Syntax-Based Guidance for Autonomous Aggressive Aerobatics in Urban Environments

by

Rodin Lyasoff

S.B., Massachusetts Institute of Technology (2002)

Submitted to the Department of Aeronautical Engineering
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2004

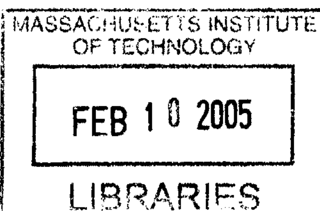
© Rodin Lyasoff, MMIV. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.

Author ....
Department of Aeronautical Engineering
August 6, 2004

Certified by ....
Eric Feron
Associate Professor of Aeronautics and Astronautics
Thesis Supervisor

Accepted by ....
Jaime Peraire
Professor of Aeronautics and Astronautics
Chairman, Department Committee on Graduate Students


MASSACHUSETTS INSTITUTE OF TECHNOLOGY
FEB 1 0 2005
LIBRARIES

AERO

Syntax-Based Guidance for Autonomous Aggressive Aerobatics in Urban Environments

by

Rodin Lyasoff

Submitted to the Department of Aeronautical Engineering
on August 6, 2004, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

This document proposes and implements a new guidance paradigm for an autonomous aerobatic helicopter. In addition, it presents the design of a hammerhead aerobatic maneuver on the same helicopter, and validates both designs in flight. The proposed guidance architecture integrates a new path-following algorithm with a syntax-based dynamically-reconfigurable automaton to enable autonomous flight with aggressive aerobatics in a spatially-constrained urban environment.

Thesis Supervisor: Eric Feron
Title: Associate Professor of Aeronautics and Astronautics

Acknowledgments

Thanks, first and foremost, to my parents: to my mother for raising me and letting me leave her for another continent, to my father, for making other continents a possibility, and to my stepmother. Their unconditional love and support have not wavered, as often as I've put them to the test. A humble bow in gratitude to my grandparents, who always answered my every question and indulged my every quest.

Many thanks to my advisor, Eric Feron, for his guidance and support, whenever necessary, and not always, and for allowing me to explore my intellectual and engineering passions, arbitrary as they sometimes are. I am much indebted to Vlad Gavrilets, who started this project and layed the foundations for my work. He has taught me much about autonomous helicopters and smart brassieres. Thanks also to Dr. James Paduano, and to everyone else closely involved in this project: Ioannis, Allen, Farmey, and Greg. They have never failed to lend a hand.

I owe thanks to all my friends at MIT who gave meaning to my undergraduate and graduate careers here and taught me much beyond academics: Riad, Amrys, Mar, Jim, Kristin, Kevin, Paul, David, Boris, Mary, Scott, Jen, Jenn, Jeff, Josh, Dale Mike, Teresa, the old Clam-Guard, and the Fort-Awesome alliance, Jan, Ji-Hyun, Bernard, Phil, Masha, Mardavij, Mario, Emily, Sommer. To everyone else who deserves it – thanks.

Finally, I would like to thank the Institvte.

To my parents.

Contents

1 Introduction **13**
1.1 Background and Motivation 13
1.2 Thesis Objectives 15
1.3 Thesis Organization 15

2 Experimental Setup **17**
2.1 Physical System 17
2.2 Avionics 17
2.3 Simulation Environment 19
2.4 Control Architecture 20
2.4.1 Velocity Control 20
2.4.2 Rate Control 21
2.4.3 State Machine Paradigm for Hybrid Control 21

3 Syntax-Based State Machine **23**
3.1 Path-Follower 24
3.2 Lateral Guidance on The MIT Helicopter 26
3.2.1 Step Response and Path Smoothing 27
3.3 Waypoints as Data Structures 27
3.4 Hovering 30
3.5 Loitering 30
3.5.1 Loiter Modes 31
3.6 Path Management 32
3.7 Action Queue 34
3.8 Advanced Mission Planning and Multi-Vehicle Support 35
3.9 Formation Flight and Moving Target Surveillance 36

4 Hammerhead Design **39**
4.1 Rate Controllers 39
4.2 Maneuver Logic 39
4.3 Hammerhead 42
4.4 Data Collection 43
4.5 Implementation and Validation 43

4.6 Hammerhead Implementation on the Yamaha R-Max Helicopter 45

5 Validation Flight 47

6 Conclusion and Future Work 51
6.1 Thesis Summary 51
6.2 Recommendations for Future Work 52

A Guidance Law Sensitivity Analysis 53

B Investigation of Loitering Behavior 57
B.1 Average Loiter Radius at Various Speeds and $\frac{L}{U}$ 57

C Condensed Path Manager Source Code 61

D Hammerhead Source Code 65

E Scripting the Mission 71
E.1 Script for Generating the Sample Mission 71

Bibliography 73

List of Figures

1-1 MIT's Helicopter performing a Split-S (courtesy of Popular Science Magazine). 14

2-1 The MIT Helicopter . . . 18
2-2 Hardware-in-the-Loop Simulation . . . 19
2-3 Longitudinal-Vertical Controller. . . . 20
2-4 Lateral-Directional Controller. . . . 21
2-5 Hybrid logic for a simple mission. . . . 22

3-1 The New Guidance State Machine . . . 24
3-2 Park's Illustration of Lateral Guidance [1]. . . . 26
3-3 Lateral Guidance Step Responses . . . 28
3-4 Lateral Guidance Step Response with Path Smoothing . . . 29
3-5 Variation of Loiter Radius with Loiter Speed for Several $\frac{L}{U}$. . . . 32
3-6 Leader Position Ambiguity . . . 33
3-7 Target Following . . . 36

4-1 Idealized Rate Trajectory and Resulting Attitude Change . . . 40
4-2 Hammerhead: Pilot Inputs and Responses . . . 42
4-3 Autonomous Hammerhead . . . 44
4-4 Autonomous Split-S linked with Hammerhead . . . 45

5-1 Planned Mission Path . . . 48
5-2 Simulated and Recorded Trajectory . . . 49
5-3 Three consecutive executions of validation flight . . . 49

A-1 Guidance Law Performance for $\frac{L}{U} = 0.5..2.5$. . . . 54
A-2 Guidance Law Performance for $\frac{L}{U} = 3..5$. . . . 55

B-1 Loitering Behavior at Varying Speeds for $\frac{L}{U} = 1.5$. . . . 58
B-2 Loitering Behavior at Varying Speeds for $\frac{L}{U} = 2.5$. . . . 59
B-3 Loitering Behavior at Varying Speeds for $\frac{L}{U} = 3.5$. . . . 60

List of Tables

3.1 $\frac{L}{U}$ and Cross-Track Error 27
3.2 The waypoint data structure. 30
3.3 Helicopter Action Codes 30
3.4 Loiter Radius and Loiter Speed 31

Chapter 1

Introduction

1.1 Background and Motivation

CURRENT trends in unmanned aerial vehicles (UAVs) indicate an interest in extending their autonomy as much as possible. Many of the most successful vehicles, such as the Predator [2], [3] are entirely remote-controlled by a human operator, while some, such as Aerovironment's Pointer and Raven, are capable of simple autonomous navigation among a limited number of waypoints [4]. Higher levels of autonomous operation are rare, and are represented by largely experimental efforts.

The MIT autonomous aerobatic helicopter is one such experimental system. It is based on an off-the-shelf airframe with a 5-foot rotor diameter, originally designed for RC hobbyists. A distinguishing feature of the helicopter is its ability to perform aggressive aerobatic maneuvers, such as an aileron roll, a split-s (Figure 1-1) or a hammerhead (Chapter 4). In this respect, the MIT helicopter is unique among UAVs. The rigidity of its rotorhead allows for large rotor control moments, while its small size results in small moments of inertia. This extreme agility and high thrust-to-weight ratio enable the helicopter to address mission scenarios previously inaccessible to UAVs. A particularly relevant application is street-level surveillance.

Urban environments are attracting growing interest as a theater for UAV deployment. Their geography makes path-following a critical component of any UAV system. Other experimental projects have addressed the problem of accurate path-following. LaCivita et al. [5] have demonstrated an H_∞ controller for trajectory-tracking. Johnson and Kannan [6], [7] have developed a neural-network approach to the same end. Both efforts have performed well in tracking simple[1] paths at moderate speeds. However, surveillance and threat avoidance in the "urban jungle" requires accurate and aggressive tracking of geometrically complex trajectories at high speeds.

Until recently, navigation on the MIT helicopter had been accomplished using a simple cross-track error minimization algorithm between discrete points. This essentially replicated

[1] Johnson has reportedly succeeded in tracking a hammerhead-like position/attitude trajectory in 2002. To the best of the author's knowledge, the results haven't been published.

Figure 1-1: MIT's Helicopter performing a Split-S (courtesy of Popular Science Magazine).

the capabilities of most commercial UAVs, such as the Raven or the Aerosonde [8], but failed to take full advantage of the vehicle's unique agility, and limited the scope of its missions.

Until now, the MIT helicopter has been largely a technology demonstrator. The main purpose of its flights has been to show off the basic capabilities described above. While its agility is unmatched by any other autonomous UAV, the guidance architecture lacks some of the features of more complete, production-ready vehicles. While fully autonomous, its flights have been predetermined, simulated, and hard-coded into firmware. The flight sequences could not be altered in real time. Doing so required direct interaction with the low-level software, extensive testing, and recompilation. Thus, the end-user was required to possess firm knowledge of the complete UAV system. This made the helicopter less accessible as a generic flight platform, and limited its applications.

The design of aerobatic maneuvers had similar limitations. It is a non-trivial problem, involving a somewhat tedious, iterative process, requiring familiarity with the inner workings of the control software. Consequently, maneuvers had previously been designed solely by the project's originator [9]. It had become necessary to demonstrate a repeatable, methodical

procedure for maneuver design, one that would be accessible to most controls engineers, and demonstrate its portability to other UAV systems.

As confidence grows in the reliability of the low-level hardware and software processes, the helicopter has begun to transition from a technology demonstrator toward a more versatile, more user-friendly platform, ready to serve the needs of a variety of users. To enable this transition, it has become necessary to address the concerns presented above.

1.2 Thesis Objectives

This thesis proposes, implements, and validates a new guidance architecture paradigm for an autonomous aerobatic helicopter. In addition, it demonstrates the design of an autonomous hammerhead aerobatic maneuver entirely in a laboratory environment, and validates the design in flight.

The guidance architecture proposed and implemented in this thesis is a syntax-based dynamically-reconfigurable automaton, which extends the capabilities of the hybrid system developed by Frazzoli [10] and Gavrilets [9]. The new automaton is a generic finite state machine (FSM), whose states and transitions can be specified dynamically in flight. The FSM logic is abstracted away from the operator, making the vehicle operable with a minimal amount of training. The automaton provides an integrated framework for prescribing complete missions and modifying them in real-time, while remaining easily adaptable to various high-level functions. The result is a more versatile platform, capable of addressing the needs of a variety of users, no longer requiring knowledge of the low-level hardware or software components. The proposed system provides for fully autonomous flight without restriction on the flight envelope, thus enhancing the usability of the vehicle without limiting its unique capabilities. The basic principles presented in this document are demonstrated on a rotary wing vehicle but are readily adaptable to any unmanned platform.

Firstly, this thesis will present the current guidance model and mode of interaction with the vehicle. Secondly, a new guidance architecture will be introduced, integrated around a much-improved path-following algorithm. Next, the complete design of a hammerhead aerobatic maneuver will be demonstrated, from the initial data collection and analysis to the final validation in flight. Selected source code will be given in the form of pseudo-code. Lastly, the new guidance architecture will be validated in the design of a complete mission across the entire envelope of the vehicle, utilizing the hammerhead maneuver.

1.3 Thesis Organization

This thesis is divided into 6 chapters, including this introduction.

Chapter 2 describes the experimental setup, and should be read by anyone who is not familiar with the project.

Chapter 3 describes the new path-following algorithm and the guidance automaton.

Chapter 4 describes the design and validation of a hammehead aerobatic maneuver. Chapter 4 is self-contained.

Chapter 5 presents the results from a validation flight test for the guidance architecture developed in Chapter 3 and the autonomous hammerhead design from Chapter 4.

Chapter 6 provides a summary of the work, and some concluding remarks, as well as suggestions for future work.

Chapter 2

Experimental Setup

2.1 Physical System

THE MIT helicopter is based on an *X-Cell .60* airframe [11], shown in Figure 2-1. The empty weight is about 10 lbs, with fuel capacity of about 1 lb. The payload consists of a 7 lb avionics suite. The main rotor is about 5 ft in diameter, and is equipped with a Bell-Hiller stabilizer bar. It is a hingeless rotor, which increases the agility of the helicopter. The helicopter is powered by a .90 hobbyist glow-fuel engine, which is to say a piston engine with displacement of 15 cc, running on a mixture of methanol, nitromethane, and oil. The engine provides a peak power output of 3 hp. The helicopter is equipped with an electronic governor to maintain a constant RPM of 1600. The governor dynamics are included in the modeled helicopter dynamics.

2.2 Avionics

The avionics sensor suite consists of an inertial measurement unit (IMU), a global positioning system (GPS) receiver, and a barometric altimeter. The particular sensor devices were selected for their high accuracy, high bandwidth and low latency.

The IMU is assembled by Inertial Science [12]. It contains three accelerometers, and three gyroscopes, based on Micro-Electro-Mechanical-Systems (MEMS) technology. The gyros have a range of ±300 deg/sec, and the accelerometers have a range of ±5 g. The helicopter's physical rates were limited in software to remain within the sensor ranges. The IMU has internal power regulation and temperature compensation. It is equipped with analog anti-aliasing filters operating at 20 Hz. Data output is at 100 Hz via serial link (RS-232 compliant).

The GPS unit is the Ashtech G14, featuring 10 Hz update rate and a maximum of 50 msec latency. The GPS provides WAAS-corrected 2-D position and velocity(Doppler) measurements.

The barometric altimeter unit is the Honeywell HPA200, with resolution of 2 ft (0.001 psi). It features internal power regulation and temperature compensation. The internal

Figure 2-1: The MIT Helicopter

sampling rate is 120 Hz, averaged to produce 5 Hz RS-232 output. Ambient pressure changes are neglected for the short duration of the flight.

The flight computer is a DSP Design TP400 single-board Pentium MMX, with 32 MB of RAM, operating at 400 MHz, and 32 MB of non-volatile Flash storage, running the QNX 4.25 real-time operating system [13]. The flight software is embedded in the computer's Flash. It is written in ANSI-compliant C. The flight software operates at a rate of 50 Hz; this is the maximum rate for state estimation and servo actuation. The state estimator is a 16-state Extended Kalman Filter, described in detail in [9].

Communication with the ground is accomplished via wireless serial modems, the MaxStream 9XStream [14], operating at 900 MHz for a range of about 7 miles. The link is one-way, and the ground station uses it to continuously monitor a reduced state vector, updated at 3 Hz.

Pilot commands are received via a standard hobbyist receiver, and converted to digital form by an integrated A/D and servo-control board (a.k.a. the servo board), designed in-house. The servo board also converts the flight computer's digital commands to analog servo commands.

The avionics suite is enclosed in an aluminum box, to minimize electromagnetic interference, and is mounted on a custom-designed vibration-isolation system to minimize sensor noise.

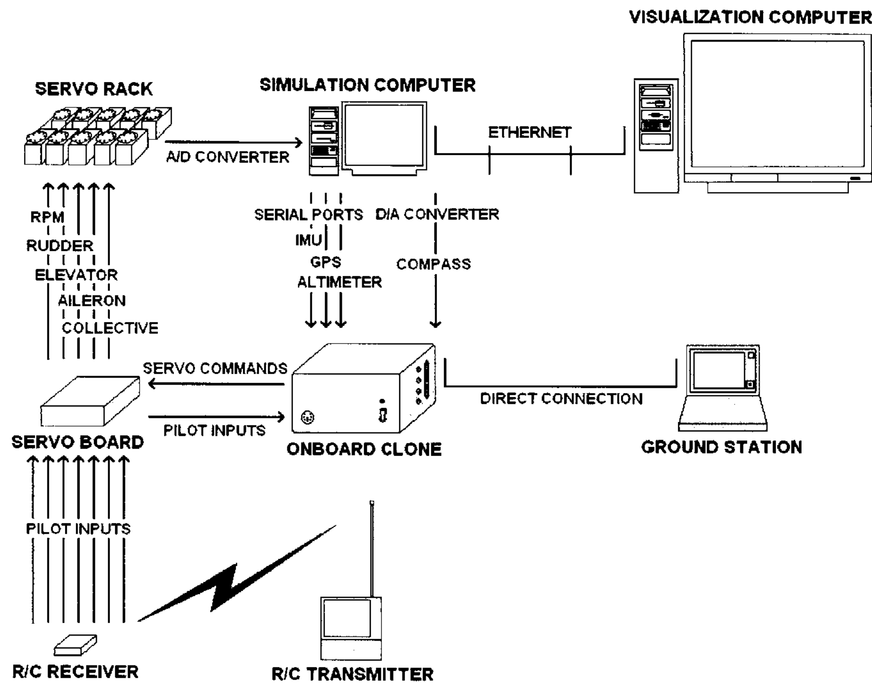


Figure 2-2: Hardware-in-the-Loop Simulation

2.3 Simulation Environment

All of the development described in this thesis took place in simulation before it was validated in flight. A hardware-in-the-loop simulation (HILSim) was employed. The overall structure is presented in Figure 2-2.

