to_anchor=(-0.2, 1))
430
431             # Show grid for better readability
432             ax1.grid(True)

```

```

433     # Show the plot
434     plt.tight_layout()
435     plt.show()
436
437
438     def plot_noise(self):
439         if not self.file_paths:
440             self.text_output.setText("No valid CSV files selected!")
441             return
442
443         plt.figure(figsize=(10, 6)) # Create a new figure for plotting
444         colors = cm.get_cmap(self.CMAP, len(self.file_paths)) # Generate colors per file
445         file_colors = {} # Dictionary to track colors assigned per file
446
447         for i, file in enumerate(self.file_paths):
448             try:
449                 df = pd.read_csv(file)
450
451                 # Ensure that the necessary columns are present
452                 if "Pressure (mbar)" in df.columns and "Distance Range (mm)" in df.columns:
453                     pressure = df["Pressure (mbar)"].values
454                     distance_range = df["Distance Range (mm)"].values
455
456                     # Assign a unique color per file
457                     color = colors(i)
458                     file_name = os.path.basename(file)
459
460                     # Determine if it's an "Up Curve" or "Down Curve"
461                     curve_type = "Up Curve" if "up_curve" in file_name else "Down Curve"
462
463                     # Avoid duplicate legend labels (one per file)
464                     if file_name not in file_colors:
465                         plt.plot(pressure, distance_range, 'o-', color=color, label=curve_type)
466                         file_colors[file_name] = color # Store assigned color
467                     else:
468                         plt.plot(pressure, distance_range, 'o-', color=color) # No label to avoid duplicate
469
470             except Exception as e:
471                 print(f"Error reading {file}: {e}")
472
473             # Set plot labels and title
474             plt.xlabel("Pressure (mbar)")
475             plt.ylabel("Distance Range (mm)")
476             plt.title("Distance Range vs Pressure for Each Curve")
477
478             # Display the legend with only "Up Curve" and "Down Curve" once
479             plt.legend(loc="upper left")
480
481             # Add grid for better readability
482             plt.grid(True)
483
484             # Adjust layout to avoid clipping
485             plt.tight_layout()
486
487             # Show the plot
488             plt.show()
489
490     def plot_range_cycles(self):
491         """Plots the standard deviation of Distance and Load across 5 cycles for each pressure setting."""
492         if not self.file_paths:
493             self.text_output.setText("No valid CSV files selected!")
494             return
495
496         plt.figure(figsize=(10, 6))
497
498         fig, ax1 = plt.subplots(figsize=(10, 6))
499         ax2 = ax1.twinx() # Secondary y-axis for Load Standard Deviation
500
501
502         # Extract unique base filenames
503         base_names = [os.path.basename(file) for file in self.file_paths]
504
505         # Generate unique colors for each base name using a colormap
506         cmap = cm.get_cmap(self.CMAP, len(base_names)) # Use the specified colormap
507         colors = {name: cmap(i) for i, name in enumerate(base_names)}
508
509         for file in self.file_paths:
510             try:
511                 df = pd.read_csv(file)
512                 file_name = os.path.basename(file)
513
514                 # Identify distance and load cycle columns
515                 distance_cols = [col for col in df.columns if "Distance Cycle" in col]
516                 load_cols = [col for col in df.columns if "Load Cycle" in col]
517
518                 # Ensure there are exactly 5 cycles
519                 if len(distance_cols) != 5 or len(load_cols) != 5:
520                     print(f"Skipping {file_name}: Expected 5 cycles, found {len(distance_cols)} distance and {len(load_cols)} load cycles.")
521                     continue

```

```

522
523         # Compute range for each pressure setting
524         distance_range = df[distance_cols].max(axis=1) - df[distance_cols].min(axis=1)
525         load_range = df[load_cols].max(axis=1) - df[load_cols].min(axis=1)
526
527         # Extract Pressure column
528         pressure = df["Pressure (mbar)"]
529
530         # Assign a unique color to this file
531         color = colors[file_name]
532
533         # Plot Range of Distance
534         ax1.plot(pressure, distance_range, marker="o", linestyle="--", label=f"{file_name} Distance Range", color=color)
535
536         # Plot Range of Load
537         ax2.plot(pressure, load_range, marker="x", linestyle="--", label=f"{file_name} Load Range", color=color)
538
539     except Exception as e:
540         print(f"Error processing {file_name}: {e}")
541
542     # Configure axes labels and title
543     ax1.set_xlabel("Pressure (mbar)")
544     ax1.set_ylabel("Distance Range (mm)", color="blue")
545     ax2.set_ylabel("Load Range (kg)", color="red")
546     ax1.set_title("Range of Distance & Load across 5 Cycles")
547
548     # Combine legends from both axes
549     ax1_handles, ax1_labels = ax1.get_legend_handles_labels()
550     ax2_handles, ax2_labels = ax2.get_legend_handles_labels()
551     ax1.legend(ax1_handles + ax2_handles, ax1_labels + ax2_labels, loc="upper left")
552
553     ax1.grid(True)
554     plt.tight_layout()
555     plt.show()
556
557 def plot_standard_deviation_cycles(self):
558     """Plots the standard deviation of Distance and Load across 5 cycles for each pressure setting."""
559     if not self.file_paths:
560         self.text_output.setText("No valid CSV files selected!")
561         return
562
563     plt.figure(figsize=(10, 6))
564
565     fig, ax1 = plt.subplots(figsize=(10, 6))
566     ax2 = ax1.twinx() # Secondary y-axis for Load Standard Deviation
567
568     # Extract unique base filenames
569     base_names = [os.path.basename(file) for file in self.file_paths]
570
571     # Generate unique colors for each base name using a colormap
572     cmap = cm.get_cmap(self.CMAP, len(base_names)) # Use the specified colormap
573     colors = {name: cmap(i) for i, name in enumerate(base_names)}
574
575     for file in self.file_paths:
576         try:
577             df = pd.read_csv(file)
578             file_name = os.path.basename(file)
579
580             # Identify distance and load cycle columns
581             distance_cols = [col for col in df.columns if "Distance Cycle" in col]
582             load_cols = [col for col in df.columns if "Load Cycle" in col]
583
584             # Ensure there are exactly 5 cycles
585             if len(distance_cols) != 5 or len(load_cols) != 5:
586                 print(f"Skipping {file_name}: Expected 5 cycles, found {len(distance_cols)} distance and {len(load_cols)} load cycles.")
587                 continue
588
589             # Compute standard deviation for each pressure setting
590             distance_std = df[distance_cols].std(axis=1)
591             load_std = df[load_cols].std(axis=1)
592
593             # Extract Pressure column
594             pressure = df["Pressure (mbar)"]
595
596             # Assign a unique color to this file
597             color = colors[file_name]
598
599             # Plot Standard Deviation of Distance
600             ax1.plot(pressure, distance_std, marker="o", linestyle="--", label=f"{file_name} Distance StdDev", color=color)
601
602             # Plot Standard Deviation of Load
603             ax2.plot(pressure, load_std, marker="x", linestyle="--", label=f"{file_name} Load StdDev", color=color)
604
605         except Exception as e:
606             print(f"Error processing {file}: {e}")
607
608         # Configure axes labels and title
609         ax1.set_xlabel("Pressure (mbar)")
610         ax1.set_ylabel("Distance Standard Deviation (mm)", color="blue")
611         ax2.set_ylabel("Load Standard Deviation (kg)", color="red")

