POLITECNICO DI TORINO

Master's Degree in MECHATRONIC ENGINEERING

Master's Degree Thesis

DEVELOPMENT OF AN AUTOPILOT SYSTEM FOR FLIGHT SIMULATION IN A PREPAR3D SIMULATION ENVIRONMENT

Supervisors

Prof. ELISA CAPELLO

Dr. STEFANO PRIMATESTA

Ing. MARIO NIGRA

Candidate

ALESSANDRO MERLINO

OCTOBER 2024

Summary

The aim of this thesis is to develop a general-purpose flight simulation station, implement a data acquisition system to collect information from the simulator, and design an autopilot capable of performing aircraft takeoff, executing a standard turn to align with a specified point based on latitude and longitude coordinates, and maintaining a desired altitude. This study is part of a larger project that also involves the creation of a wearable device providing haptic and visual feedback to the simulator user, making the experience more realistic. The project is a collaboration between the Politecnico di Torino and Sipal SPA.

The simulation station consists of a configurable seat equipped with various hardware attachments, including a joystick, pedals, and throttle. For visual display, an ultra-short throw projector and two touchscreens were selected to display the cockpit and onboard instruments, and to allow interaction with them. The software used is Prepar3D, a professional license product by Lockheed Martin. C++ code was implemented to develop the data acquisition and autopilot modules, facilitating interaction with the software. Several communication protocols were utilized for the data acquisition system: SimConnect, a TCP/IP-based protocol developed by the simulator manufacturer, was used to extract data directly from Prepar3D; and MQTT, also TCP/IP-based, was employed to forward data to another module that controls the actuators for sensory feedback. The application is configurable via a configuration file that allows the selection of data to be extracted. The autopilot was designed for use with the Lockheed Martin F-35. The system comprises several units: the takeoff system, which enables the aircraft to take off; the controller that manages the standard turn maneuver; and the controller that maintains the desired altitude. For the turn maneuver, a controller was developed to execute a standard turn, a turn at a constant angular rate. After setting the bank angle between 30° and 45° using a PID controller, control actions are applied to the angular velocity relative to the aircraft's z-axis, with control executed by the aircraft's elevators. For altitude control, a PID controller was implemented, which adjusts altitude through the aerodynamic surfaces, specifically the elevators. Overall, the system provides faster and more precise performance compared to the autopilot offered by Prepar3D.

Acknowledgements

Giunto alla fine di questo percorso, è doveroso rivolgere un sincero ringraziamento a tutte le persone che, con il loro sostegno, mi hanno accompagnato fino a questo traguardo.

Innanzitutto, desidero esprimere la mia profonda gratitudine alla mia relatrice, la Prof.ssa Elisa Capello, per la sua guida, i suoi preziosi consigli e per essere stata presente fin dall'inizio di questo lavoro. La sua disponibilità e competenza sono state fondamentali per la buona riuscita di questa tesi.

Un ringraziamento speciale va anche al corelatore, il Dr. Stefano Primatesta, per il suo supporto, la disponibilità e i suggerimenti che hanno arricchito il progetto.

Desidero inoltre ringraziare il mio tutor aziendale, l'Ing. Mario Nigra, per l'assistenza costante e per avermi dato l'opportunità di lavorare in un ambiente stimolante e professionale, contribuendo alla mia crescita personale e professionale.

Il mio pensiero va poi ai miei genitori, a mio fratello, a Duca e Isotta, che con amore e pazienza mi hanno sempre sostenuto, credendo in me anche nei momenti più difficili. Senza il loro incoraggiamento, non avrei mai potuto raggiungere questo importante obiettivo.

Ringrazio tutta la mia famiglia, i miei nonni, i miei zii e i miei cugini che hanno sempre avuto fiducia nelle mie capacità e mi hanno offerto il calore e il supporto necessari per affrontare questo cammino con serenità.

Grazie Simone, per la tua amicizia, per aver reso piacevole le lezioni e lo studio per gli esami e per avermi fatto compagnia in questi anni.

Grazie a tutti gli amici e colleghi conosciuti a Torino per aver reso più semplice

il mio trasferimento e per i momenti passati insieme

Grazie a tutti gli amici conosciuti agli scout, per essermi sati vicini e per non essersi allontanati in questi anni, nonostante vivessi lontano.

Grazie anche a Matteo, Carmelo e Matteo. Grazie per essere venuti a farmi visita fino a Torino, per la vostra presenza, per avermi accompagnato e sostenuto, regalandomi momenti di allegria e spensieratezza.

Un pensiero particolare va alla mia fidanzata, Rebecca, per la sua pazienza, il supporto e per essere stata al mio fianco, anche se non fisicamente vicino a me.

Infine, un grazie lo dedico a me stesso, perchè alla fine, li esami li hai fattu iu.

"Non è nelle stelle che è scritto il nostro destino, ma in noi stessi: uomini forti, destini forti; uomini deboli, destini deboli. Non c'è altra strada."
William Shakespeare

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Objective of the Thesis

The primary objective of this thesis is to develop an autopilot system for an aircraft that can navigate to a given set of geographical coordinates and a desired altitude. This ambitious project has been conducted in close collaboration with Sipal SPA, which provided the application context, essential support, and invaluable industry insights throughout the research process.

This research aims to address the critical need for optimized aircraft control by focusing on the design and implementation of a sophisticated controller that ensures superior performance in terms of stability, responsiveness, and robustness. The primary goal is to integrate advanced control techniques and algorithms to develop an autopilot system capable of operating effectively under a wide range of conditions, thereby enhancing the reliability and efficiency of modern aviation technology.

For the purposes of this thesis, the highly advanced F-35 model by Lockheed Martin was utilized as the primary test platform. A comprehensive flight simulation laboratory was established and developed, featuring two simulation stations: one primary and one secondary. These stations were equipped with Prepar3D, a state-of-the-art simulation software developed by Lockheed Martin, which provided a realistic and robust environment for testing and development.

In addition to the simulation setup, a configurable data acquisition module was developed to extract and analyze data during the simulation runs. This capability was crucial for refining the autopilot system and ensuring it met the desired performance criteria. This part of the research also extended into a broader project that involved the creation of a wearable device designed to provide haptic and visual feedback to pilots during simulated flights. This device aimed to enhance the immersive experience of flight simulation, thereby offering pilots a more realistic

training environment.

The expected outcomes of this research are significant. By improving the performance of autonomous vehicles, the developed controller is anticipated to contribute greatly to operational safety and efficiency. The implementation of this autopilot system has potential applications across various sectors, including both terrestrial and aerial transportation. It represents a substantial advancement in autonomous driving technologies, promising to foster progress and innovation in multiple fields.

This project exemplifies the synergy between academic research and industrial collaboration, highlighting the importance of leveraging industry expertise to address complex engineering challenges. The partnership with Sipal SPA has been instrumental in providing the necessary application context and technical support, ensuring that the research is grounded in practical, real-world requirements. Through this collaborative effort, the thesis aims to advance the field of autonomous aviation, fostering innovation and progress in multiple related areas.

1.2 Thesis structure

This thesis is organized into several chapters, each addressing a specific aspect of the research. Now, we will examine the content of the various chapters:

1. Introduction: The initial chapter provides an introduction and a detailed overview of the current state of the art in flight simulation and the related softwares. This includes a discussion on existing technologies, methodologies, and relevant literature in the field. This part will be discussed in the following pages.

2. Description of the Simulation System and the Wearable System: In this chapter, the focus will be on the components and architecture of the simulation laboratory and wearable systems. The subtopics covered will include:

    - Prepar3D and SimConnect: An overview of the Prepar3D flight simulation software and the SimConnect API used for interfacing with the simulation environment, and the reason that led me to chose this software.
    - Hardware Components: A detailed description of the hardware utilized in the simulation setup.
    - Wearable System: An in-depth look at the wearable system, its architecture, and the tests conducted to evaluate its performance.

3. Autopilot Architecture: This chapter will delve into the design and implementation of the autopilot system. The topics covered will include:

- Introduction to Controllers in Aviation: A discussion on the role and importance of controllers in aviation.
- Regulations on Controllers: An overview of the regulations and standards governing the use of controllers in the aviation industry.
- Dynamic Specifications of Aircraft: A detailed examination of the dynamic specifications the aircrafts, with a particular focus on the F-35 due to its known model in Prepar3D.
- Implementation of the Autopilot: A comprehensive explanation of how the autopilot system was implemented in the simulation.

4. Test: This chapter will cover the testing phase of the research, including:
   - Data Collection: Methods and processes used for collecting data during the simulation.
   - Graphical Representation: Techniques for visualizing the collected data.
   - Data Analysis: Analysis of the data to evaluate the performance and effectiveness of the simulation and autopilot systems.
5. Conclusion: The final chapter will present the conclusions drawn from the research, including the discussion of main findings, a summary and discussion of the key findings from the research, and future developments: potential future developments and improvements based on the results and insights gained from this study.

Each chapter aims to provide a comprehensive understanding of the various components and aspects involved in the development and evaluation of flight simulation and wearable systems. This structured approach ensures a thorough exploration of the research topic and facilitates a clear presentation of the findings and conclusions.

1.3 Context and state of the art

The advancement of autonomous systems in aviation has become a pivotal focus in modern engineering, driven by the necessity for enhanced safety, operational efficiency, and technological reliability. The development of autopilot systems, capable of precise navigation and control, is a key area of research, aimed at addressing these critical needs. In this context, the collaboration with Sipal SPA has provided invaluable industry insights and essential support, shaping the direction and execution of this research project.

Simulators are especially suited for training situations which are impractical, difficult, dangerous or expensive to reproduce in a live environment. There are

many potentially dangerous situations that aircrew may only encounter infrequently. If these situations are encountered they need to be dealt with efficiently to avoid serious consequences. Simulators can be used to present trainees with such unusual scenarios in a repeatable and controllable manner without presenting risk to the crew, the aircraft, other operators or to the environment. In the subsequent paragraphs, a comprehensive analysis will be conducted on the various types of simulators and simulation software currently available in the market. This examination will encompass an in-depth review of their functionalities, applications, and the technological advancements that distinguish them.

1.3.1 Simulator

Flight Simulation Training Devices (FSTDs) play a crucial role in modern aviation training, offering a safe, controlled, and cost-effective environment for aircrew training and testing. These devices are particularly valuable for scenarios that are impractical, dangerous, or expensive to reproduce in a live setting, such as emergency procedures, system failures, and adverse weather conditions. By enabling the repetition of complex and critical maneuvers, FSTDs help ensure that aircrew are well-prepared to handle rare but high-risk situations efficiently and safely.

There are several types of FSTDs, each with varying levels of complexity and capability:

1. Flight Navigation Procedure Trainer (FNPT): This fixed-base generic system is used primarily for initial and refresher training, including basic and safety procedures, emergencies, navigation, instrument rating (IR), and multi-crew cooperation (MCC). It provides a foundational platform for aircrew to develop essential skills.

2. Flight Training Device (FTD): Also a fixed-base system, the FTD is type-specific, simulating a particular model of aircraft. It builds on the capabilities of the FNPT, designed for type rating training. However, it does not include motion or vibration systems, which limits its use for certain testing purposes.

3. Full Flight Simulator (FFS): The FFS is the most advanced type of FSTD, featuring a motion-base system that provides realistic motion and vibration cues. This high level of technical complexity makes it suitable for proficiency checks and skill tests, offering a comprehensive training experience that closely replicates real flight conditions.

In addition to these primary categories, other training devices exist that, while not regulated by official standards, still provide significant training benefits:

Figure 1.1: FNTP Simulator

- Computer Based Trainer (CBT): Used during initial training, CBTs are effective for self-learning via desktop computers. These trainers are interactive, allowing trainees to engage with touchscreen or video interfaces to learn about human-machine interface (HMI) instruments and displays.

- Basic Instrument Training Device (BITD): Designed for instrument familiarization and training, BITDs replicate the behavior of aircraft instruments using software, providing a practical and accessible tool for initial training phases.

- Part Task Trainer (PTT), Cockpit Part Task Trainer (CPT), Virtual Interactive Procedure Trainer (VIPT): These devices focus on basic procedure training, such as pre-flight checklists and engine start procedures. They

Figure 1.2: FTD Simulator

Figure 1.3: FFS Simulator

are typically fixed-base replicas of cockpits without visual systems, used to familiarize aircrew with cockpit instruments and procedures.

- Helicopter Mission Trainer (HMT): Specifically designed for collective mission training, HMTs involve multiple crew members or several aircraft in a coordinated exercise. These trainers enhance skills in communication, coordination, navigation, and mission rehearsal, simulating real-life scenarios like search and rescue or law enforcement operations.

Overall, the structured use of FSTDs ensures comprehensive and effective training for aircrew, enhancing their ability to manage a wide range of scenarios. This approach not only improves operational safety but also provides a cost-effective and efficient means of maintaining and advancing aircrew proficiency.

1.3.2 Software simulator

In this section, we will delve into flight simulation software, analyzing them based on several key categories. These categories will help us understand the capabilities and limitations of different simulation platforms. The aspects we will cover include:

- Programmability: We will examine the software development kits (SDKs) availability, the programming community, supported languages, and the extent to which the software can be programmed or customized, including the creation of new aircraft models.
- Multi-domain: This will cover the software's ability to simulate various types of scenarios, as well as the range of vehicles that can be simulated.
- Multi-user: We will look into the software's capability to simulate multiple objects that interact within the simulation environment.
- Multi-screen: This involves the software's support for separate screens for different purposes, such as scenario display and instrumentation, as well as compatibility with touch screens.
- Compatibility with VR and AR: We will assess how well the software integrates with virtual reality (VR) and augmented reality (AR) technologies.
- Multi-server: This examines the software's ability to distribute the simulation workload across multiple servers, such as having one server simulate the scenario and another simulate the aircraft.
- Hardware and Software Requirements: We will identify the necessary hardware and software specifications required to run the simulation software efficiently.

- Offline Functionality: We will consider whether the software can operate without an internet connection.
- Licensing: Finally, we will analyze the licensing options available, including whether the software is available for personal and commercial use, and the associated costs.

By evaluating these categories, we aim to provide a comprehensive overview of the capabilities and suitability of various flight simulation software options.

Flight Simulator

Flight Simulator is a flight simulator developed by Microsoft, offering a wide range of aircraft, airports, and detailed scenarios worldwide. It was first released in 1982 and has had many iterations over the years, with the latest version, Microsoft Flight Simulator 2020, released in 2020. This latest version is particularly notable for its graphics and attention to detail, using real-world data to create a highly realistic flight experience. It is popular among aviation enthusiasts and flight simulator fans.

Figure 1.4: Flight Simulator logo

Programmability: Flight simulation software offers a robust Software Development Kit (SDK) allowing developers to create custom content such as:

1. Aircraft: Detailed aircraft with interactive cockpits, accurate modeling, and realistic flight behaviors.
2. Scenarios: Detailed airport, city, and famous landmark scenarios using advanced modeling and texturing tools.
3. Instruments: Custom navigation instruments, control panels, and additional displays.

4. Effects: Special effects like lighting, smoke, and fire to enhance realism

The SDK is typically designed to be flexible and powerful, enabling the development of a wide range of custom content. For example, the SDK for Microsoft Flight Simulator (MFS) is written in C++ and supports a large developer community, offering extensive 3D modeling tools.

Multi-Domain: Flight Simulator software often includes a variety of aircraft models by default, primarily civilian. However, additional model are also available online, allowing for diverse simulation scenarios. Websites like Simviation and FlightSim offer extensive resources for adding new aircraft to simulators.

Multi-User: Flight simulators support multiple user interactions through several features:

1. Standard Multiplayer: Allows multiple players to fly in the same environment, each with their own aircraft. Players can see and follow each other but with limited direct interaction.

2. Organized Events and Races: The community often organizes multiplayer events and races with predetermined routes or specific challenges.

3. Third-Party Software: Some third-party applications provide advanced multiplayer functionalities, including shared cockpits, voice communication, and coordinated missions.

Multi-Screen: Flight Simulator software supports multi-screen configurations, enhancing the immersive experience:

1. Monitor Configuration: Users can connect multiple monitors to extend the desktop and use each as part of the flight simulation experience.

2. Display Settings: The simulator allows for adjusting resolution and display settings to fit multiple screens.

3. View Customization: Users can customize the view layout, setting different views (e.g., cockpit view on one screen, external view on another).

Touch screens are also supported, allowing users to interact with the simulator interface directly, adjusting settings, navigating menus, and interacting with onboard instruments.

Compatibility with VR and AR: Flight simulator support VR headsets, providing an immersive, panoramic, and three-dimensional flight experience. Although direct AR support within the simulator is limited, external AR applications can enhance the experience by providing additional information on aircraft and surroundings.

Multi-Server: While native multi-server support is generally not available, third-party software can offer advanced networking features to distribute workloads across multiple servers. For example, JoinFS or VATSIM can create a distributed network for users to connect and fly together on remote servers.

Hardware and Software Requirements: To run flight simulation software efficiently, certain hardware and software specifications are necessary:

- Operating System: Windows 10 (v. 1909) or later.
- Processor: Intel i5-4460, Ryzen 3 1200, or equivalent.
- GPU: NVIDIA GTX 770, Radeon RX 570, or equivalent.
- Memory: 8 GB RAM, 2 GB VRAM.
- Storage: 150 GB of free space.
- DirectX: Version 11 or higher.
- Internet Connection: Required for initial activation, updates, and additional content downloads.

Offline Functionality: Flight simulation software can function offline, allowing users to fly freely and use downloaded aircraft and scenarios. However, some features, such as multiplayer, real-time weather, and traffic data, require an active internet connection.

Licensing and Costs: Flight simulation software is available in various editions with different pricing:

1. Standard Edition: Typically priced between $59.99 and 69.99$ USD.
2. Deluxe Edition: Includes additional aircraft and airports, priced between $89.99 and 99.99$ USD.
3. Premium Deluxe Edition: The most comprehensive edition, priced between $119.99 and 129.99$ USD.

Commercial licenses are available for flight schools, airlines, and content developers, offering specialized features and rights for training, content creation, and distribution.

Prepar3D

Prepar3D is a flight simulator based on the game engine of Microsoft Flight Simulator X, developed by Lockheed Martin. Prepar3D is primarily oriented towards professional use, such as pilot training, research, and development. It offers a wide range of features, including aircraft, scenarios, and development tools to customize and enhance the simulation experience. It is known for its robustness and flexibility.

Figure 1.5: Prepar3D logo

Programmability: The Prepar3D Software Development Kit (SDK), which provides developers with the necessary tools and resources to create custom add-ons, extend the simulator's functionalities, and develop additional scenarios and content for Prepar3D (P3D). The Prepar3D SDK includes a wide range of resources and documentation for developers, including:

1. Creation of New Aircraft and Aircraft Models: Developers can utilize the tools provided in the SDK to create new aircraft models, including 3D models, textures, special effects, and cockpit instruments.

2. Development of Custom Scenarios: Using the scenario development tools included in the SDK, it is possible to create custom scenarios that encompass new airports, terrains, scenario objects, and environmental details.

3. Implementation of Custom Functionalities: Developers can enhance Prepar3D's capabilities by creating add-ons that introduce new functionalities and avionics systems, new flight controls, air traffic control systems, and more.

4. Development of Custom Cockpit Instruments: Developers can create custom cockpit instruments, control panels, and user interfaces to add functionalities and additional information within the simulator.

5. Integration with Other Systems: Prepar3D offers various integration options with other systems and technologies, such as real-time navigation systems, air traffic management systems, pilot assessment systems, and more.

The SDK supports programming languages such as C++ and C# and offers 3D modeling tools. The community is highly developed, facilitating collaboration and knowledge sharing among developers.

Multi-Domain: Prepar3D offers extensive multi-domain simulation capabilities, providing more variety compared to other simulators. It includes a range of aircraft, from commercial airliners to military jets, and supports various add-ons. The multi-domain capabilities include:

1. Aircraft: Commercial, military, historical, and more.

2. Helicopters: Civilian and military models.

3. Ground Vehicles: Cars, trucks, buses, trains, and more through third-party addons.

4. Boats and Ships: Navigation on rivers, lakes, seas, and oceans.

5. Drones: Simulation of UAVs with dedicated addons.

6. Ground Equipment and Airport Infrastructure: Service vehicles, ground support equipment, and more.

Multi-User: Prepar3d supports Multi-user functionality, however it is necessary to have multiple licenses and to host the session having a professional plus license is required.

Multi-Screen: Prepar3D provides robust multi-screen support, enhancing the simulation experience through:

1. Monitor Configuration: Set up multiple monitors for different views, such as the main flight environment and cockpit instrumentation.

2. Window Management: Open, drag, and resize multiple windows for customized information display across monitors.

3. Touchscreen Compatibility: Supports touchscreen interaction for controlling the simulator, selecting commands, and moving the camera.

4. Advanced Configuration: Utilize third-party tools for advanced multi-screen setups, managing windows, distributing views, and integrating with external hardware and software.

VR and AR compatibility: Prepar3D supports virtual reality (VR) natively, enhancing the immersive experience of flight simulation. It supports devices like Oculus Rift, HTC Vive, and Valve Index, although is required a professional plus license in order to use VR functionalties. While augmented reality (AR) is not natively integrated, it can be explored using third-party solutions and compatible tracking technologies.

Multi-Server: Is possible to use a multi-server architecture by using third-party software. Native multi-server architecture is not supported.

Hardware and Software Requirements: Prepar3D has specific hardware and software requirements to ensure optimal performance:

- Hardware:
  - CPU: Quad-core or higher with at least 3.0 GHz; multi-core processors with higher clock speeds are ideal.
  - RAM: 16 GB recommended, 32 GB or more ideal for complex scenarios and heavy addons.
  - GPU: Dedicated graphics card with at least 4 GB of video memory; 6-8 GB or more for optimal performance and high-resolution graphics.
  - Storage: SSD for the operating system and Prepar3D installation; additional SSD or HDD for simulation data and addons.
  - Network: Internet connection for updates and addons; high-speed connection for online resources and multiplayer support.
- Software:
  - Operating System: Windows 10 (64-bit) is recommended.
  - DirectX: Version 11 or later.
  - Additional Software: Updated drivers for hardware components, and antivirus software that does not interfere with Prepar3D.

Offline Functionality: Prepar3D can be used offline without needing additional downloads, making it suitable for environments with limited internet access.

Licensing: Prepar3D offers various licensing options tailored to different user needs:

- Academic License: For students, educators, and academic institutions for non-commercial teaching and research.
- Professional License: For commercial use by aviation professionals, simulation companies, and software developers.
- Developer License: For addon developers, including the ability to distribute commercial addons and access to the SDK.
- Enterprise License: For organizations with specific simulation and training requirements, offering advanced and customized features.

For commercial use, the Professional License is required, costing $350. The Professional Plus License, costing $2,750, includes additional functionalities such as multi-user and VR functionalities.

XPlane 12

X-Plane 12 is a popular flight simulator that offers a realistic flying experience. Developed by Laminar Research, X-Plane is known for its fidelity to physical flight dynamics and its flexibility. It is available on multiple platforms including Windows, macOS, and Linux. It is utilized by aviation enthusiasts and industry professionals alike for various purposes, including pilot training, aircraft design, and research.

Figure 1.6: XPlane 12 logo

Programmability: X-Plane 12 features a robust Software Development Kit (SDK) that empowers developers to create plugins and additional modules for the simulator. The SDK includes documentation, code examples, libraries, and development tools necessary for building plugins in C/C++, along with APIs to interact with the simulator. This toolkit allows developers to extend X-Plane's capabilities or create new tools, aircraft, scenarios, and more. The SDK facilitates the development of various add-ons:

1. Aircraft Plugins: Developers can create new aircraft models, including custom flight physics, graphics, and sounds.

2. Navigation Tools: Advanced navigation instruments such as GPS, VOR, ADF, HSI can be developed to enhance user navigation experiences.

3. Cockpit Components: Custom instruments, control panels, or user interfaces can be created to improve the in-cockpit experience.

4. Custom Scenarios: New or enhanced scenarios with detailed buildings, terrain, vegetation, and 3D objects can be developed.

Multi-Domain: X-Plane 12 supports a wide range of aircraft, including both military and civilian models. This includes:

1. Civilian Aircraft: Simulations of commercial airliners like the Airbus A320, Boeing 737, and recreational aircraft like the Cessna 172 and Piper PA-28.

2. Cargo Aircraft: Simulations of cargo planes such as the Boeing 747 Freighter, Antonov An-124, and Lockheed C-130 Hercules.

