Applied Fast Maneuvering Using a Hybrid Controller

by

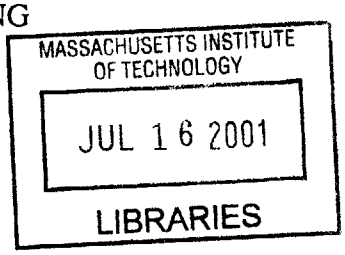
Byron Stanley

S.B. Mechanical Engineering
Massachusetts Institute of Technology, 1999

SUBMITTED TO THE DEPARTMENT OF MECHANICAL ENGINEERING IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE IN MECHANICAL ENGINEERING
AT THE
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

JUNE 2001



Signature of Author_____
Department of Mechanical Engineering
May 11, 2001

Certified by_____
Dr. Marc McConley
Charles Stark Draper Laboratory
Thesis Supervisor

Certified by_____
Jean-Jacques Slotine
Professor of Mechanical Engineering
Thesis Advisor

Accepted by_____
Ain A. Sonin
Professor of Mechanical Engineering
Chairman, Graduate Thesis Committee

Applied Fast Maneuvering Using a Hybrid Controller

by

Byron Stanley

Submitted to the Department of Mechanical Engineering on
May 8, 2001 in partial fulfillment of the requirements for the
Degree of Master of Science in Mechanical Engineering

Abstract

Guidance and control of autonomous vehicles is a difficult and often calculation-intensive process. Current control approaches limit the functionality of autonomous vehicles. The approach applied in this thesis is to use a discrete-state model of the vehicle dynamics to build a hybrid automaton. By creating a set of stable trim states and building a library of maneuvers, the on-line optimization problem was made significantly simpler. The implementation of the control methodology for an autonomous helicopter was expanded to include three-dimensional path planning and pilot-inspired maneuvers. The hybrid controller was then adapted and applied to an unpowered parafoil in simulation. The parafoil hybrid controller was also implemented on a DSP chip and used in a real-time DSP-processor-based simulation of the system. This demonstrated the ability to apply the hybrid controller logic to a variety of vehicles.

Technical Supervisor: Dr. Marc McConley
Title: Senior Member of the Technical Staff

Thesis Advisor: Professor Jean-Jacques Slotine
Title: Professor of Mechanical Engineering

Acknowledgments

This thesis is the result of not only my effort, but the help of many friends and associates. I would like to start by thanking my advisor at Draper, Dr. Marc McConley for his tireless help and guidance throughout my years as a Draper Fellow. I would also like to thank Professor Slotine for not only agreeing to read and assist with my thesis, but also for his help throughout my years at MIT as my advisor. Many thanks to Rainuka Gupta for keeping me sane and seeing sunlight, even in the darkest hours, as well as her help proofreading the document. Thanks also to Carl Dietrich, who spent a significant amount of time helping me understand the parafoil model, as well as making sure I made it out every so often.

I would also like to thank several other members of the Draper Staff: Bill Kreamer for his help with the parafoil simulation and C coding; Scott Rasmussen for his help with microcontrollers and the navigation system; Brent Appleby for making the project possible in the first place and providing guidance and help when needed. I'd also like to thank my parents, without whose help guidance and assistance, none of this could have happened.

This thesis was prepared at The Charles Stark Draper Laboratory, Inc., under Draper Internal Research and Development Project Number 927.

Byron Stanley
May 11, 2001

[This Page Intentionally Left Blank]

Table of Contents

Table of Figures

Table of Tables

Chapter 1: Introduction

1.1 Motivation

Autonomous vehicles are useful in a wide range of applications from commercial surveying and communications to military data collection. Autonomous controllers and path-planning algorithms could be used in commercial jets to avoid collision or help land the aircraft. Autonomous aerial vehicles have been in high demand in military applications ranging from surveillance of targets to deployment of supplies to troops. Without reasonable reconnaissance information about a hostile area, it is dangerous to send in troops. With the addition of remotely controlled vehicles, images and information can be collected before sending in troops, without risking lives. Additionally, supplies can be delivered through hostile areas to a homing beacon on mobile troops without human interaction or guidance. Autonomous vehicles ideally require no input from the troops to perform their missions, which allows for more mobility and flexibility in situations where the ability to change locations quickly is critical.

Autonomous vehicles have the potential to be extremely useful. These vehicles are often in hostile environments and need the ability to survive and quickly navigate towards their target. Controlling an autonomous aerial vehicle is a difficult problem, which takes significant processing power. Additional objectives, such as optimization of time spent in visible flight or fuel consumption, require considerably more processing

power. Since current processing capabilities limit the performance of autonomous control, novel approaches to the autonomous control problem are required.

1.2 Overview of Investigation

In this section, the overall project and scope will be described. The project had two main phases: the first phase involved adding functionality to already existing hybrid control logic in the XCell-60 model helicopters; the second phase involved adapting the hybrid controller to a completely new platform, the unpowered parafoil.

1.2.1 Overview of the References

The general approach taken to the problem and the literature referenced is discussed in this section. References [1-4,6-9] present the background and application of the hybrid control architecture to autonomous helicopters. References [10-18] define and describe the dynamics and modeling of various parachutes. In [1], a concise overview of the hybrid controller for obstacle avoidance for an XCell-60 helicopter is presented. Reference [2] is a book on dynamic programming and optimal controls that explains the theory and application of optimal control and shortest path selection. Both [3] and [4] present a good overview of hybrid controls, while [4] is a more in-depth overview of the mathematics behind the hybrid automaton. In [5], the results of continuing research on parafoil homing techniques and the applicable mathematics are described. This reference also describes parafoil modeling and controls.

Reference [6] derives and explains the mathematics behind the proposed robust hybrid automaton and the mathematical proof of the system stability. An overview of the hybrid control theory and more detail on obstacle avoidance is shown in [7]. Reference

[8] describes the application of human-based maneuvers to autonomous vehicles. The inner-loop backstepping controller for the helicopter is described in [9].

The use of computer control to deliver parachutes to precise targets is described in detail in [10]. A model of a parafoil with respect to the control inputs from the toggles is presented in [11]. In [12,13], the simulation of parachute guidance and control are discussed. References [14,15,16] describe in detail several models of the parachute and parafoil dynamics.

Reports [17] and [18] are technical memos from NASA that describe the development of landing systems for spacecraft using autonomous controls. Books [19,20,21] describe in more detail the design and implementation of low-level controllers. In [19], nonlinear systems and controllers are described and designed. Reference [20] describes the design of traditional linear controllers through frequency and root-locus methods. Reference [21] describes linear controllers designed through more modern state-space methods. Reference [22] covers a high-fidelity model of an underwater autonomous vehicle and was very helpful in determining the appropriate model for the parafoil.

1.2.2 Overview of the Project and Scope

The first step in this research was to devise a method of reducing the computing power required to guide a vehicle from one location to another. In addition, the problem of completing a secondary objective, obstacle avoidance, was considered for the helicopter platform. Once a suitable guidance methodology had been found, the XCell-60 stunt helicopter, was selected for implementation. For more information on this part

of the project, refer to [1,3,4,6,7]. The vehicle dynamics and controller were simulated, along with the full guidance code in two-dimensional flight. This was later extended to three-dimensional flight as part of the thesis. Additional study, included in the thesis project, was done in extending the available maneuvers to remote-control pilot-inspired maneuvers. For more information on the initial studies in pilot-inspired maneuvers, see [8].

The main addition proposed and tested in this thesis is the application of the hybrid automaton to a second vehicle. The second implementation of the algorithm was required to show the adaptability of the algorithm to multiple platforms. The second platform chosen was an autonomous parafoil. A six degree-of-freedom model of the parafoil was developed as part of the testing. The code from the autonomous helicopter guidance was adapted to the parafoil model. A controller board, for a hardware implementation, was selected. The algorithm was adapted to the microcontroller and implemented in a simulation of the entire system and environment. A basic description of the scope of the thesis can be seen in Table 1.

	Helicopter Platform	Parafoil Platform
Dynamic Model	Previously Implemented	Included in Research
Low-Level Controller	Previously Implemented	Included in Research
Visual Simulation	Previously Implemented	Included in Research
Two-Dimensional Guidance	Previously Implemented	Not Applicable
Three-Dimensional Guidance	Included in Research	Included in Research
Pilot-Inspired Maneuvers	Included in Research	Not Implemented
Digital Signal Processor	Not Applicable	Included in Research

Table 1: Overview of Project Scope

The scope of this thesis, as seen in Table 1, includes adaptation of the guidance code from two dimensions to three dimensions, adaptation to the parafoil, the dynamic model

of the parafoil, selection and implementation of the microcontroller board, and simulation of the parafoil model.

1.3 Roadmap of report

In Chapter 1, the problem and an overview of the thesis are introduced and discussed. Chapter 2 covers hybrid control and the full background of the thesis. The background includes the theory behind hybrid control and the basic mathematics. Chapter 3 includes the explanation of the helicopter guidance code in two and three dimensions. Chapter 3 also discusses the addition of pilot-inspired maneuvers to the helicopter maneuver library. Chapter 4 is an explanation of the dynamics of the parafoil and the adaptation of the guidance code to the parafoil for simulation. Chapter 5 describes the selection and coding for the microcontroller path-planning board. Chapter 6 presents the results from the simulations. Chapter 7 discusses the results and draws conclusions from the data.

Chapter 2: Hybrid Control Theory

Hybrid control is introduced as a solution to the complex problem of controlling and guiding a highly nonlinear, large-dimensional, high-bandwidth system [4]. The approach described in [4] solves the problems typically encountered while applying online optimization to autonomous flight control systems. These problems include two main items: the extreme computational complexity of the real-time implementation and the inability of the autonomous controller to perform close to even a pilot-controlled response, much less near the boundary of optimal performance. Figure 1 shows the gaps

in the limitations between human controllers, typical autonomous controllers, and he ability of the system to perform.

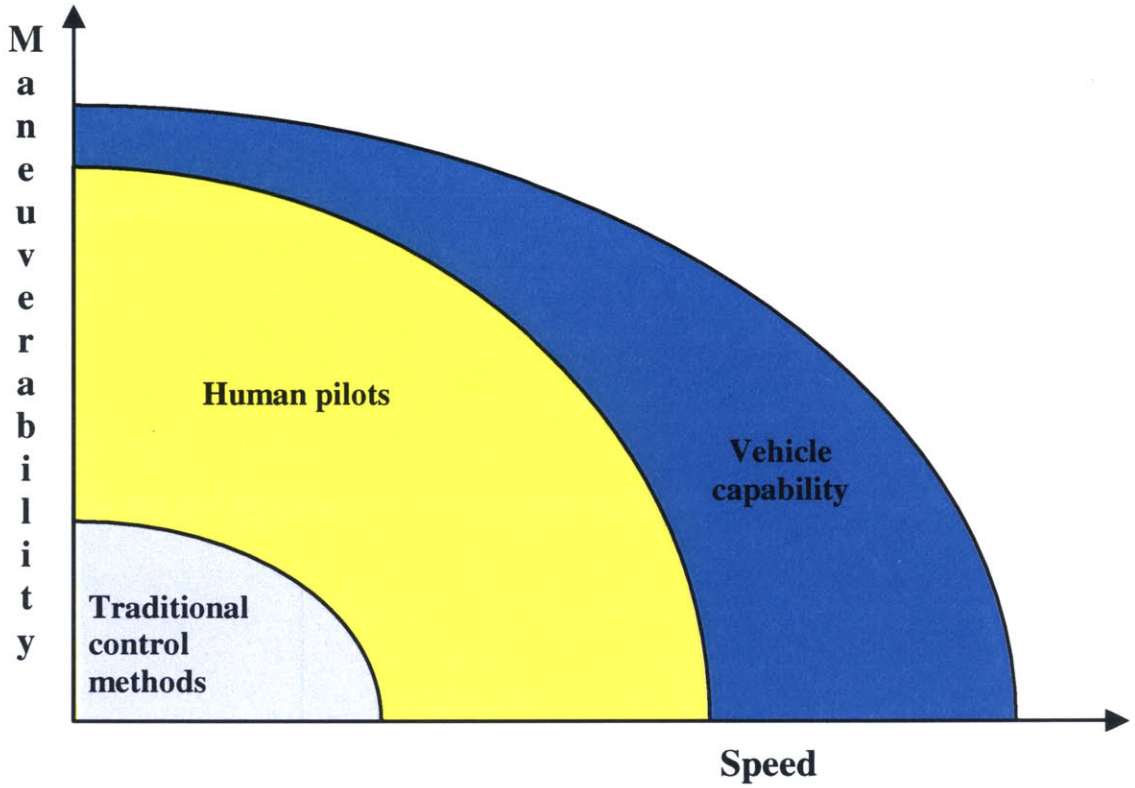


Figure 1: Diagram of Typical Controllers and Their Limitations [1]

It is obvious that there is significant room to improve the current autonomous controllers.

Section 2.1 describes the typical approaches to control of autonomous systems. The typical approaches include gridding the entire state-space and simplifying the system to extremes. The disadvantages of each of these approaches are discussed. Section 2.2 describes the suggested approach using a hybrid automaton to reduce the computation complexity of the problem. The main theoretical approach is the subject of the section. It also presents an overview of the mathematics behind the robust hybrid automaton and

why it is a stable and simpler system. Section 2.3 describes the breakdown of the robust hybrid automaton into offline calculations done before running the controller and the real-time calculations done while the vehicle is in motion.

2.1 Typical Approaches to Autonomous Control

Autonomous controllers can be used not only to control the system directly, but also to control higher order guidance and path-planning algorithms. Both applications will be discussed, with a focus on the guidance and path-planning side. The hybrid controller is a guidance system and is designed to be implemented along with a stable low-level control system, as is shown in Figure 2.

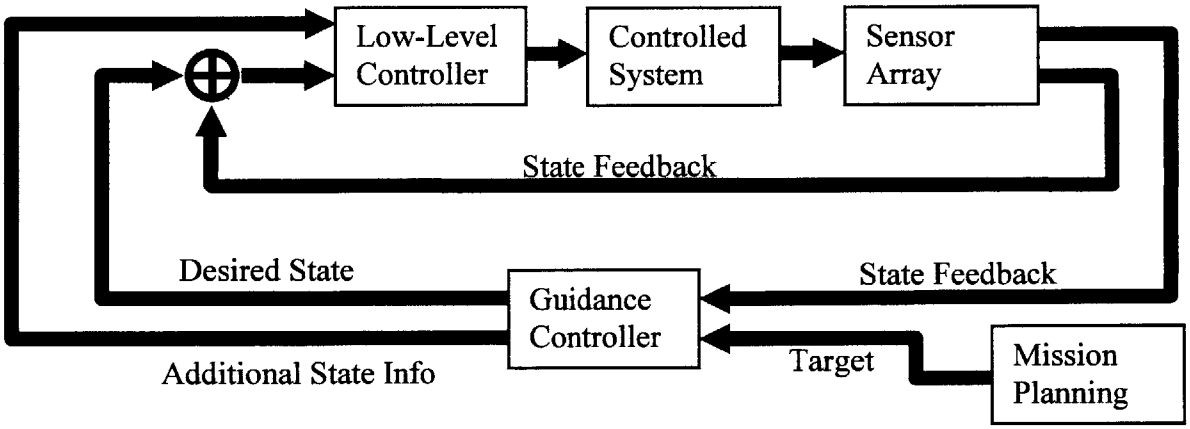


Figure 2: Block Diagram of an upper and lower-level control system.

The low-level control can be implemented using linear and non-linear controllers to track trajectories and inputs. Essentially, the basic flight surfaces of the vehicle are controlled such that the vehicle responds to the desired inputs. The lower level controller will determine how effectively the vehicle can follow the optimized path determined by

the guidance controller. It is an integral part of the system and can be studied in more detail in [19,20,21].

In order to understand the theory behind optimal guidance, it is necessary to look at dynamic programming theory. Dynamic programming theory is based on the "principle of optimality". Given the optimal path from A to C, the segment from any point B, on the optimal path, to C is, in fact, also the optimal path from B to C. This principle is show by Equation 1.

Equation 1: $$J_k^*(x_k) = \min_{\Pi^k} \underset{w_k,\ldots,w_{N-1}}{E} \left\{ g_N(x_N) + \sum_{i=k}^{N-1} g_i(x_i, \mu_i(x_i), w_i) \right\}$$

In Equation 1, J is the optimum cost, g(x) is a given function of the "cost-to-go", and π is the set of admissible policies or controller inputs. It says that the optimal path can be constructed from the end to the beginning of the path, based on smaller sub-paths that are also optimized. The mathematical statement of the "principle of optimality" is expressed in Equation 2.

Equation 2: $$J_k^*(x_k) = \min_{\Pi^k} \underset{w_k}{E} \{ g_k(x_k, \mu_k(x_k), w_k) \} + J_{k+1}^*(x_{k+1})$$

The mathematics in Equation 2 can be summed up as the cost at any point, X, is given by the optimal cost of getting from that point to the next one, plus the cost of getting to the endpoint from the next point.

In [2], a good description is given by the fact that the optimal path from LA to Boston passing through Chicago would also contain the optimal path from Chicago to Boston. Figure 3 shows a diagram of the theorem described by Equation 1.

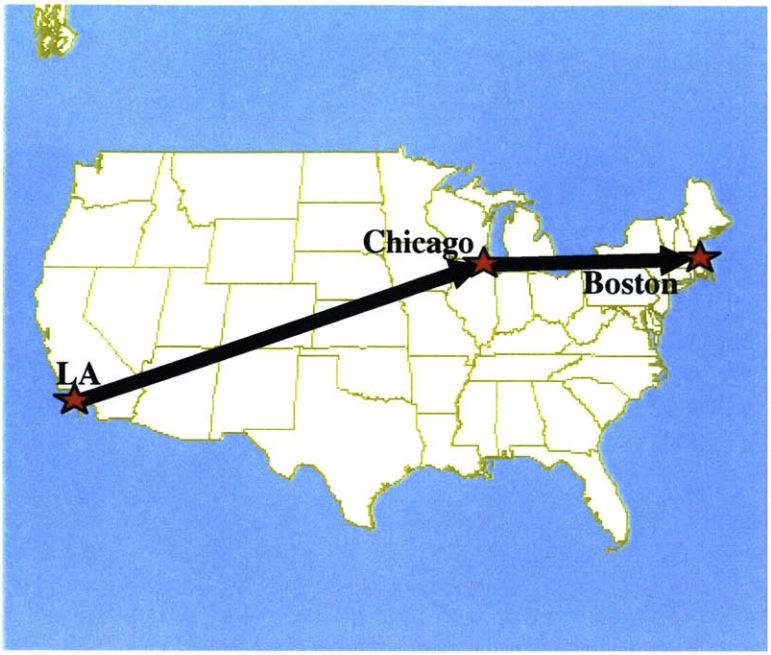


Figure 3: Diagram of Shortest Path Theorem

A "cost-to-go" function is a function where each step forward holds the value of the optimal cost to go until the end is reached.

The problem with the dynamic programming algorithm is that it doesn't lend itself easily to being solved analytically. Numerous simplified models exist that allow analytical solutions, which can be the basis for the solution of many more complex functions. Most of these simplified models are based on the dynamic programming algorithm.

Additionally, there is the numerical approach to solving path optimization. This approach, however, demonstrates the inherent problem in the algorithm. It is an extremely complex problem to solve. The whole state-space has to be discretized. The number of discretizations then controls the complexity of the problem. Unfortunately, it also controls the accuracy and stability of the system. For extremely complex systems, the numerical approach may be beyond the capabilities of a processor. Dynamic

programming is essentially the only general approach and can be used as the basis for sub-optimal approaches as well. Some example applications and a more detailed mathematical approach can be found in [2].

In the interest of further optimizing the function, additions can be made to the dynamic programming algorithm. By augmenting the state-space with known data that can help in the selection of a control function, the controller can be further optimized. Unfortunately, the cost of doing so is that the state-space becomes much more complex and the solutions require considerably more calculations and processing power [2]. One of the simpler forms of augmentation is the incorporation of previous states in the current control law as seen in Equation 3.

Equation 3:
$$\begin{pmatrix} x_{k+1} \\ y_{k+1} \\ s_{k+1} \end{pmatrix} = \begin{pmatrix} f_k(x_k, y_k, u_k, s_k, w_k) \\ x_k \\ u_k \end{pmatrix}$$

In Equation 3, "y" and "s" are additional state variables that keep track of past values of the states and commanded inputs. Equation 3 is a general form of the equation, which demonstrates a function that depends on its previous inputs and states. Additionally, the ability to use forecasts is explained in detail by reference [2].

In terms of systems modeling, there are two main systems: stochastic and deterministic systems. Stochastic systems, by definition, are dynamic systems driven by random input signals. Deterministic systems, on the other hand, are systems where the inputs and disturbances are modeled as a single value and are therefore predictable. This also means that while a closed loop system in a stochastic system will provide better performance, it will not do so in deterministic systems. The difference comes in practice,

where the system model is rarely the same as the system, and so a closed-loop controller will perform better than an open loop system.

A deterministic system can be optimized initially without the need to see the system feedback and without loss of performance, assuming a perfect system model. A feed-forward system will do just as well as a feedback system. Deterministic finite-state problems are shortest path problems. They can be solved by both a typical backwards algorithm, the closed-loop controller, or a forward algorithm. There is no forward algorithm that can be applied to stochastic systems. Any shortest path problem can be converted into a deterministic finite-state problem. For more information on deterministic state-space problems, see [2].

There are other methods of approaching the shortest path problem. They are usually used in particular cases, where the individual algorithm is suited to the problem. Usually the dynamic programming algorithm performs better or is easier to use because of its focus on sequences of numbers. One such algorithm deals with the fact that systems with large numbers of nodes will also have many that are not reasonable to consider. It then removes the nodes that are clearly at high cost to reduce processing time. In the dynamic programming algorithm all of the nodes are considered in the computation, no matter the relevance.

One of the main problems with dynamic programming is the large dimensional space over which the computations take place. Essentially, the optimization takes place over a discretized space. The calculations for each of the discrete points then are taken over a range of states and values, which leads to a limitation on the possibility of

numerically calculating solutions in real-time. Sub-optimal solutions are generally taken for larger order systems. By definition, these solutions limit the vehicle performance.

2.2 The Hybrid Approach

One of the methods of solving problems of high order and high bandwidth is to reduce the state, the position, the calculations, or other spaces into discretizations, as described in the following sections. Since most systems are continuous in nature, some of the capabilities of the system are lost in this approach. This is becoming less of an issue now that processors are getting faster and discrete systems can be almost approximated as continuous systems in certain cases [6].

2.2.1 Definition of a Hybrid System

Any system that contains both continuous and discrete components can be labeled a "hybrid" system, as shown in Figure 4. Some hybrid systems are simpler in nature to solve due to the reduced number of states allowed in the system [6]. This is only the case when the discretizations reduce the order or dimensions of the system.

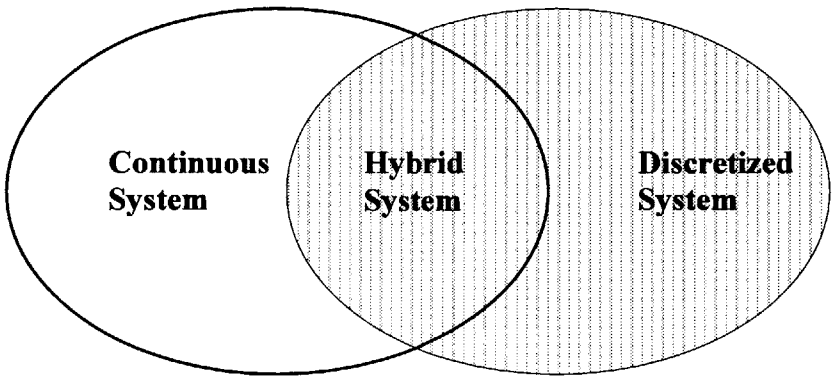


Figure 4: Representation of a Hybrid System

Since the complexity and computational power to model the dynamics and known obstacles alone is significantly larger than is possible to be implemented on current controllers, by using hybrid controllers, probabilistic planners, and relaxing tolerances, the computational problem is more tractable [6].

2.2.2 Robust Hybrid Automaton

The method that was chosen for this research was a hybrid controller, developed in [4], which used a hybrid automaton. The hybrid automaton is essentially a "quantization of the system dynamics" in such a way that the "feasible nominal system trajectories" are restricted to "the family of time-parameterized curves that can be obtained by the interconnection of appropriately defined primitives" [6]. A simple automaton model is presented in Figure 5.

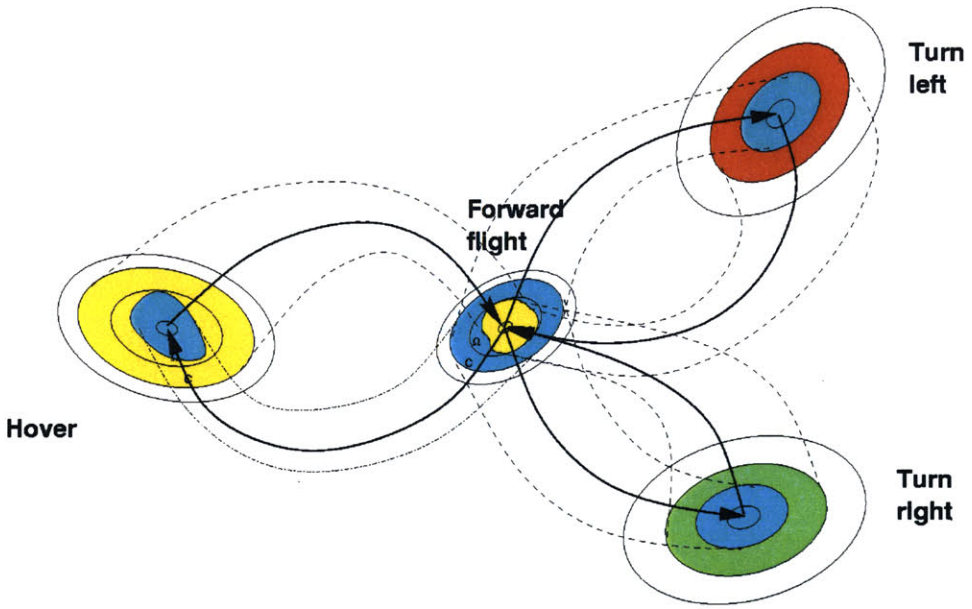


Figure 5: Simple Representation of a Hybrid Automaton [6]

Essentially the system dynamics are broken down into discrete primitives, or trajectories. Each of these trajectories is a stable state of the dynamic system and can be effectively

held constant if desired. The connection of these trajectories through maneuvers allows for a significant reduction in the overall complexity of the problem [6].

