Title: Development of Control and Interface Systems for a Flight Simulation Experience

# Abstract

This report covers the design and development process of a control and interface system for a 'Flight Simulation' ride experience. The project is being carried out in collaboration with other final-year students responsible for the ride’s accessibility features, as well as the Royal Aeronautical Society, which plans to use the completed ride at outreach events such as school visits and expositions.

The specific programme this project is a part of is the “Falcon 2 programme” [1], a STEM outreach programme for schools, colleges and youth groups aged (6-19) to contribute their design and engineering skills to develop and build a real-life mobile flight simulator which will travel to Special Educational Needs and Disability (SEND) schools and public events around the UK to introduce people from all backgrounds to the wonder of flight.

The final ride is designed to be a highly immersive experience, allowing the user to feel as though they are truly piloting an airplane. The chair moves in sync with the plane’s motions, and the user can observe their surroundings through a wide monitor (or VR headset ideally), interacting with them as if they were inside the airplane.

# Introduction

First and foremost, some research was done into existing solutions . A high-level outline of the ride control system is shown in **Fig 0.0**.

A diagram of a computer system

Description automatically generated

**Fig 0.0: High level Flowchart showing how different components within the ride communicate with each other.**

The core goal of this project is to develop a comprehensive set of ready-to-use software tailored for the RaeS Falcon 2 programme that enables the virtual flight simulator (or potentially a variety of simulators) to interface seamlessly with the physical system.

Additionally, the project aims to provide direct human operator manual chair control through a peripheral device. Furthermore, the software will prioritize operational safety by incorporating extensive failsafe mechanisms throughout the system such as stopping the simulation if the wheelchair dock sensor is released or if the chair muscle attachment is released. As far as this project is concerned, the sensors are just incoming signals. Details on actual hardware won’t be delved on. During the development these sensors will be referred to as “safety sensors” which can have either a “safe” or “un-safe” state that’ll be incorporated into the logic of the software.

A significant focus will also be placed on quality-of-life features and aiming to create a fully complete system, ensuring it is user-friendly and accessible to individuals who may not be familiar with the system's inner workings.

# Week 1-Preliminary research

According to some pre-liminary research, the main simulator of choice for the project is “X-Plane”.

The client in charge of getting the telemetry of the plane from X-Plane (or Unity) is written in python and the communication protocols between software/hardware are done via UDP protocol. This includes the communication from X-Plane to the python client to send the plane telemetry as well as the communication from the python program to the PLC to control the mechanical chair PLC.

There’s some pre-existing work by MDX on a similar motion platform which can be adapted into the mechanical chair. The project is open source and can be found on Git Hub [2]. It’s worth noting this project inherits several parts from the MDX rollercoaster ride [3].

From some pre-liminary research, I also found out that the chair can either be “parked” or “un-parked”. When “parked”, the chair is flat against the ground and the user can either gen on or off the chair. When “un-parked”, the chair is ready to move around and simulate the ride experience.

So far the library of choice to interface with X-Plane is XPPython [4] which enables access to airplane telemetry and other data from the game. The project will later look into controlling the game through python to start the simulation but for now the main focus is getting data from X-Plane through the XPPython API.

The python client is in charge of several jobs, the most important of which is to translate the xyzrpy (xyz translation, roll, pitch and yaw) into actuator distances for the platform (inverse kinematics). The actuators in the chair are fluidic muscles, therefore for a given weight, the client sends out a given pressure value via UDP to actuate each one by the desired distance.

**Fig 1.0** shows a more specific software layout of the system.

A diagram of a flowchart

Description automatically generated

**Fig 1.0: High level Flowchart showing a more specific software layout of the entire ride system**

The Unity User Interface is designed to serve as an interface for the operator, enabling easy control of the ride through a user-friendly system. This is achieved by establishing a connection between the Unity UI and the Python controller client, where the Unity UI primarily displays system information and relays control commands to the Python controller, which acts as the core control software for the chair.

The main reason for selecting Unity is its robust set of UI tools, which allow developers to create interfaces efficiently without the need to code every individual element. Another advantage of using Unity is its inherent capability to handle 3D simulations. This not only facilitates the creation of a well-rounded and intuitive UI but also enables the real-time display of a 3D digital twin of the chair. This feature allows the operator to debug and manually control the chair as needed, further enhancing the system's usability and functionality.

