Adaptive Mode Transition Control Architecture with an Application to Unmanned Aerial Vehicles

A Dissertation
Presented to
The Academic Faculty

By:

Luis Benigno Gutiérrez Zea

In Partial Fulfillment
Of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
July 2004

Adaptive Mode Transition Control Architecture with an Application to Unmanned Aerial Vehicles

Approved by:

Dr. George Vachtsevanos, Advisor

Dr. Bonnie Heck, Co-Advisor

Dr. Linda Wills

Dr. Magnus Egerstedt

Dr. J. V. R. Prasad

May 19, 2004

To my family and the memory of my father

ACKNOWLEDGEMENT

I would like to thank Dr. George Vachtsevanos, my advisor, for his encouragement and support during my PhD studies at Georgia Institute of Technology. I thank Dr. Bonnie Heck, my co-advisor, and also Dr. Linda Wills, Dr. Magnus Egerstedt, and Dr. J. V. R. Prasad for serving on my examination committees and for their valuable suggestions that guided me through the completion of my thesis. I thank the UAV lab team headed by Dr. Eric Johnson for their help and support for the flight tests. I acknowledge the financial support of DARPA/AFRL, sponsor of the Software Enable Control research program. Finally, I would like to acknowledge the support received from my home country, Colombia, from the Colombian Fulbright Commission, Colciencias, and Pontificia Bolivariana University at Medellín.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Unmanned Aerial Vehicles (UAVs) are required to possess levels of autonomy in order to execute complex missions robustly and reliably. Intelligent/hierarchical control techniques have been suggested as a means to address critical autonomy issues. The objective of this research is to develop a hierarchical/intelligent control architecture for an UAV. The architecture consists of three levels: high level, middle level, and low level. Mission planning routines occupy the highest level. At this level, information about waypoints that the vehicle must follow is used to generate the sequence of actions that should be performed to go through those waypoints while maintaining some physical constraints. These actions are split into a sequence of tasks; each of them containing target position, target speed, target heading, heading mode, and target direction of the flight path. The tasks are then stored in a task queue and sent in an orderly manner to the middle level. The middle-level controller coordinates task execution while a trajectory generation component receives the task information from the high-level module and provides set points for low-level stabilizing controllers whose function is to maintain the vehicle in a stable state and follow accurately the commanded trajectory. An adaptive mode transitioning control algorithm resides at the lowest level of the hierarchy consisting of two components: a mode transitioning controller and the accompanying adaptation mechanism. The mode transition controller is composed of a mode transition manager, a set of local controllers, and a set of active control models. Local controllers operate in local modes and active control models operate in transitions between two local modes. The mode transition manager determines the actual mode of operation of the

vehicle based on a set of mode membership functions and activates a local controller or an active control model accordingly. The adaptation mechanism uses an indirect adaptive control methodology to adapt the active control models. For this purpose, a set of plant models is trained based on input/output information from the vehicle and used to compute the linearized models required by the adaptation algorithms. The core of the adaptation mechanism is a finite horizon optimal control algorithm, which determines the optimal control signal that in turn is used to train the active control models. The adaptation routine may be turned on only when needed. The transitioning algorithm operates in real-time while adapting on-line to disturbances and other external inputs. This intelligent/hierarchical architecture has been implemented using a novel software infrastructure called Open Control Platform (OCP), which facilitates interoperability, plug-and-play and other functionalities. Simulation and flight test results validate the proposed scheme.

The main contributions of this research are:

- Development of a hierarchical architecture for the implementation of the adaptive mode transition control, flexible enough to be able to accommodate future enhancements and more intelligent at the highest level of the hierarchy.
- Development of a new approach to the adaptive mode transition control problem addressing main concerns from previous accomplishments in this area.
- Exploitation of new software technologies including the OCP and hybrid controls API to show how they enable the implementation of advanced control algorithms for UAVs.

- Implementation of the architecture and verification of its performance in software in the loop simulation, hardware in the loop simulation and through flight testing.

CHAPTER 1

INTRODUCTION

1.1 Motivation

Control of unmanned aerial vehicles (UAVs) presents unique challenges not only in the design of control algorithms, but also in the strategies and methodologies used to integrate and implement those algorithms on actual vehicles. The dynamics of UAVs are usually highly non-linear and difficult to model accurately due to the complexity of the aerodynamic and propulsive forces acting on the vehicles. The environment in which UAVs operate is also uncertain, leading to unexpected disturbances. This means that control algorithms have to be able to cope with the uncertainty associated with the UAV dynamics (parametric and structural) as well as those associated with the environment (external perturbations) using robust or adaptive techniques. The first UAVs were remotely piloted, but current tendency is to eliminate the remote pilot and give the vehicles enough intelligence so that they can perform their missions autonomously. To achieve the required intelligence, it is necessary to develop hierarchical architectures that consider not only low level control objectives, stabilization and tracking, but also incorporate high level objectives such as mission planning, scheduling, etc. Therefore, new technologies need to be developed for UAVs including control algorithms to improve the degree of autonomy/intelligence and architectures to implement these algorithms in an efficient manner.

1.2 Problem Statement

Before presenting the problem statement, some required definitions related to a UAV are presented.

Definition 1. Local mode.

A local mode is a region of the state space around an operating point in which the vehicle exhibits quasi steady state behavior. Quasi steady state behavior means that some of the state variables remain constant in that operating point.

For example, some local modes for a helicopter are hover, forward flight at a constant speed, backward flight at a constant speed, and sideward flight at a constant speed. For the operating points associated with these local modes the velocity is constant and the angular rates are zero.

Definition 2. Local Controller.

A local controller is a controller that guarantees the stability and some tracking performance of the closed loop system for any feasible reference trajectory in a local mode.

Definition 3. Transition Region.

A transition region is a region of the state space outside any local mode that includes all the feasible trajectories between two local modes.

For example, some transition regions for a helicopter are hover to forward flight at a constant speed, hover to backward flight at a constant speed, and hover to sideward flight at a constant speed.

Definition 4. Operating Region.

The operating region is the region of the state space generated by the union of all the local modes and transition regions.

These definitions are illustrated in Figure 1 where a map of the state space for a rotorcraft UAV is presented. In that map only forward and sideward velocity are considered since they are the ones that determine major changes in the dynamics. For the case showed, five local modes are considered (in blue) with four transitions (in light blue). The designer selects the number of modes and transitions, so more modes or transitions can be selected if the operating region of the vehicle needs to be extended.

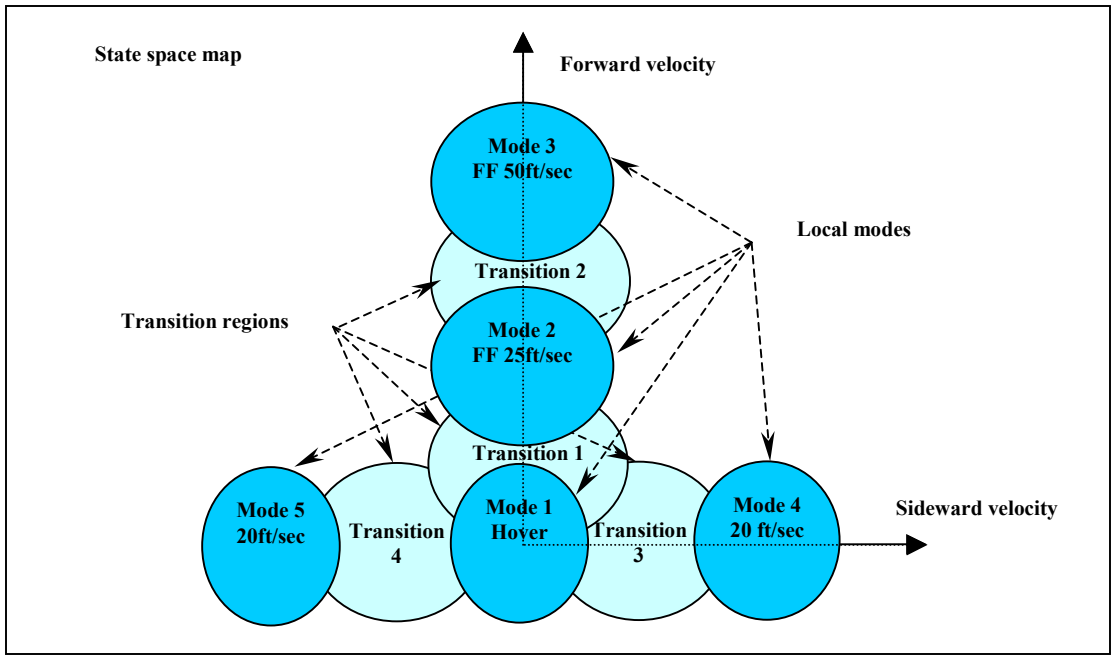


Figure 1. Illustration of Local Modes and Transition Regions for a Rotorcraft UAV

The problem addressed in this research can be stated as follows: consider a rotorcraft UAV (also called rotary wing UAV).

- Given a set of local modes, design local controllers able to control the vehicle in their respective local modes in a stable fashion.
- Develop a controller able to operate in a stable fashion in the operating region including the local modes and the transitions among them based on the local controllers. This controller should operate as one controller in the whole operating region.
- The controller should be able to adapt to internal and external disturbances maintaining a stable operation in the whole operating region.
- Develop the architecture for the implementation of the control methodology.

1.3 Assumptions

The work presented in this thesis was done under the following assumptions:

- Desired UAV trajectories are produced by a trajectory generator based on commands received from a mission planning component.
- Trajectory commands do not induce controller saturation.
- Local modes are selected and local controllers are designed such that the stability and prescribed performance of the closed loop system is guaranteed on the local modes and at least the stability is guaranteed on transition regions adjacent to corresponding local modes.
- Local Modes and transition regions are selected such that they cover the operating region of the vehicle

1.4 Overview

This dissertation is divided into nine chapters. After this brief introduction, Chapter 2 presents background and related work including a discussion of the state of the art in the area. A discussion about fuzzy neural networks and the development of two new learning algorithms for them are presented in Chapter 3. In Chapter 4, the active modeling framework used to represent the models of the plant under control is presented. Chapter 5 presents the main topic of this dissertation, the overall architecture for the adaptive mode transition control. Chapter 6 describes the software implementation of the proposed architecture. Chapter 7 describes the implementation using the Open Control Platform (OCP). In Chapter 8, simulation and flight test results are presented. Finally, Chapter 9 presents conclusions of the work completed and future research needed in this topic.

CHAPTER 2

BACKGROUND

The complex nonlinear dynamics of UAVs, as other large-scale systems, usually present multiple modes of operation with very different dynamic behaviors that require stable, robust and smooth transitions between them. Several control techniques have been developed to cope with the non-linearities across different operating modes, model uncertainties, external disturbances, and, in some cases, input saturation. These techniques include gain scheduling, sliding mode control, adaptive control, and recently model predictive control.

In the gain scheduling methodology a nonlinear controller is constructed combining a family of linear controllers [1, 2]. A scheduling variable is selected that could be a function of the state, the outputs of the system, or an exogenous variable. Linear controllers are designed for a finite number of operating conditions corresponding to different values of the scheduling variable. Then, the controller parameters used at any time are obtained by interpolation based on the actual value of the scheduling variable [3-5]. This technique relies on a slow variation of the scheduling variable and usually requires the design of many linear controllers to cover the operating region of the system. For conventional gain scheduling only linearizations of the plant at equilibrium operating points are considered for the design of the controller, an extension to the case of non-equilibrium operating points was given in [6].

Developments in recent years have given a more rigorous treatment to the gain scheduling approach, leading to design methodologies that guarantee the stability and

robustness of the closed loop system given that the scheduling variable remains in a compact set. For instance, in [7] a interpolation technique for the controller and observer gains was developed that guarantees the local stability at any value of the scheduling variable whenever some easily computed bounds on the rate of the scheduling variable are met. In some of the new methods for robust gain scheduling, the nonlinear dynamics of the plant is represented by a linear parameter varying (LPV) system, i.e. a linear time-varying system whose state-space matrices are fixed functions of a vector of varying parameters [8-10]. In [8] a LPV controller is developed that guarantees H_∞ performance for LPV polytopic plants, i.e. those whose parameter vectors are in a polytope in the parameter space and whose state-space matrices are affine functions of the parameter vector. The case of LPV plants whose state-space matrices have a linear fractional dependence on the parameters is considered in [9]. These approaches are too conservative since they do not assume any bounds on the rate of change of the parameters. A less conservative and more general approach is presented in [10]. In all these cases the problem is solved in the context of convex semidefinite programming [11], being reduced to the solution of a set of linear matrix inequalities (LMIs). LMI techniques are now used as a powerful tool to solve many problems in control [12]. Currently, there is efficient optimization software that allows the solution of these kind of problems [13]. Fuzzy gain scheduling has also been presented in the literature as a way for implementing gain scheduling controllers [14, 15]. In those cases the advanced gain scheduling techniques like the ones discussed before are applied to a system modeled by a Takagi-Sugeno fuzzy system.

A controller for a small UAV that uses gain scheduling can be found in [16, 17]. They have demonstrated autonomous extreme maneuvers for a small unmanned helicopter. They used two controllers, one for longitudinal-vertical dynamics and other for the lateral-directional dynamics. Both controllers are based on an LQ design augmented with integrators for accurate tracking of angular rates, controller gains were scheduled on forward velocity. Notch filters were used for dynamic compensation of fuselage-rotor dynamics given that LQ controllers were designed using a reduced order model discarding the rotor flapping dynamics [18].

In the sliding mode control methodology (more generally called variable structure control), a high speed switching control strategy is used to force the state of the system to be in a surface called the sliding mode or switching surface that is a manifold of the state space chosen for the designer to meet the desired control goal like stabilization, tracking, or regulation [19]. This technique is very attractive because it makes the controller very robust to model uncertainties and external disturbances, i.e. once the state gets to the sliding surface the behavior of the system becomes independent of system parameters. The major disadvantage associated to this methodology is the chattering effect in the actuators; however, there are ways to avoid that problem [20, 21]. Fuzzy sliding mode control techniques have been proposed in the literature that combine fuzzy models with sliding mode control [22, 23]. Those methods alleviate the effect of chattering and allow the sliding mode control of systems without a previous knowledge of their model. Applications of variable structure control or sliding mode control have been reported in the literature for a variety of nonlinear systems including, for instance, aircrafts [24, 25]

and robot manipulators [26, 27]. An application of sliding mode control for close formation flight of multiple UAVs is presented in [28].

There are many adaptive control methodologies proposed in the literature [29]. In adaptive control the controller parameters are adapted online to accommodate for uncertainties in the model or improve control performance in presence of external disturbances. This adaptation or continuous change in the parameters is determined by an adaptation rule that is based on input/output information from the plant being controlled and should guarantee the stability of the closed loop system. Adaptive control techniques are classified as either direct or indirect. In direct adaptive control, controller parameters are directly adapted based on input/output information of the plant and at times the output of a desired reference model. On the other hand, indirect adaptive control methods try to estimate the parameters of the plant model based on input/output information from the plant and then use these estimates to adapt the controller parameters. Many of the indirect adaptive control schemes are based on the certainty equivalence approach, i.e. the uncertainty of estimated parameters is not taken into account during the controller design so the parameter estimates are used by the controller as if they were the true values. Adaptive dual control methods have been proposed looking for optimal adaptive control techniques that consider uncertainties in the parameter estimations [30, 31]. A dual adaptive control system should satisfy two properties: the control signal ensures that the output cautiously tracks the desired reference value, and it excites the plant sufficiently to accelerate the parameter estimation to improve the performance of the controller [30, 31]. A survey of adaptive flight control can be found in [32]. Neural networks have been applied successfully to adaptive nonlinear flight control for a variety of aircraft [33-36].

For the specific case of UAVs some adaptive control methodologies have been developed. For instance, in [37] a structured adaptive model inversion approach is used, in which kinematical nonlinearities are incorporated so the only uncertainties considered are the ones from the aerodynamic and propulsive forces. Some schemes for nonlinear adaptive control of UAVs based on neural networks can be found in the literature. For instance, in [38] a neural network control for an unmanned helicopter based on approximate model inversion and feedback linearization is proposed, but it lacks a rigorous proof of stability. A rigorous approach to the nonlinear adaptive control for a UAV using neural networks is presented in [39, 40]. This approach is based on a multi loop structure (inner loop for attitude stabilization, and outer loop for trajectory tracking), where each loop uses approximate dynamic inversion plus a neural net for feedback linearization compensating for the imperfect inversion. The neural net weights are adapted online to minimize the tracking error. To avoid the effects of saturation in the adaptation a technique called pseudo control hedging is used [41, 42]. Rigorous proofs of stability based on Lyapunov theory are given.

The use of multiple models have been proposed in the literature as a way to improve the performance of adaptive control systems especially when large changes in the parameters or the environment happen [43]. Similar techniques have been used in the context of failure accommodation and fault tolerant flight control [44].

A type of multi-mode adaptive control called adaptive mode transition control was first introduced by Rufus et al in [45-48]. In that approach the problem of transitioning from a start mode to a goal mode and from a family of start modes to a family of goal modes was considered. A (local) mode was defined to be a region of the

state space in which the system exhibits steady state behavior. Furthermore, a tool was developed to assess the robustness of these mode transition controllers. A systematic procedure for designing off line the mode transition controllers and an online adaptation scheme were developed. The method of blending local mode controllers (BLMC) was the basis for this mode transition control scheme. Even though some good results were presented on simulation for the control of a helicopter from hover to forward flight, several problems have been detected which makes some aspects of the original methodology impractical for implementation in an actual UAV:

- In that methodology there were some so called "desired transition models" that were trained off line to model the trajectories followed during the transitions. In practice it is impossible to know in advance all the possible maneuvers that are going to be performed by a vehicle so it would require a huge amount of memory to store so many models. Also there would be a lot of computation time required for the generation of the transition trajectories and training of the fuzzy neural nets that implement them.
- A factor contributing to the necessity of having a lot of local modes for this methodology is the fact that local controllers were regulators, i.e. they were designed to keep the vehicle in some predefined operating point. In an actual UAV there may be too many operating points corresponding to the trajectories required for all its maneuvers.
- Another weakness detected in the original methodology is in the stability of the adaptation scheme. The control adaptation algorithm was designed based on a one step ahead optimal control value computed using a weighted least squares

method. That method uses the sensitivity control matrix obtained from the active plant model. On the computation of the optimal one step ahead control value no soft or hard constraints were imposed to the control signal. This produces control signals that are excessively aggressive generating instability. In fact, a simulation of this control scheme applied to a complete model of a rotorcraft UAV showed that the closed loop system becomes unstable. It is important to note that in the results presented in [45-48] no simulations were performed using the complete model of the vehicle. In fact, for the design and simulation in the transitions, just a reduced set of the state variables were considered, which could result in poor robustness of the control scheme.

- The algorithms of the original adaptive mode transition control methodology used fuzzy neural networks on the active control models, active plant models, and the desired transition models. The adaptation algorithms used for the fuzzy neural network in the original methodology were based on: a structure learning method to generate new input membership functions and also parameter learning methods based on offline least squares for the consequent weights and an on line gradient descent scheme for input membership function parameters and consequent weights. The only method used for online adaptation was the gradient descent scheme. For such a scheme it is difficult to find a good set of adaptation gains ensuring fast adaptation without incurring instability.

Model predictive control, sometimes called receding horizon control, is a control technique in which the control input is obtained from solving an optimal control problem over a usually finite time horizon. Only the control computed at the actual time is used

and then the optimization problem is solved again. This optimization predicts the future behavior of the system based on its model; this explains the name of the technique. Most applications of these techniques are employed in slow industrial process control [49] given its computational intensity. The methodology has been applied to linear systems and also to nonlinear systems and naturally involves constraints in control inputs, outputs and states [50]. Fuzzy model predictive control schemes combining fuzzy models with the model predictive control scheme for control of nonlinear systems have been also reported in the literature [51]. Neural networks have also been used to provide the nonlinear models required in the model predictive control scheme [52]. In [53] a nonlinear model predictive control (NMPTC) scheme for a UAV was formulated. The NMPTC algorithm also allowed for planning of paths with input and state constraints while tracking the generated position and heading trajectories.

Other techniques that have been applied to the control of UAVs are: in [54] a controller that takes into account the constraints on the control amplitude and the control rate for trajectory tracking of a vertical take off and landing UAV (VTOL-UAV) is presented, in that approach a sequential quadratic programming (SQP) algorithm computes a feasible reference as close as possible to the desired reference that ensures the control does not induce constraint violations; in [55] an autopilot for a fixed wing UAV was developed based on approximate discrete feedback linearization and disturbance accommodation control, the discrete nonlinear model used in this approach was obtained using the Adams-Bashforth method, the disturbance accommodation part was used to compensate for model errors and rejection of external disturbances.

The challenges appearing in UAVs and also in other complex system applications have led to the development of new software enabled control technologies [56-58]. One of the challenges for advanced control algorithms actually being developed for UAVs is that they are usually implemented on a variety of hardware/software platforms making their integration more difficult. Moreover, higher level algorithms and their integration with middle and low level routines is quite demanding, requiring tools that facilitate the implementation of hybrid and multirate systems that can be distributed to several platforms with guaranteed quality of service constraints. To face this challenge a new open software infrastructure especially developed for the implementation of complex reconfigurable control systems for UAVs was developed: the Open Control Platform (OCP) [59-63]. The OCP is a middleware-enabled software framework and development platform for the implementation of advanced embedded control systems especially targeted for UAVs. The OCP allows the implementation of hierarchical control systems including low level, middle level and high level controls permitting the interoperability of different control platforms (in several UAVs, control stations, etc.). Some of the OCP characteristics are support for hard and soft real time algorithms, innovative scheduling techniques, adaptive resource management, and support for dynamic reconfiguration.

An architecture for robust motion planning of autonomous vehicles is presented in [64], where a robust hybrid automaton is used to solve the motion planning problem for a nonlinear, high dimensional system. That work relates to the one presented in this thesis in the sense that a quantization of the state space is performed to reduce the computational complexity of the problem. There is an analogy between the trim trajectories and maneuvers of that work and the local modes and transitions of this work.

CHAPTER 3

FUZZY NEURAL NETWORKS

3.1 Fuzzy Neural Network Structure

In this research, fuzzy neural networks (FNN) are used to approximate nonlinear functions representing unknown nonlinear mappings in some cases and models for nonlinear dynamic plants in others. Specifically, a FNN is used as a nonlinear function approximation system for the implementation of active control model and active plant model components of the adaptive mode transition control scheme described in Chapter 5. The FNN is the core adaptive element used in those components.

The FNN constructs are neural-network-based connectionist models that implement the functions of a fuzzy logic system [65]. The FNN comprises a set of Takagi-Sugeno IF-THEN fuzzy rules whose consequents are affine mappings of the input vector. The structure of the FNN is divided into three major parts as shown in Figure 2: the premise part, the consequent part, and the defuzzification part.

The mapping generated by the FNN can be expressed as

$$y = \frac{\sum_{j=1}^{N} \mu_j(x) M_j x}{\sum_{k=1}^{N} \mu_k(x)}, \tag{1}$$

where x is the augmented input vector and y is the output vector for the FNN so

$$x = [1 \quad x_1 \quad \cdots \quad x_m]^T,$$
$$y = [y_1 \quad y_2 \quad \cdots \quad y_n]^T,$$

μ_j is the jth input membership function for premise of rule j, and M_j is a matrix representing the linear mapping for the consequent part of rule j (j=1,2,…,N).

Input membership functions are Gaussian functions of the form

$$\mu_j(x) = e^{-\frac{z_j^T(x) z_j(x)}{2}}, \tag{2}$$

where $z_j(x) = \rho_j(x - m_j)$,

and $\rho_j = diag(0, \gamma_{j1}, \gamma_{j2}, \ldots, \gamma_{jm})$.

m_j is the center of input membership function j and γ_{ji} parameters represent the inverse of the usual deviations for the input membership functions. The reason for using a representation based on the inverse of the deviations is that allowing a value of zero for any γ_{ji}, implies allowing an infinite deviation. This would permit representing a linear mapping with respect to the associated input. This was not possible with the representation used in [45-48] that had an upper limit for the deviations.

The main reasons for choosing a Fuzzy Neural Network instead of a Feed forward Neural Network for the approximation and identification problem are:

- It allows for structure learning, so its structure is suited to the actual input space. Thanks to this feature, the FNN is computationally more efficient since only the required input membership functions and fuzzy rules are created.
- There is no need for random initialization of any parameters as usual for other neural network schemes. The structure learning mechanism initializes all the parameters automatically based on provided input/output information. That way the FNN can generate a good approximation of the desired input/output mapping even without extensive parameter learning.

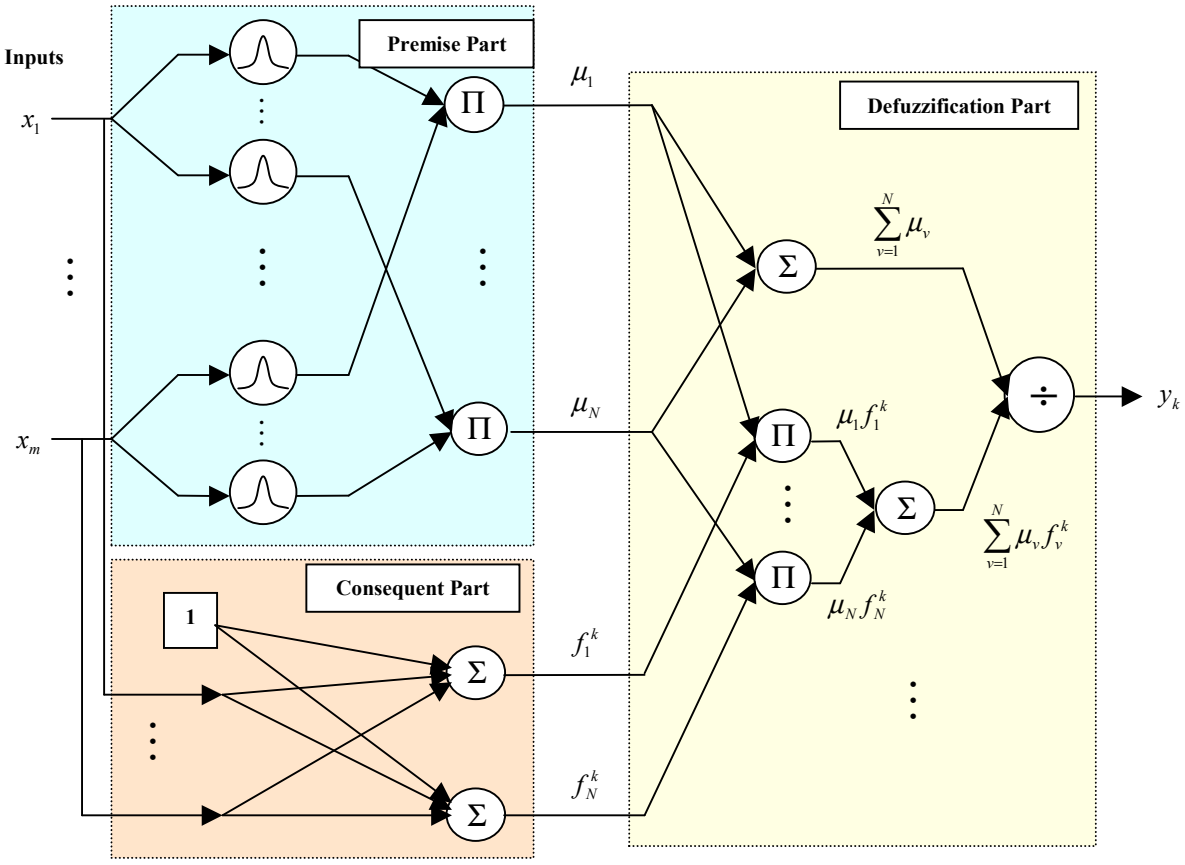


Figure 2. Fuzzy Neural Network Structure

- It is adaptive thanks to its parameter learning mechanisms allowing the implementation of the adaptation mechanisms described later in section 5.3.2.
- It implements a smooth differentiable mapping permitting the easy computation of a linear incremental approximation at any operating point. This feature is exploited in the adaptive mode transition control scheme to compute a linearized model of the UAV based on the FNN of the active plant model corresponding to a transition, as it will be discussed in section 3.3.