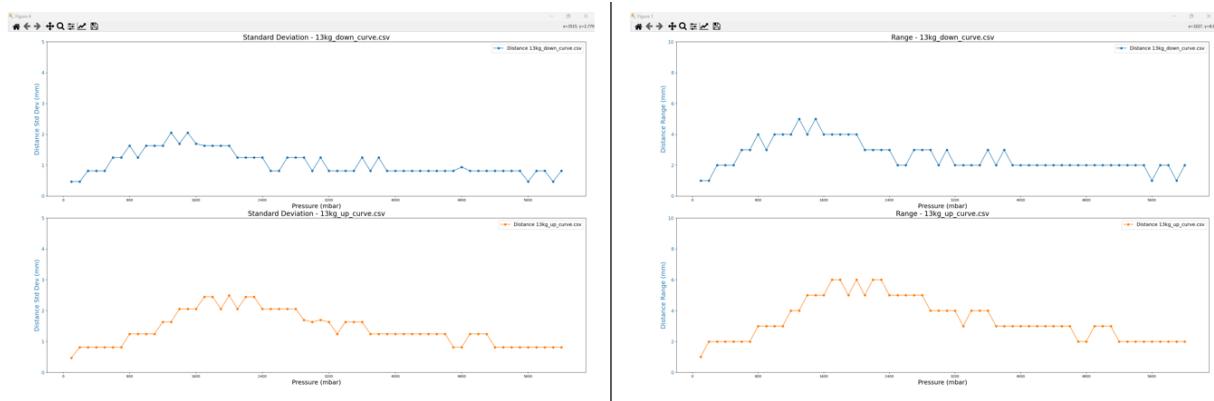
```

```

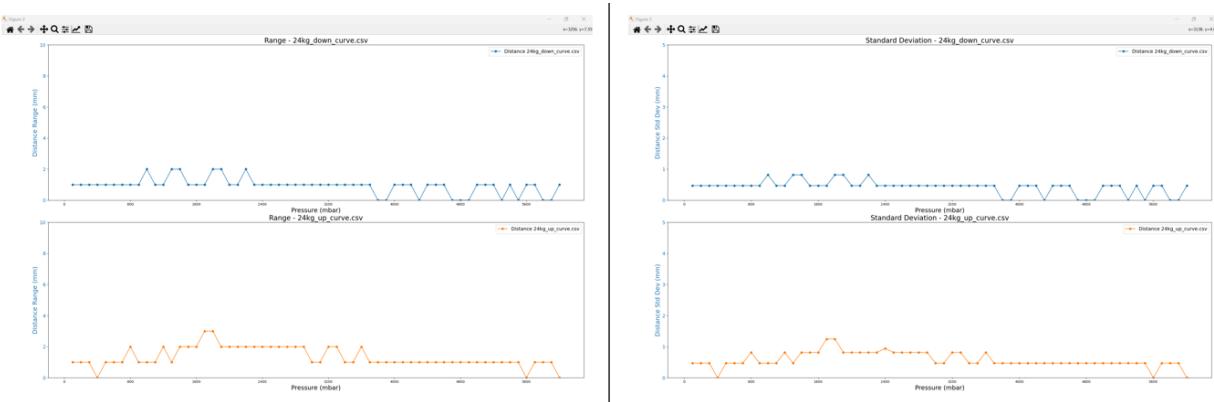
612     ax1.set_title("Standard Deviation of Distance & Load across 5 Cycles")
613
614     # Combine legends from both axes
615     ax1_handles, ax1_labels = ax1.get_legend_handles_labels()
616     ax2_handles, ax2_labels = ax2.get_legend_handles_labels()
617     ax1.legend(ax1_handles + ax2_handles, ax1_labels + ax2_labels, loc="upper left")
618
619     ax1.grid(True)
620     plt.tight_layout()
621     plt.show()
622
623
624 if __name__ == "__main__":
625     app = QApplication(sys.argv)
626     window = CSVPlotter()
627     window.show()
628     sys.exit(app.exec_())

```

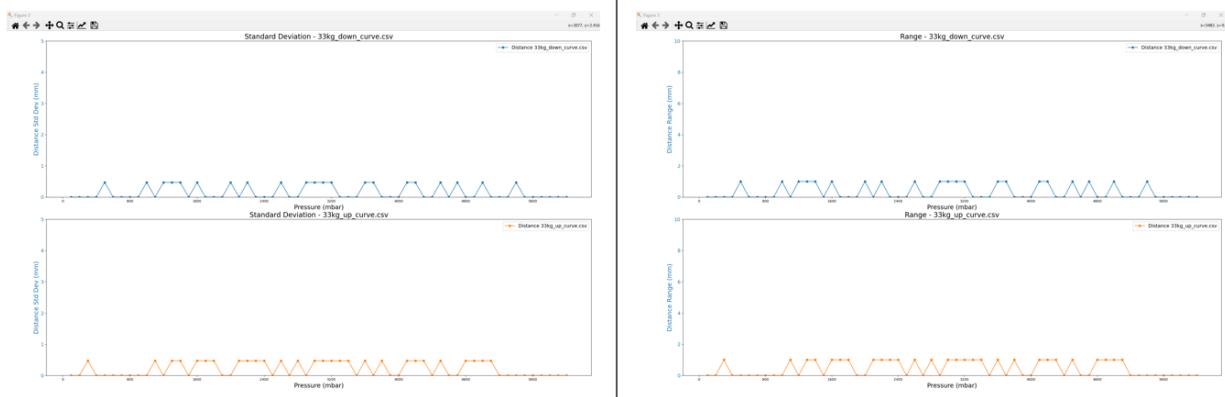
Appendix 21-stdev and range plot of final 13kg plot



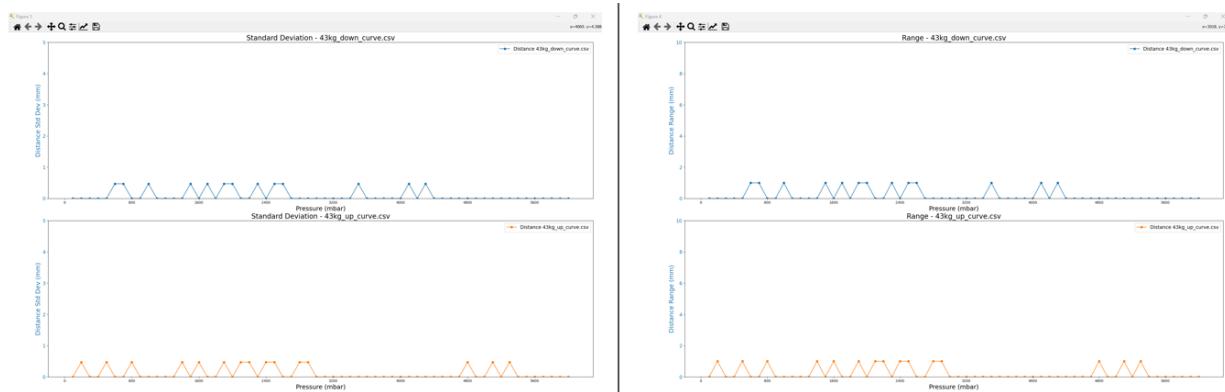
Appendix 22-stdev and range plot of final 24kg plot



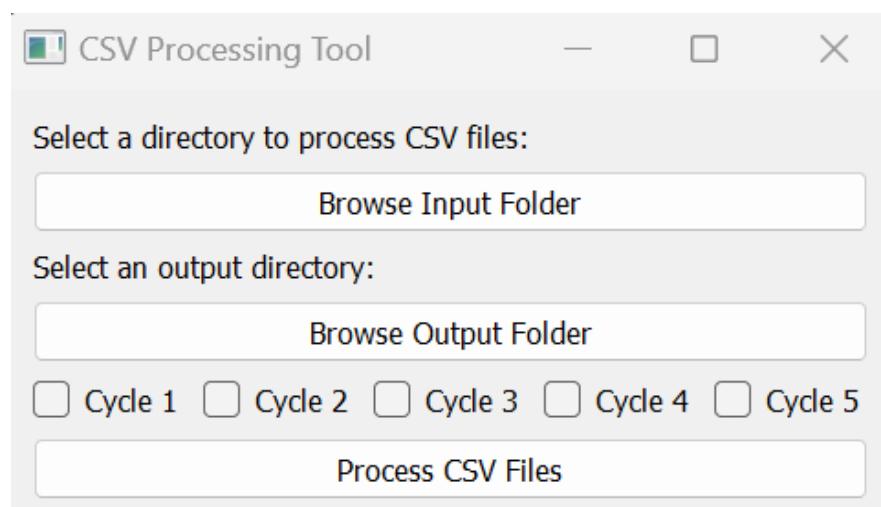
Appendix 23-stdev and range plot of final 33kg plot



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