3. Historic Aircraft: Simulations of historic planes like the Boeing 707, Douglas DC-3, and Lockheed Constellation, enabling users to relive aviation history.

4. Helicopters: A variety of helicopters, from light models like the Eurocopter EC135 to larger ones like the Sikorsky UH-60 Black Hawk, are available for simulation.

5. Military Aircraft: Simulations of military aircraft such as fighters, bombers, transports, and combat helicopters, including models like the F-16 Fighting Falcon, F/A-18 Hornet, and military C-130 Hercules.

6. Experimental Aircraft: Simulations of experimental, prototype, and futuristic aircraft, allowing exploration of advanced aviation concepts.

Multi-User: X-Plane 12 includes extensive multi-user capabilities: uers can fly together in the same airspace, connecting via local networks or the Internet for real-time interaction, structured multiplayer events or casual flight sessions can be hosted on dedicated servers or by individual users.

Multi-Screen: For multi-screen setups, X-Plane 12 supports:

1. Monitor Configuration: Users can configure additional monitors to display the main cockpit, additional instrumentation, or external views.
2. View Customization: Views on each monitor can be customized, allowing users to decide which instruments and views to display.
3. Touch Screen Compatibility: X-Plane 12 is compatible with touch screens, allowing interaction with the user interface through touch gestures. This is particularly useful for instrumentation or other interactive cockpit functions.

VR and AR compatibility: X-Plane 12 supports both virtual reality (VR) and augmented reality (AR), enhancing the immersive experience.It is compatible with a variety of VR devices, including Oculus Rift, HTC Vive, and Valve Index. VR mode can be enabled within the X-Plane settings, allowing users to fully immerse themselves in the flight environment, interact with cockpit instruments, and enjoy panoramic views in a 3D setting. AR mode can be used with third-party solutions to create AR experiences. AR can provide additional flight information, such as navigation data, points of interest, or weather data, overlaid on the real view through the device's camera.

Multi-Server: The configuration can be complex, and while multi-server support is not native to X-Plane, there are methods to achieve this through networking and additional software.

Hardware and Software Requirements:

1. CPU: A modern multi-core processor with a frequency of at least 3 GHz is required. High-end CPUs are recommended for optimal performance, especially when using detailed scenarios or additional plugins.
2. RAM: At least 8 GB of RAM is required, but 16 GB is recommended for a better experience, particularly with high-resolution scenarios or detailed textures.
3. GPU: A dedicated graphics card with at least 2 GB of VRAM is necessary.

4. Storage: A minimum of 150 GB of free disk space is required for installation, including additional content such as scenarios and aircraft.

5. Input Devices: A range of input devices, including joysticks, flight controllers, pedals, and keyboards, can be used. Reliable and well-configured input devices are essential for an optimal flight experience.

6. Operating System: X-Plane 12 is compatible with Windows, macOS, and Linux. An updated and supported operating system is necessary for compatibility and optimal performance.

7. DirectX / OpenGL: X-Plane uses DirectX on Windows and OpenGL on macOS and Linux for graphics and 3D rendering. Ensure the latest graphics drivers and support for DirectX or OpenGL are installed.

Offline Functionality: X-Plane 12 can be used offline by downloading all relevant scenario data beforehand.

Licensing: X-Plane 12 offers several licensing options:

1. Personal License: Priced around $80, or $100 for the DVD version, suitable for individual users.

2. Academic License: Available for educational purposes.

3. Professional License: Priced at $1,000 for the online version and $1,250 for the USB dongle version, suitable for software development and commercial use.

These comprehensive features and capabilities make X-Plane 12 a powerful tool for both casual users and professional developers in the aviation simulation community.

AeroFly

Aerofly is a flight simulator available on various platforms, including PC, macOS, mobile devices, and VR. Developed by IPACS (Interactive Panorama Aircraft Cockpit Systems), it offers a highly detailed flying experience with a focus on high-quality graphics and user-friendliness. However, it may not provide the same complexity or depth of simulation as some other simulators like X-Plane or Prepar3D.

Figure 1.7: Arofly logo

Programmability: Aerofly provides developers with a Software Development Kit (SDK) that enables the creation of customized content for the flight simulator. This SDK offers a variety of tools and resources useful for developing additional aircraft, detailed scenarios, custom functionalities, and more. Developers can use the SDK to integrate their work with Aerofly and enhance the user experience. Typical features of the Aerofly SDK include:

- 3D Modeling Tools: Allow the creation of 3D models of aircraft and scenery.
- Aircraft and Scenery Importing: Facilitates the integration of new content into the simulator.
- Flight System Programming: Enables modification of existing flight systems or creation of new custom systems.

Multi-Domain: Aerofly supports a wide range of aircraft, including:

- Helicopters: Various types of helicopters for complex flight operations.
- Airliners: Models of commercial aircraft for airline flight simulations.
- Business Jets: Aircraft for private and corporate flights.
- Historical Aircraft: Historical models to relive the aviation of the past.

Multi-User: Aerofly does not natively support multi-user functionality, limiting the possibility of multiplayer flights.

Multi-Screen: Although Aerofly does not natively support multi-screen configurations, solutions depending on the hardware used do exist. Additionally, Aerofly supports the use of touch screens for interacting with the simulator.

VR and AR compatibility: Currently, Aerofly supports Virtual Reality (VR) on some platforms, offering users an immersive flight experience with a sense of presence and realism. VR availability depends on the specific version of Aerofly and the platform on which the simulator is run. For example, Aerofly FS 2 is known for its VR support on platforms like Oculus Rift, HTC Vive, and Windows Mixed Reality.

Multi-Server: Similar to other simulators, Aerofly can be configured to support multi-server setups, although this is not a native feature. Such configurations require multiple simulator licenses.

Hardware and Software Requirements:

- Processor (CPU): A quad-core processor or higher is recommended for an optimal experience. Aerofly benefits from powerful CPUs to handle flight simulation and physics calculations.
- Memory (RAM): At least 8 GB of RAM is recommended to avoid lags and long loading times.
- Graphics Card (GPU): A dedicated graphics card with at least 2 GB of VRAM is required. A mid-range or higher-end graphics card is preferable for smoother and more detailed graphics.
- Disk Space: It is advisable to have at least 20-30 GB of disk space for installing the simulator and additional content.
- Operating System: Aerofly is available for various platforms, including Windows and macOS. Ensure that the operating system is compatible with the simulator.
- DirectX / OpenGL: Aerofly requires DirectX on Windows and OpenGL on macOS for graphics. The appropriate versions of these libraries must be installed on the system.

Offline Functionality: Aerofly can be used offline, provided all relevant data is downloaded beforehand.

Licensing: Aerofly is available for personal use. For commercial licensing, it is advisable to contact IPACS directly, as there is no standard commercial option available on the official website.

FlightGear

FlightGear is an open-source and free flight simulator developed by a community of enthusiasts and programmers from around the world. FlightGear offers a realistic flight simulation experience, with a strong emphasis on physical realism and fidelity to details. However, since it is developed by a community of volunteers, it lacks corporate resources behind it like some other commercial flight simulators, so it may not have the same amount of pre-packaged content.

Figure 1.8: FlightGear logo

Programmability: FlightGear offers a development toolkit known as the "FlightGear Development Kit" (FGSDK), which provides developers with the necessary tools and resources to create customized content. This SDK includes various features that allow for the creation and integration of new aircraft, scenarios, cockpit instruments, and more. Key components of the FGSDK include:

1. 3D Modeling Tools: The FGSDK includes tools for creating and modifying 3D models for aircraft and scenery objects. These tools enable developers to create detailed models that can be integrated into the simulator.

2. Animation Tools: These tools allow developers to add animations to 3D models, such as aircraft control movements, retractable wheels, opening doors, and more.

3. Documentation and Tutorials: The FGSDK provides comprehensive documentation and tutorials to help developers familiarize themselves with the development process and utilize the SDK resources effectively.

4. Code Examples: The FGSDK offers code examples and reference materials to assist developers in understanding how to integrate their creations into the simulator.

Multi-Domain: FlightGear supports a wide range of aircraft and aviation types, including:

1. Civil Aircraft: A variety of civil aircraft can be simulated, from small training planes to airliners and commercial jets, including single-engine, twin-engine, jet, and turboprop aircraft such as the Cessna 172, Boeing 737, and Airbus A320.

2. Historical Aircraft: FlightGear offers a selection of historical aircraft from various eras, including World War II planes and aircraft from the 1950s and 1960s. Notable examples include the Supermarine Spitfire, Messerschmitt Bf 109, and North American P-51 Mustang.

3. Military Aircraft: A variety of military aircraft are available for simulation, including fighters, bombers, transport planes, and helicopters. Modern aircraft such as the F-16 Fighting Falcon, F/A-18 Hornet, and Sukhoi Su-27 are included, as well as historical aircraft like the B-17 Flying Fortress and P-38 Lightning.

4. Helicopters: Both civil and military helicopters can be simulated, such as the Bell 206 JetRanger, Eurocopter AS350 Écureuil, and Boeing AH-64 Apache.

5. Gliders and Hang Gliders: FlightGear also supports the simulation of gliders and hang gliders, allowing users to experience motorless flight and take advantage of thermals and updrafts for long-distance travel.

Multi-User: FlightGear supports multiplayer mode, enabling multiple users to participate in the same flight session and interact with each other and the virtual environment.

Multi-Screen: FlightGear allows for multi-screen configurations, which can be set up using various camera view windows. Although not natively supported, there are community-driven solutions available for setting up multi-screen displays.

VR and AR compatibility: FlightGear has experimental support for virtual reality (VR). Although not natively integrated, there are ongoing community efforts to implement VR functionality.

Multi-Server: FlightGear supports multi-server configurations, allowing for distributed computing to enhance performance and capabilities. Although not a native feature, experimental setups can be found through community resources.

Hardware and Software Requirements:

1. Processor: A modern processor with at least two cores is recommended. More processing power will result in better performance.
2. Memory (RAM): At least 4 GB of RAM is recommended, though 8 GB or more is preferable for an optimal experience.
3. Graphics Card: A dedicated graphics card with at least 1 GB of video memory is recommended for optimal performance, especially for high graphical settings. Mid-range or high-end graphics cards are preferred.
4. Disk Space: FlightGear requires at least 10 GB of disk space for installation and additional files, but more space is recommended for custom scenarios and aircraft.
5. Operating System: FlightGear is compatible with Windows, macOS, and Linux, and is also available for other less common platforms.
6. FlightGear Version: Use the latest stable version of FlightGear for the best performance and stability.

Offline Functionality: FlightGear can be used offline, and it is distributed under the GNU General Public License (GPL), an open-source license. This means the source code is available for free, and users have the freedom to use, modify, and distribute the software under the terms of the GPL.

Licensing: The GPL is a copyleft license, which requires that any derivative works or software using FlightGear code be distributed under the same open-source license, with the source code made available to users. This ensures that FlightGear remains an open-source project, benefiting all users. FlightGear can be used freely and without restrictions for personal and commercial purposes. However, modifications or distributions of the software must comply with the GPL terms, including providing the license text with the work.

Chapter 2

Simulation environment and wearable system

In this chapter, we will analyze the reasons behind choosing Prepar3D as our simulation platform, providing a detailed examination of SimConnect, an application programming interface (API) that allows external applications to interact with the simulation environment, enabling a wide range of functionalities and extensions. This capability is crucial for our project as it facilitates seamless integration with the wearable system we are developing.

We will describe the simulation environment in detail, outlining the architecture of the wearable system and how it interfaces with Prepar3D. This includes an in-depth look at both the hardware and software components that constitute the system, as well as the communication protocols employed.

Furthermore, we will highlight my contributions to this part of the project, specifically focusing on the development and integration efforts that were necessary to bring the simulation environment and the wearable system together.

By the end of this chapter, readers will have a comprehensive understanding of the simulation environment, the wearable system architecture, and the rationale behind the choice of Prepar3D, supported by a detailed comparison with alternative solutions.

2.1 Prepar3d and Simconnect

As previously mentioned, we will analyze the motivations that led to the selection of Prepar3D as the software for the simulation environment. This analysis will begin with a comparison of the advantages and disadvantages of the software discussed in the previous chapter.

Starting with Aerofly, several key disadvantages led to its exclusion as a viable

option. Firstly, Aerofly does not support multi-user mode, which is a significant limitation for collaborative simulation environments where multiple users need to interact simultaneously. Secondly, it lacks native support for multiple monitors. In complex simulation setups, the ability to manage multiple monitors is crucial for creating an immersive and comprehensive simulation experience. Lastly, Aerofly does not offer a commercial license. This absence is a critical factor, as a commercial license is essential for professional and business applications to ensure compliance with legal and operational standards. These limitations collectively render Aerofly an impractical choice for our specific requirements.

A similar evaluation applies to Microsoft Flight Simulator. While this software boasts a large and active community of developers, which ensures continuous updates and a wide range of add-ons, it also falls short in several critical areas. Despite its extensive feature set that aligns well with many of our requirements, the lack of a commercial license is a substantial drawback. Without the availability of a commercial license, Microsoft Flight Simulator cannot be legally used in a professional or corporate setting, which makes it unsuitable for our purposes.

Let's proceed with our analysis focusing on FlightGear. One of its primary advantages lies in its open-source nature, which provides a cost-free license suitable for corporate environments. Moreover, FlightGear has a notable history of involvement in significant past projects. However, a notable drawback is its lack of recent updates, resulting in a decline in current usage. Therefore, despite FlightGear's advantageous open-source license, its stagnation in updates led to its exclusion from consideration.

Moving forward, the more prominent contenders include X-Plane 12 and Prepar3D. Both platforms deliver exceptional simulation performance and realism by offering: high-fidelity graphics, realistic physics, and a comprehensive suite of simulation tools that are essential for creating a realistic training environment. Moreover they are bolstered by active developer communities and robust SDKs facilitating module and application integration within simulations. These attributes make them viable choices.

In the selection process, Prepar3D was preferred over X-Plane 12 for several reasons. The Prepar3D community is notably aligned with professional applications, offering tailored solutions and support that cater specifically to corporate and industry needs. In contrast, X-Plane 12 is often more associated with academic and educational settings, although it also offers strong simulation capabilities.

This analysis underscores Prepar3D as the optimal choice for our simulation environment, aligning closely with the professional requirements outlined in this thesis.

Let's conclude this section by discussing the specific features of Prepar3D's SDK and SimConnect.

Prepar3D stands out due to its comprehensive SDK (Software Development Kit)

and the integration capabilities offered by SimConnect. The SDK provides developers with a robust framework to extend and customize the simulation environment according to specific project requirements. It includes tools for creating aircraft models, scenery, and customizing the behavior of various simulation elements.

SimConnect, a fundamental component of Prepar3D, facilitates real-time communication between external applications and the simulation platform. It enables seamless integration of third-party software, allowing the development of complex simulations that can interact with external databases, control systems, or even other instances of Prepar3D running concurrently.

This combination of a powerful SDK and SimConnect's versatility makes Prepar3D a preferred choice for professional and enterprise-level simulations. These tools not only enhance the realism and functionality of the simulations but also enable the development of tailored solutions that meet specific industry needs.

In summary, the robust SDK and the integration capabilities provided by SimConnect reinforce Prepar3D's suitability for our simulation environment, aligning perfectly with the requirements discussed in this thesis.

2.2 Hardware components

We will now analyze the structure of the simulation environment, focusing on the hardware components that were acquired for the simulation setup and how they were integrated into the laboratory.

Starting with the central computer, we utilized an existing company-owned machine equipped with a 7th generation Intel Core i7 processor and two Nvidia T1000 graphics cards. The decision to use two graphics cards was made to provide additional DisplayPort inputs for connecting multiple screens and to distribute the GPU load across multiple cards, ensuring better overall performance.

The first graphics card is connected to a projector, providing a large, high-resolution display essential for immersive simulation experiences. The second graphics card is connected to three monitors, which are used to provide additional viewing angles and control interfaces. This setup allows for a comprehensive and interactive simulation environment.

While it may appear that a single graphics card, with its four available inputs, would suffice, opting for two cards offers several significant advantages. Firstly, it provides the flexibility to add more screens in the future without the need for additional hardware upgrades. This scalability is crucial for adapting to evolving simulation requirements.

Secondly, and more importantly, distributing the GPU load across two cards enhances performance. Initially, when using only one card, the GPU load was consistently high, averaging around 80% and sometimes peaking at 90%. This high

load resulted in a less fluid and optimal simulation experience, with noticeable performance drops during intensive simulation tasks. By using two graphics cards, we effectively balance the load, ensuring that each card operates within an optimal range, thus delivering a smoother and more reliable simulation experience.

This thoughtful integration of hardware components not only maximizes current performance but also provides a robust foundation for future expansion and enhancement of the simulation environment. The choice of utilizing two Nvidia T1000 graphics cards exemplifies our commitment to achieving high performance and scalability in our simulation setup.

Regarding the projection management, we selected an ultra-short throw projector: the Epson EH-LS650B. This decision was driven by several factors. The laboratory environment did not permit the installation of ceiling-mounted projectors, and placing the projector at the front of the setup avoids casting shadows over the seating area. This placement enhances the overall visual experience by ensuring that the projected images are unobstructed.

The projector, with its ultra-short throw capability, allows for a large, high-quality display even in limited space. This feature is particularly advantageous in our setup, where space optimization is crucial. The Epson EH-LS650B also offers excellent brightness and color accuracy, which are essential for creating a realistic and immersive simulation environment.

In addition to the projector, we have integrated multiple screens into the simulation setup. We acquired a 24-inch touchscreen monitor (DELL P2424HT) and a 14-inch touchscreen monitor (Verbatim PMT-14). These screens are used to display additional aircraft controls, enhancing the interactivity and realism of the simulation. Specifically, the 24-inch screen is used to show the multifunctional display of the F-35, providing crucial flight information and system status. The 14-inch screen displays the GPS, allowing easy navigation and situational awareness during the simulation.

Furthermore, a dedicated monitor serves as the console for managing the simulation. This console monitor allows to a second operator to control various aspects of the simulation, monitor performance, and make adjustments in real-time. The presence of this console is vital for ensuring smooth operation and providing the flexibility to modify simulation parameters as needed.

Overall, the careful selection and integration of these hardware components significantly enhance the functionality and realism of our simulation environment. The ultra-short throw projector and the strategically placed touchscreens work in tandem to create an engaging and immersive experience for users.

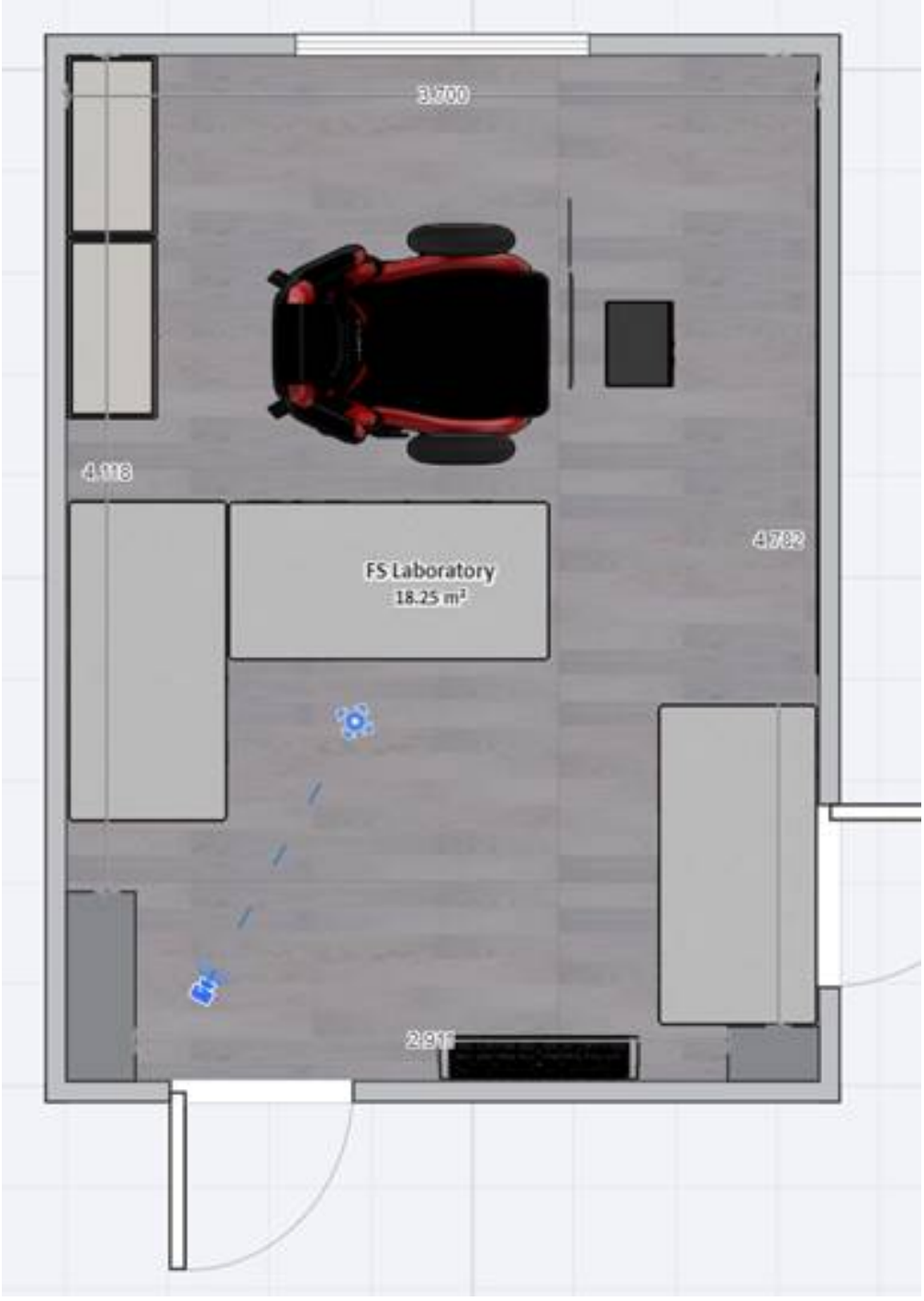

Figure 2.1: Laboratory's layout

To conclude, we also acquired various controls and a seat to enhance the immersive experience of the simulation. The controls purchased are equipped with multiple configurable buttons, as shown in Figure 2.2. These controls provide users with the ability to interact with the simulation in a highly detailed and realistic manner, mimicking the actual cockpit environment of an aircraft.

Figure 2.2: Commands

The seat chosen for the setup is the Next Level Racing Flight Simulator: Boeing Military Edition. This seat is designed to offer a high level of comfort and realism, essential for extended simulation sessions. It is fully adjustable and can be configured with various attachments and platforms to accommodate additional controls and tablets. This flexibility ensures that the seat can be tailored to the specific needs of different simulation scenarios and users.

Figure 2.3: Seat

These additions significantly contribute to the overall immersive experience, making the simulation environment more realistic and engaging. The combination of high-quality controls and a versatile, comfortable seat ensures that users can interact with the simulation in a manner that closely replicates real-world conditions.

Figure 2.4: Setup

2.3 Werable system

As previously mentioned, the objective of this thesis is not only to develop an autopilot capable of reaching a desired point given specific coordinates and altitude, but also to contribute to a broader project. This collaborative project, undertaken with a colleague, involves the development of an advanced wearable system. This system is equipped with a variety of actuators, including vibrational actuators,

heating plates, and other components that will be detailed further in the course of this thesis.

The primary goal of developing this wearable system is to significantly enhance the immersive experience of flight simulation. By integrating various actuators, the system is able to provide real-time haptic feedback to the user based on the events and conditions occurring during the simulation. For instance, vibrational feedback can simulate turbulence, while heating elements can mimic changes in temperature, thereby creating a more realistic and engaging environment for the pilot.

This approach not only aims to improve the sensory experience of simulation but also offers substantial economic benefits. Traditional Full Flight Simulators (FFS) are highly sophisticated and expensive pieces of equipment that require considerable financial investment for their acquisition, maintenance, and operation. In contrast, the wearable system developed in this project provides a cost-effective alternative without compromising on the quality of training. It enables more accessible and widespread use of advanced simulation technologies, potentially transforming training practices in the aviation industry.

Moreover, the modular nature of the wearable system allows for easy updates and customization, catering to specific training needs and scenarios. This flexibility ensures that the system can be adapted to various aircraft models and training requirements, offering a versatile tool for pilot education and skill development.