The structure learning algorithm for the FNN used in this work is taken from [48, 65]. This algorithm generates the parameters for input membership functions μ_j creating new rules whenever the excitation level of all existing membership functions for a given input is below a prescribed threshold. When that happens, a new rule is added with center in the input value and deviations chosen such that the new membership function exhibits a prescribed degree of overlapping with the closest membership function (the most excited one from existing input membership functions). The consequent matrix of the new rule is set so the output of the FNN matches the corresponding desired output. That way, the FNN structure is suited to the input space for the application at hand, and the number of required rules is minimized improving the computational efficiency.

In [65] a gradient descent method based on back propagation is used for parameter learning, tuning input membership parameters and also consequent parameters. In [48] the back propagation method is used for training the FNN on-line, and a least squares method is also used to train consequent parameters off-line. The gradient descent method requires just local error information at each iteration. If learning gains are not appropriately chosen, then it could become very slow (for small values) or unstable (for

large values). This makes it difficult to find a good set of learning parameters and does not guarantee convergence in all situations for a set of parameters. Even with a good set of learning parameters this method tends to be slow. The least squares method as presented in [48] requires not just local but a large set of input/output data to generate a good approximation. However, it avoids learning instability and guarantees minimal square error with respect to the training set.

For the reasons given above it was decided not to use the back propagation method for on-line training of the FNNs used in this work. Instead, a recursive least squares method was developed for off-line initialization and on-line training of the consequent parts of rules. Structure learning was used for off-line initialization and on-line generation of new rules. Once the rules were created, the input membership function parameters were left untouched.

3.2 Development of a Recursive Least Squares Learning Algorithm for Fuzzy Neural Networks

A new recursive least squares learning methodology has been developed here for the FNN based on [66], so just local information is used at each iteration. The idea behind this methodology is that at each iteration the parameters are updated so they minimize the total square error with respect to all the data including old data and the new data presented in that iteration. A detailed description of this method follows.

Let us assume there is a training data set of input/output pairs (x_i, y_i^d) for i=1,2,…,M, where y_i^d is the desired output vector and let y_i be the FNN output vector corresponding to input vector x_i.

Assume we look for an update of matrix M_p, so all the matrices M_j are kept constant except for M_p. Hence, we look for a new matrix M_p^* such that the square error between y_i and y_i^d is minimized on average over all the training set (i=1,2,…,M).

The output, y_i, can be written

$$y_i = \frac{\mu_p(x_i) M_p x_i}{\sum_{k=1}^{N} \mu_k(x_i)} + \frac{\sum_{\substack{j=1 \\ j \neq p}}^{N} \mu_j(x_i) M_j x_i}{\sum_{k=1}^{N} \mu_k(x_i)},$$

and

$$y_i^* = \frac{\mu_p(x_i) M_p^* x_i}{\sum_{k=1}^{N} \mu_k(x_i)} + \frac{\sum_{\substack{j=1 \\ j \neq p}}^{N} \mu_j(x_i) M_j x_i}{\sum_{k=1}^{N} \mu_k(x_i)} = y_i + \frac{\mu_p(x_i) \Delta M_p x_i}{\sum_{k=1}^{N} \mu_k(x_i)},$$

where y_i^* represents the output vector obtained after replacing M_p by M_p^* and $\Delta M_p = M_p^* - M_p$. It is required to find M_p^*, and therefore ΔM_p, that minimizes

$$J = \frac{1}{2} \sum_{i=1}^{M} \left\| y_i^d - y_i^* \right\|_2^2 = \frac{1}{2} \sum_{i=1}^{M} \left\| y_i^d - y_i - \mu_{pi} \Delta M_p x_i \right\|_2^2, \tag{3}$$

with $\mu_{pi} = \frac{\mu_p(x_i)}{\sum_{k=1}^{N} \mu_k(x_i)}$ for i=1,2,…,M.

The input, output, and desired output vectors from the training data set can be stacked together to form the matrices X, Y, and Y^d so

$$X = \begin{bmatrix} x_1 & x_2 & \cdots & x_M \end{bmatrix}^T,$$
$$Y = \begin{bmatrix} y_1 & y_2 & \cdots & y_M \end{bmatrix}^T, \text{ and}$$
$$Y^d = \begin{bmatrix} y_1^d & y_2^d & \cdots & y_M^d \end{bmatrix}^T.$$

Thus, J can be rewritten as

$$J = \frac{1}{2} trace\left(\left[Y^d - Y - QX\Delta M_p^T \right]^T \left[Y^d - Y - QX\Delta M_p^T \right] \right), \tag{4}$$

with $Q = diag(\mu_{p1}, \mu_{p2}, ..., \mu_{pM})$.

Setting $\frac{\partial J}{\partial \Delta M_p} = 0$ to find ΔM_p that minimizes *J,*

$$\frac{\partial J}{\partial \Delta M_p} = -\left[Y^d - Y - QX\Delta M_p^T \right]^T QX = 0 .$$

Solving for ΔM_p,

$$\Delta M_p = \left[Y^d - Y \right]^T QX (X^T Q^2 X)^{-1} . \tag{5}$$

The value of ΔM_p is kept in a separate variable because it is required for the recursive part of the algorithm, so instead of equation (1) further evaluations of the fuzzy neural net use the following equation

$$y = \frac{\sum_{j=1}^{N} \mu_j(x)(M_j + \Delta M_j)x}{\sum_{k=1}^{N} \mu_k(x)} . \tag{6}$$

Now assume there is some new data stacked in matrices X_{new}, Y_{new}, and Y_{new}^d, similar to the data contained in *X*, *Y*, and Y^d.

Equation (5) can be rewritten as

$$\Delta M_p (X^T Q^2 X) = \left[Y^d - Y \right]^T QX . \tag{7}$$

To compute the new value of ΔM_p, $\Delta M_{p,new} = \Delta M_p + \delta M_p$, considering new and old data, the following replacements need to be made in (7):

ΔM_p by $\Delta M_{p,new} = \Delta M_p + \delta M_p$,

X by $\begin{bmatrix} X \\ X_{new} \end{bmatrix}$,

Q by $\begin{bmatrix} Q & 0 \\ 0 & Q_{new} \end{bmatrix}$,

Y by $\begin{bmatrix} Y \\ Y_{new} \end{bmatrix}$, and

Y^d by $\begin{bmatrix} Y^d \\ Y^d_{new} \end{bmatrix}$.

Therefore,

$$(\Delta M_p + \delta M_p)\begin{bmatrix} X \\ X_{new} \end{bmatrix}^T \begin{bmatrix} Q^2 & 0 \\ 0 & Q^2_{new} \end{bmatrix} \begin{bmatrix} X \\ X_{new} \end{bmatrix} = \left(\begin{bmatrix} Y^d \\ Y^d_{new} \end{bmatrix} - \begin{bmatrix} Y \\ Y_{new} \end{bmatrix} \right)^T \begin{bmatrix} Q & 0 \\ 0 & Q_{new} \end{bmatrix} \begin{bmatrix} X \\ X_{new} \end{bmatrix}. \quad (8)$$

Using (7) and solving for δM_p,

$$\delta M_p = \left[Y^d_{new} - Y_{new} - Q_{new} X_{new} \Delta M_p^T \right]^T Q_{new} X_{new} \left(X^T Q^2 X + X^T_{new} Q^2_{new} X_{new} \right)^{-1}. \quad (9)$$

Notice that the information required from old data in equation (9) is given by matrices $X^T Q^2 X$ and ΔM_p, which can be updated and stored each iteration. A problem with (9) is that the matrix $X^T Q^2 X$ will increase without bound and new data will have less relative importance in the least squares minimization problem given that the amount of data available increases. To avoid this problem, an improvement can be made to (9) to include a forgetting factor for old data. That is, assume that old data represented by the matrices X, Y, and Y_d are weighted by a forgetting factor γ in (3). This will lead to the following modification in (9),

$$\delta M_p = \left[Y^d_{new} - Y_{new} - Q_{new} X_{new} \Delta M_p^T \right]^T Q_{new} X_{new} \left(\gamma X^T Q^2 X + X^T_{new} Q^2_{new} X_{new} \right)^{-1}. \quad (10)$$

Notice that in (10) the effect of the matrix $X^T Q^2 X$ will be bounded by the proper choice of the forgetting factor γ, with $\gamma < 1$. After computing δM_p from (10), ΔM_p and $X^T Q^2 X$ are updated so $\Delta M_p \leftarrow \Delta M_p + \delta M_p$, and $X^T Q^2 X \leftarrow \gamma X^T Q^2 X + X_{new}^T Q_{new}^2 X_{new}$.

In summary, the recursive least squares learning algorithm is applied as follows

- All the rules are initialized for the FNN using structure learning, as described in [48, 65]. From this initialization each rule has an initial matrix M_p.
- Based on an initial training set the least squares method is applied to compute correction matrices ΔM_p using equation (5). The input/output data are pre-classified based on the values of the membership functions μ_{pi} before they are used by the least squares algorithm, i.e. an input/output pair (x_i, y_i^d) is considered for training of rule *p* only if $\mu_{pi} > \mu_{trheshold}$, where $\mu_{threshold}$ is a threshold value for the membership function values (usually 0.5). Then use equation (6) instead of equation (1) for further evaluations of the fuzzy neural net output.
- Apply the recursive least squares method to compute correction matrices δM_p corresponding to the new input/output pair (x_i, y_i^d) using equation (10). This correction is applied to rules for which $\mu_{pi} > \mu_{trheshold}$. Let $\Delta M_p \leftarrow \Delta M_p + \delta M_p$ and $X^T Q^2 X \leftarrow \gamma X^T Q^2 X + X_{new}^T Q^2 X_{new}$ for p=1,…,N. These values are stored and used for ΔM_p and $X^T Q^2 X$ the next iteration.
- Repeat last step as required.

3.3 Learning Based on Adjustment of Output Values

In some situations it may be required to adjust FNN parameters so that the output given a certain input matches a prescribed value without interfering with the learning algorithms already described. These situations may arise when it is needed that the FNN learn a high confidence value. Instead of waiting for many iterations of back propagation or least squares learning for the FNN to learn this value, it is possible to adjust directly consequent matrices of existing rules or perform structure learning to learn the value in one iteration. It is important to reiterate that this learning method can only be applied when there is high confidence in the training data. The motivation this method was one of the new algorithms developed for the adaptive mode transition control called automatic trimming, discussed later in Chapter 5.

Let us assume the FNN needs to be adjusted so a high confidence input/output pair (x_i, y_i^d) is represented, i.e. when presented with input vector x_i, the output will be exactly y_i^d with minimal modification of the outputs produced by inputs far enough from in the input space. Two situations are possible:

- The FNN is not sufficiently excited by input vector x_i, i.e. all current input membership functions evaluated in x_i are below the threshold specified for structure learning. In this case just perform structure learning using the input output pair (x_i, y_i^d), a new rule will be created and the output will match y_i^d for input x_i.
- The FNN is sufficiently excited by input x_i. In this case adjust consequent parameters of current rules as described in the sequel.

Let y_i be the output produced by the FNN when presented with input x_i, $\Delta y_i = y_i^d - y_i$ be the correction required in y_i and ΔM_j be the corrections required in the consequent parameters M_j such that the new output of the FNN matches y_i^d. Hence, using equation (1) the following expression is obtained

$$\Delta y_i = \frac{\sum_{j=1}^{N} \mu_j(x_i) \Delta M_j x_i}{\sum_{k=1}^{N} \mu_k(x_i)}. \tag{11}$$

The problem here is finding suitable values of parameters ΔM_j satisfying equation (11). Let $\Delta y_{ji} = \Delta M_j x_i$, so (11) can be rewritten as

$$\sum_{k=1}^{N} \mu_k(x_i) \Delta y_i = \sum_{j=1}^{N} \mu_j(x_i) \Delta y_{ji}\ . \tag{12}$$

Now choose

$$\Delta y_{ji} = \mu_j(x_i) a(x_i) \Delta y_i\,, \tag{13}$$

where $a(x_i)$ is a function to be determined. This pick is arbitrary, but it makes sense since it means that the output correction of the consequent of each rule is proportional to the level of excitation of that rule, represented by the input membership function value $\mu_j(x_i)$.

From equations (12) and (13) it follows that

$$a(x_i) = \frac{\sum_{k=1}^{N} \mu_k(x_i)}{\sum_{k=1}^{N} \mu_k^2(x_i)},$$

hence,

$$\Delta y_{ji} = \mu_j(x_i) \frac{\sum_{k=1}^{N} \mu_k(x_i)}{\sum_{k=1}^{N} \mu_k^2(x_i)} \Delta y_i \tag{14}$$

Equation (14) gives the correction required in the consequent output of rule *j*. To achieve this, a suitable correction in the consequent parameter of rule *j* will be

$$\Delta M_j = \left[\Delta y_{ji} \quad \vdots \quad O \right] \tag{15}$$

After evaluating equations (14) and (15) for each $j = 1,...,N$, consequent parameters are adjusted so $M_j \leftarrow M_j + \Delta M_j$.

CHAPTER 4

ACTIVE MODELING FRAMEWORK

The active plant models are key components of the adaptive mode transition control architecture that will be presented in Chapter 5. These models represent the nonlinear dynamics of the plant under control and are adapted on-line to changes in those dynamics. This chapter is devoted to explain how these models represent the dynamics of the plant, how to incorporate known nonlinearities in the models for the case of vehicles in 3D space, and how incremental models are obtained from them to enable the control adaptation that will be discussed in Chapter 5.

4.1 Active Plant Models for a Nonlinear Discrete Time System

Consider a continuous time nonlinear system represented by the state equation

$$\dot{x}_c(t) = f_c(x_c(t), u_c(t)), \text{ with initial condition } x_c(0) = x_0. \tag{16}$$

$x_c(t)$ is the continuous time state vector, $u_c(t)$ is the continuous time input vector, and f_c is a nonlinear function representing the nonlinear dynamics of the system. Given that a digital controller is going to be used to control this system, an equivalent discrete time model representation will be more appropriate for control design purposes. The discrete time version of the model, so called discretized model, can be represented by

$$x(k+1) = f(x(k), u(k)), \text{ with } x(0) = x_0, \tag{17}$$

where $x(k)$ and $u(k)$ are discrete time versions of $x(t)$ and $u(t)$ given by

$$x(k) = x_c(kT)$$
$$u(k) = u_c(kT)$$

with sample period T, and f is a nonlinear function representing the discrete time version of the dynamics of the system. It is assumed that the sample period is chosen small enough to be able to capture the continuous dynamics in (16).

In this research, the model of the plant is discretized like in equation (17), with function f approximated by a FNN. The FNN construct enables adaptation in the model, so the model can be trained off-line and adapted on-line through the structure and parameter learning algorithms discussed in Chapter 3. In the adaptive mode transition control scheme presented in Chapter 5, the state space is partitioned in local modes and transition regions. That is the reason why instead of using a single FNN, several FNNs are used to represent the dynamics of the plant across the whole operating region of the system. The FNN representing the dynamics of the system in one of the regions is what is named an active plant model. Therefore, assuming there is a mechanism to determine the region of operation of the plant at each sample time, only the plant model corresponding to that region will be active at that instant. This multi model approach improves the computational efficiency of the overall model since each FNN has a smaller structure.

According to the discussion above, the active plant model l is represented by

$$x(k+1) = FNN_{APM_l}(x(k), u(k)) \text{ , with } x(0) = x_0 \text{ ,} \tag{18}$$

where FNN_{APM_l} is the function representing the nonlinear mapping generated by the FNN associated to that model.

4.2 Incorporation of Known Nonlinearities in Active Plant Models for Vehicles in 3D Space

In some situations exact knowledge about some nonlinearities in the model of the plant under control exists. Whenever possible, it is a good idea to incorporate that knowledge into the active plant models to reduce the size of associated FNNs and speed up their learning. This happens for instance in the case of vehicles in 3D space where there is exact knowledge of vehicle kinematics.

For the Unmanned Aerial Vehicle considered in this work, the state vector was chosen as

$$x(k) = [x, y, z, \phi, \theta, \psi, \dot{x}, \dot{y}, \dot{z}, \dot{\phi}, \dot{\theta}, \dot{\psi}, \Omega]^T, \tag{19}$$

where $[x, y, z]^T$ represent the position in earth frame, $[\dot{x}, \dot{y}, \dot{z}]^T$ is the velocity, $[\phi, \theta, \psi]^T$ is the attitude in term of Euler angles, $[\dot{\phi}, \dot{\theta}, \dot{\psi}]^T$ are the Euler angle rates, and Ω represents the angular velocity of the main rotor. The reason for choosing the state variables this way is that there is a linear relationship between the variables representing position and velocity, i.e. the later is just the derivative of the former, facilitating the model linearization.

From a modeling point of view, the best choice for the state variables should involve the velocity in body frame $[U, V, W]^T$ and the angular rates $[P, Q, R]^T$. Therefore, given that the nonlinear relationships between these variables and the ones included in the state are exactly known, the active plant models use the variables in body frame. The gravity effect was also considered separately since it is almost exactly known. Hence, FNNs in the active plant models were used to approximate aerodynamic and

propulsive forces in the vehicle representing the most uncertain part in the model of the vehicle.

According to the discussion above, the active plant models approximate the model of the vehicle as

$$x_B(k+1) = g_B(\phi(k),\theta(k)) + FNN_{ACM_l}(\phi(k),\theta(k),x_B(k),u(k)), \quad (20)$$

where $x_B(k) = [U,V,W,P,Q,R,\Omega]$ includes the velocity in body frame, the angular rates, and the angular velocity of the main rotor at instant *k*, $\phi(k)$ and $\theta(k)$ are roll and pitch angles at instant *k*, and $u(k) = [\delta_t,\delta_c,\delta_{mp},\delta_{mr},\delta_p]$ is the control input to the vehicle at instant *k*. The known gravity effect in body frame is given by

$$g_B(\phi,\theta) = [-Tg\sin(\theta), Tg\sin(\phi)\cos(\theta), Tg\cos(\phi)\cos(\theta),0,0,0,0]^T, \quad (21)$$

where *g* is the acceleration of gravity and *T* is the sample period. In (20) dependencies on the position were dropped given that for a UAV flying at low altitude the variation of the model with altitude is minimal. Dependency on the heading angle ψ was also dropped using always a reference frame rotated about the z axis such that ψ is always zero respect to that frame.

4.3 Computation of Incremental Models from Active Plant Models

For a plant represented by equation (17), it is possible to find an incremental (or linearized) model about the operating conditions (x_*,u_*) such that

$$x(k+1) = f(x(k),u(k)) \approx \Phi(x(k)-x_*) + \Gamma(u(k)-u_*) + f(x_*,u_*), \quad (22)$$

where

$$\Phi = \left.\frac{\partial f(x(k),u(k))}{\partial x(k)}\right|_{x_*,u_*}, \tag{23a}$$

$$\Gamma = \left.\frac{\partial f(x(k),u(k))}{\partial u(k)}\right|_{x_*,u_*}. \tag{23b}$$

If the system is modeled by active plant models following equation (18), then the incremental model can be approximated by

$$x(k+1) \approx \Phi(x(k)-x_*)+\Gamma(u(k)-u_*)+FNN_{APM_l}(x_*,u_*), \tag{24}$$

with

$$\Phi = \left.\frac{\partial FNN_{APM_l}(x(k),u(k))}{\partial x(k)}\right|_{x_*,u_*}, \tag{25a}$$

$$\Gamma = \left.\frac{\partial FNN_{APM_l}(x(k),u(k))}{\partial u(k)}\right|_{x_*,u_*}, \tag{25b}$$

given that active plant model l is active.

For the Unmanned Aerial Vehicle considered in this work, given that the state vector was chosen according to (19) and the active plant models follow equation (20), computing the incremental models require a small amount of work. Define

$$f_B(\phi,\theta,x_B,u) \equiv g_B(\phi,\theta)+FNN_{ACM_l}(\phi,\theta,x_B,u), \tag{26}$$

and let

$$x_h = [\dot{x},\dot{y},\dot{z},\dot{\phi},\dot{\theta},\dot{\psi},\Omega]^T,$$

there exist transformation matrices T_{hB} and T_{Bh} such that

$$\begin{aligned} x_h &= T_{hB}x_B, \\ x_B &= T_{Bh}x_h. \end{aligned} \tag{27}$$

From (20), (26), and (27) the following is obtained

$$x_h(k+1) = T_{hB} f_B(\phi(k), \theta(k), T_{Bh} x_h(k), u(k)) \,. \tag{28}$$

Notice that $x_h(k)$ is the lower part of the state vector $x(k)$, so equation (28) captures the nonlinear dynamics represented by the active plant models, according to equation (20), transformed in terms of the state vector $x(k)$.

Let us define

$$\Phi_h = \frac{\partial T_{hB} f_B(\phi(k), \theta(k), T_{Bh} x_h(k), u(k))}{\partial x(k)},$$

$$\Gamma_h = \frac{\partial T_{hB} f_B(\phi(k), \theta(k), T_{Bh} x_h(k), u(k))}{\partial u(k)}.$$

Then the incremental model for the vehicle can be represented by

$$x(k+1) = \Phi(x(k) - x_*) + \Gamma(u(k) - u_*) + f_*, \tag{29}$$

with

$$\Phi = \begin{bmatrix} M \\ \cdots \\ \Phi_h \end{bmatrix}, \qquad \Gamma = \begin{bmatrix} 0 \\ \cdots \\ \Gamma_h \end{bmatrix}, \tag{30a}$$

and

$$f_* = \begin{bmatrix} x_* + T\dot{x} \\ y_* + T\dot{y} \\ z_* + T\dot{z} \\ \phi_* + T\dot{\phi} \\ \theta_* + T\dot{\theta} \\ \psi_* + T\dot{\psi} \\ \cdots \\ T_{hB} f_B(\phi_*, \theta_*, T_{Bh} x_{h*}, u_*) \end{bmatrix}, \tag{30b}$$

where M represents the integrators for the linear part of the model (relationship between $[x, y, z, \phi, \theta, \psi]^T$ and its derivative $[\dot{x}, \dot{y}, \dot{z}, \dot{\phi}, \dot{\theta}, \dot{\psi}]^T$), that is

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & T & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & T & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & T & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & T & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & T & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & T & 0 \end{bmatrix}.$$

CHAPTER 5

OVERALL ARCHITECTURE FOR CONTROL OF UNMANNED AERIAL VEHICLES

In this chapter a new approach to the adaptive mode transition control problem and a hierarchical architecture to implement it are presented. The architecture is flexible enough to enable the future integration of additional intelligent attributes at the high level, for instance the introduction of a situation awareness module and collision avoidance algorithms.

The proposed architecture for control of UAVs consists of a hierarchy of three levels (Figure 3). At the highest level, a mission planning component stores information about the overall mission, generates a low level representation of that mission, and coordinates its execution with the middle level. The middle level includes a trajectory generation component, which receives information from the high level in terms of the next task to be executed to fulfill the mission, and generates the trajectory (set points) for the low-level controller. At the lowest level, an adaptive mode transition controller coordinates the execution of the local controllers and the active control models, which stabilize the vehicle and minimize the errors between the set points generated by the middle level and the actual state of the vehicle. A more detailed description of each level as applied to the case of a rotary wing UAV is given below.

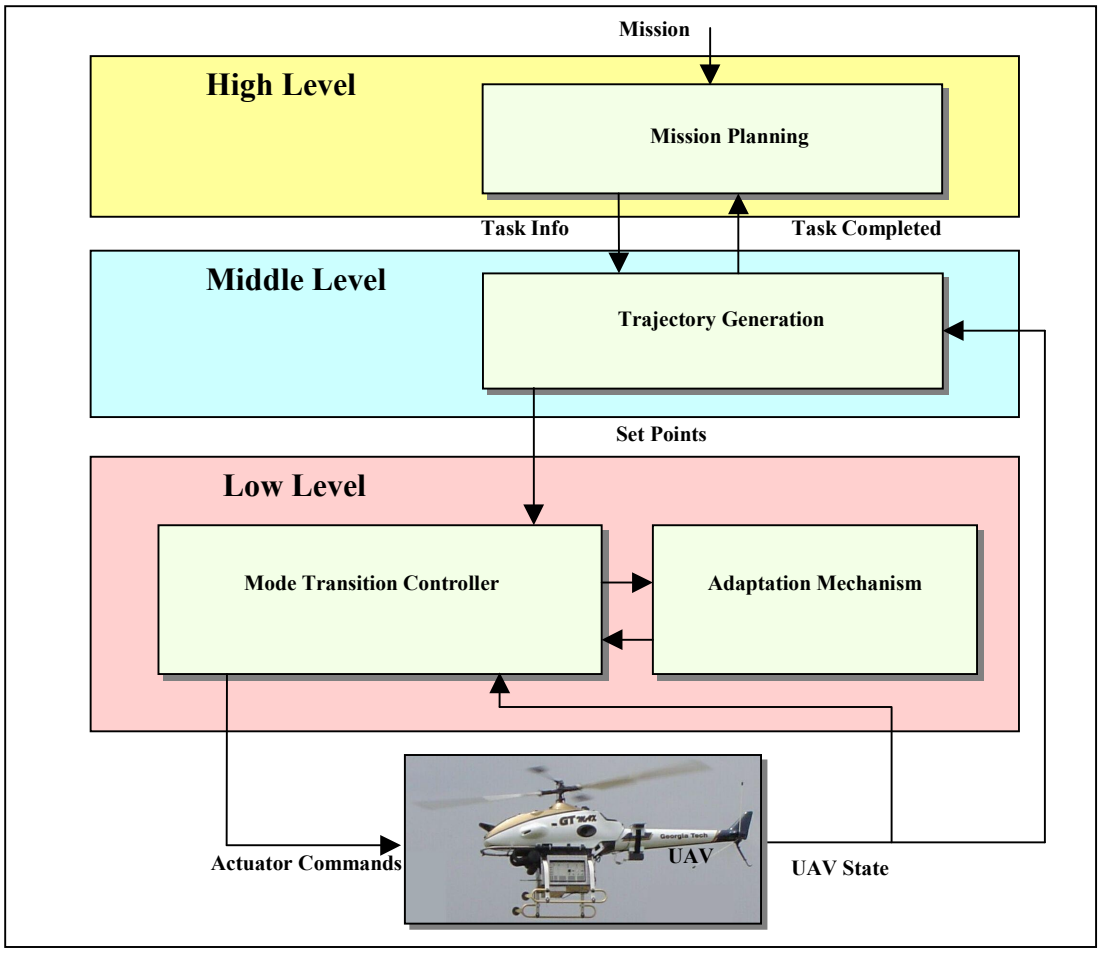


Figure 3. Overall Architecture for the Adaptive Mode Transition Control

5.1 High level: Mission Planning

The mission planning component translates a high level representation of the mission into a low-level task queue and coordinates the execution of low-level tasks with the trajectory generation component at the middle level. The mission can be established as a sequence of actions to be executed, for instance: fly to a way point and hover there, fly to a way point at certain speed, keep the same velocity and heading for a certain period of time, etc. Kinematical constraints like maximum speed and acceleration are specified and can be changed for each section of the mission.

Every action is specified through a high level command given to the mission planning module. When a new action is suggested, the sequence of tasks that must be performed are generated and added to the tail end of the task queue. Each task represents a maneuver that takes the vehicle from the actual state to a target state and includes the following information:

- Time to complete the task
- Target position
- Target direction of the flight path for this task
- Target heading angle
- Heading mode: specifies whether the value of the heading is absolute or relative to the direction of the flight path for coordinated flight
- Target speed
- Maximum acceleration

A mission may be completely specified before it is executed but may also be modified, re-planned, or expanded at run time. This feature enables the modification or

extension of the mission at run time. Re-planning is particularly important for the future incorporation of obstacle and collision avoidance algorithms. The functionality of the mission planning component is illustrated in Figure 4.

