UNIVERSITY OF CINCINNATI

Date: _____

I, _____,
hereby submit this work as part of the requirements for the degree of:

in:

It is entitled:

This work and its defense approved by:

Chair: _____

Control Design and Implementation for the Self-Separation of In-Trail Aircraft

A thesis submitted to the
Division of Research and Advanced Studies
Of the University of Cincinnati

In partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

In the Department of Aerospace Engineering and Engineering Mechanics of
the College of Engineering

2005

by

Elena Alonso

Bachelor of Engineering Science (B.E.S.)
The University of Western Ontario, London, Canada, 2003

Committee Chair: Dr. Gary. L. Slater

Control Design and Implementation for the Self-Separation of In-Trail Aircraft

Abstract

One of the responsibilities of air traffic controllers is to maintain aircraft separation in flight. However, with growing air traffic demands, current research is investigating the possibility of delegating some of that responsibility to the cockpit. One suggested implementation of this concept is the design of a control law that will assist pilots in a stream of in-trail aircraft to maintain separation.

The goal of this thesis is to develop and apply control laws that can aid in the maintenance of separation of long strings of aircraft. Automatic Dependence Surveillance – Broadcast (ADS-B) is used to communicate state vectors. The dynamics and controls of the aircraft are numerically simulated with the design of a MATLAB program.

The results for the tests run show promising results. Further tests can be run to potentially optimize the control schemes chosen as well as analyze different flight trajectories.

Acknowledgements

First and foremost, I would like to acknowledge and thank my research advisor, Dr. Gary Slater for his guidance and support. His expertise and patience were invaluable to me in this endeavor. I also want to thank Dr. Bruce Walker and Dr. Trevor Williams for agreeing to put in the time and effort to be on my defense committee. Furthermore, this thesis was made possible by the support of NASA Ames Research Center under Grant NAG 2-1581.

Many thanks are indebted to the University of Cincinnati for their wonderful and helpful staff, professors, and facilities. In particular, thank-you to Brenda Smith, the Aerospace Department Secretary, for always being helpful and for always having a smile and a friendly greeting.

In addition I would like to thank Jonathan Benn for his help in teaching me how to design a software program, how to program with MATLAB, and helping with getting me started on my research. I would also like to thank all my family and friends for their continual love and support.

Table of Contents

Abstract	ii
Acknowledgements.....	iii
Table of Contents	iv
List of Figures.....	vi
Nomenclature.....	vii
Acronyms and Symbols	viii
1 INTRODUCTION.....	1
1.1 BACKGROUND AND MOTIVATION	1
1.2 RESEARCH GOALS AND APPROACH	2
1.3 THESIS OUTLINE.....	2
2 LITERATURE REVIEW	3
2.1 SELF-SPACING DYNAMICS	3
2.2 SIMULATED SELF-SPACING	4
2.3 CONCEPTUAL SOLUTIONS TO SELF-SPACING.....	5
2.4 OVERVIEW	6
3 DYNAMICS AND CONTROL.....	7
3.1 AIRCRAFT DYNAMICS	8
3.2 REFERENCE PATH FINDER FOR LEAD AIRCRAFT	9
3.3 AIRCRAFT FOLLOWING CONTROLS	9
3.3.1 Continuous Separation Control	10
3.3.2 Continuous Separation Control with Lead Feedback.....	11
3.4 REFERENCE PATH FOLLOWING CONTROLS.....	11
3.5 CONTROL LIMITS.....	13

4	NUMERICAL SIMULATION	14
4.1	ADS-B SIMULATION	14
4.2	MATLAB PROGRAM DESIGN AND ARCHITECTURE	15
4.2.1	<i>Lead Aircraft Reference Path Design Requirements</i>	<i>15</i>
4.2.2	<i>Follower Aircraft Reference Path Design Requirements.....</i>	<i>15</i>
4.2.3	<i>Reference Path Follower Design Requirements</i>	<i>16</i>
4.2.4	<i>Program Architecture</i>	<i>16</i>
4.3	TEST RUNS.....	20
4.3.1	<i>Descent Profile.....</i>	<i>20</i>
4.3.2	<i>Descent with no Lead Feedback</i>	<i>24</i>
4.3.3	<i>Descent with Lead Feedback</i>	<i>24</i>
4.3.4	<i>Descent with an N-2 Feedback</i>	<i>25</i>
4.3.5	<i>Test Runs with Initial Disturbance in the Fourth Aircraft.....</i>	<i>25</i>
5	RESULTS AND DISCUSSIONS	26
5.1	DISTURBANCE ONLY IN LEAD AIRCRAFT	27
5.1.1	<i>Descent with no Lead Feedback</i>	<i>27</i>
5.1.2	<i>Descent with Lead Feedback</i>	<i>33</i>
5.1.3	<i>Descent with N-2 Feedback</i>	<i>37</i>
5.1.4	<i>Comparing Results.....</i>	<i>39</i>
5.2	DISTURBANCE IN LEAD AND FOURTH AIRCRAFT.....	42
5.2.1	<i>Descent with Lead Feedback</i>	<i>42</i>
5.2.2	<i>Descent with N-2 Feedback</i>	<i>44</i>
6	CONCLUSIONS AND RECOMMENDATIONS.....	47
	References	50
	Appendices.....	52

List of Figures

Figure 4-1: Runtime Schematic of Program Architecture	19
Figure 4-2: Descent Profile in 2D Space	21
Figure 4-3: CAS Profile during Descent for Lead Aircraft	22
Figure 4-4: TAS Profile during Descent for Lead Aircraft.....	23
Figure 4-5: TAS and CAS vs. East Position	23
Figure 5-1: Separation Distance from Lead Aircraft (No Lead Feedback)	28
Figure 5-2: Separation Distance from Front Aircraft (No Lead Feedback).....	29
Figure 5-3: True Airspeed vs. Time (No Lead Feedback).....	31
Figure 5-5: True Airspeed vs. Time for 700 Seconds (no Lead Feedback).....	32
Figure 5-6: x-Direction vs. Altitude for all Aircraft (no Lead Feedback)	33
Figure 5-7: Separation Distance from Lead Aircraft (Lead Feedback)	34
Figure 5-8: Separation Distance from Front Aircraft (Lead Feedback)	35
Figure 5-9: True Airspeed vs. Time (Lead Feedback).....	36
Figure 5-11: Separation Distance from Lead Aircraft (N-2 Feedback)	37
Figure 5-12: Separation Distance from Front Aircraft (N-2 Feedback)	38
Figure 5-13: True Airspeed vs. Time (N-2 Feedback)	39
Figure 5-14: Separation Distance from Lead for Aircraft 8	40
Figure 5-15: Aircraft 8 Separation Distance from Aircraft 7	41
Figure 5-16: Separation from Front with AC 4 Offset (Lead Feedback)	43
Figure 5-17: Separation from Front with AC 4 Offset (N-2 Feedback)	45

Nomenclature

Current Aircraft: The aircraft in consideration.

Following Aircraft: All aircraft that have at least one aircraft ahead in the string.

Free-Flight: Flight that entails the usage of all airspace by aircraft instead of being restricted to airways.

In-Trail Aircraft: Aircraft that follow the same airway.

Self-Separation: The delegation of aircraft separation to the pilot

String of Aircraft: Multiple aircraft in cue along the same airway, with each separated by a similar distance.

Acronyms and Symbols

ACRONYMS:

A/C – Aircraft

ADS-B – Automatic Dependent Surveillance - Broadcast

ATC – Air Traffic Control

ATM – Air Traffic Management

CAS – Calibrated Airspeed

P-D – Proportional Derivative

LQR – Linear Quadratic Regulator

ODE – Ordinary Differential Equation

R/C – Rate of Climb

RTCA - Radio Technical Commission for Aeronautics

TAS – True Airspeed

SYMBOLS:

a_i – Acceleration of the Current Aircraft (and thus $i-1$ is the leading aircraft)

C_L – Lift Coefficient

$C_{L_{max}}$ – Maximum Lift Coefficient

C_D – Drag Coefficient

D – Drag

g – Acceleration Due to Gravity (32.2 ft/s^2)

h – Altitude

K_d – Derivative Gain used for velocity errors between current aircraft and front aircraft

K_p – Proportional Gain used for separation errors between current aircraft and front aircraft

K_{dL} – Derivative Gain used for velocity errors between current aircraft and lead aircraft

K_{pL} – Proportional Gain used for separation errors between current aircraft and lead aircraft

L – Lift

m – Mass of the Aircraft

p – Bank Angle Rate of Change
 S_w – Wing Span
 s_i – Horizontal Position of the i^{th} Aircraft
 s_D – Desired Spacing Distance
 T – Thrust
 v_i – True Airspeed of the i^{th} Aircraft
 V – True Airspeed
 V_s – Stall Speed
 W – Weight
 x – East Position
 y – North Position
 γ – Flight Path Angle
 ρ – Atmospheric Density
 ϕ – Bank Angle
 ψ – Heading Angle
 ω_n – Natural Frequency
 χ_{wind} – direction of the wind

1 Introduction

1.1 *Background and Motivation*

Guidance and route planning of all aircraft are the responsibilities of air traffic controllers. Their job is one of stress and high-energy, and with little to no margin for error. The air travel industry is continually growing and demanding for increases in air traffic. Obviously, any increase in air traffic will only add to the stress on an air traffic controller and increase their potential for error. The current infrastructure therefore needs to be updated to allow for these demands. Many have suggested that the best solution is to delegate part of the task of aircraft separation to the pilot. This is meant to relieve the stress of the air traffic controller's task as well as increase available airspace.

The question is: how can pilots safely maintain separation? First and foremost, they must know where all aircraft in their vicinity are in space. This knowledge can be conveyed via radio broadcast as in ADS-B – as used in this thesis. Furthermore, pilots need tools to aid them in maintaining separation. Research has been done in exploring the possibilities of self-separation and/or free-flight, including doing experimental simulations with flight simulators and real pilots. The main purpose of this thesis is to create a tool to assist pilots in the task of self-separation. This is done using a numerical simulation of aircraft dynamics and controls for a string of in-trail aircraft. These aircraft will follow pre-determined airways but the main task of separation maintenance will be done using ADS-B communicated data and separation controllers.

1.2 Research Goals and Approach

The goal of this project is to examine the control of a string of aircraft, with each aircraft in the string following the aircraft in front of it. The idea is to take the control laws designed by Slater [15] and to modify it to include dynamics of the aircraft. From this, a program using a technical computing language can numerically simulate the flight of a stream of in-trail aircraft in a descent profile. The approach taken was to use MATLAB as the chosen technical computing language and to create an aircraft model using aircraft equations of motion. To implement the control laws, knowledge of aircraft state vectors in the string is needed, and thus the use of ADS-B as a state vector broadcast is explored. Some of the limitations of this system are also included in the simulation for more realistic results.

1.3 Thesis Outline

Chapter two details the state of the art by examining the existing literature on previous research done in the fields of free-flight and self-separation. Chapter three lays out the dynamics and controls used in the design of the simulation program. Chapter four goes over the design and architecture of the program and also the flight runs tested. Chapter five details the results obtained, and gives a short discussion on each of those results. Chapter six concludes the thesis and offers recommendations. The references used and cited are to be found at the end of this thesis.

The appendices, found after the references, hold all the code written for the program as well as the data files. Nomenclature, acronyms and symbols are listed at the beginning of this thesis.

2 Literature Review

2.1 *Self-Spacing Dynamics*

For this thesis, the study of self-spacing dynamics is the most important. The work done in this thesis was mainly based on the following two works.

Hoffman, Ivanescu and Zeghal [6] evaluated the performance of in-trail following aircraft using a guidance law and Automatic Dependent Surveillance – Broadcast (ADS-B) data characteristics. The guidance law in this paper is designed to maintain spacing from the lead aircraft. This is tested with two separate spacing criteria: one by maintaining constant time spacing and the other by maintaining constant distance spacing. The control law used is a proportional-integral (PI) controller to prevent steady-state error in following distance. The test scenarios used include a string of five aircraft undergoing a descent and each following their preceding aircraft. The ADS-B information is modeled with update rate, latency and accuracy. The results showed that the constant time spacing criteria responds better and is more robust to the ADS-B characteristics.

Slater's [15] proposed model of self-separation is a mathematical approach; applying control laws to a string of vehicles in a simplified 1-D model. This paper presents a new concept of analyzing longer strings of aircraft for self-separation, rather than just pairs of aircraft. It looks at two separate proportional-derivative (P-D) control laws, one that has each aircraft only following the aircraft directly ahead of it, and another that incorporates lead vehicle feedback. It is assumed that all aircraft know where the lead aircraft is at all times. The results show an improved response for the lead vehicle feedback control law. One purpose of this thesis is to use the control laws presented in this paper and to expand on these ideas by incorporating aircraft dynamics.

2.2 Simulated Self-Spacing

Other notable research includes experimentally simulated self-spacing and/or free-flight. These required pilots and flight simulators. They allowed pilots, using their own judgment and any series of rules and regulations, to maintain separation, given the location of the other aircraft in the string.

Cashion, Lozito, Mackintosh, and McGann [1] used simulations to examine communication and procedural issues associated with self-separation in the en-route environment. These simulations were conducted in a Boeing 747-400 simulator which included extra features in its flight deck display to allow for increased traffic and maneuvering information to the flight crew. Furthermore, a new alerting logic was designed for this purpose. Ten flight crews were employed and each flew eight different scenarios with different conflict types. The results of this study are meant to be a guideline for further study in the implementation of self-separation.

Grimaud, Hoffman and Zeghal [3] proposed flight crew separation assurance in the en-route airspace as well as in the terminal area. The task of speed control for maintaining separation distance is allotted to the flight crew. Furthermore, merging operations are investigated. The suggested solution to these two operations is a combined effort between controller and pilot. The pilot, having been given increased situational awareness, communicates with the controller about what task to undertake, and then performs the maneuver independently. This results in a reduction of instructions given, reducing the task of the ATC.

Pritchett and Yankosky [12] focused on involving pilots to a greater extent in guidance while keeping the controller in charge of Air Traffic Management (ATM). This means that the pilot is given a higher-order command from the ATC and is responsible for performing lower-level guidance. In this study, the pilot is given either of two operations: in-trail spacing, and merging into a string of in-trail aircraft. This was tested experimentally. The experiment utilized twelve airline pilots flying arrival routes in a simulator, with each pilot being responsible for maintaining a commanded relative

position within the traffic flow. The results were positive, and modifications in the cockpit avionics system were recommended.

2.3 Conceptual Solutions to Self-Spacing

Finally, work has been done on theoretical concepts regarding air traffic management and free-flight and/or self-spacing. These studies lack experimental and analytical results but bring innovative ideas to the forefront.

Chang, Livack and McDaniel [2] explored new technology and procedures that might enhance aviation safety as well as enable free-flight. They suggest that the solution to achieving free-flight is to implement digital avionics to improve the quality of communications available to the pilot and the ATC. This way, the aircraft can go in any direction and the ATC will always know its position while passively monitoring its passage through space. If conflict should arise, the ATC can redirect traffic accordingly. No analysis or simulations were performed to justify the plausibility of this solution.

Hoffman and Zeghal [5] examined implementing different levels of delegation-giving the pilots different levels of autonomy dependent on the type and phase of flight. Their study was also theoretical with no experimental or analytical results.

Kuchar and Schild [8] suggested the need for maneuver coordination rules to decentralize the current system and allow for more self-separation environments. In this decentralized system, each aircraft in a conflict must inter-communicate to come to a resolution. This is done using a set of designed rules. These rules outline how a pilot must determine priority to resolve conflict. They then evaluated their rules using simulations of different conflict situations. The results show that higher rule complexity result in better safety and maneuver efficiency.

2.4 Overview

The main focus of this thesis is to develop control laws to aid in separation maintenance of long strings of aircraft in a descent. Therefore the biggest influence on all the work done in this project comes from Slater's work [15]. Additionally, Hoffman, Ivanescu and Zeghal's [6] study was helpful in seeing how one may use dynamics as well as ADS-B characteristics in aircraft separation during a descent. The descent model used was at a higher altitude and faster speeds than what was chosen for this project. They came to the conclusion that a constant time separation was more effective in this scenario, but they did not include lead feedback in their controller design. One of the main goals of this project is to analyze the effects of lead feedback in separation maintenance.

All other work done with respect to free flight and self separation has been an analysis of limited delegation to the cockpit. They suggest a coordinated effort for conflict resolution. Experimental studies focus mainly on giving pilots increased situation awareness to combine efforts of cockpit and controllers in ATM. These are important concepts to consider as rules need to be included in the implementation of mathematical control laws for self-separation. This is because the control laws implemented cannot account for all eventualities, and the close monitoring of ATC will always be necessary.