There are two main approaches to reducing the computational complexity of the problem through approximation. This first approach is to approximate the system and make it simpler to work with. The second approach is to make the computational algorithm simpler.

Approximating the system can be done by replacing the continuous state space with a discrete space with a finite number of states. The optimal "cost-to-go" can then be calculated using dynamic programming. This can be optimally constructed in a continuous way so that as the discretizations become finer, the system approximates closely the continuous system. For more details, see reference [2].

An additional method that is applied to the systems in this thesis is to grid the state-space, not just the time, and approximate the "cost-to-go" by interpolating between points on the grid. Since it may not always be easy to grid the state space, another option is to grid the algorithm such that the "cost-to-go" is calculated at the nodes on the grid. The "cost-to-go" is then interpolated by least squares or another numerical approximation. This hybrid automaton allows for a higher bandwidth controller in real-time.

The hybrid automaton described in [4-9] involves both approximations. The state-space is discretized into trim states, which limit the possible states to those that the vehicle can achieve. Additionally, the "cost-to-go" function is also discretized and the cost for a particular point on the grid is calculated by interpolating linearly between four or eight nodes depending on if it is two or three dimensions.

The discretizations in the hybrid automaton are represented by trim states and the maneuvers, which are transitions between these trim states. Maneuvers must be feasible so that they will not cause collisions with obstacles, in the case of obstacle avoidance, and are within the realm of the system's ability to perform.

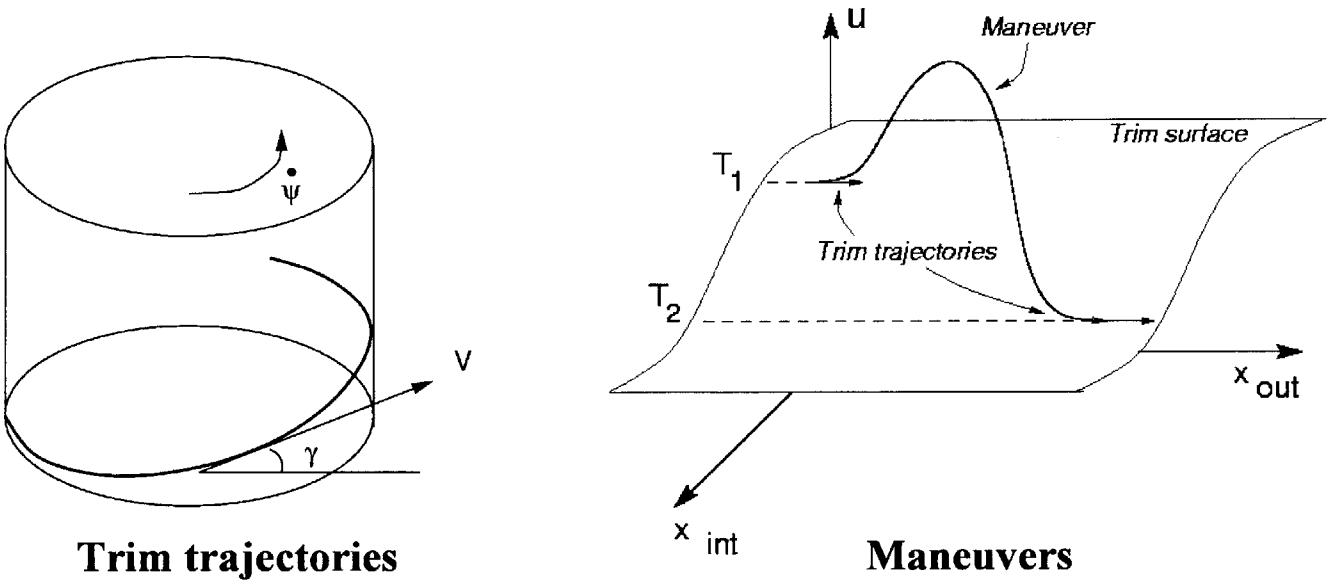


Figure 6:Feasible Maneuvers in Maneuver Space [3]

The diagram in Figure 6 graphically illustrates the maneuver space and the hybrid implementation. [7]

2.2.3 Optimization of Hybrid Maneuvers

A complete path-planning algorithm should both allow for feasible maneuvers and satisfy the initial and final boundary conditions. A feasible maneuver is a maneuver that transitions between two trim states and maintains stable control over the vehicle. Some trajectories take into account an obstacle-avoidance algorithm along the path as a secondary objective function. Normally this takes a considerable amount of computing

power; however, with the hybrid control formulation of the maneuver space, the problem is considered feasible [7].

A cost function assigns a cost to every operation and maneuver in a path-planning formulation. The maneuver selection is based upon the weighted value of the possible paths to the objective trajectory. Since the hybrid implementation frees up enough processing power, the ability to computer the secondary objective, obstacle-avoidance, is achievable and desirable. More information on optimization can be found by referencing [2].

2.3 Breaking Down the Problem

In order to complete the optimization for processors, the hybrid controller is broken down into two parts. The initial calculations, including the cost function and the maneuver space, are processed ahead of time so as not to increase the real-time process requirements. A basic model is represented graphically in Figure 7.

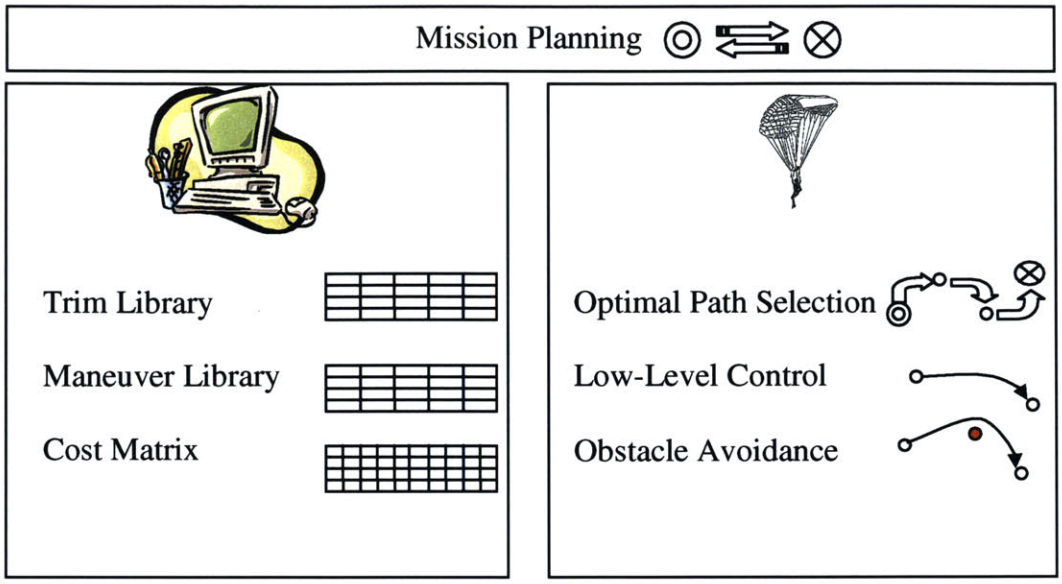


Figure 7: Distribution of Calculations – Offline vs. Onboard

The real-time calculations, including the optimal path, are done using the results of the initial calculations and are designed as light processor loading.

2.3.1 Offline Calculations

The offline calculations are the set of instructions that can be performed before implementing the real-time portion of the controller. This reduces considerably the requirements for the controller. The portions that can be calculated ahead of time include the cost function based on the state of the vehicle relative to the target state, the trim states, and the maneuver set. The trim states create a quantization of the dynamic state space of the system.

The trim states can be selected in fine or coarse grid depending on the available processor and memory for the optimization. The finer the grid, the closer the final

solution will be to the optimal solution. The coarser the grid, the less resources are required for processing the data in real-time.

The maneuver set is chosen based on the trim states and the dynamic model of the system. A maneuver is defined as the transition between two primitives, or trim states. The dynamic model is used to derive the characteristics of the system while changing between each of the sets of trim states.

The trim states and maneuver sets are used, along with the cost function, to calculate the approximate cost of guiding the vehicle from the initial state to the final state. The cost matrix is a set of data derived from the costs of reaching the target based on position. The cost matrix is passed on to the onboard code for path-planning algorithms.

2.3.2 Onboard Calculations

The onboard calculations occur in real-time and are expected to be run on the controller's processor onboard the vehicle. Due to the initial calculation of the cost matrix, the calculations for guiding the vehicle are much simpler and more easily implemented. Essentially the main objective is to select the lowest cost method of reaching the end target. All that the real-time processor has to do is determine the set of maneuvers, based on the cost matrix, which optimizes the cost function and gets the vehicle to the target. The processor should continually loop through this selection process in order to minimize error due to changes in the environment.

The ability to maneuver in an environment with moving obstacles (possibly hostile in nature) could be implemented using this approach as well. Essentially optimizing the approach and path of the vehicle to deal with these constraints in a

dynamic fashion is the key idea. The vehicle should be used to the peak of its abilities, such that the path is truly optimal for the vehicle in such an environment [7].

Chapter 3: Hybrid Control Applied to Xcell-60 Helicopter

The first implementation of the hybrid controller was on the Xcell-60 stunt helicopter. The helicopter dynamic model and basic guidance system were created and described in references [1-4,6-9].

3.1 Helicopter Dynamic Model

The dynamic model of the Xcell-60 helicopter is a full model of the overall dynamics. The objective is to simplify the dynamics for the purposes of control and yet accurately portray the motions of the helicopter. Reference [6] develops the dynamic model for the general class of systems and then for the helicopter in particular.

The basic model in [6] is that of a rigid body system. The kinematics are expressed in Equation 4, where g is defined in Equation 5, and $\hat{\xi}$ is defined in Equation 6.

Equation 4: $\dot{g} = g\hat{\xi}$

The matrix, g, in Equation 5 is defined by the rotational matrix R and the translational matrix p. The g matrix is a homogenous transformation matrix.

Equation 5: $g = \begin{bmatrix} R & p \\ 0 & 1 \end{bmatrix}$

The $\hat{\xi}$ matrix defined in Equation 6 is a "twist" matrix.

Equation 6: $$\hat{\xi} = \begin{bmatrix} \hat{\omega} & v \\ 0 & 0 \end{bmatrix}$$

The $\hat{\omega}$ and v matrices are the angular and translational velocities in the body axes. For more detail, see reference [6].

The general equations for the dynamics of this class of autonomous vehicle are defined by Equation 7 and Equation 8. Equation 7 defines the rotational and torque components in relation to the inertial components of the system. J_b is the inertia tensor of the vehicle and M_b is the matrix of the moments applied to the vehicle.

Equation 7: $$J_b\dot{\omega} = -\omega \times J_b\omega + M_b(g, \hat{\xi}, u, \omega)$$

Equation 8 defines the translational and force components of the system in respect to the inertial components of the system. The matrix m defines the masses of the system. The matrix F_b is the set of forces applied to the system.

Equation 8: $$m\dot{v} = -\omega \times mv + F_b(g, \hat{\xi}, u, \omega)$$

Reference [6] contains more information on this model and the specifics of the F_b and M_b matrices for the helicopter framework.

3.2 Helicopter Hybrid Controller

The helicopter was set up with a basic back-stepping controller, as defined in [9]. The controller was used to keep the helicopter tracking the requested trajectory. The selection of the desired trajectory was handled by the guidance system. Section 3.2 addresses the hybrid controller and the implementation of the code. The subsections are listed in order of implementation and consideration as part of breaking down and solving the problem of hybrid guidance.

3.2.1 Quantizing the Trim States

The hybrid automaton, as described in Section 2.2.2 , consists of a model of the system where the dynamics of the vehicle were selectively discretized to reduce the complexity of the system and still allow for the selection among a large number of feasible maneuvers and trim conditions. A simple diagram of the automaton, as seen earlier, is shown again in Figure 8.

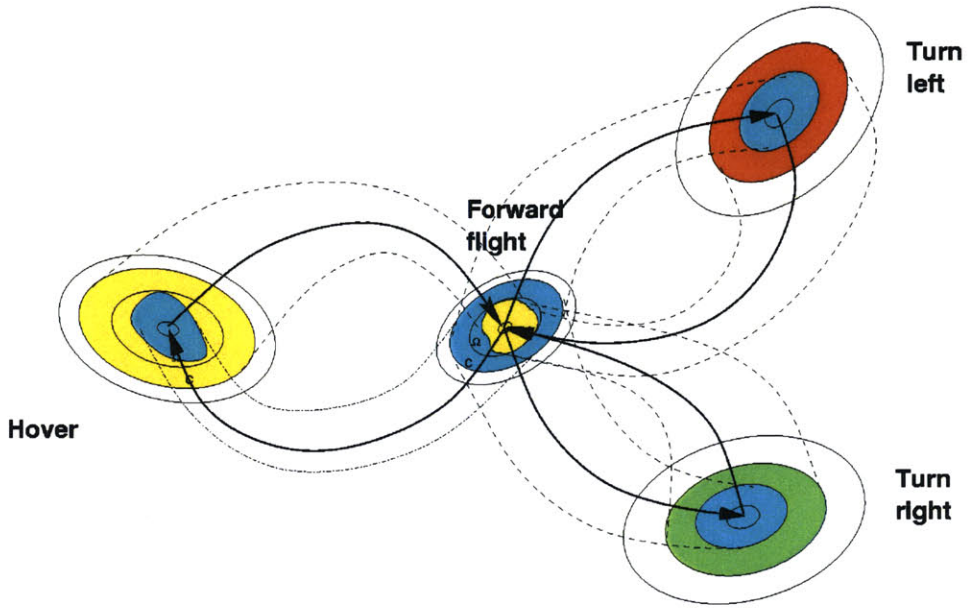


Figure 8: The Robust Hybrid Automaton [1]

Once the theory was proposed, the task of selecting the appropriate discretizations was necessary. The vehicle would have to have a set of trim conditions, where the vehicle could maintain the state for extended periods. Maneuvers would be the conditions that allowed for the transition between the trim states. The first task was to select the trim state variables such that the whole state, or at least the most significant portion of the state could be reconstructed easily from the information stored. The trim

states would be calculated and listed in a library of possible trim conditions such that the guidance algorithm could select the optimal path.

Given the dynamics of the helicopter, the velocity (V), the flight path angle (γ), the yaw rate ($\dot{\psi}$), the sign of the thrust force (which allows for inverted flight if desired), and the sideslip angle (β) determine all of the possible trim states for the helicopter. An example of the trim library is listed in Table 2.

ID	V [m/s]	$\dot{\psi}$ [rad/s]	β [rad]	γ [rad]	Sign	Notes
0	0	-0.5	0	0	1	Rotate Left
1	0	0	0	0	1	Hover
...	...	...	...	...	...	...

Table 2: Example of the Helicopter Trim Library [3]

In Table 2, all of the variables are in the body frame of the vehicle. In steady-state, a trim uniquely characterizes the vehicle state. Since the states are discrete, the full dynamics of the vehicle are not reached, but the calculations are greatly simplified.

The final implementation of the trim library was through a Matlab script that input the requested range for each variable and then generated the trim library in the columnar format in Table 2 by iterating through each of the desired values of each of the variables and exporting it to a file.

3.2.2 Creating the Maneuver Library

The maneuver library, unlike the trim library, was completely derived from the trim library and the dynamic model of the vehicle. Maneuvers are essentially the

transitions between the trim states. When the maneuvers, which bridge the gap between the states, are constructed, they are tested for feasibility using a dynamic model. Because of this, all of the calculated maneuvers are feasible and within the realm of controlled flight.

The maneuver library was created by iterating through each of the trim states and creating a maneuver between it and each of the other trim states in the trim library. The estimated change in each of the states was calculated using the dynamic system model of the helicopter. Maneuvers can also be created from recorded maneuvers with remotely controlled vehicles commanded by test pilots. A partial example of the maneuver library is shown in Table 3.

ID	T_0	T_f	dt [s]	dx [m]	dy [m]	dz [m]	$d\psi$ [rad]	Type	Data File
0	0	0	0	0	0	0	0	0	NULL
1	0	1	0.02	0	0	0	-0.005	0	NULL
...	...	...	...	...	...	...	...	...	...

Table 3: Example of the Helicopter Maneuver Library

The choice of variables in Table 3 was based on the information that the cost optimization algorithm required in order to select the optimal maneuver. In this case, time was the function being optimized, so the change in time, dt, was required.

Additionally, the optimization function also needed to know what the maneuver would change in order to select the correct transition path. The state changes due to the maneuver were based on the change in position and yaw angle in the inertial frame. The rest of the differentials in states were easily obtained through subtracting the initial and

final trim states from each other. The data file column will be addressed later in Section 3.4 with respect to pilot-inspired maneuvers.

The maneuvers were calculated by choosing the initial and final trim states and calculating an 8^{th}-order Savitsky-Golay spline between the states. Since the calculations are done offline, there is no loss of processing time online.

3.2.3 Calculation of the Cost Matrix

Once the trim library and maneuver library were calculated and written, the next step was to calculate the cost matrix. The cost matrix is the offline numerical matrix of the cost-to-go at each point on a cylindrical (or polar in two-dimensional guidance) grid. The grid was defined based on the range at which tactical maneuvering may have been necessary. The grid in Figure 9 shows a basic breakdown of the space around the target. The spacing of the gridding was actually a logarithmic function, which provided much finer detail nearer to the target, where more maneuvering might have been necessary.

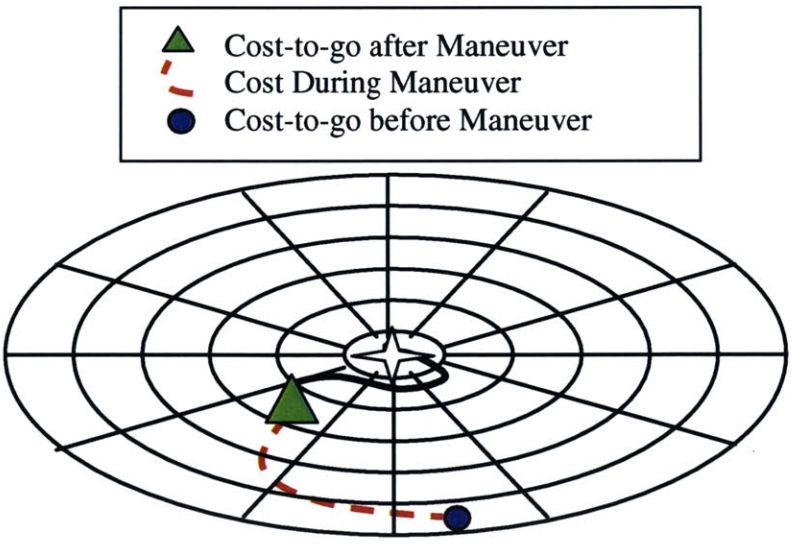


Figure 9: The Positional Grid for the Cost Matrix [3]

The optimization, as explained in Chapter 2, is based on the dynamic programming algorithm. The dynamic programming algorithm is essentially an implementation of the principle of optimality, which states that the "cost-to-go" before a maneuver is equal to the sum of the cost incurred during the maneuver and the "cost-to-go" after the maneuver is completed.

Optimization of the cost function requires that a certain cost be assigned to every selectable maneuver from the current node. In the current implementation, time was the function to be optimized. Each maneuver or change between trims was assigned a certain cost. Initially, the cost grid was selected and a certain value was assigned to each of the nodes based on a basic scenario for each node. For example, for a far away point, the initial time estimate would be for the vehicle to turn around towards the target, accelerate, coast at full speed, and decelerate in time to arrive at the destination trim state and position. The scenario for a close target includes the estimate of time for overshooting the target and turning around. The cost within an acceptable tolerance of the target is just proportional to the radius from the target. There are a number of cases that cover all possible starting points of the vehicle relative to the target. These cases are the bases for assigning the initial cost to each of the nodes in the cost matrix.

Once the initial values were assigned, the cost-optimization function was evaluated based on the position and heading relative to the target, as well as the current trim state. The initial values were then run through a process called "value iteration", where the cost values were updated based on recalculated optimal paths. After a significant number of iterations through the cost grid, the cost matrix represented a close

approximation to the optimal cost between each node and the target for the discrete maneuver and trim space.

Iteration of the cost matrix optimization function occurred in a set order. Each of the possible maneuvers, beginning with the last one, was evaluated over the time period for each maneuver. The cost of each maneuver was updated based on three functions: the cost to remain in the trim trajectory until the beginning of the maneuver, the cost to execute the maneuver, and the "cost-to-go" at the end of the maneuver. The maneuver yielding the lowest "cost-to-go" was selected after the additional costs were added. The "cost-to-go" at that node was then updated with the new value. The updates were saved in a separate location until the whole iteration was complete to yield an accurate iterative method between the nodes. The operation was iterated until the cost improvement reached a certain threshold or a certain number of iterations were completed. [3]

3.2.4 Calculating the Optimal Path

Calculation of the optimal path was done in real-time onboard the vehicle. The cost matrix, which was calculated offline, allowed for simple selection of the optimal path. The cost optimization function is another implementation of the dynamic programming algorithm. The function is given the current position, heading, and trim state, with which it begins the optimization routine. The lowest cost maneuver is selected by iterating through each of the possible maneuvers and adding the cost of the maneuver to the "cost-to-go" at the end of the maneuver. The realm of possible maneuvers is selected based on maneuvers that begin in the current trim state. The selection algorithm is diagrammed in Figure 10.

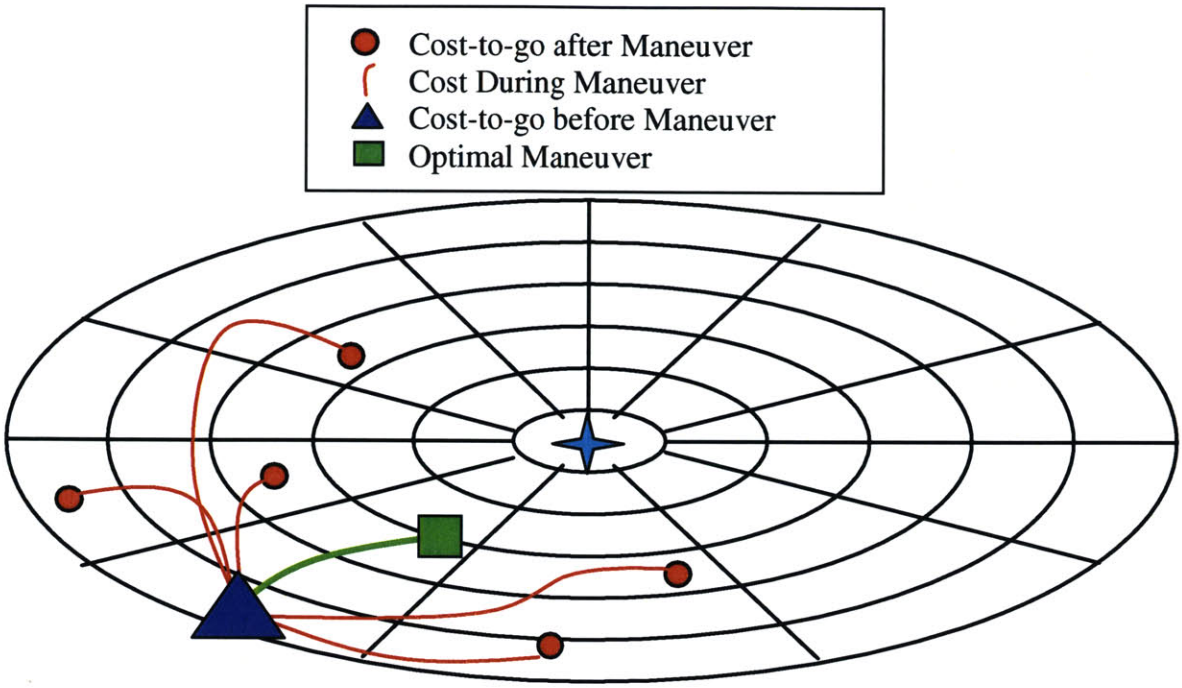


Figure 10: Online Selection of the Optimal Path [3]

In Figure 10, the triangle is the initial position of the vehicle. The algorithm is in the process of selecting the next maneuver. The dynamic programming algorithm iterates through all of the possible maneuvers from the current trim state, as shown by the circles and square. In each iteration it sums the cost of the maneuver with the "cost-to-go" at the end of the maneuver. The dynamic programming algorithm then selects the lowest cost maneuver from the resulting sum of costs, as shown by the square in Figure 10. The dynamic programming algorithm then repeats the same selection process for all maneuvers starting at the square, which represents the next trim state. The process continues until the vehicle path reaches a tolerance limit around the intended target.

3.2.5 Results of the Two-Dimensional Hybrid Controller

The two-dimensional hybrid controller worked well in simulation. The initial takeoff and final landing were hard coded into the simulation. Since the two-dimensional simulation was not part of the research for this thesis, a basic path is shown in Figure 11 to demonstrate that it functioned.

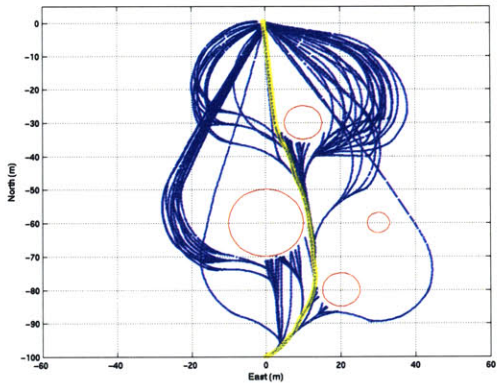


Figure 11: 3D Plot of the Two-Dimensional Hybrid Controller Simulation [6]

For more information on the simulation and the results, refer to reference [6]. In Figure 11, the plot contains numerous plots for the path-planning algorithm, which was also implemented on the two-dimensional helicopter model. The plot was generated using the obstacle avoidance algorithm to select the optimal path that allows passage through the maze.