```

1 import sys
2 import os
3 import pandas as pd
4 from PyQt5.QtWidgets import (
5     QApplication, QWidget, QPushButton, QFileDialog, QVBoxLayout, QLabel, QCheckBox, QHBoxLayout, QMessageBox
6 )
7
8 class CSVProcessor(QWidget):
9     def __init__(self):
10         super().__init__()
11         self.initUI()
12         self.selected_cycles = []      # List to store selected cycle numbers
13         self.output_directory = ""    # Path for output directory
14
15     def initUI(self):
16         layout = QVBoxLayout()
17
18         # Label and button to select input directory
19         self.label = QLabel("Select a directory to process CSV files:")
20         layout.addWidget(self.label)
21
22         self.btnBrowse = QPushButton("Browse Input Folder")
23         self.btnBrowse.clicked.connect(self.browseFolder)
24         layout.addWidget(self.btnBrowse)
25
26         # Label and button to select output directory
27         self.output_label = QLabel("Select an output directory:")
28         layout.addWidget(self.output_label)
29
30         self.btnOutputBrowse = QPushButton("Browse Output Folder")
31         self.btnOutputBrowse.clicked.connect(self.browseOutputFolder)
32         layout.addWidget(self.btnOutputBrowse)
33
34         # Create checkboxes for selecting cycles 1 to 5
35         self.checkboxes = []
36         self.checkbox_layout = QHBoxLayout()
37         for i in range(1, 6):
38             checkbox = QCheckBox(f"Cycle {i}")
39             checkbox.stateChanged.connect(self.updateSelectedCycles)
40             self.checkboxes.append(checkbox)
41             self.checkbox_layout.addWidget(checkbox)
42         layout.addLayout(self.checkbox_layout)
43
44         # Button to process CSV files
45         self.btnProcess = QPushButton("Process CSV Files")
46         self.btnProcess.clicked.connect(self.processFiles)
47         layout.addWidget(self.btnProcess)

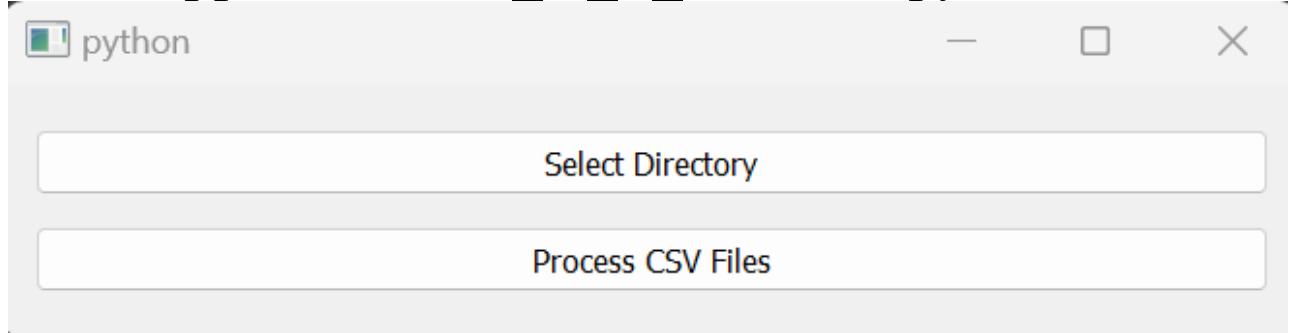
```

```

48         self.setLayout(layout)
49         self.setWindowTitle("CSV Processing Tool")
50
51
52     def updateSelectedCycles(self):
53         # Update the list of selected cycles based on checkbox states
54         self.selected_cycles = [i+1 for i, cb in enumerate(self.checkboxes) if cb.isChecked()]
55
56     def browseFolder(self):
57         # Prompt user to select input folder
58         folder = QFileDialog.getExistingDirectory(self, "Select Input Directory")
59         if folder:
60             self.label.setText(f"Selected Input Directory: {folder}")
61             self.directory = folder
62
63     def browseOutputFolder(self):
64         # Prompt user to select output folder
65         folder = QFileDialog.getExistingDirectory(self, "Select Output Directory")
66         if folder:
67             self.output_label.setText(f"Selected Output Directory: {folder}")
68             self.output_directory = folder
69
70     def processFiles(self):
71         # Validate input and output folder selections
72         if not hasattr(self, 'directory') or not self.directory:
73             QMessageBox.warning(self, "Warning", "Please select an input directory first.")
74             return
75
76         if not self.output_directory:
77             QMessageBox.warning(self, "Warning", "Please select an output directory.")
78             return
79
80         if not self.selected_cycles:
81             QMessageBox.warning(self, "Warning", "Please select at least one cycle to average.")
82             return
83
84         # Walk through all CSV files in input directory and process them
85         for root, _, files in os.walk(self.directory):
86             for file in files:
87                 if file.endswith(".csv"):
88                     file_path = os.path.join(root, file)
89                     self.processCSV(file_path)
90
91         QMessageBox.information(self, "Success", "CSV files have been processed and saved to the output folder.")
92
93     def processCSV(self, file_path):
94         df = pd.read_csv(file_path)
95         pressure_col = "Pressure (mbar)"
96
97         # Build column names for selected cycles
98         distance_cols = [f"Distance Cycle {i} (mm)" for i in self.selected_cycles]
99         load_cols = [f"Load Cycle {i} (kg)" for i in self.selected_cycles]
100
101        # Calculate averages across selected cycles
102        df["Distance (mm)"] = df[distance_cols].mean(axis=1)
103        df["Load(kg)"] = df[load_cols].mean(axis=1)
104
105        # Keep only pressure, averaged distance and averaged load columns
106        df = df[[pressure_col, "Distance (mm)", "Load(kg)"]]
107
108        # Construct new filename with selected cycle range
109        start_cycle = min(self.selected_cycles)
110        end_cycle = max(self.selected_cycles)
111
112        new_file_name = os.path.basename(file_path).replace(".csv", f"_averaged_{start_cycle}__{end_cycle}.csv")
113        new_file_path = os.path.join(self.output_directory, new_file_name)
114        df.to_csv(new_file_path, index=False)
115
116    if __name__ == "__main__":
117        app = QApplication(sys.argv)
118        window = CSVProcessor()
119        window.show()
120        sys.exit(app.exec_())
121

```

Appendix 27-D_to_P_converter.py GUI



Appendix 28-D_to_P_converter code snippets