3 Dynamics and Control

The simulation of all aircraft in this work was programmed in MATLAB. This function based language allowed for the computationally intensive calculations required by this project. A detailed description of the program can be found in section 4.2 and all MATLAB code of the program can be found in the appendix. The functions created were based on the dynamics and controls detailed in this chapter.

The first step to creating the lead aircraft's flight path consists in creating a reference path for it to follow. This reference path is created using segments of constrained equations of motion. The constraints consist of maintaining one to several variables constant. This ideal path is then followed using a linear quadratic regulator (LQR) controller and linearized equations of motion. Furthermore, to make the linearized equations realistic, control limits are set. The final result is the lead aircraft's flight.

All following aircraft follow a slightly different algorithm. First, each aircraft uses separation controls via a proportional-derivative (P-D) controller to minimize separation and velocity errors. There are two control schemes tested in this project. The first scheme attempts to minimize errors between each aircraft and the aircraft directly ahead of it, and the second scheme incorporates into the control algorithm errors from the lead aircraft in the string. With these controls in place, each following aircraft can set up a reference path. This is followed using the LQR controller.

This section covers all the equations of dynamics and controls used to make this simulation possible. The control limits used are also included. Finally, all symbols used in these equations are identified in the "Acronyms and Symbols" section.

3.1 Aircraft Dynamics

The aircraft dynamics model uses the equations of motion based on the three-dimensional point-mass differential equations, and are taken directly from [6]. These state equations are as follows:

$$\dot{V} = \frac{T - D}{m} - g \cdot \sin \gamma \dots\dots\dots(1)$$

$$\dot{h} = V \cdot \sin \gamma \dots\dots\dots(2)$$

$$\dot{x} = V \cdot \cos \gamma \cdot \cos \psi - V_{wind} \cdot \cos \chi_{wind} \dots\dots\dots(3)$$

$$\dot{y} = V \cdot \cos \gamma \cdot \sin \psi - V_{wind} \cdot \sin \chi_{wind} \dots\dots\dots(4)$$

$$\dot{\gamma} = \frac{L}{m \cdot V} \cdot \cos \phi - \frac{g}{V} \cdot \cos \gamma \dots\dots\dots(5)$$

$$\dot{\phi} = p \dots\dots\dots(6)$$

$$\dot{\psi} = \frac{g \cdot \tan \phi}{V} \dots\dots\dots(7)$$

where V is the true airspeed, T is the thrust, D is the drag, m is the mass of the aircraft, g is the acceleration due to gravity (32.2 ft/s²), γ is the flight path angle, h is the altitude, x is the east position, χ_{wind} is the direction of the wind, y is the north position, L is the lift, ϕ is the bank angle, p is the bank angle rate of change, and ψ is the heading angle. A dot appearing above a variable symbolizes its derivative (rate of change).

The wind conditions have been neglected in this study as well as the change in mass due to fuel consumption.

3.2 Reference Path Finder for Lead Aircraft

To create a reference path for the lead aircraft, conditions were imposed on the state equations to allow the aircraft to maintain constant conditions in two states during a segment of flight. For example, if the aircraft was to descend at a constant flight path angle and constant calibrated airspeed (CAS), the simulation would impose $\dot{\gamma}$ to equal zero and instead of solving equation (1), it would impose that the velocity be maintained at constant CAS. The user can then choose to create any reference flight profile by placing segments one after the other and imposing desired initial conditions as well as the values of the states that the user wants to maintain constant. Since there are two control variables in the reference path following controls (see section 3.4), two constraints can be imposed to solve for the controls required to fly the trajectory.

The segments available in the simulation are as follows:

- 1- Constant CAS and flight path angle,
- 2- Constant CAS and rate of climb (R/C),
- 3- Constant thrust setting and flight path angle,
- 4- Constant thrust setting and R/C,
- 5- Constant CAS and altitude,
- 6- Idle thrust (not constant) and constant flight path angle,
- 7- Idle thrust and constant R/C.

A reference path consisting of combinations of these segments is then followed by the leader using the reference path following controls which are detailed in section 3.4.

3.3 Aircraft Following Controls

All following aircraft also need a reference path to follow. This reference path must maintain proper separation from the lead aircraft. To determine the commanded velocity and horizontal position of each following aircraft, a continuous proportional-derivative (P-D) controller is used. For this controller, there are two different set-ups

used and compared in this study. One uses feedback from the aircraft directly ahead, the other combines that data with feedback from another leading aircraft in the string.

To mimic the airway followed by all aircraft, a space reference path is created. It maps out the spatial position and orientation (horizontal position, altitude, flight path angle, bank angle and heading angle) of the airway in advance. During the simulation, the P-D controller will determine at each time interval the velocity of the aircraft it is flying. From this, a horizontal position can be extrapolated. Using the pre-determined space reference path, the three dimensional position and orientation of that aircraft can then be mapped out. This creates a full reference state at that moment for that aircraft.

This system assumes that all aircraft closely follow the pre-determined airway. Therefore if there are any major deviations in vertical positioning in an aircraft in the string, this control system will fail and it will be required to have an air traffic controller placed into the loop for security reasons.

3.3.1 Continuous Separation Control

The P-D control law of the i^{th} aircraft in the string was taken from [15] and is as follows:

$$a_i(t) = K_p(s_{i-1} - s_i - s_D) + K_d(v_{i-1} - v_i) \dots\dots\dots(8)$$

where a_i is the commanded acceleration of the current aircraft, and $i-1$ is the aircraft immediately ahead, K_p is the proportional gain, K_d is the derivative gain, s_i is the commanded horizontal position of the current aircraft, s_D is the desired spacing distance, and v_i is the true airspeed of the current aircraft.

3.3.2 Continuous Separation Control with Lead Feedback

In some cases it is desirable to use an aircraft further ahead in the string (the j^{th} aircraft) in addition to the immediately preceding aircraft in the P-D controller. The P-D control law used for incorporating lead feedback was taken from [15] and is as follows:

$$a_i(t) = K_p(s_{i-1} - s_i - s_D) + K_d(v_{i-1} - v_i) + K_{pL}(s_j - s_i - (i - j)s_D) + K_{dL}(v_j - v_i) \dots (9)$$

where, in addition, K_{pL} is the proportional gain for the control of separation distance between the i^{th} aircraft and the j^{th} aircraft and K_{dL} is the derivative gain for the control of velocity between the i^{th} aircraft and the j^{th} aircraft.

3.4 Reference Path Following Controls

Each aircraft has a reference path to follow, as previously determined. An LQR control law is applied to each flight path for trajectory following using linearized equations of motion. These equations of motion are:

$$\delta \ddot{x} = A \delta \dot{x} + B \delta u \dots (10)$$

Where:

$$\delta \dot{x} = [\delta \dot{v} \quad \delta \dot{h} \quad \delta \dot{x} \quad \delta \dot{y} \quad \delta \dot{\gamma} \quad \delta \dot{\phi} \quad \delta \dot{\psi}]^T \dots (11)$$

$$\delta x = [\delta v \quad \delta h \quad \delta x \quad \delta y \quad \delta \gamma \quad \delta \phi \quad \delta \psi]^T \dots (12)$$

$$\delta u = [\delta T_c \quad \delta L_c]^T \dots (13)$$

$$A = \begin{bmatrix} -2\cos\phi_o \cos\gamma_o \left(\frac{D}{L}\right)_o \frac{g}{v_o} & 0 & 0 & 0 & -g \cos\gamma_o & 0 & 0 \\ \sin\gamma_o & 0 & 0 & 0 & v_o \cos\gamma_o & 0 & 0 \\ \cos\gamma_o \cos\psi_o & 0 & 0 & 0 & -v_o \sin\gamma_o \cos\psi_o & 0 & -v_o \cos\gamma_o \sin\psi_o \\ \cos\gamma_o \sin\psi_o & 0 & 0 & 0 & -v_o \sin\gamma_o \sin\psi_o & 0 & v_o \cos\gamma_o \cos\psi_o \\ 2\cos\gamma_o \frac{g}{v_o^2} & 0 & 0 & 0 & \frac{g}{v_o} \sin\gamma_o & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -\frac{g}{v_o^2} \tan\phi_o & 0 & 0 & 0 & 0 & \frac{g}{v_o} \sec^2 \phi_o & 0 \end{bmatrix} \quad (14)$$

$$B = \begin{bmatrix} \frac{g}{W} & \frac{g}{L_o} \frac{C_D}{C_L} \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & \frac{g}{W \cdot v_o} \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \dots\dots\dots (15)$$

The weighting matrix Q in the LQR performance index was selected as follows:

$$Q = \begin{bmatrix} 10 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1000 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 5000 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 5000 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.01 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.01 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.01 \end{bmatrix} \dots\dots\dots (16)$$

The R matrix is a two by two identity matrix.

Here, the δT_c is the delta thrust control and δL_c is the delta lift control. These are generated by $\delta u = -K \cdot \delta x$, where K is the LQR optimal gain matrix. The gains in the Q matrix were selected such that they simulated realistic error intervals.

Using these controls, aircraft are able to follow their previously calculated reference trajectories. To get a clear idea of the algorithm used, refer to section 4.2.3.

3.5 Control Limits

Since the dynamics of this system are linearized, control limits were set in place to allow for realistic results. These limits were set with the goal of realistically modeling a representative transport aircraft.

First, the acceleration command was limited. For these tests the acceleration of the vehicle (and equally the deceleration) was set to an absolute value of 1.6 ft/s^2 or $0.05g$'s, which is a comfortable acceleration for passengers.

Furthermore, velocity limits were set. Although the chosen limits were a rough approximation, a lower limit was set to prevent stalling. This was determined as follows:

$$V_s = \sqrt{\frac{2 \cdot W}{C_{L_{\max}} \cdot S_w \cdot \rho}} \dots\dots\dots(17)$$

Where V_s is the stalling airspeed, W is the weight, $C_{L_{\max}}$ is the maximum lift coefficient, S_w is the wing area and ρ is the atmospheric density. The $C_{L_{\max}}$ chosen for the simulation was 1.4.

4 Numerical Simulation

4.1 ADS-B Simulation

ADS-B broadcasts a state vector and intent [13]. To simulate the short-term intent, the MATLAB program attempts to maintain constant acceleration between every broadcast. Furthermore, the simulation has a path which is already pre-determined, which mimics the state position intent. The maximum errors present in the state broadcasted by ADS-B are a standard deviation (σ) of 0.25 ft/s for horizontal velocity and a σ of 20 m (~ 65.6 ft) for horizontal position. In terms of the magnitudes of velocity and position involved, these errors are minimal and do not affect the ability of the aircraft to maintain safe separation. Therefore these errors are neglected in the simulation.

The RTCA recommends that the state vector be broadcast every 6 seconds. To simulate this, the program has the follower aircraft receive data from the aircraft every 6 seconds and, as previously stated, assumes the acceleration generated by the P-D controller during the broadcast phase remains constant in between these broadcasts. It is assumed that each aircraft will broadcast its state vector at non-synchronous intervals for the control scheme that does not incorporate lead feedback. To simulate the non-synchronous characteristic of these broadcasts, each aircraft in the string receive data out of phase from each other. However, in the control scheme that does incorporate lead feedback, it is necessary for broadcasts to be synchronous as each following aircraft must receive data from two aircraft at the same time. This can only be done if these aircraft are synchronized in their broadcasts and therefore in this simulation they are assumed to be so.

4.2 MATLAB Program Design and Architecture

4.2.1 Lead Aircraft Reference Path Design Requirements

The first requirement for the lead aircraft reference path creator was that it could integrate trajectory segments for different conditions, as stated in section 3.2. There should also be stopping conditions that can be set by the user to allow the program to know when to move to the next segment in the profile. The required stopping conditions are as follows:

- 1- Stop after a certain amount of time (in seconds) has passed since the start of the segment
- 2- Stop once you have reached a certain climbing altitude (in feet)
- 3- Stop once you have reached a certain descending altitude (in feet)
- 4- Stop after you have slowed down to a certain velocity (in ft/s)
- 5- Stop after a maximum heading angle (in radians) has been reached
- 6- Stop after a certain x-distance (in ft) has been reached
- 7- Stop after reaching a minimum bank angle (in radians)
- 8- Stop after accelerating to a certain velocity (in ft/s)

Furthermore, all the data must be recorded at some discrete time interval to then track and report the state vector of the lead aircraft to all following aircraft. Finally, these segments must then be concatenated to form a flight profile for the lead aircraft to follow.

4.2.2 Follower Aircraft Reference Path Design Requirements

The follower reference path design required two main sections: (1) to implement the follower control law laid out in section 3.3, and (2) to map that newly attained horizontal position into three-dimensional space. To create the three-dimensional “map”, the same program as the lead aircraft reference path creator can be implemented over a larger span to map the three-dimensional path of all the following aircraft in space.

4.2.3 Reference Path Follower Design Requirements

Both lead and following aircraft are required to follow their respective reference path. The main requirement in designing the reference path follower program is to implement the LQR controller outlined in section 3.4. This must be done after every time interval and recorded to a file.

4.2.4 Program Architecture

Figure 4-1 at the end of this section is a run-time schematic of the program architecture. It is a simplified view of the dynamic operation of the simulation. Each box represents a function. Each arrow represents a function invocation. The horizontal ellipses represent multiple iterations of the function call. The vertical ellipsis represents the missing function calls in between the two functions that it links. Refer to the appendices for all the code in this program. All function names in this program begin with FPS_ for “Flight Path Simulator”. They are all collected in a folder that can be added as a MATLAB toolbox.

Several drivers were created to run different test scenarios. Each driver invokes all the main function blocks in the program. They do not perform any task on their own except to plot out results obtained from the functions that they call. The purpose of a driver is to replace a user interface and is mainly for testing the program functions. All input variables in the program are set in the driver. The driver runs all the main function blocks: the two reference path creators (one leader, one space path) and every aircraft function in the string. The output of all these functions are data files in the form of matrices. The data in these matrices are built from state vectors over time. These state vectors are objects, with the following variables: time, velocity, altitude, x-position, y-position, flight path angle, bank angle, heading angle, drag, lift, thrust, and weight. There is also a last slot in this state object that allows for any input that needs to be passed down from one file to the other. This slot is called “mode” and can be used to store the mode in which the aircraft is currently flying. Once the data files have all been created, the driver

calls to these files and plots the results. This can also be done on the command line or in another m-file but are put into the driver for ease of use.

The first function called by the driver is the leader reference path. The function used to create all reference paths is called `FPS_FlightPath`. This function checks to see how many segments are input, and then does iterations to calls to the segment function `FPS_segment` for each segment. Each segment represents different flight conditions, as outlined in section 3.2. The flight path function feeds each segment with the following information: the data file name, the initial conditions, the time intervals at which to write data, the stopping conditions as outlined in section 4.2.1, and all relevant data pertaining to the flight condition (e.g. if it wanted to maintain constant CAS of 300 [kts] and constant flight path angle of 0.01 [rad], the flight condition of constant CAS and constant flight path angle must be inputted, as well as 300 [kts] and 0.01 [rad]).

The segment function runs the step function until the stopping condition has been met. The step function, `FPS_step` in turn runs the integrator for the equations of motion outlined in section 3.1, but these equations are constrained by the flight conditions imposed on them and as such there are many different aerodynamic model functions in the program. Each flight condition is represented by a different aerodynamic model function. To continue the previous example, a constant CAS and constant flight path angle flight condition would force $\dot{v}$ to equal zero, v to equal the TAS for that particular CAS at the given altitude. Note that this is not making the TAS constant, but bypassing the integration by specifying the desired TAS. Also, in descending or ascending constant CAS flight, $\dot{v}$ is not zero, but for the purposes of calculation $\dot{v}$ is assumed to be negligible. Furthermore, it would force $\dot{\gamma}$ to equal zero and γ to equal 0.01 [rad]. The step function would then run that particular “constrained” set of equations of motions in its integrator, writing its data at every specified time interval and checking the stopping conditions before proceeding on to the next time interval. Once a stopping condition has been met, the program continues on to the next segment or is terminated.

The next function that is called by the driver is the lead flight path, called `FPS_leadFlightPath`. The purpose of the lead flight path function is to compare its

position with its desired position. This desired position is what has been created in the reference path file and stored away as data. The lead flight path function calls to this data, finds the delta errors and inputs these into the LQR controller outlined in section 3.4. The controller feeds back the control settings and adjusts the lead aircrafts positions to stay on target.