The system is composed of several actuators designed to enhance the immersive experience of the simulation. These actuators include:

- LED lights
- Vibrational actuators
- Peltier cells
- Heating plates
- LED matrix
- LED strip

These actuators are directly controlled by a Raspberry Pi microcontroller through a series of relays. The activation of these actuators is contingent upon the detection of errors generated during the simulation process. These errors are identified based on data extracted from Prepar3D through SimConnect.

Each type of actuator is associated with specific simulation conditions:

1. LED Lights: These are activated when the landing gear is either not extended or not retracted at appropriate times during the takeoff or landing phases. This visual cue serves as a critical reminder to the pilot to manage the landing gear correctly, thus enhancing procedural training.

2. Vibrational Actuators: These are triggered when the aircraft encounters a stall condition. The tactile feedback provided by these actuators simulates the physical sensation of stall warning systems found in real aircraft, thereby training the pilot to recognize and respond to stall situations promptly.

3. Peltier Cells: These cells are engaged to simulate the presence of ice on the aircraft's fuselage. The cooling effect created by the Peltier cells provides a realistic sensation of ice formation, emphasizing the importance of monitoring and managing in-flight icing conditions.

4. Heating Plates: These are activated when the aircraft's engine overheats. The heat generated by these plates mimics the increase in temperature experienced during engine overheating, providing an additional layer of realism and urgency to the simulation.

5. LED Matrix: This component is used to display the direction of the wind. By providing a visual representation of wind direction, the LED matrix helps the pilot understand and react to wind conditions during flight operations.

6. LED Strip: This strip is controlled based on the aircraft's bank angle. The LED strip offers visual feedback regarding the aircraft's attitude, which is particularly useful in low-visibility conditions where external visual references such as the horizon are not available. This assists the pilot in maintaining spatial orientation and ensures safe maneuvering of the aircraft.

The data variables extracted from Prepar3D that trigger these actuators are detailed in Figure 2.5.

These variables are critical for ensuring that the actuators provide accurate and timely feedback based on the simulation's real-time conditions.

By integrating these actuators, the system not only enhances the sensory feedback available to the pilot but also ensures that the simulation experience is as close to real flight conditions as possible. This innovative approach bridges the gap between virtual training and actual flight experience, offering a cost-effective and highly immersive training solution compared to traditional Full Flight Simulators (FFS).

Regarding the communication architecture, Figure 2.6 illustrates the communication schema in detail. Starting from the left, the necessary variables for generating errors are extracted from Prepar3D using SimConnect. These variables are then forwarded via the MQTT protocol on the local host to a module responsible for managing the error generation logic.

MQTT (Message Queuing Telemetry Transport) is a lightweight messaging protocol designed for efficient data communication between devices, especially in environments with limited bandwidth and high latency. MQTT operates on the

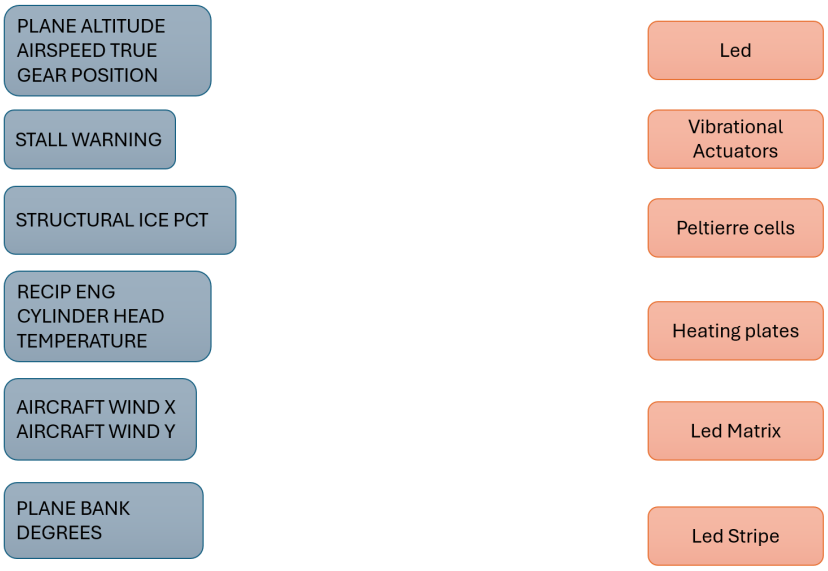

Figure 2.5: Actuators

client-server model and is commonly used in Internet of Things (IoT) applications to facilitate real-time data exchange.

The MQTT protocol involves three main components:

1. Broker: The central server in the MQTT architecture that manages and routes messages between clients. The broker ensures that messages are delivered to the appropriate subscribers based on their subscriptions. It maintains the state of all connected clients and manages message delivery.

2. Publisher: A client that sends messages to a specific topic on the broker. Publishers are responsible for generating and transmitting data to the broker, which then distributes this data to any clients that have expressed interest in that topic.

3. Subscriber: A client that receives messages from the broker by subscribing to specific topics. Subscribers express their interest in certain topics, and the broker delivers relevant messages to them. Subscribers can react to these messages in real time, enabling dynamic and responsive interactions.

MQTT's publish-subscribe model decouples the producers and consumers of data, allowing for scalable and flexible communication. Publishers and subscribers do not need to be aware of each other's existence, as all interactions occur through the broker. This model is particularly advantageous in scenarios where network

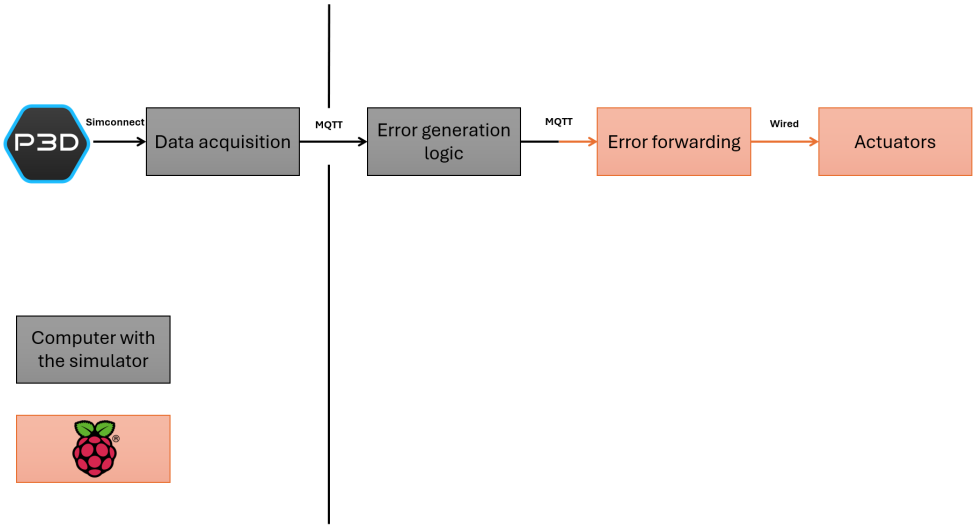

Figure 2.6: Comunication

conditions are variable or where a large number of devices need to communicate efficiently.

The error management module processes the incoming data to determine if any errors have occurred based on predefined conditions. Once errors are detected, this module forwards them, again using the MQTT protocol, to the Raspberry Pi microcontroller. The Raspberry Pi, in turn, controls the relays that switch the actuators on and off. This architecture ensures a robust and efficient communication flow between the simulation software and the hardware components.

My contribution to this part of the thesis project, which involved collaboration with a colleague, was the implementation of the module that interfaces with Prepar3D for data extraction. This module also handles the forwarding of data to the error logic management module. The implementation was done in C++, leveraging the language's performance and efficiency to handle real-time data processing and communication tasks. Is possible to see the specific code used for this implementation at the appendix A of the thesis.

The extraction of variables from Prepar3D is fully configurable through a text file. This configuration file requires the user to enter the key of the variable and its unit of measure, separated by a comma. This allows flexibility and customization based on the simulation requirements. An example of this configuration is shown in Figure 2.7. The complete list of variables available from Prepar3D can be consulted at the following link: `https://www.prepar3d.com/SDKv3/LearningCenter/utilities/variables/simulation_variables.html`.

Once the variables are extracted, they are managed dynamically within the

```
1   PLANE ALTITUDE,feet
2   AIRSPEED TRUE,knots
3   STALL WARNING,Bool
4   GEAR POSITION,Enum
5   AMBIENT WIND VELOCITY,Feet per second
6   STRUCTURAL ICE PCT,Percent over 100
7   RECIP ENG CYLINDER HEAD TEMPERATURE,Celsius
8   AMBIENT WIND DIRECTION,Degrees
9   YOKE X POSITION,Position
```

Figure 2.7: Config.txt file

system. This dynamic management ensures that the system can adapt to different simulation scenarios and conditions, providing accurate and timely feedback to the pilot.

For communication with the error logic management module, the MQTT protocol is used. We utilized an existing open-source project, Mosquitto (`https://mosquitto.org/`), for implementing the communication. In the implemented module, a batch file is created that runs the Mosquitto publisher with the extracted variable as the payload. This setup allows for efficient and reliable data transmission between the modules.

The batch file is updated and executed in each cycle, with a separate batch file for each variable. The latency of these operations was measured to ensure system performance. On average, it takes 120-130 milliseconds to extract and forward all variables per cycle. An additional 50 milliseconds delay is introduced to prevent flooding the MQTT broker with packets, ensuring smooth and stable communication.

Regarding the testing phase, the individual modules were rigorously tested to verify their functionality and performance. After successful individual tests, the modules were gradually integrated. This step-by-step integration approach ensured that any issues could be identified and resolved early, resulting in a stable and well-functioning system. The integrated system was then tested as a whole, ensuring that all components worked seamlessly together and provided the desired simulation experience.

Chapter 3

Autopilot architecture

As previously mentioned in the introduction of this thesis, this chapter will discuss the architecture of the implemented autopilot in detail. Initially, an overview of the types of controllers commonly used in the aeronautical field will be provided. This will include a discussion of the principles behind these controllers, their various applications, and the regulatory standards that govern their use. Subsequently, specific details about the F-35 model will be presented, highlighting its relevance and suitability as a reference for the controller design. The F-35 model, available through the Prepar3D simulation platform, serves as an ideal basis for this study due to its advanced technological features and comprehensive data availability. This combination of theoretical background and practical application will provide a thorough understanding of the autopilot's architecture and its implementation.

3.1 Controllers in aeronautic field

In the field of aeronautics, several types of control systems are employed to enhance the stability, control, and overall performance of aircraft. These systems are crucial for ensuring safe and efficient flight operations, as they help manage the complex dynamics of aircraft behavior under various conditions. The main categories of these control systems include Stability Augmentation Systems (SAS), Control Augmentation Systems (CAS), fly by wire systems (FBW), and Autopilot Systems. SAS, CAS, and FBW can be grouped in a wider type of control systems called: Inner-loop feedback control systems, while Autopilot Systems are also called: Outer-loop systems. Each of these systems has specific functions and applications that contribute to the overall handling and performance of the aircraft.

Stability Augmentation Systems (SAS)

Stability Augmentation Systems (SAS) are used to enhance the stability of aircraft that exhibit undesirable flying characteristics. These systems provide artificial stability by adjusting control surfaces to improve the aircraft's dynamic response. SAS are particularly useful for aircraft that operate over a wide range of flight conditions, where natural stability can vary due to changes in configuration or flight parameters such as Mach and Reynolds numbers. SAS can include devices such as pitch rate dampers, which use feedback from rate gyros to provide additional damping and improve handling qualities . The Stability Augmentation System (SAS) needed to be integrated with the aircraft's primary mechanical control system, which includes the stick, pushrods, cables, and bellcranks leading to the control surface or the hydraulic actuator that activates the control surface. The control authority of SAS, defined as the percentage of full surface deflection available, was typically limited to about 10%. One of the challenges associated with SAS was that the feedback loop provided commands that opposed pilot control inputs, making the aircraft less responsive to stick inputs. To address this issue, a washout filter was often added to the feedback loop to attenuate the feedback signal for constant values of the aircraft motion parameter. Another concern was the limited authority of the SAS actuator, necessitated by safety-of-flight requirements. Additionally, SAS sensors and computers were usually non-redundant or only dual redundant, which did not match the reliability of the mechanical flight control system. Despite these challenges, SAS was effective in improving the flying qualities of the aircraft.

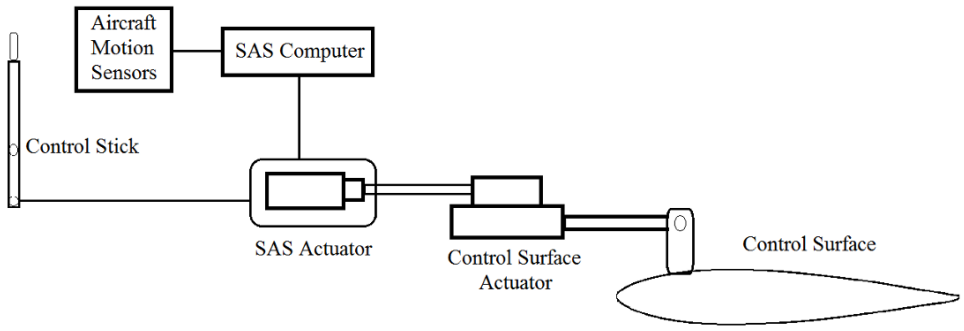

Figure 3.1: Simplified SAS

Control Augmentation Systems (CAS)

The next step in the evolution of aircraft feedback control systems was the development of Control Augmentation Systems (CAS). With a CAS, a pilot's stick input is processed in two ways: through the mechanical system and through the CAS electrical path. The design of CAS effectively eliminated the issue present in SAS where pilot inputs were opposed by the feedback loop. Control Augmentation Systems (CAS) are designed to improve the handling qualities and performance of aircraft by augmenting pilot inputs. These systems use a variety of sensors and actuators to adjust control surfaces dynamically, thereby enhancing maneuverability and stability. CAS can compensate for factors such as adverse yaw or control surface inefficiencies, ensuring more precise and responsive control during different phases of flight. Additional reliability was incorporated into CAS, allowing for an increase in control authority to approximately 50%. With CAS, the aircraft's dynamic response is typically well-damped, and control response is adjusted with the system gains to maintain desirable characteristics throughout the flight envelope. CAS provided significant improvements in aircraft handling qualities, enabling both dynamic stability and control response characteristics to be tailored and optimized according to the aircraft's mission.

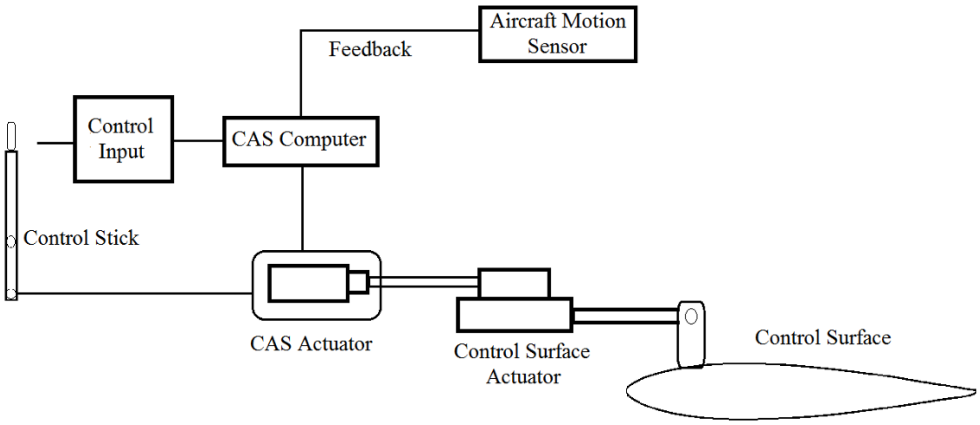

Figure 3.2: Simplified CAS

Fly by wire Systems (FBW)

Building on the excellent handling qualities achieved with Control Augmentation Systems (CAS), the next logical step in the development of feedback control systems was to eliminate the mechanical control system entirely, giving CAS full authority.

These systems are known as fly-by-wire (FBW) systems. A major advantage of using FBW systems is their enhanced reliability, achieved through the use of triple and quadruple redundancy components along with self-testing software. The full authority provided by FBW systems allows for significant customization of stability and control characteristics. This capability has led to the development of FBW systems with multiple feedback parameters and the weighting of feedback gains based on flight conditions and other parameters. Consequently, block diagrams for FBW systems can become quite complex due to the numerous feedback sensors involved.

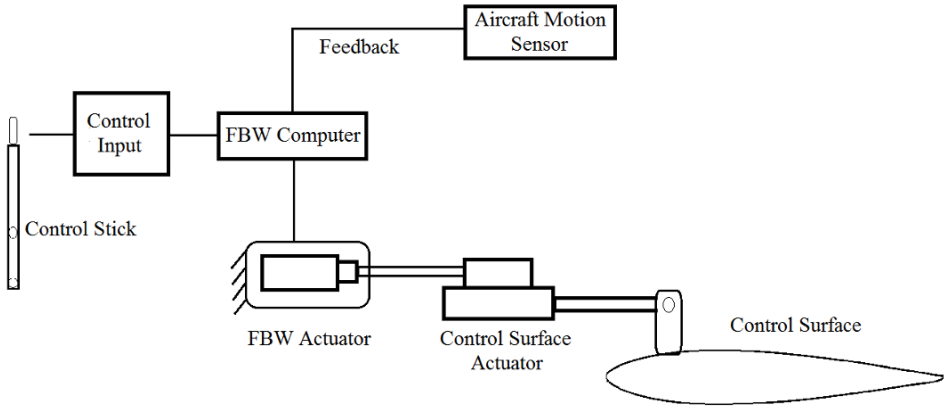

Figure 3.3: Simplified FBW

Autopilot Systems

Autopilots are a type of automatic control system used to reduce pilot workload by maintaining the aircraft's attitude, altitude, and speed. The evolution of autopilots began with simple displacement systems that could maintain pitch, roll, and heading angles. Modern autopilots integrate more advanced control laws and sensors to perform complex tasks such as automatic landing and maintaining a glide path during descent.

1. Displacement Autopilot: This type of autopilot maintains specific angular orientations (e.g., pitch, roll) by comparing the current attitude with a desired set point and adjusting control surfaces to minimize the error. It uses gyros to sense deviations and actuates the elevator or rudder to correct the aircraft's orientation .

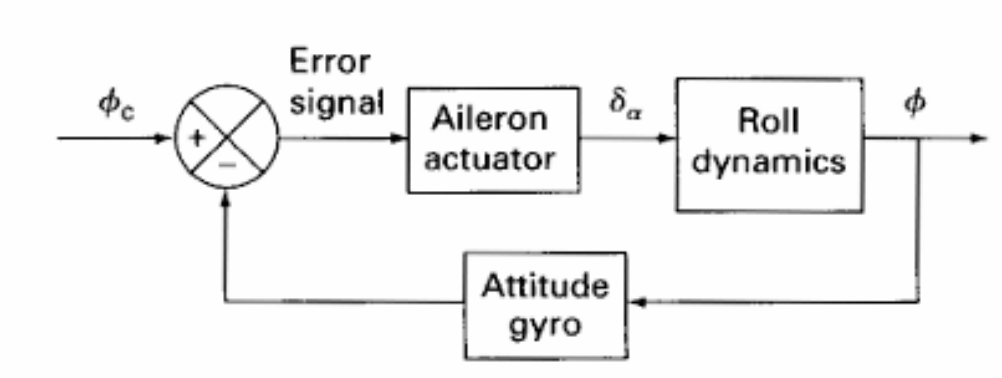

Figure 3.4: Roll attitude control system

2. Altitude Hold Autopilot: This system maintains the aircraft at a specified altitude by controlling the pitch attitude and thrust. It uses sensors to measure altitude deviations and adjusts the elevator and throttle to maintain the desired flight level. An altitude hold autopilot simplifies pilot tasks during cruise by keeping the aircraft stable at a set altitude without continuous manual input .

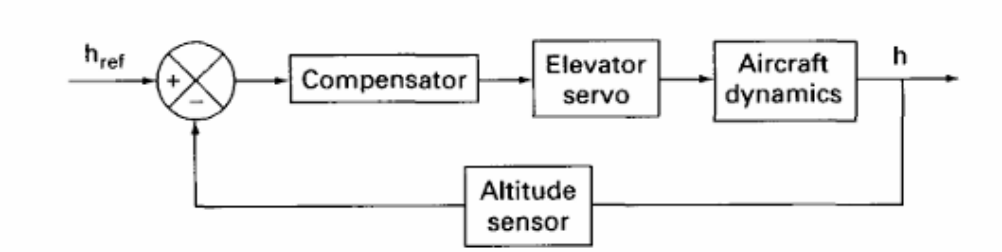

Figure 3.5: Altitude hold control system

3. Speed Control Autopilot: This system maintains a constant airspeed by adjusting the throttle. It uses feedback from airspeed sensors to regulate engine power, ensuring the aircraft remains at a desired speed. This type of autopilot is crucial for maintaining optimal flight conditions and fuel efficiency during different phases of flight .

4. Advanced Autopilots: Modern autopilot systems integrate multiple control functions, including lateral and longitudinal stability, automatic landing capabilities, and complex navigation tasks. These systems use state feedback control and optimal control theory to ensure precise and reliable aircraft performance under various conditions .

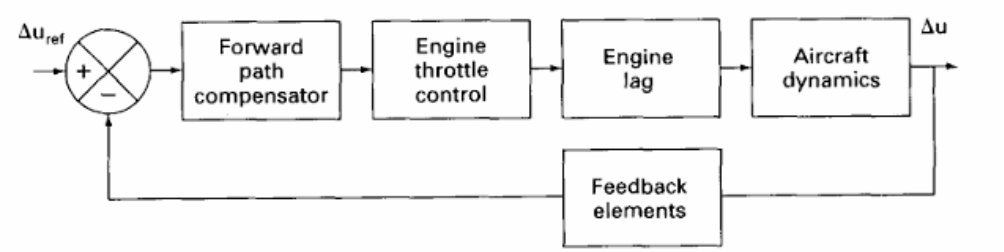

Figure 3.6: Speed hold control system

In summary, SAS, CAS, FBW, and autopilot systems collectively enhance the stability, control, and performance of aircraft, reducing pilot workload and improving safety across a wide range of flight operations.

3.2 Regulatory Standards for Controllers Used in the Aeronautical Field

The aviation industry, both military and civil, relies on stringent regulatory standards to ensure the safety, reliability, and interoperability of equipment used in aircraft, including autopilot and other flight control systems. Among these standards, the United States Military Standards (MIL-STD), Federal Aviation Administration (FAA) Advisory Circulars (ACs), and NATO Standardization Agreements (STANAG) play crucial roles in setting the criteria for performance, testing, and interoperability of these systems. This chapter will discuss the key regulatory standards relevant to the controllers used in aeronautical applications, focusing on MIL-STD 810F, MIL-STD 461F, STANAG 4586, and FAA AC 25.1329-1C.

3.2.1 MIL-STD 810F: Environmental Engineering Considerations and Laboratory Testing

Overview

MIL-STD 810F, established by the U.S. Department of Defense, provides a comprehensive set of environmental test conditions and laboratory testing methodologies designed to ensure that military equipment can withstand the rigors of various environmental stresses throughout its service life. These tests cover a wide range of environmental factors, including vibration, acceleration, humidity, rain, and temperature extremes.

Purpose and Application

The primary goal of MIL-STD 810F is to simulate the environmental conditions that military equipment is likely to encounter during its operational life. By doing so, it ensures that the equipment remains functional and reliable under these conditions. The standard is not limited to military applications; non-defense organizations and industries also adopt it to demonstrate that their products meet similar rigorous criteria.

Testing Methodologies

MIL-STD 810F outlines specific test procedures, durations, and cycles to replicate the environmental stresses on equipment. These include:

- Vibration Testing: Ensures that equipment can withstand mechanical vibrations encountered during transportation and operation.
- Acceleration Testing: Verifies that equipment can endure forces encountered during rapid acceleration or deceleration.
- Humidity Testing: Assesses the equipment's resistance to prolonged exposure to high humidity levels.
- Temperature Testing: Evaluates the equipment's performance across a range of temperatures, including extreme heat and cold.

The results of these tests are analyzed to identify any design deficiencies or defects, which must be addressed to meet the standard's requirements. In the case of the UAV Navigation VECTOR autopilot, no redesign was necessary, validating the robustness of its design and quality control processes.