```

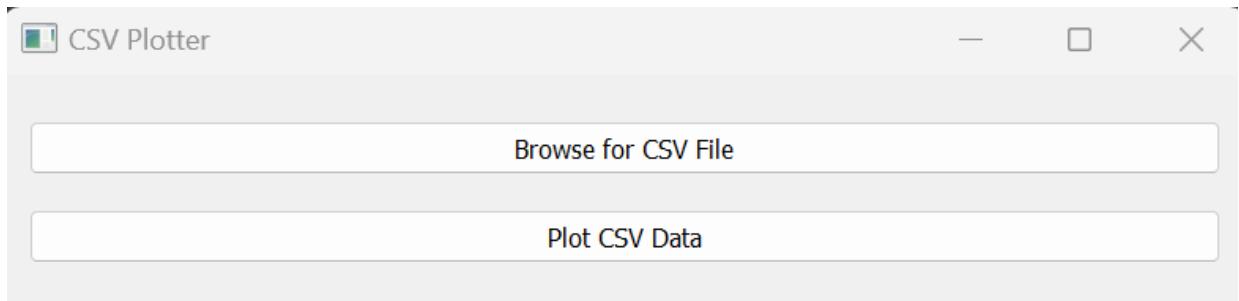
1  import os
2  import pandas as pd
3  import numpy as np
4  from PyQt5 import QtWidgets
5  from PyQt5.QtWidgets import QFileDialog, QMessageBox
6  import scipy.interpolate as interp
7
8  # Interpolation method used for generating evenly spaced pressure values
9  INTERPOLATION_TYPE = 'linear' # Change to 'cubic' if needed
10
11 class CSVProcessor:
12     def __init__(self, directory):
13         self.directory = directory
14         self.file_pairs = self.find_csv_pairs() # Find matching up/down file pairs by weight
15
16     def find_csv_pairs(self):
17         # List all CSV files in the directory
18         files = [f for f in os.listdir(self.directory) if f.endswith('.csv')]
19         weights = {}
20
21         # Group files by weight and direction (up/down)
22         for file in files:
23             parts = file.split('_')
24             try:
25                 weight = int(parts[0][-2:]) # Extract weight from filename (e.g., "13kg_up_..." → 13)
26                 direction = 'up' if 'up' in file else 'down' if 'down' in file else None
27                 if direction:
28                     weights.setdefault(weight, {})[direction] = file
29             except ValueError:
30                 continue # Skip files that don't match expected naming format
31
32         # Return only complete up/down pairs
33         return {w: p for w, p in weights.items() if 'up' in p and 'down' in p}
34
35     def process_and_save(self):
36         # Process each up/down pair
37         for weight, pair in self.file_pairs.items():
38             up_file = os.path.join(self.directory, pair['up'])
39             down_file = os.path.join(self.directory, pair['down'])
40
41             # Read CSV files
42             df_up = pd.read_csv(up_file)
43             df_down = pd.read_csv(down_file)
44
45             # Extract distance and pressure columns (assumed to be the first two columns)
46             distance_up, pressure_up = df_up.iloc[:, 1], df_up.iloc[:, 0]
47             distance_down, pressure_down = df_down.iloc[:, 1], df_down.iloc[:, 0]
```

```

48         max_distance = 200 # Maximum distance to interpolate over
49         interpolated_distances = np.arange(0, max_distance + 1, 1) # Step of 1 mm
50         print(interpolated_distances) # Debug print
51
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

```

1 import sys
2 import pandas as pd
3 import matplotlib.pyplot as plt
4 from PyQt5 import QtWidgets
5 from PyQt5.QtWidgets import QFileDialog, QMessageBox
6
7 class PlotterApp(QtWidgets.QWidget):
8     def __init__(self):
9         super().__init__()
10        self.initUI()
11
12    def initUI(self):
13        self.layout = QtWidgets.QVBoxLayout()
14
15        # Button for selecting CSV file
16        self.browse_file_btn = QtWidgets.QPushButton('Browse for CSV File')
17        self.browse_file_btn.clicked.connect(self.browse_file)
18
19        # Button for plotting the data
20        self.plot_btn = QtWidgets.QPushButton('Plot CSV Data')
21        self.plot_btn.clicked.connect(self.plot_data)
22
23        # Add buttons to the layout
24        self.layout.addWidget(self.browse_file_btn)
25        self.layout.addWidget(self.plot_btn)
26
27        # Set layout
28        self.setLayout(self.layout)
29
30        self.selected_file = None
31
32    def browse_file(self):
33        # Open file dialog to select a CSV file
34        file_name, _ = QFileDialog.getOpenFileName(self, 'Select CSV File', '', 'CSV Files (*.csv)')
35        if file_name:
36            self.selected_file = file_name
37            QMessageBox.information(self, 'File Selected', f'Selected file: {file_name}')
38
39    def plot_data(self):
40        # Check if file is selected
41        if self.selected_file:
42            try:
43                # Read CSV file (skip the first row which contains weight info)
44                df = pd.read_csv(self.selected_file, skiprows=1, header=None)
45

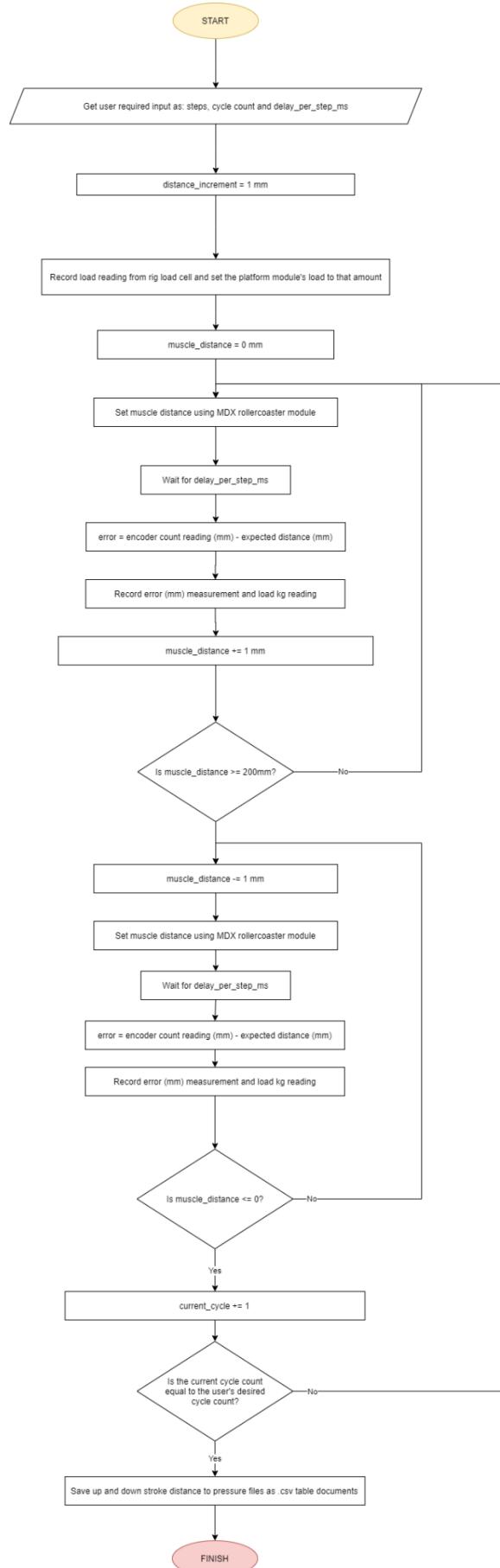
```

```

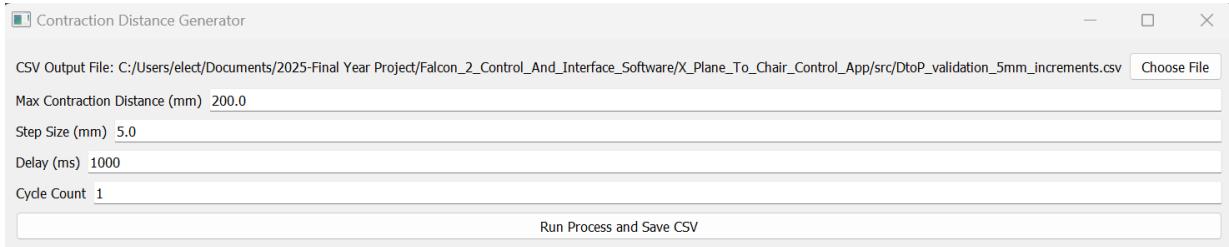
46     # Ensure we have at least two rows
47     if df.shape[0] < 2:
48         raise ValueError("CSV file doesn't contain enough data for plotting.")
49
50     # Extract the second and third rows for plotting (pressure values)
51     pressure_up = df.iloc[0, :].values # First row of pressures
52     pressure_down = df.iloc[1, :].values # Second row of pressures
53
54     print(pressure_up)
55
56     # The x-axis (distance) is just the index of the values, starting from 1
57     distance = list(range(1, len(pressure_up) + 1))
58
59     # Plot the data
60     plt.figure(figsize=(8, 6))
61
62     plt.plot(distance, pressure_up, label="Pressure Up",
63               color='b', linestyle='-', marker='o') # Line with points for Pressure Up
64
65     plt.plot(distance, pressure_down, label="Pressure Down",
66               color='r', linestyle='--', marker='x') # Line with points for Pressure Down
67
68     # Add scatter for individual points (optional, as we already added markers to the plot)
69     plt.scatter(distance, pressure_up, color='b', zorder=5)
70     plt.scatter(distance, pressure_down, color='r', zorder=5)
71
72     plt.xlabel('Distance (mm)')
73     plt.ylabel('Pressure (mbar)')
74     plt.title('Pressure vs Distance Plot')
75     plt.legend()
76     plt.grid(True)
77     plt.show()
78
79     except Exception as e:
80         QMessageBox.warning(self, 'Error', f'Failed to read and plot the CSV file.\n{str(e)}')
81     else:
82         QMessageBox.warning(self, 'Error', 'No file selected. Please select a CSV file first.')
83
84 if __name__ == '__main__':
85     app = QtWidgets.QApplication(sys.argv)
86     window = PlotterApp()
87     window.setWindowTitle("CSV Plotter")
88     window.setGeometry(100, 100, 300, 150) # Set window size and position
89     window.show()
90     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