The HILSim uses exact copies of the flight computer (referred to as the clone computer), servos, RC receiver and servo board. The RC transmitter and ground station are the ones used during actual flights. The servo rack couples each actuator servo to a modified servo used only for sensing the actuator's position. The servo positions are measured by potentiometer. The measurements pass through an analog-to-digital converter and on to a simulation computer which calculates the aerodynamic forces and moments resulting from the control surface deflections. These data are used to simulate sensor output, which is then fed to the clone computer. The simulation also produces position and attitude estimates which are passed on to a visualization computer to provide a real-time 3D visualization of the flight.

While the servos should ideally be loaded to simulate forces and inertias, this was deemed unnecessary, given the bandwidth of the actuators [9].

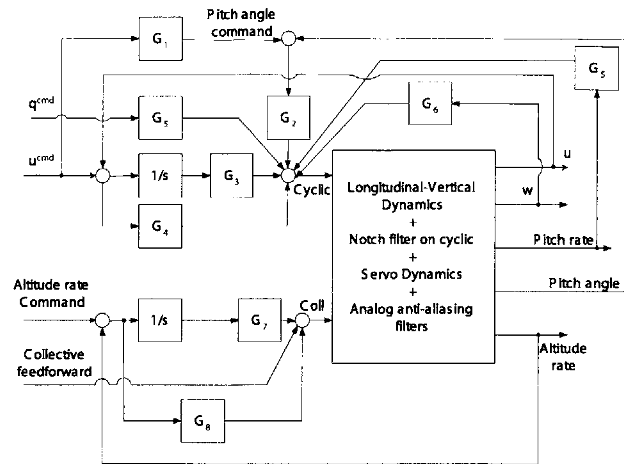


Figure 2-3: Longitudinal-Vertical Controller.

2.4 Control Architecture

Guidance on the MIT helicopter utilizes several low-level controllers. Navigation, or trim-trajectory-following is accomplished by a velocity-tracking loop, which implements 2D control of body-frame velocities (u, v), and a complementary altitude loop for full 3D navigation. Aerobatic maneuvers are executed by rate-tracking loops, providing control of the body-frame angular rates (p, q, r). Ultimately, the guidance architecture must enable the execution of advanced missions, combining accurate navigation in spatially-constrained environments with the ability to execute aerobatic maneuvers at points along the trajectory.

2.4.1 Velocity Control

Velocity control is accomplished by two decoupled loops, as developed in [9]: a longitudinal-vertical controller (Figure 2-3), and a lateral-directional controller (Figure 2-4). The combination of these controllers allows constant-altitude flight at a commanded forward speed, with coordinated turns at prescribed turn rates. Using this functionality, waypoint navigation was previously implemented by Gavrilets [9], utililizing a classical cross-track error minimization algorithm. The forward speed for waypoint navigation was a constant parameter (8 m/s). The guidance was aware of up to four waypoints at a time, and was capable of navigating among them in a prescribed order. A modified version of this guidance logic has been demonstrated by MIT's partner, Nascent Technologies, which allows the waypoint coordinates to be updated dynamically from the ground [15].

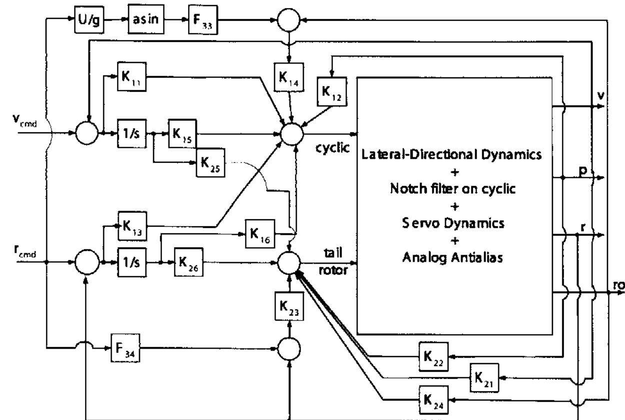


Figure 2-4: Lateral-Directional Controller.

2.4.2 Rate Control

In addition to the velocity controllers, the MIT helicopter is equipped with rate-tracking control loops, which implement independent command of the vehicle angular rates and of the main rotor collective. While the velocity controller provides the capabilities needed for trim-trajectory navigation the rate controllers take full advantage of the helicopter's agility, and are particularly well-suited for performing aerobatics, as will be shown in Chapter 4. The two control modes are mutually exclusive. It is the responsibility of the guidance logic to integrate these controllers into a coherent architecture, handing off vehicle control to the appropriate set of loops, as necessary for various phases of the mission.

2.4.3 State Machine Paradigm for Hybrid Control

The helicopter has evolved to perform a particular type of autonomous mission. Or, rather, some structure has been imposed on the mission scenarios to make the algorithmic implementation more feasible. The missions usually consist of navigation among several waypoints of interest, performing aerobatics in between. The navigation is performed by the velocity-and-turn-rate-tracking control loop, while the aerobatic maneuvers are executed by the body-rate-tracking control loops. Thus, the autonomous guidance layer operates in two major modes: trim-trajectory-tracking, and rate-tracking. Switching between the modes is delegated to the guidance automaton.

Figure 2-5 illustrates the basic logic for a simple mission. The helicopter climbs to 120 meters, heads toward a waypoint, accelerating to its maximum allowed forward speed, performs a Split-S and descends, heading toward another waypoint, and finally slowing down to a hover. The guidance logic is a finite state machine (FSM) that implements the desired sequence as it appears in the figure. In this case, the FSM consists of six states:

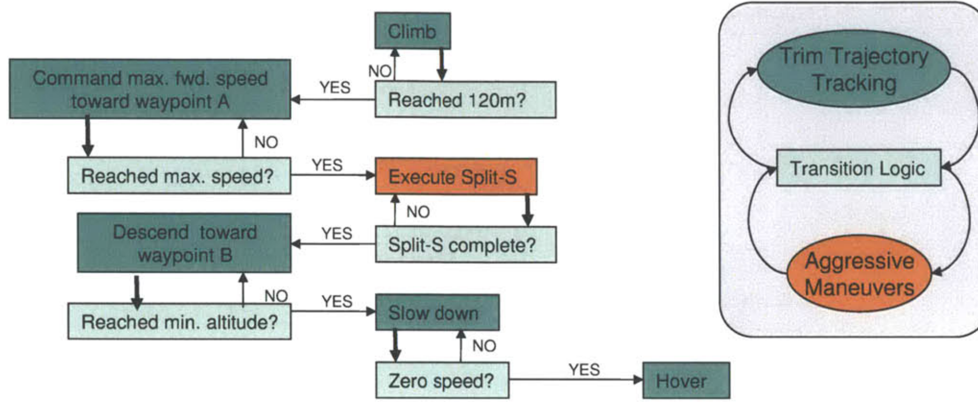


Figure 2-5: Hybrid logic for a simple mission.

climb, acceleration, Split-S, descent, deceleration, and hover. An actual mission may have as many as fifteen states.

Note, however, that the helicopter only switches control modes twice: it transitions from velocity-control to rate-control prior the executing the maneuver, and transitions back to velocity control after the maneuver is complete. The rest of the states are simply used to generate a sequence of commands to the velocity controller. This begs the question whether such a large number of states is really necessary. This is largely the motivation for Chapter 3.

Chapter 3

Syntax-Based State Machine

THE ORIGINAL guidance architecture had several shortcomings that limited its applicability in spatially constrained environments. Waypoint navigation, as previously implemented, used a classical navigation algorithm to minimize the cross-track error. This algorithm was well-suited to following straight lines, but was prone to overshoot oscillations when the cross-track error grew large. The algorithm was ill-suited for tracking corners, which are a dominant feature of urban terrains.

The FSM implementation of the guidance logic imposed another constraint on the vehicle capabilities. Varying speeds and altitudes could be commanded from state to state, but this paradigm required a new state for every new command, which dramatically increased the number of states, and resulted in unwieldy flight code. The parameters for each state and the conditions for state transition were hard-coded in the actual FSM. Thus, the FSM code served as its own data storage unit, as well the logic execution unit. This made the missions entirely static, as the logic could not be altered in flight.

The following sections describe a new guidance architecture which eliminates these problems by employing a new path-following algorithm, and changing the role of the FSM. The new path-follower is capable of accurately tracking continuous, geometrically complex paths, including corners. It is a variation of proportional navigation, and was first proposed by Park [1] for fixed-wing vehicles. It has been adapted for the helicopter, and expanded to act as a data storage and retrieval unit for the FSM. The new state machine is significantly simpler, and acts as an interpreter for the supplied data. This allows for missions of any complexity to be encoded with only a small, fixed number of states.

The new guidance paradigm was motivated by two important observations about UAV missions:

- The main component of any mission is the vehicle's commanded trajectory.
- Many parameters, such as altitude, forward speed, and the actions taken by the vehicle, are defined by its position along the trajectory.

For instance, the vehicle's desired altitude depends on terrain features, and thus on position. The same is true of forward speed: the complexity of the commanded path, when

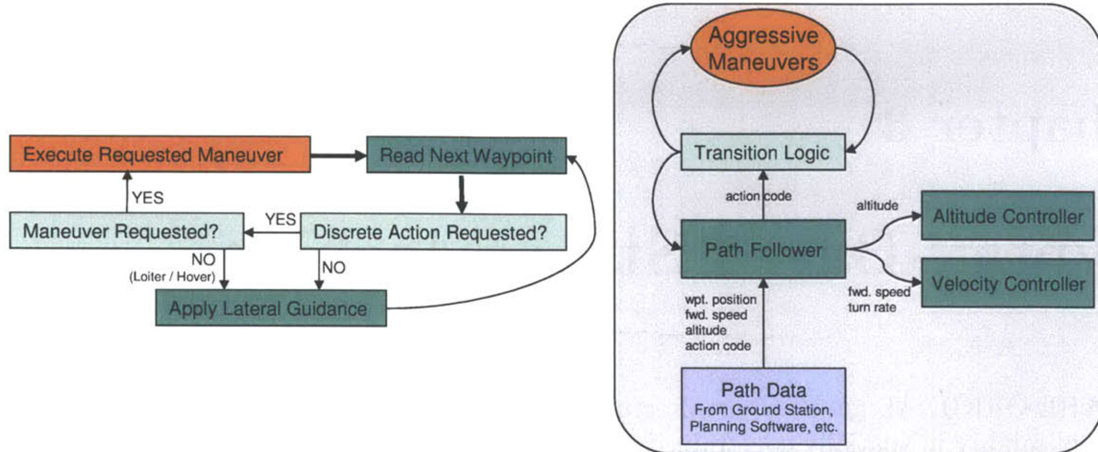


Figure 3-1: The New Guidance State Machine

combined with the vehicle dynamics, imposes strict limits on the maximum vehicle speed. Evasive aerobatic maneuvers are usually necessitated by obstacles at specific points along the path. Loitering is only performed in particular areas of interest.

Consequently, the parameters of a mission may be specified along its trajectory. A syntax can be established so that every point along the path prescribes a forward speed, an altitude, and a particular action to be taken, such as loitering or the execution of a maneuver. When the path-follower achieves a particular point, it would extract these parameters and pass them to the appropriate controllers. In the event that a maneuver is requested, the FSM could interrupt the path-follower and hand control over to the rate-tracking loops.. After the maneuver has been completed, the path-follower would be put in control again, and return the vehicle to the desired trajectory, as in the sample mission we examined in Section 2.4.3.

Thus, the guidance state machine can remain generic, with a minimal number of states, and effectively act as an interpreter for the mission parameters specified along the path. Figure 3-1 illustrates the proposed state machine architecture. The details of its operation are described in the following section.

3.1 Path-Follower

To address the problem of navigation in the "urban jungle", a vehicle must be able to track a quasi-continuous path in space, allowing for varying levels of complexity. The path is specified as a series of individual waypoints, relatively close together, but preferably with

no restrictions on their separation, so that any desired level of fidelity could be achieved. Few algorithms have been developed to accurately track such a path.

Navigation between waypoints is traditionally implemented using cross-track error techniques, i.e. designing a simple PD feedback controller to minimize the cross-track error. This was the method previously employed on the MIT helicopter, and it had several shortcomings. The transition from one waypoint to another required some complex switching heuristics, and the algorithm was prone to overshoot whenever the vehicle wandered too far off the desired path, which could easily happen when performing aerobatics, or under adverse wind conditions.

These problems can be alleviated by employing a nonphysical *virtual leader*, as proposed by Niculescu [8]. The leader travels between the waypoints, always ahead of the vehicle, which tries to follow it. While this algorithm gives satisfactory performance in following straight lines between waypoints, it is ill-suited for following other trajectories. In general, a simple virtual leader path-follower, when applied to a complex path, acts as a low-pass filter, parameterized by the lead distance, which is chosen to account for the dynamics of the vehicle. However, the smoothing effect of this approach effectively discards the high-frequency features of the trajectory, e.g. sharp corners, which are predominant in urban terrains.

Another approach to the tracking of individual waypoints is classical proportional navigation. Developed for missiles, proportional navigation aims to maintain a constant angle of the line-of-sight (LOS) between missile and target. This is generally done in two dimensions, though it has been extended to 3D by Adler [16]. For the purposes of this document, only the 2D case is relevant. Zarchan [17] gives the formal control law as follows:

$$n_c = N' V_C \dot{\lambda}$$

Here, n_c is the commanded acceleration perpendicular to the LOS, V_C is the closing speed between missile and target, $\dot{\lambda}$ is the rate of change of the LOS angle, and N' is a designer parameter gain. Proportional navigation is widely acknowledged as an accurate and efficient approach to interception, and thus to waypoint tracking. With some modifications, it can be successfully adapted to the problem of following continuous paths.

This has been done at MIT by Park [1]. His algorithm, titled *Lateral Guidance*, combines proportional navigation with a virtual leader, and has shown excellent performance in disturbance rejection and the tracking of high-frequency features. Its basic operation is shown in Figure 3-2.

The formal statement of lateral guidance is as follows:

$$a_{s_{cmd}} = 2\frac{U^2}{L}\sin\eta \tag{3.1}$$

Here, $a_{s_{cmd}}$ is a commanded lateral acceleration, U is the forward speed, L is the distance to the virtual leader, and η is the angle between the LOS to the leader and the vehicle's heading (assuming no side-slip, the latter coincides with the forward velocity vector). The lateral acceleration command effectively produces a turn rate, which causes the vehicle's

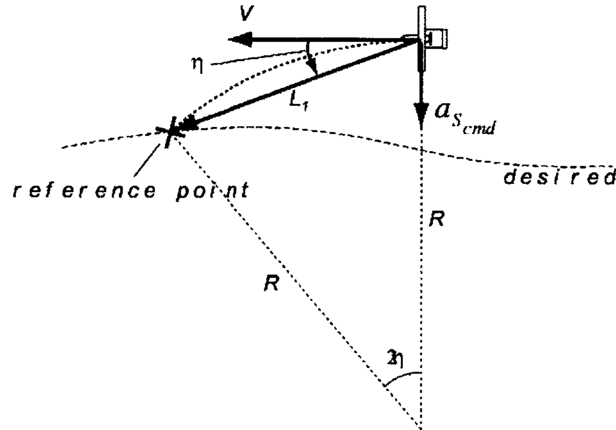


Figure 3-2: Park's Illustration of Lateral Guidance [1].

path to quickly converge with the desired path.

3.2 Lateral Guidance on The MIT Helicopter

Lateral guidance was developed for a fixed-wing vehicle with a limited range of forward speeds and turn-rates, whereas agile flight in urban environments requires large variations in both. Since the smoothing effects of the algorithm depend on L, it was desirable to be able to vary the parameter in real-time, to accomodate the complexity of the path. Since the tracking peformance also depends on forward speed, it was assumed that lower speed will be commanded through the high-frequency features of the trajectory. This would be the responsibility of a higher-level path-planning, or mission-planning algorithm, such as those developed at MIT by Schouwenaars [18].