3.2.2 MIL-STD 461F: Electromagnetic Interference Control

Overview

MIL-STD 461F specifies the requirements for controlling the electromagnetic interference (EMI) characteristics of electronic, electrical, and electromechanical equipment and subsystems. This standard ensures that military equipment can operate without causing or being affected by EMI, which is critical for maintaining the integrity and performance of sensitive electronic systems.

Figure 3.7: MIL-STD 461F Logo

Purpose and Application

The standard aims to guarantee electromagnetic compatibility (EMC) among various subsystems and equipment used in military operations. It applies to equipment with electronic enclosures, discrete electrical interconnections, wiring harnesses, and power inputs derived from primary power sources.

Testing Methodologies

MIL-STD 461F includes several tests to evaluate the EMI characteristics of equipment:

- Radiated Emissions Testing: Measures the electromagnetic energy emitted by the equipment to ensure it does not interfere with other electronic systems.
- Conducted Susceptibility Testing: Assesses the equipment's ability to withstand EMI from external sources without performance degradation.

The UAV Navigation VECTOR autopilot successfully passed these tests, demonstrating its compliance with MIL-STD 461F and its ability to function effectively in environments with potential electromagnetic interference.

3.2.3 STANAG 4586: Standard Interfaces for NATO UAV Interoperability

Overview

STANAG 4586, developed by NATO, specifies the standard interfaces for Unmanned Aerial Vehicle (UAV) Control Systems (UCS) to ensure interoperability among different UAV systems used by NATO member countries. This standard addresses

the need for a common framework to enable seamless integration and operation of UAVs from various manufacturers.

Purpose and Application

The objective of STANAG 4586 is to define the interfaces required to achieve the necessary Level of Interoperability (LOI) for UAV systems, ensuring that they can be used interchangeably and cooperatively across different NATO operations. This standard is vital for joint missions, where multiple UAVs from different countries must operate together efficiently.

Functional Architecture and Interfaces

STANAG 4586 establishes a functional architecture for UCS, comprising several key elements:

- Air Vehicle (AV)
- Vehicle Specific Module (VSM)
- Data Link Interface (DLI)
- Core UCS (CUCS)
- Command and Control Interface (CCI)
- Human Computer Interface (HCI)
- Command and Control Interface Specific Module (CCISM)

These components and interfaces facilitate the communication and control of UAVs and their payloads, ensuring interoperability and efficient operation.

Levels of Interoperability (LOI)

STANAG 4586 identifies five LOIs to accommodate different operational requirements:

1. LOI 1: Indirect receipt and/or transmission of sensor product and metadata.
2. LOI 2: Direct receipt of sensor product data and metadata.
3. LOI 3: Control and monitoring of the UAV payload.
4. LOI 4: Control and monitoring of the UAV, excluding launch and recovery.

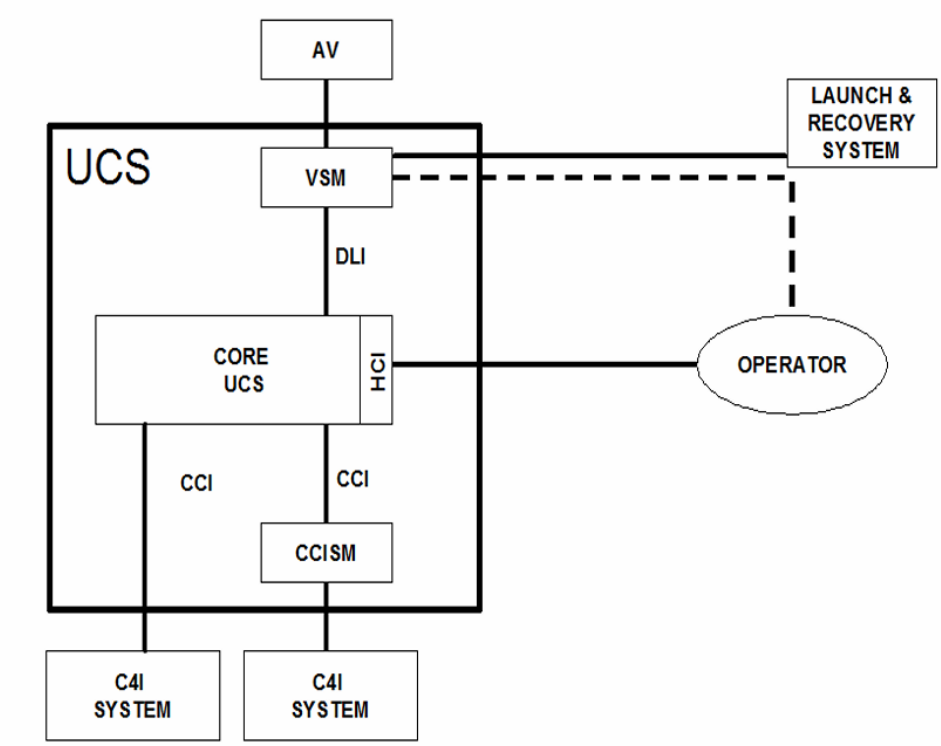

Figure 3.8: UCS Functional Architecture

5. LOI 5: Full control and monitoring, including launch and recovery.

By implementing these levels, NATO ensures that UAV systems can be tailored to specific mission needs while maintaining interoperability.

3.2.4 FAA AC 25.1329-1C: Approval of Flight Guidance Systems

Overview

FAA Advisory Circular (AC) 25.1329-1C provides guidance on the approval of flight guidance systems (FGS), which include autopilot functions, flight directors, and automatic thrust control functions. This AC is crucial for ensuring that these systems meet the airworthiness standards required for transport category airplanes.

Purpose and Application

The primary purpose of AC 25.1329-1C is to outline acceptable means for demonstrating compliance with the requirements of Title 14, Code of Federal Regulations

(CFR) 25.1329. This includes the design, testing, and certification of FGS to ensure they provide safe and reliable operation throughout the aircraft's operational envelope.

Key Aspects of Compliance

AC 25.1329-1C details various aspects of compliance, including:

- Autopilot Engagement and Disengagement: The criteria for safe engagement and disengagement of the autopilot, including the provision of appropriate alerts to the flight crew .
- Performance in Normal and Rare Conditions: The performance standards for FGS under normal and rare conditions, such as severe turbulence and significant wind gradients .
- Human Factors Evaluation: The importance of evaluating the human-machine interface to ensure that FGS operations are intuitive and do not impose excessive workload on the flight crew .
- Failure Conditions and Safety Assessment: Guidance on assessing the impact of potential failure conditions on the FGS and the aircraft, including pilot recognition and recovery from such conditions .

Testing Methodologies

The AC outlines specific testing methodologies for demonstrating compliance, which may include both flight tests and simulator evaluations. These tests are designed to validate the FGS's performance across a range of operational scenarios and configurations, ensuring that the system can safely handle both typical and adverse conditions .

The regulatory standards discussed in this chapter—MIL-STD 810F, MIL-STD 461F, STANAG 4586, and FAA AC 25.1329-1C—play essential roles in ensuring the reliability, safety, and interoperability of controllers used in aeronautical applications. These standards provide comprehensive guidelines for testing and evaluation, addressing both environmental and electromagnetic considerations, as well as the need for interoperable UAV control systems and robust flight guidance systems. Adherence to these standards is crucial for the successful deployment and operation of UAVs and other aeronautical systems in both military and civil contexts.

3.3 Specifications and Dynamics of Aircraft Models with Focus on the F-35

The study of aircraft models involves understanding their aerodynamic properties, structural dynamics, and control mechanisms. This chapter provides an in-depth analysis of various aircraft models with a special focus on the F-35 Lightning II. Using three key documents, we will explore the general aircraft dynamics, the specific technologies employed in the F-35, and mathematical models to simulate aircraft behavior under various conditions.

3.3.1 General Aircraft Dynamics

Equations of Motion

The dynamics of an aircraft can be described using the rigid body equations derived from Newton's laws. These equations account for forces (F) and moments (M) acting on the aircraft as it maneuvers through the atmosphere. The general equations of motion in the body axes reference frame are given by:

$$F_x = X - W\sin\theta = m(\dot{u} + qw - rv) \tag{3.1}$$
$$F_y = Y + W\cos\theta\sin\phi = m(\dot{v} + ru - pw) \tag{3.2}$$
$$F_z = Z + W\cos\theta\cos\phi = m(\dot{w} + pv - qu) \tag{3.3}$$
$$M_x = L = I_x\dot{p} + (I_z - I_y)qr - I_{xz}(\dot{r} + pq) \tag{3.4}$$
$$M_y = M = I_y\dot{q} + (I_x - I_z)rp - I_{xz}(p^2 - r^2) \tag{3.5}$$
$$M_z = N = I_z\dot{r} + (I_y - I_x)pq - I_{xz}(\dot{p} - rq) \tag{3.6}$$

These equations assume the aircraft is a rigid body, its mass distribution is symmetric, and the Earth's rotation and curvature are negligible.

Aerodynamic Forces and Moments

Aerodynamic forces and moments are crucial in determining aircraft behavior. For the Airbus A340-300, the lift (L), drag (D), and pitch moment (M) are expressed as follows:

$$L = C_L qS \tag{3.7}$$
$$M = C_M qSc \tag{3.8}$$
$$N = C_N qS \tag{3.9}$$

where q is the dynamic pressure, S is the wing area, and c is the chord length. The coefficients C_L, C_N, and C_M vary with the angle of attack, control surface deflections, and other factors.

The dynamics of an aircraft are often divided into longitudinal and lateral-directional components for analysis.

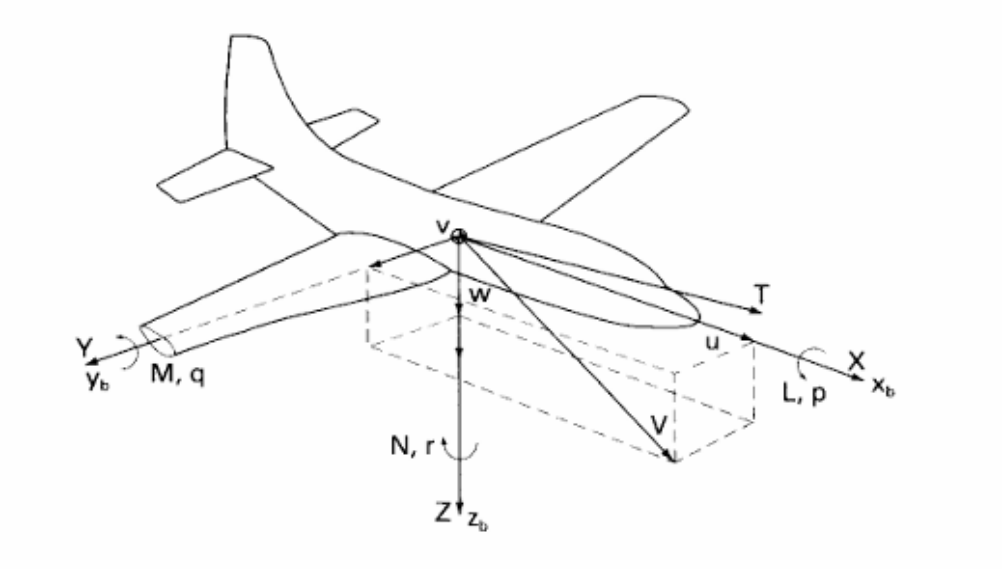

Figure 3.9: Dynamical model for aircraft motion

Longitudinal Dynamics Longitudinal dynamics involve the motion in the plane defined by the aircraft's forward velocity and vertical motion. For a simplified analysis using small perturbation theory, the equations of motion are linearized around an equilibrium state:

$$\Delta F_x = m\left(\dot{u} + qw_{eq}\right) \tag{3.10}$$

$$\Delta F_z = m\left(\dot{w} - qu_{eq}\right) \tag{3.11}$$

$$\Delta M_y = I_y \dot{q} \tag{3.12}$$

These equations are useful in studying the stability and control of aircraft during flight.

Lateral-Directional Dynamics Lateral-directional dynamics describe the motion involving side-to-side and rotational movements about the vertical axis. The linearized equations for these dynamics are:

$$\Delta F_y = m\left(\dot{v} + ru_{eq} - pw_{eq}\right) \tag{3.13}$$
$$\Delta M_x = I_x\dot{p} - I_{xz}\dot{r} \tag{3.14}$$
$$\Delta M_z = I_z\dot{r} - I_{xz}\dot{p} \tag{3.15}$$

These equations help in analyzing yaw and roll stability of the aircraft.

3.3.2 Mathematical Modeling of Aircraft

Basic Assumptions

The mathematical modeling of aircraft dynamics relies on several fundamental assumptions to simplify the complex reality of flight:

1. Constant Mass: The mass of the aircraft remains constant during the considered time interval.
2. Rigid Body: The aircraft is treated as a rigid body with no deformation.
3. Symmetric Mass Distribution: The mass distribution is symmetric relative to the aircraft's center line.
4. Neglect of Earth's Rotation and Curvature: The Earth's rotation and curvature are considered negligible for the time scales involved in flight dynamics.

Linearized Equations of Motion

For small perturbations around an equilibrium flight condition, the equations of motion can be linearized. This simplifies the analysis and control design. The linearized longitudinal equations are:

$$\Delta\dot{u} = \frac{1}{m}\left(-mg\sin\theta_0 + X_u\Delta u + X_w\Delta w + X_q\Delta q + X_{\delta_e}\Delta\delta_e\right) \tag{3.16}$$
$$\Delta\dot{w} = \frac{1}{m}\left(Z_u\Delta u + Z_w\Delta w + \left(Z_q + \frac{mU_0}{g}\right)\Delta q + Z_{\delta_e}\Delta\delta_e\right) \tag{3.17}$$
$$\Delta\dot{q} = \frac{1}{I_y}\left(M_u\Delta u + M_w\Delta w + M_q\Delta q + M_{\delta_e}\Delta\delta_e\right) \tag{3.18}$$
$$\Delta\dot{\theta} = \Delta q \tag{3.19}$$

where:

- Δu, Δw, Δq, and $\Delta\theta$ are perturbations in forward velocity, vertical velocity, pitch rate, and pitch angle, respectively

- m is the mass of the aircraft
- g is the acceleration due to gravity
- θ_0 is the equilibrium pitch angle
- U_0 is the equilibrium forward velocity
- X, Z, and M and their subscripts denote stability derivatives, which are partial derivatives of aerodynamic forces and moments with respect to the state variables
- δ_e is the elevator deflection.

Stability Derivatives

Stability derivatives are essential in understanding how changes in flight conditions affect aerodynamic forces and moments. They are typically obtained through a combination of analytical methods, wind tunnel tests, and flight experiments. For example:

$$X_u = \frac{\partial X}{\partial u}, \tag{3.20}$$
$$X_w = \frac{\partial X}{\partial w}, \tag{3.21}$$
$$X_q = \frac{\partial X}{\partial q}, \tag{3.22}$$
$$X_{\delta_e} = \frac{\partial X}{\partial \delta_e}, \tag{3.23}$$

$$Z_u = \frac{\partial Z}{\partial u}, \tag{3.24}$$
$$Z_w = \frac{\partial Z}{\partial w}, \tag{3.25}$$
$$Z_q = \frac{\partial Z}{\partial q}, \tag{3.26}$$
$$Z_{\delta_e} = \frac{\partial Z}{\partial \delta_e}, \tag{3.27}$$

$$M_u = \frac{\partial M}{\partial u}, \tag{3.28}$$

$$M_w = \frac{\partial M}{\partial w}, \tag{3.29}$$

$$M_q = \frac{\partial M}{\partial q}, \tag{3.30}$$

$$M_{\delta_e} = \frac{\partial M}{\partial \delta_e}, \tag{3.31}$$

where X, Z, and M represent aerodynamic forces and moments, and u, w, q, and δ_e are the perturbations in forward velocity, vertical velocity, pitch rate, and elevator deflection, respectively.

These derivatives allow for the construction of linearized models suitable for control system design and stability analysis.

3.3.3 F-35 Lightning II Technology Overview

General Characteristics

The F-35 Lightning II, developed by Lockheed Martin, is a fifth-generation multirole fighter designed to meet the needs of modern warfare. The F-35 is built in three variants: F-35A for conventional takeoff and landing (CTOL), F-35B for short takeoff/vertical landing (STOVL), and F-35C for aircraft carrier operations (CATOBAR). Each variant is equipped with advanced technologies and capabilities that ensure superior performance in diverse mission profiles. The F-35 features a blended wing body design that enhances its aerodynamic efficiency and stealth characteristics. Its airframe is constructed using advanced composite materials that reduce weight while maintaining structural integrity. The aircraft's internal weapons bays further enhance its stealth capabilities by minimizing radar cross-section.

1. **Stealth Capabilities**

   The F-35's stealth design incorporates a variety of technologies aimed at reducing its radar cross-section and infrared signature. The aircraft's shape, materials, and coatings are optimized to deflect radar waves and absorb radar energy, making it difficult for enemy radars to detect and track.

   Radar-Absorbing Materials The F-35 is coated with radar-absorbing materials (RAM) that significantly reduce its radar signature. These materials are applied to critical surfaces and edges, where radar reflections are most

likely to occur. The use of RAM, combined with the aircraft's stealthy shape, ensures a low observable profile.

Infrared Signature Management To manage its infrared signature, the F-35 utilizes an advanced cooling system that minimizes the heat emitted by its engine and other components. The engine exhaust is also designed to disperse heat quickly, reducing the aircraft's vulnerability to infrared-guided missiles.

2. **Advanced Avionics and Sensor Fusion**

The F-35 is equipped with a state-of-the-art avionics suite that provides pilots with unparalleled situational awareness. The avionics system integrates data from multiple sensors, including radar, infrared, and electronic warfare systems, to create a comprehensive picture of the battlefield.

AN/APG-81 AESA Radar The F-35's AN/APG-81 Active Electronically Scanned Array (AESA) radar offers superior detection and tracking capabilities. It can perform multiple functions simultaneously, such as air-to-air and air-to-ground targeting, electronic warfare, and intelligence gathering. The radar's advanced signal processing ensures high-resolution imagery and precise target identification.

Distributed Aperture System (DAS) The Distributed Aperture System (DAS) provides the F-35 with 360-degree situational awareness. It consists of multiple infrared cameras mounted around the aircraft, which detect and track incoming threats, such as missiles and aircraft. The DAS feeds real-time imagery to the pilot's helmet-mounted display, allowing for rapid threat identification and response.

Electro-Optical Targeting System (EOTS) The Electro-Optical Targeting System (EOTS) combines forward-looking infrared (FLIR) and laser targeting capabilities into a single sensor. The EOTS enables the F-35 to perform precision targeting and tracking of ground and airborne targets. It provides high-resolution imagery and accurate targeting data, enhancing the aircraft's effectiveness in both offensive and defensive operations.

3. **Propulsion and Performance**

Powered by the Pratt & Whitney F135 engine, the F-35 achieves supersonic speeds and exceptional agility. The F135 is the most powerful fighter engine in the world, providing the F-35 with unmatched thrust and efficiency.

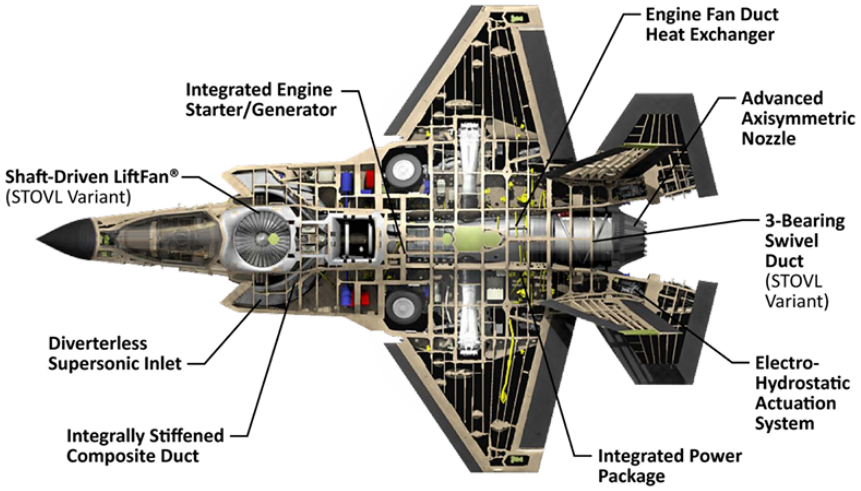

Figure 3.10: Propulsion System

Pratt & Whitney F135 Engine The F135 engine is a two-stage turbofan with an afterburner, capable of generating up to 43,000 pounds of thrust. Its advanced design includes features such as a three-stage fan, a six-stage compressor, and a low-emission combustor. The engine's high thrust-to-weight ratio and fuel efficiency contribute to the F-35's superior performance.

Short Takeoff/Vertical Landing (STOVL) The F-35B variant features a unique LiftFan system that enables short takeoff and vertical landing (STOVL) capabilities. The LiftFan, developed by Rolls-Royce, works in conjunction with the F135 engine to provide vertical thrust. This capability allows the F-35B to operate from short runways and amphibious assault ships, significantly enhancing its operational flexibility.

Range and Endurance The F-35 has an impressive range of over 1,200 nautical miles without external fuel tanks. Its internal fuel capacity and efficient engine design allow for extended missions without the need for aerial

refueling. The aircraft's endurance is further enhanced by its ability to carry a variety of mission-specific external fuel tanks and sensors.

Understanding the specifications and dynamics of aircraft, particularly advanced models like the F-35, is crucial for both design and operational efficiency. Through rigorous modeling and simulation, engineers can predict aircraft behavior, optimize performance, and ensure safety. This chapter highlighted the foundational principles and specific technological advancements that define modern aviation.

3.4 Development of the Autopilot System

This chapter provides an in-depth analysis of the development process of the autopilot system, which forms the cornerstone of my master's thesis. The chapter meticulously delineates the various stages undertaken to achieve the final outcome: a sophisticated controller designed to autonomously navigate an aircraft to a specified target point identified by coordinates (latitude and longitude) and a predetermined altitude, all the three variable are settable in a configuration file. The system enables the aircraft to execute a seamless takeoff, perform a precise turning maneuver to align with the designated point, and subsequently stabilize at the desired altitude, all within strict operational tolerances that ensure high accuracy.

The development of the controller was executed using the C++ programming language. The integration with the simulation environment was achieved by interfacing with Prepar3D, a widely used flight simulation software. Input control variables were extracted from Prepar3D, and the resultant output control variables were communicated back to Prepar3D, utilizing SimConnect as the communication interface. This setup allowed for a robust feedback loop, where the simulated aircraft within Prepar3D served as the plant for the controller's development and fine-tuning. Figure 3.11 illustrates a simplified example of the structure of the developed controller.

3.4.1 Data Acquisition and Transmission

The first step undertaken was to acquire data from Prepar3D using SimConnect and to transmit data back to Prepar3D through the same interface. The data acquisition component had already been implemented in the data acquisition module used for the wearable item project, discussed in detail in Chapter 2.3. This module was modified to capture the data relevant to the control variables of interest, which will be analyzed in detail later in this chapter.

To transmit data to Prepar3D, instead of using simulation variables directly, specific "events" were utilized to modify the positions of the aircraft's control

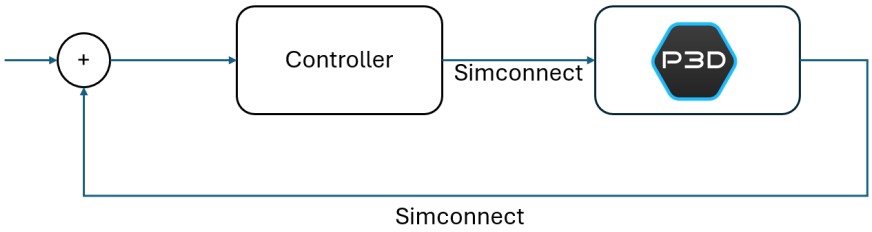

Figure 3.11: Simplified controller schematic

surfaces, such as ailerons and elevators. This approach provided a more flexible control mechanism, ensuring precise adjustments based on the controller's outputs. The variables and events used for controlling the aircraft will be specified and described in detail in the subsequent sections of this chapter.

This setup of data acquisition and transmission established a robust communication loop between the controller and the simulation environment, essential for the development of an effective autopilot system.