```

1 import sys
2 import time
3 import csv
4 from platform_pose import Platform
5 from common.serialProcess import SerialProcess
6 from PyQt5.QtWidgets import (
7     QApplication, QWidget, QLabel, QLineEdit, QPushButton,
8     QVBoxLayout, QFileDialog, QHBoxLayout, QMessageBox
9 )
10
11 from d_to_p_ver_2 import DistanceToPressure
12
13 class ContractionApp(QWidget):
14     def __init__(self):
15         super().__init__()
16         self.initUI() # Initialize the user interface
17         self.output_file = "" # Output file path for saving data
18         self.platform = Platform() # Platform instance for motion control
19         self.encoder = SerialProcess() # Serial connection for encoder
20         self.d_to_p = DistanceToPressure("output\\wheelchair_DtoP.csv") # Distance-to-pressure conversion
21         self.d_to_p.set_load(24) # Set load factor for the conversion
22
23     def initUI(self):
24         """Set up the user interface components"""
25         self.setWindowTitle("Contraction Distance Generator")
26
27         layout = QVBoxLayout()
28
29         # Output file selection section
30         file_layout = QHBoxLayout()
31         self.file_label = QLabel("CSV Output File: Not selected") # Label to show selected file
32         file_btn = QPushButton("Choose File") # Button to select file
33         file_btn.clicked.connect(self.choose_file) # Connect button to choose_file function
34         file_layout.addWidget(self.file_label)
35         file_layout.addWidget(file_btn)
36         layout.addLayout(file_layout)
37
38         # Input fields with default values
39         defaults = {
40             "Max Contraction Distance (mm)": "200.0", # Default max contraction distance
41             "Step Size (mm)": "5.0", # Default step size
42             "Delay (ms)": "1000", # Default delay in milliseconds
43             "Cycle Count": "1" # Default cycle count
44         }

```

```

44     }
45
46     self.inputs = {} # Dictionary to store input fields
47     for label, default in defaults.items():
48         row = QHBoxLayout()
49         lbl = QLabel(label) # Label for input field
50         inp = QLineEdit() # Input field
51         inp.setText(default) # Set default text
52         row.addWidget(lbl)
53         row.addWidget(inp)
54         layout.addLayout(row)
55         self.inputs[label] = inp # Store input field in dictionary
56
57     # Process Button
58     run_btn = QPushButton("Run Process and Save CSV") # Button to run process
59     run_btn.clicked.connect(self.run_process) # Connect button to run_process function
60     layout.addWidget(run_btn)
61
62     self.setLayout(layout) # Set layout for the window
63
64 def choose_file(self):
65     """Open a file dialog to choose output CSV file"""
66     file_name, _ = QFileDialog.getSaveFileName(self, "Save CSV", "", "CSV Files (*.csv)") # File dialog
67     if file_name:
68         self.output_file = file_name # Set the selected file path
69         self.file_label.setText(f"CSV Output File: {file_name}") # Update label to show selected file
70
71 def GetDistanceAndLoad(self):
72     """Retrieve extension and load values from the encoder"""
73     data = self.encoder.read() # Read data from encoder
74     print(data)
75     if data is None:
76         print("function does not function!")
77         return (0.0, 0.0) # Return default values if no data
78     array = data.split(",") # Split the data into components
79     return (-float(array[1]), float(array[3]) / 9.81) # Return extension (mm) and load (kg)
80
81 def run_process(self):
82     """Run the contraction process and save the results to CSV"""
83     if not self.output_file:
84         QMessageBox.warning(self, "No File", "Please select an output CSV file.") # Warning if no file is selected
85
86
87     try:
88         # Get values from input fields and convert to appropriate types
89         max_dist = float(self.inputs["Max Contraction Distance (mm)"].text())
90         step_size = float(self.inputs["Step Size (mm)"].text())
91         delay_ms = int(self.inputs["Delay (ms)"].text())
92         cycles = int(self.inputs["Cycle Count"].text())
93     except ValueError:
94         QMessageBox.critical(self, "Invalid Input", "Please enter valid numeric values.") # Error if input is invalid
95         return
96
97     port = "COM10" # Serial port for encoder connection
98     print(port)
99     self.encoder.open_port(port, 115200) # Open serial connection to the encoder
100    time.sleep(0.1) # Wait for the queue to fill up
101    self.encoder.write("R.encode()") # Send "R" command to the encoder
102    _, load = self.GetDistanceAndLoad() # Get initial load value
103    self.LOAD = round(load) # Round the load value
104    print("THE LOAD IS ", self.LOAD)
105
106    # Build the contraction profile (one rising + falling pass)
107    distances = [] # List to store contraction distances
108    d = 0.0
109    while d <= max_dist:
110        distances.append(round(d, 4)) # Add each distance to the list
111        d += step_size
112    d = max_dist - step_size
113    while d >= 0:
114        distances.append(round(d, 4)) # Add each distance for the falling pass
115        d -= step_size
116
117    # Create data structure to hold results
118    results = {dist: {} for dist in distances}
119
120    # Run the cycles
121    for cycle in range(1, cycles + 1):
122        for dist in distances:
123            # Get pressure value based on the contraction distance
124            pressure = self.d_to_p.get_pressure(max(int(dist - 2), 0))

```