Further analysis by Park, Deyst, and How [19] shows that, when linearized, the lateral guidance control law reduces to a PD feedback controller around the cross-track error. Taking $y \equiv$ cross-track error, and using small-angle approximations, Equation 3.1 becomes:

$$a_{s_{cmd}} = 2\frac{U}{L}\left(\dot{y} + \frac{U}{L}y\right) \tag{3.2}$$

Note that the ratio $\frac{L}{U}$ essentially determines the gains of the controller, and thus the tracking performance of the vehicle. Furthermore, the authors show the system to be critically damped, i.e. $\zeta = \frac{1}{\sqrt{2}}$, with a natural frequency $\omega_n = \sqrt{2}\frac{U}{L}$, where the transfer function input is the lateral position of the leader relative to the vehicle.

For implementation on the helicopter, it was decided to keep this ratio constant throughout the flight, commanding forward speed U, and adjusting the lead distance L accordingly. To be able to perform safe and effective surveillance in an urban environment, it was determined that the helicopter must be able to fly around corners at 8 m/s with a maximum

Table 3.1: $\frac{L}{U}$ and Cross-Track Error

$\frac{L}{U}$ (Lead/Speed)	Max. Cross-Track Error
0.5	17 m
1	2.3 m
1.5	0.8 m
2	1.2 m
2.5	2.3 m
3	3.2 m
3.5	5.0 m
4	7.0 m
4.5	8.0 m
5	9.0 m

cross-track error of 2 meters. The appropriate $\frac{L}{U}$ value for this mission was determined experimentally. Sensitivity analysis was performed in simulation; the results are summarized in Table 3.1, and in Appendix A. Finally, the value of 1.5 was selected, based on the simulation results. This has proven adequate, as shown in Chapter 5.

3.2.1 Step Response and Path Smoothing

The helicopter's step responses under the lateral guidance are is presented in Figure 3-3. Somewhat counterintuitively, the overshoot varies inversely with the amplitude of the step input. This suggests that the oscillatory tracking behavior observed in flight (Chapter 5, Appendix A) is more likely a non-linear effect. The trajectories observed in Appendix A suggest the existence of limit cycles, most likely resulting from a commanded turn rate which exceeds the physical limits of the vehicle. Increasing the lead distance of the path follower is very effective in eliminating these effects, albeit at the cost of tracking performance, as shown in the same appendix.

Alternately, step commands in position can generally be avoided during the path planning phase. Sharp corners can be replaced by smooth turns, whose turn radius is no smaller than the vehicle's turn radius ($\approx$ 10 meters @ 10 m/s for the MIT helicopter). The above step command was smoothed out in this fashion, and the path-following performance improved drastically, as shown in Figure 3-4.

3.3 Waypoints as Data Structures

A waypoint is simply a data structure, usually containing the GPS coordinates of a location in space. It can just as easily include the desired velocity at that point, or a trigger for the execution of a maneuver, or for initiating loitering or surveillance.

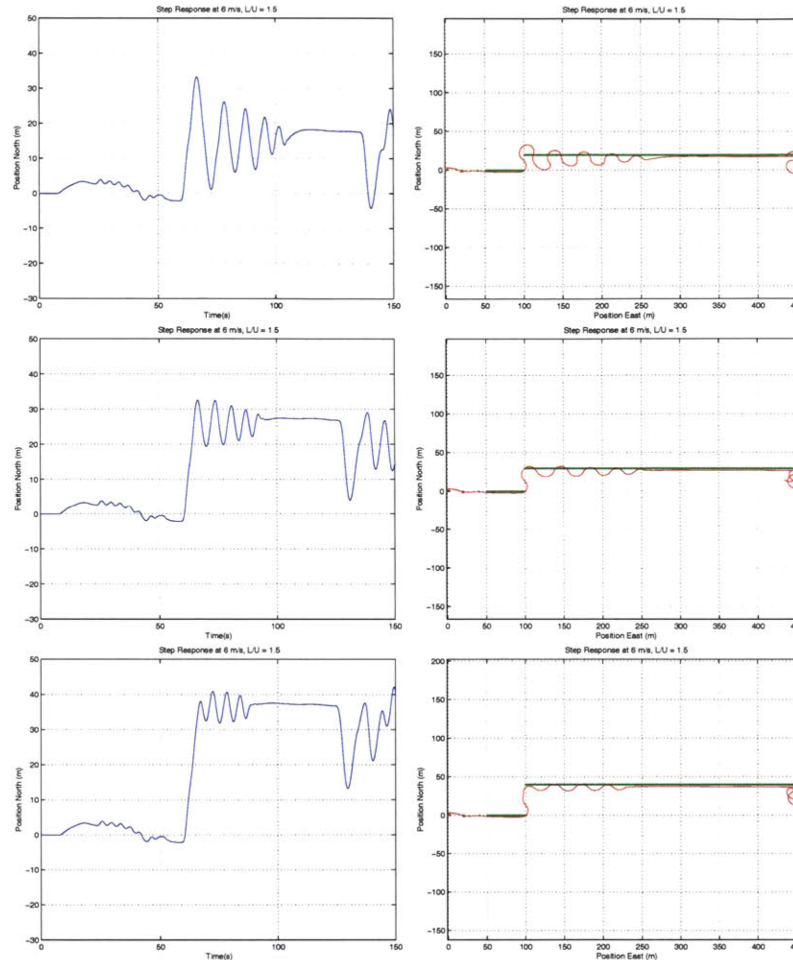


Figure 3-3: Lateral Guidance Step Responses

The helicopter's agility allows it to navigate most urban terrain at 7 to 15 m/s. At these speeds, a waypoint separation of 1 m is sufficiently small to establish a relatively smooth path [1]. Then, while traversing a given trajectory, the path-follower would advance through the waypoints at the rate of 7-16 per second. If each waypoint were associated with a commanded forward speed and altitude, the command bandwidth will be 7-16 Hz, which is higher than the bandwidth of the inner velocity and altitude loops. Currently, no hard limits are imposed on the input frequency. Rather, the path-generating component is expected to provide smooth velocity and altitude commands. It has been implemented and flown on the MIT helicopter (see Chapter 5), and has proven to be a reliable control strategy.

[1]This nominal separation was used for all flight tests and simulations in the rest of this thesis.

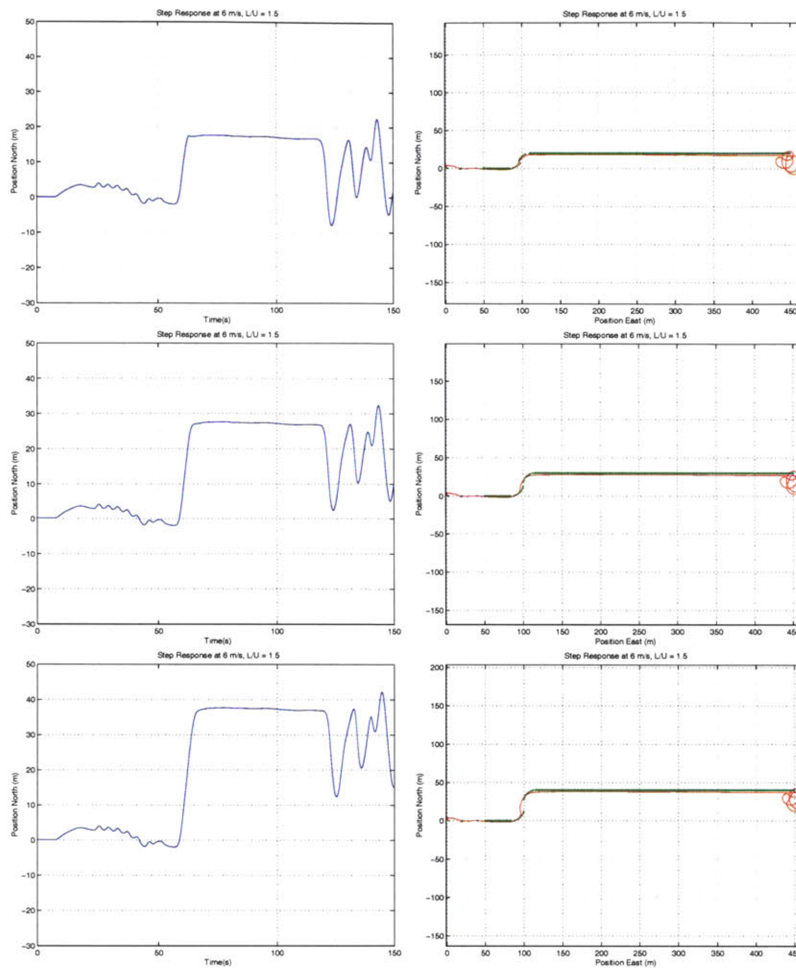


Figure 3-4: Lateral Guidance Step Response with Path Smoothing

Table 3.2 shows the current syntax specification for waypoints. Each waypoint structure is defined by its position (East, North), the desired altitude and forward speed at that position, and an integer that represents a possible action to be taken at that waypoint.

The actions are denoted by integers codes, separated into several classes, as follows. Codes $0-99$ are reserved for aerobatic sequences. These are performed in rate-tracking mode and imply mode-switching. Codes $100-499$ are reserved for loitering. A value of 130 will command loitering for 30 seconds. A value of 100 will command indefinite loitering, until interrupted by another guidance routine, or possibly by a ground operator. Codes $500-999$ command hover, with 500 corresponding to unconditional hover, and 530 causing hover for 30 seconds.

Table 3.2: The waypoint data structure.

Index	Position East	Position North	Forward Speed	Altitude	Action
⋮	⋮	⋮	⋮	⋮	⋮
34	150.3 m	25.56 m	15.12 m/s	25.2 m	0
35	151.3 m	25.56 m	14.62 m/s	24.9 m	0
36	152.3 m	25.56 m	14.12 m/s	24.6 m	0
37	153.3 m	25.56 m	13.62 m/s	24.3 m	3
⋮	⋮	⋮	⋮	⋮	⋮

Table 3.3: Helicopter Action Codes

Action Code	Code Meaning
0	Do Nothing (follow path)
1	Perform an Aileron Roll
2	Perform a Split-S
3	Perform a Hammerhead
4-99	Reserved for other aerobatic sequences
100	Loiter indefinitely (until instructed otherwise)
101-499	Loiter for $(ActionCode - 100)$ seconds
500	Hover indefinitely (until instructed otherwise)
501-999	Hover for $(ActionCode - 500)$ seconds

3.4 Hovering

Hovering allows the vehicle to remain at a particular waypoint, e.g. for surveillance purposes. This is implemented using the velocity controller, as follows. If a waypoint requests hover mode, the path-follower is applied to bring the vehicle in the waypoint's proximity, at which time position hold is requested, and the velocity controller continuously attempts to minimize the position error to the given waypoint.

3.5 Loitering

During surveillance missions, the helicopter is likely to remain near the named area of interest (NAI) for extended periods of time, which makes it an attractive target. Since the vehicle is inherently more vulnerable while stationary, it is desirable to loiter, rather than hover, maintaining some forward speed while remaining in the vicinity of the NAI. It is also desirable that the loiter path not be easily predictable.

Table 3.4: Loiter Radius and Loiter Speed

L/U	Fwd. Speed	Avg. Loiter Radius	L/U	Fwd. Speed	Avg. Loiter Radius	L/U	Fwd. Speed	Avg. Loiter Radius
1.5	2 m/s	4 m	2.5	2 m/s	4 m	3.5	2 m/s	4 m
1.5	3 m/s	5 m	2.5	3 m/s	6 m	3.5	3 m/s	8 m
1.5	4 m/s	10 m	2.5	4 m/s	11 m	3.5	4 m/s	12 m
1.5	5 m/s	12 m	2.5	5 m/s	15 m	3.5	5 m/s	15 m
1.5	6 m/s	15 m	2.5	6 m/s	17 m	3.5	6 m/s	15 m
1.5	7 m/s	18 m	2.5	7 m/s	20 m	3.5	7 m/s	17 m

The properties of lateral guidance make it particularly suitable for an implementation of loitering that adresses all of the above concerns. Namely, if the virtual leader stops advancing, the vehicle will continue moving toward it, at the last commanded speed, and will overshoot the leader position, passing through it, and heading away from it. The guidance law will then turn the vehicle back towards the leader, causing another overshoot, and so on. The turns can be quite aggressive, depending on forward speed and the choice of $\frac{L}{U}$. The direction of the turns is determined by the sign of η immediately after the overshoot, which is very sensitive to small perturbations, making the vehicle's trajectory less predictable. The radius of the turns is determined by the speed of the vehicle. Thus a "loiter radius" may be enforced, restricting the overall excursions from the coordinates of the NAI. Some investigation was conducted on the variation of loiter radius with forward speed, and $\frac{L}{U}$. The results are presented in Table 3.4, and in Appendix B.

3.5.1 Loiter Modes

Two loiter modes are available on the MIT helicopter: timed loiter, and indefinite loiter. Timed loiter mode is triggered by action codes > 100. The vehicle loiters for $(actioncode - 100)$ seconds. The implementation uses a countdown timer, which is initialized at the desired number of seconds, and decremented with each control cycle. The virtual leader does not advance until the timer has reached zero.

Indefinite loiter is exactly that: the helicopter stops following the path and loiters about the last specified waypoint, at the last specified speed, indefinitely. This capability allows the mission to be "paused," by the ground operator should there be need to re-plan, to generate a new path, or to simply wait for further instructions. When the mission can proceed, the indefinite loiter is deactivated and the vehicle continues along the path.

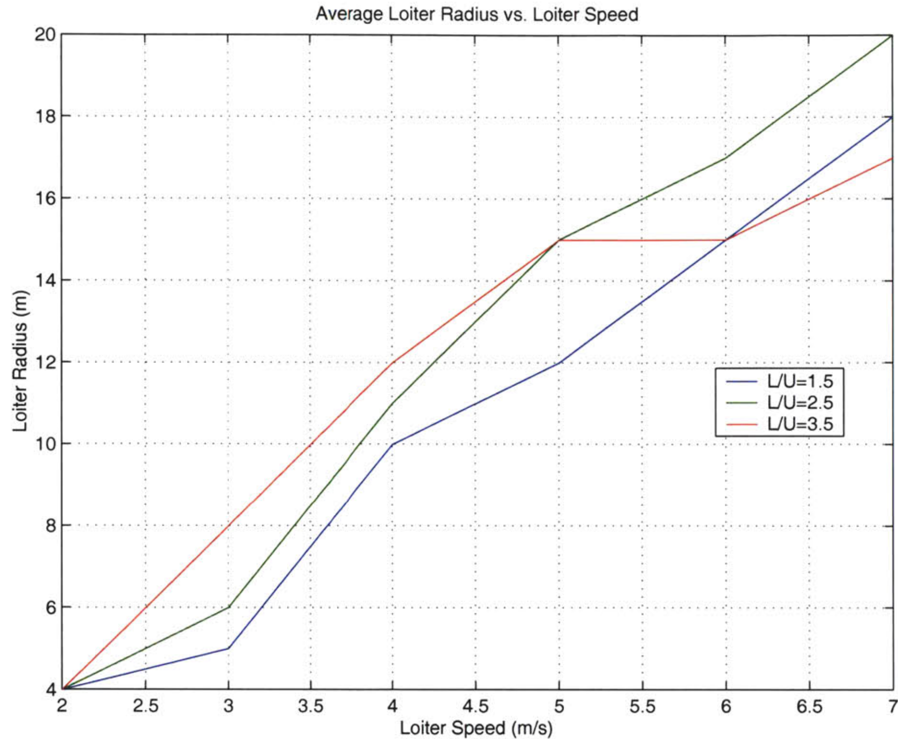


Figure 3-5: Variation of Loiter Radius with Loiter Speed for Several $\frac{L}{U}$.

3.6 Path Management

Since lateral guidance relies on advancing a virtual leader along the trajectory, robust and efficient path management logic is critical for the guidance architecture. The path must extract the mission parameters embedded in each waypoint and pass them to the appropriate controllers. It must also accomodate non-linearities in the trajectory resulting from the aerobatic sequences.