3.3 Helicopter Three-Dimensional Hybrid Controller

The initial hybrid controller for the Xcell-60 helicopters was implemented in the two-dimensional case. The next extension was to add the third dimension to the code. The changes were mostly just code-related and had little effect on the application of the dynamic programming algorithm to the helicopter. In addition to the logarithmic polar grid used to determine the distance of the vehicle from the target, the z dimension was

added and gridded as well. The z-axis was gridded on a linear scale so as to create a cylindrical coordinate frame, as shown in Figure 12.

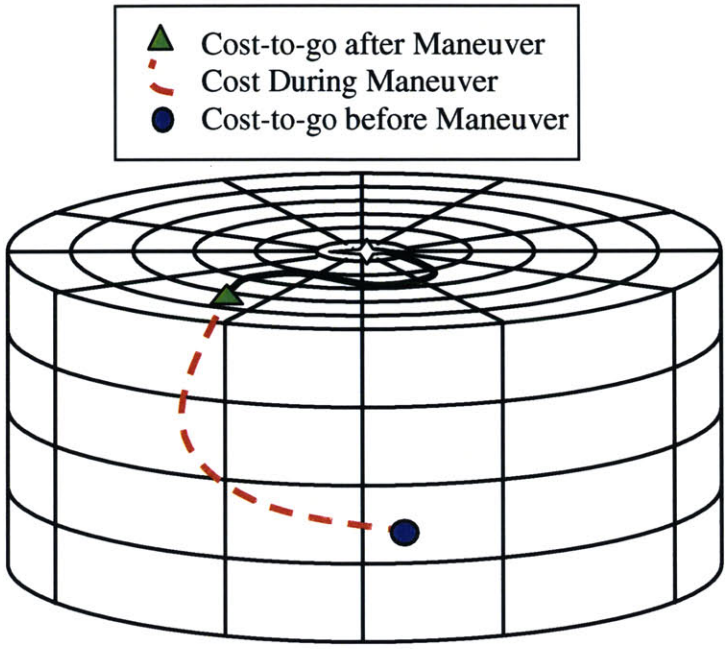


Figure 12: Gridding of the Three-Dimensional Space [3]

The addition of the third dimension also required that the trim states be appropriately updated with changes in the upward and downward directions. Essentially this only required adding in a range of flight path angles in the z direction. Once the values had been added, creation of the trim and maneuver libraries was already automated to reflect changes in the z direction.

Once the trim and maneuver libraries were equipped to handle flight in three dimensions, the rest of the cost selection function had to be updated as well. This was done in several locations throughout the code. The indexing and cost iteration functions had to be updated to include the third dimension. Once these functions were able to read and iterate through the values, the spline function had to be updated to take the third dimension as an input and to output a three-dimensional spline. This allowed for the

assignment of the appropriate cost to the maneuvers and the proper values in the cost matrix.

3.3.1 Results of the Simulation

The three-dimensional simulation worked well. The simulation demonstrated that the hybrid controller could effectively select paths in three dimensions. A basic three-dimensional path selection is shown in Figure 13.

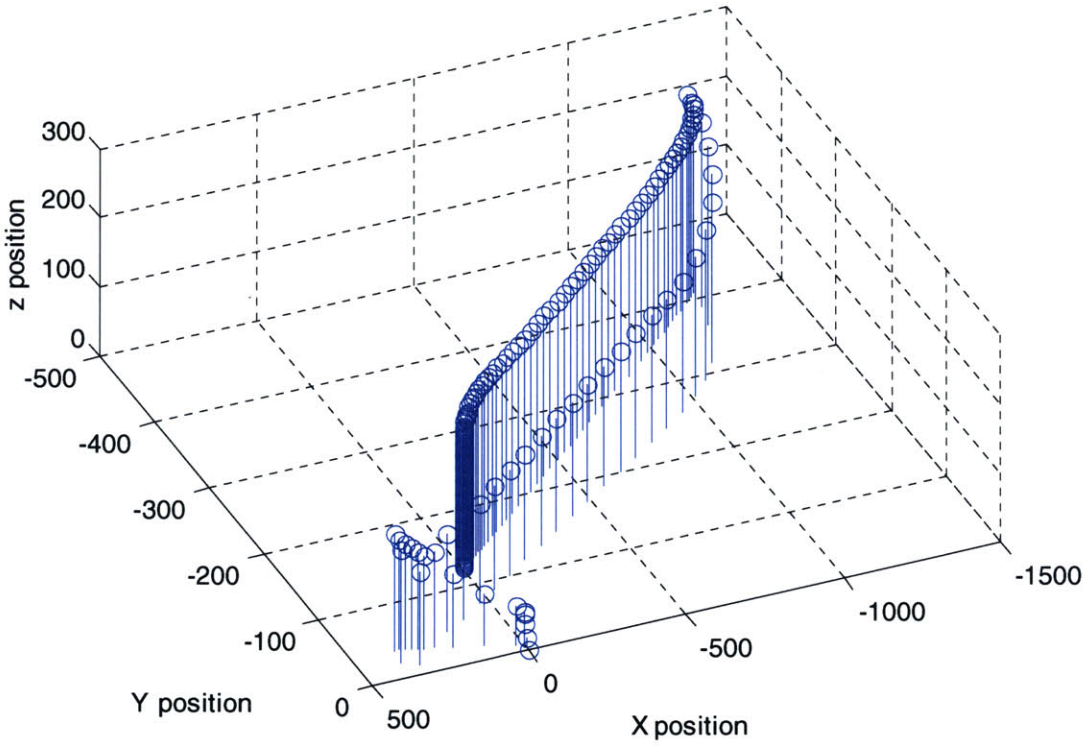


Figure 13: 3D Plot of Three-Dimensional Hybrid Controller Simulation

Additionally, the helicopter simulation could be commanded to implement hard-coded pilot maneuvers, as discussed in Section 3.4 .

3.4 Pilot Inspired Maneuvers

The typical autonomous controllers severely limit the performance of the vehicle compared to the actual vehicle capability. The performance of a human pilot is significantly better than the typical autonomous controllers, but still not near the full capabilities of the vehicle [8]. Figure 14 is reprinted below to demonstrate the comparison graphically.

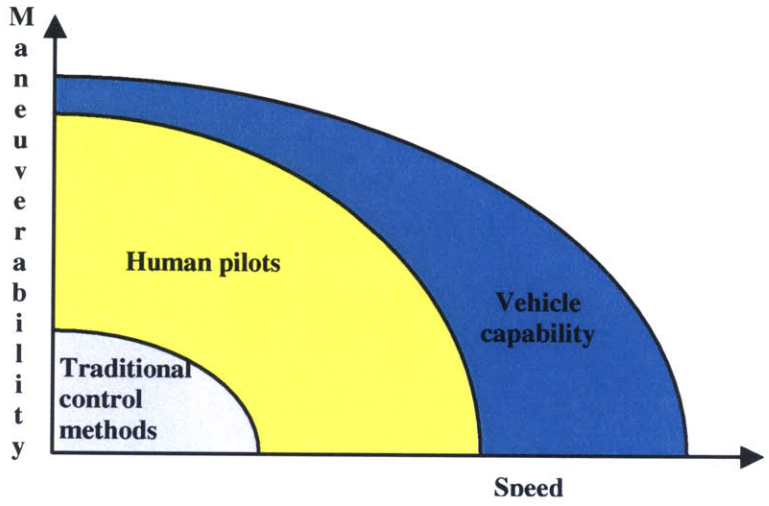


Figure 14: Maneuverability and Speed of Vehicles vs. Controller [1]

Since human commanded maneuvers are outside the reach of typical autonomous controllers, it was asserted that pilot-inspired maneuvers would also be a good application of the hybrid controller. The idea was to capture the stick inputs of the helicopter stunt pilots and use them to reproduce the maneuvers as part of the maneuver library. The autonomous guidance system would then be able to select from a library of pilot-inspired maneuvers as well as the typical spline maneuvers between trim states [8]. Typical controllers are limited or unable to perform pilot-like maneuvers due to several problems. The maneuvers themselves are extremely difficult to perform given the bandwidth and processing requirements. Additionally, the linearized models used by

typical controllers are limited to small angles for accuracy, which nearly precludes the maneuvers in the first place.

The pilot-inspired maneuvers have many benefits. The maneuvers were initially learned intuitively by humans and require little processing power to calculate. Mapping out each of the maneuvers can be done with a good sensor array and data recorders. By allowing pilot-inspired maneuvers, the helicopter is able to perform complex maneuvers based on the trajectories commanded by the human pilots and therefore already demonstrated to be stable [8]. The additional advantage of pilot-inspired maneuvers is that the computer is executing and selecting the maneuvers, so this pushes the vehicle performance closer to its maximum.

3.4.1 Capturing the Stick Commands

The methodology and procedure used to capture the stick commands of the helicopter stunt pilots are described in detail in reference [8]. The actual procedure was relatively simple. The idea was to reduce the complexity of the data gathering, so only the stick commands were recorded digitally. Additionally, a video camera recorded some of the states that could be easily determined visually. The stick commands were repeated on a high-fidelity model of the Xcell-60 helicopter in a simulation. This allowed for the capturing of the approximate states of the vehicles based on the stick commands. Additionally the maneuvers were repeated with a high degree of accuracy [8].

3.4.2 Mapping the Maneuvers

After the data was recorded, each of the maneuvers was characterized by the states and stick commands. In particular, three maneuvers were characterized for implementation on the helicopter: the split-s, the barrel roll, and the forward flip. Each of

the maneuvers had characteristic state changes that allowed easy identification, as well as marked timings that would allow for a more easily implemented autonomous controller.

The forward flip is a relatively self-explanatory maneuver. The helicopter is heading in one direction at a particular velocity and proceeds to rotate around the y-axis until it has done a complete revolution and is facing the same direction in which it began. The explanation in terms of the states of the helicopters is a bit more complex, but fairly straightforward. The forward flip is demonstrated in Figure 15.

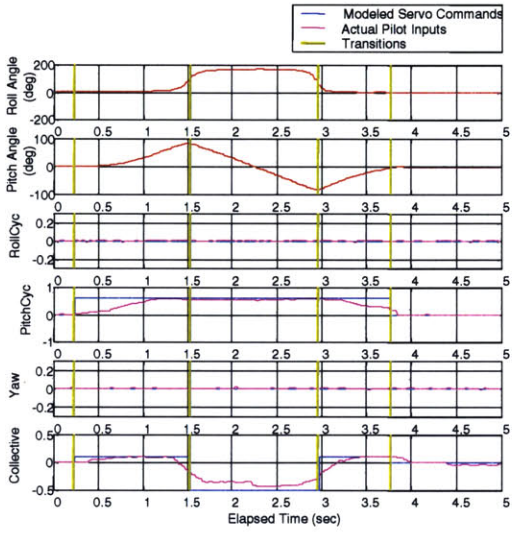


Figure 15: The Forward Flip Maneuver States [1]

The split-s is a complex maneuver. It involves rotating the helicopter around the x-axis 180 degrees and then rotating around the y-axis 180 degrees to fully reverse the direction of the vehicle. The split-s maneuver also has a characteristic state-change signature, as shown in Figure 16.

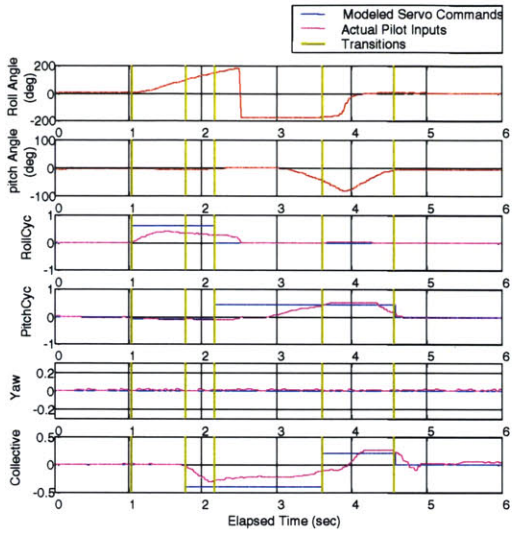


Figure 16: The Split-S Maneuver States [1]

The barrel roll is also a complex maneuver. The maneuver requires the helicopter to do a full rotation around the x-axis while in forward flight. While this is not as complex as the split-s maneuver, it does require significant motion of the collective and the cyclic roll pattern. The barrel roll is demonstrated in Figure 17.

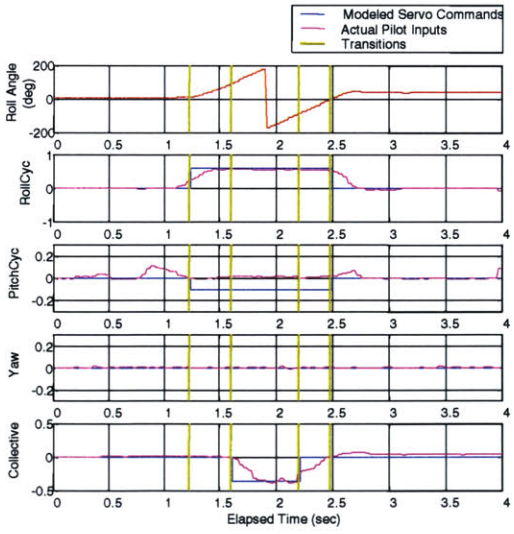


Figure 17: The Barrel Roll Maneuver States [1]

In addition to storing the recorded states of the helicopter, the simulation also allowed for the recording of the time or cost required for each maneuver. This data was then extracted from the data files and placed in the maneuver library in order to allow the ability to select the pilot-inspired maneuvers as part of maneuver set. The implementation is demonstrated in Table 4.

ID	T_0	T_f	dt [s]	dx [m]	dy [m]	dz [m]	$d\psi$ [rad]	Type	Data File
0	0	0	0	0	0	0	0	0	NULL
1	0	1	0.02	0	0	0	-0.005	0	NULL
...	...	...	...	...	...	...	...	...	...
625	17	17	1.5	-0.5	0	2	3.14	1	SplitS.dat
626	17	17	.5	1	5	0	0	2	Barrel.dat
...	...	...	...	...	...	...	...	...	...

Table 4: Maneuver Library with Pilot-Inspired Maneuvers

The columns were filled in with the simulated data from the simulation, which closely matched the data in the experimental tests. The simulated data was used such that the tests could be repeatable and debugging would be simpler.

Full implementation of the pilot-inspired maneuvers will be implemented in future research. The time constraints on this thesis require that the scope be limited to certain aspects of the project. The pilot-inspired maneuvers have the potential to add significantly to the abilities of the path-planning algorithm and vehicle performance.

Chapter 4: Parafoil

Once the adaptation of the hybrid automaton was completed for the helicopter, adaptation began on a second platform, a parafoil. The parafoil platform was chosen because it was a useful application that was both challenging and available at the right time. Once the parafoil had been selected, work began on developing a dynamic model

of the parafoil. Section 4.1 describes the full dynamic model in detail. Section 4.2 decribes the implementation of a low-level flight controller for the parafoil simulation. Section 4.3 then goes into the code adaptations required to adapt the cost functions and code over to the parafoil. The implementation of the guidance algorithm and hardware within the model are covered in Chapter 5.

4.1 Parafoil Dynamic Model

Initially, a six-degree-of-freedom dynamic model of the parafoil was developed. This simulation served as the main model of the vehicle, since no physical vehicle was available for experimentation. Additionally, applying the functions to a vehicle presented a significant project within itself and was not set within the scope of this thesis. The model was based on models in references [10-18]. The model, though a reasonable description of the system response, contained several approximations. Essentially this model assumed that there was a single mass composed of the mass of the parachute and the mass of the load. Orientation of the load relative to the parachute makes for more complex dynamics, but should not interfere with the guidance assuming no huge perturbations in the system. The parafoil and parachute are inherently stable systems that allow for adjustment in velocity and rotation in the plane of symmetry of the vehicle, but have little power over the angle of approach.

The model included drag, lift, and side-load on the parachute; the single entity has a moment of inertia, which is a close approximation to the actual system. The force of gravity acts in the z direction, while drag acts against velocity and lift acts perpendicular

to both. Lift and drag act in the plane of symmetry of the vehicle, while the side load acts perpendicularly to both lift and drag in the asymmetric plane.

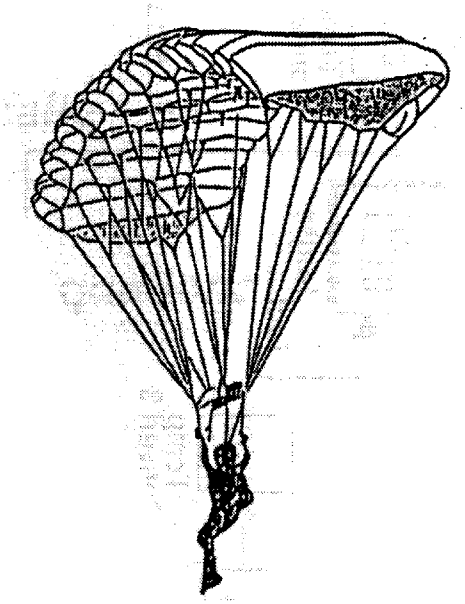


Figure 18: Diagram of a Parafoil.

Typical parachutes have limited control and positioning, where pulling cords adjusts the shape of the canopy in order to guide the system. The parafoil, which is a relatively recent invention, allows for guidance through increasing the drag on the foil through flaps on the edges of the surface. A diagram of a typical parafoil can be seen in Figure 18. The system is controlled using toggles or flaps in the back that control the amount of the drag force. Each of the two toggles is modeled using a moment arm from the center of gravity, with the end drag force proportional to the amount the toggle is actuated. The steady-state performance of the toggles is highly nonlinear, as shown in Figure 19.

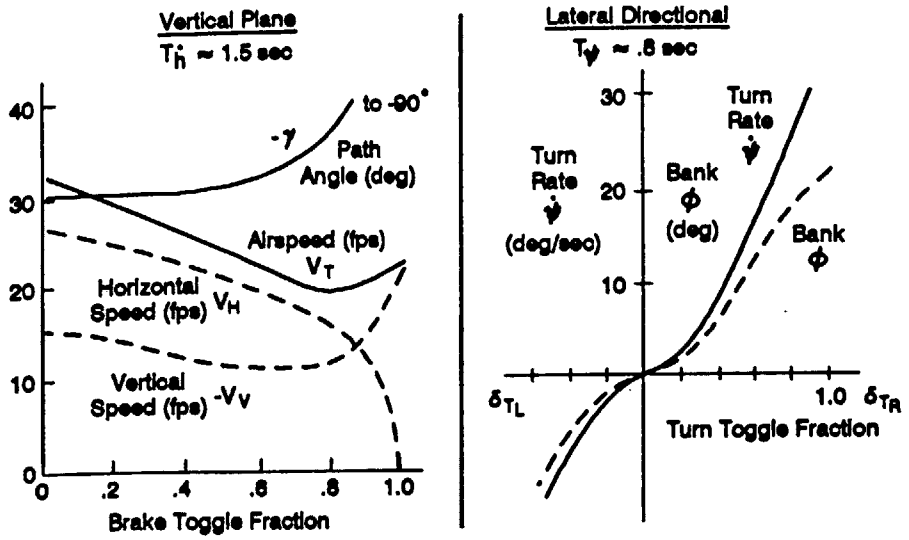


Figure 19: Ram-Air Parachute Experimental Steady-State Data [10]

The coordinate frames, as seen in Figure 20, were set up such that there was a moving frame inside of the parafoil body, centered at the center of gravity, and a wind-relative frame on the ground. The actual position of the vehicle, in absolute terms, can be seen by adding the wind velocity to the model velocity during integration. Once the two coordinate frames were set up and the integration technique was chosen, the next step was to draw the free body diagrams and model the forces.

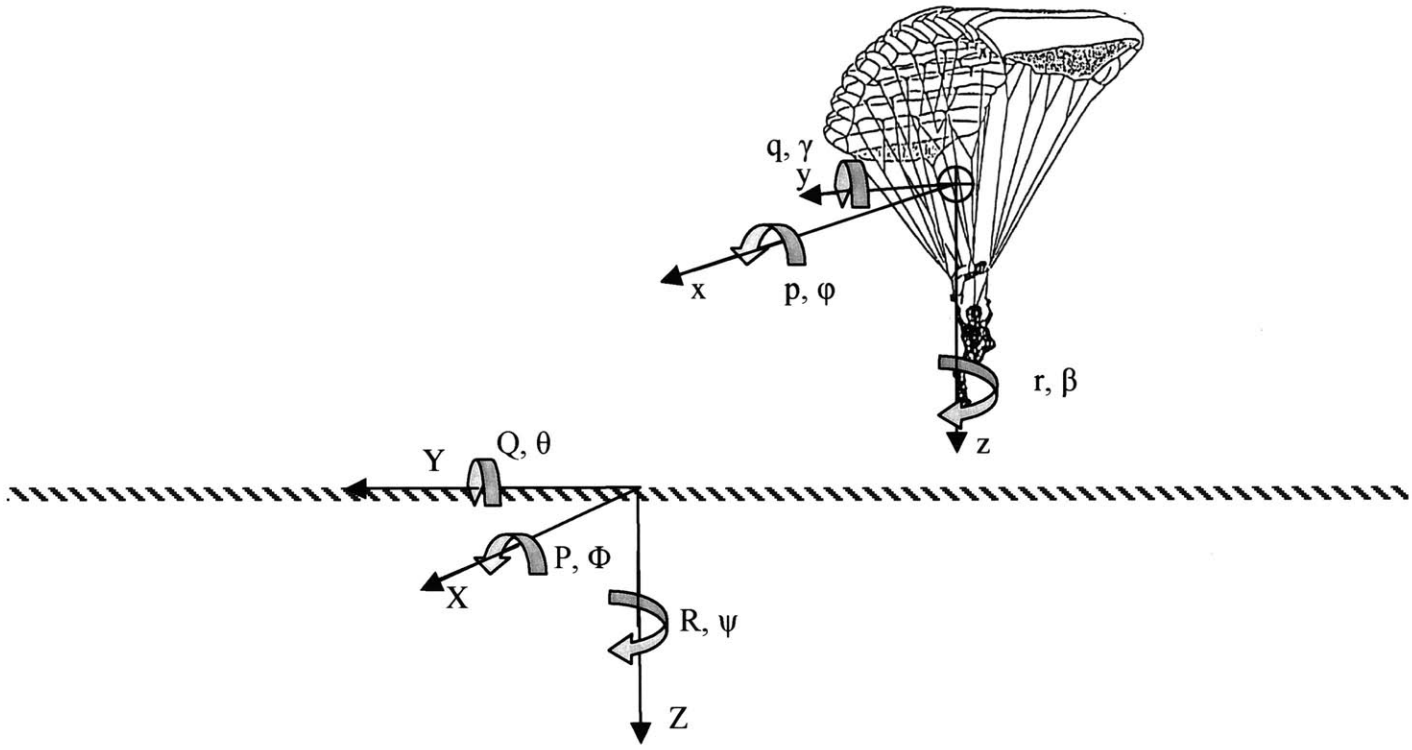


Figure 20: Diagram of Coordinate Axes for 6-DOF Parafoil Model

The free body diagram consists of the components from gravity, lift, drag, and side forces. The force of gravity is always in the Z direction (where Z is the wind-relative ground coordinate frame in the downwards direction). The force of drag is always directly opposed to the velocity of the vehicle. The lift force is perpendicular to both the drag and velocity vectors. The lift force is also generally in the Z and X plane, except when the parafoil is turning around the Z axis. The lift force is applied at a length, L, away from the center of gravity (at the parafoil). The toggles also applied a separate drag force at a distance, r, from the center of gravity.

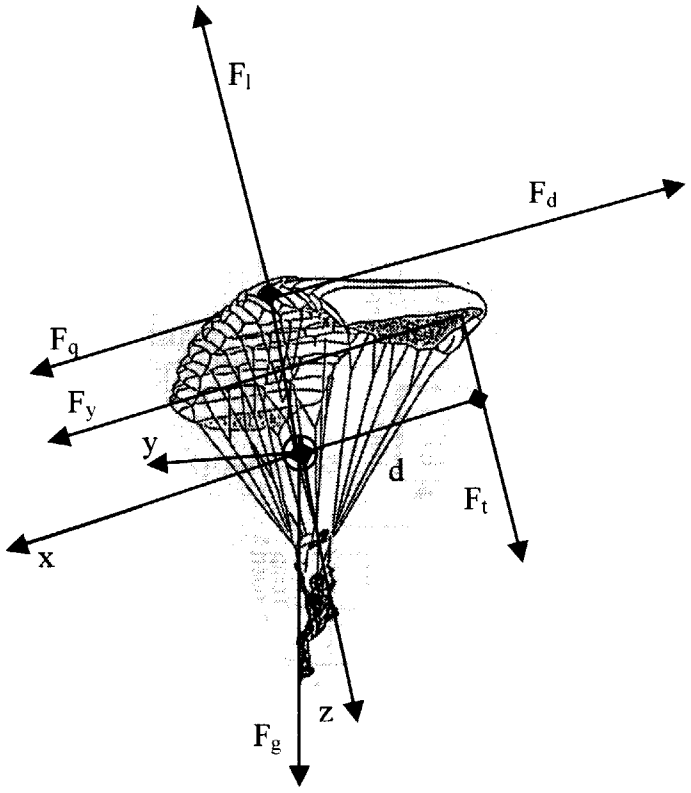


Figure 21: Free Body Diagram of 6-DOF Parafoil Model

Each of the angles and positions in the body and inertial frames has been labeled for reference. The superscript is the frame from which the components are viewed. The subscript is the description of the object being referenced. The letter "i" stands for the inertial frame, while the letter "b" stands for the body or body frame.