3.4.2 Development of Stabilization Controllers

The next step involved developing controllers to stabilize the aircraft's lateral and longitudinal attitudes, using bank and pitch as the control variables. This intermediate step was crucial for ensuring the aircraft could maintain a stable flight configuration before executing more complex maneuvers. Stabilizing the aircraft's orientation was essential for providing a solid foundation upon which more advanced control algorithms could be built, and for ensuring safe and reliable operation during various phases of flight.

To stabilize the bank and pitch, the aircraft's bank and pitch angles were used as the input variables for the controllers. The ailerons and elevator were manipulated to control these variables. The desired bank and pitch angles for stabilization were set to 0°, indicating level flight. The type of controller developed for this purpose was a PID controller.

For tuning the PID parameters, the Ziegler-Nichols method was initially employed. However, the results obtained were not satisfactory, leading to manual tuning. The manual tuning process involves first adjusting the proportional (P) gain to achieve a response with a steady oscillation. Next, the integral (I) gain is adjusted to eliminate any residual steady-state error, and finally, the derivative (D) gain is fine-tuned to dampen the oscillations and improve stability. This sequential

tuning approach helps to systematically address different aspects of the controller's performance, ensuring a balanced and responsive control system.

In practice, two separate controllers were developed: one for stabilizing the bank and another for stabilizing the pitch, each with distinct gain values. As previously mentioned, bank and pitch were used as input variables. Additionally, to achieve more accurate performance for the PID controller, the angular rates of bank and pitch were used. The derivative gain utilized these measurements directly rather than deriving them from the bank and pitch angles.

As previously mentioned, bank and pitch were used as input variables, and the output was a numerical value to actuate the ailerons (for bank control) and the elevator (for pitch control). To manipulate the ailerons and elevator, SimConnect events were used. Both control surfaces accept values ranging from -16383 to +16383, where the minimum value corresponds to the maximum deflection in one direction, and the maximum value corresponds to the maximum deflection in the opposite direction ($\pm 25°$ for the ailerons and $\pm 30°$ for the elevator). Figure 3.12 and Figure 3.13 illustrates the control schema of the implemented controllers.

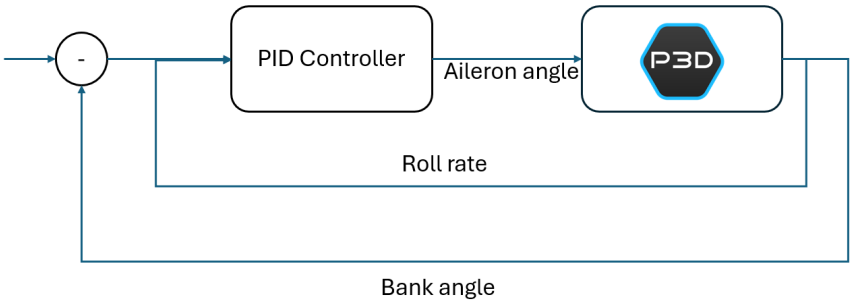

Figure 3.12: Bank controller schematic

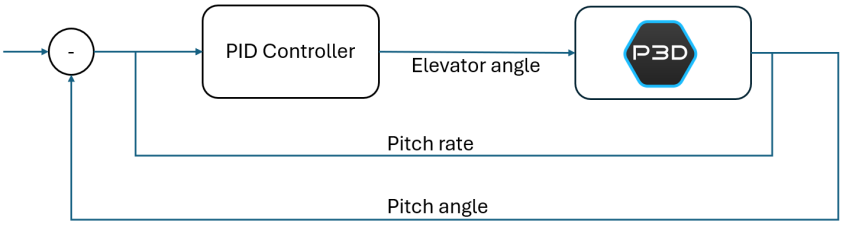

Figure 3.13: Pitch controller schematic

The complete code for these controllers is available in Appendix B of this thesis.

3.4.3 Development of More Complex Control Systems

The subsequent step involved developing more complex control systems that, when integrated, would form the final autopilot system. Specifically, these steps facilitated the implementation of the takeoff system, the controller for executing turns to align with the desired point, and the controller for achieving and maintaining the desired altitude. We will examine each of these systems in detail, analyzing their functionality and underlying principles.

Takeoff System

The takeoff system is relatively simple. In real-world applications, such systems do not exist; takeoff operations are still manually performed by pilots during flights. However, these operations follow standard procedures, and the takeoff system automates these operations. The operations performed by the system are:

- Disengage the parking brake
- Set the throttle lever to full throttle
- Once the aircraft reaches 200 knots, pitch up to achieve a 15° pitch angle
- Retract the landing gear

The automation of these steps ensures a smooth and consistent takeoff process. By following these standard procedures, the system replicates the actions a pilot would take, ensuring the aircraft achieves a stable climb after takeoff. Disengaging the parking brake allows the aircraft to begin its takeoff roll. Setting the throttle to full power ensures that the aircraft has sufficient thrust to accelerate down the runway. Pitching up at 200 knots ensures that the aircraft lifts off the ground at the appropriate speed, and achieving a 15° pitch angle helps establish a safe climb rate. Retracting the landing gear reduces drag and allows the aircraft to continue climbing efficiently.

In practice, this automated takeoff system can greatly assist in scenarios where precise control is needed, such as in unmanned aerial vehicles (UAVs) or in testing environments where consistent takeoff performance is crucial. By automating the takeoff process, the system can ensure that each takeoff is executed with the same precision and adherence to standard operating procedures, thus improving overall safety and reliability.

This foundational takeoff system serves as a critical building block for the overall autopilot system, enabling subsequent phases of flight control to be executed more effectively.

Turn Maneuver

To align the aircraft with the desired point, a controller was developed to enable the aircraft to execute a turn maneuver. Turning is a complex operation in aviation, requiring precise control and coordination. Specifically, two controllers were developed for this purpose to facilitate a comparative analysis between two different approaches. The first controller uses the difference between the azimuth of the target point and the aircraft's heading relative to true north as a reference. The second controller executes a standard turn. The control schemes for these controllers are illustrated in Figures 3.14 and 3.15. It can be seen from the control diagrams that both include a filter on the heading angle to determine when the controller should cease operation, with a tolerance of ± 0.005 radians.

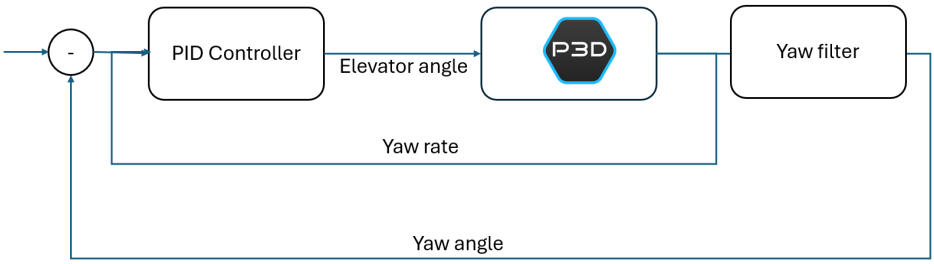

Figure 3.14: Turn Maneuver controller

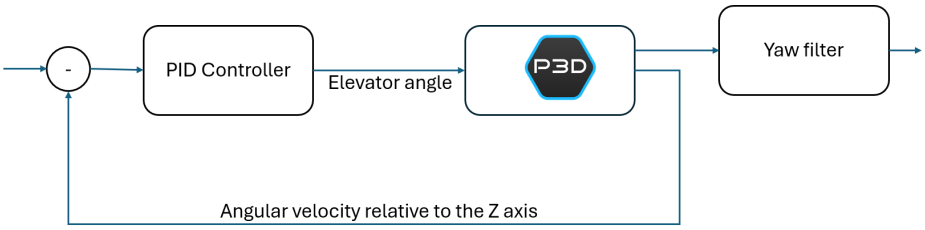

Figure 3.15: Standard Turn Maneuver controller

The first controller sets the bank angle to 40°, either to the right or left depending on the location of the target point, utilizing the previously developed bank controller but setting the reference angle to 40° instead of 0°. This adjustment allows the aircraft to achieve the necessary roll to initiate the turn. To execute the turn, a PID controller was developed, with the aircraft's heading and the yaw rate as the control variables. The derivative component of the controller uses the yaw rate

directly, enhancing the responsiveness and accuracy of the control action. The reference angle, which is the angle between the target point, the aircraft, and true north, is calculated using the azimuth formula:

$$\text{Azimuth} = \arctan 2\left(\sin(\Delta\lambda)\cdot\cos(\phi_2), \cos(\phi_1)\cdot\sin(\phi_2) - \sin(\phi_1)\cdot\cos(\phi_2)\cdot\cos(\Delta\lambda)\right)|$$

where ϕ_1 e ϕ_2 are the latitudes, and $\Delta\lambda$ is the difference the longitude between the two points.

The actuators used to control the yaw are the elevators. The control system continuously adjusts the elevators to maintain the desired yaw rate and align the heading with the calculated azimuth. Additionally, a filter is implemented to stop the control action once the aircraft is aligned with the desired point. This filter ensures that minor deviations do not cause unnecessary adjustments, maintaining stability during the maneuver.

The tuning of the controller was performed using the same approach as described for the bank and pitch controllers in the previous section. The tuning was specifically adjusted to ensure a slow control action to avoid any overshoot, which is critical in this application to maintain stability and precision. By fine-tuning the proportional, integral, and derivative gains, the controller achieves a balance between responsiveness and smoothness, essential for a coordinated and stable turn.

Once the aircraft is aligned with the desired point, it returns to a horizontal attitude. This is accomplished by changing the bank controller's reference angle from 40° back to 0°. This adjustment gradually rolls the aircraft back to level flight, ensuring a smooth transition and preventing abrupt changes in attitude. The system's ability to manage these transitions seamlessly enhances the overall reliability and effectiveness of the autopilot.

With regard to the standard turn, the angular velocity about the aircraft's vertical axis is used as the control variable. Consequently, the aircraft performs a turn at a constant angular velocity of 3 degrees per second. The control scheme is similar to the one previously analyzed. A PID controller has been developed, and the tuning was performed in the same manner. Additionally, a filter is applied to the yaw angle to interrupt the control action. Let us now analyze in more detail what constitutes a standard turn.

A standard turn is a fundamental maneuver in aviation where the aircraft is flown through a curved flight path while maintaining balanced forces. This ensures that the aircraft does not slip or skid, resulting in a smooth and efficient turn. The standard turn is achieved by banking the aircraft, which requires a balance between the lift generated by the wings and the centripetal force needed to follow the curved trajectory.

The physics of a standard turn involve several key parameters and equations. The primary forces acting on the aircraft during a turn include the lift L, weight W, and centripetal force F_r. For a turn to be coordinated, the resultant lift must be inclined to provide the necessary centripetal force.

Vertical Force Equilibrium The lift L is inclined at a bank angle ϕ. The vertical component of the lift must equal the weight of the aircraft:

$$L\cos\phi = W$$

$$\cos\phi = \frac{W}{L}$$

Horizontal Force Equilibrium The horizontal component of the lift provides the centripetal force required for the turn:

$$L\sin\phi = F_r$$

Given the centripetal force $F_r = \frac{mV^2}{R}$, where m is the mass of the aircraft, V is the velocity, and R is the radius of the turn, we have:

$$L\sin\phi = \frac{mV^2}{R}$$

Load Factor (n) The load factor n is defined as the ratio of the lift to the weight:

$$n = \frac{L}{W}$$

Combining the above equations, we get:

$$n\cos\phi = 1$$

$$n = \frac{1}{\cos\phi}$$

This implies that as the bank angle ϕ increases, the load factor n also increases. The load factor impacts the radius of the turn and the stall speed of the aircraft.

Turn Radius and Rate The radius R of the turn can be derived from the centripetal force equation:

$$R = \frac{V^2}{g\tan\phi}$$

The rate of turn ω (angular velocity) is given by:

$$\omega = \frac{g \tan \phi}{V}$$

Thus, the turn radius and rate depend on the bank angle ϕ and the velocity V of the aircraft.

Higher bank angles increase the load factor, which in turn increases the stall speed. For example, if an aircraft stalls at 50 m/s in level flight, in a 60° bank (n = 2), the stall speed increases to:

$$V_{stall_{turn}} = V_{stall}\sqrt{n} = 50\sqrt{2} \approx 70.7 \text{ m/s}$$

To achieve the minimum turn radius and maximum turn rate, it is advantageous to fly at the lowest possible speed for the given load factor. However, the structural limits of the aircraft (maximum load factor n_{max}) must be considered.

As the load factor increases, the drag increases, requiring more thrust to maintain speed. The available thrust must be sufficient to counteract the increased drag, especially at higher load factors and bank angles.

The standard turn is a complex maneuver requiring precise control of bank angle, speed, and power. Understanding the underlying physics and performing accurate calculations are essential for ensuring safe and efficient turns in various flight conditions.

Altitude control

To achieve and maintain a desired altitude, an algorithm has been developed to precisely control the aircraft's altitude. This approach reutilizes the previously implemented pitch control algorithm, making necessary adjustments to the reference input in order to effectively reach the target altitude. Specifically, the pitch controller's reference is dynamically modified based on the aircraft's current altitude during the ascent phase. As the aircraft gradually approaches the desired altitude, the reference input is smoothly decreased to ensure a stable and safe ascent, minimizing abrupt changes that could affect the aircraft's performance.

Similarly, if the target altitude is lower than the current altitude, the reference is adjusted accordingly to facilitate a controlled and stable descent. This gradual adjustment mechanism helps in maintaining the aircraft's stability and ensures passenger comfort by avoiding sudden altitude changes.

The developed controller demonstrates excellent performance, maintaining the desired altitude within a tolerance range of approximately ±10 feet. Such precision in altitude control is crucial for various flight phases, including cruising, where maintaining a consistent altitude ensures optimal fuel efficiency and flight safety.

3.5 Complete System

The complete system represents an integrated solution combining the components discussed in Chapter 3.4.3. The system operates as follows: during takeoff, it employs the previously described implementation to manage the initial phase of flight. Once the aircraft has reached an altitude of 7,000 feet and a speed of at least 400 knots, it transitions to a stabilized flight phase. At this point, the system activates the turn maneuver controller to manage any necessary directional changes.

Subsequently, the altitude control algorithm takes over to ensure the aircraft reaches and maintains the desired altitude with precision. This step involves dynamically adjusting the pitch control reference to achieve a stable ascent or descent, as required. The combined operation of these controllers ensures smooth transitions and maintains the aircraft's performance within specified tolerances.

The code implementing these systems is detailed in Appendix C of the thesis. This code has been designed with configurability in mind, allowing users to set parameters such as desired coordinates and target altitude through a configuration file. This feature provides flexibility and adaptability, making it possible to tailor the system's behavior to different flight scenarios and requirements.

Overall, this integrated approach demonstrates a robust solution for managing complex flight operations, combining several control mechanisms to enhance stability, precision, and adaptability in various flight phases.

Chapter 4

Simulation Results

In this chapter, the tests that were conducted will be presented, detailing how they were performed, how the data were collected, the results of the tests, and considerations regarding the obtained results. Multiple tests were carried out for each component and module of the final system, which were subsequently integrated and tested gradually. The strategy used to conduct the tests is the Software-in-the-Loop (SIL) strategy.

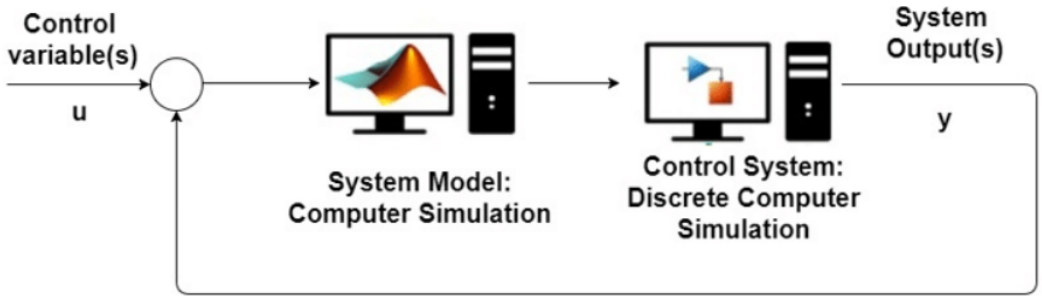

Figure 4.1: SIL schematic

Software-in-the-Loop (SIL) testing is a method where the software components of a system are tested within a simulated environment. This approach allows for the validation of the software's functionality and performance under controlled conditions, prior to integration with the actual hardware. SIL testing provides several advantages that are critical in the development of complex systems.

Firstly, SIL enables early detection of software issues. By simulating the operational environment, developers can identify and address bugs, performance bottlenecks, and integration problems at an early stage. This early intervention reduces the risk of encountering critical issues during later stages of development, which can be more costly and time-consuming to resolve.

Secondly, SIL testing supports iterative development. As software components

are developed and refined, they can be repeatedly tested within the simulated environment. This iterative process allows developers to continuously improve the software, ensuring that each iteration brings the system closer to the desired performance and reliability standards.

Moreover, SIL testing allows for comprehensive testing scenarios that might be impractical or impossible to replicate with actual hardware. For instance, extreme conditions, rare edge cases, and failure modes can be simulated to assess how the software handles such situations. This thorough testing ensures that the software is robust and capable of maintaining functionality under a wide range of conditions.

Additionally, SIL provides a safe testing environment. Testing in a simulated environment mitigates the risk of damaging physical hardware or causing unsafe conditions. This is particularly important for systems that interact with critical or hazardous environments, where testing failures could have severe consequences.

Finally, SIL testing facilitates better resource management. By leveraging simulation, developers can conduct extensive testing without the need for multiple physical prototypes, reducing costs and development time. This efficiency allows for more frequent testing cycles, leading to faster development and higher-quality software.

In conclusion, SIL testing is a powerful tool in the software development process, offering early issue detection, iterative improvement, comprehensive scenario testing, safety, and resource efficiency. By employing SIL testing, developers can ensure a more reliable and robust final product.

4.1 Data collection

To evaluate the performance of the various control units, an extensive series of tests was conducted. These tests aimed to verify the correct functionality of each controller and to ensure that the control units operated within the expected parameters. The collected data was essential for analyzing the behavior of each unit under different conditions and for conducting comparative analyses.

The data acquisition process was carried out using Simconnect. The data was sampled at a frequency of 10 Hz, resulting in the collection of one data point every 100 milliseconds. This sampling rate was chosen to provide a detailed temporal resolution that captures the dynamic responses of the control units effectively.

Once collected, the data was stored in files with a .csv extension. This format was selected for its versatility and ease of use, allowing for straightforward manipulation and analysis using various data processing tools. The .csv format also facilitated the generation of graphs and visual representations, which are crucial for interpreting the performance metrics and identifying trends or anomalies.

The specific data points gathered during the tests varied depending on the

control unit being evaluated. However, the core parameters typically included the following:

Table 4.1: Collected Data and Corresponding Units

Parameter	Unit of Measure
Bank	Radians
Pitch	Radians
Heading	Radians
Desired Heading	Radians
Plane Latitude	Radians
Plane Longitude	Radians
Altitude	Feet (ft)
Airspeed	Knots (kts)
Angular Velocity	Radians per second (1/s)

Each of these parameters provided insights into different aspects of the control unit's performance, such as its ability to maintain stability, accuracy in following a desired trajectory, and responsiveness to control inputs.

Overall, the systematic collection and analysis of this data were fundamental to understanding the capabilities and limitations of each control unit, enabling informed decisions to be made regarding their potential deployment in real-world applications.

4.2 Data analysis

The data collected during the testing of the various control modules will now be analyzed. The data will be presented in the same order as outlined in Chapter 3.4. The testing environment was consistently maintained; each time a test was conducted, the simulation was reset to ensure uniform conditions.

The specific environment in which the control modules were tested is as follows:

- Latitude: 47.64210°
- Longitude: -122.13010°
- Pitch: 0.0°
- Bank: 0.0°
- Heading: 0.0°
- Initial speed: 550 knots

- Altitude: 7000 feet
- Throttle position: 60%

In some tests, certain parameters may differ due to practical testing requirements. Throughout the analysis, any modifications to these parameters will be explicitly specified when necessary to execute specific tests.

4.2.1 Test of Stabilization Controllers

In this section, we will analyze the tests conducted for the stabilization controllers of the bank and pitch, presenting relevant graphs and providing detailed considerations. We will begin with the tuning of the controllers, which was performed using a systematic approach. Specifically, the PID controller tuning was executed by sequentially adjusting the proportional (P), integral (I), and derivative (D) gains. The process began by tuning the proportional gain to achieve a satisfactory level of responsiveness. Once an appropriate P value was established, the integral gain was adjusted to eliminate steady-state error. Finally, the derivative gain was fine-tuned to minimize overshoot and improve the stability of the system, ensuring a balanced and robust control performance. The tests for the bank and pitch controllers were conducted separately. For the bank, the initial condition was set to 30°, while for the pitch, the initial condition was set to -20°. This approach allowed for precise tuning and evaluation of each controller's performance under specific starting conditions.

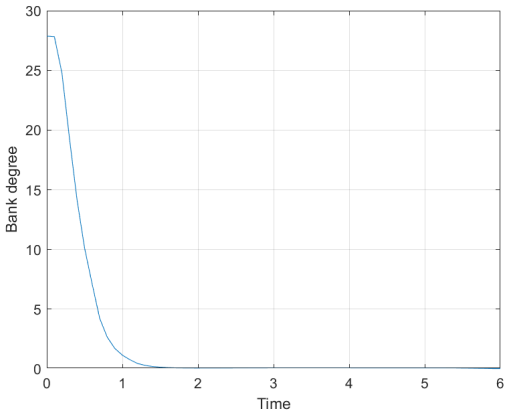

Figure 4.2: Bank P controller

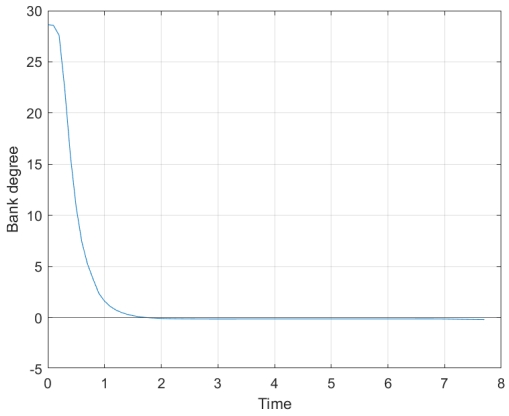

Figure 4.3: Bank PI controller

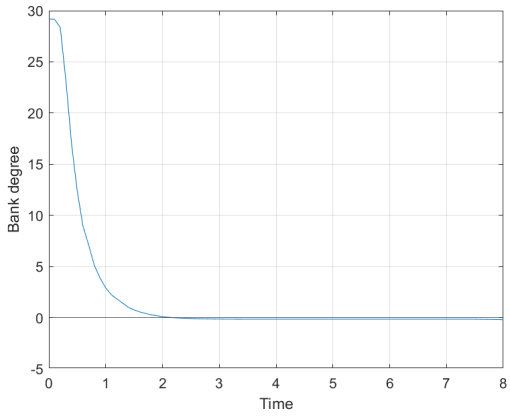

Figure 4.4: Bank PID controller

Additionally, it is useful to compare the developed controller with the existing system on the aircraft under the same initial conditions, specifically with a bank angle of 30° and a pitch angle of -20°.

It can be observed that the developed controller not only reaches the desired bank and pitch angles more quickly, but it also demonstrates a higher level of precision. This enhanced performance is particularly evident in its ability to achieve the target angles without experiencing significant overshoot. The absence of excessive overshoot indicates that the controller is more stable and reliable, effectively reducing the potential for oscillations or deviations from the desired trajectory. Consequently, the developed controller ensures smoother and more accurate control, which is critical in maintaining optimal aircraft performance and safety.

Further tests were conducted using varying initial speeds to thoroughly assess the behavior of the controller under different flight conditions. The results indicate that as the speed increases, there is a corresponding increase in overshoot. However, it is important to note that this increase in overshoot remains within acceptable limits and does not compromise the stability of the system. The controller effectively manages this behavior, ensuring that the overshoot is controlled and does not become excessive, even when operating at supersonic speeds. This demonstrates the robustness of the controller in handling high-speed scenarios, maintaining reliable performance across a wide range of operating conditions.