Leader Position

Lateral guidance aims to brings the vehicle closer to a virtual leader. It is the responsibility of the path manager to maintain the desired lead distance, i.e. keep advancing the leader along the trajectory. This is accomplished by exhaustively checking the waypoints lying ahead of the vehicle until the best candidate for a leader is found. Let W be an array of waypoints, representing the path. Let $W(n)$ denote the nth waypoint along the path. Let P denote the current 2D position of the vehicle. Let L denote the desired lead distance for lateral guidance. Then, a possible approach for locating the leader is finding *the closest point along the trajectory which is at a distance greater than or equal to L.* A simple algorithm

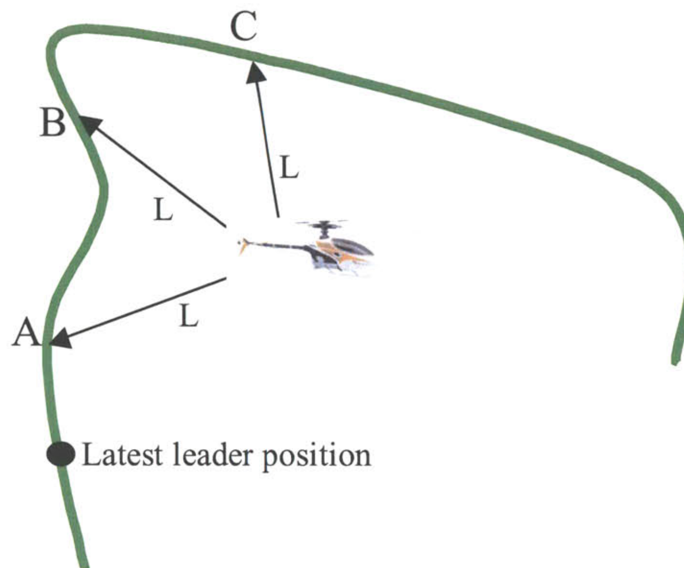


Figure 3-6: Leader Position Ambiguity

for advancing the leader is a single iterative loop:

```
while (distance between P and W(n) < L) { n = n + 1; }
```

If the vehicle is far away from the path, this algorithm will stop advancing the leader and guide the vehicle toward the latest leader position. When the vehicle finds its way within range L of the leader, the algorithm will simply keep moving forward along the path. This approach guarantees a minimum lead distance of L, and prevents overshoot.

Note, however, that the chosen criterion for finding the leader is usually satisfied at two or more points along the path, as shown in Figure 3-6. Our simple algorithm will simply select the first one (point A). But consider the algorithm's behavior when the helicopter executes an aileron roll. The guidance logic hands control over to the rate-tracking loops for the duration of the maneuver, about 5 seconds. When the path follower regains control of the vehicle, it is about 70 meters ahead of the last leader position. This is the situation shown in Figure 3-6 (assuming $L << 70$). The algorithm will force the vehicle to turn back toward the location where the maneuver was initiated. Once within L meters of that location, the leader will finally advance to point A, causing another 180° turn. This behavior is inefficient, and the wide turns may be dangerous in a spatially constrained environment.

It can be avoided by requiring that *the leader is always placed ahead of the vehicle.* This would place the leader at point B, which is clearly a better choice.

The modified algorithm proceeds as follows: the leader advances as before, to the closest point at a distance greater than or equal to L. Then we examine the gradient of the vehicle's distance to this waypoint. If the gradient is negative, i.e. the distance is decreasing, there necessarily exists another waypoint, further along the path, that also satisfies our criterion. We keep advancing until this gradient becomes positive, then reapply the original algorithm. This yields *the first leader waypoint ahead of the vehicle*, in this case, point B. This is a satisfactory solution and the logic will not proceed any further, thus skipping only a small portion of the path, at most $2L$ in length.

The pseudo-code for this implementation is presented below:

```
while (distance between P and W(n) < L)
{
     n = n + 1;
}

if (distance between P and W(n) > distance between P and W(n+1))
{
     while (distance between P and W(n) >= distance between P and W(n+1))
     {
          n = n + 1;
     }

     while (distance between P and W(n) < L)
     {
          n = n + 1;
     }
}
```

A condensed version of the path-management source code is presented in Appendix C.

3.7 Action Queue

As the leader advances along the path, the path follower retrieves the data at the leader waypoint. It applies the lateral guidance law, as previously described, and stores the specified forward speed, the calculated turn rate, and the prescribed altitude in a set of global variables. At the end of the control cycle, the velocity and altitude control loops access those variables and use the given values as their command inputs.

The action codes require a bit more effort to handle properly, as they involve controller switching. Recall that the guidance FSM is the first piece of code to get executed at every control cycle, and it then calls the path-follower, or initiates a maneuver sequence, utilizing the appropriate control loops. The FSM's decision is simply based on the action code.

Namely, a zero action code would cause the vehicle to be controlled by the path-follower, as would an action code of 100 or higher, albeit in loiter/hover mode. A code between 1 and 99 will command a state transition that hands control to the rate-tracking loops for aerobatic maneuvering.

In the most basic implementation, the FSM can simply check the action code at the current leader, along with speed and altitude. However, the path management stategy, as developed in Section 3.6 will occasionally skip some waypoints, omitting the action codes. For this purpose, an *action queue* was implemented. Even if the leader skips some waypoints, the path manager still advances through all of them. When it encounters a non-zero action code, it *pushes* it at the end of a finite queue (currently of size 7). At every control cycle, the guidance FSM checks for a non-zero action code *at the beginning of the queue* and takes the appropriate action, *pulling* that action code from the queue, and shifting the rest of the queue down. When the action code has been carried out to completion, the FSM once again examines the first position and proceeds as before. This also allows for maneuvers to be easily chained together, by simply specifying a maneuver at several consecutive waypoints.

This functionality is sometimes dangerous, as maneuvers and loitering require space and can only be executed in certain areas. Thus, if a maneuver takes the vehicle away from the path, a subsequent maneuver may in fact carry it into a building. For this reason, some extra functionality was added, to ensure that selected action codes only get enqueued when there are no other actions in the queue, and when the vehicle is not skipping portions of the path[2]. This is accomplished by specifying negative action codes. If a negative code is encountered, and the above criteria are met, the absolute value of the code is enqueued. Otherwise, the action code is ignored.

3.8 Advanced Mission Planning and Multi-Vehicle Support

Each waypoint data structure is uniquely identified by its index. This allows a path, and thus an entire mission to be manipulated in real-time, splicing in or clipping out waypoints at desired locations. For instance, in a multi-vehicle setting, portions of the mission can be handed off to different vehicles by simply uploading the appropriate section of the mission path. In the event that a vehicle malfunctions, or is destroyed, its portion of the mission may be spliced into that of the nearest available vehicle. Should the mission parameters change suddenly, an appropriate sequence of waypoints can easily be added or deleted on all vehicles. If an obstacle appears in front of the vehicle, an action code may be inserted at the appropriate waypoint, causing the vehicle to stop, or to execute an evasive maneuver.

In a well-known environment, such as a city grid, with dynamic target evolution, all vehicles may be provided with the full database of waypoints. The ground station would then simply command different arrangements of the waypoints and append various action codes, as necessary. This would conserve bandwidth while providing a flexible, responsive network of UAVs, capable of addressing any mission requirements. It would also allow

[2]which would indicate large external disturbances or emergence from a maneuver

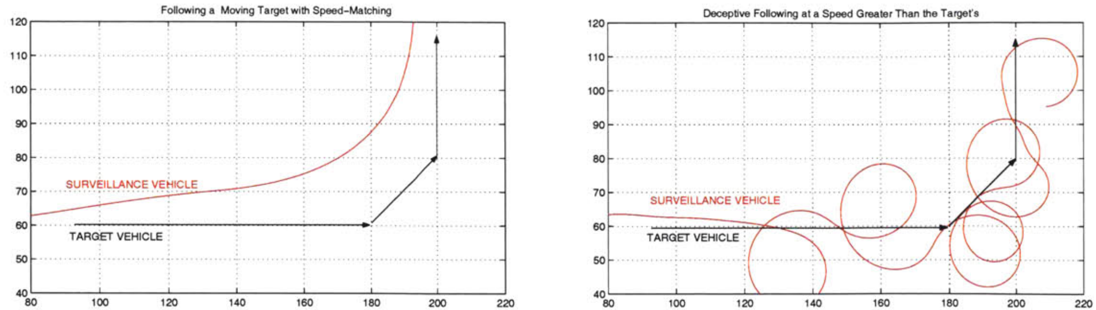


Figure 3-7: Target Following

vehicles to be added or removed to the network at any time. This deployment strategy is particularly well-suited to full-time surveillance of a fixed area.

A locking mechanism is provided for path-manipulation, in order to avoid race conditions in reading/writing waypoints. When the path is being manipulated, the path-follower is suspended and the vehicle simply hovers at its current position, until modifications to the path are complete. Currently, this command has no noticeable effect, since the flight code runs in a single thread, and manipulation of the path will always be completed before the path follower accesses it.

The path is implemented as a finite array of waypoints, currently limited to a maximum of 500 points. However, the array can easily accomodate a much larger path. Since the path follower only moves forward, waypoints behind the leader can be overwritten with new points, and the leader index reset to the first new waypoint. The path-follower needs not be aware of these modifications, as it will simply continue advancing the leader.

3.9 Formation Flight and Moving Target Surveillance

The path management stategy presented above allows for the ready implementation of several valuable features, such as the tracking of a moving target, and basic formation flight.

The following of a moving target is very similar to path following, with the small difference that the leader is no longer virtual. If the target is another UAV in close proximity, the follower will effectively be maintaining a a loose formation with the leader. If we require that the speed of the follower matches that of the leader, we can guarantee their separation. Formations of more than two vehicles can be implemented with a corresponding formation of virtual leaders; in this case, more stringent separation algorithms may be required.

As a proof of concept, a target follower was implemented and tested in simulation. The results are shown in Figure 3-7. The helicopter was commanded to loiter at $(50, 60)$ until

a pop-up target appeared. It then followed the target, matching its own speed with the target's speed.

Interesting results can be observed if the helicopter follows the target at a speed greater than the target's speed. In this case, the loitering behavior is recreated about a moving point, thus producing a trajectory that appears somewhat random, but in fact continues to follow the target in a more covert fashion. This may be especially desirable when flying above rooftop level in an urban environment, as it allows the follower to stay in the vicinity of the target, while remaining largely out of sight. Furthermore, this strategy makes the target harder to identify by external observers.

Chapter 4

Hammerhead Design

4.1 Rate Controllers

AEROBATIC maneuvers are inherently defined by their angular rates, rather than their longitudinal velocities. Thus, they are implemented using the rate-tracking control loops. Direct control over attitude, position, or velocity is generally unnecessary over the short duration of the maneuver, and would result in poor vehicle response. The challenge of maneuver design is determining the necessary rate command trajectories.

4.2 Maneuver Logic

Flight tests conducted at MIT have shown that a human RC pilot, performing aggressive aerobatics, commands rate trajectories that are essentially piecewise linear functions as observed by Gavrilets, Frazzoli, Mettler, Piedmonte, and Feron [20]. Repeatedly, these trajectories have been found to conform to the following general form:

1. The pilot commands a sharp (linear) rate increase until a particular rate is achieved. *(ramp-up stage).*

2. The pilot maintains this rate until the vehicle reaches a certain attitude along the respective axis of rotation *(constant rate stage).*

3. The pilot commands a sharp (linear) drop in rate until the rate returns to zero *(ramp-down stage).*

Figure 4-1 shows an idealized rate trajectory and the effected attitude change. This figure could represent an aileron roll, for instance, although in this case the scale is entirely arbitrary. It becomes apparent that these trapezoidal trajectories are described by several important parameters:

1. Ramp-up time

2. Maximum rate command

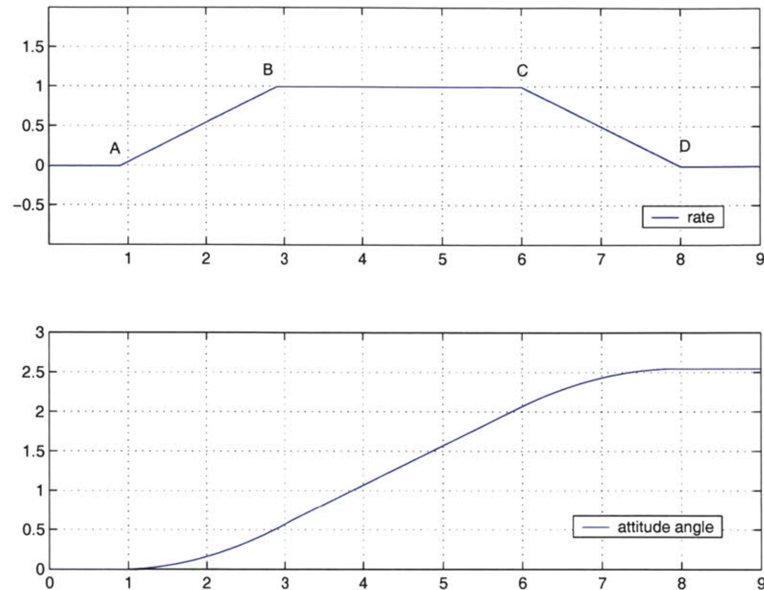


Figure 4-1: Idealized Rate Trajectory and Resulting Attitude Change

3. Attitude threshold for ramp-down
4. Ramp-down time

Thus, the entire maneuver can be conveniently encoded as a finite state machine, where each state controls the appropriate body rate for generating a single leg of a single trapezoid. The rate trajectory from Figure 4-1 may be defined by the following pseudo-code:

```
/* define the trapezoid parameters */
RAMP_UP_TIME = 2;
Q_MAX = 1.0;
THETA_RAMP_DOWN_THRESHOLD = 2.1;
RAMP_DOWN_TIME = 2;
SETTLE_TIME = 0.5;

/* declare variables */
t = 0.0;             /* counts time since the beginning of the maneuver */
dt = 0.02;           /* control cycle frequency */
Q = 0.0;             /* start with zero pitch rate */
theta = 0.0;         /* we'll integrate theta ourselves */
t_entered_state_CD = 0.0;  /* when do start ramping down? */
t_entered_state_SETTLE = 0.0; /* when do we start settling? */
```

```
STATE = AB;         /* begin by generating leg A-B */
loop{
  t = t + dt;
  theta = theta + Q * dt;

  /*******************************************
   * First, take care of the state transitions *
   *******************************************/

  if ( (STATE == AB) and (t > RAMP_UP_TIME) ){
    STATE = BC;     /* Generate Leg B-C */
  }

  if ( (STATE == BC) and (theta > THETA_RAMP_DOWN_THRESHOLD) ){
    STATE = CD;     /* Generate Leg C-D */
    t_entered_state_CD = t;
  }

  if ( (STATE == CD) and ((t - t_entered_state_CD) > RAMP_DOWN_TIME)
    STATE = SETTLE;     /* settle any transients */
    t_entered_settle = t;
  }

  if ( (STATE == SETTLE) and ((t - t_entered_settle) > SETTLE_TIME)
    exit();             /* FINISHED */
  }

  /****************************************
   * Now, specify what to do in each state *
   ****************************************/

  if (STATE == AB){
    Q = Q + dt * Q_MAX / RAMP_UP_TIME;
  }

  if (STATE == BC){
    Q = Q_MAX;
  }

  if (STATE == CD){
    Q = Q_MAX - dt * Q_MAX / RAMP_DOWN_TIME;
  }
```

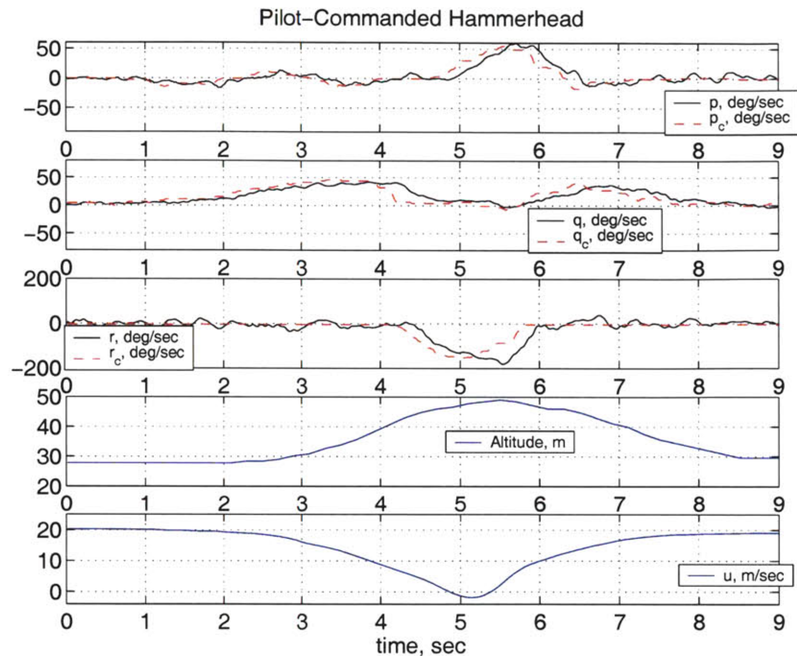


Figure 4-2: Hammerhead: Pilot Inputs and Responses

```
    if (STATE == SETTLE){
      Q = 0;
    }

}
```

Note the settling state at the end of the sequence. This allows any transients to settle, so the helicopter can safely initiate another maneuver, or return to velocity-tracking mode.

4.3 Hammerhead

A hammerhead was designed and flown on the MIT helicopter. In a hammerhead, the rate trajectory consists of several trapezoids, and involves more than one axis. First, a human pilot flew the helicopter by directly controlling the rate control loops. This is not difficult, as the response is similar to actuating the control surfaces directly. The collected data for a sample maneuver is presented in Figure 4-2.