The forces were broken down into gravitational forces, Equation 9, and aerodynamic forces, Equation 10 through Equation 14. The gravitational force is applied directly to the center of gravity and is essentially the acceleration due to gravity times the mass of the parafoil and load. Equation 9 describes the resulting acceleration due to gravity.

Equation 9: $$G_b^i = \begin{bmatrix} 0 \\ 0 \\ g \end{bmatrix}$$

The side force, as shown in Equation 10, is based on the dynamic pressure, the coefficient of yaw, and β, the sideslip angle.

Equation 10: $$F_s = SQC_{yb}\beta$$

The drag force, in Equation 11, is essentially the dynamic pressure, shown in Equation 12, times the coefficient of drag times the visible cross-sectional area of the parafoil times the toggle brake drag coefficient, d_b. Although this is not a perfect model for the effects of the toggle due to nonlinearities in drag of the toggles, it does serve as a reasonable approximation.

Equation 11: $$F_d = SQC_d(1 + d_b)$$

The dynamic pressure, shown in Equation 12, is proportional to the square of the velocity of the vehicle, as well as the density of air, ρ.

Equation 12: $$Q = \frac{1}{2}\rho\left\|V_{wr}^i\right\|^2$$

The velocity in Equation 12 is the wind relative velocity of the vehicle. This velocity, defined in Equation 13, is the difference between the Velocity of the body of the vehicle in the inertial frame and the velocity of the wind in the inertial frame.

Equation 13: $$V_{wr}^i = V_b^i - V_{wind}^i$$

Adding the wind in this manner allows for accounting of the wind in the dynamics, but without additional complications imposed by adding the wind into the inertial frame.

The lift force, shown in Equation 14, is similar to the drag, but acts perpendicularly and uses the coefficient of lift as opposed to the coefficient of drag. The

lift force and the drag force add together to determine the rate of descent and the angle of attack.

Equation 14: $F_l = QC_{l\alpha}\alpha$

The variable α is the angle of attack.

The toggle torque, shown in Equation 15, is the result of the motion of the toggles. The toggles are measured based upon the fraction of motion between fully off and fully on.

Equation 15: $M_t = QC_{dt}d_d d$

The toggle torque acts at a distance d from the center of gravity along the y-axis. The left and right toggle fractions are then combined to produce the braking fraction, shown in Equation 16, and the differential fraction, Equation 23. The toggle torques cause the roll and yaw since they are the only torques acting in those orientations. The toggles are also proportional to the drag force.

Equation 16: $d_b = \frac{d_l + d_r}{2}$

Equation 17: $d_d = \frac{d_l - d_r}{2}$

The combined drag of the two toggles is added together to obtain the braking coefficient in Equation 16. The effect of the toggles is not linear, as shown earlier in Figure 19, but can be approximated as such except near stall of the vehicle. The difference between the two toggle deflections gives the differential toggle coefficient in Equation 17. The difference between the drag on the left side of the vehicle and the right side of the vehicle provides the differential toggle, which helps determine the force applied to the vehicle in yaw and roll.

The moment from the lift force is damped by the damping moment in Equation 18. The damping force is based on the angular velocity and shape of the vehicle.

Equation 18: $M_d = QSC_m \omega L$

The coefficient of damping, as well as the drag coefficient is generally derived experimentally. L is the distance between the center of mass and the location where the force is applied. The angular velocity of the vehicle in pitch in the body frame determines the variable ω in Equation 19.

Equation 19: $\omega = [0 \quad 1 \quad 0](C_i^b W_b^i)$

The variable W_b^i is the angular velocity of the vehicle in the inertial frame. The rotation matrix C_i^b is the angular matrix that rotates components from the inertial to the body frame.

Equation 20: $C_i^b = [C_b^i]^T$

It is the inverse of the rotation matrix C_b^i, as is shown in Equation 20. The rotation matrix C_b^i rotates a matrix from the body frame into the inertial frame, as shown in Equation 21.

Equation 21:

$$C_b^i = \begin{bmatrix} \cos(\psi)\cos(\theta) & \cos(\psi)\sin(\theta)\sin(\phi) - \sin(\psi)\cos(\phi) & \cos(\psi)\sin(\theta)\cos(\phi) + \sin(\psi)\sin(\phi) \\ \sin(\psi)\cos(\theta) & \sin(\psi)\sin(\theta)\sin(\phi) + \cos(\psi)\cos(\phi) & \sin(\psi)\sin(\theta)\cos(\phi) - \cos(\psi)\sin(\phi) \\ -\sin(\theta) & \cos(\theta)\sin(\phi) & \cos(\theta)\cos(\phi) \end{bmatrix}$$

The angles ψ,θ, and ϕ are the yaw, pitch, and roll respectively as defined in [22].

The general dynamic equations for the parafoil are based on the simple system described by Equation 22 through Equation 28. Equation 22 describes the change in the

position of the vehicle in the inertial frame based on the velocities calculated by integrating Equation 23.

Equation 22: $$\dot{X}_{\mathrm{b}}^{\mathrm{i}} = V_{\mathrm{b}}^{\mathrm{i}} = C_b^i V_b^b$$

Equation 23 defines the change in velocity of the vehicle in the body frame based on the applied aerodynamic and gravitational forces. This equation comes from the simple summation of the forces acting on the vehicle. The equation is divided by the mass of the vehicle, which makes it an acceleration balance equation. The variable $V_{\mathrm{b}}^{\mathrm{b}}$, the velocity of the system in the body frame, is an intermediate variable that reduces the order of the system for easier integration.

Equation 23: $$\dot{V}_{\mathrm{b}}^{\mathrm{b}} = \frac{\mathrm{F}_{\mathrm{b}}^{\mathrm{b}}}{\mathrm{m}} + C_{\mathrm{i}}^{\mathrm{b}} \mathrm{G}_{\mathrm{b}}^{\mathrm{i}} + w \times v$$

The aerodynamic force term in Equation 23 is calculated based on the aerodynamic forces calculated earlier. Equation 24 describes the forces as they apply to the body frame of the vehicle. The drag force, F_d, works in the negative x direction. The side force, F_s, acts in the y direction. . The lift force, F_l, works in the negative z direction to slow the descent of the parafoil.

Equation 24: $$F_{\mathrm{b}}^{\mathrm{b}} = \begin{bmatrix} -F_d \\ F_s \\ -F_L \end{bmatrix}$$

Since the accelerations and velocities are calculated in the body frame, the force of gravity must be rotated into the body frame, as shown in Equation 25.

Equation 25: $$F_{\mathrm{gb}}^{\mathrm{b}} = C_{\mathrm{i}}^{\mathrm{b}} G_{\mathrm{b}}^{\mathrm{i}} m$$

The rotation matrix, $C_{\mathrm{i}}^{\mathrm{b}}$, is defined in Equation 20.

The rotational dynamics, expressed in Equation 26 and Equation 27, are similar to the positional dynamics, except written in terms of angles and torques instead of distances and forces. Equation 26 describes the change in the angular position of the vehicle by the integrated angular velocity in Equation 27

Equation 26: $$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} p \\ q \\ r \end{bmatrix}$$

The change in angular velocity in Equation 27 is expressed as a function of the inverse of the inertia matrix times the torques applied in the inertial reference frame.

Equation 27: $$\dot{W}_{\mathrm{b}}^{\mathrm{b}} = \mathrm{I}^{-1}\mathrm{T}_{\mathrm{b}}^{\mathrm{b}} + w \times Iw$$

The torques applied to the body frame are shown in Equation 28. The torques are based on the applied forces that were described earlier. The toggle torque, M_t, acts around the x-axis. The lift and damping moments act around the y-axis. The side moment acts around the z-axis.

Equation 28: $$T_{\mathrm{b}}^{\mathrm{b}} = \begin{bmatrix} M_t \\ F_L L - M_d \\ F_s L \end{bmatrix}$$

The integration method implemented was a simple Newton-Euler integration. Essentially the time step was multiplied by the state derivative and added to the current state for times between the starting and ending time. The basic equation is shown in Equation 29.

Equation 29: $[X] = [X] + [\dot{X}] * dt$

In this case, X represents a matrix of states and $\dot{X}$ represents its derivative. The derivative is multiplied by the time step, dt, and then added to the original value. This was done in the body coordinates to avoid dependence on Euler Angles by using small angular motion increment approximations. The body frame velocity and angular velocity vectors are then rotated into the inertial frame, using the rotation and translation matrices. The rotated velocities are then used to calculate the position changes in the inertial frame. This is also done by Newton-Euler integration, as shown in Equation 29. For more information on these processes, see reference [22].

4.2 Parafoil Low-Level Flight Controller

The controller is also implemented in the body frame. The inputs to the system are rotated into the body frame as well. The diagram in Figure 22 shows the basic controller structure.

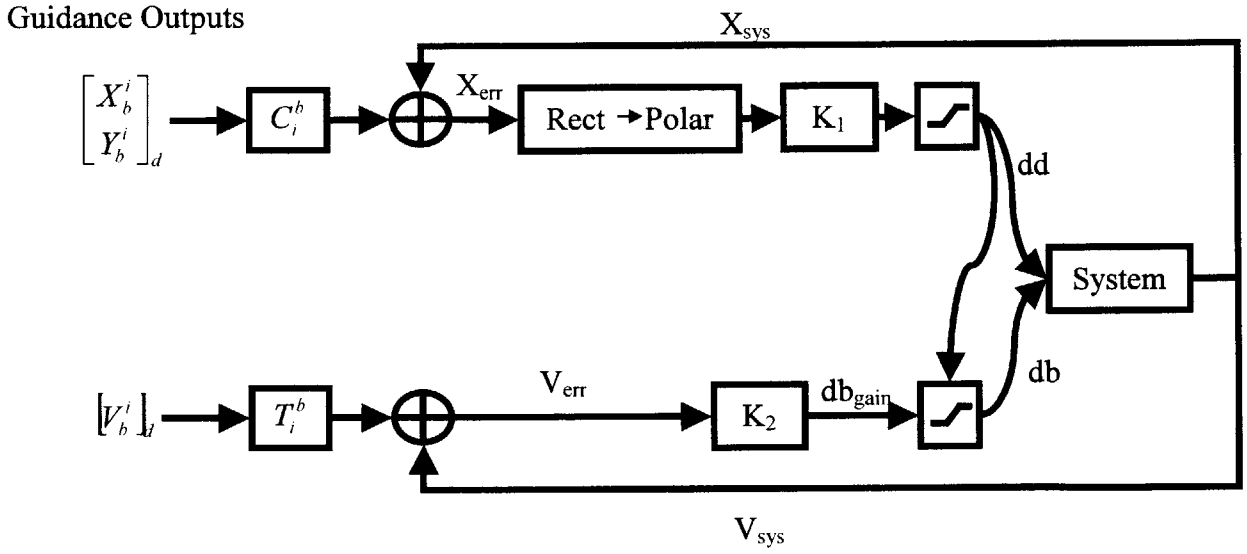


Figure 22: Diagram of the Proportional Controller

The system bandwidth can be estimated from the system model shown in section 4.1 . The dependant rotational portion of the system model has been reproduced in Equation 30.

Equation 30:
$$\dot{W}_{\mathrm{b}}^{\mathrm{b}} = \mathrm{I}^{-1}\begin{bmatrix} \frac{1}{2}\rho\left\|V_{wr}^{i}\right\|^{2} SC_{\mathrm{dt}}\mathrm{d}_{\mathrm{d}}d \\ \frac{1}{2}\rho\left\|V_{wr}^{i}\right\|^{2} SC_{\mathrm{l}\alpha}\alpha L - \frac{1}{2}\rho\left\|V_{wr}^{i}\right\|^{2} SC_{\mathrm{mq}}\omega L \\ \frac{1}{2}\rho\left\|V_{wr}^{i}\right\|^{2} SC_{\mathrm{yb}}\beta L \end{bmatrix}$$

The dependant translational portion of the system model is reproduced in Equation 31.

Equation 31:
$$\dot{V}_{\mathrm{b}}^{\mathrm{b}} = m^{-1}\begin{bmatrix} -\frac{1}{2}\rho\left\|V_{wr}^{i}\right\|^{2} SC_{\mathrm{d}}(1+\mathrm{d}_{\mathrm{b}}) \\ \frac{1}{2}\rho\left\|V_{wr}^{i}\right\|^{2} SC_{\mathrm{yb}}\beta \\ -\frac{1}{2}\rho\left\|V_{wr}^{i}\right\|^{2} SC_{\mathrm{l}\alpha}\alpha \end{bmatrix} + C_{i}^{b}\begin{bmatrix} 0 \\ 0 \\ g \end{bmatrix}$$

These equations can be linearized and then solved for as a transfer function. The linearized systems can be seen in Equation 32. Nonlinear systems are generally linearized about a certain point with a series function and then approximated. In this case, the nonlinear term comes from the $\left\|V_{wr}^{i}\right\|^{2}(1+\mathrm{d}_{\mathrm{b}})$ term, where the input and output are multiplied. This system can be assumed to operate around a set velocity, $\mathrm{V}_{0\mathrm{wr}}^{\mathrm{i}}$, with small changes approximated as $\mathrm{dV}_{\mathrm{wr}}^{\mathrm{i}}$. Once this approximation is introduced, the system can be linearized by assuming that any small terms of higher order are too small to consider and can be eliminated. Once this occurs, the system in Equation 32 can be calculated.

Equation 32:
$$\dot{V}_{\mathrm{b}}^{\mathrm{b}} = m^{-1}\begin{bmatrix} -\frac{1}{2}\rho\left\|V_{0wr}^{\prime}\right\|^{2} SC_{\mathrm{d}}(1+\mathrm{d}_{\mathrm{b}}) \\ \frac{1}{2}\rho\left\|V_{0wr}^{\prime}\right\|^{2} SC_{\mathrm{yb}}\beta \\ -\frac{1}{2}\rho\left\|V_{0wr}^{\prime}\right\|^{2} SC_{\mathrm{l}\alpha}\alpha \end{bmatrix} + \mathrm{C}_{\mathrm{i}}^{\mathrm{b}}\begin{bmatrix} 0 \\ 0 \\ g \end{bmatrix}$$

The transfer function, as seen in Equation 33, represents the system in the frequency domain. It gives the frequency response of the system due to a particular input. The angles are also approximations about a central point, to completely linearize the system. Given that this is a multiple input multiple output system, the equations can get somewhat complex without linearization. Additionally, the linearizations are only for small regions, so the accuracy of the linearization goes down with the number of approximations required.

Equation 33:
$$H_b(s) = \frac{u(s)}{\mathrm{d}_{\mathrm{b}}(s)} = \frac{-\frac{1}{2}m^{-1}\rho {u_0}^2 SC_{\mathrm{d}}}{\mathrm{s} + m^{-1}\rho u_0 SC_{\mathrm{d}}(1+db_0)}$$

The linearized equation, assuming constants, yields a frequency domain plot as seen in Figure 23.

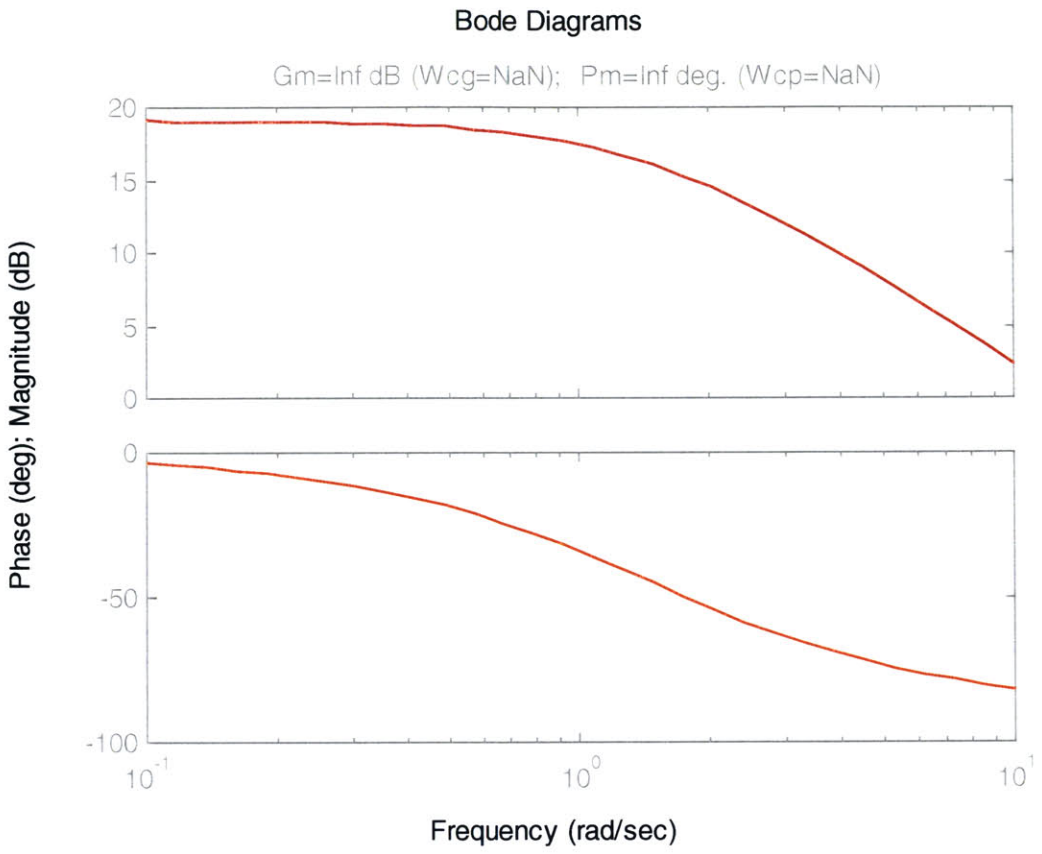


Figure 23: Bode Plot of Linearized Translational System

The rotational dynamic equations also yield a linearized system in a similar manner. The rotational dynamic equations yield results in terms of d_d as an input. The linearized rotational system can be seen in Equation 34.

Equation 34:

$$\dot{W}_b^b = I^{-1}\begin{bmatrix} \frac{1}{2}\rho\left\|V_{0wr}^{i}\right\|^2 SC_{dt}d_d d \\ \frac{1}{2}\rho\left\|V_{0wr}^{i}\right\|^2 SC_{l\alpha}\alpha L - \frac{1}{2}\rho\left\|V_{0wr}^{i}\right\|^2 SC_{mq}\omega L \\ \frac{1}{2}\rho\left\|V_{0wr}^{i}\right\|^2 SC_{yb}\beta L \end{bmatrix}$$

Once the system is linearized, the equations then come together in a similar manner to the translational system. The transfer function between the angular velocity, p, and the differential toggle, d_d, is shown in Equation 35.

Equation 35: $$H_d(s) = \frac{p(s)}{\mathrm{d_d}} = \frac{\mathrm{I}^{-1}\frac{1}{2}\rho\left\|V_{0wr}^{i}\right\|^2 SC_{\mathrm{dt}}d}{s}$$

The linearized rotational system transfer function also yields a similar Bode Plot to the linearized translational system. The plot can be seen in Figure 24.

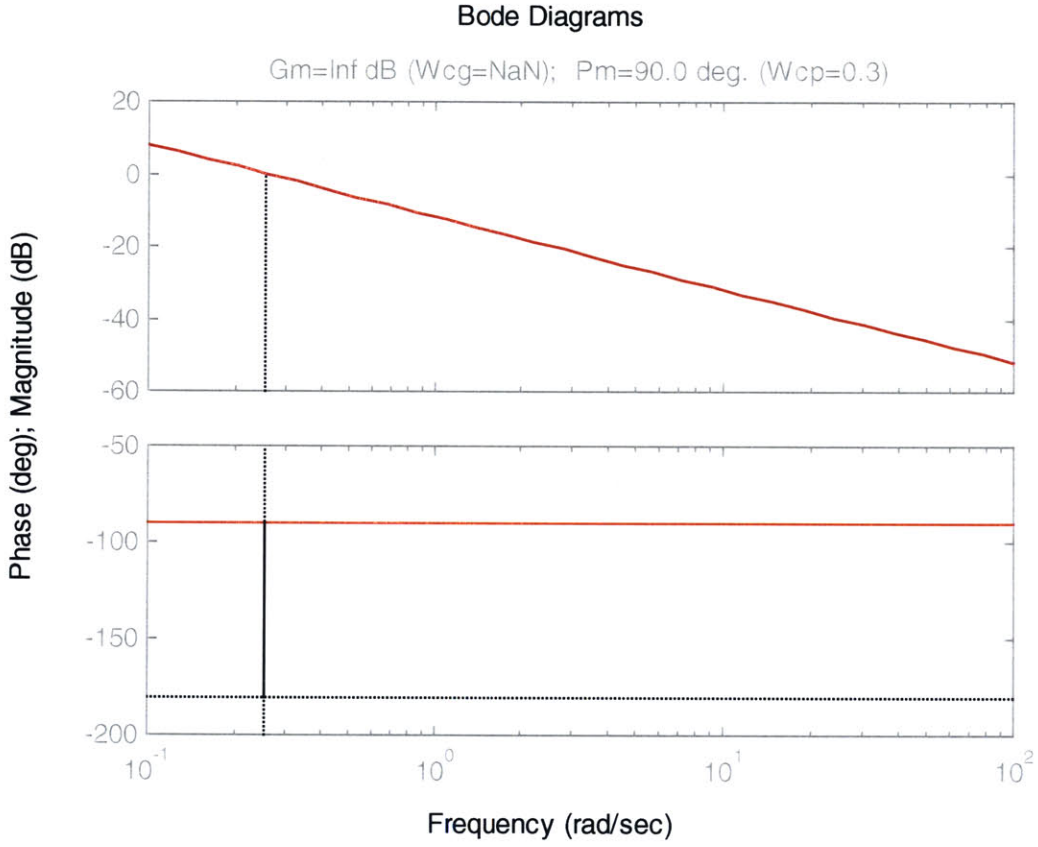


Figure 24: Bode Plot of the Linearized Rotational Transfer Function

The two transfer functions help determine the system response to the toggles. Although there are significant assumptions based on the linearization of the system, the model provides a glimpse of how the system will react around a certain point or set of points. This allows for the design of some basic controllers.

Since the linearized system is a basic set of single-poled systems, adding a lead-lag compensator will allow the adjustment of the bandwidth of the system based on the desired pole placement method. This method, however, is based on the assumption that the system maintains a somewhat constant trim state, which is not the actual case. A more realistic controller for the system would be a nonlinear controller. An adaptive tracking controller would possibly solve the problem better. For the initial model, a linear proportional controller will be used. For more information on nonlinear controllers, see reference [19].

4.3 Parafoil Cost Matrix Adaptation

The cost matrix is the weighting matrix that allows the path-planning algorithm to select the optimal path from the trim and maneuver library. The cost matrix is formed by initially weighting a location matrix based on an initial estimate of cost for each location in a cylindrical position grid. The base of the cylinder is centered around the target trim state, which is not necessarily a point on the ground. In the case of this thesis, the target state is the state from which the parafoil can do a flare maneuver and land the vehicle. The flare maneuver itself is outside of the scope of this thesis, but is well within the realm of a controller to perform by adding it to the maneuver library.

The main change between the helicopter algorithm and the parafoil algorithm was the cost function. The cost function had to define the "cost-to-go" such that the controller would select the optimal path no matter the state of the system.

The helicopter cost function was completely based on the time that it took to execute each maneuver in order to arrive at the destination. In this case, the optimization

is time and the target will be hit, no matter the state. The helicopter was allowed a certain tolerance, within which it had reached the target area.

In the case of the parafoil, the problem is that the time is a bounded period due to the flight path angle and the minimum velocity, but the target will not always be reached. In order to derive a cost function that arrived at the best position in the best time, both factors had to have been taken into account. Additionally, the position is in three dimensions, so a method of mapping the altitude above the target into a final cost had to also be developed.

The cost function that was eventually chosen was based on three factors: the distance from the target in the x and y directions, the difference between the target velocity and the end velocity, and the actual time cost of the maneuver. The parafoil cost function is shown in Equation 36.

Equation 36: $J = A(dt_{err} + dt_{des}) + B(P_{err}) + C(V_{err})$

The initial cost to get to the endpoint of the path is given by dt_{des}. The errors and initial cost are weighted by the three constant weighting matrices A,B, and C. The time error, dt_{err}, is a function of the altitude, the velocity, and the flight path angle, as shown in Equation 37.

Equation 37: $$dt_{err} = \frac{H_{tgt}}{V_{wr}^{i} \sin(\gamma)}$$

The altitude of the parafoil above the desired position is H_{tgt}, while V_{wr}^{i} and γ are the velocity and flight path angle of the parafoil respectively. Given that the velocity and flight path angle are nearly constant for most of the vehicle flight, so the time for a maneuver is basically based on the altitude above the target.

The position error, P_{err}, is defined in Equation 38. The position error is only in the X and Y directions since the vehicle will always be able to reach the correct Z location unless it is located below the target initially.

Equation 38: $$P_{err} = \sqrt{X_{err}^{\ 2} + Y_{err}^{\ 2}}$$

The X and Y errors are defined in Equation 39 and Equation 40. The error is basically the difference between the desired X position and the final X position. In addition, the Z axis also contributes to the X and Y final position error based on the assumption that the vehicle continues down the same path until it reaches the appropriate Z axis.

Equation 39: $$X_{err} = X_d - X_f - dt_{err}\cos(\gamma)\cos(\psi)$$

Equation 40: $$Y_{err} = Y_d - Y_f - dt_{err}\cos(\gamma)\sin(\psi)$$

The third error measurement in the cost function is the velocity error, which is shown in Equation 41. The velocity error is just as costly either direction, so the absolute value of the function is taken.

Equation 41: $$V_{err} = \left|V_d - V_f\right|$$

The weighting matrix is set so that the velocity error is weighted the heaviest and the position and time costs are weighted relatively evenly. This is set somewhat arbitrarily based on the idea that a higher velocity will cause the highest overall error in the flight or possible damage.