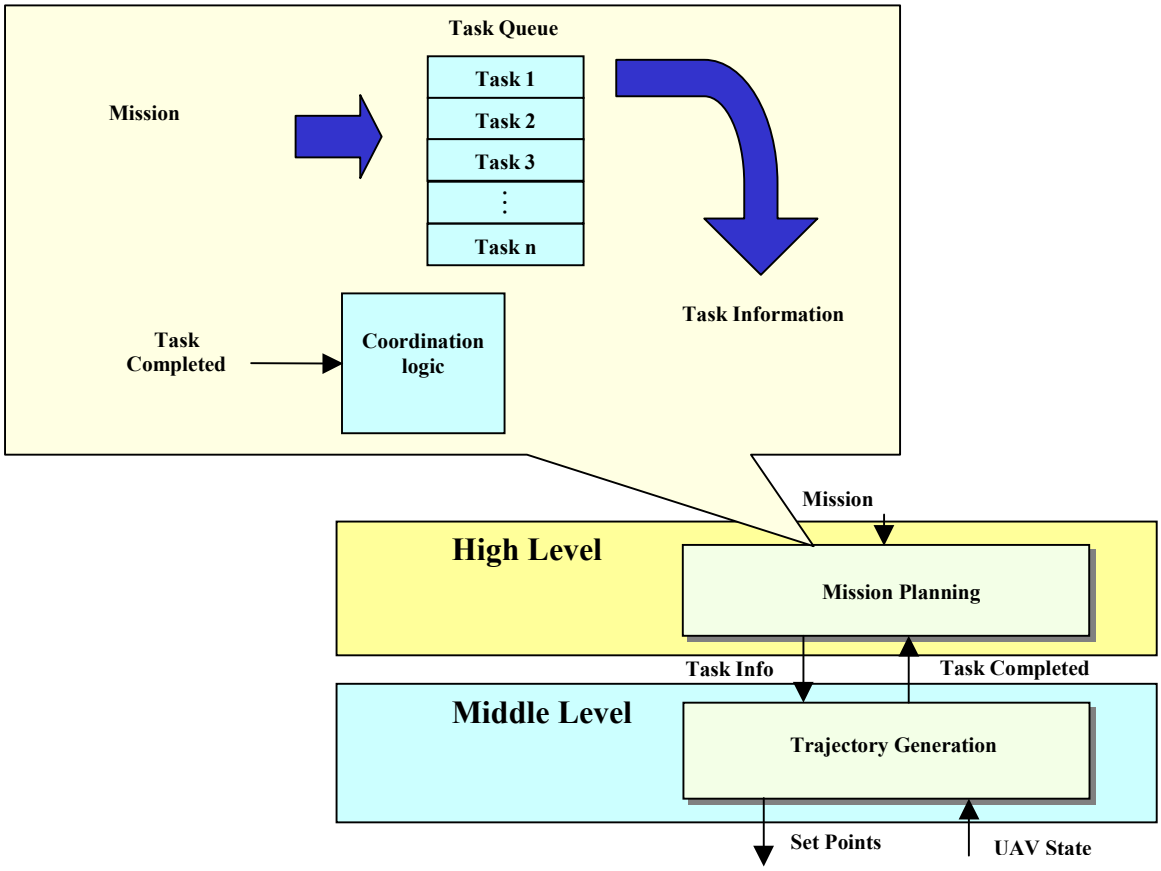


Figure 4. Mission Planning Component Functionality

At run time, the mission planning component coordinates the execution of the low-level tasks with the trajectory generation component at the middle level in the following way: first, the mission planning component takes the task at the head of the task queue, removes it from the queue and sends the task information to the trajectory generation component. Then, the trajectory generation component executes the task and,

when completed, sends a signal back to the mission planning component indicating that the last task has been completed. Finally, when the mission planning component receives the signal, it sends the task at the head of the task queue and the cycle is repeated until no tasks remain in the task queue.

5.2 Middle Level: Trajectory Generation

The trajectory generation component generates the set points required for the low-level controllers to complete the most recent task received from the mission planning module. When the trajectory generation component receives the next task information, it computes a 3D spline to generate a continuous path linking the actual position with the target position (Figure 5). At each sample time, the actual speed is evaluated based on the initial speed for the task, the final speed for the task, and the maximum acceleration available according to the curvature of the path at that time. Position and velocity over the path are computed next using the spline representation. A similar spline is used for the heading of the vehicle and is used according to the heading mode. The heading can be determined in two different ways according to the heading mode defined for the task: either directly from the heading spline if the heading mode is set to the absolute heading, or from a combination of the heading spline and the heading computed from the direction of the path, if the heading mode is set to coordinated heading. Also the heading rate is computed in a consistent manner.

After generating the set points corresponding to the actual sample time, a comparison is made with the actual state of the vehicle to determine if the task was completed successfully or not. A signal is sent to the high level module indicating the

termination status of the task so that the next one can be initiated. For instance, when the mission planning component at the high level receives a signal of successful termination of the task, it retrieves the next task information from the task queue and sends it to the trajectory generation component at the middle level, so the trajectory generation continues smoothly. When the trajectory generation component completes a task and does not receive a new task to perform from the mission planning component, it generates set points consistent with the last set point, i.e. it maintains the same speed, path direction and heading, and computes the positions accordingly. The functionality of the trajectory generation component is illustrated in Figure 5.

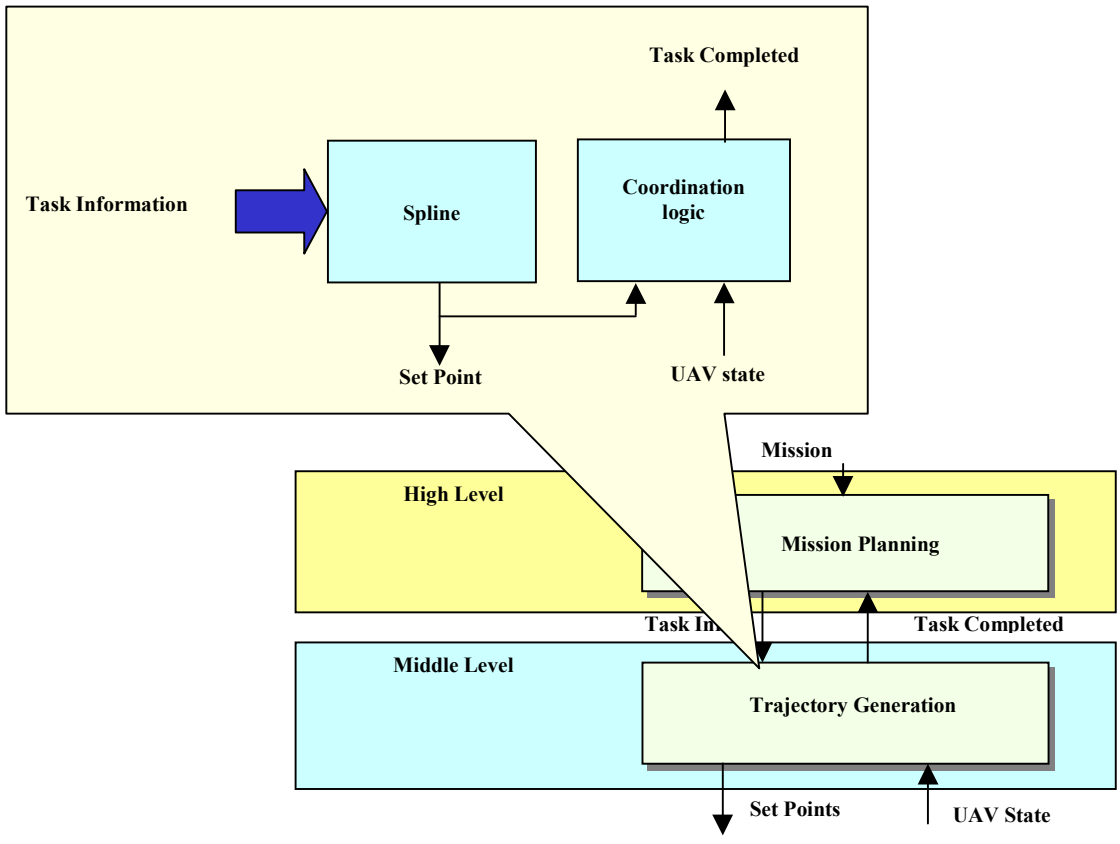


Figure 5. Trajectory Generation Component Functionality

5.3 Low Level: Adaptive Mode Transition Control

The purpose of the low level controllers is to stabilize the vehicle and force it to follow accurately the commanded trajectory generated by the middle level. In this architecture, a new approach to the adaptive mode transition control is introduced. The adaptive mode transition control consists of the mode transition control component and the adaptation mechanism component (Figure 6). The following description refers to the case of a rotary wing UAV.

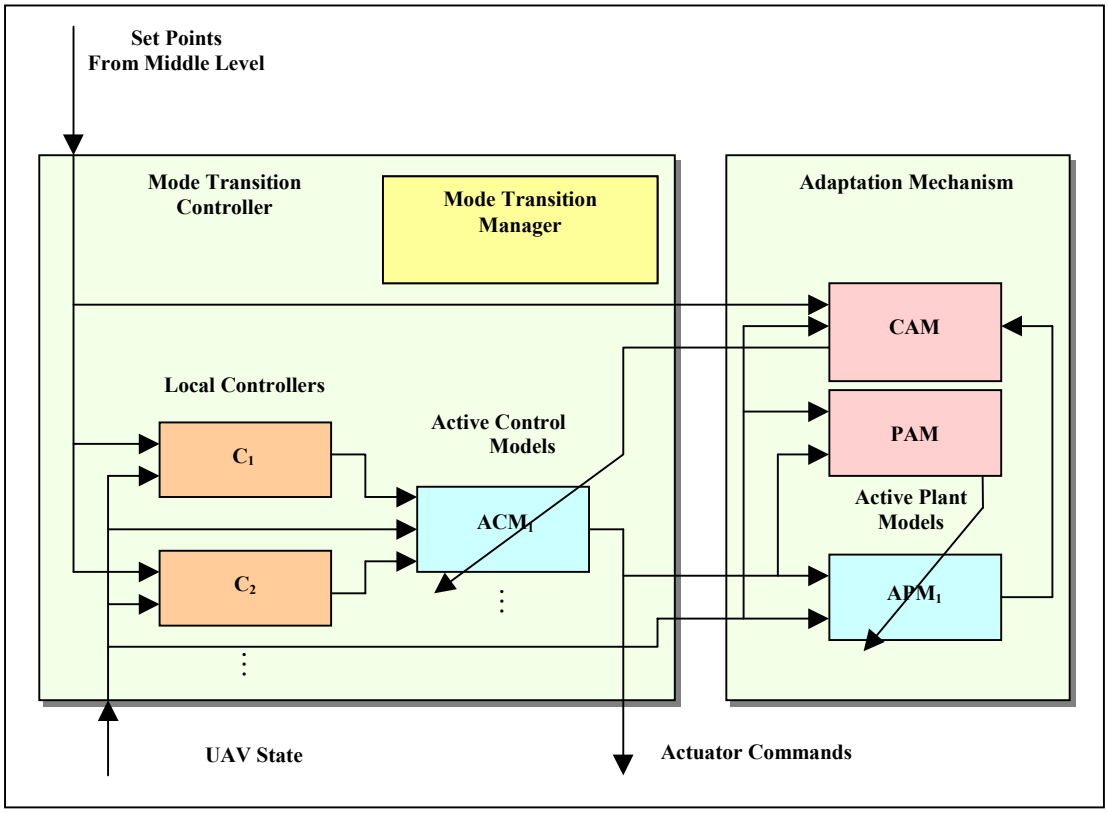


Figure 6. Structure of the Adaptive Mode Transition Control

5.3.1 Mode Transition Control Component

The mode transition control component consists of several subcomponents (Figure 7): a set point filter, a state filter, a set of local controllers (one for each local mode), a set of active control models (one for each transition), the mode transition manager, the automatic trimming mechanism, and a dynamic compensation filter. The mode transition manager decides which controller to use at a given time (a local controller or an active control model) based on the actual state of the UAV. The mode transition control by itself does not perform any adaptation on local controller gains nor in the blending gains of active control models; however, it tunes the trim values of local controllers using a new automatic trimming mechanism described later.

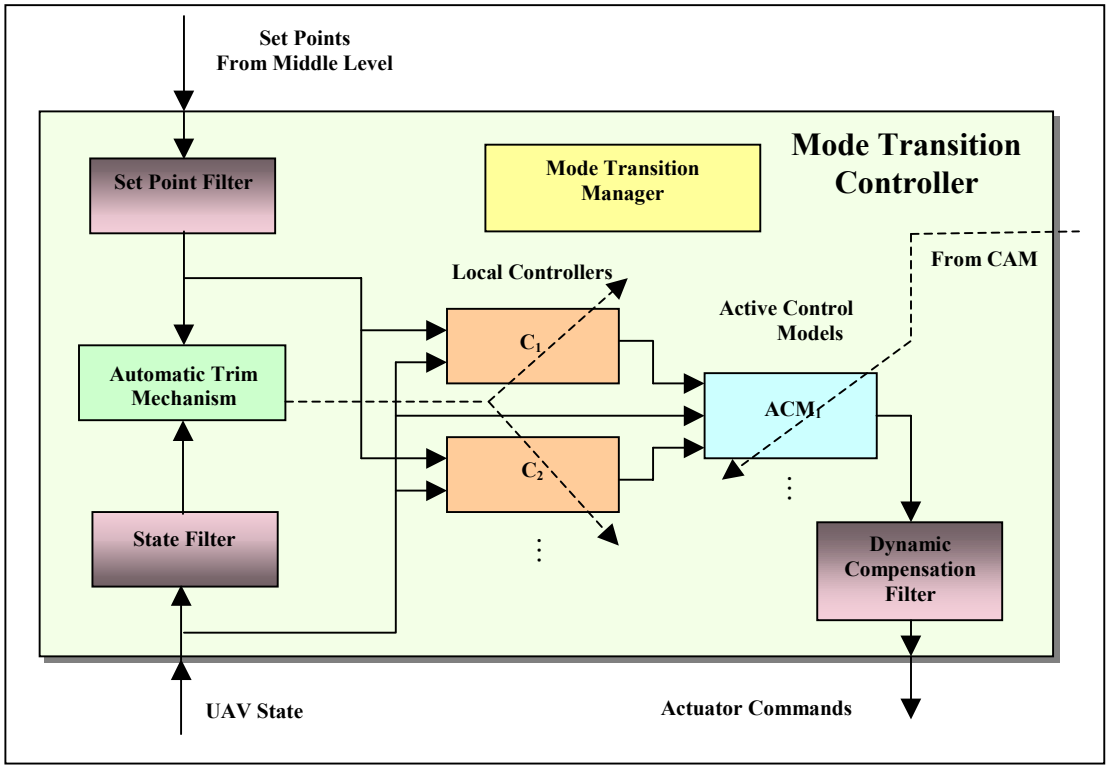


Figure 7. Mode Transition Controller

5.3.1.1 The Set Point Filter

This new component was introduced to guarantee acceleration and velocity constraints and also sufficient smoothness and consistency in the set points used by local controllers. The set point filter is composed of the series of two filters as shown in Figure 8. First, a limiting filter guarantees consistency in position and velocity set points and enforces acceleration and velocity limits. Second, a smoothing filter smoothes the set points, keeping the consistency between positions and velocities.

To illustrate the processing performed by these filters, let us consider the position and velocity in x-axis, x and $\dot{x}$ respectively. The same applies to the pairs $(y, \dot{y})$, $(z, \dot{z})$, $(\phi, \dot{\phi})$, $(\theta, \dot{\theta})$, and $(\psi, \dot{\psi})$.

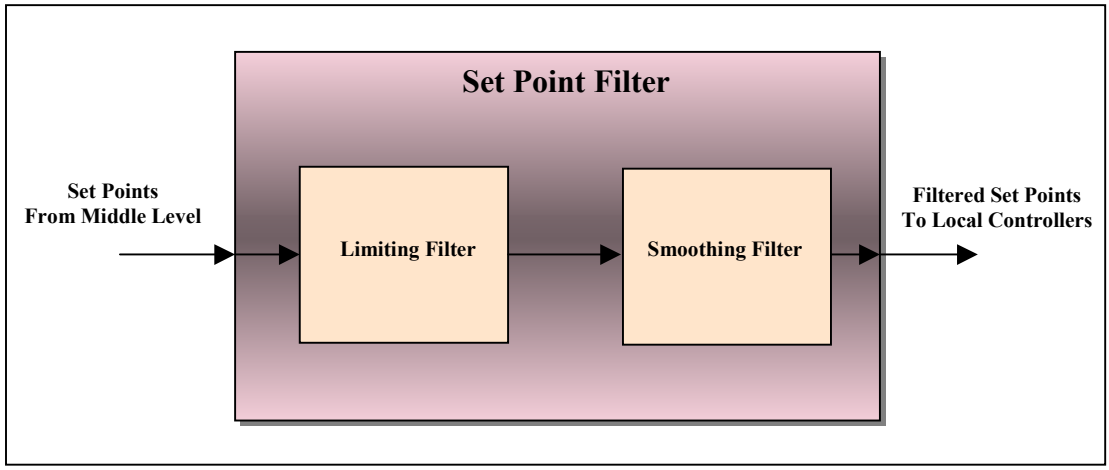


Figure 8. Set Point Filter

The limiting filter is composed of an inner loop and an outer loop as seen in Figure 9. The inner loop, fed with the desired velocity $\dot{x}_d$, is simply a first order low pass filter with cutoff frequency w_c, which includes saturation before and after the integrator

to guarantee limits in the acceleration and velocity respectively. The outer loop feeds the error in position multiplied by a gain G to the inner loop and uses an integrator to generate a consistent position set point. Assuming there is no saturation, the transfer function of the limiting filter is

$$H_l(s) = \frac{Gw_c}{s^2 + w_c s + Gw_c}. \tag{31}$$

Therefore, to achieve an equivalent damping factor ζ, the gain G is chosen as

$$G = \frac{w_c}{4\zeta^2}, \tag{32}$$

and the resulting cutoff frequency of the limiting filter is $\frac{w_c}{2\zeta}$. When there is saturation given that one or both limits are hit, the set point filter will try to reach the desired position x_d as fast as possible, but respecting maximum acceleration and velocity limits.

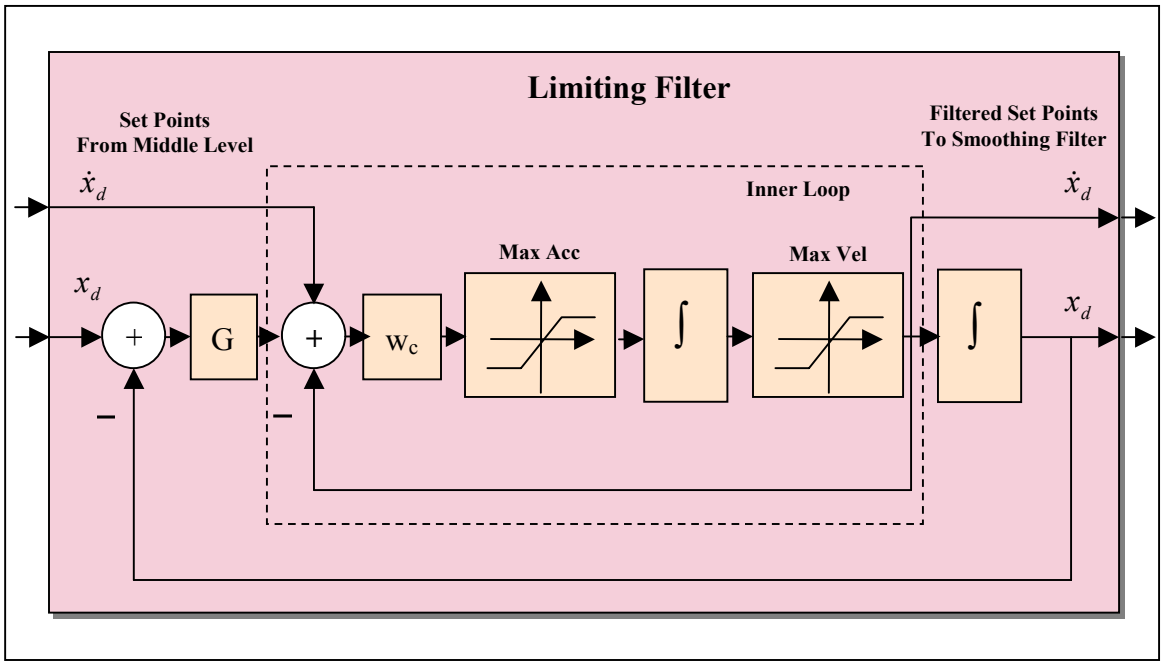


Figure 9. Limiting Filter

The smoothing filter is chosen as a fourth order low pass Butterworth filter with transfer function

$$H_s(s) = \frac{a_0}{s^4 + a_3 s^3 + a_2 s^2 + a_1 s + a_0}. \tag{33}$$

The cutoff frequency for this filter is chosen as the same w_c used in the limiting filter. Given that this is an all pole filter, the output results from a cascade of integrators generating consistent values of velocity and acceleration, as shown in Figure 10.

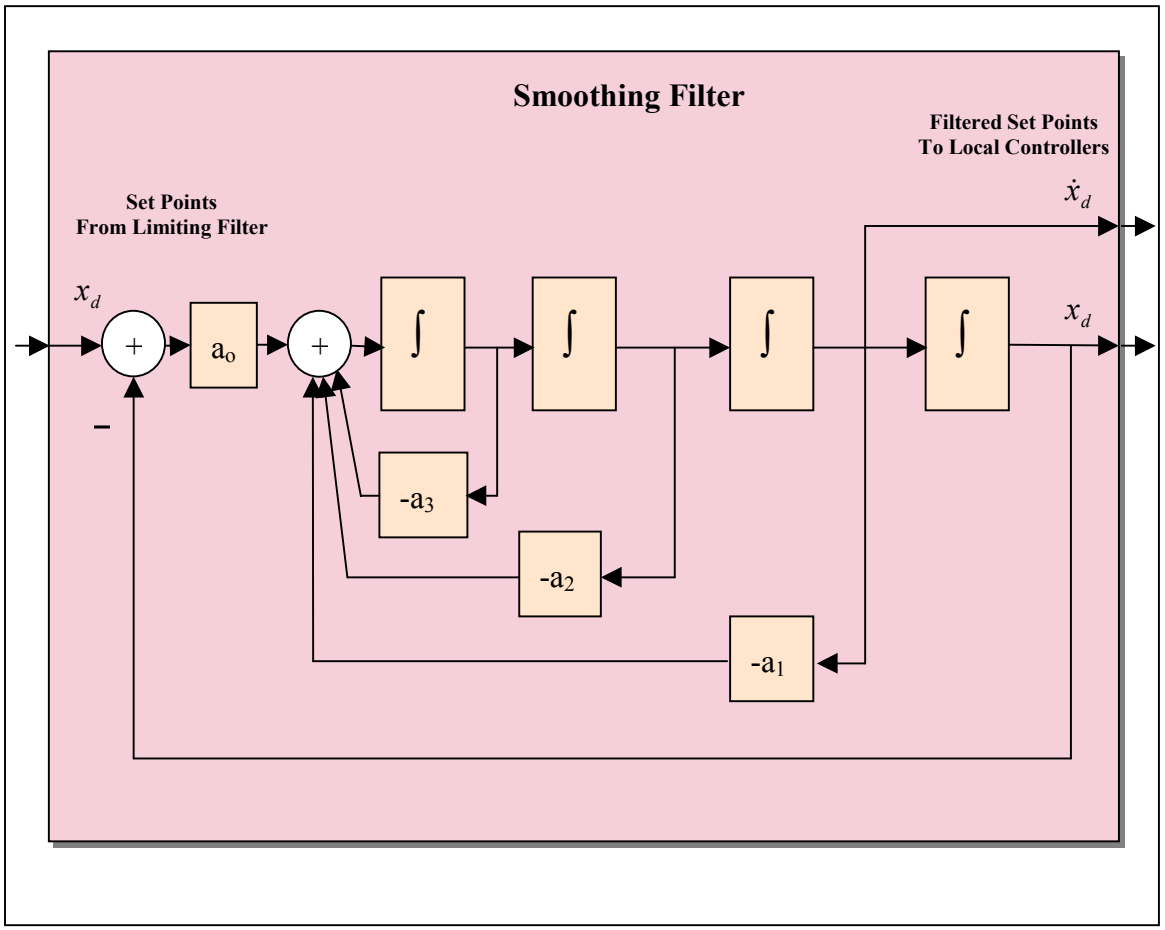


Figure 10. Smoothing Filter

Each integrator used in the filters presented here is discretized using the bilinear transformation, equivalent to a trapezoidal integrator with a correction in the sample period to avoid shifting of the cutoff frequency. The transfer function of each discretized integrator is

$$H_I(z) = \frac{T_c}{2}\frac{1+z^{-1}}{1-z^{-1}}, \tag{34}$$

with

$$T_c = \frac{\tan\left(\frac{Tw_c}{2}\right)}{\frac{w_c}{2}}, \tag{35}$$

where T is the sample period.

5.3.1.2 The State Filter

The purpose of the state filter is to generate values of velocity and acceleration consistent with the measurement of the state of the plant. These values are used by the automatic trimming mechanism discussed later, but they are not used by the local controllers since the lag generated could affect the stability of the control loop. The state filter is a second order low pass Butterworth filter with transfer function

$$H_I(s) = \frac{b_0}{s^2 + b_1 s + b_0}. \tag{36}$$

The state filter is shown in Figure 11. The cutoff frequency of this filter is set to eliminate as much noise as possible from the state without significantly affecting the estimate of the velocity and acceleration. Integrators in this filter are discretized in the same way as it was discussed for the set point filter.

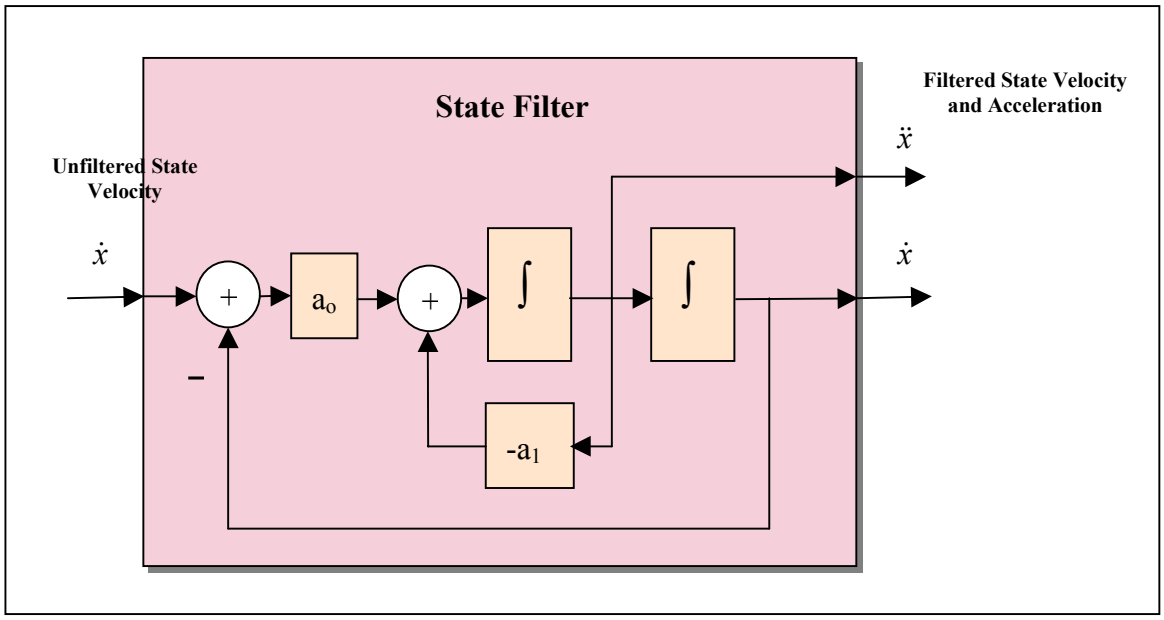


Figure 11. State Filter

5.3.1.3 Local Controllers

In this new approach, the local controllers are discrete linear quadratic trackers running at a fixed sample rate. The control law for these controllers is given by

$$u(k) = K_i e(k) + u_{trim,i} \quad , \tag{37}$$

where k represents the discrete time, $u(k)$ is the actuator command vector, $e(k)$ is the error between the desired state (set point) generated by the trajectory generation component and processed by the set point filter ($x_d(k)$) and the actual state of the vehicle obtained from on-board sensors ($x(k)$). The parameters for local controller i are the gain matrix K_i, and the trim value of the actuator command $u_{trim,i}$. $u_{trim,i}$ is adjusted on-line by the automatic trimming mechanism discussed latter.