## Requirements

The following list shows what the final project software must be capable of achieving in conjunction with the actual chair system. This is shown in **Table 1.0**. Requirements with \* are “wish” requirements which are prioritized below the ones without an \*.

**Table 1.0: Table showing final project requirements**

|  |  |  |
| --- | --- | --- |
| Requirement number | Requirement Description | Is wish requirement? (\*) |
| 1 | Software is able to move mechanical chair in real time with the flight simulation. The ride should not cause motion sickness after at least 2 repeated uses. |  |
| 2 | Software comes with a basic user interface. |  |
| 3 | Software provides fail START/PAUSE/STOP functionality to control the ride simulation as required. |  |
| 4 | User interface is friendly and can be easily used by an inexperience operator. | \* |
| 5 | User interfaces allows operator to manually control the chair’s pose and see a side by side simulation of it. | \* |
| 6 | Software is allows support to connect chair to other flight sims aside from X-Plane. | \* |
| 7 | Final ride has VR support | \* |

It is important to note that the latency of the mechanical chair must be kept below 200 ms, as motion lag becomes noticeable to humans beyond this threshold. Latency exceeding 200 ms can lead to motion sickness, which is highly undesirable for a platform intended to attract users and immerse them in the experience.

These are the initial requirements. The project is likely to grow and become more detailed as the dev weeks go by and further requirements will likely be introduced as more and more of the project is cleared.

# Week 1-High Level Design

Once enough pre-liminary research was completed, the next stage was to design the high level outline of how the software is to function. **Flowchart 2.0** outlines the high level layer architecture design of how the python client is to interact with the simulator and PLC controlling the chair. The core aim of this chapter will be to finalize a basic skeletal system for the software to later evolve from.

A diagram of a software company

Description automatically generated

**Flowchart 2.0: High level outline of how python client is to interact with the simulator and chair PLC**

The scripts in the “TELEMETRY LAYER” will solely be in charge of obtaining the desired airplane telemetry data from their respective simulator. This modular approach leaves room in the future to work with other simulators by simply creating an appropriate “\_Getter.py” script. The controller layer code does not need to be changed in that case.

In this design, “Chair\_Controller.py” will handle most of the workload. This node (running script) will be in charge of receiving the airplane telemetry data in the aforementioned format (Chapter 1) and translate it into distance and subsequently pressure values to send to the motion platform. **Flowchart 2.1** further outlines how the Chair\_Controller.py script will operate.

A diagram of a control system

Description automatically generated

**Flowchart 2.1: State machine diagram of how Chair\_Controller.py will operate**

A state machine design approach was chosen to clearly define the actual operation of the script. Moreover a state machine description will allow to easily create a basic python GUI later on. The definition of each state and what exactly happens in is explained below:

**IDLE\_STATE:**

During this state the node does nothing. It’ll simply wait until it gets airplane telemetry. In the meantime, parameters like weight and actuator intensity can be set by the operator through the GUI. The operator can also choose to manually control the chair for any debugging purposes. The chair is in the “parked” state during this state.

**READY\_STATE:**

Once the node is getting airplane telemetry data. The GUI will notify the user about this and then the operator is allowed to press the start button to “unpark” the chair (raising it) and begin making it move according to the telemetry it receives from the flight sim. In other words this state allows user to go to RUNNING\_STATE once airplane telemetry is available and all safety sensors have the correct readings. The chair is in the “un-parked” state during this state.

**MANUAL\_CONTROL\_STATE:**

During this state the operator can move the chair through the GUI to debug any issues with the chair itself. The operator is able to manually adjust its roll, pitch, yaw and xyz translation through a separate GUI screen. The operator can at any time go back to either the IDLE or READY state from this state.

**RUNNING STATE:**

As the name implies, during this state, the node send the desired pressure values for each actuator in the chair via UDP to the chair’s PLC. During this state, the airplane in the sim replicates the simulation’s cockpit movement and will the be main part of the ride

**STOP STATE:**

During this state, the chair is parked back down to allow the user to get on or off the ride.