Once the cost algorithm was derived, the next problem was to implement the initial cost estimates. The helicopter has a relatively uncoupled system, so the solution to the optimal path problem was as big a challenge as the parafoil, where the states of the vehicle are coupled. The problem comes in that every position needs an initial estimate of the optimal cost, and if the numbers are off, it can lead to a faulty path.

The problem of initial cost estimation was addressed by dividing the position grid into smaller zones and enacting scenarios based on those zones. The zones were divided up as shown in Figure 25.

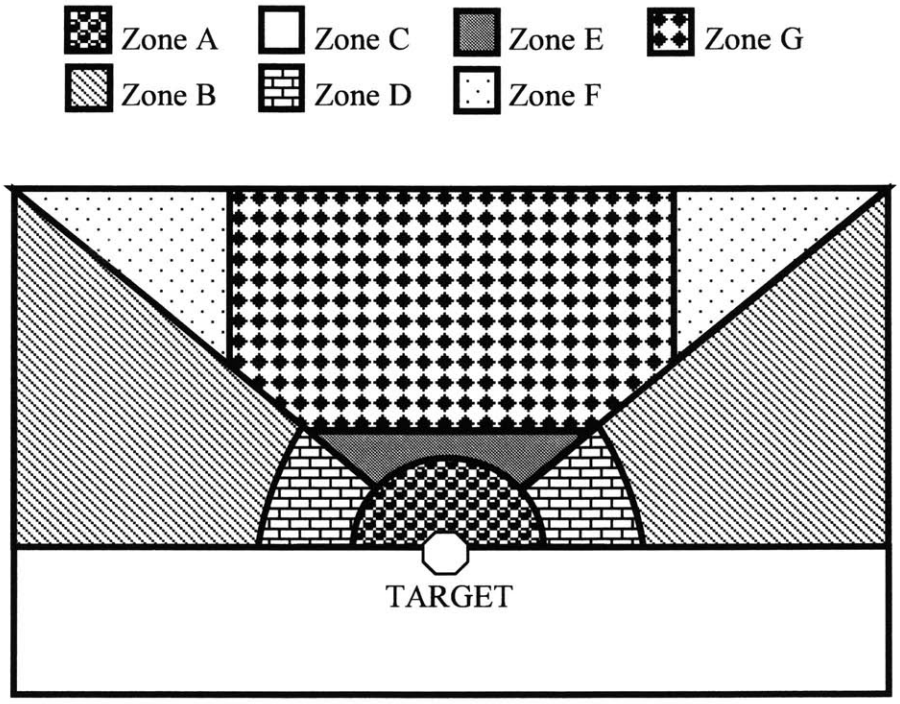


Figure 25: Division of the Position Grid for Initial Cost Estimates

The regions depicted in Figure 25 are a cross-sectional slice of the cylindrical positional grid that the cost matrix is based upon. The zones were divided based on the possible maneuvers the parafoil could perform given the distance and altitude from the target. The four main zones are Zone A, Zone C, Zones B and D, and Zones E, F, and G. Zone A is the tolerance region around the target state in which no maneuvers are necessary and that the cost is zero or based only on velocity errors. Zone C is the region where a maneuver cannot be made but has nonzero cost due to its distance from the target. The

cost function is applied directly to this area, since any movement would cause further loss of altitude and likely end in a crash.

The maximum glide distance that the parafoil can take divides the two remaining zones. This is calculated by taking the glide slope and extending it away from the target state. Zones B and D demark the area in which the vehicle will not reach the target state, even at minimum glide slope. This region is further divided based on the feasible sets of maneuvers the parafoil can make given its current location. Zone B is the area in which the vehicle can transition to the fastest descent rate before transitioning to the target trim state. Zone D is marked off from Zone B due to the fact that the vehicle will not have the altitude left to transition to another maneuver before reaching the target tolerance in Zone A.

Zones E, F, and G are marked off in a similar fashion. These zones demark the region where the vehicle will definitely reach the target state. The region is further divided into three zones. Zone E is the region where the vehicle doesn't have time to complete a single loop before hitting the target condition. The cost is then based on a simple maneuver to keep the parafoil within a close distance to the target. Zone G is the region where the parafoil has time to spiral to the final state, but doesn't have time to transition to an accelerated straightforward trim state before beginning the spirals. Zone F is the region where the parafoil can transition to a speedy descent, reach the appropriate altitude and position, then spiral to the target state.

Chapter 5: Parafoil DSP Board and Sensors

5.1 Parafoil Guidance Controller Board Selection

The parafoil guidance controller board's main purpose was to provide guidance to and receive information from the 68338 Persistor chip which runs the navigational components of the parafoil Sensor Avionics System (SAS). The requirements of an appropriate processor were broken up into several components: instructions per second, RAM, ROM, frequency, and accessories. In the next few subsections, the method of calculating each of the required values and components will be addressed.

5.1.1 Calculation of Frequency Required

The main program needs to run at a certain acceptable rate before the guidance and controller code will be effective in controlling and monitoring the parafoil system. Essentially the main components of computing an acceptable frequency are the sensor sampling rates and the system response rate. The acceptable frequency will be a frequency that is higher than the system response rate and limited on the high end by the sensor sampling rate. Any frequency higher than the sensor rate will not improve the response of the controlled system appreciably. Any frequency below the system bandwidth could lead to an unstable system and uncompensated errors in the controlled system.

The transfer function is then used to find the frequency response of the system with a bode plot, as seen in Figure 26.

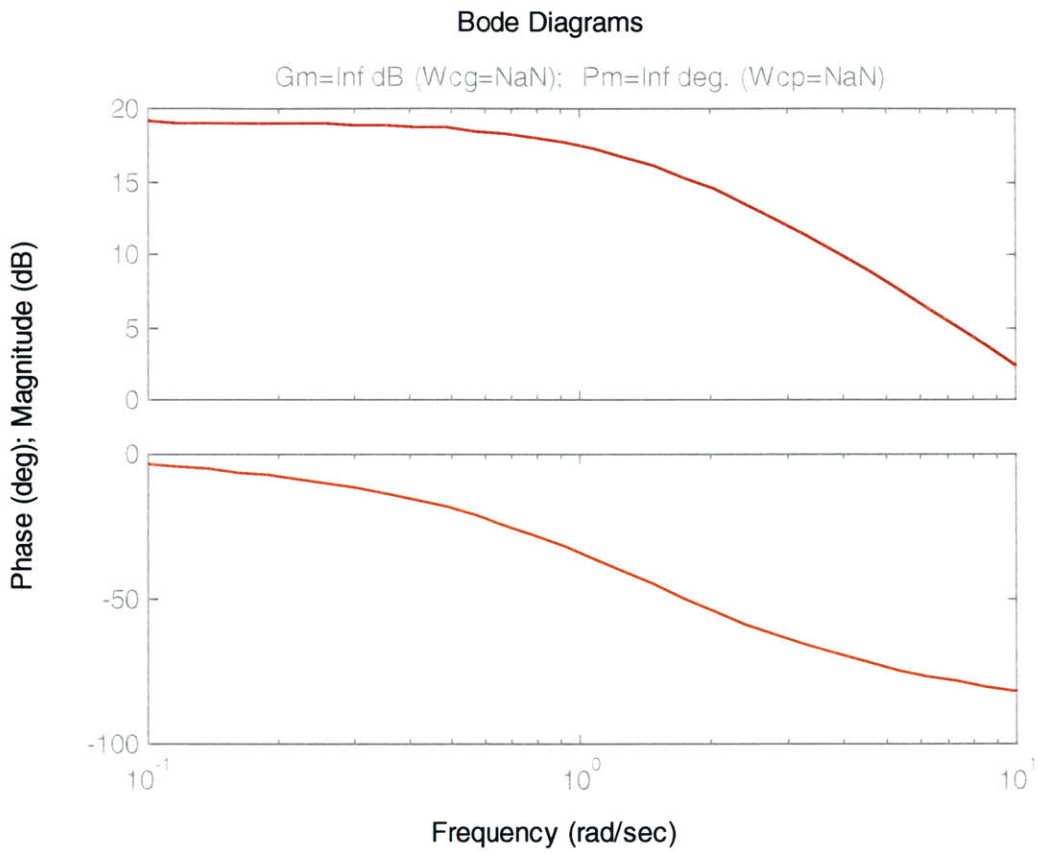


Figure 26: Bode Plot of Linearized System.

The sensor package selected contains a Global Positioning System card, a set of gyros for stability, and a compass. Each of the sensors has a respective sampling rate. The fastest sampling rate should be limited by the fastest sampling rate of the sensors. Anything updating over the fastest sampling rate of the fastest sensor would not improve the response of the controller system. The GPS card has an update rate of around 1 Hz. The gyros update at 70 Hz. Since the gyros are the fastest of the sensors and system bandwidth, they are the limiting factor on the program operation speed. Anything over 70 Hz would be unnecessary processing.

5.1.2 Calculations of an Acceptable Instructions per Second Rate

The power of a processor is measured by more than just speed. Some processors can complete more than one instruction per cycle. An instruction is a single basic process, like an addition or subtraction, which a processor completes as part of a program. A program can consist of hundreds if not thousands or more of these instructions. Instructions at the assembly level are different than the higher-level instructions that are coded in C programming, which could contain multiple assembly level instructions. The number of these instructions that can be completed per cycle, η, times the frequency of the processor,ϕ, (cycles per second) gives the instructions per second, ι, as seen in Equation 42.

Equation 42: $\iota = \eta \times \phi$

Since the speed that the program needs to operate is calculated in section 5.1.1 , the only part of Equation 42 required is the number of instructions the program will run.

A program can be broken into several parts for study. First, there is the initialization and basic code, which is run once to prepare for running the main loop. This code, which contains on the order of 3000 instructions, only has a slight bearing on the running of the program. The two sections that iterate, the controller main loop and the guidance main loop, have approximately 12000 and 3000 instructions respectively. In order to get a rough approximation of the processor IPS requirement, the two loops are assumed to iterate at the same speed, in serial. Operations, according to Equation 42, require 1.05 million IPS. The actual speed will be somewhat diminished by the communications rate between the pc and the controller.

5.1.3 Calculation of Read Only Memory and Random Access Memory Required

When calculating the amount of memory and storage space required, there are two concerns. The first concern is the amount of memory required to store the main program and its data files, the Read Only Memory or ROM. The second concern is the amount of memory required to run the program in realtime, the Random Access Memory or RAM. Both types of memory are requirements for a board to function as a main controller for the parafoil.

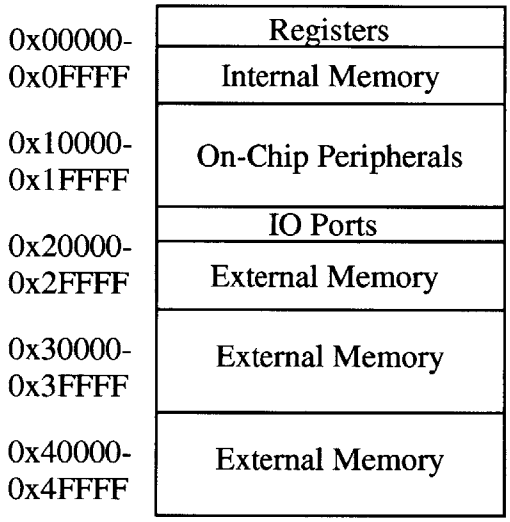


Figure 27: Memory Map of a Typical Microcontroller

The RAM is generally on the processor and possibly on a secondary chip depending on the required amount of memory. The RAM is erased every time that power to the chip is turned off or the chip is reset. The RAM functions as the working memory, and allows read, write, and execute permissions to the user. The amount of RAM required is set by the most amount of memory required, at any point in time, to run the program in real time. In the case of the parafoil controller, the program itself, approximately twenty kilobytes in size, is much smaller than the cost matrix, one to three

megabytes in size. The means that the minimum amount of RAM required, assuming that the whole cost matrix is utilized at one time in the main program, is on the order of one to three megabytes.

The ROM is generally on a chip that can be electronically "burned" with the main program and will allow only read and execute access to the information stored. There is a certain amount of ROM that comes with most microprocessors, microcontrollers, and even digital signal processors or DSP's. The program, which is around twenty kilobytes in size, is much smaller than the cost matrix, which is around one to three megabytes large. The overall ROM, like the RAM, is therefore controlled by the cost matrix size, and on the order of one to three megabytes in size.

5.1.4 Preferred Accessories

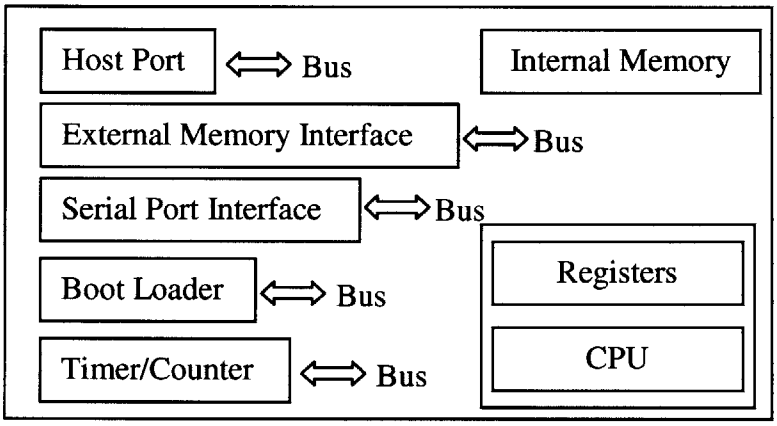


Figure 28: Peripheral Map of a Typical Microcontroller

In addition to the basic requirements for each chip and the overall circuit board, there are additional features on the processors that can reduce the number of additional circuits added to make the system function properly. In this case, the guidance system has to interact with a Motorola 68338 chip, which uses a Serial Peripheral Interface (SPI)

bus to communicate with other peripheral chips, devices, and sensors. Optimally, the guidance processor should contain or have the ability to communicate using the SPI bus without additional programming. This would reduce the amount of coding and additional chips required to make the system properly run.

5.1.5 Selection of the Processor Boards

Selection of the main guidance boards was based on all of the factors and requirements listed in sections 5.1.1 through 5.1.4 . Each board was also evaluated on ease of use, ability to interact with the simulation, availability, and cost. The basic capabilities of each of the chips are compared in Table 5.

Board Name	Manufacturer	Processor	MHz	MIPS	ROM	RAM	Cost
TMDS3200031	Texas Instruments	32-bit floating C31	50	25	16 MB	2K	$99.00
TMDS320006711	Texas Instruments	32-bit floating C6711	150	1200	64K	16MB	$299.00
Motorola	M5407C3	32-bit floating MCF5407a	162	257	2MB	32MB	$699.00
PC-104 Core	Ampro	Pentium II	150	150	HD	16MB	$800.00
486Core	Compulab, Israel	486	100	100	136MB	32MB	$1,200.00

Table 5: Table of Processors and Capabilities

The first evaluation board examined was the Texas Instruments TMDS3200031 evaluation kit. The evaluation board contained a 32-bit floating point C31 processor, which can be used for relatively complex operations and has been around for a number of years. The board has sixteen megabytes (MB) of read only memory (ROM), but only two kilobytes (KB) of onboard random access memory (RAM). Since the board only has a limited amount of RAM installed and the program requires one to three megabytes of memory, an additional board would be required to add on extra RAM. The chip itself will calculate twenty-five million instructions per second (MIPS). This board was by far the cheapest available, with a cost around one hundred dollars. The issue with the cost was that the addition of RAM cost another hundred and fifty dollars and required adding

and debugging a protoboard. The time constraints make the addition of a protoboard undesirable.

The next evaluation board examined was the Texas Instruments TMDS320006711 developer's startup kit (DSK). The digital signal processor (DSP) could easily handle the required workload, featuring a 32-bit floating point C6711 processor. Additionally, the board featured 128 KB of external flash ROM and 16 MB of external RAM with 64 KB of on-chip cache. The flash memory allows easy reprogramming of the ROM through a communications port on the card. In typical ROM chips, the memory is burned on a chip burning station before being placed on the card. The processor ran at 150 MHz and could execute up to eight instructions per cycle yielding 1200 MIPS. Since the processor had plenty of RAM, the program could be downloaded into the external RAM and executed without need of additional hardware. The tradeoff comes in that the board had to be programmed each time the board was reset; this could take a long time for a 3 MB file being transmitted over a serial port. The board cost around three hundred dollars and was readily available from several distributors.

After having sorted through the Texas Instruments processors, the next set of boards was the Motorola evaluation kits. The first Motorola kit researched was the M5407C3 evaluation kit. There were two versions of this kit available. The first kit had more memory and worked at a faster clock speed than the second. The faster kit featured a 32-bit floating point MCF5407a processor, which also ran at 150 MHz. It featured 2 MB of external flash ROM and 32 MB external RAM with 64 KB on-chip cache. The board offered almost enough flash ROM to not require an additional board and still keep

the program when resetting the board. The only major issue with this board was that the cost was relatively high at nearly seven hundred dollars. The second Motorola board was similar to the first, using the same processor, but had half the memory and ran at ninety megahertz. It cost one hundred less at around six hundred dollars. Additionally, it was not easily locatable.

The fourth board that was examined was the PC104 Core board, which was produced by Ampro. These boards are small pc's compacted onto miniature boards. They generally have the ability to stack together vertically to create a full system. The Core board is the main processor, with a 150 MHz Pentium II processor and a PC104 bus to expand to other boards. This board featured 16 MB of RAM and allowed the connection of a hard-drive to create a full computer system. Besides being larger than the other boards, this board cost a more at around eight hundred dollars, including a small hard-drive and RAM. The advantages include the unlimited space on the hard drive and the ability to use an operating system to run the programs in the first place. The software development would have been significantly easier for this setup.

The last boards that were selected were the 486Core boards, produced by Compulab in Israel. The 486Core boards were much smaller than the PC104 boards and yet had similar features. It featured a 486 processor running at up to 100 MHz, with a variety or other processors from which to choose. The 486Core boards also had memory upgradeable to 32 MB of DRAM, as well as allowing flash cards up to 136 MB in addition to the normal computer peripherals. This computer would have the ability to run an operating system as well as keep itself small. The main problem with this board was

cost. The 486Core board cost around twelve hundred dollars without the addition of peripherals.

Once the evaluation boards and processors were selected and researched, the next step was to select the appropriate processor for the application within constraints. The main constraint was cost, since all of the boards met the processing requirements. The budget was low enough that only a couple of the boards were able to qualify – both of the Texas Instruments boards were well within reason, while the Motorola boards were a hundred or more above the acceptable limit. The choice between the two Texas Instruments boards was made based on time restraints. Since the 6711DSK was available without significant hardware changes, it was selected. The hardware changes for the C31DSK were likely to cause significant time delays in debugging.

5.2 Parafoil Code Implementation

The parafoil optimal path-planning algorithm is what plans the optimal path of the vehicle from its current state to the target. The code was originally developed for the XCell-60 helicopters using aggressive maneuvering. This code was then adapted for use of the parafoil, with the addition of a DSP to the simulation. The code therefore required the adaptation for a confined memory and processor space, communication between the PC simulation and the path planning on the DSP, and adaptation of the code for the parafoil itself. Section 5.2 will describe the full implementation in detail.

5.2.1 Overview of the Parafoil Code Implementation

Since the full simulation took place on both a personal computer (PC) and on a digital signal processor (DSP), a significant amount of structural planning was required. The basic scheme is shown in the diagram in Figure 29.

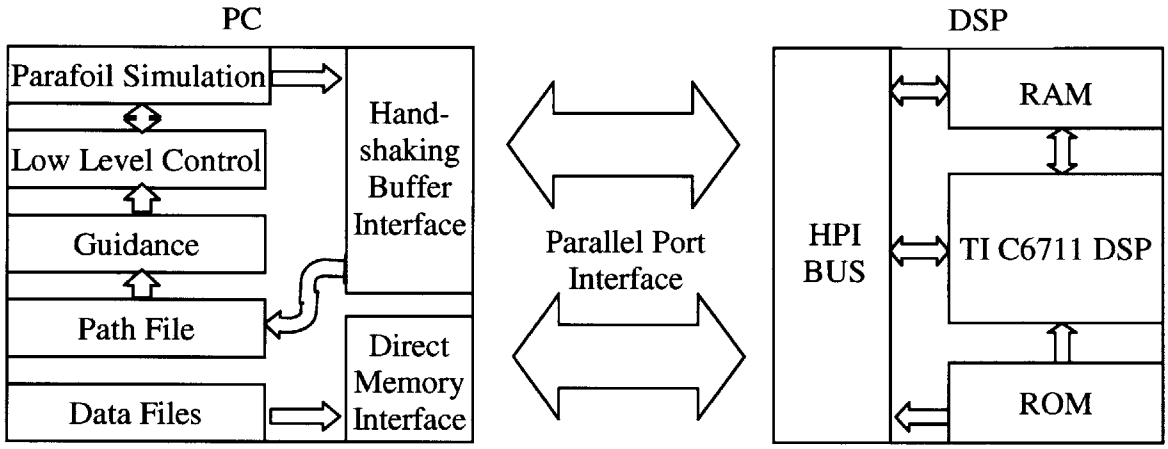


Figure 29: Diagram of General PC and DSP Communications

The personal computer contained the simulation of the parafoil dynamics, as well as the low-level control and guidance algorithms. The personal computer also contained the initial data files, including the cost matrix, the positional grid, the trim library, and the maneuver library. The creation of these files was discussed in section 4.3 . Additionally, the personal computer contained the binary code file for the digital signal processor and the PC's own main loop with communications functions. The personal computer coding and data flow will be described in much greater detail in section 5.2.2 .

The digital signal processor eventually contained all of the data files, the binary code file, which contained the communications protocols, and the initial boot code, which allowed the personal computer to initially communicate with the digital signal processor

board. The digital signal processor also contained the main path optimization code for the simulation. The code and communications of the digital signal processor will be described in greater detail in section 5.2.3 .

The communications, shown in Figure 29, are between a host, the controlling code, and a remote processor, the digital signal processor. The personal computer was the host, so it controlled the communications and flow of the information between itself and the digital signal processor. The information flow was divided between the simulation data flow, which used handshaking as a communications protocol, and the initialization data flow, which used the direct memory interface as its communications method. Communications between the personal computer and the digital signal processor was over a parallel port connection.

The two methods of communication used to transmit data between the personal computer and the digital signal processor are direct memory access and handshaking. Direct memory access, which is preferable for large data transfer, is essentially writing to and reading from an exact memory location on the digital signal processor. The method allows the host personal computer to write and read set amounts of data directly into the RAM of the digital signal processor or a peripheral device. A diagram of the direct memory access communications is shown in Figure 30.

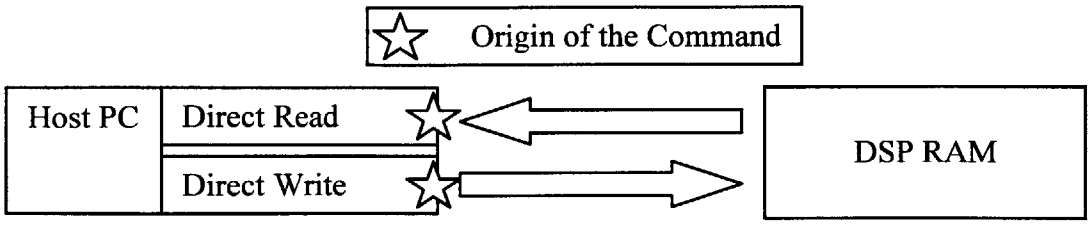


Figure 30: Direct Memory Access from Host PC to Remote DSP

The benefits of using this method are that there is low processing overhead for the data transfer and the data will be accessible at the same location every time. The problems

with this are that the destination address of the data must be known before writing it. Additionally, the digital signal processor must allocate a set space in memory for reading and writing. Setting aside a space in memory means that it cannot be dynamically allocated for storage of other information throughout the course of the program.

Handshaking, on the other hand, is a solution that allows a small amount of memory to be used for data transfer. A small buffer zone is set-aside for communications between the host and the remote device. The buffer zone, located on the digital signal processor, can be queried and written to by both devices. The use of this protocol requires that the host and the remote device await commands from each other before continuing to the next step. The buffer is divided into individually addressed words. Each word, made of 32 bits, is assigned a certain function. A typical handshaking buffer is shown in Table 6.

DSP Memory Address (Bytes in Hexadecimal)	Data Label	Size	Data Type
0x00000000	Buffer Data Word 1	4 Bytes	Unsigned Long
0x00000004	Buffer Data Word 2	4 Bytes	Unsigned Long
0x00000008	Buffer Data Word 3	4 Bytes	Unsigned Long
0x0000000C	Buffer Command	4 Bytes	Unsigned Long
0x00000010	Buffer Status	4 Bytes	Unsigned Long
0x00000014	Buffer Error Out	4 Bytes	Unsigned Long

Table 6: A Typical Handshaking Buffer

In an example handshaking case, the host would read the handshaking buffer until the status read that the remote digital signal processor was ready with some information. The DSP would then go into a wait mode to see when the status read that the host was done

processing or had new information for the DSP. The host, with its new information, would then read the data out of the data buffer, check the command buffer for any requests, and see if the error buffer had produced any errors. The host would process information and proceed to send more information to the DSP. The cycle would repeat until the processing was completely done.

The benefits of the handshaking approach are that each processor knows what state the other processor is in, the memory requirements are minimal, and the remote processor is able to communicate information and requests. The problems with handshaking are that it requires that only one processor is functioning at a time (or at least that only one can write into the buffer at a time), as well as a significant communications overhead for each bit of data transmitted. The processors can both function at once if the routine is implemented using hardware interrupts on either side. Additionally, sending data through handshaking requires care as to when each portion of the buffer is updated. If the status word is updated before the information is placed in the data words, then it is likely that the device that's cuing off of the status buffer will read in the information in the data buffers before the data has been written to them.

The progression of the code in the main data flow was implemented through the communications structures described above. The main data loop is shown in Figure 31.