The space reference path block uses the same `FPS_flightPath` function as the leader reference path block, but has an extra segment in the beginning to account for the path of all the follower aircraft in the string.

Finally, the driver calls to all the following aircraft in the string, each run by the either of two functions: `FPS_followPath` and `FPS_followPathLFB`. The additional LFB appended to the function name stands for “Lead FeedBack”. Both functions perform the same task but the latter uses the lead feedback controller and the former uses the controller without lead feedback. The followPath function performs several tasks. First, to simulate ADS-B characteristics, the function checks to see if it has received data from the lead aircraft. If it has, then it runs the P-D controller specified by its name to determine its velocity. Knowing its velocity, it can then determine its horizontal reference position. From this it can use the space reference path to determine its three dimensional reference position. From this reference state, the follow path function employs the same reference path follower integrator as the leader to control its flight. However, if the follow aircraft does not receive state information from the lead aircraft at that particular time interval, it attempts to maintain constant acceleration, creates a reference path, and uses the LQR controller to determine its flight path.

Each aircraft creates its own data file that can be used to plot and analyze results.

Figure 4-1: Runtime Schematic of Program Architecture

4.3 Test Runs

4.3.1 Descent Profile

The tested descent trajectory for the lead aircraft is a typical descent profile and is comprised of four segments with the following characteristics:

Segment 1:

- Descent at a constant -3° angle,
- Constant CAS flight at 280 kts,
- Begin at an altitude of 15,000 ft,
- End at an altitude of 10,000 ft.

Segment 2:

- Level flight,
- Idle thrust setting,
- Decelerate to 250 kts TAS.

Segment 3:

- Descent with constant rate of sink of 1,000 ft/min,
- Idle thrust setting,
- End at 5,000 ft.

Segment 4:

- Level flight,
- Idle thrust setting,
- Decelerate to 180 kts CAS.

The heading of this profile is due east (moving positively in the x-direction). The following aircraft starts $(i-1) \cdot s_D$ away and maintains separation and a constant altitude of 15,000 ft before starting the descent. After completing the descent, all aircraft (who arrive at that position) continue at a constant altitude of 5,000 ft. Figure 4-2 maps the descent profile in the x-h space. Figure 4-3 maps the reference descent CAS profile for

the lead aircraft (not for the followers as they want to match TAS, not CAS). Figure 4-4 maps the reference descent TAS profile.

Figure 4-2: Descent Profile in 2D Space

Figure 4-2 shows the descent profile in 2-dimensional space. The descent always starts at 70 [nmi] due east of the defined starting point. This is because the s_D was chosen to be 10 [nmi] and the test runs had strings of 8 aircraft. This means that the first aircraft starts $7 \cdot N$ away from the eighth aircraft in the string, which starts at the zero position in x-y space. Prior to the descent, all following aircraft maintain level flight at 15,000 [ft] and vary their velocities to maintain separation. After the descent, all aircraft maintain level flight at 5,000 [ft], as shown, for the duration of the tests run.

Figure 4-3: CAS Profile during Descent for Lead Aircraft

Figure 4-3 plots the CAS profile during the descent for the lead aircraft. The lead aircraft begins in a descent, which ends at around 690 seconds. At the end of the descent, the lead aircraft maintains a constant CAS of 180 [kts], as mentioned before. Since the CAS is dependent on the altitude, only the lead aircraft maintains constant CAS for the first segment of the descent, since the following aircraft at that time are all maintaining level flight. Following aircraft attempt to maintain a similar TAS profile to the lead aircraft (see Figure 4-4).

Figure 4-4 plots the TAS variations of the lead aircraft during its descent. Note that the lead aircraft does not maintain constant TAS until the end of the descent, when the TAS is constant because the CAS is constant and the aircraft is flying level (it continues constant for the remainder of the flight- not shown in this time interval).

Figure 4-4: TAS Profile during Descent for Lead Aircraft

Finally, to be able to compare with Figure 4-2, the TAS and CAS are plotted against the east position in Figure 4-5.

Figure 4-5: TAS and CAS vs. East Position

4.3.2 Descent with no Lead Feedback

The first test run was a descent with continuous spacing control and no lead feedback, as detailed in section 3.3.1. All aircraft start with the same initial conditions except for their initial x-position, which is always $(N-i) \cdot s_D$, where i is the aircraft number in the string and s_D is the separation distance. To create some disturbance in the system, the lead aircraft starts with an initial disturbance. This is done by starting it off 10 knots faster than the other aircraft in the string. The simulation is run with a string of eight aircraft. The test is run for 2,032 seconds (almost 34 minutes). This allows all aircraft to complete the descent.

The gains selected are as follows: a proportional gain (K_p) of $1/30^2$ [s^{-2}] such that the natural frequency of the response, which is $\omega_n = \sqrt{K_p}$, or $1/30$ [rad/s], and a derivative gain (K_d) of $2/30$ [sec^{-1}] such that the damping ratio, which is $\zeta = \frac{K_d}{2 \cdot \omega_n}$, is 1.

4.3.3 Descent with AC 1 Feedback

The second test run was a descent with continuous spacing control and the first aircraft in the string (AC 1) feedback, as detailed in section 3.3.2. The initial conditions of all the aircraft are the same as for the case without lead feedback. This run also includes the initial disturbance of +10 knots for AC 1. It is also run with a string of eight aircraft, for the same time interval as the first test.

The gains selected are as follows: a proportional gain for the front aircraft error (K_p) of $1/30^2$ [s^{-2}], and a derivative gain for the front aircraft error (K_d) of $2/30$ [s^{-1}]. Furthermore, the proportional gain for the lead aircraft error (K_{pL}) of $1/30^2$ [s^{-2}], and a derivative gain for the lead aircraft error (K_{dL}) of $\frac{2\sqrt{2}-1}{30}$ [s^{-1}]. This allows for a natural frequency of $\frac{\sqrt{2}}{30}$ [rad/s] ($\omega_n = \sqrt{K_p + K_{pL}}$) and a damping ratio of 1 ($\xi = \frac{K_d + K_{dL}}{2 \cdot \omega_n}$).

4.3.4 Descent with an N-2 Feedback

The third test run was also a descent with continuous spacing control and lead feedback. It uses the same controls and gains used by the lead feedback test case, as detailed in sections 3.3.2 and 4.3.3. AC 1 starts off with the same initial disturbance, and the test runs for the same time interval. However, instead of following AC 1 and the front aircraft, each following aircraft follows its two preceding aircraft. Therefore, the front aircraft for all cases remains the same, but the N-2 aircraft will always become the “lead” aircraft. This new lead feedback scheme is referred to in this thesis as an “N-2 Feedback” scheme. This is because the lead aircraft in this scheme is always the N-2 aircraft.

This case results in having the first three aircraft in the string fly identically to the ones in the AC 1 feedback case. What is important here is to analyze the behavior of aircraft further back in the string (i.e. from the fourth aircraft on).

4.3.5 Test Runs with Initial Disturbance in the Fourth Aircraft

The two tests with lead feedback were run again, but this time, in addition to the initial disturbance of +10 knots in AC 2, aircraft 4 starts off at 41 nautical miles instead of 40 nautical miles. This is primarily to see whether the N-2 feedback case would have an advantage over the AC 1 feedback in the situation where disturbances are present in any vehicle further in the string, not just in AC 1. Furthermore, this offset was chosen to see how well following aircraft can maintain separation from a lead vehicle which has violated the separation distance.

The gains chosen and the time interval of the runs remained the same as with the first set of tests. The string also consisted of eight aircraft.

5 Results and Discussions

The simulated descent results for all test runs are shown in this section. Each aircraft is labeled using AC and a number representing its position in the string, starting from the first aircraft in the string, or AC 1. All aircraft are color-coded: aircraft 1 is always red, aircraft 2, blue, aircraft 3, green, aircraft 4, magenta, aircraft 5, cyan, aircraft 6, black, aircraft 7, yellow and finally aircraft 8, orange.

The most important parameter to be analyzed in the project is the ability for the aircraft to maintain safe distances during the descent profile. There were two different distances looked at in this study. The first is the distance each aircraft maintains from AC 1. This is calculated simply as follows:

$$(s_{current})_k = \sqrt{((x_{current})_k - (x_{lead})_k)^2 + ((y_{current})_k - (y_{lead})_k)^2} \dots\dots\dots(18)$$

Where $s_{current}$ is the distance that the current aircraft (the one that is currently being analysed) maintains from AC 1, $x_{current}$ is the x-position of the current aircraft, x_{lead} is the x-position of AC 1, $y_{current}$ is the y-position of the current aircraft, and y_{lead} is the y-position of AC 1. Equation 18 is used to calculate the distance at each time interval, denoted by ‘k’.

The second important distance is the one each aircraft maintains from its front aircraft in the string. This is calculated as in equation 18 but using the values from the N-2 aircraft instead of the values from AC 1.

The second parameter analyzed is the variation of velocity over time. It is important to make sure that no aircraft is exhibiting strange behaviors in their velocity profiles to enable them to maintain separation. The analysis of these curves is important in determining the feasibility of that flight. Note that the control limits are included in running the simulation (as detailed in section 3.5).

As another visual check, the flight profile of all aircraft in 2D space is mapped out. This shows whether all aircraft are correctly following their pre-determined airway and not deviating from this path.

Finally, results from different tests are plotted on the same plot to emphasize their differences.

5.1 *Disturbance only in AC 1*

The first set of simulations was run with an AC 1 initial disturbance of +10 knots. The first and second aircraft in the string of all test runs are flown in exactly the same way. Differences between the schemes only occur after AC 3. No other disturbances are included in any following aircraft- it is assumed that they comply with the control requirements unless the physical limits set are reached.

5.1.1 Descent with no Lead Feedback

Figure 5-1 and Figure 5-2 show the distance of each following aircraft from AC 1 and from their preceding (N-1) aircraft, respectively, for the case of a descent with no lead feedback.

Figure 5-1: Separation Distance from Lead Aircraft (No Lead Feedback)

It can be seen from Figure 5-1 that the further an aircraft is from AC 1, the more disturbances it undergoes. In this figure, the scale is such that smaller oscillations and offsets are not noticeable. Regardless, all aircraft do maintain proper separation.

Figure 5-2: Separation Distance from Front Aircraft (No Lead Feedback)

Although Figure 5-1 gives a good idea of how far aircraft are from each other, it is hard to tell just how far they are from their front aircraft, as well as any oscillations or offsets that may be present in their paths. Figure 5-2 plots the separation distance of all aircraft from their preceding (N-1) aircraft. In terms of safe distances, no aircraft violate regulations. However, by AC 6, more erratic behavior is exhibited. AC 6 approaches AC 5 by the closest distance, at around 9.68 [nmi]. It is also the first to not return to maintaining the initial separation of 10 [nmi]. The slight offset from the 10 [nmi] datum exhibited in AC 2 through 5 is less than 0.01 [nmi] (about 60 [ft]) is negligible and is a precision error due to the step size used in the simulation for the controllers.

A pattern in the separation of each aircraft from their lead is seen from aircraft 3 on. The initial offset in AC 1 does affect the separation but only in the first few minutes of the flight, and as such is not seen in Figure 5-2. The two major dips each correspond

to the two big velocity changes in the velocity profile of AC 1, as can be seen in Figure 5-3 and in more detail in Figure 5-4. Note that after each aircraft in the string, the dip becomes more and more pronounced, as the aircraft try to follow their preceding aircraft.

The important parameter in terms of air traffic safety is the separation between adjacent aircraft. FAA regulations stipulate that aircraft should never infringe upon a separation distance of 5 nautical miles [11]. All aircraft in this string maintain safe separation for the entire duration of this test run. Furthermore, since no aircraft in the string deviates severely from its initial separation from the lead aircraft, an initial separation distance of 5 [nmi] can also be safely achieved. However, major disturbances within the string may cause aircraft to violate regulations and as such it is preferred that aircraft are positioned 7 or 10 [nmi] apart.

The next important parameter to observe is the variation of TAS over time. In Figure 5-3, it can be seen that all following aircraft follow a similar change in velocity. The following aircraft initially accelerate at a faster rate than the lead aircraft, which results in the downward dip in the separation distances seen in Figure 5-2. From then they attempt to “catch up”, with each subsequent aircraft lagging further and further behind. AC 6, 7 and 8 do not behave as smoothly, and do not settle in the tested time interval. This erratic behavior- especially that of AC 8- points to an inadequacy in this control system.

For a closer look of the variations in velocity during the descent, Figure 5-4 shows the velocity profile for the first 700 seconds.

Figure 5-3: True Airspeed vs. Time (No Lead Feedback)

Figure 5-4 is a closer look at the variation of TAS over time, by looking at a shorter time interval. The time interval chosen was the first 700 seconds, where all major velocity changes should occur. This is because during this time AC 1 is undergoing its descent (see Figure 4-4) and is changing velocities to accommodate that descent. At around 580 seconds, AC 1 settles to a CAS of approximately 180 [kts] (which at its altitude is approximately a TAS of 190 [kts]). Since following aircraft are trying to match the AC 1's velocity profile, they also settle at a TAS of 190 [kts] soon after, with, on the most part a lag of around 200 seconds.

Figure 5-4: True Airspeed vs. Time for 700 Seconds (no Lead Feedback)

Looking carefully at the start of the lead profile, one can see the initial disturbance of 10 knots. This is in part what causes aircraft 1 to initially have a greater separation from its front aircraft in the beginning of the test run.

Another thing to observe is the offset in the velocity profiles from one aircraft to its following aircraft for aircraft 3 to 8. This offset is about 30 seconds. This delay in response is a result of the gains chosen for this control system. Also note that aircraft 6 through 8 “bottom out” on the velocity profile figure at around the 650 [s] mark; this is due to the velocity limit set upon them.

Figure 5-5: x-Direction vs. Altitude for all Aircraft (no Lead Feedback)

Figure 5-5 illustrates the path undertaken by all aircraft in the string. It is shown from the 40 [nmi] east mark since before that mark all aircraft maintain an altitude of 15,000 [ft]. It can be seen that all aircraft maintain proper course and do not deviate from the airway. Aircraft 7 and 8 do not initiate a descent in this time interval, and aircraft 6 only descends to about 12,000 [ft].

5.1.2 Descent with AC 1 Feedback

Figure 5-6 and Figure 5-7 show the distance of each aircraft from AC 1 and from their N-1 aircraft, respectively, for the case of a descent with AC 1 feedback.

Figure 5-6: Separation Distance from Lead Aircraft (Lead Feedback)

Looking at Figure 5-6 and comparing it to Figure 5-1, already a noticeable difference can be noted starting from around AC 4, where the oscillations become more pronounced in the case of the descent with no lead feedback scheme. Due to the scaling of these plots, however, it is hard to pinpoint the exact differences, or even know how far aircraft deviate from their initial separation distance. Section 5.1.4 gives a closer look at these graphs and plots them with each other to compare results and give a better idea of the response in each case.

Figure 5-7: Separation Distance from Front (N-1) Aircraft (AC 1 Feedback)

Figure 5-7 shows the separation of aircraft from their N-1 aircraft. Comparing Figure 5-7 and Figure 5-2 really shows the differences between the scheme with lead feedback and the one without. Aircraft 2 is flown with only one lead, and thus is the only aircraft in this string with no lead feedback in its control scheme. As a result, its path is identical to the aircraft 2 in the case with no lead feedback.

The addition of lead feedback remarkably improves the behavior of following aircraft. By AC 3 the separation errors are minimum and aircraft maintain approximately 10 [nmi] separation at all times.

Figure 5-8 shows the velocity changes over time for all aircraft in the string for an interval of the 800 [s]. This is because after the 800 [s] mark, all aircraft maintain the same velocity as their lead aircraft. Furthermore, looking at this interval gives a clear view of the velocity response of all aircraft.

Figure 5-8: True Airspeed vs. Time (Lead Feedback)

Since it does not employ a lead feedback scheme, aircraft 2's velocity profile is the same as it was in the previous case. What is interesting to note is that all subsequent aircraft match almost identically the velocity profile of aircraft 2. There is no noticeable offset in the curves of AC 2 through 8. This mirroring of the velocity profile is what allows the following aircraft to maintain perfect separation from their N-1 aircraft. It also demonstrates that the lead feedback scheme is much better at allowing aircraft to match the velocity of their front aircraft.

As the descent profile is very similar to the one shown in Figure 5-5, it is omitted for this case, as well as for the case that follows. No aircraft deviate from their spatial path sufficiently enough for it to be noticeable.