It was observed that pilot commands can be broken down into three strictly sequential phases:

1. a trapezoid in pitch rate, while climbing into the maneuver,

2. a trapezoid in yaw rate, while turning around at the top of the maneuver,
3. a trapezoid in pitch rate, while climbing out of the maneuver.

4.4 Data Collection

The exact parameter values for the trapezoids are arbitrarily determined by the human pilot, but are consistent between instances of the same maneuver. The only hard constraint on the parameters is that the rate trajectory integrates to the desired attitude change about the respective axis, e.g. during a roll, the roll rate trajectory should integrate to 2π. Obviously, this condition is insufficient, as infinitely many trapezoids will satisfy it. A number of qualitative restrictions must also be satisfied in order for a trapezoid to produce an effective, aesthetically pleasing maneuver. For instance, the exact temporal coordination between the various axes is difficult to derive analytically, but is crucial to the overall appearance of the maneuver.

A skilled human pilot intuitively accounts for these subtleties. After some practice, the human's commands generate rates that approach some "best rate trajectory." Thus, the pilot inputs can provide reasonable guesses for the parameters of each trapezoid. The trapezoids can then be hard-coded on-board the helicopter and fed to the rate loops, whenever a maneuver needs to be executed. This produces repeatable maneuvers, as long as the entry conditions are consistent between instances, which can be guaranteed using the velocity control loops. This is the approach currently employed on the MIT helicopter. The pilot's inputs and the corresponding rate responses were recorded during multiple hammerheads. Five of the most consistent and well-behaved maneuvers were analyzed and used for autonomous maneuver design. The pilot's commands were fairly smooth curves that were approximated by trapezoids. The relevant parameters for each trapezoid were estimated by taking into account the maximum rate command, the average slope of the ramp-up and ramp-down phase, and checking that the area of the trapezoid produced the needed change in attitude. The slope estimates were intentionally conservative, so as to eliminate overshoot in the rate responses. The final maneuver parameters were generated by averaging the values from the five selected maneuvers.

Dever [21] has proposed an optimization approach to maneuver design, which uses the vehicle dynamics as constraints for the maneuver parameters, and interpolates in the space of feasible maneuvers to satisfy certain maneuver entry and exit conditions.

Johnson et al. have demonstrated a neural networks approach to maneuver generation [6], [7], directly tracking position and attitude to effect the maneuver trajectory. Their work appears to mainly address the problem of path following.

4.5 Implementation and Validation

The FSM implementation for the hammerhead looks similar to the code presented in the previous section, albeit with twelve states (four states per trapezoid). The actual C code is quite readable and is presented in Appendix D.

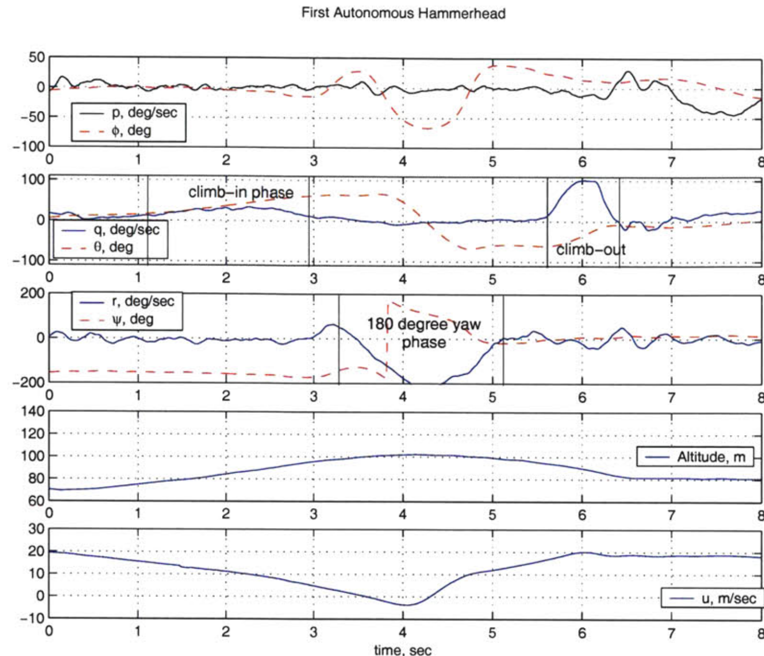


Figure 4-3: Autonomous Hammerhead

For safety reasons, it was decided that the hammerhead exit altitude should be 10-15 meters above the entry altitude. The recorded pilot inputs were modified accordingly to produce a sligtly slower climb into the maneuver, and a faster pull-out.

The code was flown in simulation, and judged to behave satisfactorily. The maneuver was incorporated into a flight mission and flown in the field. The physical responses of the vehicle were recorded and found to be comparable to the pilot's performance[1]. The recorded data is presented in Figure 4-3. The three phases of the maneuver are clearly marked.

This was also the first mission to chain-up two maneuvers together into a single maneuver sequence. A split-s and hammerhead were connected by sequentially executing their respective state machines. The combined maneuver executed perfectly, without any unusual behavior. The success of this experiment proves that maneuver sequences, as implemented on the MIT helicopter, can truly be used as path primitives, in support of innovative path-planning approaches, such as those developed by Frazzoli [10].

[1]In the end, the best validation for the hammerhead design was provided by the pilot's reaction: *I couldn't have done it better myself.*

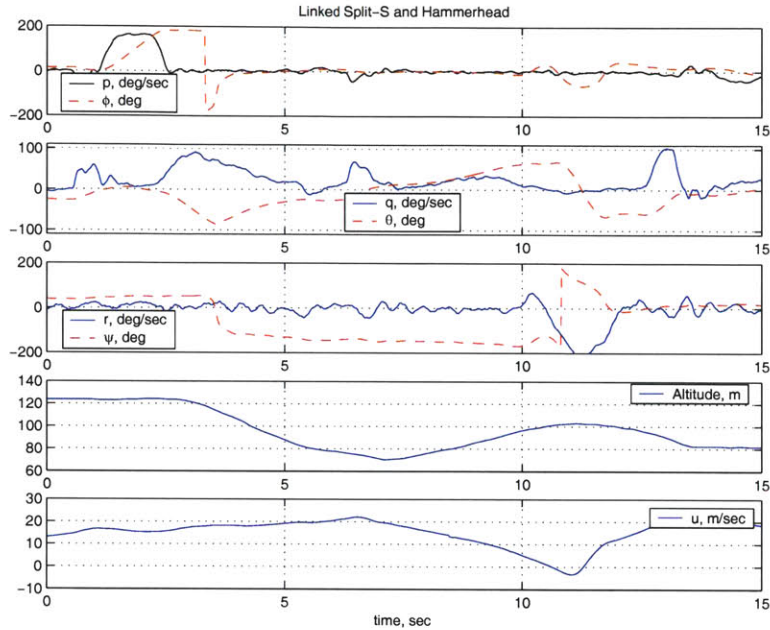


Figure 4-4: Autonomous Split-S linked with Hammerhead

4.6 Hammerhead Implementation on the Yamaha R-Max Helicopter

In the summer of 2003, the author repeated the above design procedure on a Yamaha R-Max helicopter at NASA's Ames Research Center as part of a technology transfer program. A hammerhead was successfully implemented over a three-week period and flown in simulation. Unfortunately, the data is proprietary and cannot be presented in this thesis. However, the technology transfer serves as further proof of the generality and portability of the design method presented above.

Chapter 5

Validation Flight

To demonstrate and validate the developments presented in this thesis, a sample mission was designed and flown in the field. The mission consists of the following phases:

- navigation to several faraway waypoints,
- tracking a 180° smooth turn at 8 m/s,
- tracking a straight line East at 8 m/s,
- tracking a 90° smooth turn at 8 m/s,
- tracking a straight line South at 8 m/s to a loiter at $(120, 0)$,
- tracking a straight line South at 6 m/s,
- tracking a 90° smooth turn at 6 m/s while descending to 35 m,
- tracking a straight line West at 6 m/s,
- tracking a 90° smooth turn while accelerating smoothly from 6 m/s to 14 m/s, in preparation for a maneuver,
- execution of Hammerhead at $(50, -30)$,
- return to a loiter at 3 m/s near $(50, -60)$.

The planned mission path is presented in Figure 5-1. The complete set of waypoints for this mission was generated by a Perl script, provided in Appendix E, should the reader have the desire or patience to examine it.

Figure 5-2 presents the commanded, simulated, and actual recorded trajectories. It is apparent that the path follower is quite reliable. A slight discrepancy between simulation and reality occurs during loiter: in simulation, the helicopter exits the loiter phase near and tangent to the commanded path, while in reality the loiter phase ends with the vehicle moving perpendicular and away from the path. This is entirely possible, due to the random

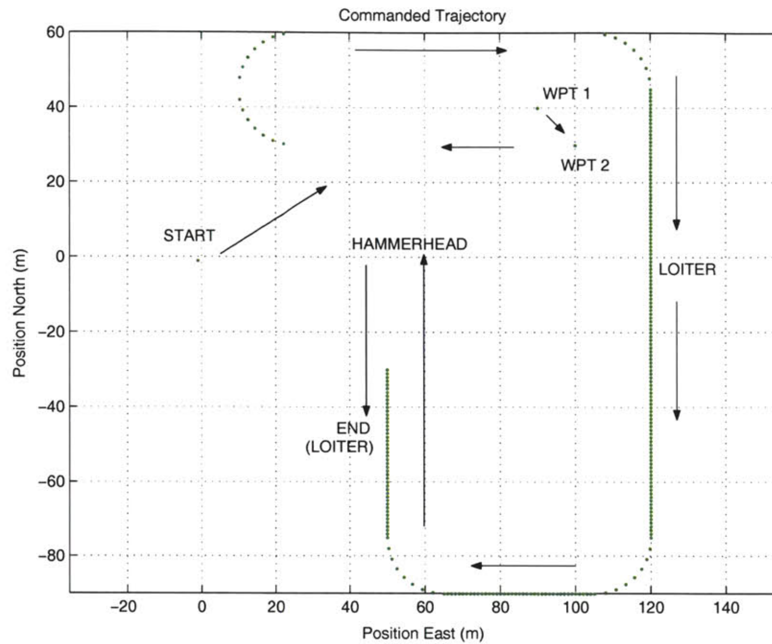


Figure 5-1: Planned Mission Path

nature of the loiter trajectory. It is worth noting that the path follower quickly and smoothly brought the vehicle to the commanded path. Another slight discrepancy is the drift distance during hammerhead. This is due to crosswinds at the field, and is to be expected.

The validation mission was executed three times in total, in order to determine repeatability. The results are presented in Figure 5-3. The path-following portion of the mission was found to be consistent between flights, with the exception of some oscillations during flight 1. These were attributed to non-linear effects, resulting from the loiter exit conditions, and discussed in Section 3.2.1. They can be avoided by increasing the lead distance.

For the curious, an online version of the helicopter simulation has been made available at http://www.formafterfunction.com/. It allows uploading a custom mission file, and outputs visualization data, as well as a MATLAB-readable record of the state trajectory.

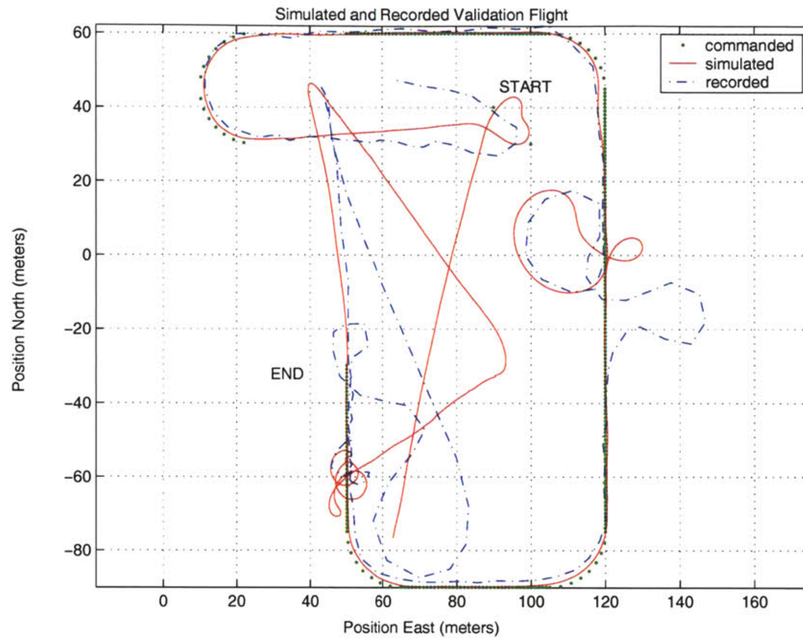


Figure 5-2: Simulated and Recorded Trajectory

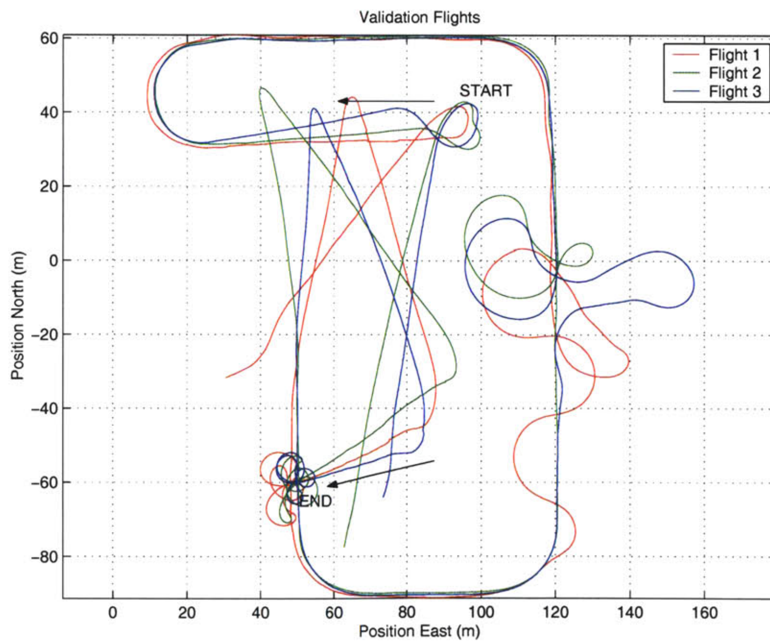


Figure 5-3: Three consecutive executions of validation flight

Chapter 6

Conclusion and Future Work

6.1 Thesis Summary

This thesis has implemented and demonstrated a new guidance architecture for an autonomous aerobatic helicopter. The guidance module is implemented as a finite state machine (FSM), with the unique development that its structure can be dynamically modified in flight. FSM parameters and other mission data are encoded along a commanded path. This data is interpreted by a path-follower, which constructs the mission states on the fly, and executes the specified mission logic. The work presented in this thesis effects a transition from a statically encoded FSM, to a guidance automaton which serves as a syntax interpreter for FSM parameters specified along a commanded path. This makes the mission specification entirely external to the flight code. All mission parameters can be modifed dynamically (e.g. from a ground station) and the syntax for mission specification is readily extendable to address a variety of scenarios. The new architecture results in a more versatile UAV platform, easily adaptable to missions of varying complexity. This work has enabled the implementation of various high-level path-planning algorithms on the MIT helicopter [18], [22].

The path follower used for the guidance implementation was developed at MIT by Sanghyuk Park, as part of his doctoral work. It was selected for its performance in tracking the high-frequency features of urban terrains. The guidance architecture takes advantage of some inherent properties of the algorithm to implement features such as high-speed loitering (Section 3.5) and deceptive target following(Section 3.9).

This thesis has also demonstrated the design of an autonomous hammerhead and has validated it in flight. The design is enabled by the observations [9] that pilot-commanded maneuvers are highly repeatable in their commanded rate trajectories, and that small, hingeless-rotor helicopters behave as rate-command systems in response to cyclic inputs. The high agility of the MIT helicopter results in high controller bandwidth, allowing the tracking of very aggressive trajectories. Thus, pilot commands were recorded and approximated as piecewise-linear rate trajectories, which were input to a set of rate controllers. The resulting hammerhead was no inferior to pilot-executed maneuvers. It was shown to be

repeatable and was chained-up with a previously designed split-s maneuver, thus demonstrating the ability to construct more complex composite maneuver sequences.

6.2 Recommendations for Future Work

As shown in the validation flights, the path-following algorithm exhibits some non-linear behavior, manifested by large (but stable) oscillations. It was suggested in Section 3.3 that these are limit cycles, resulting from a commanded turn rate that exceeds the maximum physical turn rate of the vehicle. This behavior requires further investigation, and analysis of the complete nonlinear system. The oscillations are effectively mitigated by increasing the lead distance (or $\frac{L}{U}$), which is the approach taken in this thesis. A more rigorous approach could involve continuously varying $\frac{L}{U}$ (resulting in a time-varying linearized system), depending on the cross-track error. Such an algorithm would find the maximum safe speed for any trajectory: a task that is currently left to the mission-planning component.