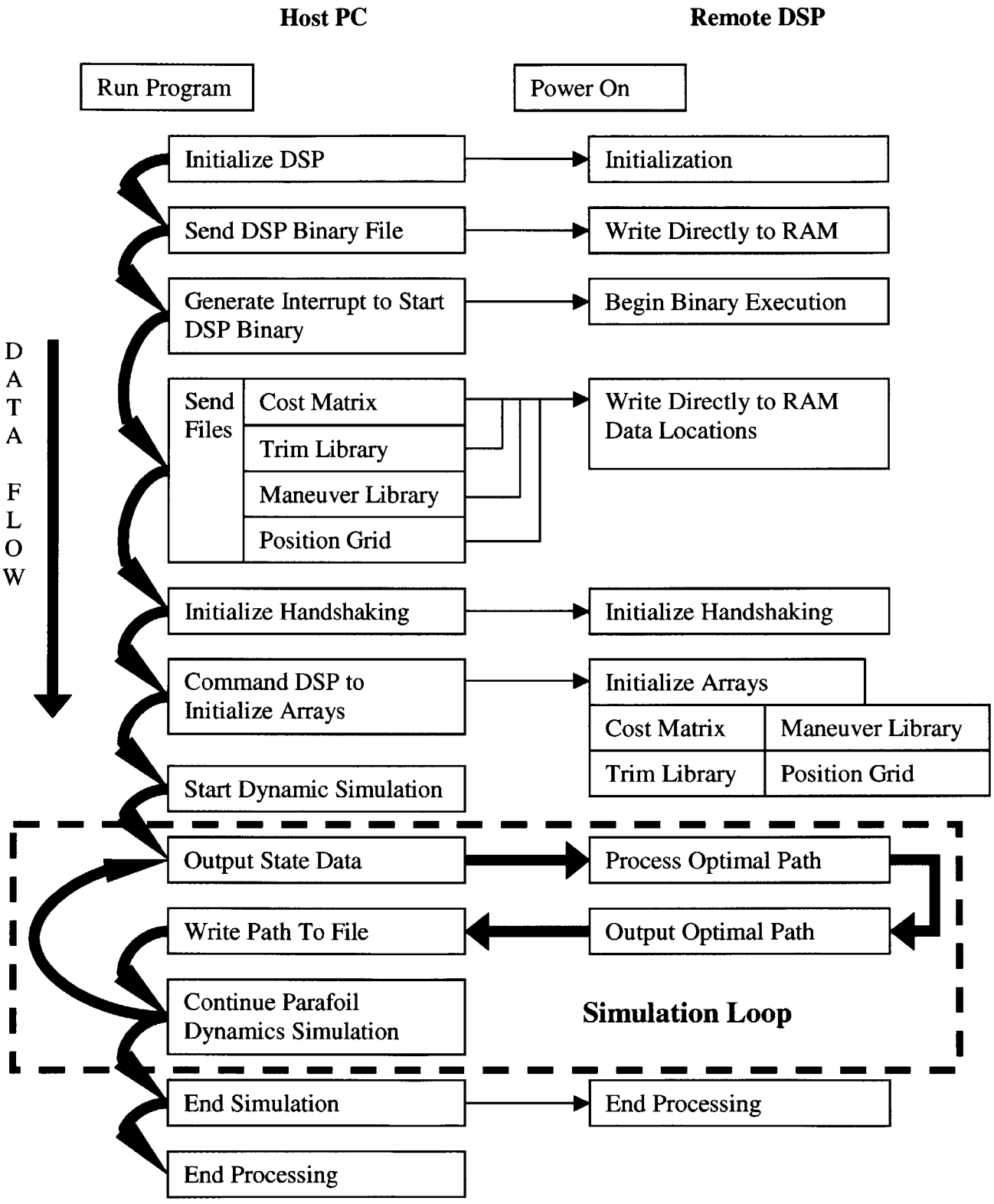


Figure 31: The Main Data Flow Diagram

The data flow diagram in Figure 31 shows the flow of data between the personal computer and the digital signal processor as it was executed on each of the chips. The bold lines are the main progression of the program, while the smaller lines indicate passing of data or commands. The smaller lines were executed before the program moved on to the next bold-faced line. The bold lines indicate where control of each process lay.

In the next part of this section, the general progression of the program will be described. The detailed description of each of the processing steps will be described in Sections 5.2.2 and 5.2.3 .

From the beginning of the program, the host personal computer was in control. The personal computer first commanded the DSP to open a communications link with it. Once the link was established, the host wrote a binary file directly into the executable section of the DSP's internal RAM. Writing the file into the executable space means that the processor will automatically begin running the binary program when commanded. The host then commanded the processor to begin running the executable by using an interrupt flag. The interrupt was programmed in the ROM to begin running the binary at a specific start address.

At this point, there are two executables running, one on the host and another on the remote processor. The binary on the remote processor does an initial memory allocation and then goes into a waiting loop until the host requests it to act. The host begins reading the cost matrix, the trim library, the maneuver library, and the position

grid from their respective files, and writing them directly into memory locations within the remote processor's external RAM.

Once the host finished transferring the files into the remote processor's external memory, the host then initialized the handshaking buffer and requested that the DSP verify that the handshaking buffer was functioning properly. Initialization was done by asserting a certain known value into the buffers and seeing if the DSP could read them. The verification was also done in the reverse direction as well. Once the handshaking protocol was properly working, the host would then command the remote processor to commence initializing arrays. The host and remote processor were both using handshaking to communicate at this point. The DSP would then initialize the main arrays for use in the optimization function, including the cost matrix, the trim library, the maneuver library, and the positional grid.

Once the arrays were initialized, the main program loop could begin. The main program loop was a combination of the simulation on the host PC and the path planning algorithm on the remote DSP. In this case, control of the data flow switched back and forth between the host and the remote processor. The dynamic simulation began running first. To simulate sensors, the simulation would halt every few seconds, output the state, and wait for a new path to be calculated. The DSP would calculate the new path based on the input state and the target state and then output it to the host PC. The host PC would output the path to a file and then continue the simulation. These commands defined the main loop.

When the simulation was finished, and the target reached, the simulation would end and send a signal to the DSP to finish processing. The main program would then release all dynamic memory and end processing as well.

5.2.2 The Personal Computer Interface

This section will cover the implementation of the code on the host personal computer. The personal computer contained the initialization files, communications protocols and dynamic simulation with low-level controls as discussed in Section 5.2.1 . This section will delve into the details of the code implementation in sections broken down by function. The functions include communications, interface with the simulations, file input and output, and the limitations of the system implementation.

The communications portion of the host personal computer implementation is broken down into two main sections: handshaking and direct memory access. The personal computer communicated with the remote processor through the parallel port, which required basic level drivers to implement. A basic file library was included with the Texas Instruments C6711 Developer's Starter Kit (DSK), which allowed high-level communications with the DSP without needing to write drivers for the parallel port. These high-level commands, as listed in Table 7, allow easy access to the DSP board without having to deal with the low-level protocols over the serial port. These commands and definitions are taken directly out of the C6711 data guide.

dsk6x_open()	Open a connection to the DSK
dsk6x_close()	Close a connection to the DSK
dsk6x_reset_board()	Reset the entire DSK board
dsk6x_reset_dsp()	Reset only the DSP on the DSK
dsk6x_coff_load()	Load a COFF image to DSP memory
dsk6x_hpi_open()	Open the HPI for the DSP
dsk6x_hpi_close()	Close the HPI for the DSP
dsk6x_hpi_read()	Read DSP memory via the HPI
dsk6x_hpi_write()	Write to DSP memory via the HPI
dsk6x_generate_int()	Generate a DSP interrupt

Table 7: High Level Commands for the Host PC to Control the DSP

Communications with the DSK must first initiate through a "dsk6x_open()" command. This sets up initial communications ports through which the DSK and the PC communicate. Once the two platforms are communicating, the PC can now reset the board with the "dsk6x_reset_board()" command and then load the main binary file into the main RAM of the remote processor by using the "dsk6x_coff_load" command. The DSP's main compiler writes the main binary file, which is in the Common Object File Format (COFF). The compilation and coding of the DSP's code will be discussed in Section 5.2.3 .

The COFF file is written to a specific memory address in the DSP's internal RAM such that the file will begin execution when commanded by an interrupt. An interrupt is a register on a processor that is continually sampled. If one of the pins or flags changes state, the hardware sends a signal to the main processor that an interrupt request has been made. The processor then takes action according to pre-programmed cases. Interrupts can have priorities such that an interrupt can take precedence over another one. Typically

a processor will run its main code until an interrupt is signaled, then it breaks from the main program, executes another set of code, and then returns to processing the main code. The interrupts can stack based on priority, so that multiple devices can be put into a prioritized hierarchy of data-requests and other functions. The benefit of an interrupt is that the processor doesn't have to sit and wait for another device to send information before it can continue processing. In this case, the interrupts were used to initially run the main code. Additionally, digital signal processors also run in idle mode, without a main code loop, until they receive an interrupt to run specific code. Even the main control loop, in this case, is an interrupt function with the lowest priority.

Once the main control loop resets the DSP, communications between the two processors can take place at a higher level. The communications between the remote processor and the host personal computer can be done through the Host Port Interface (HPI) on the DSK. The HPI will be discussed in more detail in Section 5.2.3 , but essentially it is a hardware device that allows direct communication with both the external RAM on the DSK and the DSK's internal RAM. By opening a communications link using the "dsk6x_hpi_open()" command, the host PC can write directly to the memory of the DSK through the HPI.

Once communications with the HPI was established, the choice of communications was between using the handshaking buffer or the direct memory writing, as discussed in Section 5.2.1 . Essentially direct memory writing is the basis for both forms of communication, but only in the limited sense for the handshaking approach. Direct memory writing was used for large data transfers due to the low communications

overhead. Handshaking was used for smaller data transfers because it required only a small amount of memory for passing information back and forth.

The choice of communications was obvious for large files like the cost matrix, which reached sizes of one to three megabytes. These files were read into memory buffers through typical file input and output streams in the C coding language. These buffers were then written directly into the external RAM of the DSK using the "dsk6x_hpi_write()" command. The buffers were written directly into pre-allocated sections of the DSK's memory, which were allocated once the main binary file was written into internal RAM and run.

All of the data written to and from the DSK's memory using the "dsk6x_hpi_write()" command must be in the unsigned long format. This means that a floating point or decimal number would be truncated unless the data was modified. The way around the direct casting method is to multiply the floating point number by a fixed constant like one thousand. Once the data is transferred into the memory of the DSP, it is read into the DSP's memory and converted back into a float by casting the number after dividing it by the same constant. In this case, the constant would save three decimal places from truncation or rounding errors. For the cost matrix, which requires higher accuracy, the constant was ten million, which preserved seven decimal places. This is also the case with the "dsk_hpi_read()" command used in the handshaking buffer's communications.

The handshaking buffer had to be setup so that it could transfer the state data for both the vehicle and the target to the DSP, as well as receive the path information from the DSP during the simulation. The size of the path was unknown, but the size of the two

states were both twelve words, where a word is four bytes on a thirty-two-bit processor such as the C6711. Since limiting the path to twenty-four words was certainly not a reasonable solution, another idea was proposed. The path could be communicated to the PC one line at a time and written directly to a file. This would eliminate the need for an expandable handshaking buffer, but still allow the transfer of all of the data from the DSP.

The vehicle and target states were received from the output of the vehicle simulation. The simulation code will be discussed in more detail in Section 5.3 . Each of the states contained twelve words worth of data. On top of the states, the basic generic buffer was used for commands, status, error handling, and debugging data. The state output buffer was finalized as shown in Table 8.

DSP Memory Address (Bytes in Hexadecimal)	Data Label	Size	Data Type
0x00000000	Buffer Data Word 1	4 Bytes	Unsigned Long
0x00000004	Buffer Data Word 2	4 Bytes	Unsigned Long
0x00000008	Buffer Data Word 3	4 Bytes	Unsigned Long
0x0000000C	Buffer Command	4 Bytes	Unsigned Long
0x00000010	Buffer Status	4 Bytes	Unsigned Long
0x00000014	Buffer Error Out	4 Bytes	Unsigned Long
0x00000018	Vehicle State 1	4 Bytes	Unsigned Long
0x0000001C	Vehicle State 2	4 Bytes	Unsigned Long
0x00000020	Vehicle State 3	4 Bytes	Unsigned Long
0x00000024	Vehicle State 4	4 Bytes	Unsigned Long
0x00000028	Vehicle State 5	4 Bytes	Unsigned Long
0x0000002C	Vehicle State 6	4 Bytes	Unsigned Long
0x00000030	Vehicle State 7	4 Bytes	Unsigned Long
0x00000034	Vehicle State 8	4 Bytes	Unsigned Long
0x00000038	Vehicle State 9	4 Bytes	Unsigned Long
0x0000003C	Vehicle State 10	4 Bytes	Unsigned Long
0x00000040	Vehicle State 11	4 Bytes	Unsigned Long
0x00000044	Vehicle State 12	4 Bytes	Unsigned Long
0x00000048	Target State 1	4 Bytes	Unsigned Long
0x0000004C	Target State 2	4 Bytes	Unsigned Long
0x00000050	Target State 3	4 Bytes	Unsigned Long
0x00000054	Target State 4	4 Bytes	Unsigned Long
0x00000058	Target State 5	4 Bytes	Unsigned Long
0x0000005C	Target State 6	4 Bytes	Unsigned Long
0x00000060	Target State 7	4 Bytes	Unsigned Long
0x00000064	Target State 8	4 Bytes	Unsigned Long
0x00000068	Target State 9	4 Bytes	Unsigned Long
0x0000006C	Target State 10	4 Bytes	Unsigned Long
0x00000070	Target State 11	4 Bytes	Unsigned Long
0x00000074	Target State 12	4 Bytes	Unsigned Long

Table 8: State Input Handshaking Buffer Allocations

Although the modified handshaking buffer is larger than the typical handshaking buffer, it allows a moderate-sized amount of data to be transferred without the static memory sections required with direct memory access. All of the data had to be multiplied by a constant in order to maintain the accuracy of the states. The vehicle and target states, which were defined in Section 4.1 , will be discussed in more detail in Section 5.2.3 .

The path file, which was read using the handshaking buffer, was written to a file one line at a time. The path statement lines were defined by their respective identification number, the expected state of the vehicle, the primitive type of the vehicle (trim or maneuver), the identification number of the primitive, the time into the path, the delta time for each step, and the target flag. This fit easily within the required thirty-word buffer, with many to spare. The path output buffer allocations can be seen in Table 9.

DSP Memory Address (Bytes in Hexadecimal)	Data Label	Size	Data Type
0x00000000	Buffer Data Word 1	4 Bytes	Unsigned Long
0x00000004	Buffer Data Word 2	4 Bytes	Unsigned Long
0x00000008	Buffer Data Word 3	4 Bytes	Unsigned Long
0x0000000C	Buffer Command	4 Bytes	Unsigned Long
0x00000010	Buffer Status	4 Bytes	Unsigned Long
0x00000014	Buffer Error Out	4 Bytes	Unsigned Long
0x00000018	Path ID	4 Bytes	Unsigned Long
0x0000001C	Expected Vehicle State 1	4 Bytes	Unsigned Long
0x00000020	Expected Vehicle State 2	4 Bytes	Unsigned Long
0x00000024	Expected Vehicle State 3	4 Bytes	Unsigned Long
0x00000028	Expected Vehicle State 4	4 Bytes	Unsigned Long
0x0000002C	Expected Vehicle State 5	4 Bytes	Unsigned Long
0x00000030	Expected Vehicle State 6	4 Bytes	Unsigned Long
0x00000034	Expected Vehicle State 7	4 Bytes	Unsigned Long
0x00000038	Expected Vehicle State 8	4 Bytes	Unsigned Long
0x0000003C	Expected Vehicle State 9	4 Bytes	Unsigned Long
0x00000040	Expected Vehicle State 10	4 Bytes	Unsigned Long
0x00000044	Expected Vehicle State 11	4 Bytes	Unsigned Long
0x00000048	Expected Vehicle State 12	4 Bytes	Unsigned Long
0x0000004C	Primitive Type	4 Bytes	Unsigned Long
0x00000050	Primitive ID	4 Bytes	Unsigned Long
0x00000054	Time	4 Bytes	Unsigned Long
0x00000058	Delta Time	4 Bytes	Unsigned Long
0x0000005C	At Target Flag	4 Bytes	Unsigned Long
0x00000060	Unused	4 Bytes	Unsigned Long
0x00000064	Unused	4 Bytes	Unsigned Long
0x00000068	Unused	4 Bytes	Unsigned Long
0x0000006C	Unused	4 Bytes	Unsigned Long
0x00000070	Unused	4 Bytes	Unsigned Long
0x00000074	Unused	4 Bytes	Unsigned Long

Table 9: Handshaking Buffer Allocation for Path Output

The Path ID is essentially just an internal line numbering system that reflects which line of the path is being referenced. The Expected Vehicle State is calculated by the DSP based on the maneuver and trim libraries. The Primitive type identifies whether the vehicle is in a trim state or a maneuver. The Primitive ID tells which trim or maneuver the path is executing. Each line of the path takes place over a certain change in time, and at a certain time period into the path. Both the time and the change in time are labeled in the path buffer.

Although this was a successful implementation of the PC's algorithm, it does have limitations based on the devices and the coding methods used. As mentioned before, the data transmitted between the two devices has a limited accuracy based on the multiplying constant. Additionally, the sizes of the files are limited based on the amount of memory allocated statically in the digital signal processor board. The processors themselves must work at a high enough speed so as to update the information and paths at real-time speeds. Additionally, data flow rates are limited through the parallel port connection. This might become an issue if the low-level controller for the simulation were to be implemented on the digital signal processor board instead of on the host PC.

5.2.3 The Discrete Signal Processor Path Planning Implementation

The digital signal processor, unlike the personal computer, has limited resources. The digital signal processor contained the path planning algorithm, the communications code, and stored data for the main path-planning algorithm to access. The path planning code from the XCell-60 helicopters had to be adapted not only to the parafoil, but also for the digital signal processor platform. The aspects of the DSP code will be approached from the functional perspective, as in Section 5.2.2 . The basic topics will be the

compiler, data storage and retrieval, communications issues, and the main path-planning algorithm.

The compilation of all code for the DSK was done on Code Composer Studio, which is the proprietary compiler included with the Texas Instruments C6711 DSK. The files were written in the C language and compiled into a binary COFF file to be sent over by the host PC. The compiler is relatively complex and will allow for various optimizations for speed and size of code, since both are generally issues for digital signal processors.

When working with the compiler, there are multiple types of files that can be created, including ones that setup memory allocation and interrupt routines. The memory allocation is setup with a "filename.cmd" file. The file allocates where the heap, the memory set aside for dynamic allocation later, is addressed. Additionally, it sets up divisions of the address space by label, length, and starting position. The "cmd" file then allows the programmer to assign the various registers and buffers into the divisions of the address space. In this case, allocations were made for the cost matrix, the trim library, the maneuver library, and the position grid. These were each allotted their own division of the address space, which setup the space for the host pc to write the appropriate file into.

The "vectors.asm" file sets up the interrupt locations. This, however, can be done at a much higher level in the compiler options. This file essentially assigns a particular address to run a program when an interrupt flag is raised. This file is rewritten from the options chosen in the compiler. The memory allocations can also be done simply in the menus of the compiler as well.

Once the data sections were assigned in the compiler, the main DSP loop also had to note the data sections. This was done using the "#pragma DATA_SECTION" command. This command allows the DSP to realize that there is an assigned data section with a particular label. This also allows the DSP code to access the data section as an array based at the address of the data section. Since the data is written in an unsigned long format, the data must be read into memory and converted to floating point. Additionally, the data must be adjusted for the constant multiplied in earlier to save the decimal places from truncation. This could be accomplished within the same block of memory, but since there was enough free space to accommodate a simpler method, the data was simply read into a floating-point array and modified. Arrays were formed for the cost matrix, the maneuver library, the trim library, and the position grid. This allowed for easy access to the data from the DSP main code.

The communications buffers for handshaking were set-aside with the same data sectioning method. The buffers could then be written to and read from on both sides of the connection. The data in the data buffers was required to be read in an unsigned integer format, so all of the data passed into the buffers had to be converted in the same method used when sending the data to the DSK. Additionally, the status of the register also had to be updated after the data had been placed in the appropriate buffers so as to assure the proper transfer of all of the data to the host PC.

The path-planning algorithm required significant changes from the helicopter code. The changes to the code based on the parafoil adaptation will be addressed first, and then the changes to the code based on the DSK adaptation. The parafoil also

required the updating of the cost function as well as the data input files and cost matrix as described in Section 4.3 .

The cost function was updated such that the cost was based not only on the X and Y positions, but also the difference in velocity, and the projected Z effects on the X and Y coordinates. Each of these was weighted based on the associated cost of an error in each case. The weights were selected in a somewhat arbitrary fashion and could be optimized to produce the type of results desired based on whether position or velocity errors were more costly.

Once the cost function was updated, the iterative optimal cost selection algorithm was implemented based on the helicopter selection algorithm described in 3.2.4 . The vehicle and target states were input from the handshaking buffer after the host PC transferred them. The path-planning algorithm took the initial state and the target state and iterated through each of the possible maneuvers. The maneuver with the optimal cost was the one where the sum of the cost of the maneuver and the cost matrix value at the end of the maneuver were the lowest. The diagram in Figure 32 shows the iteration process for the DSP code.

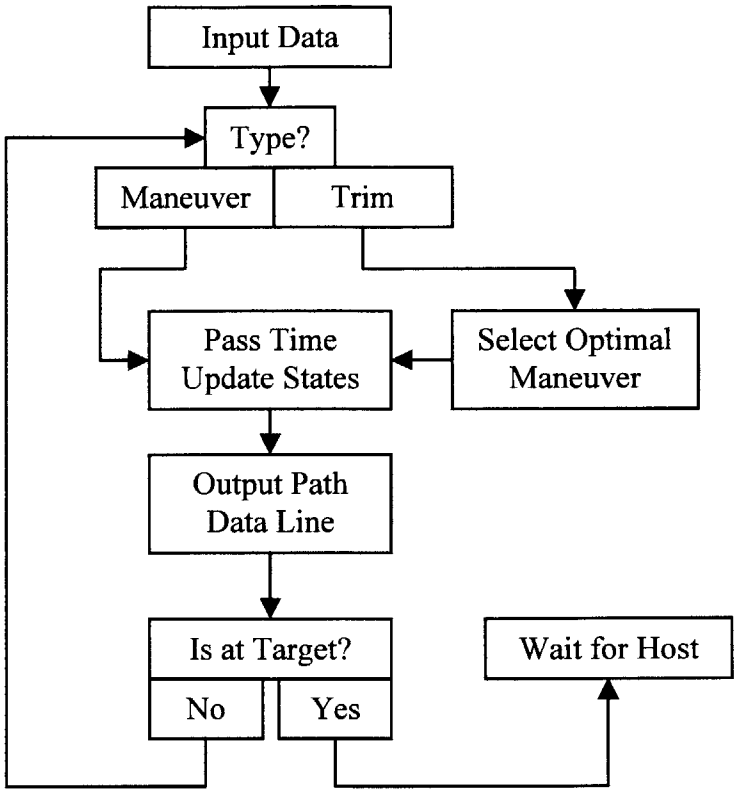


Figure 32:DSP Path Optimization Data Flow Chart

Once the first maneuver was selected, then the optimal path algorithm output the maneuver identification number and the change in states based on the maneuver library data. The data was updated in small time segments determined by a maximum time-step constant. Each of the time segments was sent back to the host pc to pass into the path output file. The iteration continued until the maneuver was completed. The next maneuver or trim state was then selected based on the resulting state values. Once the vehicle reached a spherical tolerance region around the target, the simulation ended and output the final values with the "at target" flag set.

Once the iteration for path planning was finished, the program went into a wait state to see what the host next requested. The host then either requested that the path

DSP end its waiting and finish processing or that the DSP process the next round of states for the simulation.

5.2.4 Results of the Path-Planning Algorithm

The path optimizations for the parafoil were relatively simple in nature. The path-planning algorithm was fed a particular state of the vehicle and a target state and asked to output the path-planning file. This method demonstrates the output of the path without counting on the simulation to follow the path closely. A basic case is shown in Figure 33.

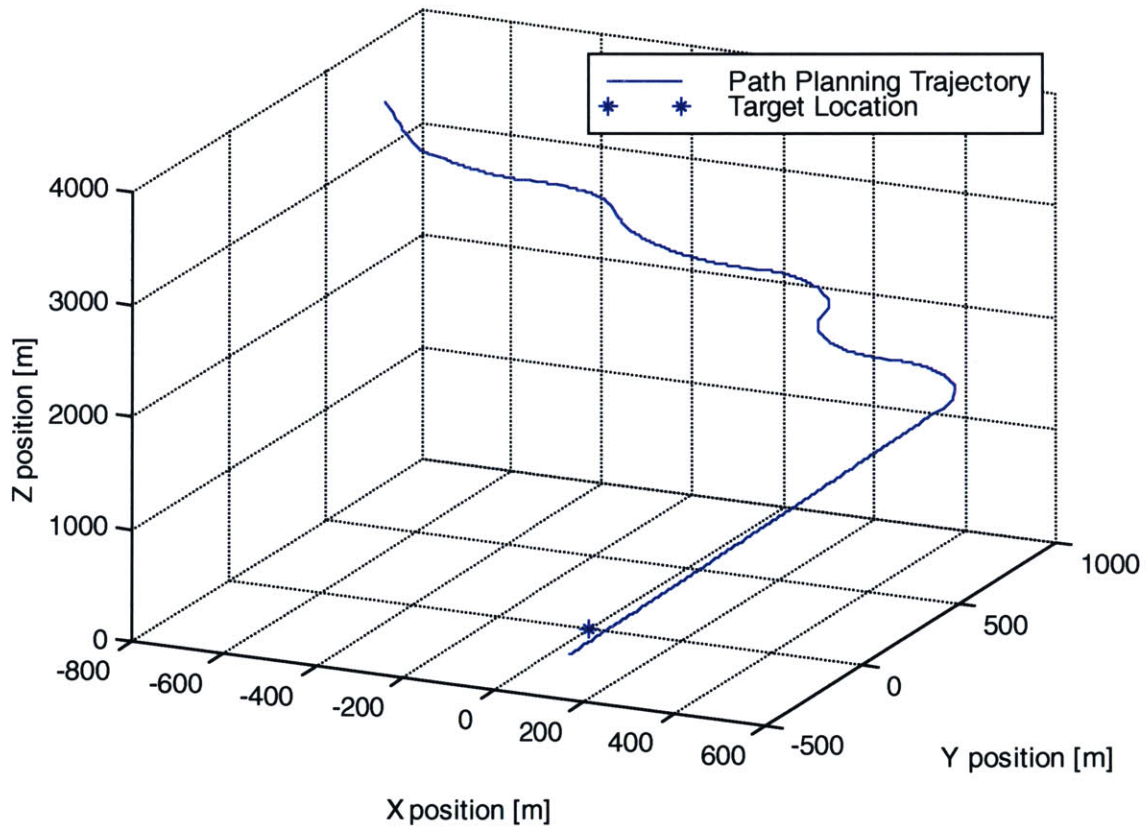


Figure 33: Basic Path-Planning Case

The figure in Figure 33 shows the optimal path of the parafoil assuming that the target state is located below it, well within the reach of the vehicle. The assumption is that the vehicle has a set glide slope.