**START/ PAUSE/RESET SIMULATION:**

These are transition states which are abstract the moment. Since it’s not yet clear whether controlling X-Plane with a python API is possible, these are placeholder states which label what should happen when going from one state from another. START SIMULATION is meant to begin the simulation by first “un-parking” the chair and lifting it so it can move according to the simulation. Whether the node un-pauses X-Plane or simply allows the chair to go to RUNNING STATE is unclear at the moment.

PAUSE SIMULATION is meant to pause X-Plane as well as shutting off the actuators in place in case a sensor. Once the operator manually un-pauses the simulation, the node goes to the STOP state whereby the chair is “parked” and the user can either get off it or go back to the simulation. Node only goes back to RESET SIMULATION ONCE all safety sensor readings are set back to their “safe state”.

RESET\_SIMULATION for now is meant to reset the X-Plane simulation if the plane has been crashed, the user is back from STOP\_STATE or a new user is being put into the simulator chair. Because it’s unclear at the moment whether directly controlling X-Plane is feasible, this is a placeholder state and it’s function is as important now as getting a skeletal system working.

# Week 2-Further Work On High Level Design And Code Stubs

On week 2’s meetup I further worked with my supervisor on the project. The key advances/takeaways from our Monday meetup were:

## Requirements touch-up:

Re-wording requirements/project goals to specifically state that the project is NOT mission critical. This means that although my project aims to fit the RaeS requirements, it’s success criteria is NOT that it must be used commercially by the RaeS.

In summary, my project aims to fit the project requirements of the RAeS but it does not aim to be commercially used over time.

The main reason for this stipulation is because testing and ensuring a project is suitable for repeated commercial use would take several weeks which I cannot afford due to the short development span.

## Further high-level code design:

This involves further elaborating on the state machine logic laid out in **Flowchart 2.1**. The main focus is the RUNNNING\_STATE logic and the rest will follow. Careful planning in this project in particular is critical due to having to collaborate with other groups such as Coventry University and Students working on different parts of the project. This complex and agile project nature in addition not many parts being decided yet makes the “code now and incrementally inch towards the goal” plan I’ve had so far for projects extremely unwise since this can lead tons of wasted time in debugging, remaking existing functionality and unexpected changes in plan.

Flowcharting the code structure first and then stubbing the entire software without using X-Plane yet has several benefits including:

-Better adaptability to unexpected changes

-Easier time debugging code and following a known structure

-Saving time by choosing where in the plan to re-use pre-written software components

-Makes best use of time I currently have which is meant for planning (for around the first 3 weeks)

**Flowchart 2.1** was reworked into **Flowchart 3.0** which better defines the high-level operation of the software by defining where the chair is park and correcting minor logical mistakes.

A diagram of a control system

Description automatically generated

**Flowchart 3.0: Revised version of Chair\_Controller.py script**

## RUNNING\_STATE logic

**Flowchart 3.1** showcases the logic behind RUNNING\_STATE. It’s core duty is to convert xyzrpy data from X-Plane and translate it to actuator pressure data so that the real life mechanical chair can achieve the pose within the simulator.

A screenshot of a diagram

Description automatically generated

**Flowchart 3.1: Flowchart detailing logic behind RUNNING\_STATE**

Aside from defining the logic behind all the states, a key goal will also be to identify how much functionality from the MDX rollercoaster repo [3] or other libraries can be repurposed into the smaller modules of each state. For example, there is a kinematics.py library within the repo [3] that could be reused to obtain the desired lengths and pressures of each actuator from an airplane pose.

## IDLE\_STATE logic

**Flowchart 3.2** showcases the logic behind the idle state. This is simply a screen which waits until all the safety sensors are in the correct state and once it they do so, the user is notified that the sensors are in the correct state and is given the option to press the “un-park” button to enter the READY\_STATE.

A screenshot of a diagram

Description automatically generated

**Flowchart 3.2: Flowchart outlining IDLE\_STATE logic**

## READY\_STATE logic

READY\_STATE logic is shown in **Flowchart 3.3**. In this state, the software waits until the user either:

-Enters MANUAL\_CONTROL\_STATE

-Unparks the chair and subsequently goes back to IDLE\_STATE

-Goes to RUNNING\_STATE once airplane UDP telemetry is available and entered user weight is in valid range

A diagram of a company

Description automatically generated

**Flowchart 3.3: Flowchart outlining READY\_STATE logic**

## MANUAL\_CONTROL\_STATE logic

MANUAL\_CONTROL\_STATE is a state in which the operator is allowed to manually control the mechanical chair’s pose using manual control sliders. At this stage, it’s simply be a GUI screen with sliders to control the chair’s pose. Pose includes xyz translation and rotation. This is done in order to give the operator some means to debug or diagnose the chair without getting onto it or having to tediously start a simulation.