The state of the vehicle is given by

$$x(k) = [x, y, z, \phi, \theta, \psi, \dot{x}, \dot{y}, \dot{z}, \dot{\phi}, \dot{\theta}, \dot{\psi}, \Omega]^T \ ,$$

where

x : x-position (ft, measured northwards)

y : y-position (ft, measured eastwards)

z : z-position (ft, measured downwards)

ϕ : roll angle (rad)

θ : pitch angle (rad)

ψ : yaw angle (rad)

$\dot{x}$: x-velocity (ft/sec)

$\dot{y}$: y-velocity (ft/sec)

$\dot{z}$: z-velocity (ft/sec)

$\dot{\phi}$: roll angle derivative (rad/sec)

$\dot{\theta}$: pitch angle derivative (rad/sec)

$\dot{\psi}$: yaw angle derivative (rad/sec)

Ω : rotor angular velocity (rpm)

The actuator command vector is given by

$$u(k) = [\delta_t, \delta_c, \delta_{mp}, \delta_{mr}, \delta_p]^T \ ,$$

where:

δ_t : throttle

δ_c : collective

δ_{mp} : longitudinal cyclic (moment actuator for pitch)

δ_{mr} : lateral cyclic (moment actuator for roll)

δ_p : pedal (moment actuator for yaw)

A transformation is performed on $x(k)$ and $x_d(k)$ before the control algorithms are applied, to make them independent of the actual heading of the vehicle. That is, if ψ_x is the actual value of the heading in $x(k)$, then the transformed values are obtained by

$$\begin{aligned} x(k) &\leftarrow T(x(k),\psi_x), \\ x_d(k) &\leftarrow T(x_d(k),\psi_x), \end{aligned} \tag{38}$$

where

$$T(x(k),\psi_x) = \left[x_{\psi_x}, y_{\psi_x}, z, \phi, \theta, \psi - \psi_x, \dot{x}_{\psi_x}, \dot{y}_{\psi_x}, \dot{z}, \dot{\phi}, \dot{\theta}, \dot{\psi}, \Omega\right]^T,$$

with

$$\begin{bmatrix} x_{\psi_x} \\ y_{\psi_x} \end{bmatrix} = A(\psi_x)\begin{bmatrix} x \\ y \end{bmatrix}, \begin{bmatrix} \dot{x}_{\psi_x} \\ \dot{y}_{\psi_x} \end{bmatrix} = A(\psi_x)\begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} \text{, and}$$

$$A(\psi_x) = \begin{bmatrix} \cos(\psi_x) & \sin(\psi_x) \\ -\sin(\psi_x) & \cos(\psi_x) \end{bmatrix}.$$

After the transformation, the tracking error is given by

$$e(k) = x_d(k) - x(k) \ . \tag{39}$$

To improve the tracking performance of the local controllers, they are augmented with an integral part. Therefore, the dynamics of the system is augmented with integrators for position, heading, and rotor angular velocity. This is equivalent to designing the controllers for a system with state vector

$$x(k) = [\int x, \int y, \int z, \int \psi, \int \Omega, x, y, z, \phi, \theta, \psi, \dot{x}, \dot{y}, \dot{z}, \dot{\phi}, \dot{\theta}, \dot{\psi}, \Omega]^T. \tag{40}$$

The error in (39) is computed based on this augmented state vector. The structure of the local controllers is presented in Figure 12.

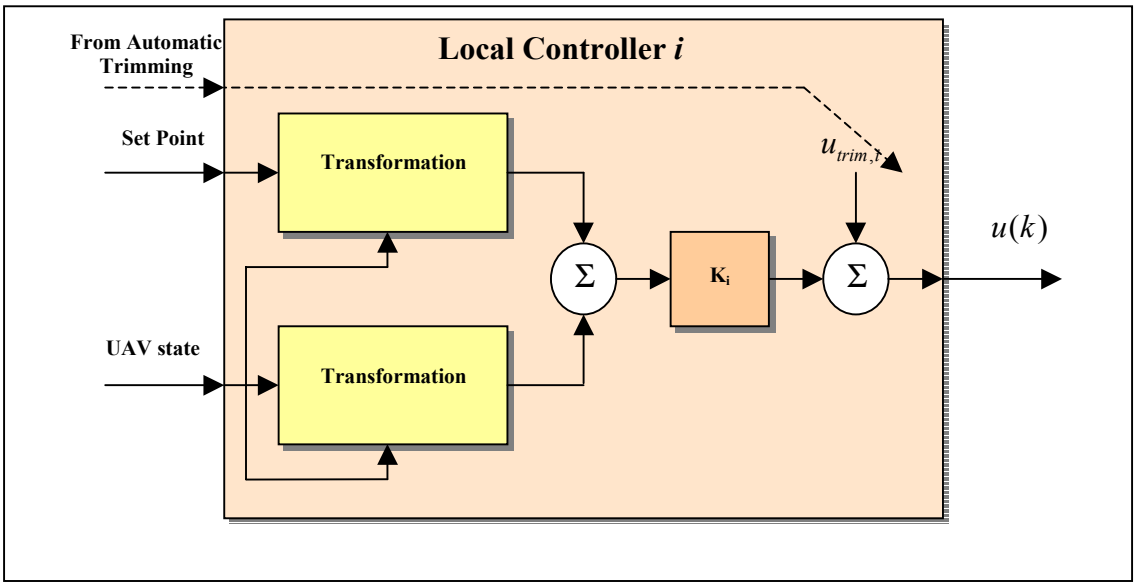


Figure 12. Local Controllers Structure

The design procedure for the local controllers is as follows: once the operating state of a local mode is decided, an approximate model of the vehicle is linearized about that state, and then discretized. Based on the discrete linearized model, augmented with the integrators discussed above, a linear quadratic regulator is computed for the gain matrix K_i. When an approximate model of the vehicle is not available, the linearized model could be obtained from a Fuzzy Neural Net model trained with input/output data from the actual vehicle in the same way it is done with the active plant models to be discussed later.

5.3.1.4 Mode Transition Manager

The mode transition manager (MTM) coordinates the transitions in this new approach. Unlike [45-48] where the transitions were pre-scheduled and a mode selector module coordinated the transitions, the MTM coordinates the transitions automatically in

the new technique based on the actual state of the vehicle. In order to accomplish this task, a mode membership function is defined for each local mode and the MTM determines which local mode or transition should be activated relying upon these constructs.

For local mode *i* the mode membership function is defined as

$$\mu_i = e^{-(x-m_i)^T \Sigma_i^T \Sigma_i (x-m_i)}, \tag{41}$$

where *x* is the state of the vehicle, m_i is the center (operating state) of the mode, and Σ_i is a positive semi-definite diagonal matrix whose elements represent the inverse of the deviations for each component of *x* for that mode.

To determine which mode is active, the MTM computes the mode membership functions for all local modes. If $\mu_l(x(k)) \geq 0.5$ for the current state, then local mode *l* will be active. Mode centers and deviations are defined so that $\mu_l(x(k)) \geq 0.5$ can be valid for only one *l*. That way the modes correspond to disjoint regions of the state space. If $\mu_l(x(k)) < 0.5$ for all *l*, then the transition corresponding to the two modes with the highest mode membership function values will be active.

When a local mode is active, the corresponding local controller is used to compute the control output whereas when a transition is active, the corresponding active control model is used to compute the control output.

5.3.1.5 Active Control Models

The active control models are in charge of the transitions between local modes. The function of an active control model (ACM) is to blend the outputs of the local controllers corresponding to one transition in a smooth and stable way, that is, the

blending of the local controllers should not deteriorate the overall performance of the closed loop system. Every ACM is linked to the local controllers corresponding to the transition, has access to their outputs, and also includes a fuzzy neural net (FNN) that generates the blending gains to compute the control output (Figure 13). The FNN has the same structure as in [45-48], but its learning capabilities have been improved via the recursive least squares learning algorithm discussed in Chapter 3. The FNN input is composed of some of the variables of the actual state of the vehicle, $x(k)$, after the transformation given in (38). Usually the variables included in the FNN input are the *x* and *y* components of the velocity. Therefore, the output of the *l*th ACM module is determined from

$$\begin{aligned} blendingGains_2 &= FNN_{ACM_l}(x(k)), \\ blendingGains_1 &= 1 - blendingGains_2, \end{aligned} \tag{42}$$

$$u(k) = blendingGains_1 u_i(k) + blendingGains_2 u_j(k) \tag{43}$$

where FNN_{ACM_l} represents the function implemented by the FNN of the *l*th ACM, *blendingGains* are the blending gains generated from that FNN, and $u_i(k)$ and $u_j(k)$ represent the control outputs of the local controllers corresponding to the *l*th ACM. The new approach differs from the one presented in [45-48] in that it uses scalar blending gains, while in [45-48] different blending gains are used for each component of $u(k)$.

When a transition is set up, the FNN of the corresponding ACM is trained off-line on the basis of an input/output data set generated automatically from a hypothetical transition trajectory from the center of the initial mode to the center of the target mode. The state is taken from this trajectory and the desired blending gains (desired outputs of the FNN) are computed based on the mode membership functions generated by the

MTM. That is, given that the state of the vehicle is $x(k)$ at some point over this hypothetical trajectory, and μ_i and μ_j are the mode membership functions for the modes involved in the transition from mode *i* to mode *j*, then the desired output for the FNN at that point is

$$\left[\frac{\mu_i(x(k))}{\mu_i(x(k)) + \mu_j(x(k))} \quad \frac{\mu_j(x(k))}{\mu_i(x(k)) + \mu_j(x(k))} \right]^T .$$

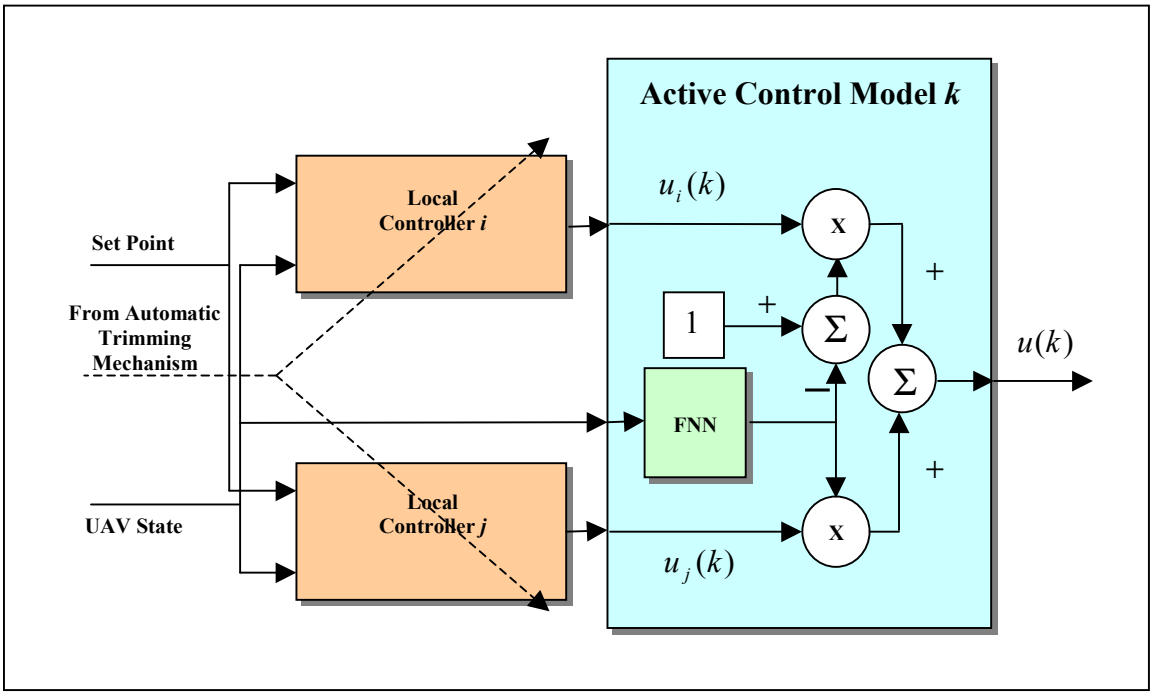


Figure 13. Active Control Models Structure

Thus, the computation of an optimal trajectory for the given transition is avoided at this stage. This new approach assumes that the mode transition controller itself does not determine the trajectory for the given transition since the trajectory generation

component specifies the trajectory at the middle level according to the tasks sent by the mission planning component.

At run time, FNNs of ACMs are adapted on-line by the control adaptation mechanism, as it is described in the sequel.

Once the local modes are defined and the local controllers are designed for each local mode, the transitions are established via the ACMs in the mode transition control component and the corresponding active plant models, which are incorporated into the adaptation mechanism.

5.3.1.6 Automatic Trimming Mechanism

The trim values of local controllers are initialized based on an approximate model of the vehicle. When the mode transition control is applied to the actual vehicle, these imprecise trim values affect the controller tracking performance reflected in a position and heading offset with respect to the desired values. This is the main reason why integral control was introduced in the local controllers. However, given that the vehicle can operate through different modes during a mission, and given that the trim values are different for different modes due to the nonlinearity of the model, integral control does not work as well as expected to compensate for the offset in the trim values unless the vehicle is steady flight, i.e. flying at constant velocity. The automatic trimming mechanism was established to allow the controller to memorize the trim values for all the operating conditions in steady flight. For this purpose a FNN was added to the mode transition controller and called auto-trim FNN. This FNN learns the trim values from the integral control when the vehicle is in steady flight as will be explained in the sequel.

The auto-trim FNN has the same inputs used to distinguish the modes of operation in the mode transition manager, i.e. forward and sideward velocities for the UAV considered in this work. Training of the auto-trim FNN proceeds on-line as follows.

- Every sample time the values of desired and actual velocities and acceleration obtained from the set point filter and the state filter are monitored. The condition for auto-trim will be set as true whenever the values of acceleration and angular velocities are under a prescribed threshold, and false otherwise. A counter indicating the number of samples that the auto-trim condition is valid will be incremented every sample that the auto-trim condition is true and reset whenever that condition is false.
- When the counter reaches a prescribed number of samples meaning that the vehicle is in steady flight, the value of the integral control at that time will be the correction required to the trim value. The desired trim value is computed by adding that correction (the integral control value) to the actual trim value estimated from the local controller or the active control model when the vehicle is in a transition region. Given that this desired trim value has a high confidence, the auto-trim FNN is adjusted using the method described in section 3.3 to produce that value when presented with the same operating condition again. At the same time, the integral control is reset to zero so the control output will not be affected by the automatic trimming mechanism. Trim values obtained from the auto-trim FNN will be used to correct the trim values of the local controller in a local mode, or the trim values for the local controllers involved in a transition.

- Whenever the auto-trim FNN presents a valid output, meaning that the FNN was already trained for the given input, the trim value generated by the auto-trim FNN will be used by local controllers instead of the default trim value defined when the local controllers were designed. Therefore, initially the controller will use the default trim values, but later on, as the automatic trimming progresses in actual flight, the trim values will be replaced by the ones learned from previous experiences in flight.

5.3.1.7 Dynamic Compensation Filter

For a rotary a helicopter UAV, like the one used for this work, there is some dynamics associated with the rotor called the flapping dynamics. The models used to design the local controllers discarded that part of the dynamics given that in the actual vehicle the states associated to this behavior are not measured nor estimated. However, the interaction of the rotor dynamics with the fuselage generate a couple of lightly damped modes that are not captured by the model and need to be compensated. Following the ideas from [18], dynamic compensation of the mentioned modes was implemented using notch filters in the cyclic control inputs of the rotor.

For both the longitudinal cyclic (δ_{mp}) and the lateral cyclic (δ_{mr}) controls a notch filter was applied at the mode transition control output, with a transfer function of the form

$$H_N = \frac{s^2 + \alpha BS + w_0^2}{s^2 + BS + w_0^2}, \tag{44}$$

where w_0 is the frequency of the mode to be eliminated, B is the bandwidth of the notch filter, and α is the gain of the filter at w_0, i.e. $0 \le \alpha < 1$ for the notch filter. The values of these parameters were adjusted based on analysis of flight data generated when the filters were not present. The dynamic compensation filters were implemented in discrete time using the bilinear transformation, equivalent to replacing the integrators in (44) with trapezoidal integrators given by

$$H_I(z) = \frac{T_N}{2}\frac{1+z^{-1}}{1-z^{-1}}, \tag{45}$$

with

$$T_N = \frac{\tan\left(\frac{Tw_0}{2}\right)}{\frac{w_0}{2}}, \tag{46}$$

where T is the sample period.

5.3.2 Adaptation Mechanism Component

The adaptation mechanism component calls the adaptation routines of the mode transition control and also includes a set of active plant models (one for each transition), which serve as partial models of the plant in the transitions. This concept is explained in next section.

5.3.2.1 Active Plant Models

For each transition there is an ACM in the MTC component and an associated active plant model (APM) in the adaptation mechanism component. The purpose of the

APMs is to serve as partial models of the plant in the transitions and provide the sensitivity matrices required for adapting the ACMs. The APMs used here follow the active modeling framework presented in section 4.2. That is, each ACM uses a FNN to represent the unknown nonlinear dynamics corresponding to the aerodynamics and propulsive forces acting on the vehicle, but also some known nonlinearities were incorporated into the models to speed up the learning of the FNNs. The new representation of the APMs given in Chapter 4 constitutes an improvement with respect to what was presented in [67].

Following the method presented in section 4.3, a linearized model of the vehicle is obtained near the actual operating point, defined by the pair $(x(k),u(k)) = (x_*,u_*)$, from the APM given by equation (29), repeated here for convenience

$$x(k+1) = \Phi(x(k) - x_*) + \Gamma(u(k) - u_*) + f_*, \tag{47}$$

where Φ, Γ, and f_* are defined in equation (30). This incremental model is used by the control adaptation mechanism to adapt the ACMs as will be discussed below.

5.3.2.2 Plant Adaptation Mechanism

The plant adaptation mechanism (PAM) is used to train the APMs. When the vehicle is in a transition, the input/output information from its sensors is used by the plant adaptation mechanism to train this model by calling the recursive least squares training routine from the FNN. To do that, the state values are transformed to body frame and the gravity effect is subtracted so the FNN accommodates to the model given by equation (20). The plant adaptation mechanism can be disabled at any time to free system

resources, if required. In that case, the last value of the APM is used by the control adaptation mechanism to compute the sensitivity matrices.

5.3.2.3 Control Adaptation Mechanism

The control adaptation mechanism (CAM) provides the adaptation functionality to the ACMs. When an ACM is active and the control adaptation mechanism is enabled, a dynamic optimization algorithm is used to find the optimal control value at each time step; the optimal blending gains that minimize the error between the optimal control and the control produced by the ACM are also computed. These optimal blending gains constitute the desired outputs for the recursive least squares training algorithm in the FNN corresponding to that ACM, which is in turn called by the control adaptation mechanism.

The dynamic optimization algorithm used to compute the optimal control value uses a finite horizon optimal control methodology (like a receding horizon control); the latter is based on the linearized model of the vehicle, which is obtained in turn from the sensitivity matrices generated from the corresponding APM, as given by (30). The objective of this optimal control problem is to minimize the following performance index

$$J = \frac{1}{2}\sum_{i=k}^{k+N} e^T(i)Qe(i) + \Delta u^T(i)R\Delta u(i), \tag{48}$$

with $Q \geq 0, R > 0$, subject to

$$\Delta x(i+1) = \Phi\Delta x(i) + \Gamma\Delta u(i) + \Delta f_* \quad \text{for } i = k, k+1, ..., k+N, \tag{49}$$

with $\Delta x(k) = x(k) - x_* = \Delta x_k$, where

$$
\begin{aligned}
e(i) &= \Delta x_d(i) - \Delta x(i), \\
\Delta x_d(i) &= x_d(i) - x_*, \\
\Delta x(i) &= x(i) - x_*, \\
\Delta u(i) &= u^*(i) - u_*, \text{ and} \\
\Delta f_* &= f(x_*, u_*) - x_*.
\end{aligned}
$$

This is a discrete linear quadratic soft terminal controller problem [68]. Equation (48) can be rewritten as

$$
J = \frac{1}{2}\sum_{i=k}^{k+N} (\Delta x_d(i) - \Delta x(i))^T Q(\Delta x_d(i) - \Delta x(i)) + \Delta u^T(i) R \Delta u(i). \tag{50}
$$

The discrete Hamiltonian for this problem is

$$
\begin{aligned}
H(i) = \frac{1}{2}\Big(&\Delta x^T(i) Q \Delta x(i) + \Delta u^T(i) R \Delta u(i) - 2\Delta x_d^T(i) Q \Delta x(i) + \Delta x_d^T(i) Q \Delta x_d(i)\Big) \\
&+ \lambda^T(i+1)\big(\Phi \Delta x(i) + \Gamma \Delta u(i) + \Delta f_*\big)
\end{aligned}. \tag{51}
$$

Necessary conditions for a stationary solution are the Euler-Lagrange equations

$$
\begin{aligned}
&\Delta x(i+1) = \Phi \Delta x(i) + \Gamma \Delta u(i) + \Delta f_*, \text{ with } \Delta x(k) = \Delta x_k, \\
&\lambda(i) = \left(\frac{\partial H(i)}{\partial \Delta x}\right)^T = Q\Delta x(i) + \Phi^T \lambda(i+i) - Q\Delta x_d(i), \text{ with } \lambda(k+N) = 0,
\end{aligned}
$$

where $$\frac{\partial H(i)}{\partial \Delta u} = \Delta u^T(i) R + \lambda^T(i+1)\Gamma = 0.$$

Assuming a sweep solution for $\lambda(i)$ of the form

$$
\lambda(i) = S(i)\Delta x(i) + g(i),
$$

and after some algebra the following equations are obtained

$$
\begin{aligned}
S(i) &= Q + \Phi^T S(i+1)\Big[\Phi - \Gamma\big(R + \Gamma^T S(i+1)\Gamma\big)^{-1}\Gamma^T S(i+1)\Phi\Big], \\
g(i) &= \Big[\Phi - \Gamma\big(R + \Gamma^T S(i+1)\Gamma\big)^{-1}\Gamma^T S(i+1)\Phi\Big]^T (g(i+1) + \Delta f_*) - Q\Delta x_d(i),
\end{aligned} \tag{52}
$$

for $i = k+N-1, k+N-2, \ldots, k$, with $S(k+N) = 0$ and $g(k+N) = 0$.

The optimal control is given by

$$\Delta u(i) = u_f(i) - K(i)\Delta x(i)\,, \tag{53}$$

where

$$u_f(i) = -\left(R + \Gamma^T S(i+1)\Gamma\right)^{-1}\Gamma^T\left(g(i+1) + \Delta f_*\right),\ \text{and}$$
$$K(i) = \left(R + \Gamma^T S(i+1)\Gamma\right)^{-1}\Gamma^T S(i+1)\Phi\,.$$

Application of the dynamic optimization algorithm gives the value of $\Delta u(k)$ which, in turn, is needed to compute $u^*(k)$ from $u^*(k) = u_* + \Delta u(k)$. This is the optimal control value used to compute the desired blending gains for the active control model.

The approach constrains the blending gains so the ACM produces a convex combination of the outputs of the local controllers and guarantees smooth transitions. That is, given the outputs of the local controllers corresponding to the ACM, $u_i(k)$ and $u_j(k)$, the objective is to minimize the magnitude of the error

$$\left\| u^*(k) - desiredGains_1 u_i(k) - desiredGains_2 u_j(k) \right\|_2^2,$$

subject to

$$0 \le desiredGains_i \le 1 \ \text{ for } i = 1,2\,, \text{ and}$$

$$desiredGains_1 + desiredGains_2 = 1\ .$$

A simple algorithm achieves this objective:

$$\alpha = sat\left(\frac{\Delta u^T(u^*(k) - u_i(k))}{\Delta u^T \Delta u}\right), \tag{54}$$

$$desiredGains_1 = 1 - \alpha\,,$$
$$desiredGains_2 = \alpha\,,$$

where $\Delta u = u_j(k) - u_i(k)$, and

$$sat(x) = \begin{cases} 0, & x < 0 \\ x, & 0 \le x \le 1 \\ 1, & x > 1 \end{cases}.$$

These desired gains become the desired outputs for the recursive least squares algorithm that trains the FNN of the ACM. Given that these blending gains are complementary, FNNs representing the ACMs were reduced to have only one output ($blendingGains_2$), the other one was computed from

$$blendingGains_1 = 1 - blendingGains_2 .$$

A drawback of the algorithm presented above is the requirement of a lot of computational power. Based on simulations performed in Simulink on a linearized model of the vehicle, it was found that the duration of the horizon (N) had to be considerable high to guarantee a stable behavior. The value originally chosen for the horizon was N=25, but that value produced instability and had to be increased to 75. Measurements showed that the computation time for this algorithm was around 50msec per sample in a processor with more computational power that the actual onboard computer, which is unacceptable given that the controller was working at a sample rate of 25Hz. This result is not strange given the required value of N and considering that the simplified dynamics of the UAV augmented with integrators had order 18. For this reason, a simplified algorithm was developed that computed the optimal blending gains directly based on the plant models. The algorithm used the same equation (54) but $u_1(k)$, $u_2(k)$, and $u^*(k)$ were replaced by vectors resulting from stacking together all the elements of matrices Φ and Γ associated to linearized models of the vehicle for the modes involved in the transition and for the actual operating condition respectively. This matrices are computed from the active plant models.

CHAPTER 6

SOFTWARE IMPLEMENTATION OF THE ADAPTIVE MODE TRANSITION CONTROL

The adaptive mode transition control architecture has been implemented using an object oriented approach in C++. This chapter describes how the code was organized; describes the components of the adaptive mode transition control library; describes the utilities developed to setup, update, test, and examine the internal structure of the adaptive mode transition control; and finally, describes the S-functions developed to allow testing the code in Simulink.

6.1 Source Code Organization

The source code for the adaptive mode transition control has been organized in a directory structure to separate the files of the adaptive mode transition control library and the various utility programs for manipulation of an adaptive mode transition control. The same directory structure includes workspace and projects files required for compilation of the code in Windows as well as the make files developed for compilation of the code for Linux and QNX. The code has been compiled for Windows using Visual C++ 6.0, for Linux using GNU gcc compiler, and for QNX using GNU gcc cross compiler v.2.96 on a Linux platform.

The root directory for the source code is called AMTCwork. This directory contains the main workspace file (AMTCwork.dsw), the main make file (GNUmakefile), a script to build the code for Windows when working on a Windows platform

(buildamtc.bat), and a script to build the code for Linux and QNX when working on a Linux platform (buildamtc).

The actual source code is organized in subdirectories as follows:

- *amtc* contains all the code included in the adaptive mode transition control library, the core of the adaptive mode transition control.
- *AMTCall* contains a project file to build all the libraries and utilities (only for Windows).
- *AMTClibraries* contains a project to build all the libraries (only for Windows).
- *AMTCob2* contains the source code, a project file and a make file to build a stand alone application that runs the adaptive mode transition control. In Windows and Linux it enables software in the loop simulation, in QNX it enables hardware in the loop simulation and actual flight testing.
- *AMTCsetup* contains the source code, a project file and a make file to build a utility to setup an adaptive mode transition control.
- *AMTCsim* contains the source code, a project file and a make file to build a library that allows stand alone simulation of an adaptive mode transition control.
- *AMTCtest* contains the source code, a project file and a make file to build a utility that allows stand alone simulation of an adaptive mode transition control.
- *AMTCtestLC* contains the source code, a project file and a make file to build a utility that allows stand alone simulation of an adaptive mode transition control configured with a single local controller.
- *AMTCupdate* contains the source code, a project file and a make file to build a utility to update an adaptive mode transition control initialization file.