The vehicle does not reach the exact target in this case because it has reached the sphere of tolerance around the target. At that point, the vehicle can implement its final maneuver or at least is within a short distance of the requested target. The target in this case, and in all cases of the parafoil implementation is a final trim state that allows the parafoil to implement its final landing maneuver. The maneuver itself is outside of the scope of this project as there have been many studies on the exact nature of landing in much more depth than is possible here.

A second case is presented in Figure 34. In this case, the parafoil was just in reach of the target state, and so no spirals or adjustments are seen in the path. The only maneuver is to increase the velocity of the vehicle, so as to minimize time to reach the target site.

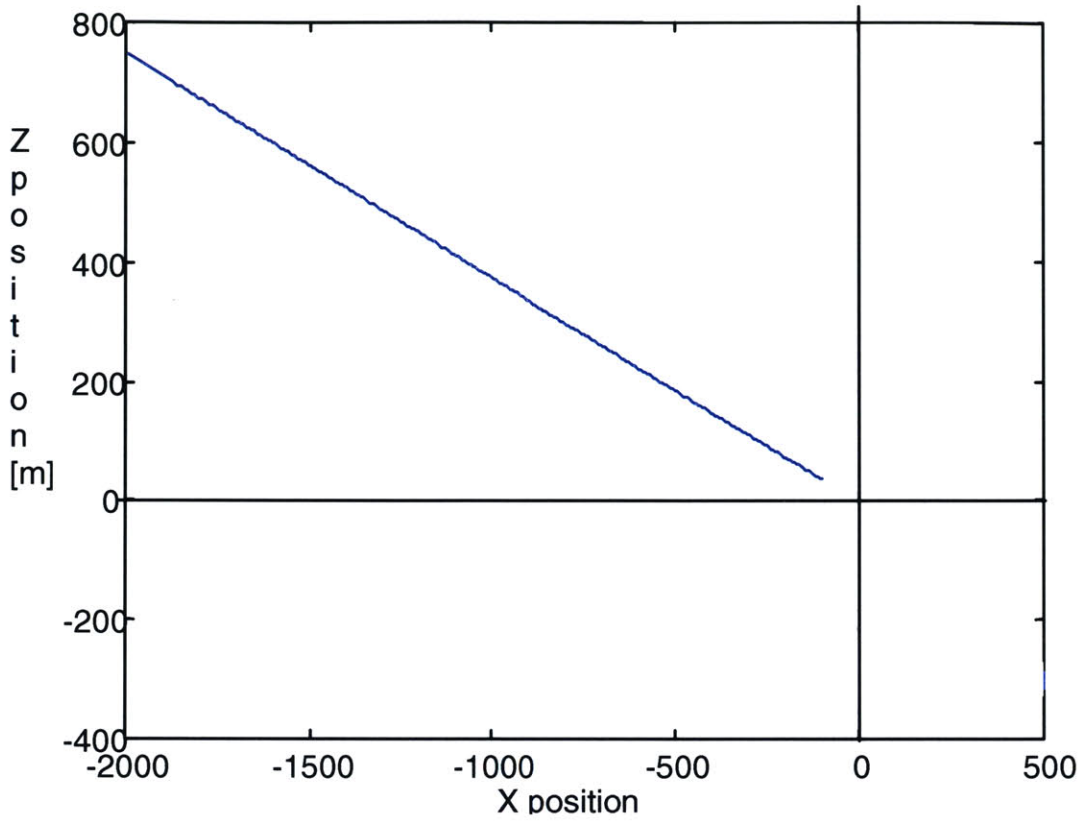


Figure 34: Basic Path-Planning Case with Boundary Target

The vehicle ends up following a straight trajectory towards the target after the initial maneuver to adjust to the correct orientation.

A third case is when the target is out of range of the vehicle. The case ends up being similar to the boundary case just presented, with the exception of the fact that the vehicle will never get within tolerances of the target. This means that the vehicle will never be at the target. This case is demonstrated in Figure 35.

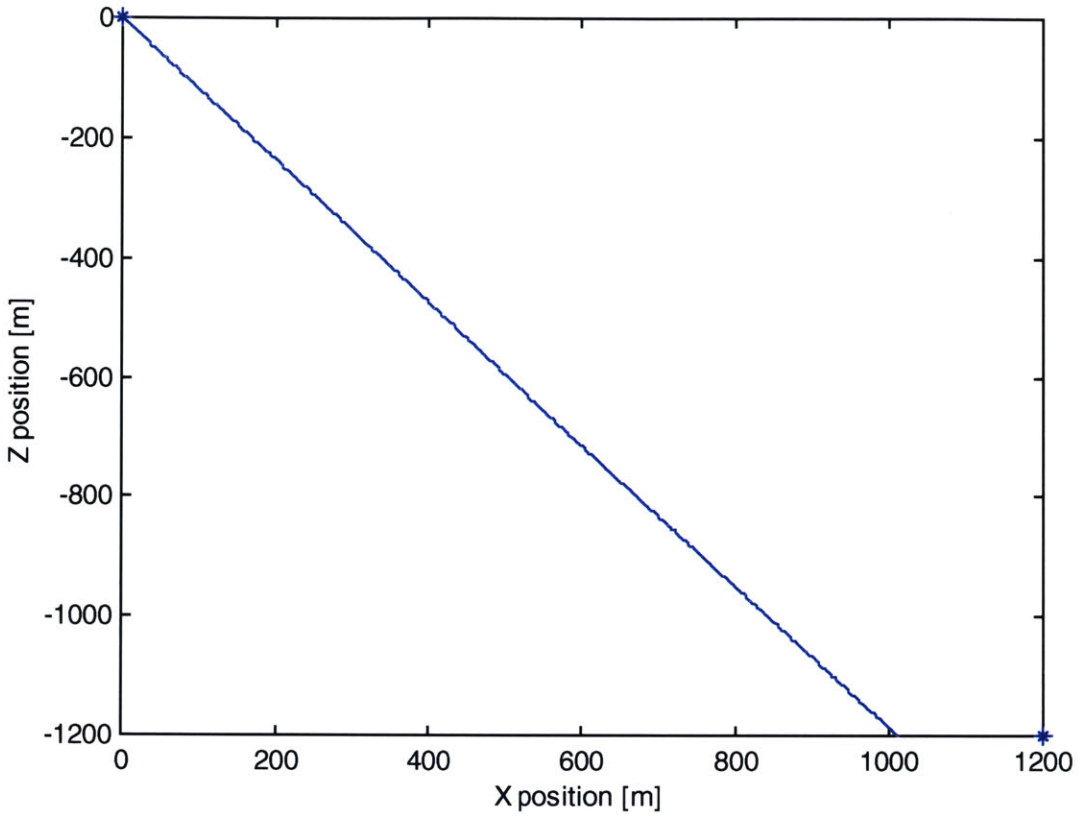


Figure 35: Basic Path-Planning Case with Target Out of Range

This exception is handled by stopping the vehicle at the proper altitude and calling that the closest that the vehicle will come to the target. The assumption is that coming in below the target is a highly undesirable effect as it could lead to some catastrophic failures.

These three cases presented above constitute the range of possible cases that the path-planning optimization would have to consider and handle. The code had to be modified in each case such that the parafoil function would work due to its dynamic model being significantly different than a helicopter. The tolerances are also a bit larger because of the reduced maneuverability of the parafoil.

5.3 Parafoil Simulation Implementation

The dynamic simulation was coded using the model described in Section 4.1 . The parafoil simulation was implemented on the DSP using Code Composer Studio, Texas Instruments' proprietary compiler that will output COFF files for the DSP. The implementation will be broken down into segments, based on function and then described. The basic functional segments include: the integration function, the interaction with host program, the input and output of data, the outputs of the simulation without controller, the controller, and the simulation outputs with the controller.

The integration function was a simple Newton-Euler function. The Newton-Euler function had a fixed time-step and iterated through the function by evaluating the integrals at each time step, multiplying by the time step, and then adding the value to the initial value of the variable. This method can be seen in Equation 43.

Equation 43: $X = X + \dot{X} * dt$

The simulation was setup to run completely independently of the main host program on the PC. The simulation was then adapted to take inputs and output data in order to interact with the host program. The interactions took place on two levels. The first level was that the host simulation had to call the simulation function to run it and wait for a new output. The second level was that the host had to output a path file that the simulation could then read; the simulation also had to output the vehicle state such that the host program could read it.

Calling the function could be done in two ways. The simulation could be initially started and keep running while the host interacted with the DSP, or the simulation could only run for short periods of time when a new vehicle state was needed.

In this case, the simplest solution, running the program for short durations, was implemented.

The data to be passed had to be split into appropriate segments. The input of the data was the path data file. This data file, written by the host program from data passed by the DSP, contained the information as shown in Table 10.

ID	State 1	···	State 12	Prim Type	Prim ID	dt	t	At Target?
1	.125	···	.5	0	23	0.5	0.5	0
...	...		...	...	...	...	...	...

Table 10: Path File Data Structure

The path input file contained the path, broken into time segments, with the projected state for each segment. Additionally, the path file contained the primitive type, the primitive identification number, the time step, the time at the beginning of each segment, and an identification number on each line.

The path file was read into the simulation and used to guide the vehicle along the path. The projected states from the path file were used as the trajectory that the vehicle should follow.

The dynamic simulation output had to include the state of the vehicle as well as the current primitive type and primitive number. The state output file layout is shown in Table 11.

State 1	···	State 12	Prim Type	Prim ID
.125	···	.5	0	23

Table 11: State Output File Layout

The end of the simulation then signaled the host program to read the state file and continue interaction with the DSP. The output of the simulation can be seen in Table 12.

At Target?
0

Table 12: Simulation Executable Output

The simulation would also signal when the simulation had hit the target or at least hit the altitude of the target. This would signal the host and therefore DSP to end processing and end the full simulation.

The simulation, after having read the path file, would then follow the trajectory. The trajectory was commanded, step-by-step, by a function that set the target values for each time step in the integration function. Once the target was set, the controller would command the vehicle to the target state.

5.4 Theoretical Implementation of the DSP with the Sensor Package

The main goal of this project is to serve as a basis for implementing the path-planning algorithm on a real vehicle. In order for this to occur, the following must occur: The vehicle must be setup and wired with a sensor and motor controller system; the Persistor chip, which controls the sensor and motor interactions, must be programmed to interact with the DSP over the Serial Peripheral Interface Bus (SPI Bus); the DSP code must be updated to include the interactions with the Persistor, as well as to adjust for the sensor inaccuracies and fewer state feedback; the controller and low-level guidance algorithm must be placed on either the DSP or the Persistor; and the DSK board must either be adapted to run on batteries or a new board be developed that runs on batteries

and possibly takes up less space. Once all of this occurs, the algorithm will be ready for testing on a real system.

Chapter 6: Results

The results of the simulations were broken down into sections as follows: the initial components, including path planning, parafoil dynamics model, and controller; the addition of components, the controller with the dynamic model using path planning; the performance of the overall system, including simulation versus real-time estimates, as well as endpoint accuracy.

The results from each of the initial components will be presented first. The path-planning algorithm took significant work to get functional. The path-planning algorithm requires that the initial cost estimates be within certain error tolerances of the actual best-case costs before iterating will improve the performance. In several attempts, the path-planning algorithm would produce unexpected results due to faulty weighting of the cost matrix. Essentially the cost to maintain a spiral was lower than the cost to head towards a target. This lead to a large position error at the end and only one transition between trim states.

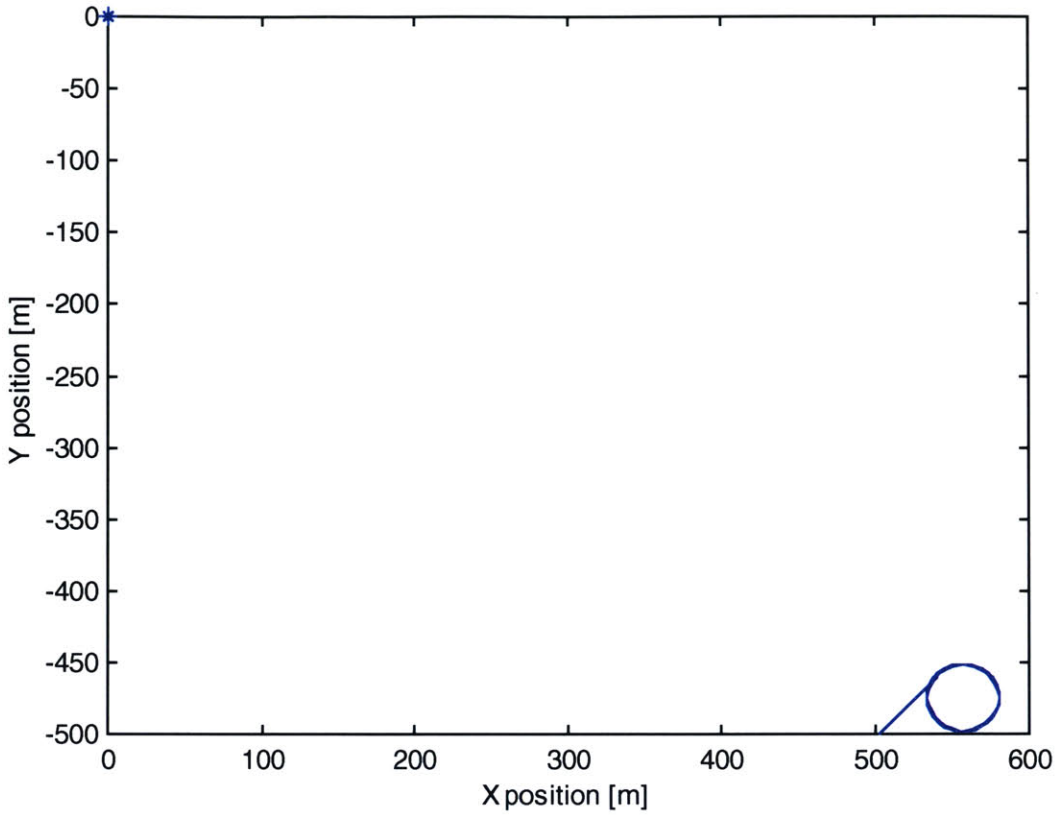


Figure 36: Top-Down View of Faulty Parafoil Path

Additionally, as can be observed in Figure 36, the path-planning algorithm doesn't even select the proper direction due to the faulty weighting and initial cost estimate errors. The circled area is a spiral that the vehicle entered until it hit the target altitude.

Once the weights were properly adjusted and the initial estimate of the costs was such that the correct path was chosen, the path planning results were considerably closer to the target. An example if provided in Figure 37.

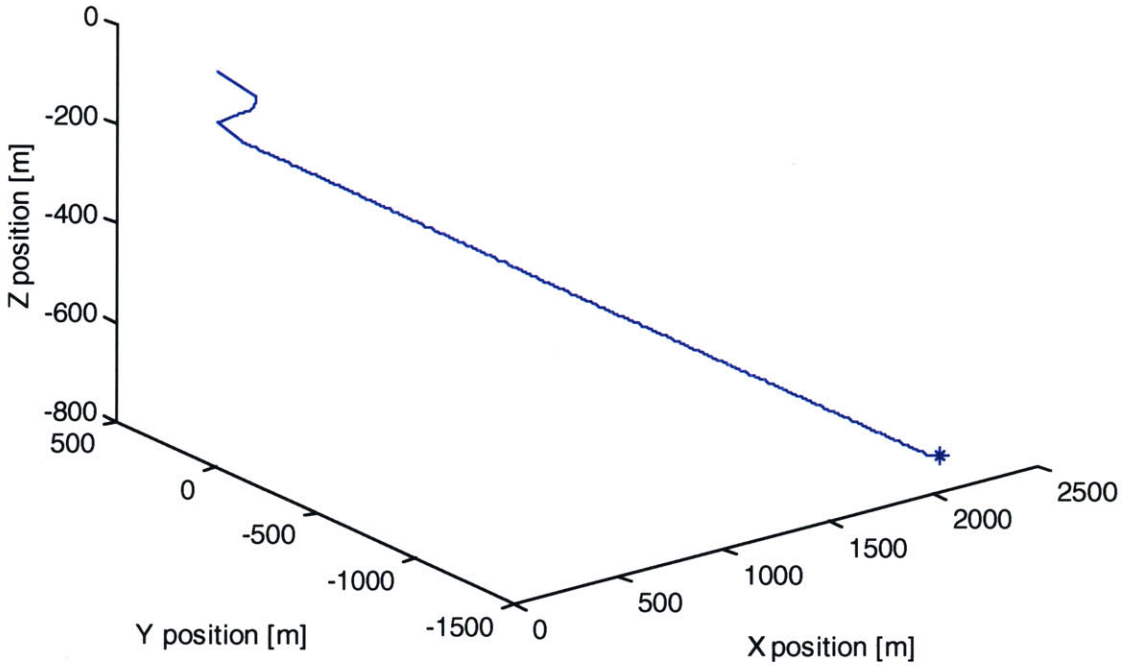


Figure 37: Properly Weighted Path-Planning Selection

The selected path reflects the optimal path between the starting point and the endpoint subject to the trajectory constraints imposed by the maneuver library. In this case, the target is barely reachable and so the algorithm chose to orient the vehicle and then head towards the target at the highest possible velocity. The initial turns are actually an eighth-order spline between the beginning and end points of the maneuver. In reality, the vehicle would follow the spline between the maneuver endpoints in order to produce a trackable trajectory. A selected portion of the path output file can be seen in Table 13.

ID	S1	S2	S3	S4	...	S12	Time	dt	Prim Type	Prim ID	Target?
0	3	0	-1	0	...	0	0.5	0.5	1	71	0
35	103	-85	-47	-2	...	0	18	0.5	0	21	0
56	14	-69	-79	-4	...	0	28.5	0.5	1	534	0
79	29	22	-109	-3	...	0	40	0.5	0	9	0
80	24	23	-110	-3	...	0	40.5	0.5	1	247	0
103	67	-48	-141	0	...	0	52	0.5	0	22	0
530	2240	-1235	-800	0	...	0	265.5	0.5	0	22	2

Table 13: Path Output File

The output file contains all of the data segments discussed earlier. The primitive type is zero for a trim state and one for a maneuver. The target column determines whether or not the vehicle has hit the target. With a value of zero, the vehicle is still in the path; with a target of one, the vehicle is within the bounded tolerances of the target; with a value of two, the vehicle is at the target altitude, but not within the bounded tolerances of the target.

In addition to the path-planning algorithm, the dynamic simulation also had to function properly. The key to the functional representation was making sure that all forces and moments were integrated in the body frame, and then transformed into the inertial coordinate frame through rotation and translation matrices. A basic glide path is shown in Figure 38.

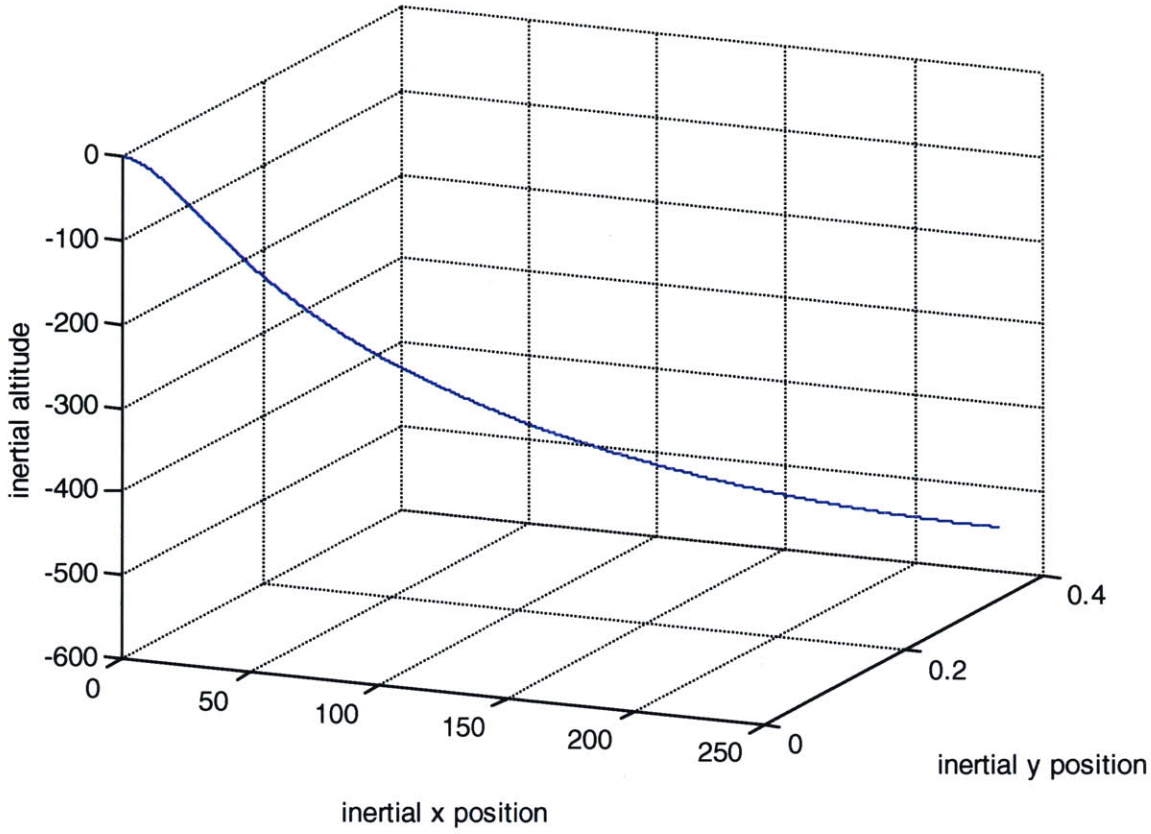


Figure 38:Basic Glide Path With No Toggle Deflection

The vehicle is given a slight initial velocity, with orientation, and essentially dropped. The vehicle accelerates and eventually hits a glide slope. The vehicle in Figure 38 has no toggles deflected. The next example, shown in Figure 39, has a deflection of the toggles such that a roll moment is created and the vehicle begins to roll.

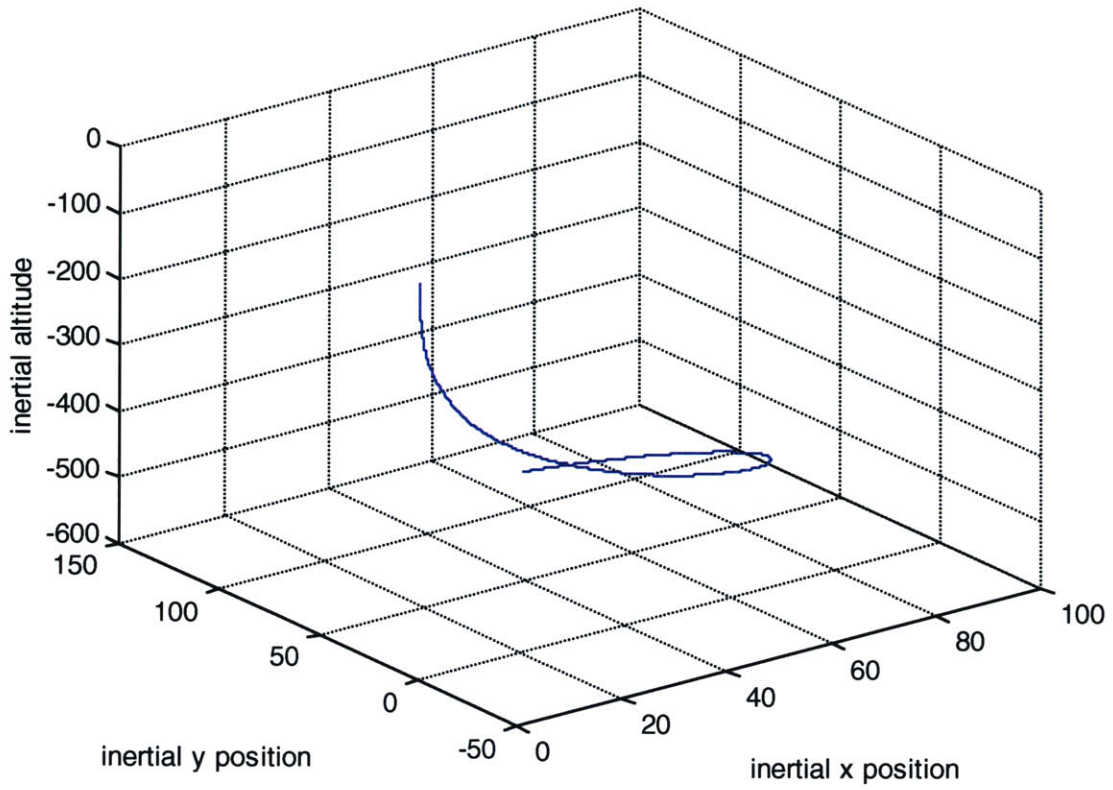


Figure 39:Glide Path with Differential Toggle Deflection

The differential toggle creates angular accelerations in the roll direction. The steady-state roll angle is proportional to the deflection of the toggle. The gravity term eventually cancels out the acceleration and just allows the turn of the vehicle in yaw.