MANUAL\_CONTROL\_STATE logic is outlined in **Flowchart 3.4**.

A screenshot of a flowchart

Description automatically generated

**Flowchart 3.4: Flowchart outlining MANUAL\_CONTROL\_STATE logic**

## STOP\_STATE logic

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. When transition to IDLE\_STATE happens, chair is put back on the parked position.

A diagram of a work flow

Description automatically generated

**Flowchart 3.5: Flowchart outlining STOP\_STATE logic**

# Week 3 – Python Code Stubs and Initial mock GUI write-up

I spent this week on creating the initial code stubs that would serve as the foundation for the control software. I divided the whole software into nodes that are interconnected to each other via UDP ports. For clarity’s sake, a node in this blog is referred to a running script in a separate terminal which runs independently of other nodes. The node network is shown in **Fig 3.0**. The main choice of library for GUI design was PyQt5. Qt Designer [6] was used to make the GUI using it’s convenient widget drag and drop functionality which allows for faster and more robust GUI class generation.

A diagram of a diagram

AI-generated content may be incorrect.

**Fig 3.0: Figure outlining communication ports between all nodes in my software**

**Table 3.0** and **Table 3.1** explain the function of each node and port within the network respectively

**Table 3.0: Table outlining function of each node in software network**

|  |  |
| --- | --- |
| Node name | Function |
| dummy\_airplane\_app.py | This is dummy node which acts as a placeholder for the flight sim and chair safety sensors. Operator can modify the airplane’s 3 axis acceleration (or position) and 3 axis rotation through a set of sliders. The operator can also set the chair’s safe state through a Boolean checkbox.  The data is sent as an array via “TELEMETRY\_UDP\_PORT” where the array is [xaccel, yaccel, zaccel roll, pitch, yaw, isSafe]. Each pose value is shown as a slider and the safety Boolean is shown as a checkbox. As previously stated accel values only affect the chair’s position (that’s why it’s labelled as xpos, ypos and zpos in GUI). |
| state\_machine.py | This script implements the state machine described in **Week 2**, **Flowchart 2.0**.  This state machine executes at a variable cycle rate set to 20ms at the moment  This machine can be fully controlled by sending commands through “CMD\_UDP\_PORT” hence it does not need to handle any of the GUI. Each state the machine can be in (IDLE, READY, RUNNING, MANUAL, and STOP).  At the moment, the UDP command format is [“state\_transition”, gain\_xaccel, gain\_yaccel, gain\_zaccel, gain\_roll, gain\_pitch, gain\_yaw]. This might change in the future.  This machine also outputs it’s 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. Again, message format may change in the future.  Finally, this state machine node listens to the airplane’s telemetry from the dummy\_airplane\_app.py node and uses it as described in past week’s flowcharts |
| platform\_control\_app.py | This app in in charge of reading the state machine’s status and allowing the user to send control commands to state\_machine.py node through a GUI. The STATUS\_UDP\_PORT is used to display the machine’s status and indicate whether the operator can do certain actions (as shown later in blog). The MANUAL\_POSE\_UDP\_PORT is used to send the manual pose values from their respective sliders to state\_machine.py. |
| stewart\_display.py | This is a display script from the “Stewart\_Py” library [5]. Since the chair is an example of an inverted Stewart platform, this script provides an effective way to visualize the chair’s motion while I do not have access to its exact display code.  Furthermore, interacting with a digital twin of the chair allows me to save time in implementing the inverse kinematics and make overall project progress while the chair is still under construction. Additionally, many aspects of the project are TBD (To Be Decided) due to its agile nature. |

**Table 3.1: Table outlining function of each network UDP port**

|  |  |  |
| --- | --- | --- |
| Node name | Port | Function |
| 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 send 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]. |

[6] and this [link](https://www.youtube.com/watch?v=QdajpwCDKjQ) contain a video showcasing all the week 3 summarized progress. The video shows each mentioned node including how to get them running and working as intended.