- *bin* contains the executables generated when the code is built.
- *genMission* contains the source code, a project file and a make file to build a utility to generate a mission file.
- *lib* contains the libraries generated when building the code.
- *MissionTest* contains the source code, a project file and a make file to build a utility to test a mission file.
- *printAMTC* contains the source code, a project file and a make file to build a utility to print the contents of an adaptive mode transition control initialization file to the standard output.
- *printMission* contains the source code, a project file and a make file to build a utility to print the contents of a mission file to the standard output.
- *RmaxModel* contains the source code, a project file and a make file to build a library including the model of a Yamaha Rmax helicopter. It uses the same source code defining de UAV model from the GTmax simulation environment.
- *Sfunctions* contains the source code and dynamic libraries associated with the S-functions wrapping the adaptive mode transition control and the UAV model for testing in Simulink.
- *SimData2m* contains the source code, a project file and a make file to build a utility to convert a binary file, containing recorded data from a simulation or flight test, to a text file readable from Matlab.

6.2 Adaptive Mode Transition Control Library

The core of the adaptive mode transition control is incorporated in the mode transition control library. This library defines the classes for the components of the architecture and also the basic classes used throughout the code. Implemented classes can be divided into three categories: low level or basic classes, intermediate level classes, and high level classes. Higher level classes use the lower levels in the implementation. The objects defined by all the classes are so distinctive and with functionality so diverse, that no particular class hierarchy applies especially in the higher levels of the implementation. Therefore the main method used for reusability of the code throughout the implementation is class composition. For most of the classes the following elements were implemented:

- a default constructor with the default parameters for the object at hand,
- a constructor based on a binary file that reads the main parameters from the file and initializes the object accordingly,
- a load member function that is able to reinitialize an existing object based on the data from a binary file,
- a save member function to save the current object to a binary file so it can be retrieved later using a constructor based on a binary file or the load member function, and
- an overload of the output stream operator able to print a text representation of the object including all the important information in a readable manner.

Member functions enabling retrieval and saving of the objects were not implemented in cases where the objects did not need to be persistent, like in the case of

the trajectory generation component. Below, a description of the classes for each category is given.

6.2.1 Low Level or Basic Classes

Low level classes are the ones defining the basic functionality required for the implementation of all the algorithms including vectors, matrices, vectors of vectors, and vectors of matrices.

Originally, the intention was to define the low level classes using template classes, but certain difficulties presented in the compilation of the code for QNX made me desist. So, the current implementation uses code that defines the specific instances of those original template classes with different names. The idea behind the implementation of these classes was to allow the coding of the control algorithms with the same ease as using Matlab, but without using Matlab libraries. The implementation of these classes was carefully crafted looking for ease of use, memory efficiency, and optimal speed.

Some of the functionalities implemented are:

- Copy constructors and assignment operators (=, +=, -=, *=, /=).
- Comparison operators (== , !=).
- Inner and outer product of vectors.
- 2-norm, infinity norm and n-norm of vectors.
- Matrix by vector and vector by matrix multiplications.
- Row wise and column wise Gaussian elimination algorithms for efficient implementation of pre and post multiplication by an inverse matrix and efficient matrix inversion.

- Subscript operators for vectors and matrices.
- Extraction of sub-vectors and sub-matrices.
- Assignment of a vector to a sub-vector and a matrix to a sub-matrix.
- Operator | for concatenation of vectors.
- Operator | for stacking of matrices side by side, and "," for stacking matrices up and down.
- Conversion functions and cast operators to convert from vectors to matrices and vice versa.
- Reshaping of matrices.
- Some basic functions of vectors like exponential, sine, cosine, etc.
- Some basic functions of matrices like exponential, inverse, determinant, rank, transposition, etc.

The basic classes included in the adaptive mode transition control library are: VectorOfDouble, VectorOfFloat, VectorOfInt, MatrixOfDouble, MatrixOfInt, VectorOfVectorsOfDouble, VectorOfMatricesOfDouble.

6.2.2 Intermediate Level Classes

This category includes the classes defining generic components used in the adaptive mode transition control architecture that are not specific to the architecture. The classes included in the adaptive mode transition control library corresponding to this category are: three classes of spline interpolators, three filter classes, and the FuzzyNN class. The classes mentioned here make extensive use of the low level classes mentioned before.

6.2.2.1 Spline Interpolator Classes

There are three classes of spline interpolators used in this implementation: CubicSplineInterpolator, FifthOrderSplineInterpolator, SeventhOrderSplineInterpolator. These classes define n-dimensional spline interpolators, as their name indicates they differ in the order of the splines used for the interpolation. These classes are used by objects of classes MissionPlanning and TrajetoryGeneration to generate the trajectories corresponding to a given mission as explained in sections 5.1 and 5.2. One of the data members of these classes is an object of class MatrixOfDouble used to store the coefficients of the polynomials representing the splines. The method ComputeCoefficients is used to calculate the coefficients and store them in the coefficients matrix. Once the coefficients have been computed the methods getValueAt, getDerivativeAt , and getSecondDerivativeAt can be used to evaluate the generated splines and their first and second derivatives at any value of the independent variable.

6.2.2.2 FuzzyNN Class

This class implements the fuzzy neural networks discussed in Chapter 3. The class Rule, defined in the same header file that FuzzyNN, represents the objects for each of the rules of the fuzzy neural network. The rules are organized in a linked list so the limit to the size of the fuzzy neural networks is determined by the memory available. However, for computational efficiency the fuzzy neural networks include a variable maxNumberOfRules to constrain their size, so no more than that number of rules are created. Each rule includes a VectorOfInt object called antecedentVector that indicates the combination of the input membership functions in each input coordinate associated

with the premise part of the rule. There are also MatrixOfDouble objects called initMatrix and matrix that represent the consequent matrices before and after the recursive least squares correction. The matrices used by the recursive least squares algorithm called XtranspX and YtranspX are also stored in each rule. The values of the mean and inverse deviation parameters for the input membership functions for each if the input coordinates are organized in objects of class VectorsOfVectorsOfDouble.

The class FuzzyNN has methods implementing the learning algorithms discussed in Chapter 3 and also to calculate the output and the Jacobian matrix at any value of the input vector.

6.2.2.3 Filter Classes

There are three classes that implement filters in the adaptive mode transition control library: SPfilter, StateFilter and MeanDevFilter. SPfilter implements the set point filter discussed in section 5.3.1.1, StateFilter implements the state filter discussed in section 5.3.1.2, and MeanDevFilter implements a special filter for computing the mean and standard deviation value of an input vector over time. An object of the last class is used by the MTC class to determine when the data generated by the active plant models is good enough.

6.2.3 High Level Classes

The classes in this category define the objects representing the main components and subcomponents of the adaptive mode transition control architecture described in Chapter 5, and also an object that encapsulates the whole structure of an adaptive mode

transition control. These classes make extensive use of the basic classes and the intermediate level classes for the implementationof the components. The high level classes in the adaptive mode transition control library are: MissionPlanning, TrajectoryGeneration, LocalController, ActiveControlModel, MTM, MTC, ActivePlantModel, AM, MonitorModule, and AMTC. A brief description of each of these classes is presented in the sequel.

6.2.3.1 MissionPlanning Class

This class defines the mission planning component functionality as discussed in section 5.1. A structure called Task is defined to capture all the information required for each task of the mission. The tasks are generated through the use of several high level methods (like hover, flyTo, holdOn, addMission), which in turn call a low level method (addTask) to add the tasks to the mission and organize them in a queue (the task queue). The method setMissionConstraints is used to set the maximum speed, maximum acceleration, maximum jerk, maximum angular speed and maximum angular acceleration constraints. setInitialState method is used to set the initial state of the vehicle for the mission at hand. When the mission is executed, method getNexTask is called to retrieve each task in an orderly manner and method setTaskCompletedFlag is used to pass the information received from the trajectory generation component to the mission planning object.

6.2.3.2 TrajectoryGeneration Class

This class defines the trajectory generation component described in section 5.2. Several methods are used for the generation of the trajectories as described in the sequel.

The tasks received from the mission planning component are set with the addTask method, which calculates the spline used for representation of the trajectory for the current task. getSetPoint method is used to calculate and retrieve the set points at each sample time. The method getTaskCompletedFlag is called to retrieve the information containing the termination status of the current task so that information can be sent to the mission planning component. There is also a reset method used to clear the tasks available in the trajectory generation object.

6.2.3.3 LocalController Class

This class implements the local controllers used in the mode transition controller as described in section 5.3.1.3. Therefore, there is an instance of an object of this class for each of the local modes in the mode transition control component. The most important method of this class is the ComputeControl method that is used to compute the value of the local control signal corresponding to given state and set point. There are some methods to set the parameters of each local controller, the most important being setParameters that allows setting the gain matrix and trimming value associated with the local controller.

6.2.3.4 ActiveControlModel Class

This class implements the functionality of the active control models presented in section 5.3.1.5. One of the data members of this class is a FuzzyNN object that is used to compute the blending gains by getBlendingGains method. The main method of this class is ComputeBlendedControl which is called by the mode transition control component to calculate the control signal when the vehicle is in a transition. Pointers to the local

controllers associated with the active control model are used to access them and blend their outputs.

6.2.3.5 MTM Class

This class implements the functionality of the mode transition manager discussed in section 5.3.1.4. A MatrixOfInt object represented by data member modeTransitionsTable is used to hold a table of the transitions for the current setup of the mode transition control. In that table, rows and columns are the identification numbers of the modes involved in the transitions and the entries are the identification numbers of the active control model and active plant model corresponding to the transitions. A zero entry in that table means that there is no active control model or active plant model associated with the transition, so the mode transition controller will switch abruptly from one mode to the other in that case. The parameters of the mode membership functions are stored in two data member objects of class VectorOfVectorsOfDouble: modeCenters and modeInvDeviations. When setting up the mode transition controller, method addMode is used to add a mode specifying its associated parameters, and method setTransition is used to set the appropriate entry in modeTransitionsTable. When the mode transition control is executed the getMode method is used to determine the current mode of operation indicating if the plant is in a local mode or a transition. This information is used by the mode transition control component to determine whether it should execute a local controller or an active control model.

6.2.3.6 MTC Class

This class implements the functionality of the mode transition control component discussed in section 5.3.1. The class includes pointers to a linked list of objects of LocalController class and a linked list of objects of ActiveControlModel class. Other important data members of this class are: an object of class MTM called ModeTransitionManager, an object class SPfilter called setPointFilter, and an object class StateFilter called UAVstateFilter. The main methods for this class can be categorized as methods for setup of the MTC object, methods for preparation of the MTC object before execution, methods for execution of the mode transition controller, and auxiliary methods used in the execution of the mode transition controller.

The most important methods for setup of the MTC object are setAM_Ptr, addMode, and addTransition. setAM_Ptr method is used to set a pointer to the adaptation mechanism object to allow some required interactions with that component. addMode method is used to add a new mode to the mode transition controller structure. When called, it adds a new object of class LocalController to the linked list of local controllers, adds a new active plant model object in the adaptive mode transition control component associated with the local mode, and calls the addMode method of ModeTransitionManager object. addTransition method is used to add a new transition to the mode transition controller structure. When called, it adds a new object of class ActiveControlModel to the linked list of active control models, it links that active control model to the local controllers associated to that transition, it adds a new active plant model object in the adaptive mode transition control component associated to the transition at hand, and calls setTransition method of ModeTransitionManager object.

The most important methods for preparation of the MTC object before execution are reset, setInitialState, and setInitialOutput. Reset method is used to reset all the variables in the MTC object in preparation for execution. setInitialState and setInitialOutput methods are called immediately before starting the mode transition controller. setInitialState method initializes some internal variables and the state filter based on the initial value of the state. setInitialOutput is used to set the initial output value, to initialize the integral control value, and to set some variables required specifically by dynamic compensation filter discussed in section 5.3.1.7. Notice that the functionality of the dynamic compensation filter is embedded in the MTC object and is not represented by a different class.

The most important auxiliary methods used in the execution of the mode transition controller are: autoTrim, correctOutput and filterOutput. These methods will be mentioned below.

The most important methods for execution of the mode transition controller are setUAVstate, setSetPoint, and computeControl. Each sample time the mode transition controller is executed in the following manner: first, setUAVstate method is called to set the value of the state, run the state filter, and call getMode method of mode transition manager object to determine the current mode; second, setSetPoint is called to set the set point value received from the trajectory generation component and run the set point filter; and finally, computeControl method is called to calculate the control signal value. computeControl method performs the following operations: first, it calls autoTrim method to execute the automatic trim mechanism discussed in section 5.3.1.6; second, it executes a local control or an active control model, which in turn executes the local

controllers associated with a transition, depending on the mode determined when setUAVstate was called; third, it calls correctOutput method to adjust the integral control to avoid discontinuities when the controller is started or when the mode changes; and finally, it calls filterOutput method to execute the dynamic compensation filter discussed in section 5.3.1.7 and also impose limits to the control signal.

6.2.3.7 ActivePlantModel Class

This class implements the active plant model components described in section 5.3.2.1. This class has an object data member of class FuzzyNN called FNN representing the model of the plant in the corresponding transition region. The class declares MTC and AM as friend classes so they can access directly the FNN object.

6.2.3.8 AM Class

This class implements the adaptation mechanism component discussed in section 5.3.2. The class includes a pointer to a linked list of objects of class ActivePlantModel representing the active plant models used in the transitions as described before. In the current implementation there is also a pointer to a linked list of objects of class ActivePlantModel representing the models of the plant in the local modes. The most important methods of this class can be categorized as methods for setup of the AM object, methods for execution of the adaptation mechanism, and auxiliary methods used in the execution of the adaptation mechanism.

The most important methods for setup of the AM object are setMTC_Ptr, addActivePlantModel, addLocalPlantModel, and setAdaptationMode. setMTC_Ptr method is used to set a pointer to the mode transition control component, which is

required to get the mode information from it. addActivePlantModel method is used to add an active plant model to the linked list of active plant models associated to the transitions, it is called by addTransition method of the adaptive mode transition control component. addLocalPlantMode method is used to add an active plant model to the linked list of local plant models; it is called by addMode method of the adaptive mode transition control component. setAdaptationMode method allows activating or deactivating the plant adaptation and the control adaptation mechanisms implemented by this class by mean of setting or resetting the flags called plantAdaptationEnabled and controlAdaptationEnabled.

The most important auxiliary methods used in the execution of the adaptation mechanism are adaptPlantModel, computeDeltaU, and adaptControlModel. adaptPlantModel method performs the plant adaptation mechanism described in section 5.3.2.2 adapting the current active plant model based on known values of the state and control signal applied to the plant. computeDeltaU method is used to calculate the value of the optimal correction in the control signal, $\Delta u(k)$, based on an incremental model obtained from current active plant model using the algorithm explained in section 5.3.2.3. adaptControlModel performs the control adaptation mechanism using the value $\Delta u(k)$, obtained by computeDeltaU, to adapt the corresponding active control model as explained in section 5.3.2.3.

The most important methods for execution of the adaptation mechanism are runAdaptation and runPlantAdaptation. runAdaptation method calls adaptPlantModel and/or adaptControlModel only when they are enabled according to the current value of the flags plantAdaptationEnabled and controlAdaptationEnabled. runPlantAdaptation

calls adaptPlantModel only when it is enabled according to the current value of the flag plantAdaptationEnabled. In current implementation of AM class some logic was added to inhibit the control adaptation when the values of the state estimated from the active plant models are no good enough.

6.2.3.9 MonitorModule Class

This class defines an object used to record the most important signals in the execution of the adaptive mode transition control in a binary file for future analysis, for instance to plot the results of a simulation or a flight test. The file generated by this module can also be used for off-line training of the active plant models and active control models of the adaptive mode transition control. A brief description of the methods implemented in this class is as follows:

setOutputFileName method is used to set the name of the binary file being generated. When this method is called, the current file, if any is closed and a new file with the given name is opened for recording the signals from that moment on.

- closeFile method is used to explicitly close the current file.
- setMode, setSetPoint, setIntegralControl, setControlOffset, setControlInput, setActualControlInput, setUAVstate, setExpectedUAVstate, setAcceleration, setFilteredVelocity, setAdaptationMode, setComputationTime methods are used to set the value of the signals to be recorded at a sample time.
- recordSignals method is used to save the actual values of the signals to the file.

6.2.3.10 AMTC Class

This class implements the whole adaptive mode transition architecture. This means that the class includes object data members representing all components of the architecture: an object of MissionPlanning class, an object of TrajectoryGeneration class, an object of MTC class, an object of AM class, and an object of MonitorModule class. The main purposes of this class are:

- provide a way to pack the structure of the whole architecture in a way that allows saving and retrieving the information of the whole adaptive mode transition control from a single file, and
- simplify the interface required to create stand alone simulations, S-functions for Simulink, and software in the loop simulations.

The main methods of this class are setOutputFileName, closeOutputFile, setAdaptationMode, setSimkDelay, generateSetPoint, and generateControl. setOutputFileName method sets the name of the file used by the MonitorModule object to save recorded data. closeOutputFile method closes the file used to save recorded data. setAdaptationMode method allows activating or deactivating plant adaptation and control adaptation mechanisms. setSimkDelay allows setting a simulated delay that is implemented in the execution of the adaptive mode transition controller. generateSetPoint method executes just the mission planning and trajectory generation components to produce the set point for current time. generateControl method executes the whole architecture for current time.

6.2.4 Other Functionalities Included in the Adaptive Mode Transition Control Library

Besides the classes explained before, there is a class called UAVlinModel that is used to represent a linearized model of the UAV under control at certain operating condition. This class is employed to pack the information about the linearized models that are used to in the setup of the adaptive mode transition control when calling addMode method of the MTC class. The main data members representing the linearized model are two objects of class MatrixOfDouble, A and B, representing the matrices of the model and two objects of class VectorOfDouble, uTrim and UAVstateTrim, representing the operating conditions of the linearized model. The class also provides the functionality required to perform a simulation of the model through methods setInitialState, setInput, setWind, and getOutput.

Some utility functions required implementing some transformations required throughout all the code are also included in the adaptive mode transition control library.

A set of functions wrapping the adaptive mode transition control architecture were developed to facilitate the integration of the code with the GTmax simulation environment for software in the loop simulation. GTmax software is a flexible environment developed at Georgia Tech that can be used for simulation or actual flight of a helicopter UAV based on the Yamaha Rmax helicopter. The mentioned functions are:

- initAMTC, used to create a new object of class AMTC and initialize the architecture based on information read from an initialization file.
- updateAMTC, used to execute all the components of the architecture at current sample time.

- saveAMTC, used to save the structure of the architecture calling save method of AMTC class.
- shutdownAMTC, used to delete the object of AMTC class and clean memory.

6.3 Utilities for Manipulation of an Adaptive Mode Transition Control

The adaptive mode transition control library described in section 6.2 contains all the code needed to carry out any implementation using the adaptive mode transition control architecture. However, there are some tools required to be able to setup a new adaptive mode transition control structure and manipulate an existing one. This section describes the tools that were implemented for that purpose.

6.3.1 Setup, Update, and Visualization of an Adaptive Mode Transition Control Initialization File

Typically, the components of the adaptive mode transition control architecture are initialized based on a binary file called AMTC.dat. This file is generated from an existing object of class AMTC using the save method. Therefore, an application is required to setup the initial structure and parameters of the adaptive mode transition controller required for the application at hand, and save the AMTC.dat file. The application created for this purpose is called AMTCsetup. The source code for that application includes just the file AMTCsetup.cpp which is compiled and linked to the adaptive mode transition control library to generate the application. AMTCsetup.cpp defines the main() function for the application, and follows the following procedure to generate the file AMTC.dat:

- First, default constructors of MTC and AM classes are used to generate default objects.
- The objects are linked together using the methods setAM_Ptr for the object of class MTC and setMTC_Ptr for the object of class AM. These objects are customized calling the methods that set all the important parameters.
- addMode method of the MTC component is called as many times as required to add and setup the local modes.
- addTransition method of the MTC class is called as many times as required to add and setup the required transitions.
- At this point the structure of the adaptive mode transition controller is setup, so the save methods of the objects are called to save them to the file AMTC.dat in such a way that the same file can be used to initialize an object of class AMTC.

Besides of the AMTCsetup utility, another utility called AMTCupdate was created to update the parameters of an existing AMTC.dat file. This application was created in the same manner that AMTCsetup, i.e. a file AMTCupdate.cpp containing the main() function for the application that is compiled and linked to the adaptive mode transition control library. The procedure followed in AMTCupdate is as follows: an object of class AMTC is created using the constructor based on a binary file using the existing file AMTC.dat, then the parameters that need to be modified are setup using the appropriate methods, and finally the save method of the object of class AMTC is called to save the new file AMTC.dat.

There is another simple application that was created to visualize the contents of the AMTC.dat file in a human readable way. This application was called printAMTC.

When this application is called, it reads the file AMTC.dat and then prints all the information in the standard output. The code for the applications is pretty simple, it just uses the constructor based on a binary file of the AMTC class to read the file AMTC.dat and then it prints the contents to standard output using the output stream operator. This application can also print the contents of an AMTC file with a different name than AMTC.dat; in that case the name of the file is given as a parameter. A text file can be generated redirecting the output of the application.

6.3.2 Generation of a Mission Initialization File

Applications were created to generate, test and visualize the content of a Mission. genMission is a program that is customized for generating the required mission in a file called Mission.dat. Than file can be loaded by the mission planning component and defines the mission to be performed. MissionTest is an application created to test any Mission.dat file. That application loads the mission and executes it as if it were executed by the adaptive mode transition control architecture. The resulting data are saved in a file that can be used in Matlab to plot the trajectory. Additionally, an application called printMission was created to print the contents of a mission given by a file Mission.dat or any mission file specifying the name of the file as a parameter. This is useful to check the content of the generated tasks.

6.3.3 Stand Alone Simulation of the Adaptive Mode Transition Control Architecture

An application called AMTCtest was created to run a stand alone simulation of the adaptive mode transition control. This application takes files AMTC.dat and Mission.dat as input and performs the simulation of the adaptive mode transition control architecture applied to a model of the Yamaha Rmax helicopter saving the results in a file called MonitoredData.dat.

Another application called AMTCtestLC was created to do a stand alone simulation like AMTCtest but this time setting up the adaptive mode transition control to use only one local controller based on the file LocalController.dat. This application was used to test local controllers.

6.3.4 Stand Alone Executable for Hardware in the Loop Simulation and Flight Testing

An application called AMTCob2 was created that allows running the adaptive mode transition controller in real time. This program initializes the adaptive mode transition control architecture based on the AMTC.dat file, and the mission to be performed based on the file Mission.dat. AMTCob2 communicates with the GTmax software to perform simulations and flight testing.

In Windows and Linux AMTCob2 enables in software in the loop simulation; in QNX it enables hardware in the loop simulation and also can be used in flight testing.

6.3.5 Conversion of Simulation and Flight Data File to a m-file for Matlab

When the adaptive mode transition control architecture is run in simulation or actual flight, a file is generated by the object of class MonitorModule containing a record of the important signals. This file, usually called MonitoredData.dat, can be converted to an m-file called MonitoredData.m using the SimData2m application. That file can be used in Matlab to plot the results of a simulation or flight test.

6.4 S-functions for Testing of the Adaptive Mode Transition Control in Simulink

For the purpose of testing the adaptive mode transition control architecture in Matlab using Simulink, several S-functions were developed that wrapped the code of the adaptive mode transition control architecture. These S-functions were implemented using the templates provided with Matlab, and linking the code to the adaptive mode transition control library. The S-functions were compiled using the mex utility of Matlab.

The following S-functions were implemented:

- TrajectoryGeneratorModel, an S-function wrapping mission planning and trajectory generation components to test the trajectory generation part of the architecture.
- TrajectoryGeneratorModelNsamp, an S-function identical to TrajectoryGeneratorModel except that it generated N samples into the future of the trajectory at a time.

- AMTCmodel, an S-function wrapping the whole adaptive mode transition control architecture.
- MTCmodel, an S-function wrapping the mode transition control component.
- RmaxModel, an S-function wrapping the model of the Yamaha Rmax helicopter
- RmaxModel_IS, an S-function identical to RmaxModel but allowing to specify the initial state as a parameter.
- RmaxModelTrim, an S-function wrapping the model of the Yamaha Rmax helicopter in a specific way required to compute the trim conditions associated to a given operating condition.

CHAPTER 7

IMPLEMENTATION ON THE OPEN CONTROL PLATFORM

The adaptive mode transition control architecture was implemented using the Open control Platform (OCP). The OCP is a software infrastructure developed by Boeing in collaboration with Georgia Tech and others to enable the implementation of advanced control algorithms for UAVs [59-63]. It allows for system reconfiguration, interoperability of different operating systems and platforms, plug and play connectivity, while it enables the implementation of sophisticated multi-rate hybrid systems. The OCP includes a controls API that allows the user to generate easily the code required for the application at hand, and customize it to include his own control algorithms. Georgia Tech has also developed a Hybrid Controls API that has been integrated into the OCP, which facilitates the implementation of certain common operations required for hybrid control systems [69-71].

7.1 Implementation on the OCP Using the Controls API

The adaptive mode transition control architecture has been implemented using the controls API of the OCP. Figure 14 shows the structure of the implementation on the OCP. For this implementation, there are five components running in one process on the OCP:

- the GTmax link component for the communication interface with the primary flight computer in the UAV,

- the high level component,
- the middle level component,
- the mode transition controller, and
- the adaptation mechanism component.

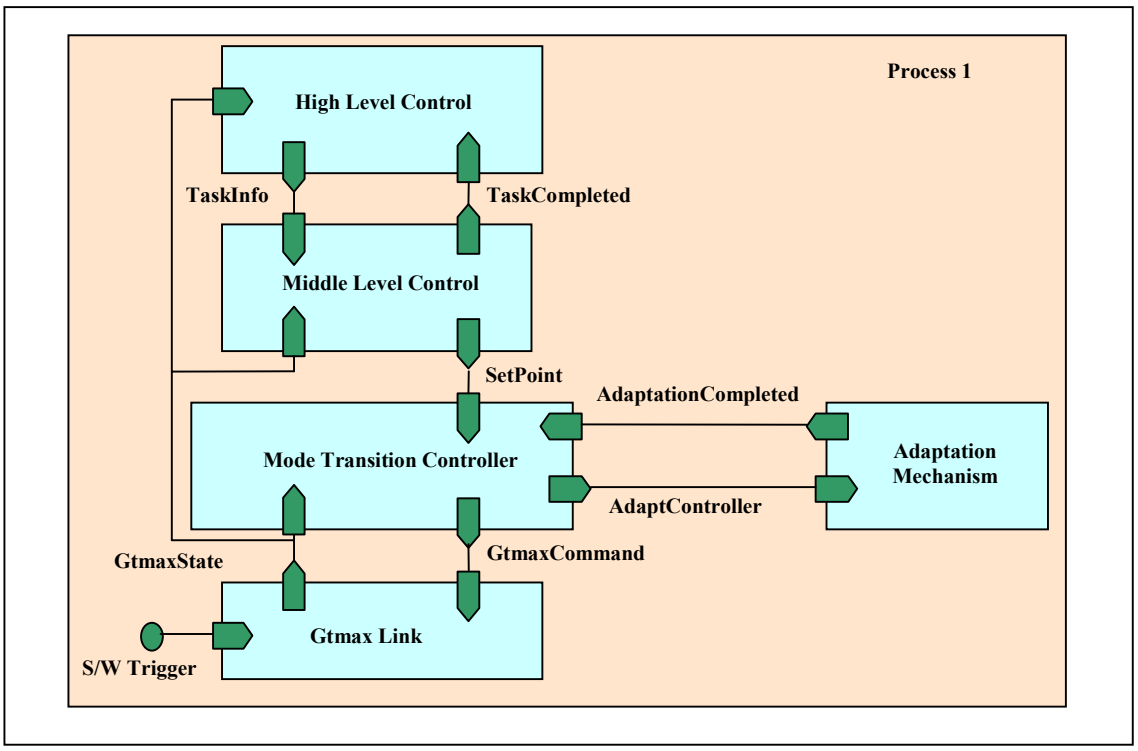


Figure 14. Structure of the Implementation on the OCP

The steps required for generating the implementation on the OCP are illustrated in Figure 15, and can be summarized as follows:

- A block diagram with the structure of the application at hand is drawn in Simulink.