Currently, the raw speed and altitude commands specified along the path are passed directly to the velocity and altitude loops. They are clipped to within reasonable limits but no checks are performed of their frequency. Thus, a path that commands high-frequency inputs can cause loop interactions that drive the system unstable. In the demonstration phase, the path-planner guaranteed smooth controller inputs along the trajectory. A more robust solution would be for the flight software to low-pass the frequencies of the inputs before sending them on to the controllers.

The aerobatic maneuvers are performed in rate-tracking mode, without any position-tracking. As a result, low-frequency disturbances such as wind gusts can cause large deviations from the planned spatial trajectory. Adding a true airspeed sensor can reduce the effect of these disturbances. In addition, the maneuver logic may be modified to take position measurements at critical points of the maneuver (e.g. at the top of a hammerhead), and correct the vehicle's attitude to compensate for the effects of the disturbance. The maneuver logic may further be improved to allow adaptive modification of the defining maneuver parameters (i.e. the trapezoidal rate trajectories from Chapter 4.2, so as to accomodate various maneuver entry and exit conditions. Some efforts in this direction have been undertaken by Dever [21]. Adaptive maneuver augmentation could allow the helicopter to enter and exit a maneuver with variable velocity vectors and at variable altitudes, which would enable the assembly of more complex paths, as proposed by Frazolli [10].

Finally, it must be acknowledged that navigation on the MIT helicopter still relies on GPS reception, which is susceptible to jamming, and may not be available in an urban environment. Alternative methods of position measurement would be required, mostly likely utilizing machine vision algorithms. In particular, terrain-matching approaches seem very applicable to the highly structured city grids.

In conclusion, it is the author's hope that the work presented in this thesis will bring UAV technology closer to a deployment-ready, fully autonomous robot, combining aerial agility with the ability to plan and execute complex mission in a variety of scenarios.

Appendix A

Guidance Law Sensitivity Analysis

The following figures present simulated flight trajectories used to establish the sensitivity of the path-follower to variations in $\frac{L}{U}$ (ratio of lead distance to forward speed). The same mission was performed for different values of $\frac{L}{U}$. The green dots define the commanded path. The maximum cross-track error along the first two legs of the trajectory was measured (i.e. the portion of the path that lies in the Northern half-plane). The commanded speed in this region was 8 m/s. The measured cross-track error was used to populate Table 3.1.

The vehicle was commanded to loiter between the Northern and Southern portions of the mission, then return to the path and continue on South. This enabled qualitative observations of the vehicle behavior (i.e. overshoot), when reacquiring the path. As expected, higher values of $\frac{L}{U}$ resulted in a much smoother path, with less overshoot.

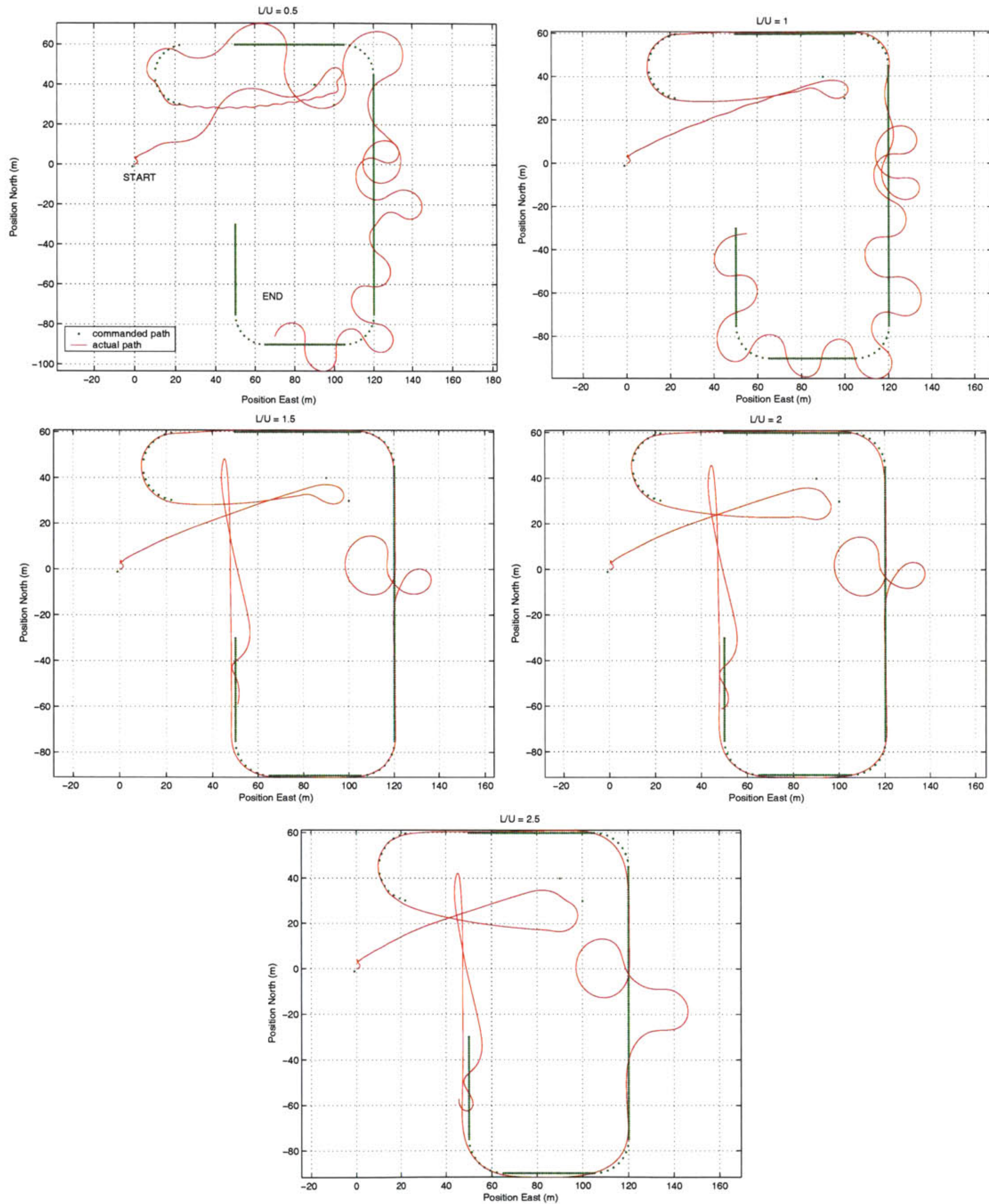


Figure A-1: Guidance Law Performance for $\frac{L}{U} = 0.5..2.5$.

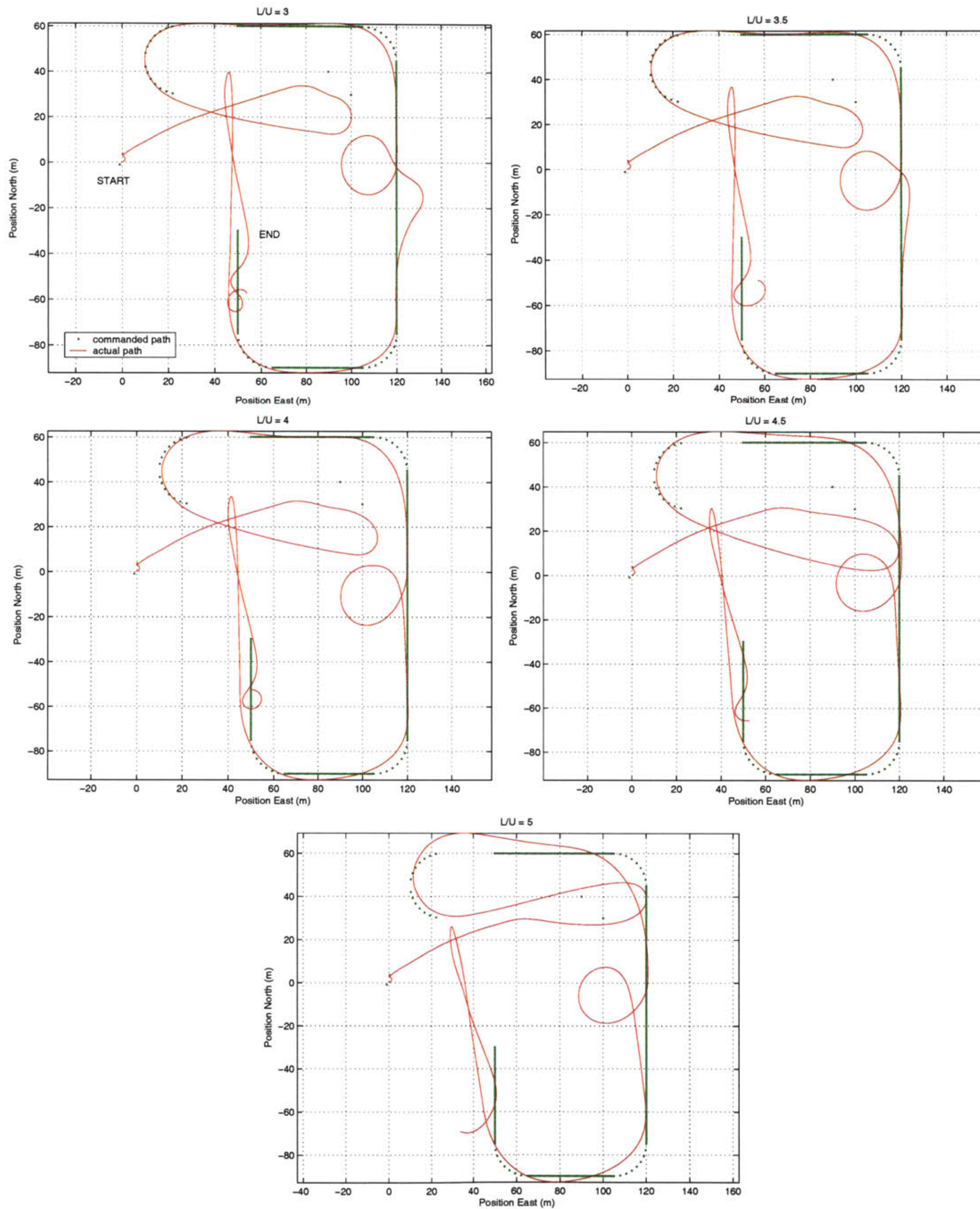


Figure A-2: Guidance Law Performance for $\frac{L}{U} = 3..5$.

Appendix B

Investigation of Loitering Behavior

B.1 Average Loiter Radius at Various Speeds and $\frac{L}{U}$

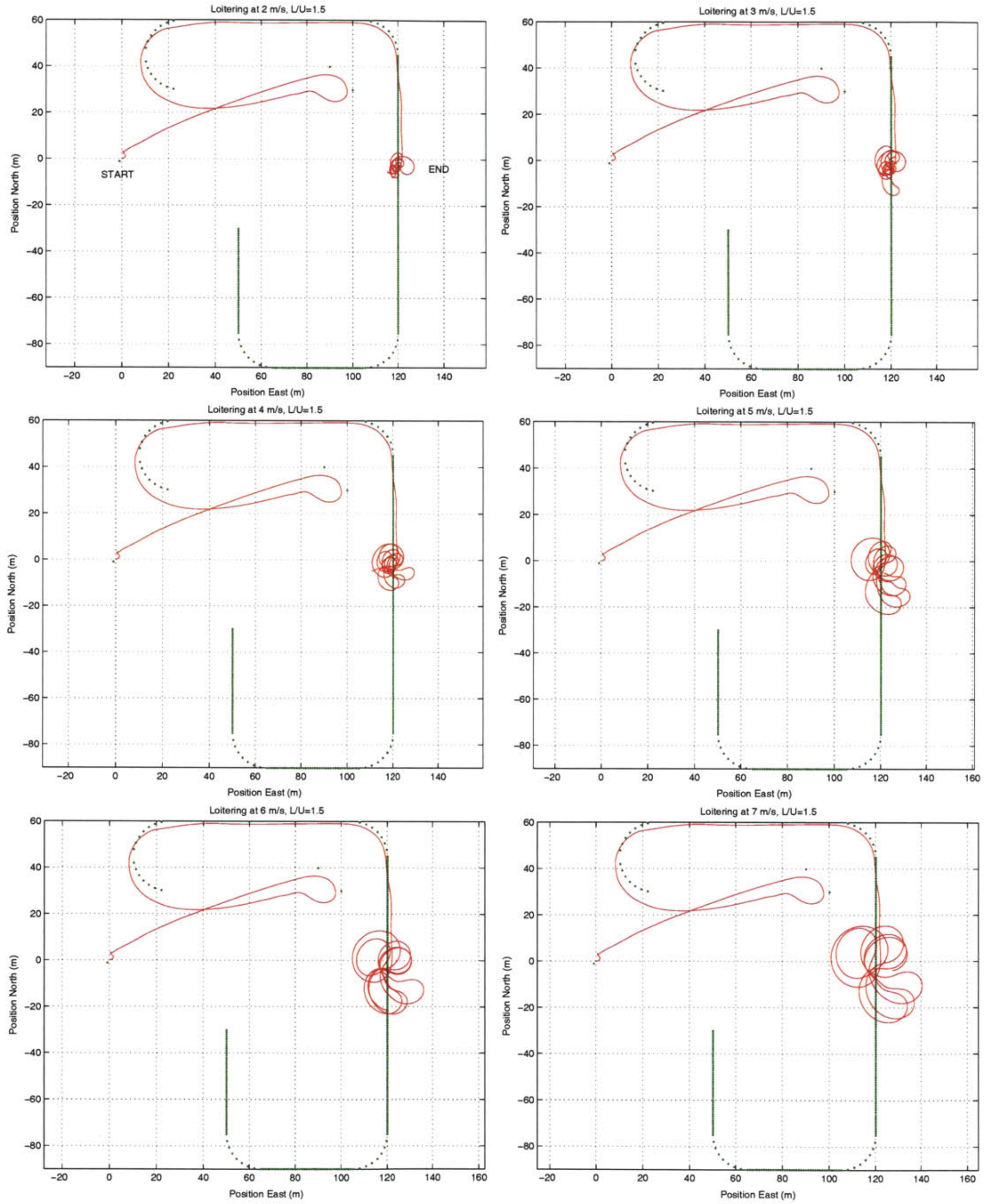


Figure B-1: Loitering Behavior at Varying Speeds for $\frac{L}{U} = 1.5$.

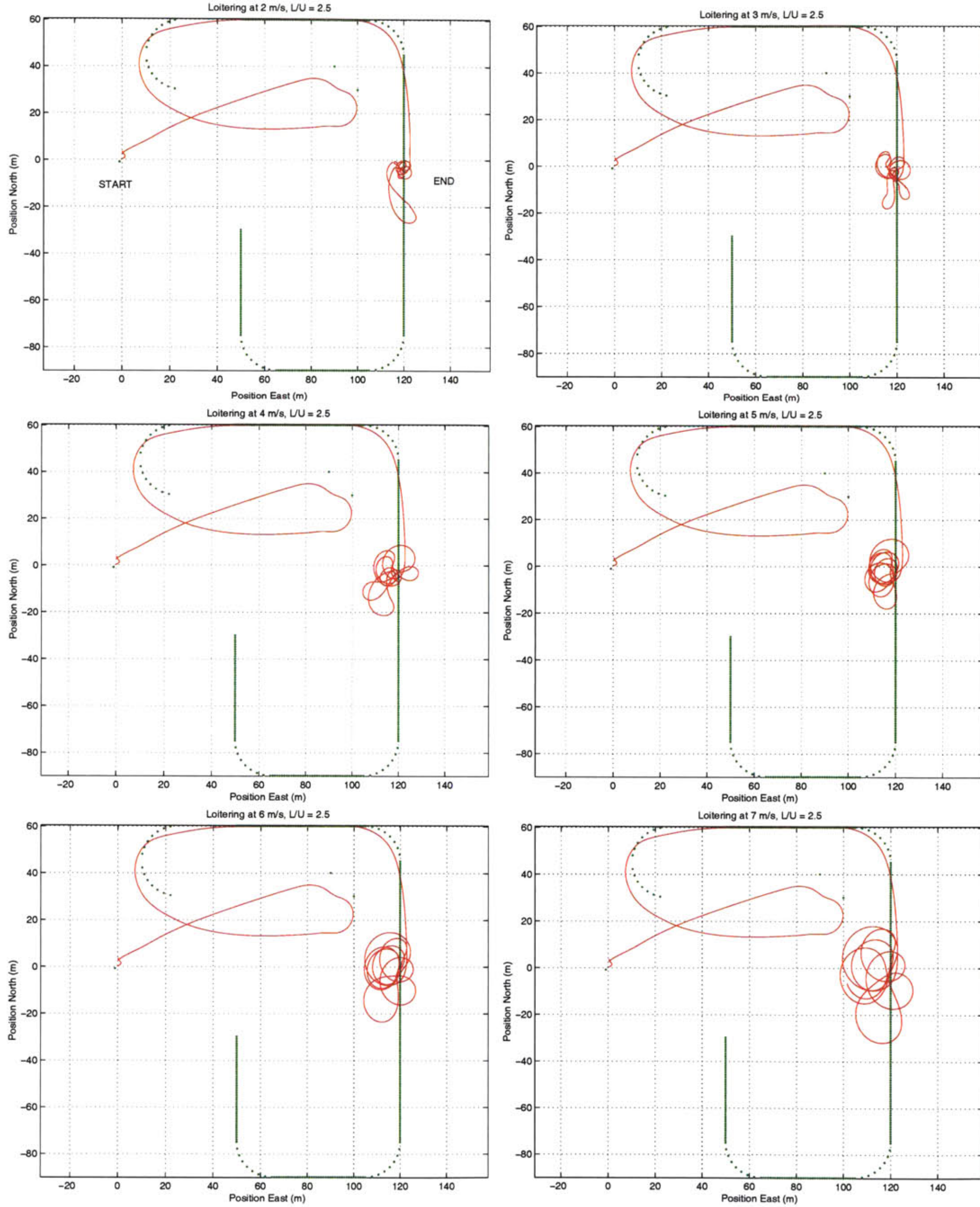


Figure B-2: Loitering Behavior at Varying Speeds for $\frac{L}{U} = 2.5$.