The following sections will contain screen snippets of each node show in **Fig 3.0**. These are annotated code and GUI (where applicable) snippets.

## dummy\_airplane\_app.py snippets

### Code:

A screenshot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.



### GUI:

A screenshot of a computer

AI-generated content may be incorrect.

## state\_machine.py snippets

A screenshot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A computer screen with text

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer

AI-generated content may be incorrect.

A computer screen with text on it

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

### GUI:

state\_machine.py has no GUI. Here are some state transitions as the software was operated.

A screen shot of a computer program

AI-generated content may be incorrect.

## platform\_control\_app.py snippets

NOTE: This node’s name was changed to main.py. It resides within X\_Plane\_To\_Chair\_Control\_App\src\ of Falcon\_2\_Control\_And\_Interface\_Software repo.

### Code:

A black screen with white text

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A computer code on a black background

AI-generated content may be incorrect.

A computer screen shot of text

AI-generated content may be incorrect.

A screen shot of a computer screen

AI-generated content may be incorrect.

A computer code on a black background

AI-generated content may be incorrect.

A black screen with green text

AI-generated content may be incorrect.

A black screen with green and white text

AI-generated content may be incorrect.

A black screen with text

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

A black screen with green and white specks

AI-generated content may be incorrect.

### GUI:

Simulation Tab:

A screenshot of a computer

AI-generated content may be incorrect.

Manual control tab:

A screenshot of a computer

AI-generated content may be incorrect.

## stewart\_display.py snippets

### Code:

A computer screen shot of text

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A black background with text

AI-generated content may be incorrect.

### GUI:

A screenshot of a computer screen

AI-generated content may be incorrect.

## Next Week’s objectives:

So far I managed to build a solid foundation for my software which I can easily maintain and add further functionality as this project evolves over the 12 week period. My goals for Week 4 are to replace the dummy\_airplane\_app.py node for the actual airplane telemetry obtained from the X\_Plane flight simulator.

Once I manage to successfully integrate the airplane telemetry from X\_Plane I will move onto integrating the inverse kinematics module for the actual mechanical chair as well as performing real life tests to obtain the distance/pressure lookup table. This will be crucial in ensuring the chair’s movements replicate the airplane’s movements as best as possible.

# Week 4 – Integrating XP-Python into code stubs

During this week’s meeting with my supervisor we covered the following targets were set:

-We’ve been given a fair amount of freedom to choose the flight GUI and experience

-There’s a pre-made UDP tx/rx premade library module which I should integrate into my existing code as soon as possible. Reason for this is to both clean up the hastily made code and improve maintainability

-Separate dummy airplane module into two UDP channels where one only sends telemetry values and the other sends safe state data

-Look into three possible contenders for safety sensor messaging: Serial, UDP and I2C

The main challenge to tackle aside from integrating XPPython into my system is to clean up my UDP socket code. Currently it’s really messy and sockets are declared where needed which leads to a messy code structure. I’ll be using a pre-exisiting udp tx/rx module which simplifies syntax and abstracts most of the previous socket code under simpler class objects.

If that can be done I also want to polish further polish the GUI and see if I can integrate an example GUI given by my supervisor which appears to be a great succession point for my current software.

Modified network map (separated dummy\_airplane\_app.py into two dedicated nodes):

A diagram of a diagram

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

**TODO:**

**-Address inconsistency in fig naming (Week 1 and 2 figs are named 2.x and 3.x)**

# References

[1]Falcon 2 programme homepage: <https://www.aerosociety.com/careers-education/schools-outreach/the-falcon-2-programme/>

[2] Motion Platform GitHub Page: <https://github.com/michaelmargolis/MdxMotionPlatformV3>

[3] MDX rollercoaster news page: <https://www.mdx.ac.uk/news/2024/12/new-scientist/>

[4] XPPython Homepage: <https://xppython3.readthedocs.io/en/3.1.5/index.html>

[5] Stewart\_Py git repository: <https://github.com/Yeok-c/Stewart_Py>

[6] Qt designer manual page: <https://doc.qt.io/qt-5/qtdesigner-manual.html>