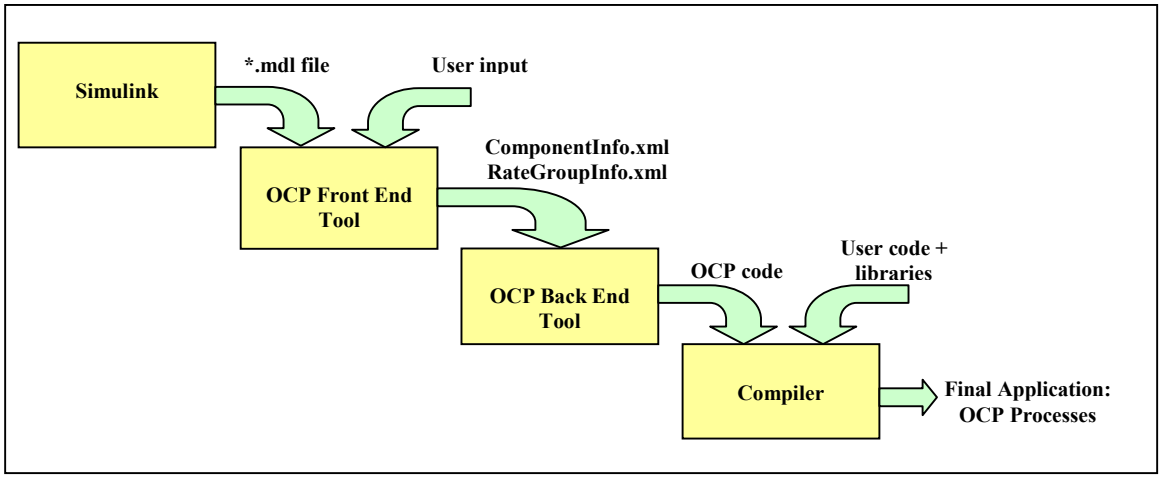


Figure 15. Steps for Implementation on the OCP

- The model file is used as input to the OCP front end tool that generates an intermediate representation of the implementation known as the component input file. In older versions of the OCP this file was a text file (ComponentInfo.txt), now considered a legacy component info file. The latest versions of the OCP use an XML format for the component info file (ComponentInfo.xml), which can be generated directly or from a legacy version using the OCP front end tool. Is also possible to write the component info file by hand following the format specified in the OCP documentation. The component info file is composed of the following information: definition of the structure of the signals used to communicate to different components; definition of the components including their input and output ports, the type of the signals associated to those ports, and the behaviors that the component implements, indicating the input ports that activate those behaviors and the output ports accessible from them; definition of the processes included in the implementation, that is, which are the executables required and which components are included in each one of them; definition of the

interconnection structure of the implementation; and finally, definition of QoS constraints. Another file generated by the OCP front end tool is a rate group info file indicating the rates that are going to be used in the implementation. The legacy text version of this file is RateGroupInfo.txt and the new XML file is RateGroupInfo.xml.

- The next step is the generation of the OCP code required for the implementation. This task is accomplished automatically using the OCP back end tool, which takes as input the component info and rate group info files and produces the OCP code as output. The project files required to compile the application are also generated automatically by the OCP back end tool. Make files required to compile the code for Linux or QNX can be generated from the project files using a make generation tool included with the OCP. Generated code employs the controls API to enable the use of the OCP infrastructure by the application at hand. However, the generated code is just a template that does not include the user code implementing the functionality of the application. This leads to the following step.
- The OCP code is populated with the user code implementing the functionality of the application at hand. For the case of the implementation of the adaptive mode transition control architecture, this step was enormously simplified given that all the components were already defined in the adaptive mode transition control library. The only code necessary on the OCP implementation was the one to create instances of the components, initialize those instances, specify in the behaviors how to execute the different components and connect their inputs and outputs to those of the OCP components.

- The final step is building the code. For the case of the adaptive mode transition control architecture, the core of the implementation was built separately in the adaptive mode transition control library and then the OCP code was compiled and linked to that library to generate the final application.

7.2 Implementation Using the Hybrid Controls API

The Hybrid Controls API establishes a framework that facilitates the implementation of hybrid systems on the OCP. The integration of the Hybrid Controls API with the OCP offers new transition management services that can be used in any component hybrid in nature. When a component is declared as a hybrid component on the OCP, a coordinator will be associated to that component. The coordinator is used to implement the coordination logic of the hybrid system. Usually this coordination logic is specified as a finite state machine that is triggered by values of the inputs and/or outputs of the component. The actual functionality of the component is implemented by several configurations defined by the user to represent each of the possible modes of operation of the component. Each configuration can have access to the inputs and outputs of the component. The coordinator decides which configuration should be active at a given time and has the ability to activate or deactivate the configurations as required.

The adaptive mode transition control architecture can be considered as a hybrid system. Specifically, the mode transition control component switches between local controllers and active control models as the plant is considered to be in a local mode or a transition according to what the mode transition manager subcomponent decides based on

the state of the plant. Therefore, the mode transition control can be implemented on the OCP using the transition management capabilities enabled by the Hybrid Controls API.

To exploit the Hybrid Controls API capabilities, two configurations are defined in the mode transition control component. One configuration is associated to the mode transition control component running a local controller in a local mode (Figure 16), and the other is associated to the mode transition control component running an active control model in a transition (Figure 17). In this implementation, the coordinator decides which configuration to use based on the information already processed by the mode transition manager subcomponent, so no additional state machine is required.

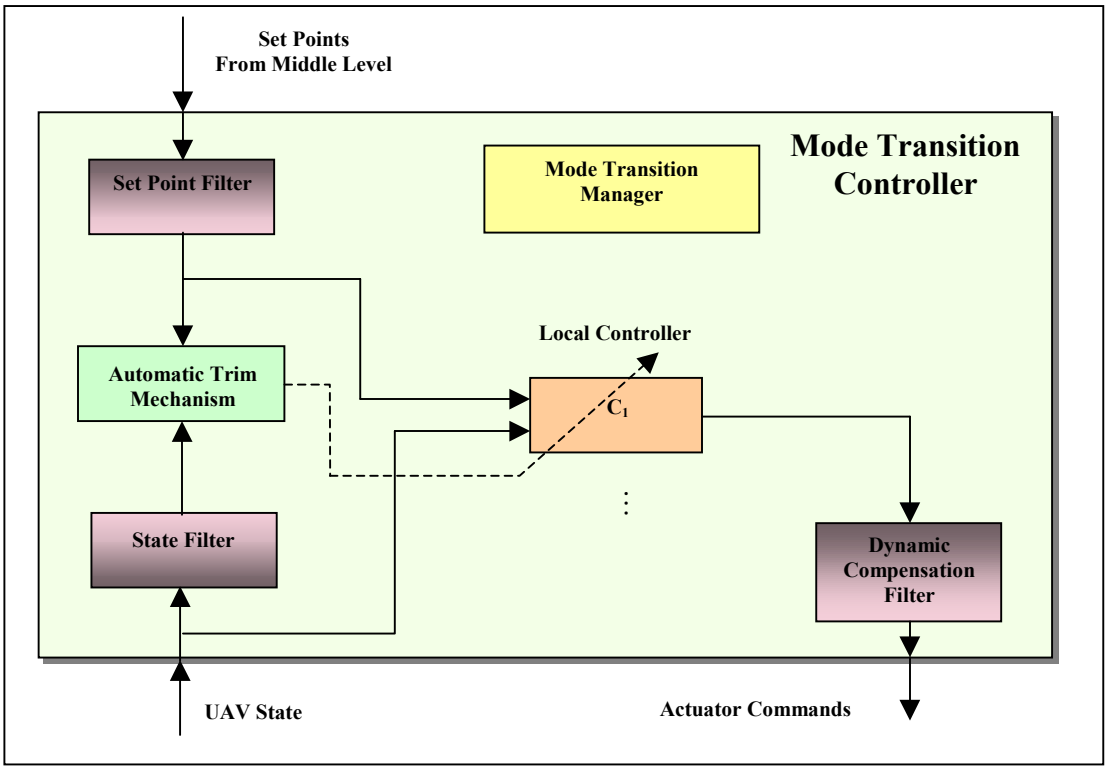


Figure 16. Local Mode Configuration for the Mode Transition Control Component

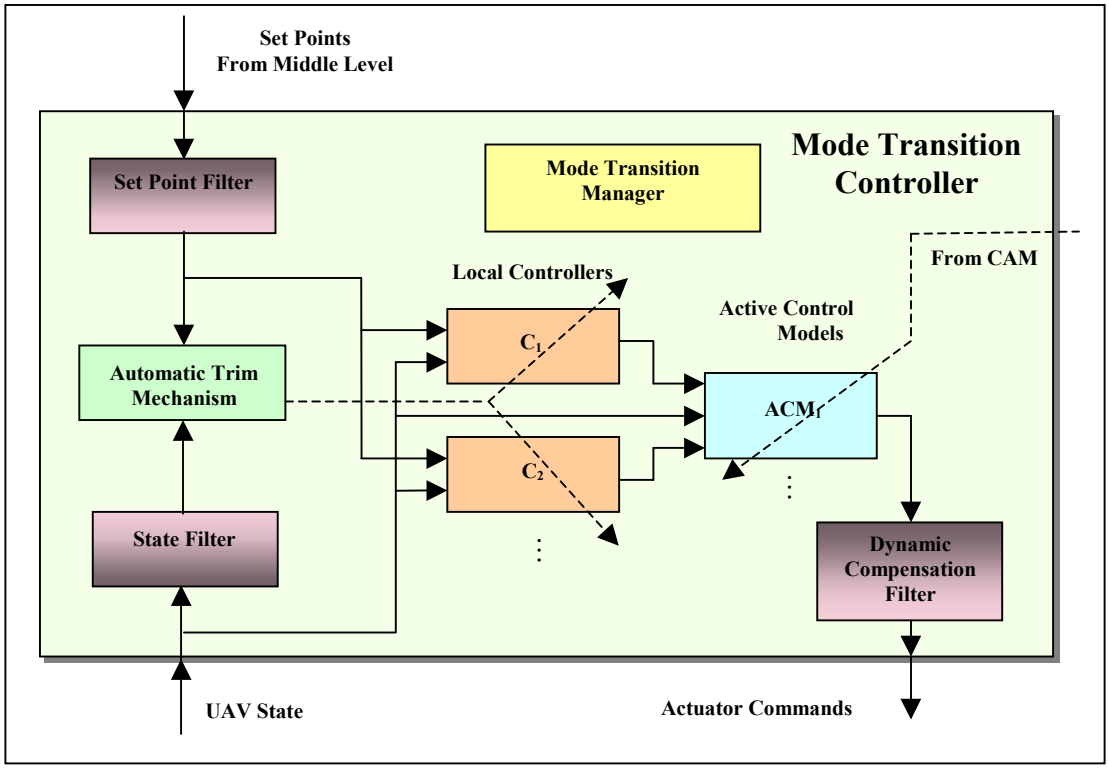


Figure 17. Transition Configuration for the Mode Transition Control Component

It would be possible to implement a configuration for each local mode and one configuration for each transition, but this is not practical. The current implementation of the mode transition control has the flexibility of allowing to define as many modes and as many transitions as required without changing a line of code. That is, the structure of the adaptive mode transition control architecture for an application is defined dynamically based on information provided in a configuration file. To keep this capability the local modes or transitions are not hard coded in the OCP implementation.

CHAPTER 8

SIMULATION AND FLIGHT TEST RESULTS

8.1 GTmax Simulation Environment

The GTmax is the UAV platform used for testing the adaptive mode transition control architecture (Figure 18). The GTmax belongs to the UAV lab of the School of Aerospace Engineering at Georgia Tech; it is based on a Yamaha Rmax industrial helicopter that has been instrumented to support research activities for UAVs [72].

Figure 18. The GTmax

The helicopter carries an avionics box instrumented with an Inertial Measurement Unit, GPS, Sonar Altimeter, a camera, and two computer processors that allow control algorithms to be implemented onboard the UAV. This box communicates with a ground control station through wireless Ethernet and wireless serial links. The ground control station includes a computer to monitor the vehicle status and send commands to the onboard controls.

The simulation environment accompanying the GTmax UAV, called the GTmax software, is a flexible software environment developed by Georgia Tech. Its modular structure not only allows performing software in the loop simulations, but also implements the software used in the vehicle to test different control algorithms. A screen shot of this simulation environment is presented in Figure 19. The GTmax software is composed of three basic modules: the ground control station, the onboard software, and the onboard 2 software. These modules are built into one or several executables depending of the configuration being tested. Three configurations are employed for testing as discussed in the sequel.

8.2 Simulation and Flight Configurations

8.2.1 Software in the Loop Simulation Configuration

For the purpose of validation and verification of all the algorithms, a strict sequence of tests has been performed. Upon development of the algorithms and implementation of their code, extensive software in the loop simulations were carried out using the GTmax simulation environment. These first tests allowed detection of glitches in the algorithms and the code, and also permitted tuning of the parameters involved

without putting the actual vehicle into risk. The GTmax software simulates not only the model of the vehicle but also the sensors, the actuators, and the communications between the components. Furthermore, given that the code used in the simulations was the same used in the actual vehicle, the results of the software in the loop simulation were pretty valuable in improving the chances of success in actual flight tests.

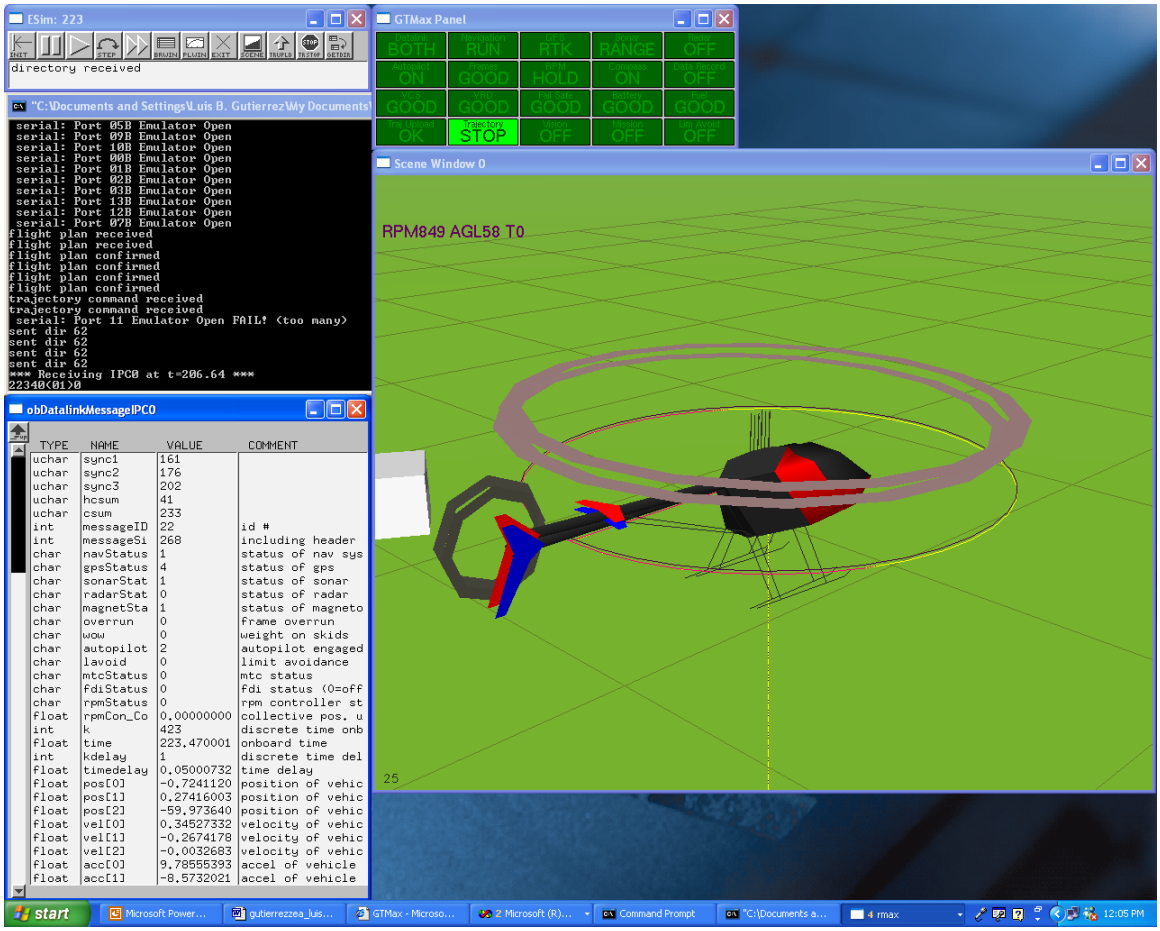


Figure 19. A Screen Shot of the GTmax Software

For this kind of simulations, the code of the adaptive mode transition control architecture was integrated with the GTmax simulation environment (Figure 20). This integration was accomplished inserting some code in the onboard2 module of the GTmax

software that called the appropriate functions to initialize and execute the adaptive mode transition control architecture. For this purpose, the software interface with the adaptive mode transition control library discussed in section 6.2.4 was used. After inserting the appropriate code, the Gtmax software was compiled and linked to the adaptive mode transition control library to generate executables for Windows and Linux. Several scripts, known as input files for the GTmax software, were created to perform the following tasks: start and stop the communication with the onboard2 module including the adaptive mode transition control architecture, start and stop the adaptive mode transition controller, load a mission, and update the structure and parameters of the adaptive mode transition controller. The same scripts were used for hardware in the loop simulations and flight testing.

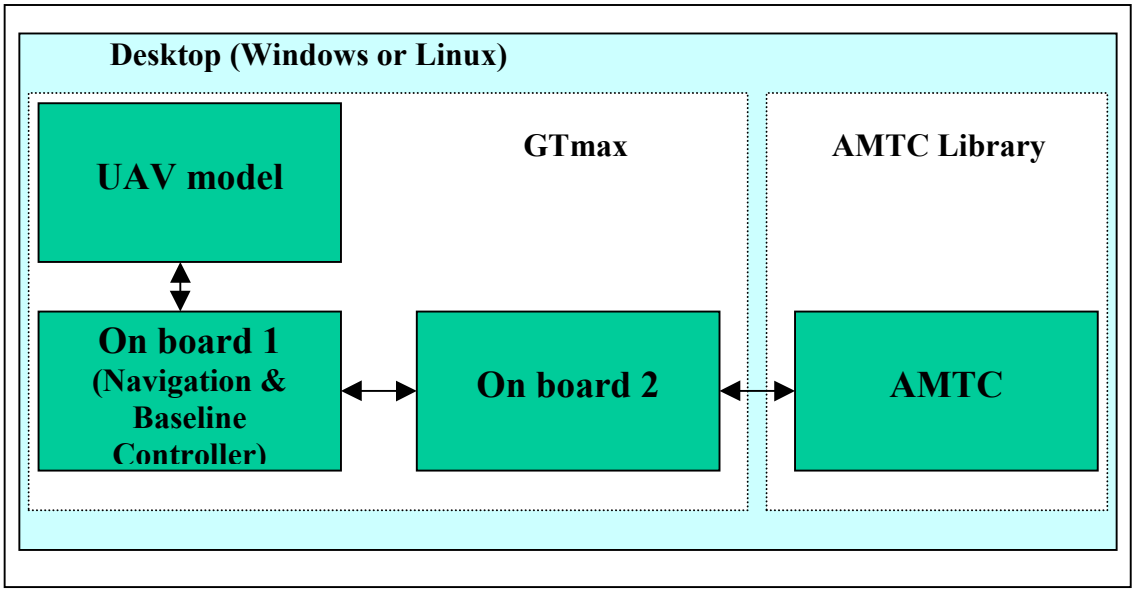


Figure 20. Software in the Loop Configuration

8.2.2 Hardware in the Loop Simulation Configuration

After extensive software in the loop simulations were completed successfully, the next step was the hardware in the loop simulation. This kind of simulation tests the algorithms on the actual hardware used on the vehicle verifying them under the processing constraints imposed by it. The software configuration for this kind of simulations is shown in Figure 21. Here, the same executable used in software in the loop simulations is employed to simulate the ground control station and the onboard module connected to a model of the dynamics of the vehicle and is sensors. The onboard 2 module in that executable is deactivated since the communications are redirected so the onboard module now communicates with the onboard 2 module in a separate executable.

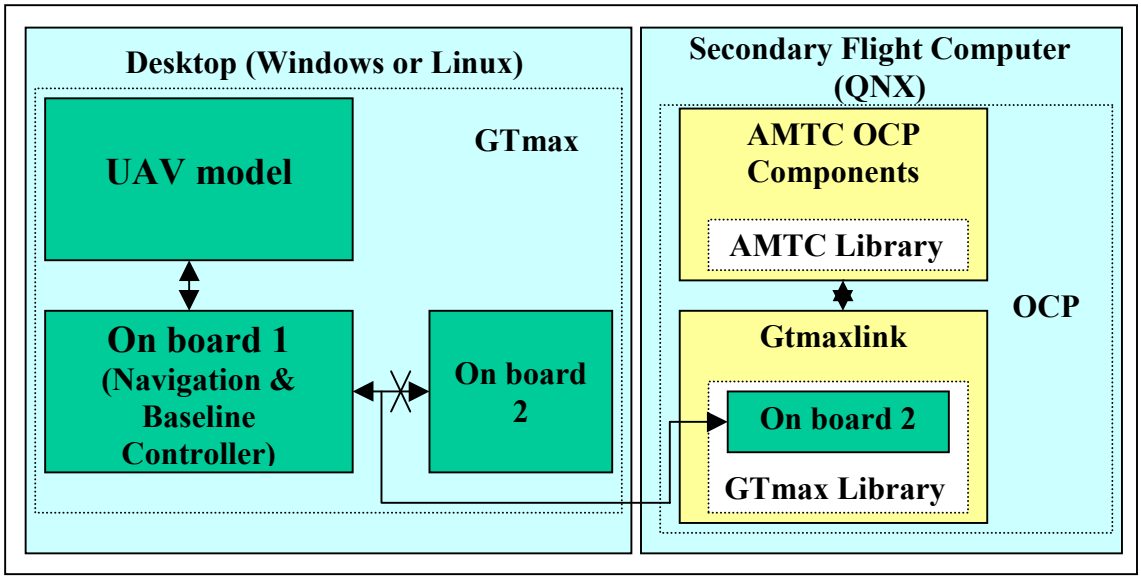


Figure 21. Hardware in the Loop Configuration

This onboard 2 module is running in the secondary computer of the GTmax UAV under QNX, exactly as it would do in flight. Given that the software of the adaptive mode

transition control architecture is running in the actual operating system and hardware used in flight, this kind of simulation is an excellent test to make sure the software will perform well in flight.

8.2.3 Flight Test Configuration

Once the results of the hardware in the loop simulation were satisfactory, the algorithms were tested in flight on the actual vehicle. The software configuration used for flight testing is shown in Figure 22. In this case all modules of the GTmax software are compiled in separate executables. The ground control station is compiled for Windows and runs in the ground control station computer. The onboard and onboard 2 modules are compiled for QNX and run in the primary and secondary flight computers onboard the vehicle. The tests are performed in the same way that the previous simulations except that

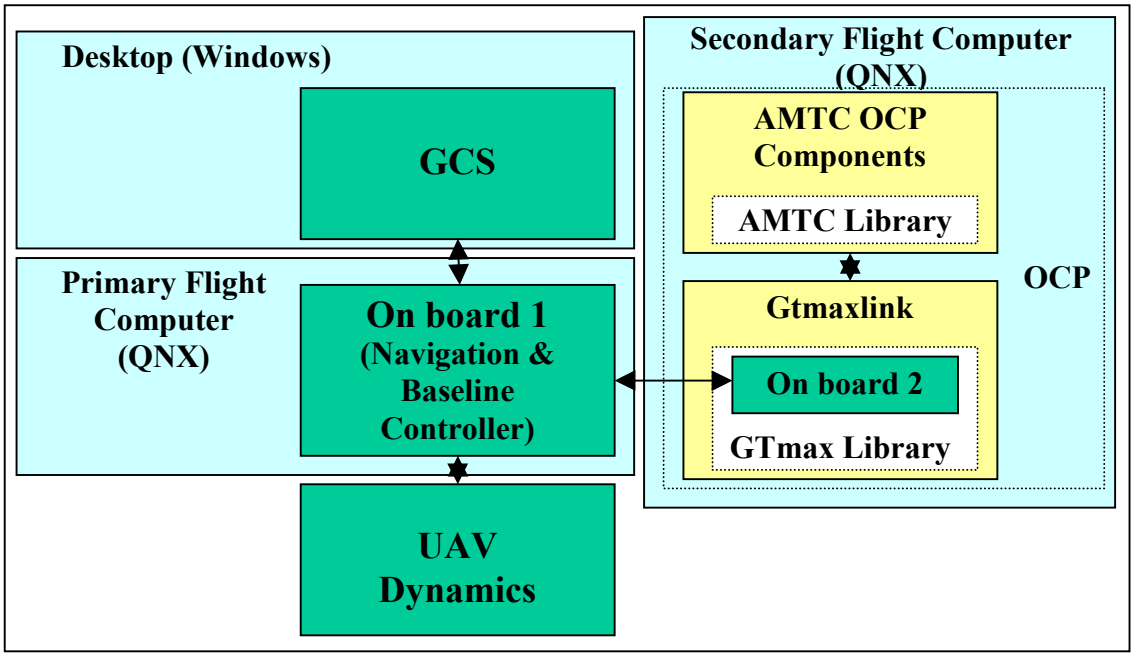


Figure 22. Flight Configuration

now the adaptive mode transition control architecture controls the actual vehicle in flight, and all the flight infrastructure is in place.

8.3 Parameters for Simulations and Flight Test

The following configuration and parameters were used for the simulations and flight tests presented here:

- The adaptive mode transition controller was set to operate at a sample rate of 25 Hz (sample period of 40msec).
- Four local modes were defined: mode one for hover (zero speed), mode two for forward flight at 25ft/sec, mode three for forward flight at 50ft/sec, and mode four for backward flight at 25ft/sec. Therefore, there were four associated local controllers and local plant models. The inverse deviations associated to these modes were set so the extent of each mode in terms of forward and sideward velocities corresponded to a circle with radius 2.5ft/sec.
- Local controllers for the augmented system including the integral control used the following weights for the LQR design:

  Q = diag([1 , 1 , 100 , 3282.81 , 1 , 1 , 1 , 100 , 3282.81 , 3282.81 , 3282.81 , 100 ,
  100 , 100 , 3282.81 , 3282.81 , 3282.81 , 4])

  R = diag([1e+006 , 40000 , 160000 , 40000 , 10000])
- Three transitions were defined: transition one between modes one and two, transition two between modes two and three, and transition three between modes one and three. Therefore, there were three active control models and three active plant models associated to the transitions.

- The set point for the main rotor angular velocity was set to 850 rpm.
- The set point filter was set to have a cutoff frequency of 1 Hz and a limiting damping factor of 2. The state filter was set to have a cutoff frequency of 1Hz.

8.4 Software in the Loop Simulation Results

The hierarchical control architecture was first tested in software-in-the-loop simulation using the GTmax simulation environment. The results for four flight segments are presented here:

- Hover pointing North. Software in the loop simulation results for this test are shown in Figures 23 to 27.
- Hover pointing North with heading changes to point East, West, and South. Software in the loop simulation results for this test are shown in Figures 28 to 32.
- Hover - flight forward at 20ft/sec - hover – flight backward at 20ft/sec – hover. Software in the loop simulation results for this test are shown in Figures 33 to 37.
- A smooth box at 20ft/sec - move 400ft North, then move 400ft East, then move 800ft South, then move 400ft west, and finally move to initial position. Software in the loop simulation results for this test are shown in Figures 38 to 42.