5.1.3 Descent with N-2 Feedback

Figure 5-9 and Figure 5-10 show the distance of each aircraft from AC 1 and from their front (N-1) aircraft, respectively, for the case of a descent with N-2 feedback.

Figure 5-9: Separation Distance from Lead Aircraft (N-2 Feedback)

Figure 5-9 demonstrates a very similar response to what is seen in the first two schemes. Aircraft 1 through 3 are flown exactly the same as in the case with lead feedback, but aircraft 4 takes aircraft 2 to be its lead aircraft (and so on down the string).

The disturbances from the desired separation are somewhat more than what can be seen in the case with AC 1 feedback, but less than what is seen in the case with no feedback scheme. Section 5.1.4 compares the results of aircraft 8 for a closer and more accurate look at how the different schemes affect the separation distances.

Figure 5-10: Separation Distance from Front Aircraft (N-2 Feedback)

Figure 5-10 maps the separation distance from the N-1 aircraft of all the aircraft in the N-2 feedback scheme. As previously mentioned, AC 2 and AC 3 are the same in this case as in the case with AC 1 feedback. However, due to the errors in the flight path of AC 2, from AC 4 on the separation curves are somewhere in between the separation curves of AC 2 and AC 3.

Figure 5-11: True Airspeed vs. Time (N-2 Feedback)

Figure 5-11 maps the change in velocity over time for all aircraft in the string. Like in the case with lead feedback, it is shown for a shorter time interval (800 [s]). After this time interval all aircraft maintain a constant velocity, matching the lead aircraft, and as such this response is omitted from the results.

The results show a response with certain characteristics shared with the case with no feedback, and other characteristics shared with the one with lead feedback. As in the case with no lead feedback, the offset in velocity profiles is present in all aircraft. However, the shapes of the curves closely resemble each other, as in the case with lead feedback.

5.1.4 Comparing Results

To get a better idea of the differences in each control scheme, the same data from each control scheme are plotted on the same graph. The results are color-coded such that

blue always represents the case with N-2 feedback, red the case with no lead feedback, and green the case with AC 1 feedback.

Aircraft 8's results were chosen as the data set to analyze. This is because, as the last aircraft in the string, its resulting data had the potential to have the most noticeable differences between the testing schemes used. Furthermore, since both it and the aircraft in front of it, aircraft 7, only fly level and never descend, it illustrates the effect of velocity variations on level flight. Two parameters were looked at: one, the aircraft's ability to maintain separation from aircraft 1 (see Figure 5-12), and two, the aircraft's ability to maintain separation from aircraft 7 (see Figure 5-13).

Figure 5-12: Separation Distance from Aircraft 1 for Aircraft 8

Figure 5-12 shows the separation distance to the lead aircraft for aircraft 8 under all control schemes. It is clear that in terms of maintaining separation from the lead aircraft, the scheme that has AC 1 feedback does the best job at returning to the initial separation of 70 [nmi], as well as not deviating too much from that initial separation. The

N-2 feedback scheme has a profile very similar to that of the lead feedback scheme. This plot clearly shows that the case with no lead feedback end up furthest away from the lead aircraft. This is because with no lead feedback, the errors of the aircraft ahead of it become cumulative, and the controller cannot compensate for that with data from the lead aircraft.

Figure 5-13: Aircraft 8 Separation Distance from Aircraft 7

Figure 5-13 maps the separation distance between aircraft 7 and aircraft 8 for all test schemes. AC 8 with lead feedback maintains virtually perfect separation from AC 7 at all times. The case with N-2 feedback is following two aircraft that have some errors in their flight path, and as such cannot be as precise as the case with lead feedback. However, it maintains closer accuracy than the case with no lead feedback.

5.2 Disturbance in First and Fourth Aircraft

To see if having an N-2 feedback would be beneficial, the two cases with lead feedback were run again for aircraft 4 through 8. Aircraft 1 through 3 are flown as seen in section 5.1. An initial x-position offset of +1 [nmi] was included in aircraft 4's initial conditions to see how this disturbance would affect the following aircraft in either schemes.

Since the 1 [nmi] position offset only affected aircraft 4 through 8, the plots in this section only include these results. The offset occurred at the beginning of the run, and once the aircraft adjusted to this offset, the rest of their run was almost identical to the cases with no offset. Therefore, the results shown in this section only map the results for the first 400 seconds.

Furthermore, the only parameter that is analyzed is the ability of all aircraft to maintain separation from their front aircraft. This is due to several reasons. First, the separation of each aircraft from the lead aircraft is not of much concern, since the disturbance happens in the fourth aircraft. In addition, the results of separation from the lead aircraft can be deceiving because it hides the effect that this disturbance in the fourth vehicle has on the later aircraft in the string. The velocity plots were also included in the 400 second interval. Finally, the flight paths in 2D space were not plotted because they were pretty much the same as in section 5.1 .

5.2.1 Descent with Lead Feedback

Figure 5-14 shows the separation distance between aircraft and their front aircraft for aircraft 4 through 8, using the lead feedback scheme. Aircraft 4 begins with a position offset, and settles in about 300 seconds, with no overshoot.

Figure 5-14: Separation from Front Aircraft vs. Time (AC 1 Feedback and AC 4 Disturbance)

As expected, AC 5 begins its flight separated from its front aircraft by the inputted offset. Its curve looks almost like a mirror image of the curve for aircraft 4, which also makes sense. It also settles in a little more than 300 seconds.

AC 6 exhibits a similar curve to aircraft 5, but is remarkably less drastic. The overshoot it exhibits is about 0.1 [nmi] (about 600 [ft]), which is much greater than what is seen when there are no disturbances (as in Figure 5-7). A similar response is seen for AC 7 and 8, with even smaller overshoots.

Figure 5-15 shows the velocity profile for AC 4 through 8 in the 0 [s] to 400 [s] time interval. Furthermore, it shows the profile of AC 1, as a datum. The behavior exhibited in this figure match perfectly what is seen in Figure 5-14. AC 4 slows down considerably to reinstate its desired separation, and AC 5 through 8 start faster to catch up to AC 4. Furthermore, by the 200 second mark, AC 5 through 8 merge along the same curve for the remainder of their flights.

Figure 5-15: True Airspeed vs. Time (AC 1 Feedback and AC 4 Disturbance)

5.2.2 Descent with N-2 Feedback

Figure 5-16 shows the separation distance between aircraft and their N-2 aircraft for AC 4 through 8, using the N-2 feedback scheme and incorporating the 1 [nmi] offset in AC 4.

AC 4 in this scheme behaves similarly to its counterpart in the scheme with AC 1 feedback. It also settles after around 300 seconds, although not as smoothly. AC 5, on the other hand, settles at a much faster rate, after about 200 seconds. AC 6 exhibits an offset, but this time in the other direction, for about 0.1 [nmi] as well. AC 7 has an almost perfect behavior and AC 8 has a small offset, influenced by the feedback it is getting from AC 6.

Figure 5-16: Separation from Front with AC 4 Offset (N-2 Feedback)

In terms of the velocity profiles shown in Figure 5-17, AC 5 through 8 demonstrate a very similar response one from the other. There is a slight offset in the response and it takes 400 seconds for these curves to merge.

Figure 5-17: True Airspeed vs. Time (N-2 Feedback and AC 4 Disturbance)

6 Conclusions and Recommendations

The goal of this project is to design, implement and test the control of a string of aircraft, with each aircraft in the string following the aircraft in front of it, and also incorporate the effects of a lead feedback in the loop. This entails creating controls to maintain separation as well as controls to maintain course. To obtain information of aircraft further ahead, an ADS-B signal needs to be simulated. Furthermore, to test the system, a simulation of the dynamics is required.

The methodology chosen to achieve these goals was as follows. First, the flight path of aircraft was simulated using a function based programming language, Matlab. The flight path tested was a descent path, where the first aircraft in the string would decelerate as it descended. Three control schemes were tested: (1) a scheme with no feedback, (2) a scheme with feedback from the first aircraft in the string and (3) a scheme with feedback from the N-2 aircraft in the string. Furthermore, two test situations were considered: (1) one that had an initial velocity disturbance of +10 [kts] in the first aircraft and (2) one that had an initial position offset of +1 [nmi] in the fourth aircraft, coupled with the same initial velocity disturbance in the first aircraft.

The tests for the control scheme with no lead feedback showed that any small disturbances in the first aircraft became more and more pronounced in the flight paths of aircraft further and further away from this first aircraft. Strings of five aircraft or more would exhibit greater disturbances in their latter aircraft flights. By the seventh aircraft, the response obtained reflects a rather erratic velocity profile. Therefore, this response shows that this control scheme only works well with short strings of aircraft.

In contrast, the control schemes with the extra lead feedback showed that a flight path controlled with additional feedback from another aircraft in the string exhibits a remarkable improvement in its trajectory.

In the case where there was feedback from the first aircraft in the string and no disturbance in the fourth aircraft, all aircraft after the second one in the string (which did not have lead feedback) maintained almost perfect separation. The aircraft settled to

steady state velocity about two minutes after the lead aircraft completed its deceleration. However, it must be taken into consideration that the lead feedback received by the aircraft in the string is of a flight path with no errors due to the aircraft following control scheme. Furthermore, since there were no disturbances in any aircraft in the string other than in the first aircraft, the results seen in this case are ideal and not very representative of the norm.

In the case where there was feedback from the N-2 aircraft in the string and no disturbances in the fourth aircraft, all aircraft in the string maintained proper separation and there were no increases in disturbances further down the string similar to those seen in the scheme with no lead feedback. The results were not as smooth and had more deviations than those with the AC 1 feedback scheme. However, all the flights were smooth with no sharp changes in velocity or position. All aircraft settled to steady state at around four minutes after the first aircraft completed its deceleration, which is about twice as long as it took in the case with AC 1 feedback.

Comparing the results with no disturbances in the fourth aircraft showed that there was very little difference in the response between the schemes that had lead feedback. The separation distance from AC 7 of AC 8 results showed that at most the scheme with N-2 feedback deviated by 0.05 [nmi] (about 300 [ft]) whereas the scheme with AC 1 feedback had negligible deviations. In terms of the magnitude of separation these aircraft are maintaining, the 300 [ft] offset can also be considered negligible.

The response to an initial disturbance in AC 4 shows that both schemes that were tested are almost equally equipped to handle this disturbance. Both schemes tested managed to maintain safe separation with no erratic behavior. Furthermore, both exhibited similar response times.

In conclusion, both schemes that incorporated lead feedback had stable responses in all aircraft. Although the results of the scheme with AC 1 feedback maintained better separation, the test was not representative of a real system. The feedback scheme with the updating lead aircraft uses aircraft with controller errors for feedback and as such does not yield as smooth results, but is most likely a better option in a realistic setting.

Further tests which include small disturbances in more aircraft in different times along the flight path may prove useful in determining the best feedback scheme. Finally, it is recommended that future tests be run to further the study. These tests could include different flight trajectories, such as turns and climbing flight.

References

- [1] Cashion, P., Mackintosh, M-A., McGann, A., and Lozito, S., “A Study of Commercial Flight Crew Self-Separation,” *Digital Avionics Systems Conference, 1999, 16th DASC.*, AIAA/IEEE, Volume: 2, 26-30 Oct. 1997, pp. 6.3-18 – 6.3-25.
- [2] Chang, G.C., Livack, G.S., and McDaniel, J.I., “Emerging Cockpit Technologies for Free Flight,” *Digital Avionics Systems Conference, 1996, 15th DASC.*, AIAA/IEEE, 27-31 Oct. 1996, pp. 63-71.
- [3] Grimaud, I., Hoffman, E., and Zeghal, K., “Limited Delegation of Separation Assurance to the Flight Crew,” *Preparing for the Global Challenges*, Aeronautics Days 2001, Hamburg, Germany, 29-31 Jan. 2001.
- [4] Guenther, R.B., Kharab, A., *An Introduction to Numerical Methods: A MATLAB Approach*, Chapman and Hall/CRC, New York, 2002.
- [5] Hoffman, E., and Zeghal, K., “Delegation of Separation Assurance to Aircraft: Towards a Framework for Analysing the Different Concepts and Underlying Principles,” *ICAS 2000 Congress*, Harrogate, UK, Aug.-Sept. 2000, pp. 192.1-192.10.
- [6] Hoffman, E., Ivanescu, D., and Zeghal, K., “Impact of ADS-B Link Characteristics on the Performance of In-Trail Following Aircraft,” *AIAA 2002-4933, AIAA Guidance, Navigation, and Control Conference and Exhibit*, Monterey, California, 5-8 Aug. 2002.
- [7] Inman, D.J., *Engineering Vibrations*, 2nd Ed., Prentice Hall, New Jersey, 2000.
- [8] Kuchar, J.K., and Schild, R., “Operational Efficiency of Maneuver Coordination Rules for an Airborne Separation Assurance System,” *3rd USA/Europe Air Traffic Management R&D Seminar*, Napoli, Italy, 13-16 June 2000.
- [9] Lewis, F.L., Stevens, B.L., *Aircraft Control and Simulation*, 2nd Ed., John Wiley & Sons, Inc., New Jersey, 2003.
- [10] Lindfield, G., Penny, J., *Numerical Methods using MATLAB*, 2nd Ed., Prentice Hall, New Jersey, 2000.
- [11] Nolan, Michael S. *Fundamentals of Air Traffic Control*. 3rd Ed. Brooks/Cole, Pacific Grove, CA, 1999.

- [12] Pritchett, A.R., and Yankosky, L.J., "Pilot-Performed In-Trail Spacing and Merging: An Experimental Study," *Journal of Guidance, Control, and Dynamics*, Vol. 26, No. 1, Jan.- Feb. 2003, pp. 143-149.
- [13] *RTCA: Minimum Aviation System Performance Standards for Automatic Dependent Surveillance Broadcast (ADS-B)*. DO # 242-A. June 25, 2002
- [14] Shevell, R.S., *Fundamentals of Flight*, Prentice Hall, New Jersey, 1983.
- [15] Slater, G.L., "Dynamics of Self-Spacing in a Stream of In-Trail Aircraft," AIAA 2002-4927, *AIAA Guidance, Navigation, and Control Conference and Exhibit*, Monterey, California, 5-8 Aug. 2002.

Appendices

APPENDICES

Appendix A:	Drivers.....	A-1
Appendix B:	State Object.....	B-1
Appendix C:	Aerodynamic Models.....	C-1
Appendix D:	Flight Path Functions	D-1
Appendix E:	Controllers and Integrators	E-1

Appendix A: Drivers

This section of the appendix comprises of an example of the MATLAB code for the drivers used to run all the test simulations.

The example chosen was of the driver titled *FPS_testDriverFastnoFBI*. It runs 8 aircraft in a string with no lead feedback. The lead aircraft starts with an initial disturbance of +10 knots.

All other simulations are small modifications of this test driver. For the case with AC 1 feedback, the function *FPS_followpath* is replaced with the function *FPS_followpathLFB*. For the case with N-2 feedback, it is the same as with AC 1 feedback but the follower number is always 2 for all following aircraft after aircraft 2. Also, instead of inputting the lead aircraft filename into the follower flight path functions, the N-2 aircraft filename is inserted.