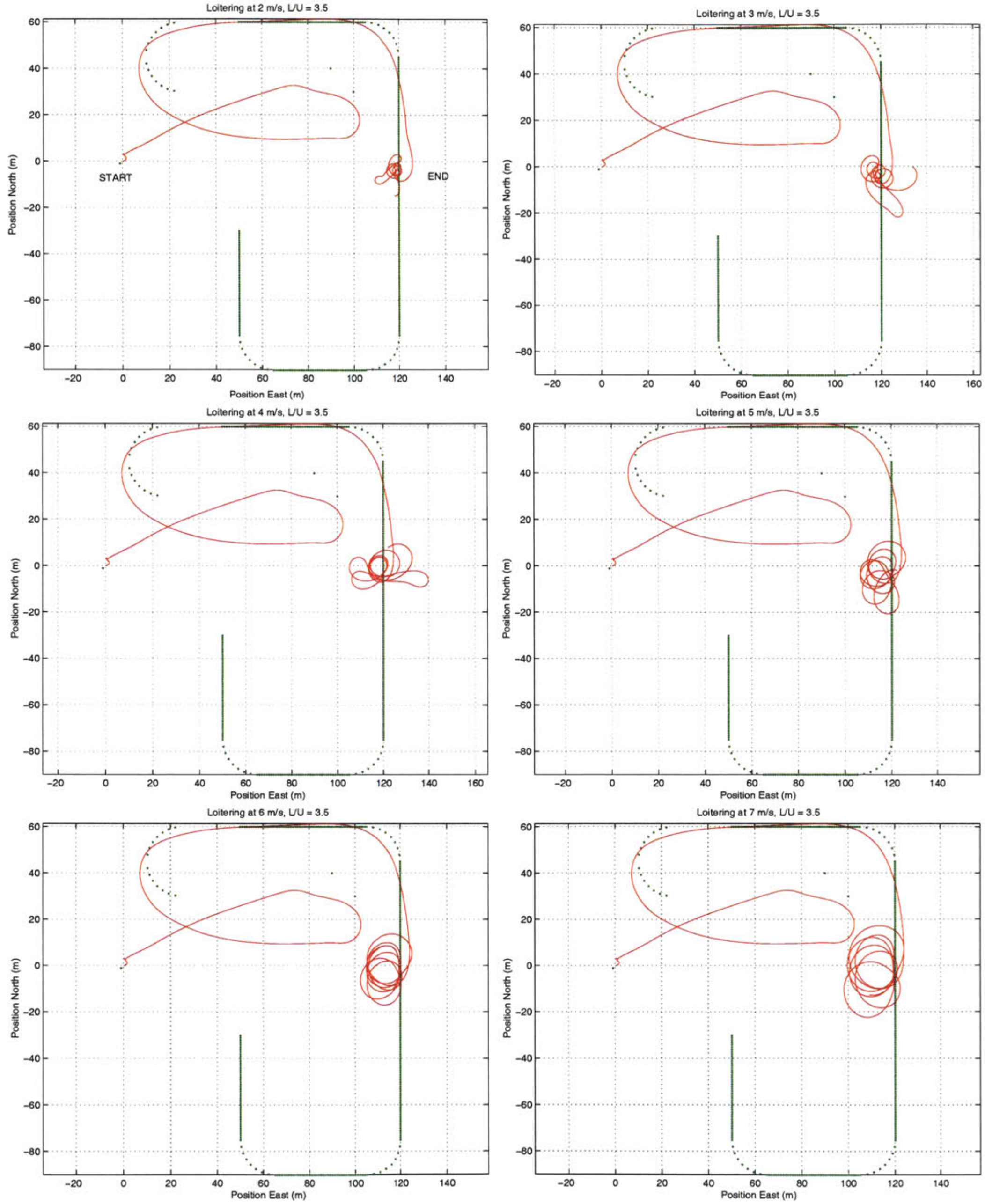


Figure B-3: Loitering Behavior at Varying Speeds for $\frac{L}{U} = 3.5$.

Appendix C

Condensed Path Manager Source Code

```
#define ROUTE_LENGTH 500
#define MANEUVER_STACK_SIZE 7

typedef struct {
5       long id;                /* index of waypoint within array (if applicable) */
        double posN;            /* m, North position relative to the local origin */
        double posE;            /* m, East position relative to the local origin */
        double speed;           /* m/s, forward speed command at waypoint */
        double alt;             /* m, altitude at waypoint */
10      int man;                /* selects a maneuver at this waypoint (0 for none) */
} waypoint_t;

typedef struct {
        waypoint_t data[ROUTE_LENGTH]; /* the array of waypoints */
15      long length;            /* length of the array */
        long cur_idx;           /* current position of leader along path */
        long last_valid_id;     /* last valid ID */
        int ready;              /* path is initialized and ready to go (unused)
                                   this is in case we want to upload/swap paths on the fly. */
20      waypoint_t cur_wpt;     /* waypoint at cur_idx (makes references to it less cumbersome */
        int man[MANEUVER_STACK_SIZE]; /* the queue of maneuvers */
        int man_idx;            /* the last element in the queue.
                                   Note however, that we always stuff out from the front (idx=0) */
        double lu_gain;         /* L/U ratio for the path follower */
25 } route_t;

/* Find the next best guess for a virtual leader along the path */
long get_leader_wpt(route_t * route, waypoint_t cur_pos, double l1)
{
30      /* retrieves a waypoint that is a distance l1 from the current position
           in the forward direction along the route */
        long idx = route->cur_idx; /* the current working index along the path */
        double d1 = 0.0;
```

```
    double ll0 = 0;
35  double ll1 = 0;
    int done = 0;
    int i = 0;

    /* if at the end, stay there */
40  if (idx == route->length - 1) {
        /* If the last action code is a loiter/hover command, use it */
        if (route->data[idx].man >= 100)
            push_man(route, route->data[idx].man, 0);
        route->cur_idx = idx;
45      route->cur_wpt = route->data[idx];
        return idx;
    }

    d1 = l1 - dist(cur_pos, route->data[idx]); /* (specified - actual)distance to leader */
50
    while (d1 > 0) {            /* the point at idx is closer than L1 */
        idx++;                  /* move forward along the path */

        /* if at the end, stay there */
55      if (idx == route->length - 1) {
            /* If the last action code is a loiter/hover command, use it */
            if (route->data[idx].man >= 100)
                push_man(route, route->data[idx].man, 0);
            route->cur_idx = idx;
60          route->cur_wpt = route->data[idx];
            return idx;
        }
        push_man(route, route->data[idx].man, 0);
        d1 = l1 - dist(cur_pos, route->data[idx]); /* compute d1 */
65  }

    if (idx > 0) {
        /* at this point we've found a point that's l1 from us (but could be
           behind us) or we were already at dist > l1 when we started.
70       */

        /*
           Make sure the leader point we just found is the one that's
           furthest forward along the path. Basically looks at the gradient
75         of the helo's distance to consecutive waypoints. If that distance
           is shrinking, the leader we just found is actually behind us. We
           can keep going until we find a leader ahead of us. So we do.
         */

80      /* First we find the closest point to us, i.e. where the gradient
         * changes sign, then we repeat what we did
         * before to go forward to L1*/

        /* Here I switch variables temporarily, but i has the same sense as
85       * idx. I only switch them while I'm looking for the nearest
         * point. */
```

```
        i = idx;
        ll1 = dist(cur_pos, route->data[i]);
        ll0 = dist(cur_pos, route->data[i - 1]);
90      done = (ll1 > ll0) ? 1 : 0;
        while (!done) {
            i++;
            /* if at the end, stay there */
            if (i == route->length - 1) { /* remove this condition to make the array circular */
95              /* If the last action code is a loiter
                 * command, use it */
                if (route->data[i].man >= 100)
                    push_man(route, route->data[i].man, 0);
                route->cur_idx = i;
100             route->cur_wpt = route->data[i];
                return i;
            }

            /* Push the code, but raise the 'skipping' flag */
105         push_man(route, route->data[i].man, 1);
            ll1 = dist(cur_pos, route->data[i]);
            ll0 = dist(cur_pos, route->data[i - 1]);
            done = (ll1 > ll0) ? 1 : 0;
            if (done) {
110             idx = i;
                /* now idx points to closest point on the path. We still want to */
                /* try to go forward if we can. This is the exact code we
                   ran before. */

115             d1 = l1 - dist(cur_pos, route->data[idx]); /* compute d1 */
                while (d1 > 0) {    /* the point at idx is closer than L1 */
                    idx++;     /* move forward along the path */
                    /* if at the end, stay there */
                    if (idx == route->length - 1) {
120                     /* If the last action code is a loiter/hover command, use it */
                        if (route->data[idx].man >= 100)
                            push_man(route, route->data[idx].man, 0);
                        route->cur_idx = idx;
                        route->cur_wpt = route->data[idx];
125                     return idx;
                    }

                    /* Push the code, but raise the 'skipping' flag */
                    push_man(route, route->data[idx].man, 1);
130                 d1 = l1 - dist(cur_pos, route->data[idx]); /* compute d1 */
                }
            }
        }
    }
135
    route->cur_idx = idx;
    route->cur_wpt = route->data[idx];
    return idx;
}
```

```
140
    /* push a maneuver at the back of the queue (if there's room) */
    void push_man(route_t * path, int man, int skipping)
    {
        /* Make sure we have a maneuver and we stay within the allocated size */
145     if ((man == 0) || (path->man_idx >= MANEUVER_STACK_SIZE - 1)) {
            return;
        }
        /* if it's a positive action code, just queue it up */
        if (man > 0) {
150         path->man[path->man_idx] = man;
            path->man_idx++;
        } else {
            /* negative action codes only get enqueued when we're not
             * skipping and when the queue is empty */
155         if (!skipping && (path->man_idx) == 0) {
                path->man[path->man_idx] = -man;
                path->man_idx++;
            }
        }
160 }

    /* pull a maneuver from the front and shift the queue down by 1
     * This returns the maneuver in question, but you might want to just refer to
     * path->man[0] and just pull_man() when you're ready for the next one */
165 int pull_man(route_t * path)
    {
        int i;
        int woman = 0;
        woman = path->man[0]; // retrieve the first maneuver in the queue
170     for (i = 0; i < path->man_idx; i++) { // shift everything down
            path->man[i] = path->man[i + 1];
        }
        path->man[path->man_idx] = 0; // and set what used to be the last maneuver to 0
        if (path->man_idx > 0) // decrement the queue size
175         path->man_idx--;
        return woman;
    }
```