To measure the performance of the controller, the following metrics were computed for each flight segment: mean magnitude of the position error, maximum

magnitude of the position error, mean magnitude of the heading error, and maximum magnitude of the heading error. The results are summarized in Table 1.

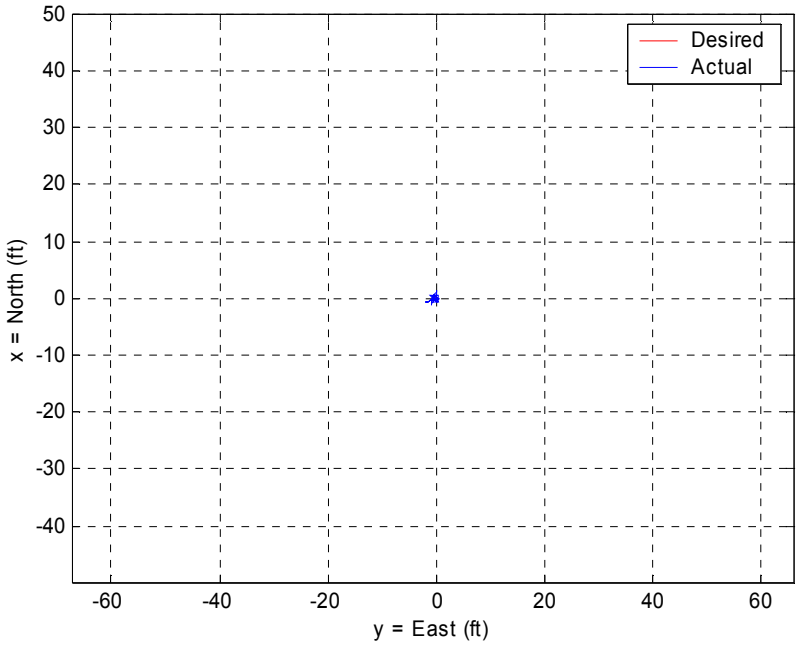
Table 1. Performance Metrics for Software in the Loop Simulations

Flight Segment	Mean Magnitude of Position Error (ft)	Max Magnitude of Position Error (ft)	Mean Magnitude of Heading Error (deg)	Max Magnitude of Heading Error (deg)
Hover	0.6267	1.8414	0.4011	1.4979
Hover with heading changes	0.7069	1.9734	2.4496	17.9465
Hover - flight forward at 20ft/sec - hover – flight backward at 20ft/sec – hover	4.3694	13.3478	2.0763	8.4715
A smooth box at 20ft/sec	3.6502	10.3857	4.1896	17.4970

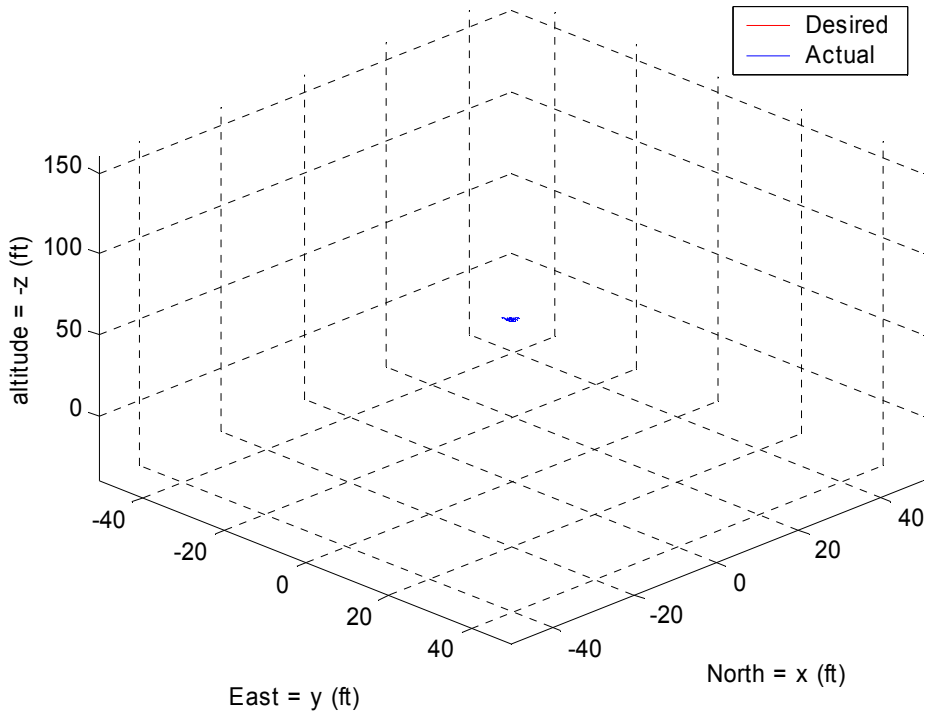
It is observed how the controller makes the vehicle track the desired trajectories keeping small errors. The first two flight segments show the behavior of the controller in the local mode corresponding to hover. Third and fourth flight segments show the performance of the controller in a local mode and transition regions. In the third flight segment the vehicle transitions from mode 1 (hover) to mode 2 (forward flight at 25ft/sec) but, given that the set points set the velocity to 20ft/sec, the vehicle stay in the transition region when flying forward. Then, the vehicle decelerates and stay in mode 1 (hover) for some time followed for a transition to mode 4 (backward flight at 25 ft/sec) flying backwards and reaching a speed of 20ft/sec corresponding to the set points and then it decelerates to return to hover. The fourth flight segment shows the ability of the

controller to track an arbitrary trajectory while performing the required transitions. Whenever the vehicle accelerates or decelerates the error is kept bounded even though there are peaks in the position error especially in the axis aligned with the acceleration vector. This is the normal behavior of any controller.

Figures 27, 32, 37, and 42 show the errors of the plant models for each flight segment. These errors correspond to the differences between the actual value of the state and the values produced by the active plant models evaluated in the previous value of the state and control input. It is observed how the mean of these errors is kept close to zero and the magnitude is kept small demonstrating that the offline training and online plant adaptation mechanism is working properly adjusting the models to represent the dynamics of the vehicle.

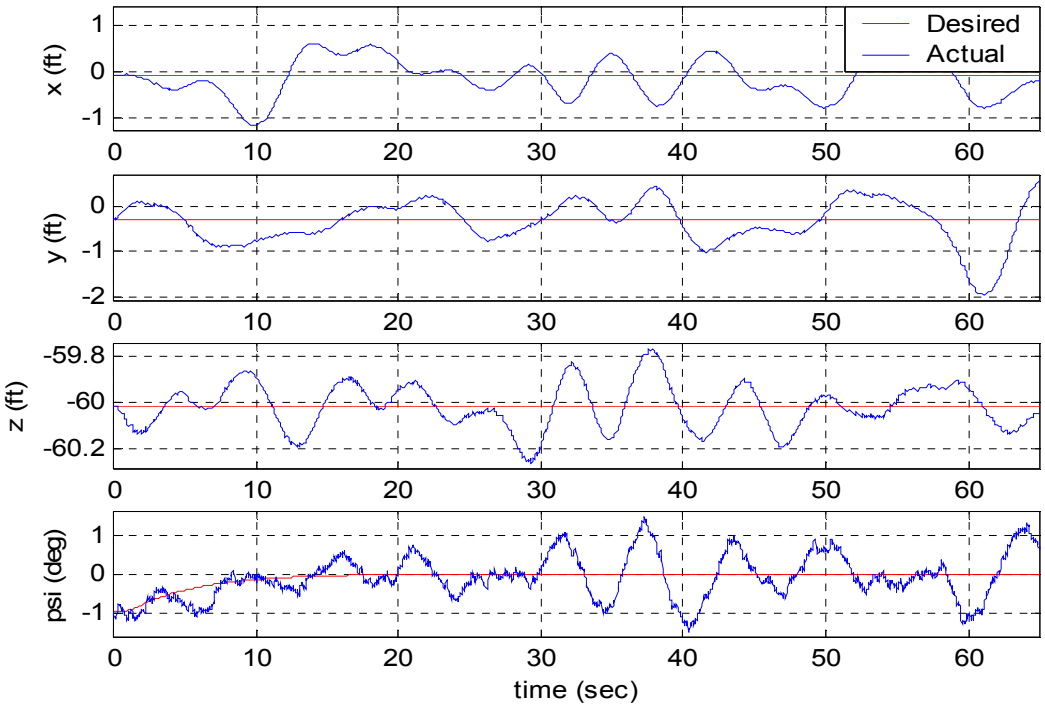


(a)

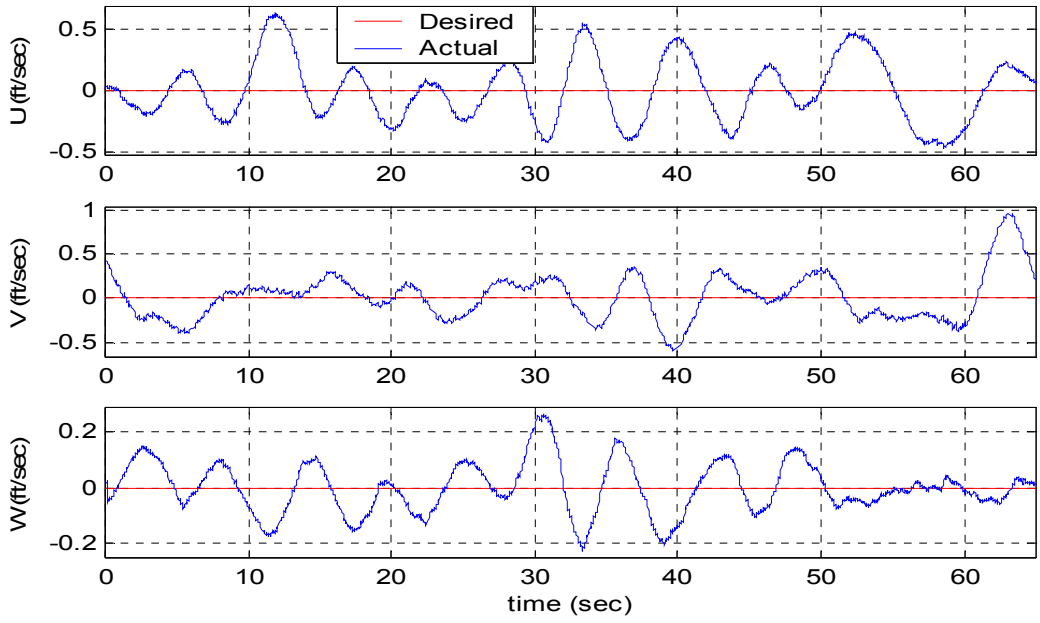


(b)

Figure 23. Software in the Loop Simulation Results for Hover: (a) Desired and Actual 2D Trajectory; (b) Desired and Actual 3D Trajectory

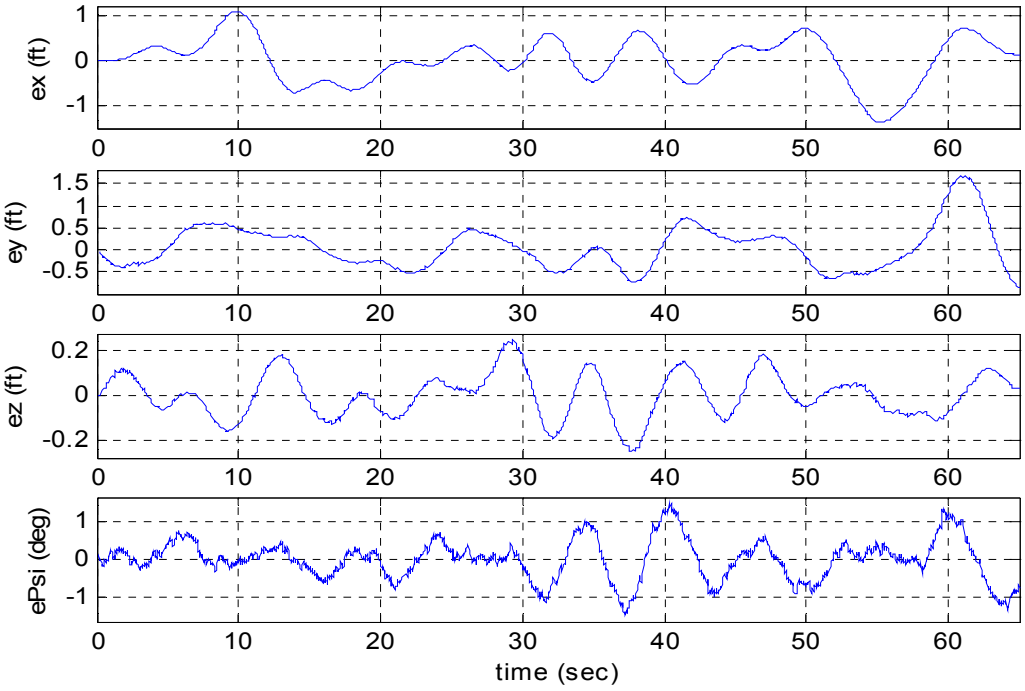


(a)

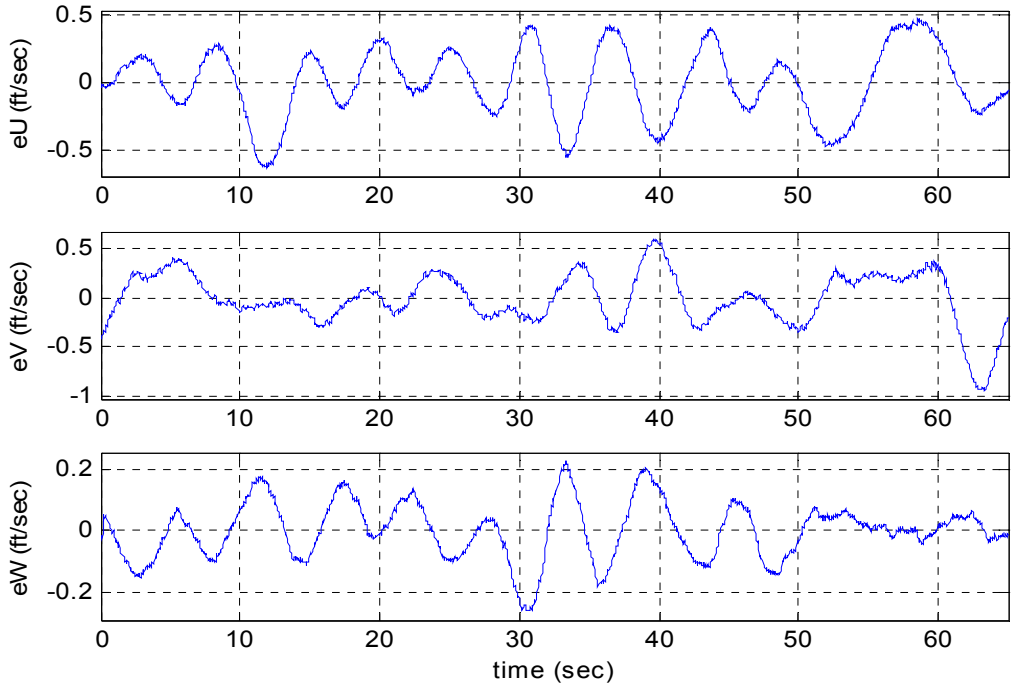


(b)

Figure 24. Software in the Loop Simulation Results for Hover: (a) Desired and Actual Position and Heading; (b) Desired and Actual Velocity in Body Frame



(a)



(b)

Figure 25. Software in the Loop Simulation Results for Hover: (a) Position and Heading Errors; (b) Velocity Errors in Body Frame

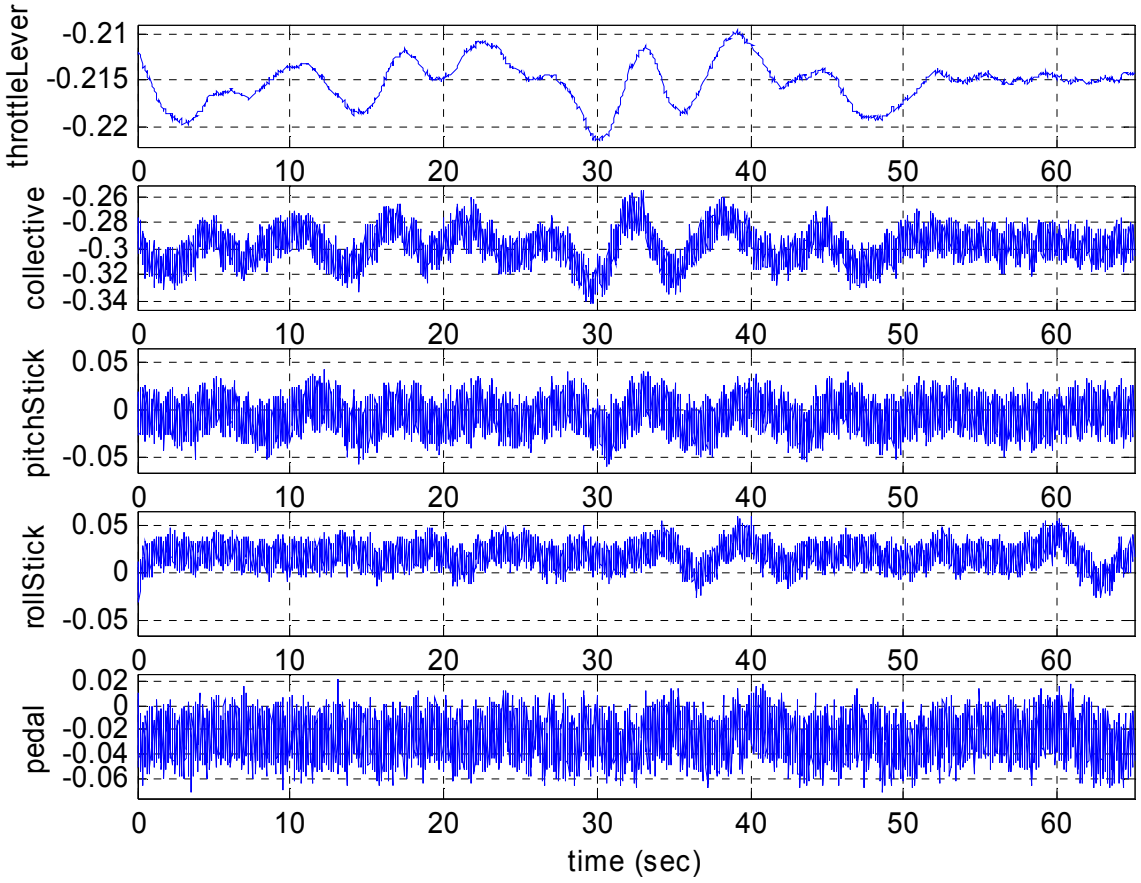


Figure 26. Software in the Loop Simulation Results for Hover: Actuator Commands

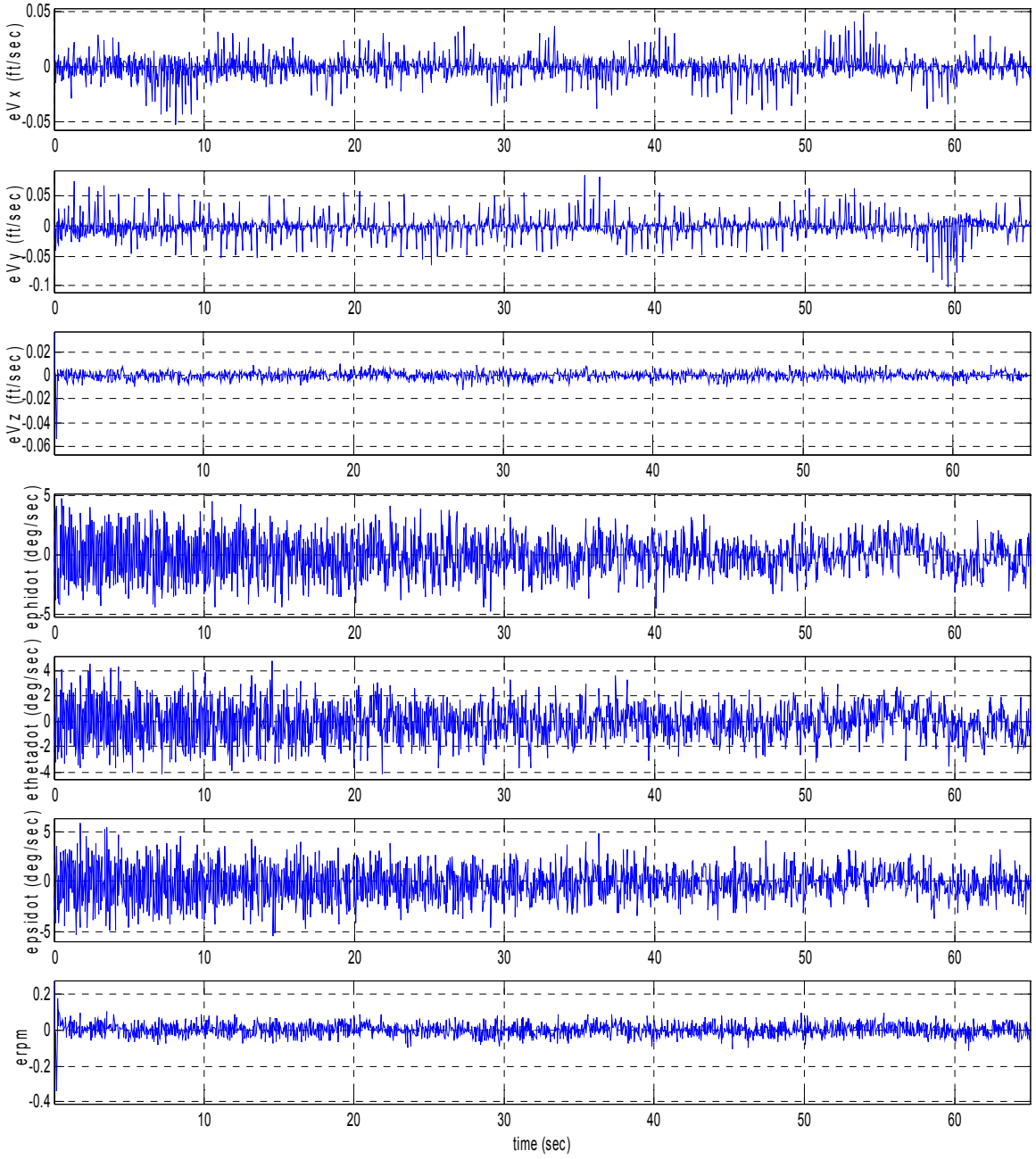
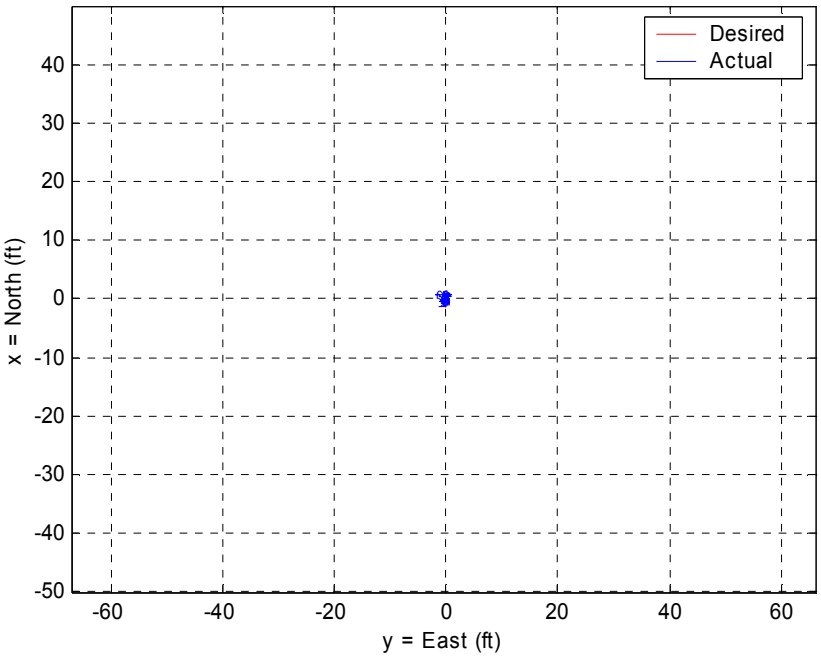
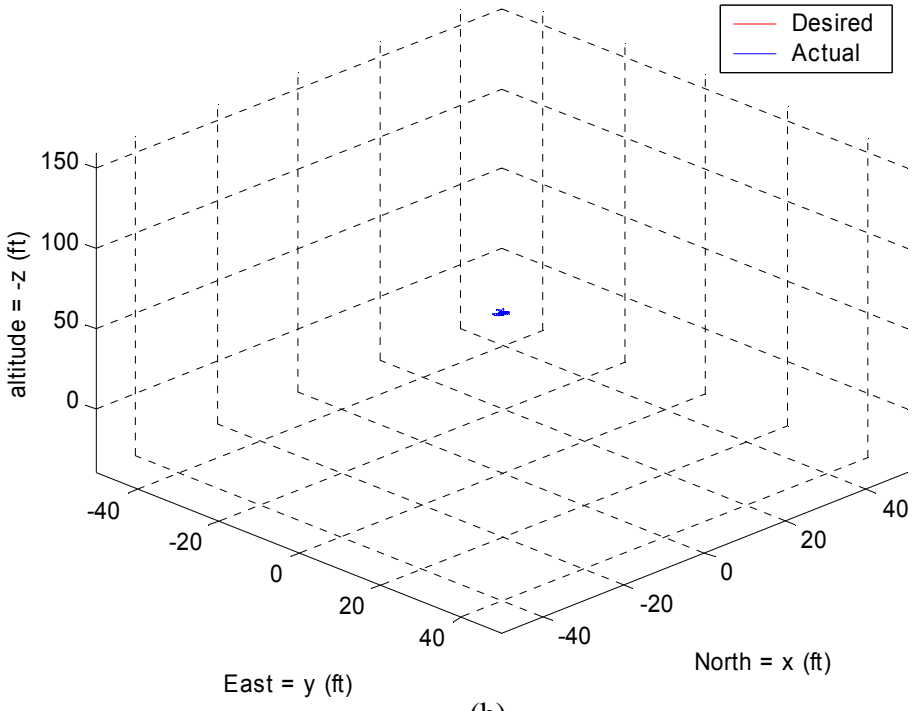


Figure 27. Software in the Loop Simulation Results for Hover: Plant Model Errors for Velocity, Euler Angle Rates and Main Rotor RPM

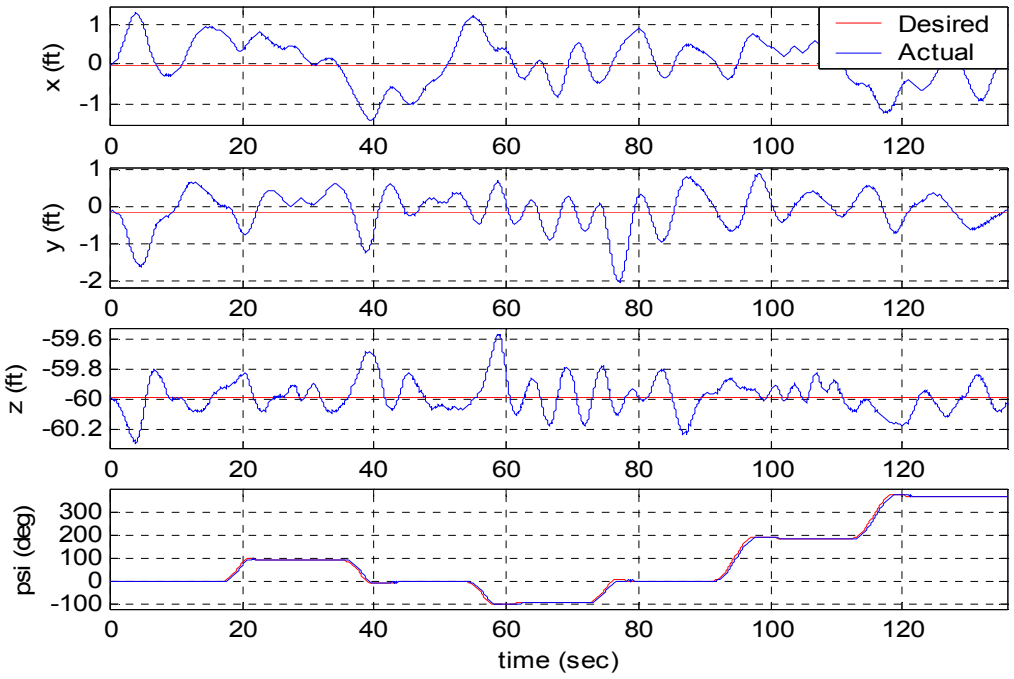


(a)