For the runs that had an initial disturbance in the fourth aircraft, the third follower's initial conditions included an initial velocity of 10 nautical miles greater than in previous runs.


```

clc;
clear;

referenceFilename = './flight path simulator/Data Files/reference1.txt';
leadFilename = './flight path simulator/Data Files/leadFast2.txt';
followerReferenceFilename = './flight path simulator/Data Files/referenceSpace1.txt';
foll_1_Filename = './flight path simulator/Data Files/AC2Fast1t2.txt';
foll_2_Filename = './flight path simulator/Data Files/AC3FastnoFB2.txt';
foll_3_Filename = './flight path simulator/Data Files/AC4FastnoFB2.txt';
foll_4_Filename = './flight path simulator/Data Files/AC5FastnoFB2.txt';
foll_5_Filename = './flight path simulator/Data Files/AC6FastnoFB2.txt';
foll_6_FilenameN = './flight path simulator/Data Files/AC7FastnoFB2.txt';
foll_7_Filename = './flight path simulator/Data Files/AC8FastnoFB2.txt';
leaddxFileName = './flight path simulator/Data Files/lead_dxFast2.txt';
dxFileName1 = './flight path simulator/Data Files/AC2dxFast1t.txt';
dxFileName2 = './flight path simulator/Data Files/AC3dxFastnoFB1.txt';
dxFileName3 = './flight path simulator/Data Files/AC4dxFastnoFB1.txt';
dxFileName4 = './flight path simulator/Data Files/AC5dxFastnoFB1.txt';
dxFileName5 = './flight path simulator/Data Files/AC6dxFastnoFB1.txt';
dxFileName6 = './flight path simulator/Data Files/AC7dxFastnoFB1.txt';
dxFileName7 = './flight path simulator/Data Files/AC8dxFastnoFB1.txt';

planeType = 'B757';      stepTime = 0.2; %seconds
UpdateInterval = 6/stepTime; %this times step time to give you it in seconds
v0 = FPS_kts2fps(352.9448);      h0 = 15000;      gamma0 = -3*(pi/180);
psi0 = 0;      [W,L,D,Th]=FPS_B757(v0,h0,gamma0);      T = D+W*sin(gamma0);
nmi = 6076.1155; %ft
sd = 10*nmi;
initialConditions = [0   v0   h0   7*sd   0 gamma0 0 psi0 D L T W 1;
                    NaN NaN 10000 NaN NaN 0 NaN NaN NaN NaN NaN NaN 7;
                    NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN 8;
                    NaN NaN 5000 NaN NaN 0 NaN NaN NaN NaN NaN NaN 7;
                    NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN 1];

stopConditions = [ 3 10000;
                  4 FPS_kts2fps(290.9485);
                  3 5000;
                  4 FPS_kts2fps(193.9198);
                  1 1000];

modes = [1; 7; 8; 7; 1];
inputsToAeroModels = [FPS_kts2fps(280) 0; 0 0; -100/6 0; 0 0; FPS_kts2fps(180) 0];
M = FPS_FlightPath(referenceFilename, planeType, stepTime, initialConditions, stopCon
ditions, modes, inputsToAeroModels);
M=csvread(referenceFilename);

% Now run the leader:

v_fast = v0 + FPS_kts2fps(10);
leadInitialConditions = [0 v_fast h0 7*sd 0 gamma0 0 psi0 D L T W 1];
N = FPS_leadFlightPath(leadFilename, planeType, stepTime, leadInitialConditions, refe
renceFilename,leaddxFileName);
N =csvread(leadFilename);

% Now run the follower reference path:
initialConditions_fr1 = [0 v0 h0 0 0 0 0 psi0 D L T W 1;
                        NaN   NaN   NaN   NaN NaN gamma0 0 psi0 NaN NaN NaN NaN 1;
                        NaN NaN 10000 NaN NaN 0 NaN NaN NaN NaN NaN NaN 7;
                        NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN 8;

```

```

NaN NaN 5000 NaN NaN 0 NaN NaN NaN NaN NaN NaN 7;
NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN 1];
stopConditions_fr1 = [6 7*sd;
3 10000; %once it gets to an altitude of 10000 ft
4 FPS_kts2fps(290.9485); % once it slows down to 290.9 kts
3 5000; %once it gets to an altitude of 5000 ft
4 FPS_kts2fps(193.9198); % once it slows down to 161.6 kts
1 1000]; %after 3000 seconds
modes_fr1 = [1; 1; 7; 8; 7; 1];
inputsToAeroModels_fr1 = [FPS_kts2fps(280) 0; FPS_kts2fps(280) 0;
0 0; -100/6 0; 0 0; FPS_kts2fps(180) 0];
M1 = FPS_FlightPath(followerReferenceFilename, planeType, stepTime, initialConditions
_fr1, stopConditions_fr1, modes_fr1, inputsToAeroModels_fr1);
M1= csvread(followerReferenceFilename);

% now run first follower:
FollowerNumber = 1;
foll_1_initialConditions = [0 v0 h0 6*sd 0 0 0 psi0 D L T W 1];
phase = 0;
P = FPS_followPath(foll_1_Filename, planeType, stepTime, foll_1_initialConditions, f
ollowerReferenceFilename, leadFilename, UpdateInterval, sd, FollowerNumber, dxFileName1,
phase);
P = csvread(foll_1_Filename);

% now second follower:
FollowerNumber = 2;
foll_2_initialConditions = [0 v0 h0 5*sd 0 0 0 psi0 D L T W 1 FPS_kts2fps(280) 0];
Q = FPS_followPath(foll_2_Filename, planeType, stepTime, foll_2_initialConditions, f
ollowerReferenceFilename, foll_1_Filename, UpdateInterval, sd, FollowerNumber, dxFileName
2, phase);
Q = csvread(foll_2_Filename);

% now third follower:
FollowerNumber = 3;
foll_3_initialConditions = [0 v0 h0 4*sd 0 0 0 psi0 D L T W 1 FPS_kts2fps(280) 0];
R = FPS_followPath(foll_3_Filename, planeType, stepTime, foll_3_initialConditions, f
ollowerReferenceFilename, foll_2_Filename, UpdateInterval, sd, FollowerNumber, dxFileName
3, phase);
R = csvread(foll_3_Filename);

% now fourth follower:
FollowerNumber = 4;
foll_4_initialConditions = [0 v0 h0 3*sd 0 0 0 psi0 D L T W 1 FPS_kts2fps(280) 0];
S = FPS_followPath(foll_4_Filename, planeType, stepTime, foll_4_initialConditions, f
ollowerReferenceFilename, foll_3_Filename, UpdateInterval, sd, FollowerNumber, dxFileName
4, phase);
S = csvread(foll_4_Filename);

% now fifth follower:
FollowerNumber = 5;
foll_5_initialConditions = [0 v0 h0 2*sd 0 0 0 psi0 D L T W 1 FPS_kts2fps(280) 0];
U = FPS_followPath(foll_5_Filename, planeType, stepTime, foll_5_initialConditions, f
ollowerReferenceFilename, foll_4_Filename, UpdateInterval, sd, FollowerNumber, dxFileName
5, phase);
U = csvread(foll_5_Filename);

% now sixth follower:

```

May 14, 2005

4:07:53 PM

```

    FollowerNumber = 6;
    foll_6_initialConditions = [0 v0 h0 1*sd 0 0 0 psi0 D L T W 1 FPS_kts2fps(280) 0];
    V = FPS_followPath(foll_6_Filename, planeType, stepTime, foll_6_initialConditions, f
ollowerReferenceFilename, foll_5_Filename, UpdateInterval, sd, FollowerNumber, dxFileName
6, phase);
    V = csvread(foll_6_Filename);

% now seventh follower:
    FollowerNumber = 7;
    foll_7_initialConditions = [0 v0 h0 0*sd 0 0 0 psi0 D L T W 1 FPS_kts2fps(280) 0];
    X = FPS_followPath(foll_7_Filename, planeType, stepTime, foll_7_initialConditions, f
ollowerReferenceFilename, foll_6_Filename, UpdateInterval, sd, FollowerNumber, dxFileName
7, phase);
    X = csvread(foll_7_Filename);

%Now plot everything!

deg = 180/pi; % rad

%Define the variables:
    t = N(:,1);

    v_lead = FPS_fps2kts(N(:,2)); %TAS
    v_foll_1 = FPS_fps2kts(P(:,2));    v_foll_2 = FPS_fps2kts(Q(:,2));
    v_foll_3 = FPS_fps2kts(R(:,2));    v_foll_4 = FPS_fps2kts(S(:,2));
    v_foll_5 = FPS_fps2kts(U(:,2));    v_foll_6 = FPS_fps2kts(V(:,2));
    v_foll_7 = FPS_fps2kts(X(:,2));

    x_lead = N(:,4)/nmi; %x-position
    x_foll_1 = P(:,4)/nmi;    x_foll_2 = Q(:,4)/nmi;
    x_foll_3 = R(:,4)/nmi;    x_foll_4 = S(:,4)/nmi;
    x_foll_5 = U(:,4)/nmi;    x_foll_6 = V(:,4)/nmi;
    x_foll_7 = X(:,4)/nmi;

    h_lead = N(:,3); %altitude
    h_foll_1 = P(:,3);    h_foll_2 = Q(:,3);
    h_foll_3 = R(:,3);    h_foll_4 = S(:,3);
    h_foll_5 = U(:,3);    h_foll_6 = V(:,3);
    h_foll_7 = X(:,3);

    s_lead = 0;
    s_foll_1 = sqrt((P(:,4)-N(:,4)).^2 + (P(:,5)-N(:,5)).^2)/nmi; %distance to lead
    s_foll_2 = sqrt((Q(:,4)-N(:,4)).^2 + (Q(:,5)-N(:,5)).^2)/nmi;
    s_foll_3 = sqrt((R(:,4)-N(:,4)).^2 + (R(:,5)-N(:,5)).^2)/nmi;
    s_foll_4 = sqrt((S(:,4)-N(:,4)).^2 + (S(:,5)-N(:,5)).^2)/nmi;
    s_foll_5 = sqrt((U(:,4)-N(:,4)).^2 + (U(:,5)-N(:,5)).^2)/nmi;
    s_foll_6 = sqrt((V(:,4)-N(:,4)).^2 + (V(:,5)-N(:,5)).^2)/nmi;
    s_foll_7 = sqrt((X(:,4)-N(:,4)).^2 + (X(:,5)-N(:,5)).^2)/nmi;

    d_foll_1 = sqrt((P(:,4)-N(:,4)).^2 + (P(:,5)-N(:,5)).^2)/nmi; %distance to N-1
    d_foll_2 = sqrt((Q(:,4)-P(:,4)).^2 + (Q(:,5)-P(:,5)).^2)/nmi;
    d_foll_3 = sqrt((R(:,4)-Q(:,4)).^2 + (R(:,5)-Q(:,5)).^2)/nmi;
    d_foll_4 = sqrt((S(:,4)-R(:,4)).^2 + (S(:,5)-R(:,5)).^2)/nmi;
    d_foll_5 = sqrt((U(:,4)-S(:,4)).^2 + (U(:,5)-S(:,5)).^2)/nmi;
    d_foll_6 = sqrt((V(:,4)-U(:,4)).^2 + (V(:,5)-U(:,5)).^2)/nmi;
    d_foll_7 = sqrt((X(:,4)-V(:,4)).^2 + (X(:,5)-V(:,5)).^2)/nmi;

```

Appendix B: State Object

For the state of each aircraft to be easily communicated between functions, an object to contain the state variables was created. This appendix holds all the MATLAB code used in creating this state object. These include the functions that are used to call to a state object, change that state object, and write that object to a file.

```
%CONSTRUCTOR for FPS_state class
%
%*****%
%HOW TO USE%
%*****%
%
%   To create an initialized state:
%       myState1 = FPS_state([1 2 3 4 5 6 7 8 9 10 11, 12,13])
%
%   To create an empty state:
%       myState2 = FPS_state
%
%   To get more help on individual functions
%       help @FPS_state/set
%       help @FPS_state/get

function object = FPS_state(input)
    %first initialize our data
    object.data = zeros(13,1); %t,v,h,x,y,gamma,phi,psi,D,L,T,W,mode

    %if there was input, copy it into our data
    if nargin == 1
        object.data(1) = input(1); % t
        object.data(2) = input(2); % v
        object.data(3) = input(3); % h
        object.data(4) = input(4); % x
        object.data(5) = input(5); % y
        object.data(6) = input(6); % gamma
        object.data(7) = input(7); % phi
        object.data(8) = input(8); % psi
        object.data(9) = input(9); % D (for drag)
        object.data(10) = input(10); % L (for lift)
        object.data(11) = input(11); % t (for thrust)
        object.data(12) = input(12); % W (for weight)
        object.data(13) = input(13); % mode (for mode)

    end

    %register the object as part of the class
    object = class(object,'FPS_state');
return
```

```
%GET function for FPS_state class
%
%%%%%%%%%%%%%
%HOW TO USE%
%%%%%%%%%%%%%
%
%   state = FPS_state
%   get(state,'v')
%
%
function returnValue = get(object, attributeName)

%We are supplying the state variables to a requestor

switch attributeName
case 't'
    returnValue = object.data(1);
case 'v'
    returnValue = object.data(2);
case 'h'
    returnValue = object.data(3);
case 'x'
    returnValue = object.data(4);
case 'y'
    returnValue = object.data(5);
case 'gamma'
    returnValue = object.data(6);
case 'phi'
    returnValue = object.data(7);
case 'psi'
    returnValue = object.data(8);
case 'D'
    returnValue = object.data(9);
case 'L'
    returnValue = object.data(10);
case 'T'
    returnValue = object.data(11);
case 'W'
    returnValue = object.data(12);
case 'mode'
    returnValue = object.data(13);
otherwise
    error([attributeName, ' is not a valid attribute of the FPS_state class'])
end
return
```

```
%SET function for FPS_state class
%
%%%%%%%%%%%%%
%HOW TO USE%
%%%%%%%%%%%%%
%
%   state = FPS_state
%   state = set(state,'v',5)
%
%
function object = set(object, attributeName, newAttributeValue)

%We are inputing state variables based on a source

switch attributeName
    case 't'
        object.data(1) = newAttributeValue;
    case 'v'
        object.data(2) = newAttributeValue;
    case 'h'
        object.data(3) = newAttributeValue;
    case 'x'
        object.data(4) = newAttributeValue;
    case 'y'
        object.data(5) = newAttributeValue;
    case 'gamma'
        object.data(6) = newAttributeValue;
    case 'phi'
        object.data(7) = newAttributeValue;
    case 'psi'
        object.data(8) = newAttributeValue;
    case 'D'
        object.data(9) = newAttributeValue;
    case 'L'
        object.data(10) = newAttributeValue;
    case 'T'
        object.data(11) = newAttributeValue;
    case 'W'
        object.data(12) = newAttributeValue;
    case 'mode'
        object.data(13) = newAttributeValue;
    otherwise
        error([attributeName, ' is not a valid attribute of the FPS_state class'])
end
return
```

```
function initializeFile(state, fileName)
%
% initializeFile(state, fileName)
%
% This function is used to start a file (of name fileName (.txt)) with the
% initial conditions
%
% Inputs are the initial conditions and the fileName (.txt) that you wish to write
% to.

writeMode = 'w';

fid = fopen(fileName, writeMode);

if fid == -1
    ERROR('@FPS_state::initializeFile: Failed to initialize file');
end

fprintf(fid, '%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f\n', state.data(1), state.data(2), ✓
state.data(3), state.data(4), state.data(5), state.data(6), state.data(7), state.data(8) ✓
,state.data(9),state.data(10),state.data(11),state.data(12),state.data(13));
fclose(fid);

return;
```

```
function writeToFile(state, fileName)
%
% writeToFile(state, fileName)
%
% This function is used to write to a specified fileName the current values
% in the state.
%
% Inputs are the state variables (in the form of the state name and the fileName
% (.txt) that you wish to write to.

    writeMode = 'a';

    fid = fopen(fileName, writeMode);

    if fid == -1
        ERROR('@FPS_state::writeToFile: Failed to open file');
    end

    %we must convert true airspeed from ft/s to knots
    fprintf(fid, '%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f\n', state.data(1), state.data(2), ↵
state.data(3), state.data(4), state.data(5), state.data(6), state.data(7), state.data(8), ↵
state.data(9), state.data(10), state.data(11), state.data(12), state.data(13));
    fclose(fid);

return;
```

```
function string = getAttribute(state, index)
    switch index
        case 1
            string = 't';
        case 2
            string = 'v';
        case 3
            string = 'h';
        case 4
            string = 'x';
        case 5
            string = 'y';
        case 6
            string = 'gamma';
        case 7
            string = 'phi';
        case 8
            string = 'psi';
        case 9
            string = 'D';
        case 10
            string = 'L';
        case 11
            string = 'T';
        case 12
            string = 'W';
        case 13
            string = 'mode';
        otherwise
            error('FPS_GetAttribute: unrecognised index %i', index)
    end
return
```

Appendix C: Aerodynamic Models

This segment contains two aerodynamic models as examples of the entire collection. The first, *FPS_aero_model*, is the aerodynamic model without any constraints applied to it. These are set up in the form of ordinary differential equations such that the step function can integrate them. The second, *FPS_aero_model_CAS_RC* is an example of an aerodynamic model with constraints. This particular function sets two constraints: it sets a constant CAS and a constant rate of climb.