Appendix D

Hammerhead Source Code

```
int
hammerHead(StateEstimate_t * state, UplinkCmd_t * pilotCmd,
        surface_angle_t * sfcCmd, double t)
{
    /* service constants */                                                        5
    static double d_coll_max = 0.00; /* rad, max change in collective command */
    static double d_coll_min = -0.06; /* rad, max -change in collective command */
    static double coll_nom = 0.15; /* rad, nominal collective (entry) */
    static double in_q_max = 0.6; /* rad/sec, maximum pitch rate command */
    static double out_q_max = 2.0; /* rad/sec, maximum pitch rate command */      10
    static double r_min = -3.1; /* rad/sec, maximum yaw rate command */
    static double theta_in_ramp_down = 0.85; /* rad, threshold for in_pitch_ramp_down */
    static double theta_out_ramp_down = -0.6; /* rad, threshold for out_pitch_ramp_down */
    static double psi_yaw_ramp_up = -145 * DEG; /* rad, threshold for yaw_ramp_up */
    static double phi_abort = 20 * DEG; /* rad, max excursion in |roll| before abort */   15
    static double theta_max = 70 * DEG; /* rad, max pitch before kicking over */
    static double u_min = 3; /* m/s, min speed before kicking over */

    /* times in different states */
    static double t_in_pitch_ramp_up = 2.0; /* sec, pitch rate ramp up time */   20
    static double t_in_pitch_ramp_down = 0.6; /* sec, pitch rate ramp down time */
    static double t_u_settle_max = 2.0; /* sec, max wait time for u_settle */
    static double t_yaw_ramp_down = 0.7; /* sec, yaw rate ramp down time */
    static double t_yaw_ramp_up = 0.7; /* sec, yaw rate ramp up time */
    static double t_yaw_settle = 0.1; /* sec, yaw settle time */                  25
    static double t_out_pitch_ramp_up = 0.2; /* sec, pitch rate ramp up time */
    static double t_out_pitch_ramp_down = 0.3; /* sec, pitch rate ramp down */
    static double t_maneuver_time_max = 7; /* sec, max time */
    static double t_coll_ramp_up = 2.0; /* sec, collective ramp up time */
    static double t_all_settle = 0.5; /* sec, settle heli at exit */               30

    /* state definitions */
    static short int just_entered = -1;
    static short int in_pitch_ramp_up = 0;
    static short int in_pitch_const = 1;                                           35
    static short int in_pitch_ramp_down = 2;
```

```
    static short int zero_velocity_settle = 3;
    static short int yaw_ramp_down = 4;
    static short int yaw_const = 5;
40  static short int yaw_ramp_up = 6;
    static short int yaw_settle = 7;
    static short int out_pitch_ramp_up = 8;
    static short int out_pitch_const = 9;
    static short int out_pitch_ramp_down = 10;
45  static short int all_settle = 11;
    static short int exit = 12;

    /* other variables */
    /* These are parameters that need to be reinitialized before the maneuver
50   * and at the end of it (for posterity)*/


    static short int maneuver_state = -1;
    static double phi = 0.0; /* rad, pseudoroll */
55  static double theta = 0.0; /* rad, pseudopitch */
    static double psi = 0.0; /* rad, pseudoyaw */
    static double theta_top = 0.0; /* rad, max theta during climb */
    static double t_in_pitch_ramp_down_begin = 0.0; /* the time when we start ramping down */
    static double t_u_settle_begin = 0.0; /* the time when we begin waiting for u = 0 */
60  static double t_yaw_ramp_down_begin = 0.0; /* when we start kicking over */
    static double t_yaw_ramp_up_begin = 0.0; /* when we start ramping back up */
    static double t_yaw_settle_begin = 0.0; /* when we're done yawing */
    static double t_out_pitch_ramp_up_begin = 0.0; /* when we start pulling out */
    static double t_out_pitch_ramp_down_begin = 0.0; /* when we ramp down on pitch during pullout */
65  static double t_all_settle_begin = 0.0; /* when we enter all_settle state */
    double Zdc = 0.0;

    /* Always command zero roll rate */
    /* Integrate phi (check last line for exit condition on phi */
70  pilotCmd->RollRate = 0.0;
    phi += state->p * EKF_TIME_STEP;

    if (t < EKF_TIME_STEP) { /* Before a single time step has passed */
        maneuver_state = just_entered; /* surely, we must have just entered the maneuver */
75  }

    /* The following are state transitions statements.
     * They are executed only once, at the beginning of each state
     **************/
80
    if ((maneuver_state == just_entered) && (t < t_in_pitch_ramp_up)) {
        maneuver_state = in_pitch_ramp_up;
    }

85
    if ((maneuver_state == in_pitch_ramp_up)
        && ((t >= t_in_pitch_ramp_up) || (state->u <= u_min))) {
        maneuver_state = in_pitch_const;
    }
```

```
90

    if ((maneuver_state == in_pitch_const)
        && ((theta > theta_in_ramp_down) || (state->u <= u_min))) {
        t_in_pitch_ramp_down_begin = t;
95      maneuver_state = in_pitch_ramp_down;
    }

    if ((maneuver_state == in_pitch_ramp_down)
        && ((t > (t_in_pitch_ramp_down_begin + t_in_pitch_ramp_down))
100         || (theta > theta_max) || (state->u <= u_min))) {
        theta_top = theta;
        t_u_settle_begin = t;
        maneuver_state = zero_velocity_settle;
    }
105
    if ((maneuver_state == zero_velocity_settle)
        && ((t > (t_u_settle_begin + t_u_settle_max))
            || (state->u <= u_min))) {
        t_yaw_ramp_down_begin = t;
110     maneuver_state = yaw_ramp_down;
    }

    if ((maneuver_state == yaw_ramp_down)
        && (t > (t_yaw_ramp_down_begin + t_yaw_ramp_down))) {
115     maneuver_state = yaw_const;
    }

    if ((maneuver_state == yaw_const) && (psi < psi_yaw_ramp_up)) {
        t_yaw_ramp_up_begin = t;
120     maneuver_state = yaw_ramp_up;
    }

    if ((maneuver_state == yaw_ramp_up)
        && (t > t_yaw_ramp_up_begin + t_yaw_ramp_up)) {
125     t_yaw_settle_begin = t;
        maneuver_state = yaw_settle;
    }

    if ((maneuver_state == yaw_settle)
130     && (t > t_yaw_settle_begin + t_yaw_settle)) {
        t_out_pitch_ramp_up_begin = t;
        theta = -theta_top;
        maneuver_state = out_pitch_ramp_up;
    }
135
    if ((maneuver_state == out_pitch_ramp_up)
        && (t > t_out_pitch_ramp_up_begin + t_out_pitch_ramp_up)) {
        maneuver_state = out_pitch_const;
    }
140
    if ((maneuver_state == out_pitch_const)
        && (theta > theta_out_ramp_down)) {
```

```
            t_out_pitch_ramp_down_begin = t;
            maneuver_state = out_pitch_ramp_down;
145     }

        if ((maneuver_state == out_pitch_ramp_down)
            && (t > t_out_pitch_ramp_down_begin + t_out_pitch_ramp_down)) {
            t_all_settle_begin = t;
150         maneuver_state = all_settle;
        }

        if ((maneuver_state == all_settle)
            && (t > t_all_settle_begin + t_all_settle)) {
155         maneuver_state = exit;
        }

        /* Things we do in the states */

160     if (maneuver_state == just_entered) {
            pilotCmd->PitchRate = 0.0;
        }

        if (maneuver_state == in_pitch_ramp_up) {
165         pilotCmd->PitchRate = t * (in_q_max / t_in_pitch_ramp_up);
            sfcCmd->coll = coll_nom + t * (d_coll_max / t_in_pitch_ramp_up);
            theta += state->q * EKF_TIME_STEP;
        }
        if (maneuver_state == in_pitch_const) {
170         pilotCmd->PitchRate = in_q_max;
            sfcCmd->coll = coll_nom + d_coll_max;
            theta += state->q * EKF_TIME_STEP;
        }
        if (maneuver_state == in_pitch_ramp_down) {
175         pilotCmd->PitchRate =
                in_q_max + (t_in_pitch_ramp_down_begin -
                            t) * (in_q_max / t_in_pitch_ramp_down);
            sfcCmd->coll =
                coll_nom + d_coll_max + (t_in_pitch_ramp_down_begin -
180                                      t) * ((d_coll_max -
                                               d_coll_min) /
                                              t_in_pitch_ramp_down);
            theta += state->q * EKF_TIME_STEP;
        }
185
        if (maneuver_state == zero_velocity_settle) {
            pilotCmd->YawRate = 0.0;
            pilotCmd->PitchRate = 0.0;
            sfcCmd->coll = coll_nom + d_coll_min;
190     }

        if (maneuver_state == yaw_ramp_down) {
            pilotCmd->YawRate =
                (t - t_yaw_ramp_down_begin) * (r_min / t_yaw_ramp_down);
195         pilotCmd->PitchRate = 0.0;
```

```
        sfcCmd->coll = coll_nom + d_coll_min;
        psi += state->r * EKF_TIME_STEP;
    }

200 if (maneuver_state == yaw_const) {
        pilotCmd->YawRate = r_min;
        pilotCmd->PitchRate = 0.0;
        sfcCmd->coll = coll_nom + d_coll_min;
        psi += state->r * EKF_TIME_STEP;
205 }

    if (maneuver_state == yaw_ramp_up) {
        pilotCmd->YawRate =
            r_min + (t_yaw_ramp_up_begin - t) * (r_min / t_yaw_ramp_up);
210     pilotCmd->PitchRate = 0.0;
        sfcCmd->coll = coll_nom + d_coll_min;
        psi += state->r * EKF_TIME_STEP;
    }

215 if (maneuver_state == yaw_settle) {
        pilotCmd->YawRate = 0.0;
        pilotCmd->PitchRate = 0.0;
        sfcCmd->coll = coll_nom + d_coll_min;
    }
220 if (maneuver_state == out_pitch_ramp_up) {
        pilotCmd->YawRate = 0.0;
        pilotCmd->PitchRate =
            (t -
             t_out_pitch_ramp_up_begin) * (out_q_max /
225                                          t_out_pitch_ramp_up);
        sfcCmd->coll =
            (t <
             t_out_pitch_ramp_up_begin + t_coll_ramp_up ? coll_nom +
             d_coll_min + (t_out_pitch_ramp_up_begin -
230                       t) * (d_coll_min / t_coll_ramp_up) : coll_nom);
        theta += state->q * EKF_TIME_STEP;
    }
    if (maneuver_state == out_pitch_const) {
        pilotCmd->YawRate = 0.0;
235     pilotCmd->PitchRate = out_q_max;
        sfcCmd->coll = (t < t_out_pitch_ramp_up_begin + t_coll_ramp_up ?
                        coll_nom + d_coll_min +
                        (t_out_pitch_ramp_up_begin -
                         t) * (d_coll_min / t_coll_ramp_up) : coll_nom);
240     theta += state->q * EKF_TIME_STEP;
    }

    if (maneuver_state == out_pitch_ramp_down) {
        pilotCmd->YawRate = 0.0;
245     pilotCmd->PitchRate =
            (t_out_pitch_ramp_down_begin -
             t) * (out_q_max / t_out_pitch_ramp_down);
        sfcCmd->coll =
```

```
            (t <
250          t_out_pitch_ramp_up_begin + t_coll_ramp_up ? coll_nom +
             d_coll_min + (t_out_pitch_ramp_up_begin -
                           t) * (d_coll_min / t_coll_ramp_up) : coll_nom);
        theta += state->q * EKF_TIME_STEP;
    }
255
    if (maneuver_state == all_settle) {
        pilotCmd->YawRate = 0.0;
        pilotCmd->PitchRate = 0.0;
        sfcCmd->coll = (t < t_out_pitch_ramp_up_begin + t_coll_ramp_up ?
260                     coll_nom + d_coll_min +
                        (t_out_pitch_ramp_up_begin -
                         t) * (d_coll_min / t_coll_ramp_up) : coll_nom);
    }

265 if (maneuver_state == exit) {
        phi = 0.0;              /* rad, pseudoroll */
        theta = 0.0;            /* rad, pseudopitch */
        psi = 0.0;              /* rad, pseudoyaw */
        theta_top = 0.0;        /* rad, max theta during climb */
270     t_in_pitch_ramp_down_begin = 0.0; /* the time when we start ramping down */
        t_u_settle_begin = 0.0; /* the time when we begin waiting for u = 0 */
        t_yaw_ramp_down_begin = 0.0; /* when we start kicking over */
        t_yaw_ramp_up_begin = 0.0; /* when we start ramping back up */
        t_yaw_settle_begin = 0.0;    /* when we're done yawing */
275     t_out_pitch_ramp_up_begin = 0.0;   /* when we start pulling out */
        t_out_pitch_ramp_down_begin = 0.0; /* when we ramp down on pitch during pullout */
        t_all_settle_begin = 0.0;      /* when we enter all_settle state */
        Zdc = 0.0;
        return 1;
280 }

    pilotCmd->autopilotMode = RATE_TRACKING; /* change mode to rate tracking */
    rate_controller(state, pilotCmd, sfcCmd);
    pilotCmd->autopilotMode = VEL_TRACKING; /* change mode to vel tracking */
285

    if ((fabs(phi) > phi_abort) || (t > t_maneuver_time_max)) {
        maneuver_state = exit;
    }
290
    return 0;
}
```

Appendix E

Scripting the Mission

The waypoints for the validation flight (Chapter 5) were specified in a text file, one per line. The text file was generated using a simple perl script, presented below.

E.1 Script for Generating the Sample Mission

```
#!/usr/bin/perl -w

#################################################################
## Generate a sequence of waypoints, specifying a mission #
#################################################################

## action code at waypoint
$man = 0;
## speed at waypoint
$speed = 8.0;
## altitude at waypoint
$alt = 50.0;
## waypoint id
$j = 0;

## These are smooth turns with a radius of 15 meters:
## ----------
## turnES means you enter the turn headed East and exit headed South

@turnESe = (2.9300, 5.7400, 8.3300, 10.6100, 12.4700, 13.8600, 14.7100);
@turnESn = (-0.2900, -1.1400, -2.5300, -4.3900, -6.6700, -9.2600, -12.0700);
@turnENe = (2.9300, 5.7400, 8.3300, 10.6100, 12.4700, 13.8600, 14.7100);
@turnENn = (0.2900, 1.1400, 2.5300, 4.3900, 6.6700, 9.2600, 12.0700);
@turnWSe = (-2.9300, -5.7400, -8.3300, -10.6100, -12.4700, -13.8600, -14.7100);
@turnWSn = (-0.2900, -1.1400, -2.5300, -4.3900, -6.6700, -9.2600, -12.0700);
@turnWNe = (-2.9300, -5.7400, -8.3300, -10.6100, -12.4700, -13.8600, -14.7100);
@turnWNn = (0.2900, 1.1400, 2.5300, 4.3900, 6.6700, 9.2600, 12.0700);
@turnNEe = reverse @turnWSe;
@turnNEn = reverse @turnWSn;
```

```
@turnSEe = reverse @turnWNe;
@turnSEn = reverse @turnWNn;
@turnSWe = reverse @turnENe;
@turnSWn = reverse @turnENn;
@turnNWe = reverse @turnESe;
@turnNWn = reverse @turnESn;


## The initial solitary waypoints

printf "$j 90.0 40.0 8.0 $alt.0 0\n";
$j++;
printf "$j 100.0 30.0 3.0 $alt.0 0\n";
$j++;


## The rest of the mission is specified in the thesis.

$speed = 5.0;

for ($i=0;$i<7;$i++){
        $x=$turnWNe[$i];
        $y=$turnWNn[$i];
        printf "$j %.2f %.2f $speed.0 $alt $man\n", $x+25, $y+30;
        $j++;
}

for ($i=0;$i<7;$i++){
        $x=$turnNEe[$i];
        $y=$turnNEn[$i];
        printf "$j %.2f %.2f $speed.0 $alt $man\n", $x+25, $y+60;
        $j++;
}

$speed = 8;

for ($i=50;$i<=105;$i++){
        printf "$j $i.0 60.0 %.2f $alt $man\n", $speed;
        $j++;
}

for ($i=0;$i<7;$i++){
        $x=$turnESe[$i];
        $y=$turnESn[$i];
        printf "$j %.2f %.2f $speed.0 $alt $man\n", $x+105, $y+60;
        $j++;
}

for ($i=45;$i>0;$i--){
        printf "$j 120.0 $i.0 %.2f $alt.0 $man\n",$speed;
        $j++;
}
```

```
    $speed = 6;
    printf "$j 120.0 0.0 6.0 $alt.0 115\n";
85  $j++;

    for ($i=0;$i>=-75;$i--){
            printf "$j 120.0 $i.0 %.2f $alt.0 $man\n",$speed;
            $j++;
90  }

    $alt = 40;
    for ($i=0;$i<7;$i++){
            $x=$turnSWe[$i];
95          $y=$turnSWn[$i];
            printf "$j %.2f %.2f $speed.0 $alt $man\n", $x+105, $y-90;
            $j++;
    }

100 for ($i=105;$i>=65;$i--){
            print "$j $i.0 -90.0 $speed.0 $alt.0 $man\n";
            $j++;
    }

105 for ($i=0;$i<7;$i++){
            $x=$turnWNe[$i];
            $y=$turnWNn[$i];
            printf "$j %.2f %.2f $speed.0 $alt $man\n", $x+65, $y-90;
            $j++;
110 }

    for ($i=-75;$i<-30;$i++){
            printf "$j 50.0 $i.0 %.2f $alt.0 $man\n",$speed;
            $j++;
115 }

    printf "$j 50.0 -30.0 %.2f $alt.0 3\n",$speed;
    $j++;

120 printf "$j 50.0 -60.0 3.0 $alt.0 0\n";
    $j++;

    print "-1 -1 -1 -100 -1337 0";
```

Bibliography

[1] S. Park, "Integration of parent-child unmanned air vehicle focusing on control system development," Master's thesis, Massachusetts Institute of Technology, May 2001.

[2] The Federation of American Scientists, *Predator Specifications.* http://www.fas.org/irp/program/collect/predator.htm.

[3] The Federation of American Scientists, *UAV Ground Control Station.* http://www.fas.org/irp/program/collect/uav_gcs.htm.

[4] Aerovironment, Inc, *Pointer FQM-151A Unmanned Aerial Vehicle (UAV) System.* http://www.aerovironment.com/area-aircraft/prod-serv/ptrdes.pdf.

[5] M. La Civita, G. Papageorgiou, W. C. Messner, and T. Kanade, "Design and flight testing of a high-bandwidth $\mathcal{H}_\infty$ loop shaping controller for a robotic helicopter," in *Proceedings of the AIAA Guidance, Navigation, and Control Conference*, (Monterey, CA), August 2002.

[6] E. Johnson, A. Calise, and E. Corban, "A six-degree-of-freedom adaptive flight control architecture for trajectory following," in *AIAA Guidance, Navigation, and Control Conference and Exhibit*, pp. 500–507, August 2002.

[7] E. Johnson and S. Kannan, "Adaptive trajectory control for autonomous helicopters," in *AIAA Journal of Guidance, Control and Dynamics*, 2004. Accepted for publication.

[8] M. Niculescu, "Lateral track control law for aerosonde uav," in *39th AIAA Aerospace Sciences Meeting and Exhibit*, 2001.

[9] V. Gavrilets, *Autonomous Aerobatic Maneuvering of Miniature Helicopters.* PhD thesis, Massachusetts Institute of Technology, May 2003.

[10] E. Frazzoli, *Robust Hybrid Control for Autonomous Vehicle Motion Planning.* PhD thesis, Massachusetts Institute of Technology, May 2001.

[11] Miniature Aircraft, USA, Orlando, FL, *X-Cell .60 graphite SE Helicopter Kit Instruction Manual*, 1999.

[12] Inertial Science, Newbury Park, CA, *Inertial Measurement Unit ISIS-IMU*, 2000.

[13] F. Kolnick, *QNX 4 Real-Time Operating System.* Basis Computer Systems, 1998.

[14] MaxStream, Inc., Lindon, UT, *XStream OEM RF Module*, 2002.

[15] Nascent Technologies, *Internal Report*, 2003.

[16] F. Adler, "Missile guidance by three-dimensional proportional navigation," *Journal of Applied Physics*, vol. 27, pp. 500–507, May 1956.

[17] P. Zarchan, *Tactical and Strategic Missile Guidance*, vol. 124 of *Progress in Aeronautics and Astronautics.* Washington, DC: AIAA, 1990.

[18] T. Schouwenaars, E. Feron, and J. How, "Safe receding horizon path planning for autonomous vehicles," in *40th Allerton Conference on Communication, Control, and Computing*, 2002.

[19] S. Park, J. Deyst, and J. How, "A new nonlinear guidance logic for trajectory tracking," in *Proceedings of the AIAA Guidance, Navigation, and Control Conference*, 2004.

[20] V. Gavrilets, E. Frazolli, B. Mettler, M. Piedmonte, and E. Feron, "Aggressive maneuvering of small helicopters: a human-centered approach," in *International Journal of Robotics Research*, 2001.

[21] C. Dever, *Parametrized Maneuvers for Autonomous Vehicles.* PhD thesis, Massachusetts Institute of Technology, September 2004.

[22] P. Root and F. Joseph, *Internal Report.* Laboratory for Information and Decision Systems, MIT, 2004.

3499-34