Once a functional dynamic model of the parafoil was created, the next step was to add in the controller. The controller commanded the vehicle to a target waypoint. The target waypoints were set from the path-planning file. The controller inputs were the two rear toggles, with displacements "dl" for the left and "dr" for the right. The toggles didn't directly affect any state of the system in an uncoupled fashion, so a new set of variables were substituted to better decouple the system. The definitions of the new variables are shown in Equation 44 and Equation 45.

Equation 44: $d_b = \frac{d_l + d_r}{2}$

Equation 45: $d_d = \frac{d_l - d_r}{2}$

The variables were "dd", for differential toggling, and "db" for brake toggling. The "dd" term directly related to the roll of the vehicle by causing a roll moment, which in turn affected the yaw and turning of the vehicle. The "db" term essentially slowed the descent of the vehicle within a certain bounded area. The exact formulation of the two variables is discussed in Section 4.1 . The range of motion of the two variables is graphed in Figure 40.

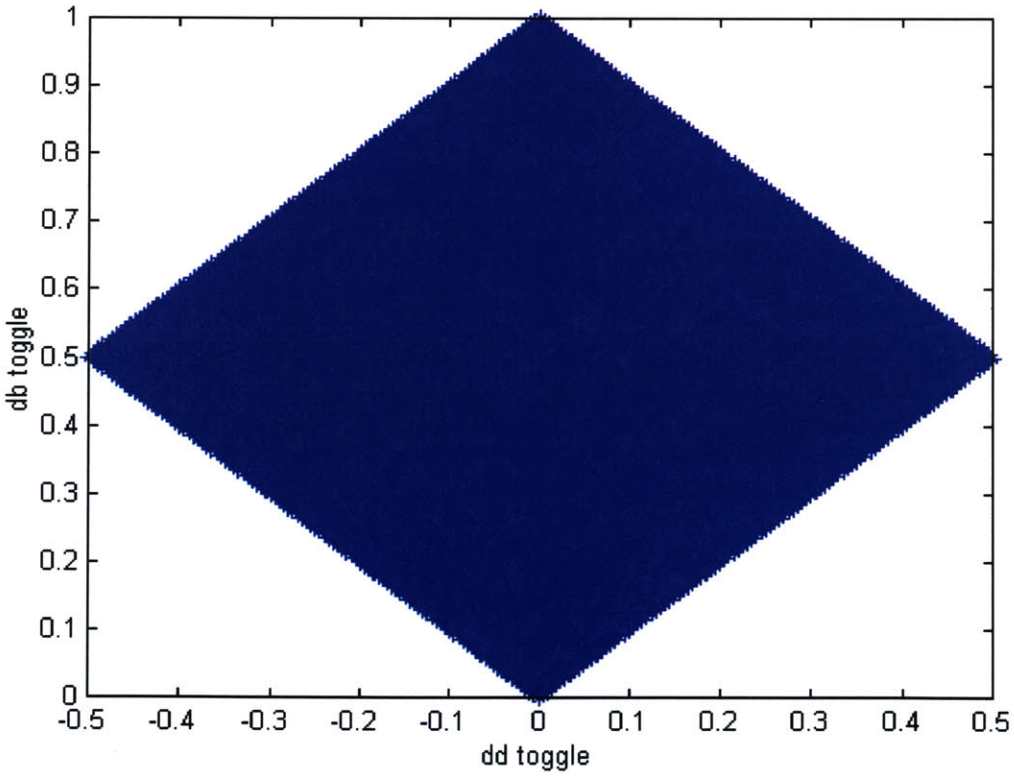


Figure 40: Range of Motion of Intermediate Toggle Variables

The diamond pattern was used to limit the output of the controller to within the achievable range of the vehicle. In the case where a requested value from the controller was outside of the variable range, the braking toggle was scaled based on the achievable area. This was due to the likelihood that changing the differential toggle would significantly throw the vehicle off of its path.

The basic proportional controller was implemented as described in Section 4.2 . The controller was setup as a dual input, dual output controller. The inputs were the desired orientation and the desired velocity. The outputs were the toggle positions. Additionally, the output of the brake toggle was scaled based on the possible positions shown in Figure 40. The controller was able to use the toggle to control the position to within a tolerance zone in the x and y positions. Examples of controlled flight paths are shown in Figure 41.

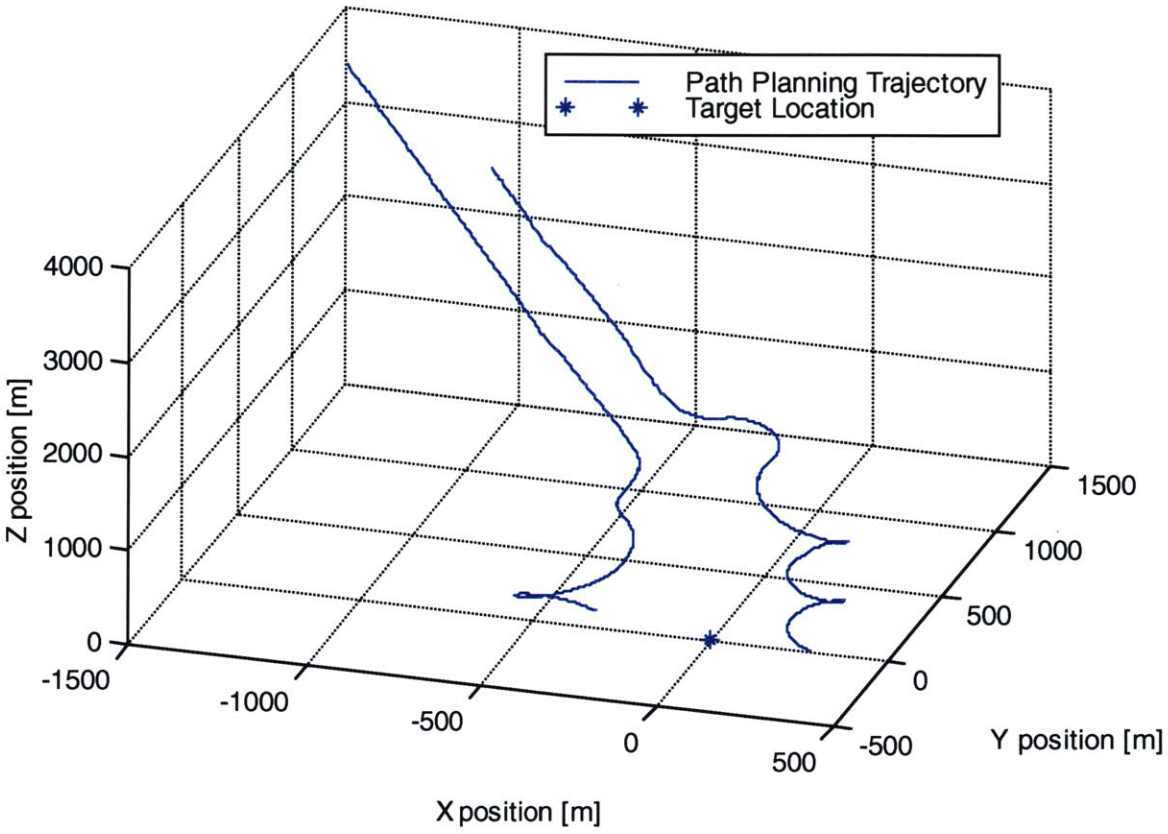


Figure 41:Controlled Position Flight Profile

The two profiles demonstrate the controller sending the vehicle to a desired position. In the case shown in Figure 41, the vehicle's initial position changes the profile of the path between the two flights. In the case of position, the flight path was changed based on two different initial positions. The end position difference is due partially to a large tolerance and partially to the rough calculations in the cost matrix.

Once the individual components were functional, they were combined into a single simulation. The simulation ran through a large command loop, with the simulation communicating with the host, and the host communicating with the DSP. The simulation iterated through several paths based on the changes in the vehicle state.

The estimates of the position based on the maneuver library didn't exactly correlate with the actual changes in states in the simulated vehicle. By iterating through several paths, the vehicle is able to adapt progressively to any changes in the target or vehicle state.

The controller followed the desired path with definite variations most probably due to the differences in the six-degree-of-freedom model and the maneuver library dynamic model. An example flight profile is shown in Figure 42.

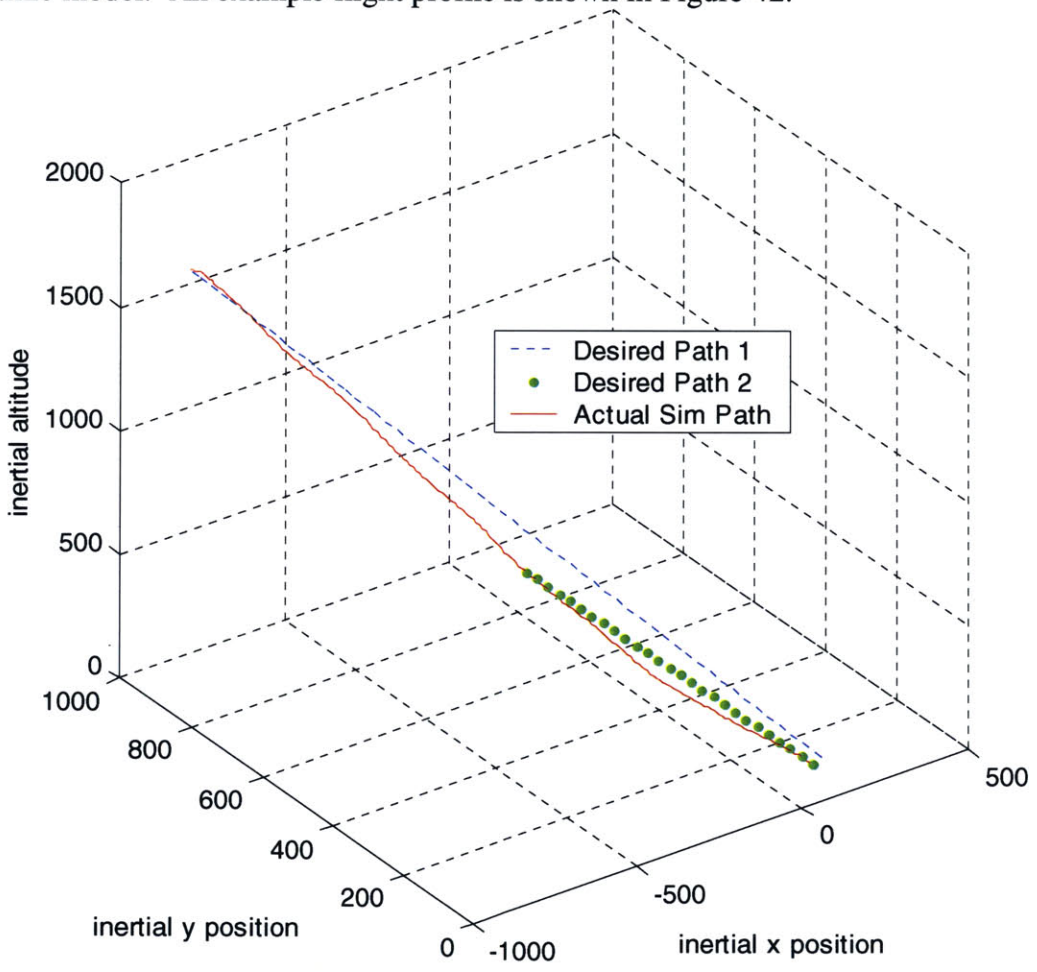


Figure 42:Controller Tracking of Desired Path

Two desired trajectories are plotted based on how they are updated in the full simulation. The end result is closer to the target than the initial desired path commands. The endpoint is offset because the vehicle was within tolerance of the target.

An additional set of measurements was carried out on the simulation itself. Since the path-planning algorithm should be able to run in real-time, it was measured with various loads. Increasing the total flight time or decreasing the time interval of the path output file could control the load. In the case of Figure 43, the load was controlled using the time interval to change the number of steps. The graph indicates that the higher the load, the longer the time interval for each step of the program, with the exception of the initialization time, which remains constant. The units on the y-xis are actually time over the time for which the simulation was set to integrate.

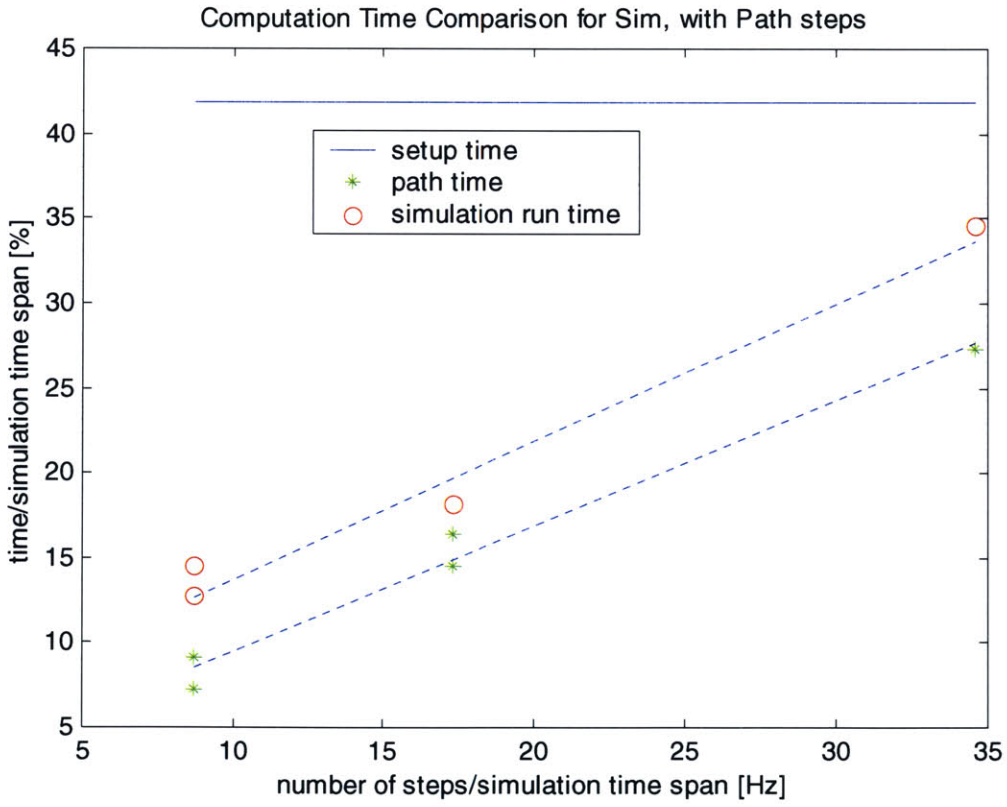


Figure 43:Simulation Time Comparison

In all of the cases mapped above, the path-planning segment took much less time than the actual simulation. Although that is within the real-time range, the time-cost for calculating the path grows with the number of steps required.

Chapter 7: Summary and Conclusions

The summary and conclusions section will be addressed by simulation component, and then the performance as a whole. The simulation components are broken down as before into the dynamic simulation, the host pc program, the path-planning DSP program, and the cost matrix calculation and analysis. Each of the sections will be broken down into the following: comments on performance, conclusions based on performance, and suggested further research.

7.1 The Parafoil Dynamics and Controller

The parafoil dynamics model was not a high-fidelity model since there was not a target parafoil for optimization. The performance of the six-degree-of-freedom model was within the expected realm and comparable to the results of any typical unpowered winged aerial vehicle. The parafoil turns were relatively large in radius, but that was to be expected of a large-scale parafoil.

The controller functioned within the expected range for a typical linear controller. The tracking wasn't perfect, but reflected a typical proportional controller. Other controllers might offer better tracking at slightly higher processing cost. The controller could be improved with a nonlinear adaptive controller or contraction analysis, assuming the extra processing was available. Contraction analysis would allow control of the

complete nonlinear system by proving that the system would converge to a particular value or trajectory. Additionally, in a full system implementation, the controller would require that the outputs of the sensors be filtered with an observer to produce an estimate of the full states. In this case, the states were fed back directly, assuming full state feedback.

The dynamic model could be improved by experimenting with a target parafoil and optimizing the equations to match performance. Additionally, parameters could have been made nonlinear so as to account for irregularities in the wings and flaps during toggle deflections. Additionally, the ability to stall could have been added into the model. The present model was created with approximations of the angle of attack and drag and lift coefficients.

The coding of the dynamic model could have been improved or changed to run in parallel with the path planning routine. The new path file would only be replaced from a buffer once it had been replaced. This would yield a larger difference in the actual position versus the path file. The controller and guidance could be adapted to the DSP rather than on the host pc or host processor. This would reduce the amount of processing on the other processor, as well as reduce time lags due to communications lags and file interactions.

7.2 The Host Personal Computer Communications Program

The host PC communications program was programmed in a basic manner as described in Section 5.2.2 . The actual programming was relatively straightforward. Allowing the dynamic simulation and the path-planning algorithm to run in parallel could

optimize the program. Additionally, the host program could be implemented to use more compact file formats to save space and time in reading them. The elimination of files by directly passing information back and forth could also speed up communications.

7.3 The Path-Planning Algorithm and Remote DSP

The path-planning algorithm performed within expected tolerances. The algorithm required that the parameters and cost matrix were setup in a rigorous, accurate fashion. The cost matrix controlled most of the performance of the algorithm. The resolution of the positional grid varied the accuracy of the end-of-path cost, since the interpolated values could only be within tolerance of the actual cost. The number of maneuvers and trim-states also limited the performance of the algorithm. The smaller the library, the shorter time that the path-planning algorithm took to create the path. The larger the library, the more maneuver-space resolution was available and the more likely that the vehicle would arrive within optimal range of the target.

The path-planning algorithm could be improved in several ways, including parallel processing, increased resolution, and time-step-adapting code. Parallel processing could be implemented by having the host and simulation output the vehicle state every time that the path-planning function was finished processing. The path could then be updated at a higher frequency. The parafoil dynamics simulation and controller would have to then run in parallel, as described in Section 7.1 . This would be a more realistic implementation of the system, since the system would obviously still be flying while the path-planning algorithm was calculating the path. A similar set of changes has already been successfully implemented on the helicopter.

The path-planning file output size also affected the speed of the output. The path-planning file output could be shrunk or grown depending on the desired resolution. The vehicle only requires a lower resolution path at the end of its flight, while the path output file could be quite large for a higher resolution. If the resolution of the path-planning output file could be adjusted for high resolution at small distances to the target, the output file would never grow beyond a smaller-sized file, and the path-planning algorithm output time would be reduced significantly at long distances.

7.4 The Cost Matrix Calculations

The cost matrix calculations required a lot of working and reworking to get proper function in the path-planning algorithm. Two main areas affected the overall implementation of the cost matrix significantly: the weighting of the cost algorithm, as well as the initial estimates of the cost matrix grid. The weighting of the cost matrix placed priorities on distance error, altitude error, and end velocity error. By changing the weights, the priority and resulting costs of each maneuver changed in such a way as to optimize for the desired parameter.

The estimation of the initial costs was difficult because it required estimating the optimal path for the vehicle to take in each position relative to the target. For a coupled system, this is a relatively complex function. The only way around it would be to decouple the system, perhaps by adding power or velocity to the parafoil. When the end values weren't properly configured, the costs could end up so that maintaining the current position was less costly than approaching the target.

Further improvements to the system could be implemented by further modeling and breaking the system into finer resolution segments, both for the maneuver and trim libraries, as well as for the gridding of the position space. Additionally, increasing the number of value iterations on the cost matrix calculations could increase accuracy due to further differentiating between the costs of each maneuver and its end cost. The higher resolution maneuver and trim libraries would allow the cost matrix to reduce the costs of some paths significantly, particularly if the maneuvers were able to be implemented within a close range of the target state. The addition of a stall maneuver could also increase the accuracy of the system, assuming the vehicle could recover from the stall condition.

7.5 The Overall Simulation Implementation

The simulation and path-planning algorithm worked well based on the simulation. Although the gridding was fairly coarse in both the maneuver space and the positional gridding, the vehicle was able to plan and follow an optimal path based on the path-planning algorithm. The overall simulation worked with all of the issues addressed in the previous sections of Chapter 7.

The main issue to be addressed in the overall simulation was to reduce the time of the simulation as much as possible such as to reduce the overall path-error of the vehicle. Time reduction, as mentioned earlier, could be implemented using parallel processing, adapting path-length settings, more efficient programming, and more efficient communication between elements of the simulation.

The next steps should be to begin implementation on a real platform. Additional model improvements should be made for a specific parafoil platform, as well as adding a sensor system. The path-planning maneuver and trim libraries could then be updated for the experimental system. At that point, more in-depth studies could be carried out on the effectiveness of the path-planning algorithm.

Bibliography

[1] McConley, Marc W., Piedmonte, Michael D., Appleby, Brent D., Frazzoli, Emilio, Dahleh, Munther A., and Feron, Eric, "Hybrid Control for Aggressive Maneuvering of Autonomous Aerial Vehicles", Digital Avionics systems Conference, October, 2000.

[2] Bertsekas, Dimitri P., *Dynamic Programming and Optimal Control*, Vol. 1, Athena Scientific, Massachusetts, 1995, pp.50-123.

[3] McConley, Marc W., Piedmonte, Michael D., Appleby, Brent D., Frazzoli, Emilio, Dahleh, Munther A., and Feron, Eric, "Hybrid Control for Aggressive Maneuvering of Autonomous Aerial Vehicles" [Presentation], Digital Avionics systems Conference, October, 2000.

[4] Frazzoli, Emilio, Dahleh, Munther A., and Feron, Eric,"Hybrid Control Architecture for Aggressive Maneuvering of Autonomous Helicopters," The Institute of Electrical and Electronics Engineers Conference for Decision and Control, MIT-Laboratory for Information and Decision Systems, December 1999.

[5] Goodrick, Thomas F., Pearson, Allan, and Murphy, A.L., Jr., "Analysis of Various Automatic Homing Techniques for Gliding Airdrop Systems with Comparative Performance in Adverse Winds", Paper 73-462, Airdrop Engineering Laboratory, US Army Natick Laboratories, Massachusetts.

[6] Frazzoli, Emilio, Dahleh, Munther A., and Feron, Eric,"Robust Hybrid Control for Autonomous Vehicle Motion Planning", Technical report LIDS-P-2468, Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, Cambridge, MA, 1999. Revised version submitted to the The Institute of Electrical and Electronics Engineers Transactions on Automatic Control, 2000.

[7] Frazzoli, Emilio, Dahleh, Munther A., and Feron, Eric, "Real-Time Motion Planning for Agile Autonomous Vehicles," AIAA paper 2000-4056, August 2000.

[8] Piedmonte, M., and Feron, E., "Aggressive Maneuvering of Autonomous Aerial Vehicles: A Human-Centered Approach," *Robotics Research*, Springer-Verlag, 2000.

[9] Frazzoli, Emilio, Dahleh, Munther A., Feron, Eric, "Trajectory Tracking Control Design for Autonomous Helicopters using a Backstepping Algorithm", ACC00-IEEE1440, The Institute of Electrical and Electronics Engineers, 2000.

[10] Hogue, Jeffrey R., Jex, Henry R., "Applying Parachute Canopy Control and Guidance Methodology to Advanced Precision Airborne Delivery Systems", AIAA Paper 95-1537-CP, 1995.

[11] Brown, Glen J., "Parafoil steady turn response to Control Input", American Institute of Aeronautics and Astronautics, Paper 93-1241, 1993.

[12] Jex, Henry R., and Hogue, Jeffrey R., "The 'Psuedo-Jumper' Parachute Guidance and Control System", [Presentation] AIAA Aerodynamic Decelerator Systems Technology Conference, May, 1995.

[13] Goodrick, T. F., "Simulation Studies of the Flight Dynamics of Gliding Parachute Systems", US Army Natick Research and Development Command, 79-0417, Natick, Massachusetts, 1979.

[14] Lingard, J. Stephen, "Ram-Air Parachute Design", AIAA Aerodynamic Decelerator Systems Technology Conference, May, 1995.

[15] Advisory Group for Aerospace Research & Development, "The Aerodynamics of Parachutes", North Atlantic Treaty Organization, AGARDograph 295, Neuilly Sur Seine, France.

[16] Doherr, K.E., "Parachute Flight Dynamics and trajectory Simulation", DLR – Institute of Flight Mechanics, Braunschweig, Germany, 1998.

[17] Murray, James E., Sim, Alex G., Neufeld, David C., Rennich, Patrick K., Norris, Stephen R., and Hughes, Wesley S., "Further Development and Flight Test of an Autonomous Precision Landing System Using a Parafoil", NASA Technical Memorandum 4599, 1994.

[18] Sim, Alex G., Murray, James E., Neufeld, David C., and Reed, R. Dale, "The Development and Flight Test of a Deployable Precision Landing System for Spacecraft Recovery", NASA Technical Memorandum 4525, September, 1993.

[19] Slotine, Jean-Jacques, and Li, Weiping, *Applied Nonlinear Control*, Prentice Hall, Englewood Cliffs, New Jersey, 1991, pp.277-459.

[20] Dorf, Richard C., and Bishop, Robert H., *Modern Control Systems*, 7th Edition, Addison-Wesley Publishing Company, Reading, Massachusetts, 1995, pp.526-748.

[21] Friedland, Bernard, *Control System Design: An Introduction to State-Space Methods*, McGraw-Hill, Inc., Boston, Massachusetts, 1986, pp.222-377, 411-469.

[22] Brutzman, Donald P., *A Virtual World for an Autonomous Underwater Vehicle*, Ph.D. Dissertation, Naval Postgraduate School, Monterey California, December 1994.