This segment also includes the aircraft thrust, lift, weight and drag characteristics called to by the aerodynamic models. This function, *FPS_B757*, simulates a representative transport aircraft model.

```
function ydot = FPS_aero_model(t, y, aircraft, inputToAeroModel)
%function ydot = FPS_aero_model(t, y, aircraft, inputToAeroModel)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%aero model for reference followers%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   INPUT
%
%       t = time (seconds) -- provided by integrator function such as ode23
%       y = [V, h, x, y, gamma, phi, psi] (ft/s, ft, ft, ft, rad, rad, rad) -- provided by integrator function such as ode23
%       aircraft = 'FPS_B757' (for example)
%       inputToAeroModel is ignored (set it equal to [])
%
%   OUTPUT
%       ydot = [Vdot; hdot; xdot; ydot; gammadot; phidot; psidot]

ydot = zeros(7,1); %creates a column vector
p = inputToAeroModel(1);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%CONSTANTS%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
[w,L,D,T] = feval(aircraft, y(1), y(2), y(5));

%   L = lift in lbs
%   D = drag in lbs
%   T = thrust in lbs
g = 32.2; %acceleration of gravity in ft/s^2
Vwind = 0; %speed of wind
Xiwind = 0; %angle of wind in radians

%ydot(1) = vdot      y(1) = v
%ydot(2) = hdot      y(2) = h
%ydot(3) = xdot      y(3) = x
%ydot(4) = ydot      y(4) = y
%ydot(5) = gammadot  y(5) = gamma
%ydot(6) = phidot    y(6) = phi
%ydot(7) = psidot    y(7) = psi

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%EQUATIONS%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
ydot(1) = (T - D)*g/w - g*sin(y(5));
ydot(2) = y(1)*sin(y(5));
ydot(3) = y(1)*cos(y(5))*cos(y(7)) - Vwind*cos(Xiwind);
ydot(4) = y(1)*cos(y(5))*sin(y(7)) - Vwind*sin(Xiwind);
ydot(5) = L*cos(y(6))/(y(1)*w/g) - (g*cos(y(5)))/y(1);
ydot(6) = p;
ydot(7) = g*tan(y(6))/y(1);

return;
```

```
function ydot = FPS_aero_model_CAS_RC(t, y, aircraft, inputToAeroModel)
%function ydot = FPS_aero_model_CAS_RC(t, y, aircraft, inputToAeroModel)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%aero model for constant CAS and Rate of Climb%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%Calculates aircraft equations of motion
%
% INPUT
%     t = time (seconds) -- provided by integrator function such as ode23
%     y = [V, h, x, y, gamma, phi, psi] (ft/s, ft, ft, ft, rad, rad, rad) -- provided
by
%         integrator function such as ode23
%     aircraft = 'FPS_B757' (for example)
%     inputToAeroModel = [CAS, R/C] where CAS is the calibrated air speed
%                       in ft/s and R/C is the rate of climb in ft/s
%
% OUTPUT
%     ydot = [Vdot; hdot; xdot; ydot; gammadot; phidot; psidot]

CAS = inputToAeroModel(1);
R_C = inputToAeroModel(2);

ydot = zeros(7,1); %creates a column vector

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%CONSTANTS%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
[w,L,D,T] = feval(aircraft, y(1), y(2), y(5));

%     L = lift in lbs
%     D = drag in lbs
%     T = thrust in lbs
g = 32.2; %acceleration of gravity in ft/s^2
Vwind = 0; %speed of wind
Xiwind = 0; %angle of wind in radians

%ydot(1) = Vdot      y(1) = v
%ydot(2) = hdot      y(2) = h
%ydot(3) = xdot      y(3) = x
%ydot(4) = ydot      y(4) = y
%ydot(5) = gammadot  y(5) = gamma
%ydot(6) = phidot    y(6) = phi
%ydot(7) = psidot    y(7) = psi

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%EQUATIONS%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

y(1) = FPS_constCAS(CAS,y(2));
y(5) = FPS_constRC(y(1), R_C);

ydot(1) = 0; %ignored and overridden
ydot(2) = R_C; %y(1)*sin(y(5));
ydot(3) = y(1)*cos(y(5))*cos(y(7)) - Vwind*cos(Xiwind);
```

```
ydot(4) = y(1)*cos(y(5))*sin(y(7)) - Vwind*sin(Xiwind);  
ydot(5) = 0; %ignored and overridden  
ydot(6) = 0;  
ydot(7) = 0;  
  
return;
```

```
function [W,L,D,T]=FPS_B757(Vt,h,gamma)
%
% [W,L,D,T] = FPS_B757(Vt,h,gamma)
%
% Calculates lift, drag and thrust for the Boeing 757
%
% Inputs are Speed (Vt) in ft/s, altitude (h) in ft, and Flight Path Angle
% (gamma) in radians.
%
% Outputs are weight (W), lift (L), drag (D) and thrust (T) all in lbs.
%
% See b757_dat.m file for original code.
%
% Edited by Elena

% Assume constant weight, neglect weight loss due to fuel consumption
W = 200000; %weight in lbs

%Aircraft data for the B757

S=1994;          % Gross Wing Area
nbreng=2;        %Number of Engines

a=FPS_asonic(h); %speed of sound
Patm = FPS_pressure(h)*2.0885; %atmospheric pressure in psf
Tatm = FPS_ttempf(h); %temp in deg Rankine
R=1.7165e03; % ft^2/(sec^2 deg Rankine)
rho=Patm/(R*Tatm); % density calculation
M=Vt/a; %aircraft's Mach number
qbar=.5*rho*Vt^2; %dynamic pressure
CL=W/(qbar*S); %coefficient of lift

T = nbreng*FPS_GetThrustPerEngine(M,h); %thrust in lbs
L=W*cos(gamma); %lift in lbs
Cdp = 0.0153; %Drag polar at low Mach
K = 0.0483;
CD = Cdp + K*CL^2; %Parabolic Drag Coefficient
D=qbar*S*CD; %Drag in lbs

return;
```

Appendix D: Flight Path Functions

This section contains the flight path functions for all the aircraft. *FPS_FlightPath* is the function that creates all reference flight paths and space reference flight paths. It calls to the segment function and feeds it the initial conditions and stopping conditions of each segment. *FPS_leadFlightPath* is the function that simulates the lead aircraft's flight. *FPS_followPath* is the function that simulates the flights of all following aircraft with no lead feedback. *FPS_followPathLFB*, although not included, is what is used for lead feedback. The only difference between it and the function with no lead feedback is that it calls to *FPS_PDIntegratorLFB* instead of *FPS_PDIntegrator* and the acceleration calculated when no ADS-B updates are received include lead feedback dynamics.


```
function M = FPS_FlightPath(outputFilename, planeType, stepTime, initialConditions, stopC ✓
onditions, modes, inputsToAeroModels)
%
% M = FPS_FlightPath(outputFilename, planeType, stepTime, initialConditions, stopConditio ✓
ns, modes, inputsToAeroModels)
%
% This is the function used to create a reference flight path.
%
% Inputs:
%     outputFilename: The file you want all the data (the M matrix) to be
%                     uploaded into.
%     planeType: Specify which aircraft model you want to use. e.g. B757
%     stepTime: What are the integration intervals (time steps)
%     initialConditions: This is in matrix form, indicating the initial
%                       conditions of the state for each segment run
%     stopConditions: This is in matrix form, indicating the ending
%                     conditions of each segment run (see FPS_Segment)
%     modes: This specifies which constrained aerodynamical to use.
%            These models hold specific values constant (see
%            FPS_Step for details)
%     inputsToAeroModel: These values correspond to the values to be held
%                        constant in the chosen mode. (see FPS_Step)
%
% Output: M matrix, State values [t v h x y gamma phi psi D L T W mode] for
%         all data points collected.

%convert plane type inputted by user to plane function usable by code. If
%user chose incorrect plane type, output error
switch planeType
    case 'B757'
        planeFunction = @FPS_B757;
    otherwise
        error('FPS_FlightPath: the aircraft %s is not in the database', planeType)
end

%set initial conditions, and write them to output file
state = FPS_state(initialConditions(1,:)); %WARNING: need to check that no values are ✓
NaN
initializeFile(state, outputFilename)

%each row corresponds to a single segment data input
numberOfRows = size(initialConditions,1);
numberOfColumns = size(initialConditions,2);

for i = 1:numberOfRows
    fprintf('Starting Segment #%i\n', i)

    %set initial conditions if necessary
    interimConditions = initialConditions(i,:);

    %Must figure out initial conditions for i + 1 segments. This is for NaN values
    %which depend on the previous segment to determine their values.

    if i > 1
        %for each initial condition entry
```

```
        for j = 1:numberOfColumns
            %if the initial condition entry is a real number
            if isnan(interimConditions(j)) == 0
                %then we set a new value for the state
                state = set(state, getAttribute(state, j), interimConditions(j));
            end
        end
    end

    %Run the segment
    state = FPS_Segment(outputFilename, state, stepTime, stopConditions(i,:), planeFu ↵
nction, modes(i), inputsToAeroModels(i,:));

    fprintf('done!\n')
    state
end

%this is how to read the file created, so that following planes can know
%what the leader is doing
M = csvread(outputFilename);

return
```

```
function M = FPS_leadFlightPath(outputFilename, planeType, stepTime, initialConditions, r
eferenceFileName, dxFileName)
%
% M = FPS_leadFlightPath(outputFilename, planeType, stepTime, initialConditions, referenc
eFileName, dxFileName)
%
% This is the function used to create the leader's flight path by following
% its corresponding reference flight path.
%
% Inputs:
%     outputFilename: The file you want all the data (the M matrix) into which
%                     it needs to be uploaded.
%     planeType: Specify which aircraft model you want to use. e.g. B757
%     stepTime: What are the integration intervals (time steps)
%     initialConditions: This is in matrix form, indicating the initial
%                       conditions of the state for each segment run
%     referenceFileName: This is the name of the file of the reference path you
%                       want to follow
%     dxFileName: This is the name of the file where you want to store
%                 the control values (created in the LQR controller during
%                 linearization)
%
% Output: M matrix, State values [t v h x y gamma phi psi D L T W mode] for
%         all data points collected.

%convert plane type inputted by user to plane function usable by code. If
%user chose incorrect plane type, output error
switch planeType
    case 'B757'
        aircraft = @FPS_B757;
    otherwise
        error('FPS_leadFlightPath: the aircraft %s is not in the database', planeType)
end

% How long to iterate? As long as reference path!
N = csvread(referenceFileName);
length = size(N,1)-1;

%set initial conditions, and write them to output file
leadState = FPS_state(initialConditions); %WARNING: need to check that no values are
NaN
initializeFile(leadState, outputFilename);

%initialize the dx file for the control values and write them to their
%output file
dxState = FPS_state(zeros(1,15));
initializeFile(dxState,dxFileName);

%This is true for all paths
mode = 10;
inputToAeroModel = [0 0];

%Now let's state that we're starting this segment in the command window
fprintf('Starting lead aircraft flight:\n')

%Run the step
```

```
for stepNumber = 1: length,

    %where are we now?
    vL = get(leadState, 'v');
    hL = get(leadState, 'h');
    xL = get(leadState, 'x');
    yL = get(leadState, 'y');
    gaL = get(leadState, 'gamma');
    phiL = get(leadState, 'phi');
    psiL = get(leadState, 'psi');
    t = get(leadState, 't');

    %second, we need to find and add the dx value:
    newStepNumber = stepNumber + 1;

    %where do we want to be?
    refState = FPS_state(N(newStepNumber, :));

    %Determine control values:
    dxState = FPS_dxIntegrator(stepTime, leadState, refState);

    %write them to file:
    writeToFile(dxState, dxFileName);

    dv = get(dxState, 'v');
    dh = get(dxState, 'h');
    dx = get(dxState, 'x');
    dy = get(dxState, 'y');
    dga = get(dxState, 'gamma');
    dphi = get(dxState, 'phi');
    dpsi = get(dxState, 'psi');

    %add the dxState values to the leadState values:

    leadState = set(leadState, 'v', (vL + dv));
    leadState = set(leadState, 'h', (hL + dh));
    leadState = set(leadState, 'x', (xL + dx));
    leadState = set(leadState, 'y', (yL + dy));
    leadState = set(leadState, 'gamma', (gaL + dga));
    leadState = set(leadState, 'phi', (phiL + dphi));
    leadState = set(leadState, 'psi', (psiL + dpsi));
    leadState = set(leadState, 't', t+1);

    %Now we need to find the D,L and T values and set them:

    y(1) = get(leadState, 'v');
    y(2) = get(leadState, 'h');
    y(5) = get(leadState, 'gamma');
    [W,L,D,T]=FPS_B757(y(1),y(2),y(5));

    T0 = get(refState, 'T');
    dT = get(dxState, 'T');

    T = T0 + dT;

    leadState = set(leadState, 'L', L);
```

```
    leadState = set(leadState,'D',D);
    leadState = set(leadState,'T',T);

    mode = get(refState,'mode');

    leadState = set(leadState,'mode',mode);

    %Finally, need to write this data to a file:
    writeToFile(leadState, outputFilename)

    %Print a little something to let the user know that Matlab is working
    fprintf('..');

    if (mod(stepNumber,50) == 0) %change lines every 50 dots
        fprintf('\n');
    end

    %now go again!
    stepNumber = stepNumber +1;

end

%Now let me know when it's done!
fprintf('lead aircraft flight done!\n')

%this is how to read the file created
M = csvread(outputFilename);

return
```

```

function M = FPS_followPath(outputFileName, planeType, stepTime, initialConditions, referenceFileName, frontFileName, UpdateInterval, sd, FollowerNumber, dxFileName, phase)
%
% function M = FPS_followPath(outputFilename, planeType, stepTime, initialConditions, referenceFileName, leadFileName, UpdateInterval)
%
% This function does not use lead feedback
%
% INPUT:
%     * outputFileName: The name of the file you want this data to be stored in
%     * planeType: What plane data do you want to use? See first switch to see what
format this should be in
%     * initialConditions: initial conditions vector in the form of the State object.
%     * referenceFileName: The desired path you want to stay on
%     * frontFileName: The state data file of the aircraft directly in front of this
aircraft
%     * UpdateInterval: The time interval that we get updates from the lead aircraft
%     * sd = separation distance required, in feet.
%     * FollowerNumber: where you are in the string, minus 1 (follower
%     #1 is first aircraft after lead)
%     * dxFileName: The name of the file you want the control values to
%     be stored in
%     * phase: update interval offset
%
% OUTPUT:
%     * M: the matrix that gets put into the outputFileName.

% Right now just have B757 but if get more data from other a/c, can add
% into this switch.
switch planeType
    case 'B757'
        aircraft = @FPS_B757;
    otherwise
        error('FPS_leadFlightPath: the aircraft %s is not in the database', planeType)
end

dxTime = stepTime;
p = phase;

% How long to iterate? As long as front path!
P = csvread(referenceFileName);
Q = csvread(frontFileName);
length = size(Q,1)-1;

%set initial conditions, and write them to output file
dxState = FPS_state(zeros(1,15));
follState = FPS_state(initialConditions); %WARNING: need to check that no values are
NaN
initializeFile(follState, outputFileName);
initializeFile(dxState, dxFileName);

%let me know when it's starting:
i = FollowerNumber;
fprintf('Starting follower #i:\n', i)

mode = 10;
inputToAeroModel = [0,0];