```

126     print(f"Pressure at {dist}mm is {pressure}mbar")
127
128     # Send pressure values to the platform
129     self.platform.muscle.send_pressures([pressure, 0, 0, 0, 0, 0])
130     time.sleep(delay_ms / 1000.0) # Wait for the delay
131
132     print("Distance: ", dist, " Cycle: ", cycle)
133
134     # Measure the distance and load after each step
135     measured_dist, load = self.GetDistanceAndLoad()
136
137     print(measured_dist, " ", dist)
138
139     # Store the results (error and load) for each distance and cycle
140     results[dist][f"error mm cycle {cycle}"] = measured_dist - dist
141     results[dist][f"load kg cycle {cycle}"] = load
142
143     # Build headers for the CSV
144     headers = ["contraction_distance mm"]
145     for cycle in range(1, cycles + 1):
146         headers.append(f"error mm cycle {cycle}")
147         headers.append(f"load kg cycle {cycle}")
148
149     # Write the results to the CSV file
150     with open(self.output_file, mode='w', newline='') as file:
151         writer = csv.writer(file)
152         writer.writerow(headers) # Write headers
153         for dist in distances:
154             row = [dist]
155             for cycle in range(1, cycles + 1):
156                 row.append(results[dist].get(f"error mm cycle {cycle}", ""))
157                 row.append(results[dist].get(f"load kg cycle {cycle}", ""))
158             writer.writerow(row) # Write row to CSV
159
160     # Show a message box to indicate the process is complete
161     QMessageBox.information(self, "Done", f"Data saved to {self.output_file}")
162
163 if __name__ == '__main__':
164     app = QApplication(sys.argv)
165     window = ContractionApp() # Create an instance of the app
166     window.show() # Show the window
167     sys.exit(app.exec_()) # Start the Qt application loop
168

```

Appendix 34-d_to_p_ver_2.py code snippets

```

1 import numpy as np
2 import pandas as pd
3
4 class DistanceToPressure:
5     def __init__(self, csv_path):
6         # Load the CSV file using pandas
7         self.data = pd.read_csv(csv_path)
8
9         rows = self.data.values.tolist() # Convert the data into a list of rows
10
11        # Ensure I am getting the row data (print all values in the second column for verification)
12        for i in rows:
13            print(i[1])
14
15        # Store the rows corresponding to weight-direction pairs in self.weight_direction_rows
16        self.weight_direction_rows = rows
17        # Define a list of known weights
18        self.weights = [13, 24, 33, 43]
19        self.is_going_up = True # Initial direction of movement (default: True, going up)
20        self.last_distance = 0 # Track the last distance
21
22    def set_load(self, load):
23        """Set the load and find the closest weights to interpolate between."""
24        # If the load is not in the predefined weights, calculate the closest lower and upper weights
25        if load not in self.weights:
26            lower_weight = max([w for w in self.weights if w < load], default=None)
27            upper_weight = min([w for w in self.weights if w > load], default=None)
28
29            # If no valid lower or upper weight, raise an error
30            if lower_weight is None and upper_weight is None:
31                raise ValueError(f"No weights available for load {load}.")
32            # If only the lower weight is available, use it and set interpolation factor to 0
33            elif lower_weight is None:
34                self.load = (upper_weight, upper_weight)
35                self.interpolation_factor = 0

```

```

36     # If only the upper weight is available, use it and set interpolation factor to 0
37     elif upper_weight is None:
38         self.load = (lower_weight, lower_weight)
39         self.interpolation_factor = 0
40     # Otherwise, use both lower and upper weights for interpolation
41     else:
42         self.load = (lower_weight, upper_weight)
43         self.interpolation_factor = (load - lower_weight) / (upper_weight - lower_weight)
44     else:
45         # If the load matches one of the predefined weights, no interpolation is needed
46         self.load = (load, load)
47         self.interpolation_factor = 0
48
49     def get_pressure(self, distance):
50         """
51             Calculate the pressure based on the given distance.
52             The pressure is calculated by interpolating between pressure values corresponding
53             to the lower and upper weights at the specified distance.
54         """
55         # NOTE FOR FUTURE ME: Change curves to be set only once when setting load for better efficiency!
56
57         lower_weight, upper_weight = self.load
58
59         # Find the index of the lower and upper weights in the list of weights
60         lower_row_index = self.weights.index(lower_weight)
61         upper_row_index = self.weights.index(upper_weight)
62
63         lower_weight_rows = []
64         upper_weight_rows = []
65
66         # Find the rows corresponding to the lower and upper weights (for both directions)
67         lower_weight_rows.append(self.weight_direction_rows[lower_row_index])
68         lower_weight_rows.append(self.weight_direction_rows[lower_row_index + 4])
69
70         upper_weight_rows.append(self.weight_direction_rows[upper_row_index])
71         upper_weight_rows.append(self.weight_direction_rows[upper_row_index + 4])
72
73         # Print values from the "up" and "down" curves to check if correct rows are selected
74         print(upper_weight_rows[0][4], upper_weight_rows[1][4])
75
76     # Determine the direction of movement based on whether the distance has increased or decreased
77     self.is_going_up = distance - self.last_distance > 0
78
79     # Initialize the pressure variable
80     lerp_pressure = -1
81
82     # Interpolate the pressure based on the direction (up or down)
83     if self.is_going_up:
84         # Interpolate for the "up" direction using the rows for the lower and upper weights
85         lerp_pressure = lower_weight_rows[0][distance] + (self.interpolation_factor * (upper_weight_rows[0][distance] - lower_weight_rows[0][distance]))
86     else:
87         # Interpolate for the "down" direction using the rows for the lower and upper weights
88         lerp_pressure = lower_weight_rows[1][distance] + (self.interpolation_factor * (upper_weight_rows[1][distance] - lower_weight_rows[1][distance]))
89
90     # Update the last distance for future calculations
91     self.last_distance = distance
92
93     # Return the pressure, rounded to the nearest integer
94     return round(lerp_pressure)
95
96     # Example usage:
97     # Create an instance of the DistanceToPressure class by providing the path to your CSV file
98     #muscle = DistanceToPressure("output\\wheelchair_DtoP.csv")
99
100    # Set the load (for example, 40kg)
101    #muscle.set_load(24)
102
103    # Retrieve the pressure for a given distance (e.g., distance = 10, up direction)
104    #try:
105    #    for i in range(0, 10):
106    #        pressure = muscle.get_pressure(i)
107    #        print(f"Pressure at {i}mm distance (up direction): {pressure} mbar")
108    #
109    #    for i in range(10, 0, -1):
110    #        pressure = muscle.get_pressure(i)
111    #        print(f"Pressure at {i}mm distance (down direction): {pressure} mbar")
112    #
113    #except IndexError as e:
114    #    print(f"Error: {e}")
115