Finally, the last series of tests were conducted under adverse weather conditions. These tests included scenarios with wind coming from the left side of the aircraft, as well as the presence of turbulence. Multiple tests were carried out to evaluate the controller's performance under varying levels of turbulence, ranging from moderate to severe. The results demonstrate that the developed controller plays a crucial role in maintaining the stability of the aircraft during flight. Despite the challenging

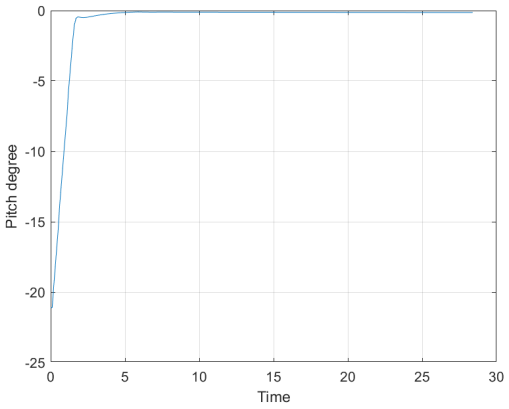

Figure 4.5: Pitch P controller

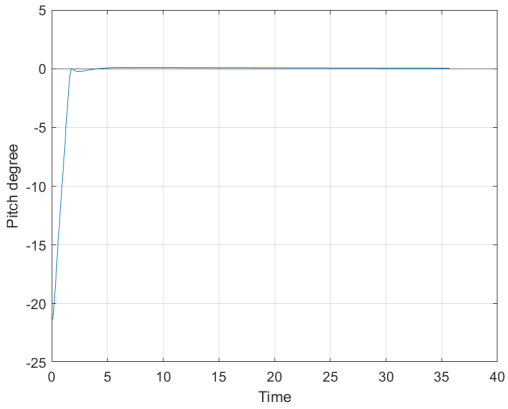

Figure 4.6: Pitch PI controller

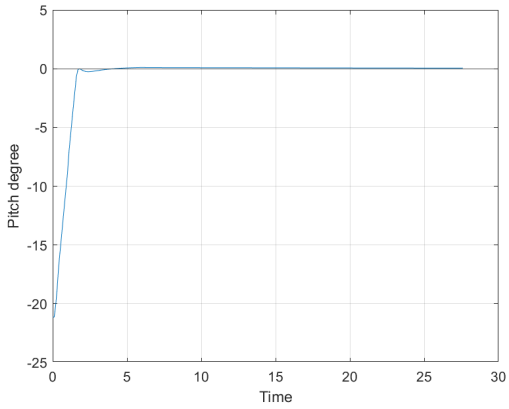

Figure 4.7: Pitch PID controller

conditions, the controller effectively contributes to keeping the aircraft stable and well-controlled, highlighting its robustness and effectiveness in managing adverse environmental factors. In Figures 4.18, 4.19, 4.20 and 4.21, it can be observed how the controller aids in maintaining the aircraft's bank and pitch angles close to 0°. This is evident from the fact that the average bank and pitch angles are 0.04° and -0.04°, respectively, when the controller is active. In contrast, with the controller deactivated, the average bank and pitch angles are 11.9° and 3.9°, respectively.

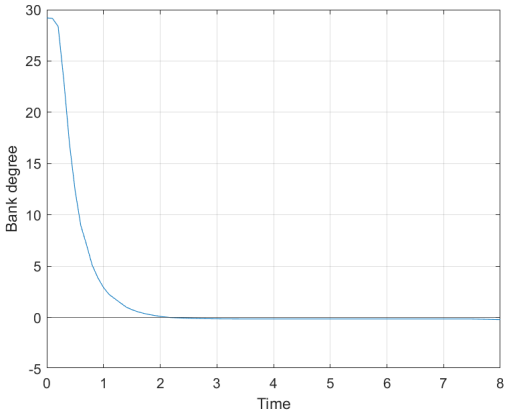

Figure 4.8: Bank PID controller

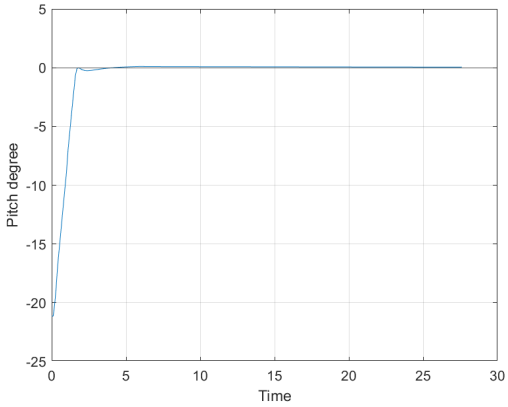

Figure 4.9: Pitch PID controller

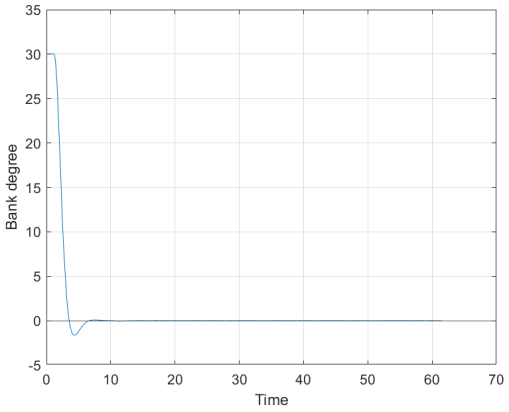

Figure 4.10: Autopilot bank

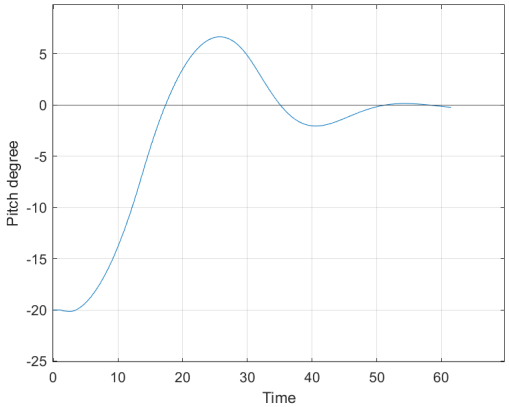

Figure 4.11: Autopilot pitch

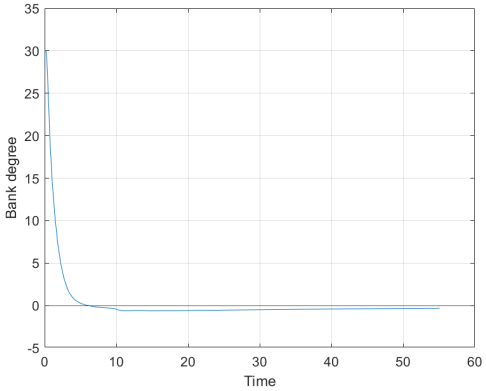

Figure 4.12: Bank PID controller 200 knts

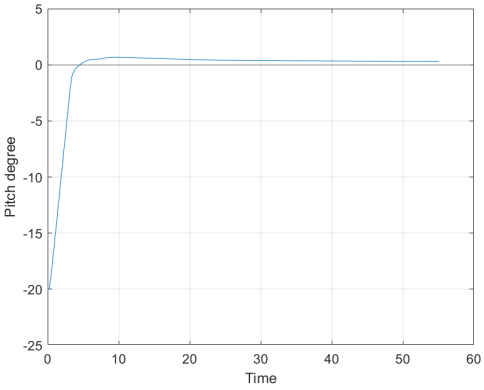

Figure 4.13: Pitch PID controller 200 knts

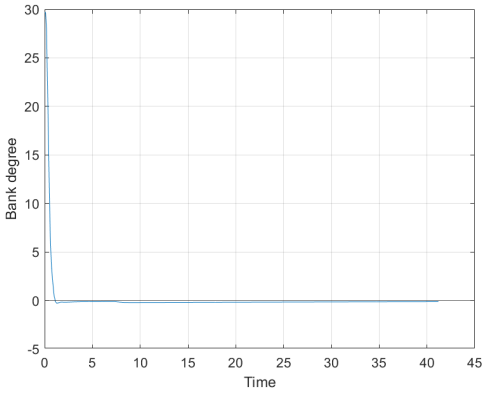

Figure 4.14: Bank PID controller 600 knts

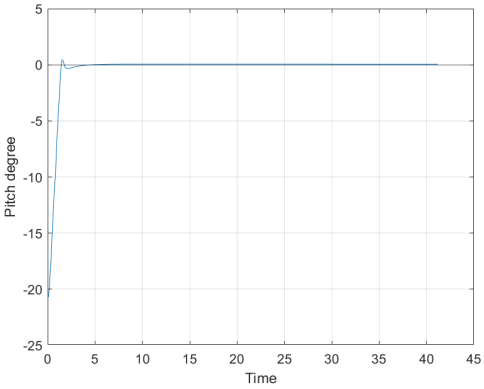

Figure 4.15: Pitch PID controller 600 knts

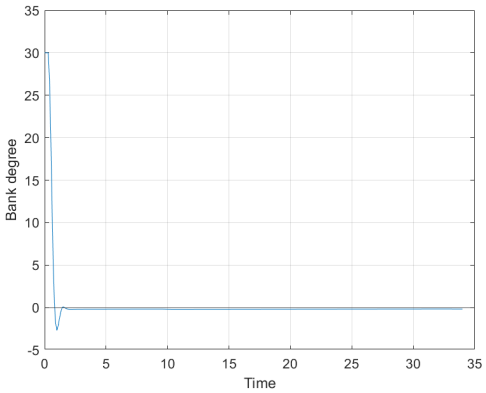

Figure 4.16: Bank PID controller 800 knts

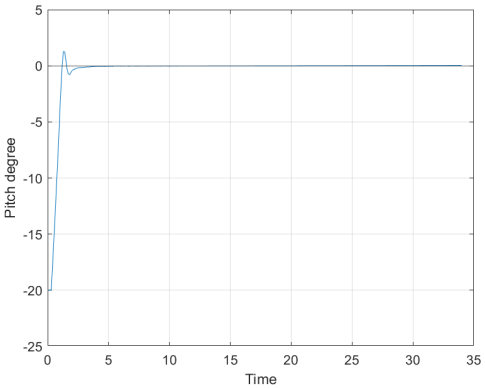

Figure 4.17: Pitch PID controller 800 knts

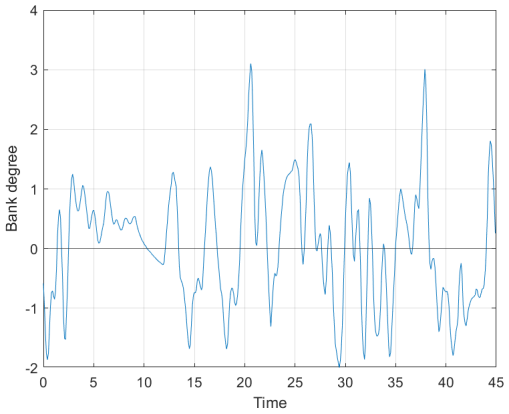

Figure 4.18: Bank severe turbolence with controller

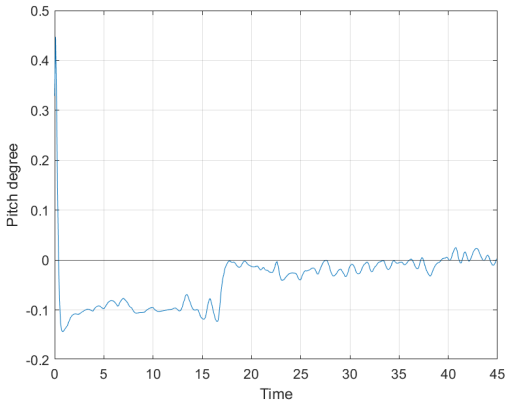

Figure 4.19: Pitch severe turbolence with controller

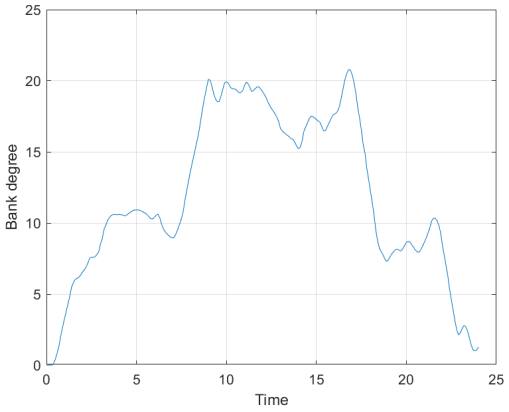

Figure 4.20: Bank severe turbolence without controller

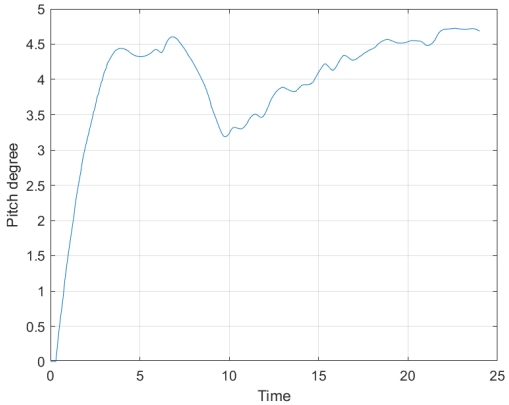

Figure 4.21: Bank severe turbolence without controller

4.2.2 Test of take off system

Subsequently, tests were conducted on the individual components of the complete autopilot system to evaluate their performance in isolation. Specifically, for the takeoff system, a series of tests were carried out to assess its functionality and effectiveness. During these tests, detailed graphs were generated to visualize key parameters, including the bank angle, pitch angle, altitude, and airspeed. These graphs provide valuable insights into the system's behavior during takeoff, allowing for a comprehensive analysis of how each component contributes to the overall performance of the autopilot system. The test was conducted at Washington Dulles International Airport.

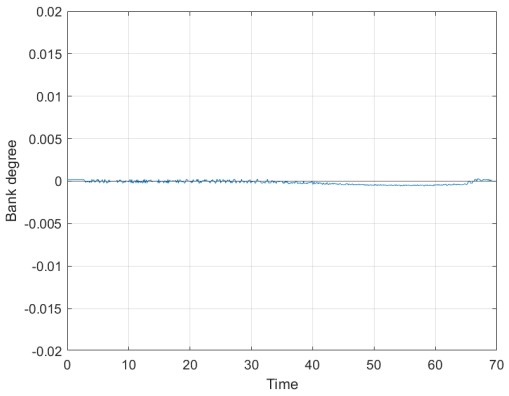

Figure 4.22: Bank angle

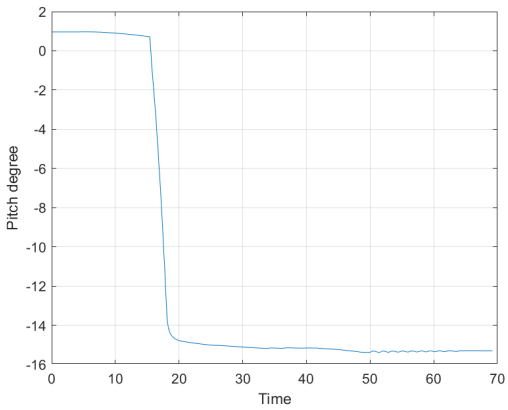

Figure 4.23: Pitch angle

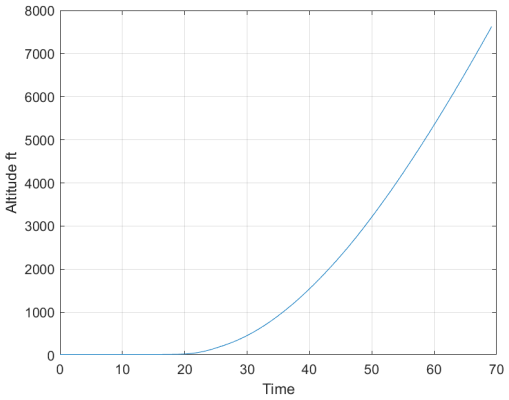

Figure 4.24: Altitude

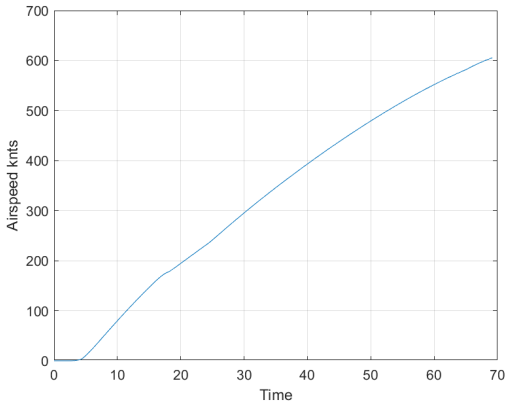

Figure 4.25: Airspeed

4.2.3 Test of altitude controller

Regarding the altitude control system, a series of tests were conducted to evaluate its performance in both altitude increase and decrease scenarios. These tests were performed under the same conditions outlined at the beginning of the chapter, which involved an altitude change of 4,000 feet. Specifically, the tests assessed the system's ability to transition from an initial altitude of 7,000 feet to either 11,000 feet or 3,000 feet.

The results of these tests indicate that the altitude is achieved with a margin of ±5 feet, and the system maintains the altitude within this range at steady state. This performance is consistent with the reference value, with oscillations limited to ±5 feet. This behavior is also observable in the pitch angle graph, where the pitch angle shows oscillations around the steady-state value, reflecting the system's fine-tuning capability in altitude control. The results are documented and can be reviewed in the subsequent figures, which provide a detailed visualization of the system's behavior and effectiveness in managing the desired altitude adjustments during both climb and descent phases.

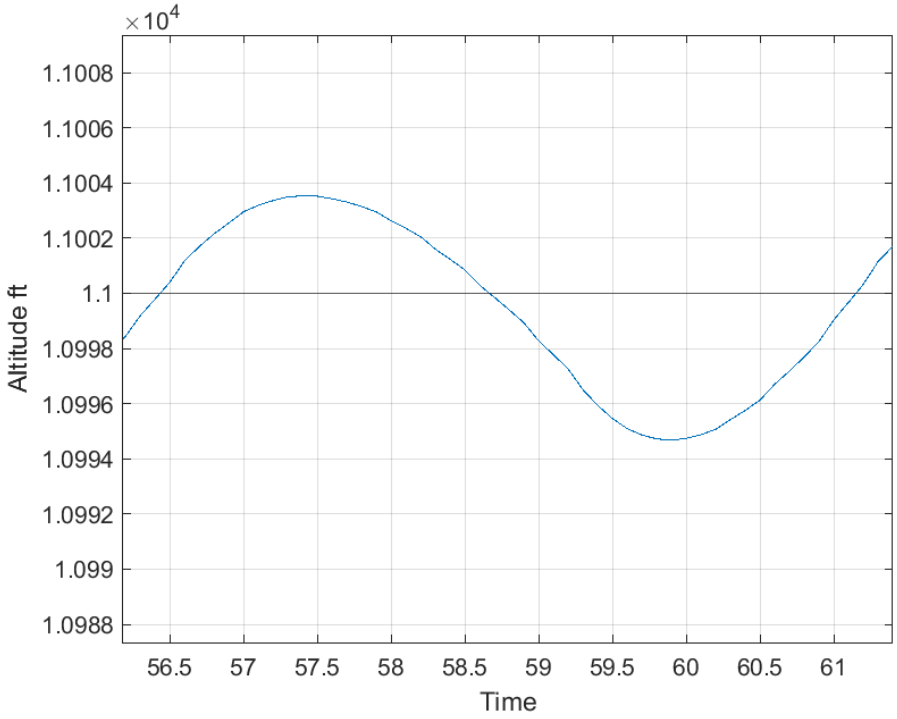

Figure 4.26: Detail on the oscillation of the altitude

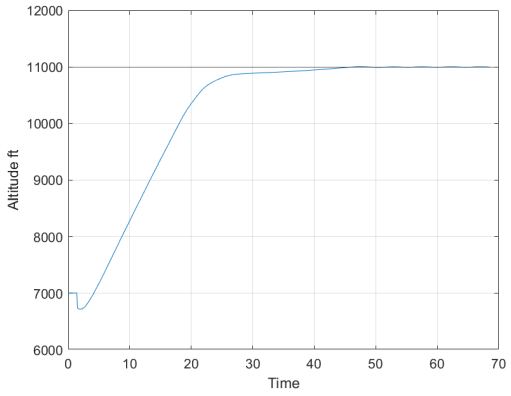

Figure 4.27: Altitude 11000 ft

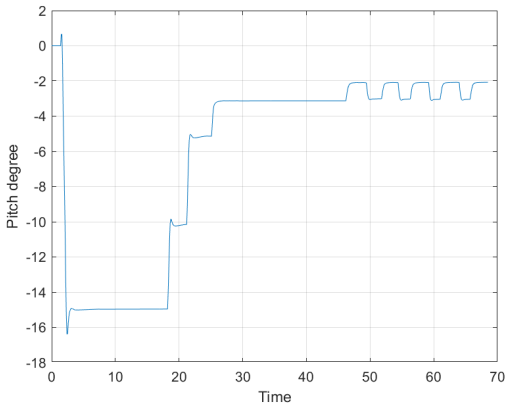

Figure 4.28: Pitch 11000 ft

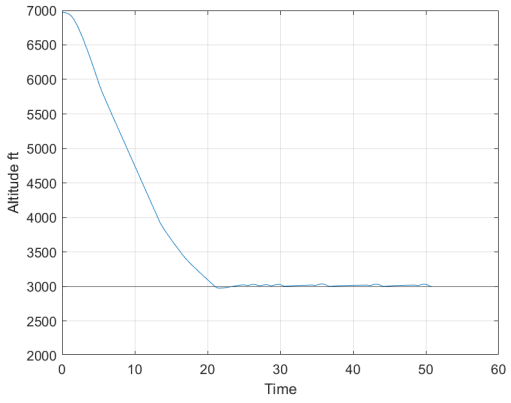

Figure 4.29: Altitude 3000 ft

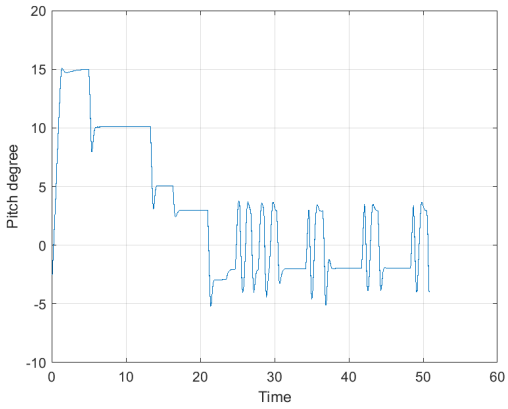

Figure 4.30: Pitch 3000 ft

4.2.4 Test of the turn maneuver

The next system tested was related to the maneuvering of turns. As previously mentioned in Section 3.4.3, two systems were developed to facilitate the execution of turn maneuvers. In this section, we will analyze the results of the tests conducted and provide a detailed comparison between the two methodologies. This comparative analysis will focus on evaluating the performance and effectiveness of each system in executing turn maneuvers, highlighting their respective strengths and weaknesses based on the test outcomes. The tests were conducted under the same initial conditions outlined at the beginning of the chapter. The target point for the maneuvers was specified with coordinates of latitude 47.64210° N, longitude 145.13012° W, and an altitude of 11,000 feet. The tests were terminated before the aircraft reached the exact target point, as achieving this would have required an extended duration. However, the tests were concluded only after the aircraft had aligned with the desired point and reached the target altitude. This approach ensured that the maneuvers were conducted effectively and provided meaningful insights into the system's performance without necessitating the completion of the full trajectory. Regarding the tests conducted for the turn maneuver, it can be observed that during the execution of the turn, the aircraft experiences a gradual loss of altitude, descending to approximately 4,500 feet. This loss of altitude occurs as the aircraft maneuvers, but the altitude is subsequently regained once the aircraft has aligned with the desired point and the altitude control system is reactivated. This behavior highlights a temporary instability during the turn, which could be a concern in certain flight scenarios. In contrast, this phenomenon does not occur with the standard turn, where the aircraft maintains its altitude more consistently throughout the maneuver. As a result, the standard turn emerges as the more reliable and safer option, offering a more stable and predictable performance in maintaining the desired flight parameters during the turn. Regarding the standard turn, multiple tests were conducted with different bank angles, specifically at 30°, 35°, 40°, and 45°. The results of these tests indicate that as the bank angle increases, the altitude gained during the maneuver decreases. This behavior can be attributed to the fact that at lower bank angles, the aerodynamic surfaces of the elevators are more effective in generating the necessary lift to increase the aircraft's altitude. In contrast, as the bank angle increases, the lift component acting vertically is reduced, limiting the aircraft's ability to gain altitude during the turn. This phenomenon is a direct consequence of the redistribution of lift forces as the aircraft banks more steeply, thereby reducing the efficiency of the elevators in contributing to altitude gain.