```

```
refState = FPS_state(zeros(15));

% have assumed front values to maintain as long as we are not
% receiving data from the lead aircraft:

vfront = get(follState, 'v');
psifront = get(follState, 'psi');
gfront = get(follState, 'gamma');
xfront = get(follState, 'x') + sd*cos(psifront)*cos(gfront);
yfront = get(follState, 'y') + sd*sin(psifront)*cos(gfront);
sfront = sqrt(xfront^2 + yfront^2);
sfoll_c = sqrt(get(follState, 'x')^2 + get(follState, 'y')^2);
v_c = get(follState, 'v');

xPos = Q(:,4);
yPos = Q(:,5);
sPos = sqrt(xPos.^2 + yPos.^2);
alt = Q(:,3);
gam = Q(:,6);
ph = Q(:,7);
ps = Q(:,8);

% Now start loop! Step by step!

for stepNumber = 1: length,

    % what do we want to follow? Lead aircraft, or intent info? First
    % we started with lead info and now we are going to move on depending on
    % update interval:

    newStep = stepNumber - 1;

    if (mod(newStep,UpdateInterval)-p == 0) % have we recieved info from the leader? ✓
If so, follow leader!

        %Run follow the front!

        frontState = FPS_state(Q(stepNumber+1,:));

        %now need data from frontState
        vfront = get(frontState, 'v');
        xfront = get(frontState, 'x');
        yfront = get(frontState, 'y');
        gfront = get(frontState, 'gamma');
        psifront = get(frontState, 'psi');
        sfront = sqrt(xfront^2 + yfront^2);

        % now get data from follower:
        vfoll = get(follState, 'v');
        xfoll = get(follState, 'x');
        yfoll = get(follState, 'y');
        psifoll = get(follState, 'psi');
        sfoll = sqrt(xfoll^2 + yfoll^2);

        vinit = get(follState, 'v');
```

```
[s, vfol1] = FPS_PDIntegrator(vfront, sfront, vfol1, sfoll, sd, stepTime, Fol  
lowerNumber);  
  
% put limit on acceleration:  
  
a = (vfol1-vinit)/stepTime; %approximate acceleration  
g = 32.2; %acceleration due to gravity  
  
if a > 0.05*g  
    a = 0.05*g;  
    vfol1 = a*stepTime + vinit;  
end  
  
if a < -0.05*g  
    a = -0.05*g;  
    vfol1 = a*stepTime + vinit;  
end  
  
%  
  
%set limits on the velocity  
W = 200000;  
C_LR = 1.4;  
S = 1994;  
h = get(follState, 'h');  
rho = FPS_alt2rho(h);  
v_stall = sqrt(2*W/C_LR/S/rho);  
  
if vfol1 < v_stall;  
    vfol1 = v_stall;  
end  
  
if FPS_fps2kts(vfol1) > 450  
    vfol1 = FPS_kts2fps(450);  
end  
  
% find x and y position  
psfol1 = get(follState, 'psi');  
gfol1 = get(follState, 'gamma');  
xfol1 = get(follState, 'x') + (vfol1*stepTime)*cos(psfol1)*cos(gfol1);  
yfol1 = get(follState, 'y') + (vfol1*stepTime)*sin(psfol1)*cos(gfol1);  
sfoll_c = sqrt(xfol1^2 + yfol1^2);  
v_c = vfol1;  
  
%determine the reference path  
  
s = sqrt(xfol1^2 + yfol1^2);  
  
%set bounds  
if sqrt(xfol1^2 + yfol1^2) <= sPos(1);  
    hfol1 = P(1,3);  
    gfol1 = P(1,6);  
    phfol1 = P(1,7);  
    psfol1 = P(1,8);  
end  
  
if sqrt(xfol1^2 + yfol1^2) >= sPos(end);  
    hfol1 = P(end,3);  
    gfol1 = P(end,6);
```



```
        phfoll = P(end,7);
        psfoll = P(end,8);
    end
    %if within bounds, then:
    if sqrt(xfoll^2 + yfoll^2) >= sPos(1) & sqrt(xfoll^2 + yfoll^2) <= sPos(end)
        hfoll = interp1(sPos, alt,s, 'linear');
        gfoll = interp1(sPos,gam,s, 'linear');
        phfoll = interp1(sPos,ph,s, 'linear');
        psfoll = interp1(sPos,ps,s, 'linear');
    end

    %set the reference state
    refState = set(refState, 't', stepNumber);
    refState = set(refState, 'v', vfall);
    refState = set(refState, 'h', hfoll);
    refState = set(refState, 'x', xfoll);
    refState = set(refState, 'y', yfoll);
    refState = set(refState, 'gamma', gfoll);
    refState = set(refState, 'phi', phfoll);
    refState = set(refState, 'psi', psfoll);

    [W,L,D,T0]=FPS_B757getPptys(vinit,vfall,hfoll,gfoll,stepTime);

    refState = set(refState, 'W', W);
    refState = set(refState, 'L', L);
    refState = set(refState, 'D', D);
    refState = set(refState, 'T', T0);

    %where is reference?
    vf = get(refState, 'v');
    hf = get(refState, 'h');
    xf = get(refState, 'x');
    yf = get(refState, 'y');
    gaf = get(refState, 'gamma');
    phif = get(refState, 'phi');
    psif = get(refState, 'psi');

    %now follow the reference path using controller:

    initialdx = FPS_delFinder(refState, follState);

    dv0 = get(initialdx, 'v'); %gets initial value for velocity
    dh0 = get(initialdx, 'h'); %gets initial value for altitude
    dx0 = get(initialdx, 'x'); %gets initial value for x
    dy0 = get(initialdx, 'y'); %gets initial value for y
    dga0 = get(initialdx, 'gamma'); %gets initial value for pitch angle
    dphi0 = get(initialdx, 'phi'); %gets initial value for bank angle
    dpsi0 = get(initialdx, 'psi'); %gets initial value for heading angle

    % get initial reference conditions for LQR controller:
    vref = get(refState, 'v');
    href = get(refState, 'h');
    gref = get(refState, 'gamma');
    phref = get(refState, 'phi');
    psiref = get(refState, 'psi');
    Dref = get(refState, 'D');
```

```
Lref = get(refState,'L');
Tref = get(refState,'L');
Wref = get(refState,'W');

inputToController = [vref, href, gref, phref, psiref, Dref, Lref, Tref, Wref,
dv0 dh0 0 0 dga0 dphi0 dpsi0];

dxState = FPS_controller(inputToController);
writeToFile(dxState,dxFileName);

dv = get(dxState,'v');
dh = get(dxState,'h');
dx = get(dxState,'x');
dy = get(dxState,'y');
dga = get(dxState,'gamma');
dphi = get(dxState,'phi');
dpsi = get(dxState,'psi');
dT = get(dxState,'T');

vnew = dv + vref;
dv_real = vfol1 - vnew;

if dv_real > 0.05*g/stepTime
    dv_real = 0.05*g/stepTime;
end
if dv_real < -0.05*g/stepTime
    dv_real = -0.05*g/stepTime;
end

hnew = dh + href;
dh_real = hfol1 - hnew;

if dh_real > 10/stepTime
    dh_real = 10/stepTime;
end
if dh_real < -10/stepTime
    dh_real = -10/stepTime;
end

ganew = dga + gref;
dga_real = gfol1 - ganew;

%next we need to add the dxState values to the refState values:
follState = set(follState,'t',(stepNumber + stepTime -1));
follState = set(follState,'v',(vfol1 - dv_real));
follState = set(follState,'h',(hfol1 - dh_real));
x = get(follState,'x') + ((vfol1 - dv_real)*stepTime)*cos(psfoll)*cos(gfol1);
y = get(follState,'y') + ((vfol1 - dv_real)*stepTime)*sin(psfoll)*cos(gfol1);
follState = set(follState,'x',x);
follState = set(follState,'y',y);
follState = set(follState,'gamma',(gfol1 - dga_real));
follState = set(follState,'phi',(phif + dphi));
follState = set(follState,'psi',(psif + dpsi));

%Now we need to find the D,L and T values and set them:

[W,L,D,Th]=FPS_B757getPptys(vinit,vf+dv,hf+dh,gaf+dga,stepTime);
```

```
    follState = set(follState, 'L', L);  
    follState = set(follState, 'D', D);  
    follState = set(follState, 'T', Th);  
  
else  
    %If we aren't getting info from front aircraft (step number does  
    %not equal update interval!)  
    vinit = get(follState, 'v');  
  
    %assuming constant errors in between updates:  
    vfol1 = a*stepTime + vinit;  
  
    %  
    %set limits on the velocity  
    W = 200000;  
    C_LR = 1.4;  
    S = 1994;  
    h = get(follState, 'h');  
    rho = FPS_alt2rho(h);  
    v_stall = sqrt(2*W/C_LR/S/rho);  
  
    if vfol1 < v_stall;  
        vfol1 = v_stall;  
    end  
  
    if FPS_fps2kts(vfol1) > 450  
        vfol1 = FPS_kts2fps(450);  
    end  
  
    % find x and y position  
    psfol1 = get(follState, 'psi');  
    gfol1 = get(follState, 'gamma');  
    xfol1 = get(follState, 'x') + (vfol1*stepTime)*cos(psfol1)*cos(gfol1);  
    yfol1 = get(follState, 'y') + (vfol1*stepTime)*sin(psfol1)*cos(gfol1);  
    sfol1_c = sqrt(xfol1^2 + yfol1^2);  
    v_c = vfol1;  
  
    %determine the reference path  
    s = sqrt(xfol1^2 + yfol1^2);  
  
    %set bounds  
    if sqrt(xfol1^2 + yfol1^2) <= sPos(1);  
        hfol1 = P(1,3);  
        gfol1 = P(1,6);  
        phfol1 = P(1,7);  
        psfol1 = P(1,8);  
    end  
  
    if sqrt(xfol1^2 + yfol1^2) >= sPos(end);  
        hfol1 = P(end,3);  
        gfol1 = P(end,6);  
        phfol1 = P(end,7);  
        psfol1 = P(end,8);  
    end  
  
    %if within bounds, then:  
    if sqrt(xfol1^2 + yfol1^2) >= sPos(1) & sqrt(xfol1^2 + yfol1^2) <= sPos(end)  
        hfol1 = interp1(sPos, alt, s, 'linear');
```

```
        gfol1 = interp1(sPos,gam,s,'linear');
        phfol1 = interp1(sPos,ph,s,'linear');
        psfol1 = interp1(sPos,ps,s,'linear');
    end

    %set the reference state
    refState = set(refState,'t',stepNumber);
    refState = set(refState,'v',vfol1);
    refState = set(refState,'h',hfol1);
    refState = set(refState,'x',xfol1);
    refState = set(refState,'y',yfol1);
    refState = set(refState,'gamma',gfol1);
    refState = set(refState,'phi',phfol1);
    refState = set(refState,'psi',psfol1);

    [W,L,D,T0]=FPS_B757getPptys(vinit,vfol1,hfol1,gfol1,stepTime);

    refState = set(refState,'W',W);
    refState = set(refState,'L',L);
    refState = set(refState,'D',D);
    refState = set(refState,'T',T0);

    %where is reference?
    vf = get(refState,'v');
    hf = get(refState,'h');
    xf = get(refState,'x');
    yf = get(refState,'y');
    gaf = get(refState,'gamma');
    phif = get(refState,'phi');
    psif = get(refState,'psi');

    %now follow the reference path using controller:
    initialdx = FPS_delFinder(refState, follState);

    dv0 = get(initialdx,'v'); %gets initial value for velocity
    dh0 = get(initialdx,'h'); %gets initial value for altitude
    dx0 = get(initialdx,'x'); %gets initial value for x
    dy0 = get(initialdx,'y'); %gets initial value for y
    dga0 = get(initialdx,'gamma'); %gets initial value for pitch angle
    dphi0 = get(initialdx,'phi'); %gets initial value for bank angle
    dpsi0 = get(initialdx,'psi'); %gets initial value for heading angle

    % get initial reference conditions for LQR controller:
    vref = get(refState,'v');
    href = get(refState,'h');
    gref = get(refState,'gamma');
    phref = get(refState,'phi');
    psiref = get(refState,'psi');
    Dref = get(refState,'D');
    Lref = get(refState,'L');
    Tref = get(refState,'T');
    Wref = get(refState,'W');

    inputToController = [vref, href, gref, phref, psiref, Dref, Lref, Tref, Wref,
    dv0 dh0 0 0 dga0 dphi0 dpsi0];

    dxState = FPS_controller(inputToController);
```

```
writeToFile(dxState,dxFileName);

dv = get(dxState,'v');
dh = get(dxState,'h');
dx = get(dxState,'x');
dy = get(dxState,'y');
dga = get(dxState,'gamma');
dphi = get(dxState,'phi');
dpsi = get(dxState,'psi');
dT = get(dxState,'T');

vnew = dv + vref;
dv_real = vfol1 - vnew;

if dv_real > 0.05*g/stepTime
    dv_real = 0.05*g/stepTime;
end
if dv_real < -0.05*g/stepTime
    dv_real = -0.05*g/stepTime;
end

hnew = dh + href;
dh_real = hfol1 - hnew;

if dh_real > 10/stepTime
    dh_real = 10/stepTime;
end
if dh_real < -10/stepTime
    dh_real = -10/stepTime;
end

ganew = dga + gref;
dga_real = gfol1 - ganew;

%next we need to add the dxState values to the refState values:
follState = set(follState,'t',(stepNumber + stepTime -1));
follState = set(follState,'v',(vfol1 - dv_real));
follState = set(follState,'h',(hfol1 - dh_real));

x = get(follState,'x') + ((vfol1 - dv_real)*stepTime)*cos(psfoll)*cos(gfol1);
y = get(follState,'y') + ((vfol1 - dv_real)*stepTime)*sin(psfoll)*cos(gfol1);

follState = set(follState,'x',x);
follState = set(follState,'y',y);

follState = set(follState,'gamma',(gfol1 - dga_real));
follState = set(follState,'phi',(phif + dphi));
follState = set(follState,'psi',(psif + dpsi));

%Now we need to find the D,L and T values and set them:
[W,L,D,Th]=FPS_B757getPtys(vinit,vf+dv,hf+dh,gaf+dga,stepTime);

follState = set(follState,'L',L);
follState = set(follState,'D',D);
follState = set(follState,'T',Th);
```

end

```
%Finally, need to write this data to a file:
writeToFile(follState, outputFileNames)

%Print a little something to let the user know that Matlab is working
fprintf(' ');

if (mod(stepNumber,50) == 0) %change lines every 50 dots
    fprintf('\n');
end

%now go again!
stepNumber = stepNumber +1;

end

%let me know when it's done:
fprintf('Follower #%i done!\n', i)

%this is how to read the file created
M = csvread(outputFileName);
return
```

Appendix E: Controllers and Integrators

This segment contains all controllers and integrators used in this simulation. It also includes *FPS_Segment*, which calls to the step integrator *FPS_Step*. These two functions are used in the reference flight path function *FPS_FlightPath*. *FPS_Segment* runs a segment until a stopping condition has been reached by calling to *FPS_Step*, and feeding it the aerodynamic model, the initial conditions, and the states that are to be constrained and their values. The step function in turn calls to the aerodynamic model and integrates it for a small interval, checking in between each interval to see if it had reached a stopping condition.

The reference path following controller is simulated using two functions. The *FPS_delFinder* is used to calculate the initial conditions and then *FPS_controller* runs the LQR on the linearized equations of motion.

The aircraft following controls are run by two functions. *FPS_PDIntegrator* calls to *FPS_PDController* for the differential equations for the case with no lead feedback. *FPS_PDIntegratorLFB* (not included) is exactly the same as *FPS_PDIntegrator* except that it calls to *FPS_PDControllerLFB* (included) for the differential equations for the case with lead feedback.

```

function finalState = FPS_Segment(outputFilename, initialState, stepTime, stopConditions,
    aircraft, aerodynamicModel, inputToAeroModel)
%
%function finalState = FPS_Segment(initialState, stepTime, stopConditions, aircraft, aero
dynamicModel, inputToAeroModel)
%
% INPUT
%
%     outputFilename: name of the file to which flight state data is
%                     written (e.g. './flight path simulator/leadFlightLog.txt')
%     initialState: initial conditions of type FPS_state
%     stepTime: length (in seconds) of one discrete step through time
%               (after which a check against the stopConditions is made)
%     stopConditions: conditions that, if true, end the segment (see
%                   below for more information)
%     aircraft: aircraft data function (e.g. @FPS_B757)
%     aerodynamicModel: aerodynamic model function (e.g. @FPS_aero_model)
%     inputToAeroModel: an array of inputs to the aerodynamic model...
%                     not all of them need this (e.g. [2,500], or [])
%
% OUTPUT
%
%     finalState: final conditions when a stopping condition has been
%               reached (of type FPS_state)
%
% STOP CONDITIONS
%
%     syntax: [stopCondition, value] where stopCondition is 1,2,3 or 4
%     stopCondition = 1 means time ('value' is in seconds)
%     stopCondition = 2 means altitude above aircraft ('value' is in ft)
%     stopCondition = 3 means altitude below aircraft ('value' is in ft)
%     stopCondition = 4 means velocity to slow down to ('value' is in fps)
%     stopCondition = 5 means to stop turning once a certain heading
%                     angle (in radians) has been reached
%     stopCondition = 6 means to stop segment once a certain x-distance
%                     has been achieved
%     stopCondition = 7 means to stop segment once a minimum bank
%                     angle is achieved
%     stopCondition = 8 means to accelerate to ('value' is in fps)
%
%     e.g. entering [2,30000] means to climb to 30,000 ft
%
finalState = initialState;
exit = 0; %set to 1 when a stop condition is reached
numberOfIterations = 0; %track the number of iterations, so that loop is not infinite
                    %if an erroneous stop condition is entered

maxIterations = 86400/stepTime; %24 hours is the maximum amount of time before we ass
ume that the user
                    %made a mistake with the stop condition

while exit == 0
    %First, we proceed one step forward
    finalState = FPS_Step(finalState, stepTime, aircraft, aerodynamicModel, inputToAe
roModel);
    writeToFile(finalState, outputFilename)