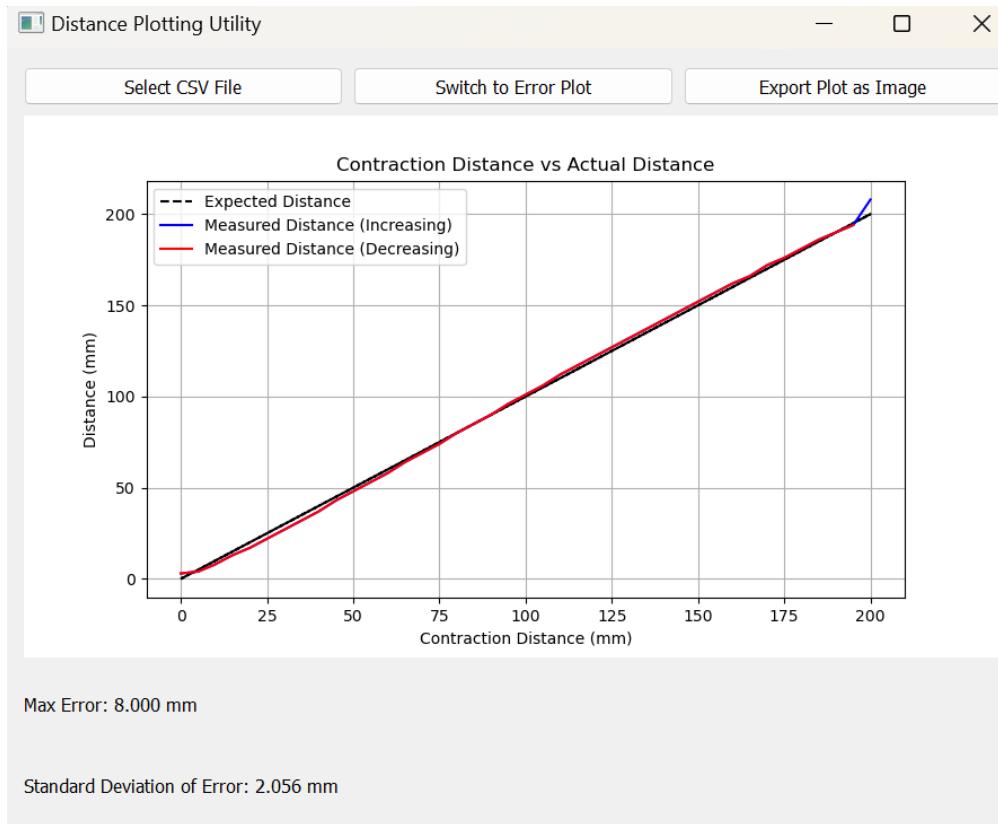
```

```

76     # Determine the direction of movement based on whether the distance has increased or decreased
77     self.is_going_up = distance - self.last_distance > 0
78
79     # Initialize the pressure variable
80     lerp_pressure = -1
81
82     # Interpolate the pressure based on the direction (up or down)
83     if self.is_going_up:
84         # Interpolate for the "up" direction using the rows for the lower and upper weights
85         lerp_pressure = lower_weight_rows[0][distance] + (self.interpolation_factor * (upper_weight_rows[0][distance]
86                                         - lower_weight_rows[0][distance]))
87     else:
88         # Interpolate for the "down" direction using the rows for the lower and upper weights
89         lerp_pressure = lower_weight_rows[1][distance] + (self.interpolation_factor * (upper_weight_rows[1][distance]
90                                         - lower_weight_rows[1][distance]))
91
92     # Update the last distance for future calculations
93     self.last_distance = distance
94
95     # Return the pressure, rounded to the nearest integer
96     return round(lerp_pressure)
97
98 # Example usage:
99 # Create an instance of the DistanceToPressure class by providing the path to your CSV file
100 #muscle = DistanceToPressure("output\\wheelchair_DtoP.csv")
101
102 # Set the load (for example, 40kg)
103 #muscle.set_load(24)
104
105 # Retrieve the pressure for a given distance (e.g., distance = 10, up direction)
106 #try:
107 #    for i in range(0, 10):
108 #        pressure = muscle.get_pressure(i)
109 #        print(f"Pressure at {i}mm distance (up direction): {pressure} mbar")
110 #
111 #    for i in range(10, 0, -1):
112 #        pressure = muscle.get_pressure(i)
113 #        print(f"Pressure at {i}mm distance (down direction): {pressure} mbar")
114 #
115 #except IndexError as e:
116 #    print(f"Error: {e}")

```

Appendix 35-DtoP_Validation_Plotter.py GUI snippet



Appendix 36-DtoP_Validation_Plotter.py code snippets

```

1  import sys
2  import pandas as pd
3  from PyQt5.QtWidgets import (
4      QApplication, QWidget, QVBoxLayout, QPushButton, QFileDialog,
5      QLabel, QHBoxLayout
6  )
7  from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as FigureCanvas
8  from matplotlib.figure import Figure
9
10
11 class PlotWindow(QWidget):
12     def __init__(self):
13         super().__init__()
14         self.setWindowTitle("Distance Plotting Utility") # Set window title
15         self.setGeometry(100, 100, 900, 700) # Set window size and position
16
17         self.layout = QVBoxLayout() # Main vertical layout for the window
18         self.setLayout(self.layout)
19
20         button_layout = QHBoxLayout() # Layout for buttons
21
22         # Button for selecting CSV file
23         self.load_button = QPushButton("Select CSV File")
24         self.load_button.clicked.connect(self.load_csv) # Connect button to load_csv method
25         button_layout.addWidget(self.load_button)
26
27         # Button for switching between distance and error plots
28         self.toggle_button = QPushButton("Switch to Error Plot")
29         self.toggle_button.clicked.connect(self.toggle_plot_mode) # Connect button to toggle_plot_mode method
30         self.toggle_button.setEnabled(False) # Initially disabled
31         button_layout.addWidget(self.toggle_button)
32
33         # Button for exporting the plot as an image
34         self.export_button = QPushButton("Export Plot as Image")
35         self.export_button.clicked.connect(self.export_plot) # Connect button to export_plot method
36         self.export_button.setEnabled(False) # Initially disabled
37         button_layout.addWidget(self.export_button)
38
39         self.layout.addLayout(button_layout) # Add button layout to main layout
40
41         # Set up the plot canvas
42         self.canvas = FigureCanvas(Figure()) # Create an empty figure canvas
43         self.ax = self.canvas.figure.add_subplot(111) # Add subplot to figure

```