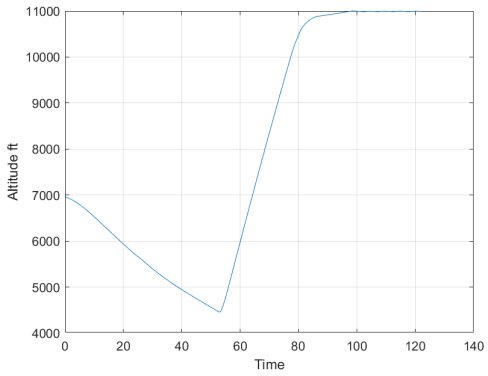

Figure 4.31: Altitude turn maneuver

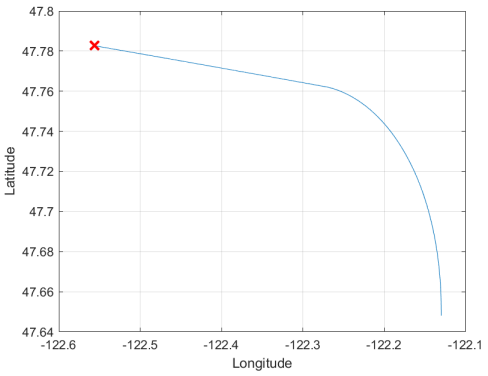

Figure 4.32: Trajectory turn maneuver

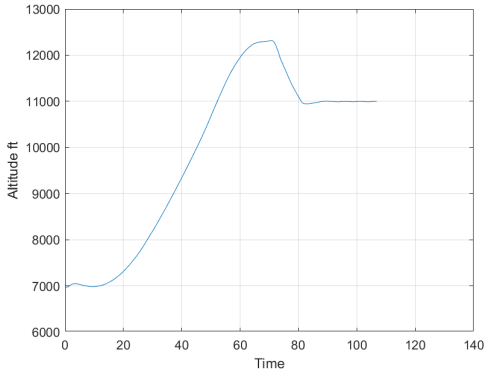

Figure 4.33: Altitude standard turn maneuver bank=30°

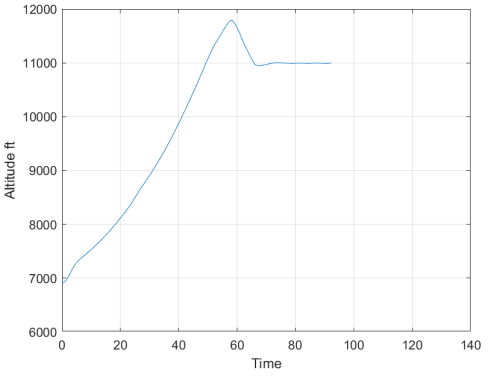

Figure 4.34: Altitude standard turn maneuver bank=35°

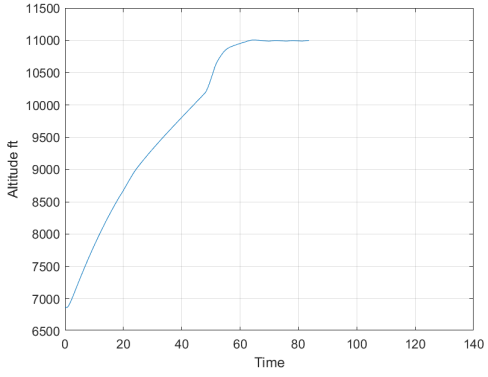

Figure 4.35: Altitude standard turn maneuver bank=40°

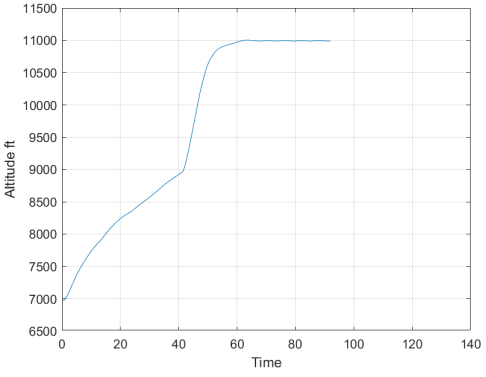

Figure 4.36: Altitude standard turn maneuver bank=45°

4.2.5 Test of the complete system

Finally, the complete autopilot system was subjected to comprehensive testing. The individual systems, including the takeoff control, altitude control, and turn maneuver systems, were integrated into a cohesive unit and tested together to evaluate their collective performance. The behavior of the fully integrated autopilot system proved to be consistent with the results observed during the tests of the individual components, demonstrating that the system functions reliably when all subsystems are combined. This consistency reinforces the robustness and effectiveness of the overall design. The performance of the complete autopilot system can be observed in a video, which is accessible via the provided link: `https://youtu.be/zFX-xf54RzQ` or by scanning the QR code displayed in Figure 4.37. The video specifically demonstrates the operation of the complete system, including takeoff, a turn maneuver, and altitude setting. The departure airport, as in previous tests, is Washington D.C. airport. The video shows that once the aircraft reaches 200 knots, it begins to pitch up, setting an angle of attack of 15°. Upon reaching 7000 feet, it initiates a turn maneuver to align with the geographical coordinates: latitude 47.6421 and longitude -145.13010. After completing the turn, the aircraft return to horizontal position and reach the target altitude of 11000 feet.

Figure 4.37: QR code of the autopilot system video

Chapter 5

Conclusions and future developments

5.1 Conclusions

The project described in this thesis led to the development of an advanced autopilot system capable of flying an F-35 aircraft to predefined geographical coordinates and maintaining a specific altitude. A key contribution of this work was the significant improvement of the aircraft stabilization controller compared to the existing one. The new developed controller demonstrated superior effectiveness in managing the aircraft's dynamics, providing a quicker and more stable response to various external disturbances, thereby enhancing flight safety and reliability.

In parallel, the development of a wearable system integrating haptic and visual feedback improved the simulation experience, offering an innovative and cheaper solution compared to traditional flight simulators. This system has proven to significantly enhance immersion and realism during simulation sessions, opening new perspectives for training and education.

5.2 Future Developments

Despite the results achieved, there are multiple directions that could be explored to further improve the system. One of the primary areas for development involves the adoption of more advanced control techniques, such as adaptive control or robust control, which could further increase the efficiency and reliability of the autopilot system, particularly under complex or unexpected operating conditions.

Additionally, expanding the wearable system to include more sensors and actuators could further enhance the user experience, providing even more detailed

and realistic feedback. It could also be beneficial to explore the integration of the system with advanced artificial intelligence technologies, improving the system's ability to adapt in real-time to variable flight conditions.

Another promising direction is the scalability of the simulation system, with the goal of integrating it into distributed simulation environments. This would allow collaboration between multiple simulators, improving training in joint missions or complex scenarios. Finally, the principles developed in this project could be extended to other sectors, such as the automotive or naval industries, where advanced control systems could significantly improve safety and operational efficiency.

Appendix A

Data acquisition module

```
#include <windows.h>
#include <tchar.h>
#include <stdio.h>
#include "SimConnect.h"
#include <strsafe.h>
#include <string>
#include <vector>
#include <fstream>
#include <sstream>
#include <iostream>
#include <unordered_map>
#include <algorithm>
#include <thread>
#include <atomic>
#include <chrono>

using namespace std;

HANDLE hSimConnect = NULL;
atomic_bool keepRunning(true);

// Struttura per memorizzare le informazioni sui parametri
struct ParameterInfo {
    string key;
    string unit;
};

// Definizione degli identificatori per dati e richieste
enum DATA_DEFINE_ID {
    DEFINITION_1,
};
```

```
32
33 enum DATA_REQUEST_ID {
34     REQUEST_1,
35 };
36
37 // Funzione per rimuovere spazi da una stringa
38 string removeSpaces(const string& str) {
39     string result = str;
40     result.erase(remove(result.begin(), result.end(), ' '), result.
   end());
41     return result;
42 }
43
44 // Funzione per leggere il file di configurazione
45 vector<ParameterInfo> readConfigFile(const char* filename) {
46     vector<ParameterInfo> params;
47     ifstream file(filename);
48
49     if (!file.is_open()) {
50         cerr << "Errore durante l'apertura del file di configurazione
   .\n";
51         return params;
52     }
53
54     string line;
55     while (getline(file, line)) {
56         istringstream iss(line);
57         string key, unit;
58
59         if (getline(iss, key, ',') && getline(iss, unit)) {
60             params.push_back({ key, unit });
61         }
62     }
63
64     file.close();
65     return params;
66 }
67
68 // Funzione per stampare i dati sulla console
69 void printData(const unordered_map<string, double>& dataMap, const
   vector<ParameterInfo>& params) {
70     for (const auto& param : params) {
71         auto it = dataMap.find(param.key);
72         if (it != dataMap.end()) {
73             printf("%-30s: %.2f %s\n", param.key.c_str(), it->second,
    param.unit.c_str());
74         }
75         else {
```

```
76              printf("%-30s: Dato non disponibile\n", param.key.c_str()
    );
77          }
78      }
79  }
80
81  // Funzione per aggiornare e eseguire il file batch
82  void updateAndExecuteBatchFile(const unordered_map<string, double>&
    dataMap, const ParameterInfo& param) {
83      string sanitizedKey = removeSpaces(param.key);
84      ofstream batchFile("batch_file/" + sanitizedKey + ".bat");
85
86      if (batchFile.is_open()) {
87          batchFile << "\"C:\\Program Files\\mosquitto\\mosquitto_pub.
    exe\" -t data/"
88              << sanitizedKey << " -m " << dataMap.at(param.key);
89          batchFile.close();
90
91          // Esegui il file batch
92          string batchFileName = "batch_file\\" + sanitizedKey + ".bat"
    ;
93          system(batchFileName.c_str());
94      }
95      else {
96          cerr << "Errore durante la creazione del file batch per " <<
    param.key << ".\n";
97      }
98  }
99
100 // Funzione di callback per gestire i dati ricevuti da SimConnect
101 void CALLBACK MyDispatchProcRD(SIMCONNECT_RECV* pData, DWORD cbData,
    void* pContext) {
102     static vector<ParameterInfo>* params = (vector<ParameterInfo>*)
    pContext;
103     static unordered_map<string, double> dataMap;
104
105     switch (pData->dwID) {
106     case SIMCONNECT_RECV_ID_SIMOBJECT_DATA_BYTYPE: {
107         SIMCONNECT_RECV_SIMOBJECT_DATA_BYTYPE* pObjData = (
    SIMCONNECT_RECV_SIMOBJECT_DATA_BYTYPE*)pData;
108
109         switch (pObjData->dwRequestID) {
110         case REQUEST_1: {
111             char* data = (char*)&pObjData->dwData;
112
113             // Pulisci la mappa dei dati prima di popolarla con i
    nuovi valori
114             dataMap.clear();
115
```

```
116             // Popola la mappa dei dati con i valori ricevuti
117             for (const auto& param : *params) {
118                 double value = *((double*)(data + sizeof(double) * (&
    param - &(*params)[0])));
119                 dataMap[param.key] = value;
120             }
121 
122             // Aggiorna e esegui il file batch per ogni parametro
123             for (const auto& param : *params) {
124                 updateAndExecuteBatchFile(dataMap, param);
125             }
126 
127             // Stampare i dati sulla console
128             printData(dataMap, *params);
129             break;
130         }
131 
132         default:
133             break;
134         }
135         break;
136     }
137 
138     case SIMCONNECT_RECV_ID_QUIT: {
139         printf("\nDisconnesso da Prepar3D.\n");
140         keepRunning = false;
141         break;
142     }
143 
144     default:
145         printf("\nReceived:%d", pData->dwID);
146         break;
147     }
148 }
149 
150 // Funzione per configurare SimConnect e iniziare la richiesta di
    dati
151 void setupSimConnectAndRequestData() {
152     HRESULT hr;
153 
154     // Connessione a SimConnect
155     if (SUCCEEDED(SimConnect_Open(&hSimConnect, "Request Data", NULL,
    0, 0, 0))) {
156         printf("\nConnesso a Prepar3D!\n");
157 
158         // Leggi il file di configurazione
159         vector<ParameterInfo> params = readConfigFile("config.txt");
160 
```

```
161         // Definisci i dati da richiedere in base ai parametri
    specificati nel file di configurazione
162         for (const auto& param : params) {
163             hr = SimConnect_AddToDataDefinition(hSimConnect,
    DEFINITION_1, param.key.c_str(), param.unit.c_str());
164         }
165
166         // Richiedi i dati ogni secondo
167         while (keepRunning) {
168             //auto start = std::chrono::high_resolution_clock::now();
169
170             hr = SimConnect_RequestDataOnSimObjectType(hSimConnect,
    REQUEST_1, DEFINITION_1, 0, SIMCONNECT_SIMOBJECT_TYPE_USER);
171             SimConnect_CallDispatch(hSimConnect, MyDispatchProcRD, &
    params);
172             Sleep(50);
173
174             /*auto end = std::chrono::high_resolution_clock::now();
175             std::chrono::duration<double> elapsed = end - start;
176
177             std::cout << "Tempo ciclo: " << elapsed.count() << "
    secondi" << std::endl;*/
178         }
179
180         // Chiudi la connessione a SimConnect
181         hr = SimConnect_Close(hSimConnect);
182     }
183     else {
184         cerr << "Errore nella connessione a Prepar3D.\n";
185     }
186 }
187
188 // Funzione principale
189 int __cdecl _tmain(int argc, _TCHAR* argv[]) {
190     setupSimConnectAndRequestData();
191     return 0;
192 }
193
194 }
```

Appendix B

Bank and Pitch controllers

```
#include <windows.h>
#include <tchar.h>
#include <stdio.h>
#include "SimConnect.h"
#include <strsafe.h>

HANDLE hSimConnect = NULL;
FILE* pFile = NULL; // CSV file pointer

struct FlightData
{
    double bank;
    double pitch;
};

struct AngularVelocityData
{
    double rollRate;
    double pitchRate;
};

struct PIDController
{
    double kp;
    double ki;
    double kd;
    double integral;
    double previous_error;
};

PIDController aileronPID;
```

```
32 PIDController elevatorPID;
33
34 enum EVENT_ID {
35     AILERON_SET,
36     ELEVATOR_SET
37 };
38
39 enum DATA_DEFINE_ID {
40     DEFINITION_1,
41     DEFINITION_2
42 };
43
44 enum DATA_REQUEST_ID {
45     REQUEST_1,
46 };
47
48 // Scale PID output to match the control surface set event's accepted
       values
49 int scaleOutputToControlSurfaceSet(double output, double
       max_deflection)
50 {
51     // Convert degrees to the -16383 to 16383 range
52     int scaledOutput = static_cast<int>((output / max_deflection) *
       16383);
53     // Clamp the output to the range of -16383 to 16383
54     if (scaledOutput > 16383) scaledOutput = 16383;
55     if (scaledOutput < -16383) scaledOutput = -16383;
56     return scaledOutput;
57 }
58
59 // Callback function to handle received data
60 void CALLBACK MyDispatchProcRD(SIMCONNECT_RECV* pData, DWORD cbData,
       void* pContext)
61 {
62     HRESULT hr;
63
64     switch (pData->dwID)
65     {
66     case SIMCONNECT_RECV_ID_SIMOBJECT_DATA_BYTYPE:
67     {
68         SIMCONNECT_RECV_SIMOBJECT_DATA_BYTYPE* pObjData = (
       SIMCONNECT_RECV_SIMOBJECT_DATA_BYTYPE*)pData;
69
70         if (pObjData->dwRequestID == REQUEST_1)
71         {
72             FlightData* pS = (FlightData*)&pObjData->dwData;
73             AngularVelocityData* pV = (AngularVelocityData*)&pObjData
       ->dwData;
74
```

```
75              // Calculate the error between the current bank angle and
     the desired bank angle (zero)
76              double bankError = 0.0 - pS->bank;
77              double pitchError = 0.0 - pS->pitch;
78
79              // Use angular velocity for the derivative action
80              double aileronDerivative = -pV->rollRate;
81              double elevatorDerivative = -pV->pitchRate;
82
83              // Apply the PID control to calculate the aileron
    adjustment
84              aileronPID.integral += bankError;
85              double aileronOutput = aileronPID.kp * bankError +
    aileronPID.ki * aileronPID.integral + aileronPID.kd *
    aileronDerivative;
86
87              // Apply the PID control to calculate the elevator
    adjustment
88              elevatorPID.integral += pitchError;
89              double elevatorOutput = elevatorPID.kp * pitchError +
    elevatorPID.ki * elevatorPID.integral + elevatorPID.kd *
    elevatorDerivative;
90
91              aileronPID.previous_error = bankError;
92              elevatorPID.previous_error = pitchError;
93
94              // Scale the PID outputs to the control surface set range
95              int aileronValue = scaleOutputToControlSurfaceSet(
    aileronOutput, 25.0);
96              int elevatorValue = scaleOutputToControlSurfaceSet(
    elevatorOutput, 30.0);
97
98              // Print to the terminal
99              printf("Bank Angle: %f, Aileron Output: %f, Aileron Set
    Value: %d\n", pS->bank, aileronOutput, aileronValue);
100             printf("Pitch Angle: %f, Elevator Output: %f, Elevator
    Set Value: %d\n", pS->pitch, elevatorOutput, elevatorValue);
101
102             // Set the ailerons based on the PID output
103             hr = SimConnect_TransmitClientEvent(hSimConnect,
    SIMCONNECT_OBJECT_ID_USER, AILERON_SET, aileronValue,
    SIMCONNECT_GROUP_PRIORITY_HIGHEST,
    SIMCONNECT_EVENT_FLAG_GROUPID_IS_PRIORITY);
104             if (FAILED(hr))
105             {
106                 printf("TransmitClientEvent for setting aileron -
    error\n");
107             }
108
```

```
109             // Set the elevators based on the PID output
110             /* hr = SimConnect_TransmitClientEvent(hSimConnect,
    SIMCONNECT_OBJECT_ID_USER, ELEVATOR_SET, elevatorValue,
    SIMCONNECT_GROUP_PRIORITY_HIGHEST,
    SIMCONNECT_EVENT_FLAG_GROUPID_IS_PRIORITY);
111             if (FAILED(hr))
112             {
113                 printf("TransmitClientEvent for setting elevator -
    error\n");
114             }*/
115         }
116         break;
117     }
118
119     case SIMCONNECT_RECV_ID_QUIT:
120     {
121         printf("\nExiting...");
122         break;
123     }
124
125     default:
126         printf("\nReceived:%d", pData->dwID);
127         break;
128     }
129 }
130
131 void InitializePIDControllers()
132 {
133     aileronPID.kp = 30; // Proportional coefficient for ailerons
134     aileronPID.ki = 0.1; // Integral coefficient for ailerons
135     aileronPID.kd = 20; // Derivative coefficient for ailerons
136     aileronPID.integral = 0.0;
137     aileronPID.previous_error = 0.0;
138
139     elevatorPID.kp = 15; // Proportional coefficient for elevators
140     elevatorPID.ki = 0.2; // Integral coefficient for elevators
141     elevatorPID.kd = 1.0; // Derivative coefficient for elevators
142     elevatorPID.integral = 0.0;
143     elevatorPID.previous_error = 0.0;
144 }
145
146 void testDataRequest()
147 {
148     HRESULT hr;
149
150     if (SUCCEEDED(SimConnect_Open(&hSimConnect, "PID Controller",
    NULL, 0, 0, 0)))
151     {
152         printf("\nConnected to Flight Simulator!");
```

```
153
154         // Set up the data definition for bank and pitch angles
155         hr = SimConnect_AddToDataDefinition(hSimConnect, DEFINITION_1
    , "PLANE BANK DEGREES", "radians");
156         hr = SimConnect_AddToDataDefinition(hSimConnect, DEFINITION_1
    , "PLANE PITCH DEGREES", "radians");
157
158         // Set up the data definition for angular velocities
159         hr = SimConnect_AddToDataDefinition(hSimConnect, DEFINITION_2
    , "ROTATION VELOCITY BODY X", "radians per second");
160         hr = SimConnect_AddToDataDefinition(hSimConnect, DEFINITION_2
    , "ROTATION VELOCITY BODY Y", "radians per second");
161
162         // Map events
163         hr = SimConnect_MapClientEventToSimEvent(hSimConnect,
    AILERON_SET, "AILERON_SET");
164         if (FAILED(hr)) {
165             printf("Errore nella mappatura di AILERON_SET - error\n")
    ;
166             SimConnect_Close(hSimConnect);
167         }
168
169         hr = SimConnect_MapClientEventToSimEvent(hSimConnect,
    ELEVATOR_SET, "ELEVATOR_SET");
170         if (FAILED(hr)) {
171             printf("Errore nella mappatura di ELEVATOR_SET - error\n"
    );
172             SimConnect_Close(hSimConnect);
173         }
174
175         while (1)
176         {
177             // Request data on the user aircraft for bank and pitch
    angles
178             hr = SimConnect_RequestDataOnSimObjectType(hSimConnect,
    REQUEST_1, DEFINITION_1, 0, SIMCONNECT_SIMOBJECT_TYPE_USER);
179
180             // Request data on the user aircraft for angular
    velocities
181             hr = SimConnect_RequestDataOnSimObjectType(hSimConnect,
    REQUEST_1, DEFINITION_2, 0, SIMCONNECT_SIMOBJECT_TYPE_USER);
182
183             // Call the dispatch function to handle received data
184             SimConnect_CallDispatch(hSimConnect, MyDispatchProcRD,
    NULL);
185
186             // Sleep for 100 milliseconds (0.1 second)
187             Sleep(100);
188         }
```

```
189
190             hr = SimConnect_Close(hSimConnect);
191         }
192 }
193
194 int _tmain(int argc, _TCHAR* argv[])
195 {
196     InitializePIDControllers();
197     testDataRequest();
198     return 0;
199 }
```

Appendix C

Navigation algorithm

```
#include <windows.h>
#include <tchar.h>
#include <stdio.h>
#include "SimConnect.h"
#include <strsafe.h>
#include <cmath>
#include <iostream>
#include <fstream>
#include <string>
#include <map>

HANDLE hSimConnect = NULL;
FILE* pFile = NULL; // CSV file pointer

//desired angle
double desired_bank;
double desired_heading;
double desired_pitch;
double desired_altitude = 11000;
double desired_latitude = 47.64210;
double desired_longitude = -145.13010;
double headingError;
double bankError;
double pitchError;
double flag = 0;
double pitch_flag = 0;
double takeoff_flag = 0;
int elevatorValue;
double pi = 3.141592;