```



```
%Second, we analyse the result, and see if a stop condition has
%been reached. If we must stop, then exit the loop, otherwise
%continue with the next step.
switch stopConditions(1)
    case 1
        if (get(finalState, 't') - get(initialState, 't')) >= stopConditions(2)
            exit = 1;
        end
    case 2
        if get(finalState, 'h') >= stopConditions(2)
            exit = 1;
        end
    case 3
        if get(finalState, 'h') <= stopConditions(2)
            exit = 1;
        end
    case 4
        if get(finalState, 'v') <= stopConditions(2)
            exit = 1;
        end
    case 5
        if get(finalState, 'psi') >= stopConditions(2)
            exit = 1;
        end
    case 6
        if get(finalState, 'x') - get(initialState, 'x') >= stopConditions(2)
            exit = 1;
        end
    case 7
        if get(finalState, 'phi') <= stopConditions(2)
            exit = 1;
        end
    case 8
        if get(finalState, 'v') >= stopConditions(2)
            exit = 1;
        end
    otherwise
        error('FPS_Segment: incorrect stop condition entered')
        exit = 1; %prevent loop from going forever
end

%Third, we keep track of the number of iterations to make sure that
%the loop is not infinite
numberOfIterations = numberOfIterations + 1;
if numberOfIterations > maxIterations
    error('FPS_Segment: infinite loop detected (segment lasted more than 24 hours ✓
)')
    exit = 1;
end

%Print a little something to let the user know that Matlab is working
fprintf('.'.');

if (mod(numberOfIterations,50) == 0) %change lines every 50 dots
    fprintf('\n');
end

end
return
```

```
function stepState = FPS_Step(initialState, stepTime, aircraft, aerodynamicModel, inputTo AeroModel)
%
%function stepState = FPS_Step(initialState, stepTime, aircraft, aerodynamicModel, inputTo AeroModel)
%
% INPUT
%
%     initialState: initial conditions of type FPS_state
%     stepTime: length (in seconds) of one discrete step through time
%     aircraft: aircraft data function (e.g. @FPS_B757)
%     aerodynamicModel: aerodynamic model mode number, see below for more info (e.g. 2)
%     inputToAeroModel: an array of inputs to the aerodynamic model...
%                       input depends on aerodynamicModel (e.g. [2,500], or [])
%
% OUTPUT
%
%     stepState: final conditions after one step (of type FPS_state)
%
% AERODYNAMIC MODEL MODE NUMBERS
%
%     Mode #      Constant Characteristics      inputToAeroModel
%     -----      -
%     1           CAS, gamma                    CAS
%     2           CAS, rate of climb (R/C)      [CAS, R/C]
%     3           thrust setting (Pi), gamma    Pi
%     4           Pi, R/C                      [Pi, R/C]
%     5           CAS, altitude (h)            CAS
%     6           CAS, gamma, turn rate (p)     [CAS, p]
%     7           Idle Thrust (not const), gamma []
%     8           Idle Thrust (not const), R/C  R/C
%     9           CAS, phi                     [CAS, phi]
%     10          none! Following just the aero model
%     11          none                        [W,L,D,T, p]
%     12          turn rate (psi dot), gamma,CAS [CAS, psidot]
%     13          acceleration, altitude       [a, p]

v0 = get(initialState,'v'); %gets initial value for velocity
h0 = get(initialState,'h'); %gets initial value for altitude
x0 = get(initialState,'x'); %gets initial value for x
y0 = get(initialState,'y'); %gets initial value for y
ga0 = get(initialState,'gamma'); %gets initial value for pitch angle
phi0 = get(initialState,'phi'); %gets initial value for bank angle
psi0 = get(initialState,'psi'); %gets initial value for heading angle
D0 = get(initialState,'D'); %gets initial value for drag
L0 = get(initialState,'L'); %gets initial value for lift
T0 = get(initialState,'T'); %gets initial value for thrust
W0 = get(initialState,'W'); %gets initial value for weight
mode = get(initialState,'mode'); %gets initial value for mode

switch aerodynamicModel
case 1
    modelFunction = @FPS_aero_model_cCAS;
case 2
    modelFunction = @FPS_aero_model_CAS_RC;
case 3
    modelFunction = @FPS_aero_model_cT;
```

```
case 4
    modelFunction = @FPS_aero_model_cT_RC;
case 5
    modelFunction = @FPS_aero_model_CAS_alt;
case 7
    modelFunction = @FPS_aero_model_gamma;
case 8
    modelFunction = @FPS_aero_model_RC;
case 10
    modelFunction = @FPS_aero_model;

otherwise
    error('FPS_Step: incorrect mode %#i given', aerodynamicModel)
end

[t,y]=ode23(modelFunction,[0 stepTime],[v0 h0 x0 y0 ga0 phi0 psi0], [], aircraft, inp
utToAeroModel);

time = get(initialState, 't') + stepTime;
vf=y(end,1);
h=y(end,2);
ga=y(end,5);

[W,L,D,T] = feval(aircraft, vf, h, ga);

stepState = FPS_state( [time, y(end,:), D, L, T, W, model] );

%After the step has been made, some data may have to be overridden

if aerodynamicModel == 1
    stepState = set(stepState, 'v', FPS_constCAS(inputToAeroModel(1), get(stepState,
'h')));
    stepState = set(stepState, 'T', D+W*sin(ga));
end

if aerodynamicModel == 2
    stepState = set(stepState, 'v', FPS_constCAS(inputToAeroModel(1), get(stepState,
'h'))); % for constant CAS
    stepState = set(stepState, 'gamma', FPS_constRC(get(stepState, 'v'), inputToAero
Model(2))); % for constant R/C
    stepState = set(stepState, 'T', D+W*sin(ga));
end

if aerodynamicModel == 4
    stepState = set(stepState, 'gamma', FPS_constRC(get(stepState, 'v'), inputToAero
Model(2))); % for constant R/C
end

if aerodynamicModel == 5
    stepState = set(stepState, 'v', FPS_constCAS(inputToAeroModel(1), get(stepState,
'h')));
    stepState = set(stepState, 'T', D+W*sin(ga));
end

if aerodynamicModel == 6
    stepState = set(stepState, 'v', FPS_constCAS(inputToAeroModel(1), get(stepState,
'h')));
```

```
        stepState = set(stepState, 'T', D+W*sin(ga));
    end

    if aerodynamicModel == 8
        stepState = set(stepState, 'gamma', FPS_constRC(get(stepState, 'v'), inputToAeroModel(1))); % for constant R/C
    end

    if aerodynamicModel == 9
        stepState = set(stepState, 'v', FPS_constCAS(inputToAeroModel(1), get(stepState, 'h'))); % for constant CAS
        stepState = set(stepState, 'T', D+W*sin(ga));
    end

    if aerodynamicModel == 12
        stepState = set(stepState, 'v', FPS_constCAS(inputToAeroModel(1), get(stepState, 'h')));
        stepState = set(stepState, 'T', D+W*sin(ga));
    end

return;
```

```
function initialdx = FPS_delFinder(refState, acState)
%
%   function initialdx = FPS_delFinder(refState, acState)
%
%   finds the del initial condition from the difference between the
%   reference path (refState) and the actual path traveled by aircraft
%   (acState)

%reference path
v0 = get(refState,'v');
h0 = get(refState,'h');
x0 = get(refState,'x');
y0 = get(refState,'y');
gamma0 = get(refState,'gamma');
phi0 = get(refState,'phi');
psi0 = get(refState,'psi');

%aircraft path
vac = get(acState,'v'); %gets initial value for velocity
hac = get(acState,'h'); %gets initial value for altitude
xac = get(acState,'x'); %gets initial value for x
yac = get(acState,'y'); %gets initial value for y
gac = get(acState,'gamma'); %gets initial value for pitch angle
phiac = get(acState,'phi'); %gets initial value for bank angle
psiac = get(acState,'psi'); %gets initial value for heading angle

%take difference
initialdx = zeros(1,15);
initialdx = FPS_state(initialdx);
newdv = v0-vac;
newdh = h0-hac;
newdx = x0-xac;
newdy = y0-yac;
newdga = gamma0-gac;
newdph = phi0-phiac;
newdps = psi0-psiac;

%set state
initialdx = set(initialdx,'v',newdv);
initialdx = set(initialdx,'h',newdh);
initialdx = set(initialdx,'x',newdx);
initialdx = set(initialdx,'y',newdy);
initialdx = set(initialdx,'gamma',newdga);
initialdx = set(initialdx,'phi',newdph);
initialdx = set(initialdx,'psi',newdps);

return
```

```

function dxState = FPS_controller(inputToController)
%
% [A1,B,C,K, dT] = FPS_controller(inputToController)
%
% Uses LQR to control aircraft flight with respect to a reference trajectory
%
% INPUT:
% v0: velocity of reference flight path
% h0: altitude of reference flight path
% gamma0: flight path angle of reference flight path
% phi0: bank angle of reference flight path
% psi0: heading angle of reference flight path
% D0: drag value of reference flight path
% L0: lift value of reference flight path
% T0: thrust value of reference flight path
% W: weight value of reference flight path
% dx: initial conditions of control values
%
% OUTPUT:
% A1: Modified A matrix from LQR
% B: Control matrix
% C: Output matrix
% K: LQR Output matrix
% dT: Thrust control

% Read the reference trajectory values from the log:

v0 = inputToController(1);
h0 = inputToController(2);
gamma0 = inputToController(3);
phi0 = inputToController(4);
psi0 = inputToController(5);
D0 = inputToController(6);
L0 = inputToController(7);
T0 = inputToController(8);
W = inputToController(9);
dv0 = inputToController(10);
dh0 = inputToController(11);
dx0 = inputToController(12);
dy0 = inputToController(13);
dga0 = inputToController(14);
dphi0 = inputToController(15);
dpsi0 = inputToController(16);

dx = [dv0 dh0 dx0 dy0 dga0 dphi0 dpsi0]';

%gravity constant:
g = 32.2;

ATM = std_atm;
R = 1545.349; %Universal Gas Constant in ft*lb/(lbmol*Rankine)
Te = ttempf(h0,ATM); %temperature at this altitude in rankine

hs=R*Te/g;

%To find Cd and Cl values: (FOR B757- must write separate program if
%introducing other planes into simulator)

```

```
S=1994;          % Gross Wing Area
a=FPS_asonic(h0); %speed of sound
Patm = FPS_pressure(h0)*2.0885; %atmospheric pressure in psf
Tatm = FPS_ttempf(h0); %temp in deg Rankine
Ra=1.7165e03; % ft^2/(sec^2 deg Rankine)
rho=Patm/(Ra*Tatm); % density calculation
M=v0/a; %aircraft's Mach number
qbar=.5*rho*v0^2; %dynamic pressure

Cl=W/(qbar*S);
L=W*cos(gamma0);
Cdp = 0.0153; %Drag polar at low Mach
K = 0.0483;
Cd = Cdp + K*Cl^2; %Parabolic Drag Coefficient

TvDv=-2*cos(gamma0)*(D0)/(L0)*g/(v0)*cos(phi0);
ThDh = -T0*g/(W*hs)+cos(gamma0)*D0*g/L0/hs;

%linearize equations of motion
A = [TvDv          ThDh 0 0 -g*cos(gamma0) 0 0;
     sin(gamma0)    0 0 0 v0*cos(gamma0) 0 0;
     cos(gamma0)*cos(psi0) 0 0 0 -v0*sin(gamma0)*cos(psi0) 0 -v0*cos(gamma0)*sin(psi0);
     cos(gamma0)*sin(psi0) 0 0 0 -v0*sin(gamma0)*sin(psi0) 0 v0*cos(gamma0)*cos(psi0);
     2*cos(gamma0)*g/v0^2 -1/v0*cos(gamma0)*g/hs 0 0 g*sin(gamma0)/v0 0 0;
     0 0 0 0 0 0;
     -g*tan(phi0)/v0^2 0 0 0 0 g*(sec(phi0))^2/v0 0];

% Control terms
B = [ g/W g/L0*(Cd/Cl) 0;
     0 0 0;
     0 0 0;
     0 g/W/v0 0;
     0 0 1;
     0 0 0];

C = eye(7);

r = 0.0005;
R = r*eye(3);

%Gain matrix
Q = [10 0 0 0 0 0 0;
     0 1000 0 0 0 0 0;
     0 0 5000 0 0 0 0;
     0 0 0 5000 0 0 0;
     0 0 0 0 0.01 0 0;
     0 0 0 0 0 0.01 0;
     0 0 0 0 0 0 0.01];

[K,S,E] = LQR(A,B,Q,R); %LQR Factorization

A1=A-B*K;
PHI = expm(A1);
```

```
dx_new = PHI*dx;  
  
dxState = FPS_State(zeros(7));  
dxState = set(dxState, 'v', dx_new(1));  
dxState = set(dxState, 'h', dx_new(2));  
dxState = set(dxState, 'x', dx_new(3));  
dxState = set(dxState, 'y', dx_new(4));  
dxState = set(dxState, 'gamma', dx_new(5));  
dxState = set(dxState, 'phi', dx_new(6));  
dxState = set(dxState, 'psi', dx_new(7));
```

```
return
```



```
function [s, v] = FPS_PDIntegrator(vfront, sfront, vfall, sfoll, sd, stepTime, FollowerNumber)
%
% [s, v] = FPS_PDIntegrator(vfront, sfront, vfall, sfoll, sd, stepTime, FollowerNumber)
%
% Integrator for the PD-Controller (separation maintenance controller)
%
% INPUTS:
%     * vfront, sfront: velocity (ft/s) and position (ft) of front
%       aircraft (N-1)
%     * vfall, sfoll: velocity (ft/s) and position (ft) of current
%       aircraft (N) (where you are)
%     * sd: separation distance required
%     * stepTime: step time interval (integration interval)
%     * FollowerNumber: where you are in the string, minus 1 (follower
%       #1 is first aircraft after lead) (N-1)
%
% OUTPUTS:
%     * s, v: final position and velocity of current aircraft (where you
%       want to be)
%
% first get data from lead aircraft
integratorInput = [sfront, vfront, sd];
%integrate!
[t,x]=ode15s(@FPS_PDController,[0 stepTime],[sfoll vfall], [], integratorInput);
s = x(end,1);
v = x(end,2);
return
```

```
function xdot = FPS_PDController(t, x, integratorInput)
%
% function xdot = FPS_PDController(t, x, integratorInput)
%
%     applies a PD controller to the position and velocity of the following aircraft
%     to maintain separation.  (according to Dr. Slater's paper)
%
%     ** non-lead feedback formulation

%pull out the data you need that was sent in from the integrator

sfront = integratorInput(1);
vfront = integratorInput(2);

sd = integratorInput(3);

% put in gain terms:
Kp = 1/(30^2);
Kd = 1/30;

eta = 0;

%get it setup!
xdot = zeros(2, 1);

    xdot(1) = x(2);
    xdot(2)= Kp*sfront - Kp*x(1) + Kd*vfront - Kd*x(2) + eta - Kp*sd;

return
```

```
function [s, v] = FPS_PDIntegratorLFB(vlead, slead, vfront, sfront, vfall, sfall, sd, stepTime, FollowerNumber)
%
% [s, v] = FPS_PDIntegratorLFB(vlead, slead, vfront, sfront, vfall, sfall, sd, stepTime, FollowerNumber)
%
% Integrator for the PD-Controller (separation maintenance controller) with
% lead feedback
%
% INPUTS:
%     * vlead, slead: velocity (ft/s) and position (ft) of lead
%       aircraft (AC 1- first aircraft in string)
%     * vfront, sfront: velocity (ft/s) and position (ft) of front
%       aircraft (N-1)
%     * vfall, sfall: velocity (ft/s) and position (ft) of current
%       aircraft (N) (where you are)
%     * sd: separation distance required
%     * stepTime: step time interval (integration interval)
%     * FollowerNumber: where you are in the string, minus 1 (follower
%       #1 is first aircraft after lead) (N-1)
%
% OUTPUTS:
%     * s, v: final position and velocity of current aircraft (where you
%       want to be)
%
% first get data from lead aircraft
integratorInput = [slead, vlead, sfront, vfront, sd, FollowerNumber];

%integrate!
[t,x]=ode15s(@FPS_PDControllerLFB,[0 stepTime],[sfall vfall], [], integratorInput);

s = x(end,1);

v = x(end,2);

return
```