```

44     self.layout.addWidget(self.canvas) # Add the canvas to the layout
45
46     # Labels for displaying the max error and standard deviation
47     self.error_label = QLabel("Max Error: N/A")
48     self.std_label = QLabel("Standard Deviation of Error: N/A")
49     self.layout.addWidget(self.error_label)
50     self.layout.addWidget(self.std_label)
51
52     # Initialize data attributes
53     self.df = None # Dataframe to hold CSV data
54     self.max_error = None # Max error value
55     self.std_dev = None # Standard deviation of error
56     self.plot_mode = 'distance' # Default plot mode is 'distance'
57
58 def load_csv(self):
59     """Load the selected CSV file and prepare data for plotting"""
60     filepath, _ = QFileDialog.getOpenFileName(self, "Open CSV File", "", "CSV Files (*.csv)") # Open file dialog
61     if not filepath:
62         return # If no file is selected, return
63
64     df = pd.read_csv(filepath) # Read CSV into dataframe
65     df.columns = [col.strip() for col in df.columns] # Clean column names (remove any extra spaces)
66
67     # Add a new column for the actual measured distance
68     df['actual_distance'] = df['contraction_distance mm'] + df['error mm cycle 1']
69
70     # Store the dataframe and calculate error statistics
71     self.df = df
72     self.max_error = df['error mm cycle 1'].abs().max() # Calculate max error
73     self.std_dev = df['error mm cycle 1'].std() # Calculate standard deviation of error
74
75     # Update the labels with error statistics
76     self.error_label.setText(f"Max Error: {self.max_error:.3f} mm")
77     self.std_label.setText(f"Standard Deviation of Error: {self.std_dev:.3f} mm")
78
79     # Enable buttons after CSV is loaded
80     self.export_button.setEnabled(True)
81     self.toggle_button.setEnabled(True)
82     self.plot_mode = 'distance' # Default to distance plot
83     self.toggle_button.setText("Switch to Error Plot") # Update button text
84
85     self.plot_distance() # Plot the distance graph initially
86
87 def toggle_plot_mode(self):
88     """Switch between distance plot and error plot"""
89     if self.plot_mode == 'distance':
90         self.plot_mode = 'error' # Switch to error plot mode
91         self.toggle_button.setText("Switch to Distance Plot") # Update button text
92         self.plot_error() # Plot the error graph
93     else:
94         self.plot_mode = 'distance' # Switch back to distance plot mode
95         self.toggle_button.setText("Switch to Error Plot") # Update button text
96         self.plot_distance() # Plot the distance graph
97
98 def plot_distance(self):
99     """Plot the contraction distance vs actual distance graph"""
100    if self.df is None:
101        return # If no data is available, return
102
103    df = self.df
104    max_idx = df['contraction_distance mm'].idxmax() # Find index of max contraction distance
105
106    self.ax.clear() # Clear the current plot
107    self.canvas.figure.suptitle("Contraction vs Actual Distance", fontsize=18, fontweight='bold') # Set plot title
108
109    # Plot expected contraction distance (dashed line)
110    self.ax.plot(df['contraction_distance mm'],
111                 df['contraction_distance mm'],
112                 linestyle='--', color='black', label='Expected Distance')
113
114    # Plot measured distance (increasing)
115    self.ax.plot(df['contraction_distance mm'][max_idx + 1],
116                 df['actual_distance'][max_idx + 1],
117                 color='blue', label='Measured Distance (Increasing)')
118
119    # Plot measured distance (decreasing)
120    self.ax.plot(df['contraction_distance mm'][max_idx + 1:],
121                 df['actual_distance'][max_idx + 1:],
122                 color='red', label='Measured Distance (Decreasing)')
123
124    # Set labels and legend
125    self.ax.set_title("Contraction Distance vs Actual Distance", fontsize=12)

```

```

126     self.ax.set_xlabel("Contraction Distance (mm)")
127     self.ax.set_ylabel("Distance (mm)")
128     self.ax.legend()
129     self.ax.grid(True) # Add grid for better readability
130     self.canvas.draw() # Redraw the canvas
131
132 def plot_error(self):
133     """Plot the error vs expected contraction distance graph"""
134     if self.df is None:
135         return # If no data is available, return
136
137     df = self.df
138     max_idx = df['contraction_distance mm'].idxmax() # Find index of max contraction distance
139
140     self.ax.clear() # Clear the current plot
141     self.canvas.figure.suptitle("Error vs Expected Contraction Distance", fontsize=18, fontweight='bold') # Set plot title
142
143     # Plot error for increasing distance
144     self.ax.plot(df['contraction_distance mm'][:max_idx + 1],
145                  df['error mm cycle 1'][:max_idx + 1],
146                  color='blue', label='Error (Increasing)')
147
148     # Plot error for decreasing distance
149     self.ax.plot(df['contraction_distance mm'][max_idx + 1:],
150                  df['error mm cycle 1'][max_idx + 1:],
151                  color='red', label='Error (Decreasing)')
152
153     self.ax.axhline(0, linestyle='--', color='gray') # Add horizontal line at y=0 for reference
154     self.ax.set_title("Error vs Contraction Distance", fontsize=12)
155     self.ax.set_xlabel("Contraction Distance (mm)")
156     self.ax.set_ylabel("Error (mm)")
157     self.ax.legend()
158     self.ax.grid(True) # Add grid for better readability
159     self.canvas.draw() # Redraw the canvas
160
161 def export_plot(self):
162     """Export the current plot as an image"""
163     if self.df is None:
164         return # If no data is available, return
165
166     filepath, _ = QFileDialog.getSaveFileName(self, "Save Plot As Image", "", "PNG Files (*.png)") # Open save dialog
167     if not filepath:
168         return # If no file path is selected, return
169
170     fig = Figure(figsize=(8, 6)) # Create a new figure
171     fig.subplots_adjust(top=0.85) # Make space for big title
172     ax = fig.add_subplot(111) # Add subplot to the figure
173     df = self.df
174     max_idx = df['contraction_distance mm'].idxmax() # Find index of max contraction distance
175
176     # Plot either the distance or error graph based on current mode
177     if self.plot_mode == 'distance':
178         fig.suptitle("Contraction vs Actual Distance", fontsize=18, fontweight='bold')
179
180         ax.plot(df['contraction_distance mm'],
181                 df['contraction_distance mm'],
182                 linestyle='--', color='black', label='Expected Distance')
183
184         ax.plot(df['contraction_distance mm'][:max_idx + 1],
185                 df['actual_distance'][:max_idx + 1],
186                 color='blue', label='Measured Distance (Increasing)')
187
188         ax.plot(df['contraction_distance mm'][max_idx + 1:],
189                 df['actual_distance'][max_idx + 1:],
190                 color='red', label='Measured Distance (Decreasing)')
191
192         ax.set_title("Contraction Distance vs Actual Distance", fontsize=12)
193         ax.set_xlabel("Contraction Distance (mm)")
194         ax.set_ylabel("Distance (mm)")
195     else:
196         fig.suptitle("Error vs Expected Contraction Distance", fontsize=18, fontweight='bold')
197
198         ax.plot(df['contraction_distance mm'][:max_idx + 1],
199                 df['error mm cycle 1'][:max_idx + 1],
200                 color='blue', label='Error (Increasing)')
201
202         ax.plot(df['contraction_distance mm'][max_idx + 1:],
203                 df['error mm cycle 1'][max_idx + 1:],
204                 color='red', label='Error (Decreasing)')
205
206         ax.axhline(0, linestyle='--', color='gray')
207         ax.set_title("Error vs Contraction Distance", fontsize=12)

```

```
208     ax.set_xlabel("Contraction Distance (mm)")
209     ax.set_ylabel("Error (mm)")
210
211     ax.legend() # Add legend
212     ax.grid(True) # Add grid
213
214     # Add error statistics to the plot
215     stat_text = f"Max Error: {self.max_error:.3f} mm\nStd Dev: {self.std_dev:.3f} mm"
216     ax.text(0.95, 0.05, stat_text, transform=ax.transAxes,
217             fontsize=10, verticalalignment='bottom', horizontalalignment='right',
218             bbox=dict(facecolor='white', alpha=0.7, edgecolor='gray'))
219
220     fig.savefig(filepath, dpi=300, bbox_inches='tight') # Save plot to the selected file path
221     print(f"Plot exported to: {filepath}")
222
223
224 if __name__ == "__main__":
225     app = QApplication(sys.argv)
226     window = PlotWindow()
227     window.show() # Show the plot window
228     sys.exit(app.exec_()) # Execute the application
229
```

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