```

```
32
33 struct FlightData
34 {
35     double bank;
36     double pitch;
37     double heading;
38     double plane_lat;
39     double plane_long;
40     double altitude;
41     double airspeed;
42     double angular_velocity;
43 };
44
45 enum GROUP_ID {
46     GROUP_INIT,
47 };
48
49 enum INPUT_ID {
50     INPUT_INIT,
51 };
52
53 struct PIDController
54 {
55     double kp;
56     double ki;
57     double kd;
58     double integral;
59     double previous_error;
60 };
61
62 PIDController aileronPID;
63 PIDController elevatorPID;
64 PIDController headingPID;
65
66 enum EVENT_ID {
67     EVENT_SIM_START,
68     EVENT_INIT,
69     EVENT_SET_ELEVATOR,
70     EVENT_SET_AILERON,
71     EVENT_SET_THROTTLE,
72     EVENT_TOGLE_GEAR
73 };
74
75 enum DATA_DEFINE_ID {
76     DEFINITION_INIT,
77     DEFINITION_GEAR,
78     DEFINITION_1,
79 };
80
```

```
81 enum DATA_REQUEST_ID {
82     REQUEST_1,
83 };
84
85 // Scale PID output to match the control surface set event's accepted
       values
86 int scaleOutputToControlSurfaceSet(double output, double
       max_deflection)
87 {
88     // Convert degrees to the -16383 to 16383 range
89     int scaledOutput = static_cast<int>((output / max_deflection) *
       16383);
90     // Clamp the output to the range of -16383 to 16383
91     if (scaledOutput > 16383) scaledOutput = 16383;
92     if (scaledOutput < -16383) scaledOutput = -16383;
93     return scaledOutput;
94 }
95
96 void writeDataToCSV(double bank, double pitch, double heading, double
       desired_heading, double lat, double lon, double altitude, double
       airspeed, double angularvelocity) {
97     // Apri il file CSV in modalità append
98     FILE* pFile = fopen("test_disturbed.csv", "a");
99     if (pFile) {
100         // Scrivi i valori di bank e pitch nel file CSV
101         fprintf(pFile, "%f,%f,%f,%f,%f,%f,%f,%f,%f\n", bank, pitch,
       heading, desired_heading, lat, lon, altitude, airspeed,
       angularvelocity);
102         // Chiudi il file
103         fclose(pFile);
104     }
105     else {
106         printf("Errore nell'apertura del file CSV\n");
107     }
108 }
109
110 // Function to calculate initial azimuth in radians
111 double calculate_initial_azimuth(double lat1, double lon1, double
       lat2, double lon2) {
112     // Convert latitudes and longitudes from degrees to radians
113     double lat1_rad = lat1;
114     double lon1_rad = lon1;
115     double lat2_rad = lat2 * pi / 180.0;
116     double lon2_rad = lon2 * pi / 180.0;
117
118     // Calculate difference in longitude
119     double delta_lon = lon2_rad - lon1_rad;
120
121     // Calculate initial azimuth
```

```
122     double y = sin(delta_lon) * cos(lat2_rad);
123     double x = cos(lat1_rad) * sin(lat2_rad) - sin(lat1_rad) * cos(
    lat2_rad) * cos(delta_lon);
124     double initial_azimuth_rad = atan2(y, x);
125 
126     // Normalize azimuth between 0 and 2*pi
127     double azimuth_rad = fmod(initial_azimuth_rad, 2 * pi);
128     if (azimuth_rad < 0) {
129         azimuth_rad += 2 * pi; // Ensure azimuth is positive
130     }
131 
132     return azimuth_rad;
133 }
134 
135 // Callback function to handle received data
136 void CALLBACK MyDispatchProcRD(SIMCONNECT_RECV* pData, DWORD cbData,
    void* pContext)
137 {
138     HRESULT hr;
139 
140     switch (pData->dwID)
141     {
142     case SIMCONNECT_RECV_ID_SIMOBJECT_DATA_BYTYPE:
143     {
144         SIMCONNECT_RECV_SIMOBJECT_DATA_BYTYPE* pObjData = (
    SIMCONNECT_RECV_SIMOBJECT_DATA_BYTYPE*)pData;
145 
146         if (pObjData->dwRequestID == REQUEST_1)
147         {
148             FlightData* pS = (FlightData*)&pObjData->dwData;
149 
150             if (takeoff_flag == 0 && pS->airspeed > 150)
151             {
152                 // Calculate the error between the current bank angle
     and the desired bank angle (zero)
153                 double bankError = 0.0 - pS->bank;
154                 double pitchError = -15 * pi / 180 - pS->pitch;
155 
156                 // Apply the PID control to calculate the aileron
    adjustment
157                 aileronPID.integral += bankError;
158                 double aileronDerivative = bankError - aileronPID.
    previous_error;
159                 double aileronOutput = aileronPID.kp * bankError +
    aileronPID.ki * aileronPID.integral + aileronPID.kd *
    aileronDerivative;
160                 aileronPID.previous_error = bankError;
161 
```

```
162                 // Apply the PID control to calculate the elevator
    adjustment
163                 elevatorPID.integral += pitchError;
164                 double elevatorDerivative = pitchError - elevatorPID.
    previous_error;
165                 double elevatorOutput = elevatorPID.kp * pitchError +
     elevatorPID.ki * elevatorPID.integral + elevatorPID.kd *
    elevatorDerivative;
166                 elevatorPID.previous_error = pitchError;
167 
168                 // Scale the PID outputs to the control surface set
    range
169                 int aileronValue = scaleOutputToControlSurfaceSet(
    aileronOutput, 25.0);
170                 int elevatorValue = scaleOutputToControlSurfaceSet(
    elevatorOutput, 30.0);
171 
172                 // Print to the terminal
173                 printf("Bank Angle: %f, Aileron Output: %f, Aileron
    Set Value: %d\n", pS->bank, aileronOutput, aileronValue);
174                 printf("Pitch Angle: %f, Elevator Output: %f,
    Elevator Set Value: %d\n", pS->pitch, elevatorOutput,
    elevatorValue);
175 
176                 // Scrivi i dati di bank e pitch nel file CSV
177                 writeDataToCSV(pS->bank, pS->pitch, pS->heading,
    desired_heading, pS->plane_lat, pS->plane_long, pS->altitude, pS->
    airspeed, pS->angular_velocity);
178 
179                 // Set the ailerons based on the PID output
180                 hr = SimConnect_MapClientEventToSimEvent(hSimConnect,
     EVENT_SET_AILERON, "AILERON_SET");
181                 if (FAILED(hr))
182                 {
183                     printf("MapClientEventToSimEvent for AILERON_SET
    - error\n");
184                     SimConnect_Close(hSimConnect);
185                 }
186 
187                 hr = SimConnect_TransmitClientEvent(hSimConnect,
    SIMCONNECT_OBJECT_ID_USER, EVENT_SET_AILERON, aileronValue,
    SIMCONNECT_GROUP_PRIORITY_HIGHEST,
    SIMCONNECT_EVENT_FLAG_GROUPID_IS_PRIORITY);
188                 if (FAILED(hr))
189                 {
190                     printf("TransmitClientEvent for setting aileron -
     error\n");
191                 }
192 
```

```
193                 // Set the elevators based on the PID output
194                 hr = SimConnect_MapClientEventToSimEvent(hSimConnect,
    EVENT_SET_ELEVATOR, "ELEVATOR_SET");
195                 if (FAILED(hr))
196                 {
197                     printf("MapClientEventToSimEvent for ELEVATOR_SET
    - error\n");
198                     SimConnect_Close(hSimConnect);
199                 }
200
201                 hr = SimConnect_TransmitClientEvent(hSimConnect,
    SIMCONNECT_OBJECT_ID_USER, EVENT_SET_ELEVATOR, elevatorValue,
    SIMCONNECT_GROUP_PRIORITY_HIGHEST,
    SIMCONNECT_EVENT_FLAG_GROUPID_IS_PRIORITY);
202                 if (FAILED(hr))
203                 {
204                     printf("TransmitClientEvent for setting elevator
    - error\n");
205                 }
206                 if (pS->airspeed > 550 && pS->altitude > 7000)
207                 {
208                     int throttlePct = 11468;  // 70% of 16383
209                     hr = SimConnect_MapClientEventToSimEvent(
    hSimConnect, EVENT_SET_THROTTLE, "THROTTLE_SET");
210                     hr = SimConnect_TransmitClientEvent(hSimConnect,
    SIMCONNECT_OBJECT_ID_USER, EVENT_SET_THROTTLE, throttlePct,
    SIMCONNECT_GROUP_PRIORITY_HIGHEST,
    SIMCONNECT_EVENT_FLAG_GROUPID_IS_PRIORITY);
211                     takeoff_flag = 1;
212                 }
213             }
214
215             if (takeoff_flag == 1) {
216
217                 desired_heading = calculate_initial_azimuth(pS->
    plane_lat, pS->plane_long, desired_latitude, desired_longitude);
218
219                 headingError = desired_heading - pS->heading;
220                 if (flag == 0 && (headingError > pi || (headingError
    < 0 && headingError > -pi)))
221                 {
222                     desired_bank = 40 * pi / 180;
223                     flag = 1;
224                 }
225                 else if (flag == 0)
226                 {
227                     desired_bank = -40 * pi / 180;
228                     flag = 1;
229                 }
```

```
230
231                 bankError = desired_bank - pS->bank;
232
233                 if (headingError > -0.005 && headingError < 0.005) {
234                     desired_bank = 0;
235                     desired_heading = pS->heading;
236                     pitch_flag = 1;
237                 }
238
239                 // Apply the PID control to calculate the aileron
    adjustment
240                 aileronPID.integral += bankError;
241                 double aileronDerivative = bankError - aileronPID.
    previous_error;
242                 double aileronOutput = aileronPID.kp * bankError +
    aileronPID.ki * aileronPID.integral + aileronPID.kd *
    aileronDerivative;
243                 aileronPID.previous_error = bankError;
244
245                 // Apply the PID control to calculate the elevator
    adjustment
246                 double angularError = -3 * pi / 180 - pS->
    angular_velocity;
247                 headingPID.integral += angularError;
248                 double headingDerivative = angularError - headingPID.
    previous_error;
249                 double headingOutput = headingPID.kp * angularError +
     headingPID.ki * headingPID.integral + headingPID.kd *
    headingDerivative;
250                 headingPID.previous_error = angularError;
251
252                 if (pitch_flag == 1)
253                 {
254                     double altitudeError = desired_altitude - pS->
    altitude;
255                     if (altitudeError > 1000) {
256                         desired_pitch = -15 * pi / 180;
257                     }
258                     else if (altitudeError < 1000 && altitudeError >
    500) {
259                         desired_pitch = -10 * pi / 180;
260                     }
261                     else if (altitudeError < 500 && altitudeError >
    200) {
262                         desired_pitch = -5 * pi / 180;
263                     }
264                     else if (altitudeError < 200 && altitudeError >
    10) {
265                         desired_pitch = -3 * pi / 180;
```

```
266                     }
267                     else if (altitudeError < 10 && altitudeError >
    -10) {
268                         desired_pitch = -2 * pi / 180;
269                     }
270                     else if (altitudeError < -3000) {
271                         desired_pitch = 15 * pi / 180;
272                     }
273                     else if (altitudeError > -3000 && altitudeError <
     -1000) {
274                         desired_pitch = 10 * pi / 180;
275                     }
276                     else if (altitudeError > -1000 && altitudeError <
     -500) {
277                         desired_pitch = 5 * pi / 180;
278                     }
279                     else if (altitudeError > -500 && altitudeError <
    -10) {
280                         desired_pitch = 3 * pi / 180;
281                     }
282                     pitchError = desired_pitch - pS->pitch;
283 
284                     // Apply the PID control to calculate the
    elevator adjustment
285                     elevatorPID.integral += pitchError;
286                     double elevatorDerivative = pitchError -
    elevatorPID.previous_error;
287                     double elevatorOutput = elevatorPID.kp *
    pitchError + elevatorPID.ki * elevatorPID.integral + elevatorPID.
    kd * elevatorDerivative;
288                     elevatorPID.previous_error = pitchError;
289                     int elevatorValue =
    scaleOutputToControlSurfaceSet(elevatorOutput, 30.0);
290                     printf("Pitch Angle: %f, Elevator Output: %f,
    Elevator Set Value: %d\n", pS->pitch, elevatorOutput,
    elevatorValue);
291                     printf("altitude: %f\n", pS->altitude);
292 
293                     // Set the elevators based on the PID output
294                     hr = SimConnect_MapClientEventToSimEvent(
    hSimConnect, EVENT_SET_ELEVATOR, "ELEVATOR_SET");
295                     if (FAILED(hr))
296                     {
297                         printf("MapClientEventToSimEvent for
    ELEVATOR_SET - error\n");
298                         SimConnect_Close(hSimConnect);
299                     }
300 
```

```
301                     hr = SimConnect_TransmitClientEvent(hSimConnect,
    SIMCONNECT_OBJECT_ID_USER, EVENT_SET_ELEVATOR, elevatorValue,
    SIMCONNECT_GROUP_PRIORITY_HIGHEST,
    SIMCONNECT_EVENT_FLAG_GROUPID_IS_PRIORITY);
302                     if (FAILED(hr))
303                     {
304                         printf("TransmitClientEvent for setting
    elevator - error\n");
305                     }
306                 }
307
308                 // Scale the PID outputs to the control surface set
    range
309                 int aileronValue = scaleOutputToControlSurfaceSet(
    aileronOutput, 25.0);
310                 if (pitch_flag == 0)
311                 {
312                     int elevatorValue =
    scaleOutputToControlSurfaceSet(headingOutput, 30.0);
313                     printf("Heading Angle: %f, Heading Error: %f,
    Elevator Set Value: %d\n", pS->heading, headingError,
    elevatorValue);
314                     printf("angular velocity: %f, Angular error: %f\n
    ", pS->angular_velocity, angularError);
315
316                     // Set the elevators based on the PID output
317                     hr = SimConnect_MapClientEventToSimEvent(
    hSimConnect, EVENT_SET_ELEVATOR, "ELEVATOR_SET");
318                     if (FAILED(hr))
319                     {
320                         printf("MapClientEventToSimEvent for
    ELEVATOR_SET - error\n");
321                         SimConnect_Close(hSimConnect);
322                     }
323
324                     hr = SimConnect_TransmitClientEvent(hSimConnect,
    SIMCONNECT_OBJECT_ID_USER, EVENT_SET_ELEVATOR, elevatorValue,
    SIMCONNECT_GROUP_PRIORITY_HIGHEST,
    SIMCONNECT_EVENT_FLAG_GROUPID_IS_PRIORITY);
325                     if (FAILED(hr))
326                     {
327                         printf("TransmitClientEvent for setting
    elevator - error\n");
328                     }
329                 }
330
331
332                 // Print to the terminal
```

```
333                 printf("Bank Angle: %f, Aileron Output: %f, Aileron
    Set Value: %d\n", pS->bank, aileronOutput, aileronValue);
334
335                 // Scrivi i dati di bank e pitch nel file CSV
336                 writeDataToCSV(pS->bank, pS->pitch, pS->heading,
    desired_heading, pS->plane_lat, pS->plane_long, pS->altitude, pS->
    airspeed, pS->angular_velocity);
337
338                 // Set the ailerons based on the PID output
339                 hr = SimConnect_MapClientEventToSimEvent(hSimConnect,
     EVENT_SET_AILERON, "AILERON_SET");
340                 if (FAILED(hr))
341                 {
342                     printf("MapClientEventToSimEvent for AILERON_SET
    - error\n");
343                     SimConnect_Close(hSimConnect);
344                 }
345
346                 hr = SimConnect_TransmitClientEvent(hSimConnect,
    SIMCONNECT_OBJECT_ID_USER, EVENT_SET_AILERON, aileronValue,
    SIMCONNECT_GROUP_PRIORITY_HIGHEST,
    SIMCONNECT_EVENT_FLAG_GROUPID_IS_PRIORITY);
347                 if (FAILED(hr))
348                 {
349                     printf("TransmitClientEvent for setting aileron -
     error\n");
350                 }
351             }
352         }
353         break;
354     }
355
356     case SIMCONNECT_RECV_ID_QUIT:
357     {
358         printf("\nExiting...");
359         break;
360     }
361
362     default:
363         printf("\nReceived:%d", pData->dwID);
364         break;
365     }
366
367 }
368
369 std::map<std::string, double> readConfigFile(const char* filename) {
370     std::ifstream infile(filename);
371     std::map<std::string, double> configValues;
372     std::string line;
```

```
373
374     while (std::getline(infile, line)) {
375         size_t delimiterPos = line.find('=');
376         if (delimiterPos != std::string::npos) {
377             std::string key = line.substr(0, delimiterPos);
378             double value = std::stod(line.substr(delimiterPos + 1));
379             configValues[key] = value;
380         }
381     }
382
383     return configValues;
384 }
385
386
387 void InitializePIDControllers()
388 {
389     aileronPID.kp = 20; // Proportional coefficient for ailerons
390     aileronPID.ki = 0.03; // Integral coefficient for ailerons
391     aileronPID.kd = 0.01; // Derivative coefficient for ailerons
392     aileronPID.integral = 0.0;
393     aileronPID.previous_error = 0.0;
394
395     elevatorPID.kp = 500; // Proportional coefficient for elevators
396     elevatorPID.ki = 0.5; // Integral coefficient for elevators //
    CON 10 E' VELOCE MA ALTA SOVRAEONGAZIONE
397     elevatorPID.kd = 0.1; // Derivative coefficient for elevators
398     elevatorPID.integral = 0.0;
399     elevatorPID.previous_error = 0.0;
400
401     headingPID.kp = 30; // Proportional coefficient for heading
402     headingPID.ki = 0.03; // Integral coefficient for heading
403     headingPID.kd = 0.01; // Derivative coefficient for heading
404     headingPID.integral = 0.0;
405     headingPID.previous_error = 0.0;
406 }
407
408 void testDataRequest()
409 {
410     HRESULT hr;
411
412     // Read the configuration file
413     std::map<std::string, double> configValues = readConfigFile("
    config.txt");
414
415     // Initialize the global variables with the values from the
    configuration file
416     desired_altitude = configValues["desired_altitude"];
417     desired_latitude = configValues["desired_latitude"];
418     desired_longitude = configValues["desired_longitude"];
```

```
419
420     if (SUCCEEDED(SimConnect_Open(&hSimConnect, "PID Controller",
        NULL, 0, 0, 0)))
421     {
422         printf("\nConnected to Flight Simulator!");
423
424         // Set up the data definition
425         hr = SimConnect_AddToDataDefinition(hSimConnect, DEFINITION_1
        , "PLANE BANK DEGREES", "radians");
426         hr = SimConnect_AddToDataDefinition(hSimConnect, DEFINITION_1
        , "PLANE PITCH DEGREES", "radians");
427         hr = SimConnect_AddToDataDefinition(hSimConnect, DEFINITION_1
        , "PLANE HEADING DEGREES TRUE", "radians");
428         hr = SimConnect_AddToDataDefinition(hSimConnect, DEFINITION_1
        , "PLANE LATITUDE", "radians");
429         hr = SimConnect_AddToDataDefinition(hSimConnect, DEFINITION_1
        , "PLANE LONGITUDE", "radians");
430         hr = SimConnect_AddToDataDefinition(hSimConnect, DEFINITION_1
        , "PLANE ALTITUDE", "feet");
431         hr = SimConnect_AddToDataDefinition(hSimConnect, DEFINITION_1
        , "AIRSPEED TRUE", "knots");
432         hr = SimConnect_AddToDataDefinition(hSimConnect, DEFINITION_1
        , "ROTATION VELOCITY BODY Z", "Radians per second");
433
434         /* // Set up a data definition for positioning data
435          hr = SimConnect_AddToDataDefinition(hSimConnect,
        DEFINITION_INIT, "INITIAL POSITION", NULL,
        SIMCONNECT_DATATYPE_INITPOSITION);
436
437         // Set up a data definition for the gear position
438         hr = SimConnect_AddToDataDefinition(hSimConnect,
        DEFINITION_GEAR, "GEAR POSITION", "index",
        SIMCONNECT_DATATYPE_FLOAT64);
439
440         // Set the user aircraft to a new position immediately
441         SIMCONNECT_DATA_INITPOSITION Init;
442         Init.Altitude = 7000.0;         // Set the altitude to 5000
        feet
443         Init.Latitude = 47.64210;       // Set the latitude to
        47.64210
444         Init.Longitude = -122.13010;  // Set the longitude to
        -122.13010
445         Init.Pitch = -10.0;               // Set the pitch to 0
        degrees
446         Init.Bank = 0.0;                // Set the bank to 0 degrees
447         Init.Heading = 0.0;             // Set the heading to 180
        degrees
448         Init.OnGround = 0;              // Aircraft is in flight
```

```
449         Init.Airspeed = 550;            // Set the airspeed to 60
    knots
450         hr = SimConnect_SetDataOnSimObject(hSimConnect,
    DEFINITION_INIT, SIMCONNECT_OBJECT_ID_USER, 0, 0, sizeof(Init), &
    Init);
451         printf("\nInitial position data sent");
452         */
453         // Set the elevator, aileron, and throttle positions
454       int elevatorPosition = 0;
455       int aileronPosition = 0;
456       int throttlePosition = 16383;  // 100% of 16383
457
458       hr = SimConnect_MapClientEventToSimEvent(hSimConnect,
    EVENT_SET_ELEVATOR, "ELEVATOR_SET");
459       hr = SimConnect_MapClientEventToSimEvent(hSimConnect,
    EVENT_SET_AILERON, "AILERON_SET");
460       hr = SimConnect_MapClientEventToSimEvent(hSimConnect,
    EVENT_SET_THROTTLE, "THROTTLE_SET");
461       hr = SimConnect_MapClientEventToSimEvent(hSimConnect,
    EVENT_TOGLE_GEAR, "GEAR_SET");
462
463       hr = SimConnect_TransmitClientEvent(hSimConnect,
    SIMCONNECT_OBJECT_ID_USER, EVENT_SET_ELEVATOR, elevatorPosition,
    SIMCONNECT_GROUP_PRIORITY_HIGHEST,
    SIMCONNECT_EVENT_FLAG_GROUPID_IS_PRIORITY);
464       hr = SimConnect_TransmitClientEvent(hSimConnect,
    SIMCONNECT_OBJECT_ID_USER, EVENT_SET_AILERON, aileronPosition,
    SIMCONNECT_GROUP_PRIORITY_HIGHEST,
    SIMCONNECT_EVENT_FLAG_GROUPID_IS_PRIORITY);
465       hr = SimConnect_TransmitClientEvent(hSimConnect,
    SIMCONNECT_OBJECT_ID_USER, EVENT_SET_THROTTLE, throttlePosition,
    SIMCONNECT_GROUP_PRIORITY_HIGHEST,
    SIMCONNECT_EVENT_FLAG_GROUPID_IS_PRIORITY);
466       hr = SimConnect_TransmitClientEvent(hSimConnect,
    SIMCONNECT_OBJECT_ID_USER, EVENT_TOGLE_GEAR, 0,
    SIMCONNECT_GROUP_PRIORITY_HIGHEST,
    SIMCONNECT_EVENT_FLAG_GROUPID_IS_PRIORITY);
467
468       printf("\nElevator, aileron, and throttle positions set");
469
470       // Request a simulation start event
471       hr = SimConnect_SubscribeToSystemEvent(hSimConnect,
    EVENT_SIM_START, "SimStart");
472       Sleep(1000);
473
474       while (1)
475       {
476           // Request data on the user aircraft
```

```
477             hr = SimConnect_RequestDataOnSimObjectType(hSimConnect,
    REQUEST_1, DEFINITION_1, 0, SIMCONNECT_SIMOBJECT_TYPE_USER);
478
479             // Call the dispatch function to handle received data
480             SimConnect_CallDispatch(hSimConnect, MyDispatchProcRD,
    NULL);
481
482             // Sleep for 100 milliseconds (0.1 second)
483             Sleep(100);
484         }
485
486         hr = SimConnect_Close(hSimConnect);
487     }
488 }
489
490 int _tmain(int argc, _TCHAR* argv[])
491 {
492     InitializePIDControllers();
493     testDataRequest();
494     return 0;
495 }
```

Bibliography

[1] *Teaching and Testing in Flight Simulation Training Devices (FSTD)*. Tech. rep. Cologne, Germany: EHEST, 2015. URL: https://www.easa.europa.eu/en/document-library/general-publications/ehest-leaflet-he-10-teaching-and-testing-flight-simulation.

[2] *Flight Guidance Systems*. Tech. rep. $AC_2 5_1 329 - 1C$. Washington, D.C.: U.S. Department of Transportation Federal Aviation Administration, 2014. URL: https://www.faa.gov/regulations_policies/advisory_circulars/index.cfm/go/document.information/documentid/1026174.

[3] *AUTO FLIGHT GUIDANCE SYSTEMS*. Tech. rep. $AC_1 20_6 7$. Washington, D.C.: U.S. Department of Transportation Federal Aviation Administration, 1997. URL: https://www.faa.gov/regulations_policies/advisory_circulars/index.cfm/go/document.list/?statusID=3&appliedFacets=%7B%22officenumber%22%3A%22AFS-400%22%7D.

[4] Mário Monteiro Marques. «STANAG 4586—Standard interfaces of UAV control system (UCS) for NATO UAV interoperability». In: *NATO Standardization Agency* 14 (2012).

[5] *F-35 Air Vehicle Technology Overview*. Tech. rep. Atlanta, Georgia: Lockheed Martin, 2018. URL: https://www.lockheedmartin.com/content/dam/lockheed-martin/eo/documents/webt/F-35_Air_Vehicle_Technology_Overview.pdf.

[6] R.C. Nelson. *Flight Stability and Automatic Control*. McGraw-Hill Aerospace Science & Technology Series. McGraw-Hill Education, 1998. ISBN: 9780070462731. URL: https://books.google.it/books?id=Z4lTAAAAMAAJ.

[7] Umberto Salsi. *Le virate standard*. 2018. URL: https://www.icosaedro.it/acm/virate.html#cosegraveunaviratastandard.