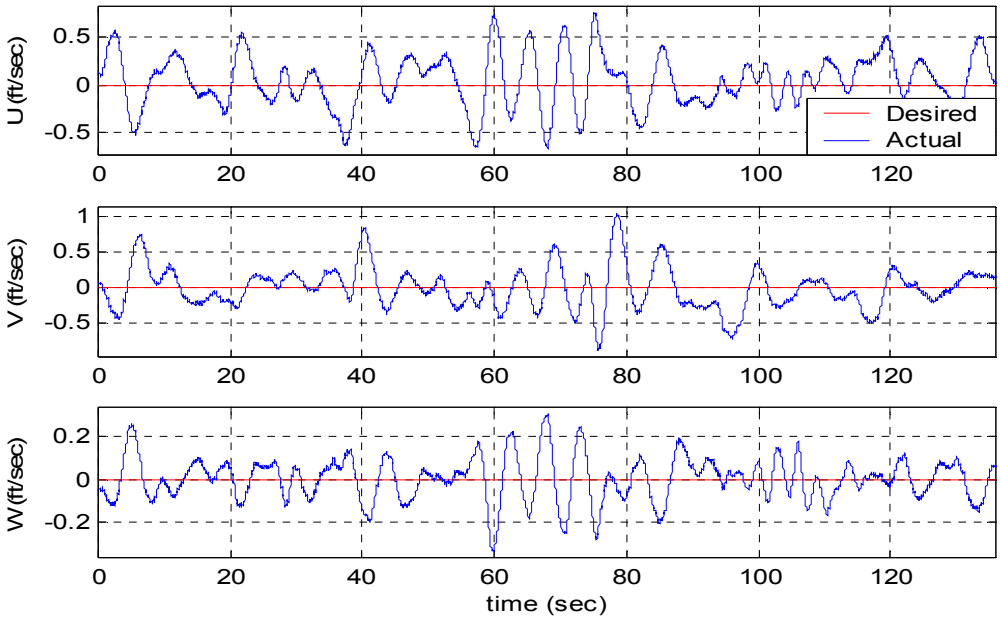


(b)

Figure 28. Software in the Loop Simulation Results for Hover with Heading Changes: (a) Desired and Actual 2D Trajectory; (b) Desired and Actual 3D Trajectory

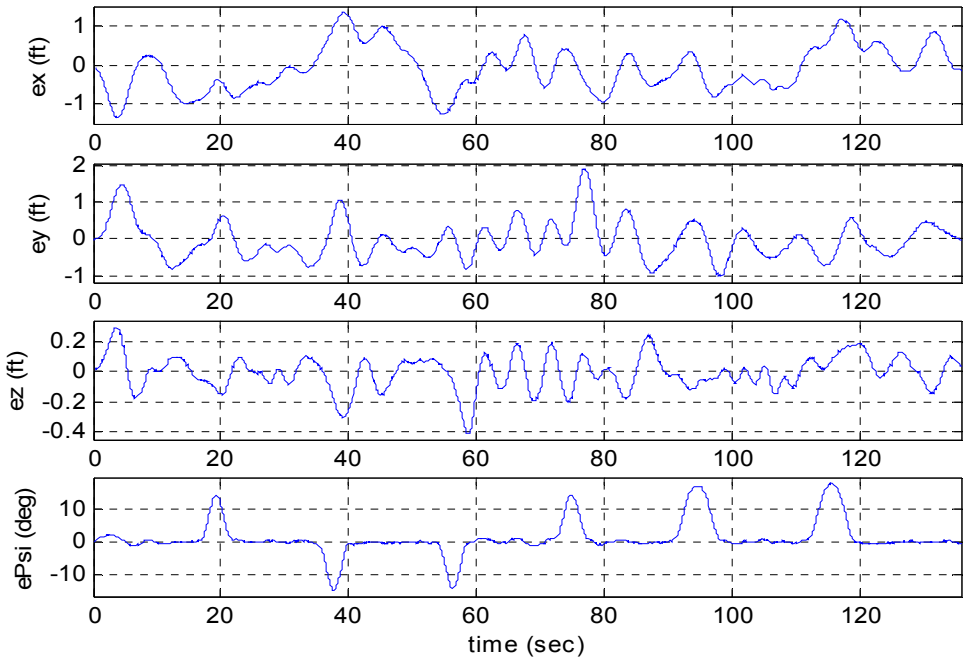


(a)

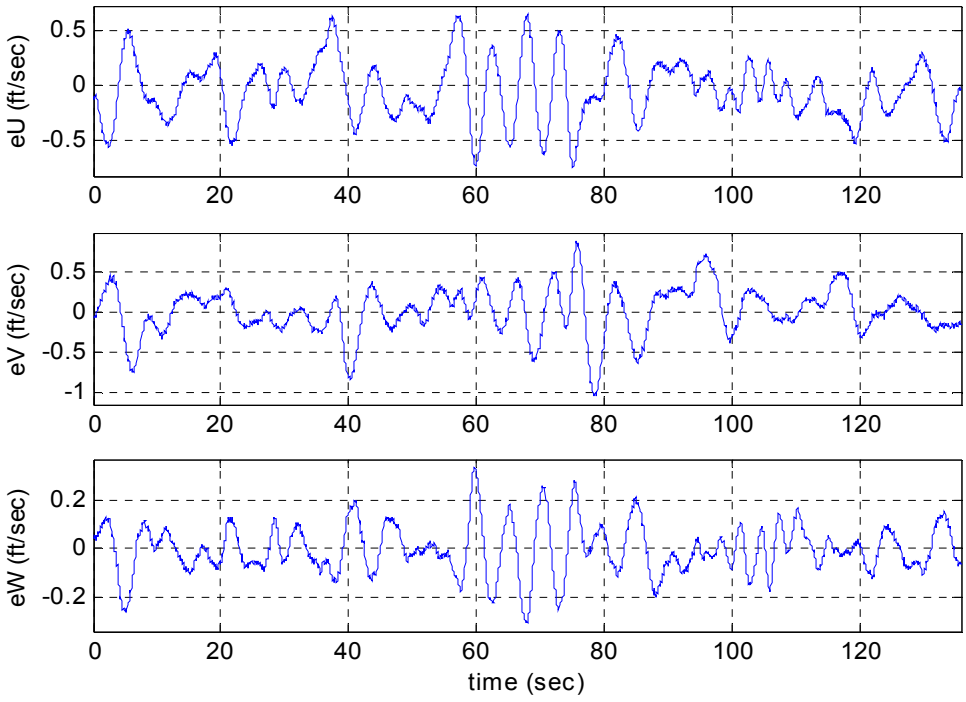


(b)

Figure 29. Software in the Loop Simulation Results for Hover with Heading Changes: (a) Desired and Actual Position and Heading; (b) Desired and Actual Velocity in Body Frame



(a)



(b)

Figure 30. Software in the Loop Simulation Results for Hover with Heading Changes: (a) Position and Heading Errors; (b) Velocity Errors in Body Frame

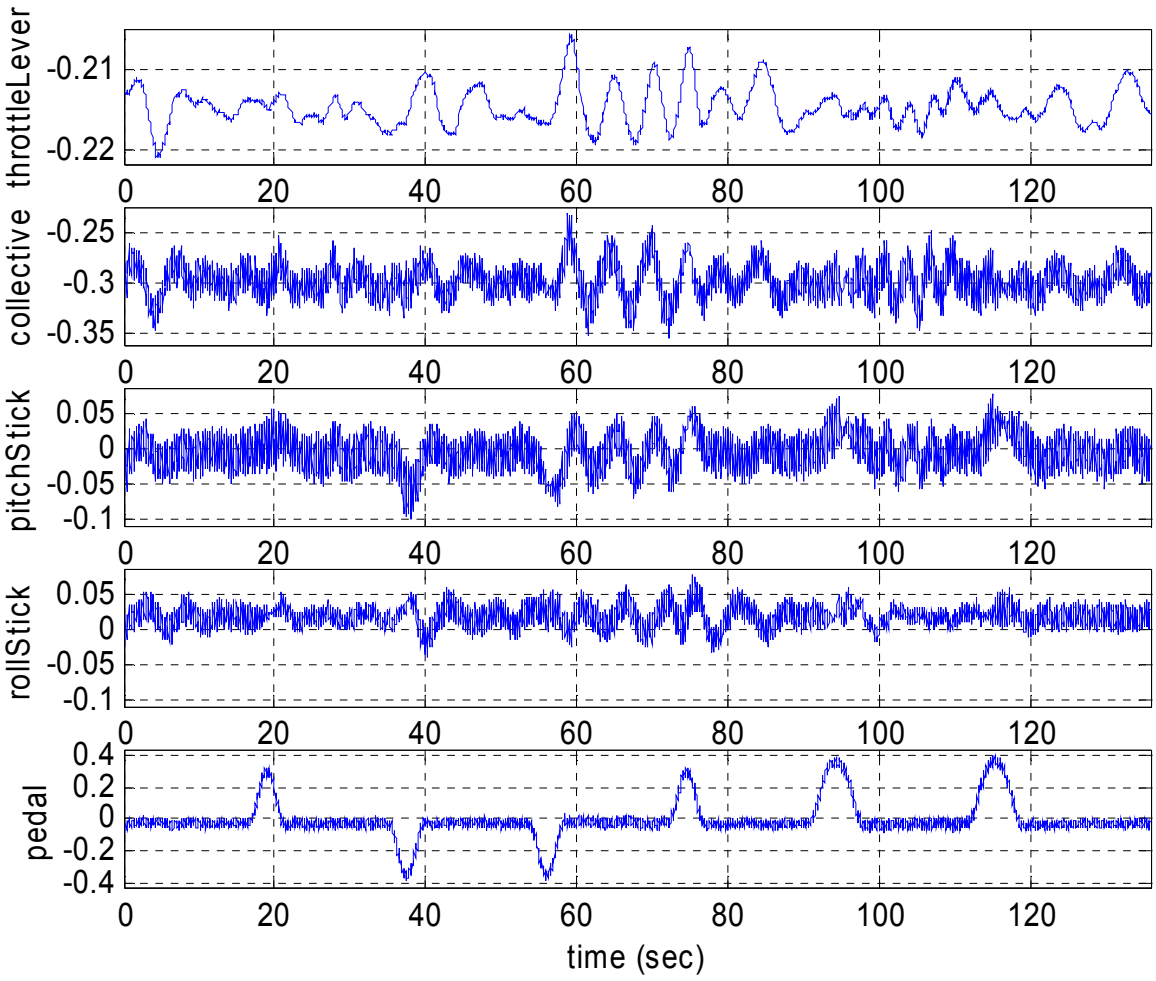


Figure 31. Software in the Loop Simulation Results for Hover with Heading Changes: Actuator Commands

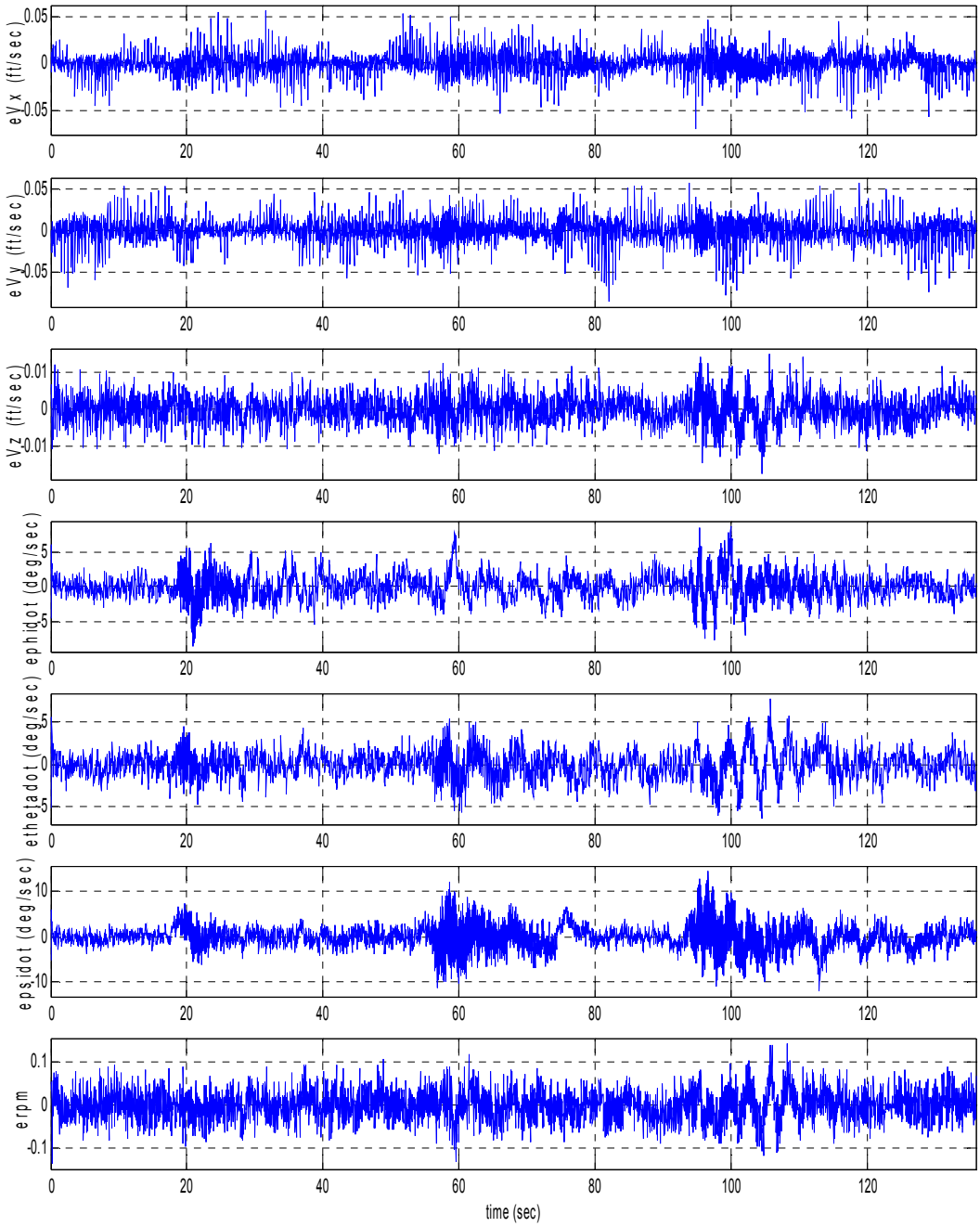
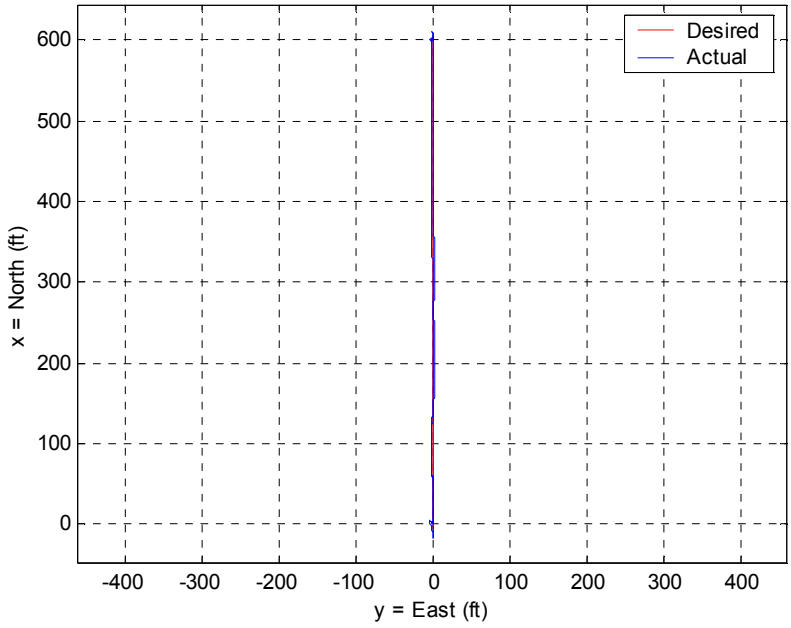
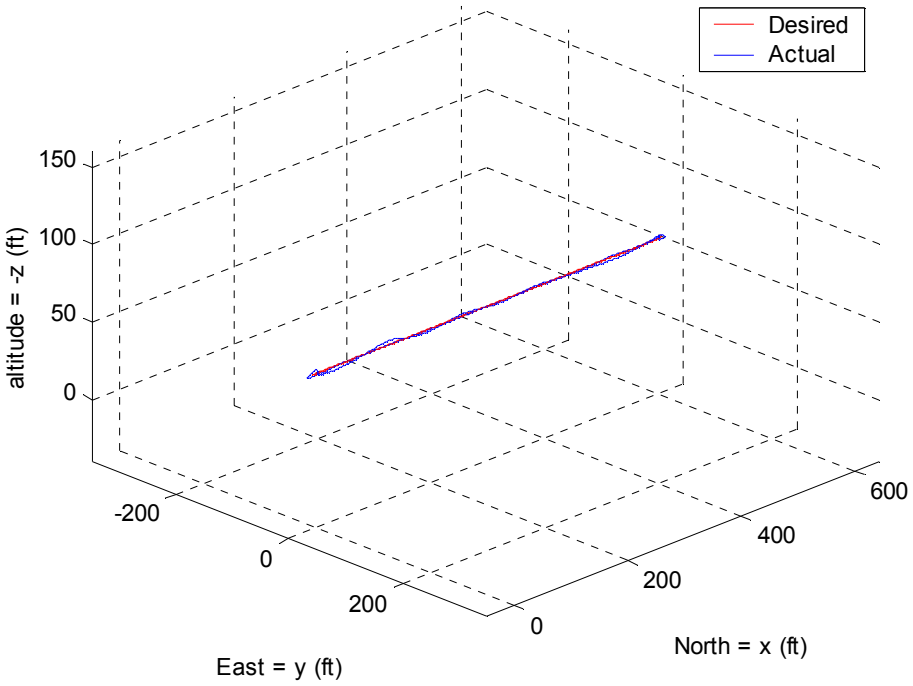


Figure 32. Software in the Loop Simulation Results for Hover with Heading Changes: Plant Model Errors for Velocity, Euler Angle Rates and Main Rotor RPM

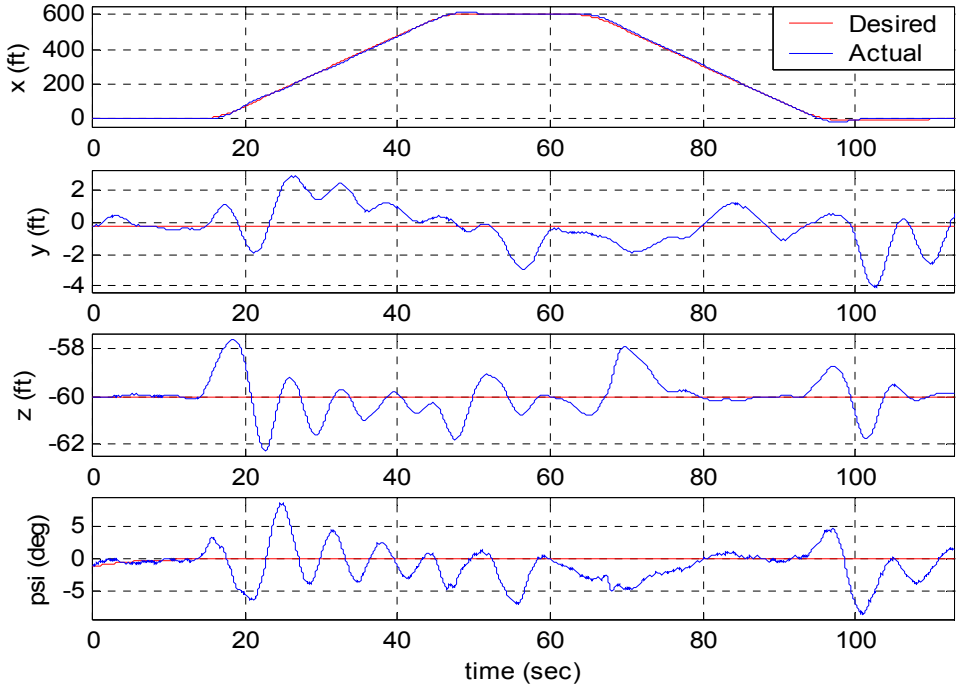


(a)

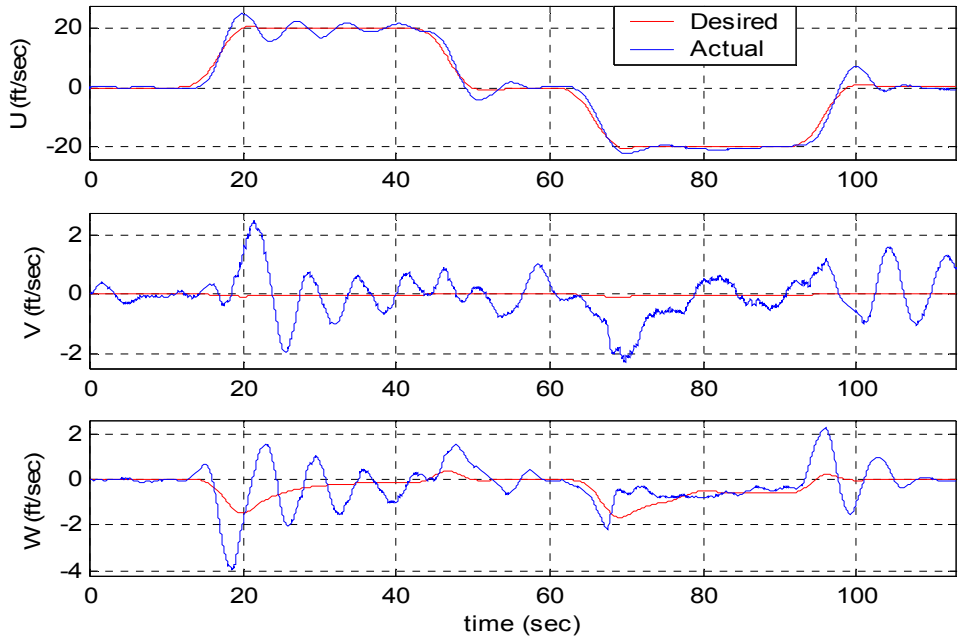


(b)

Figure 33. Software in the Loop Simulation Results for Hover - Flight Forward at 20ft/sec - Hover – Flight Backward at 20ft/sec – Hover: (a) Desired and Actual 2D Trajectory; (b) Desired and Actual 3D Trajectory

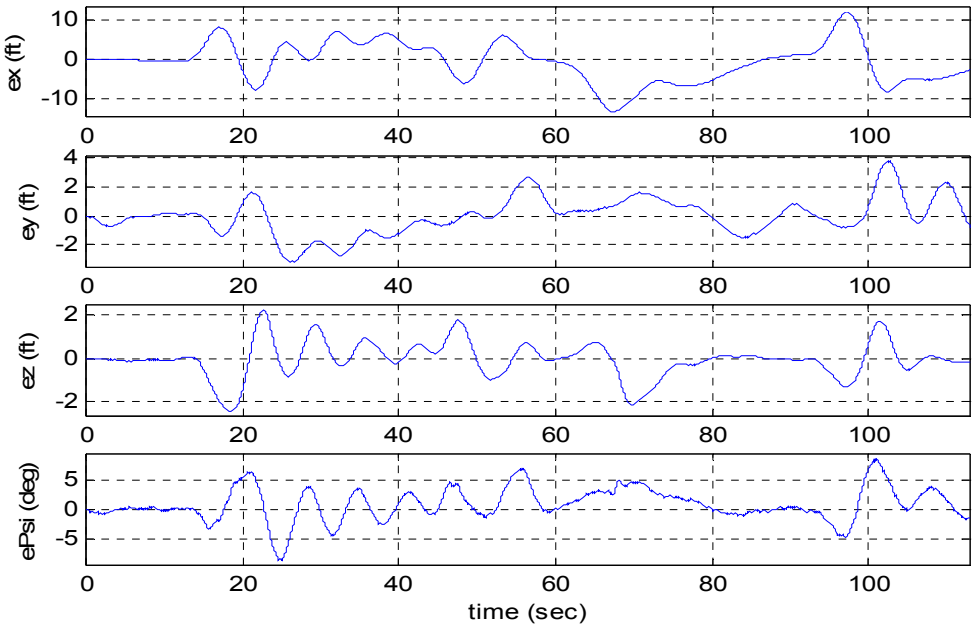


(a)

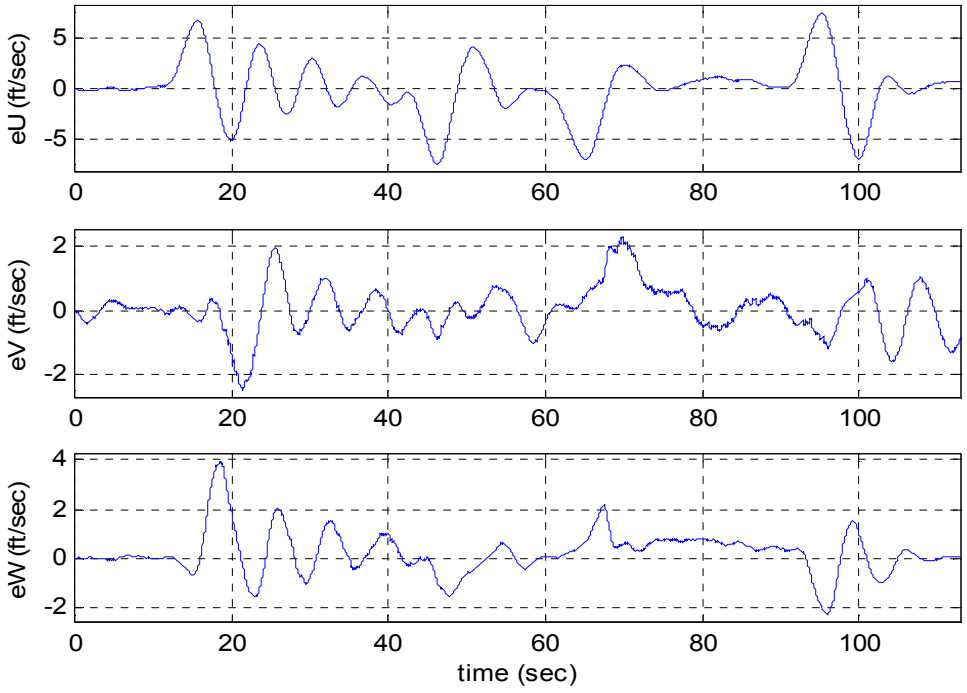


(b)

Figure 34. Software in the Loop Simulation Results for Hover - Flight Forward at 20ft/sec - Hover – Flight Backward at 20ft/sec – Hover: (a) Desired and Actual Position and Heading; (b) Desired and Actual Velocity in Body Frame



(a)



(b)

Figure 35. Software in the Loop Simulation Results for Hover - Flight Forward at 20ft/sec - Hover – Flight Backward at 20ft/sec – Hover: (a) Position and Heading Errors; (b) Velocity Errors in Body Frame

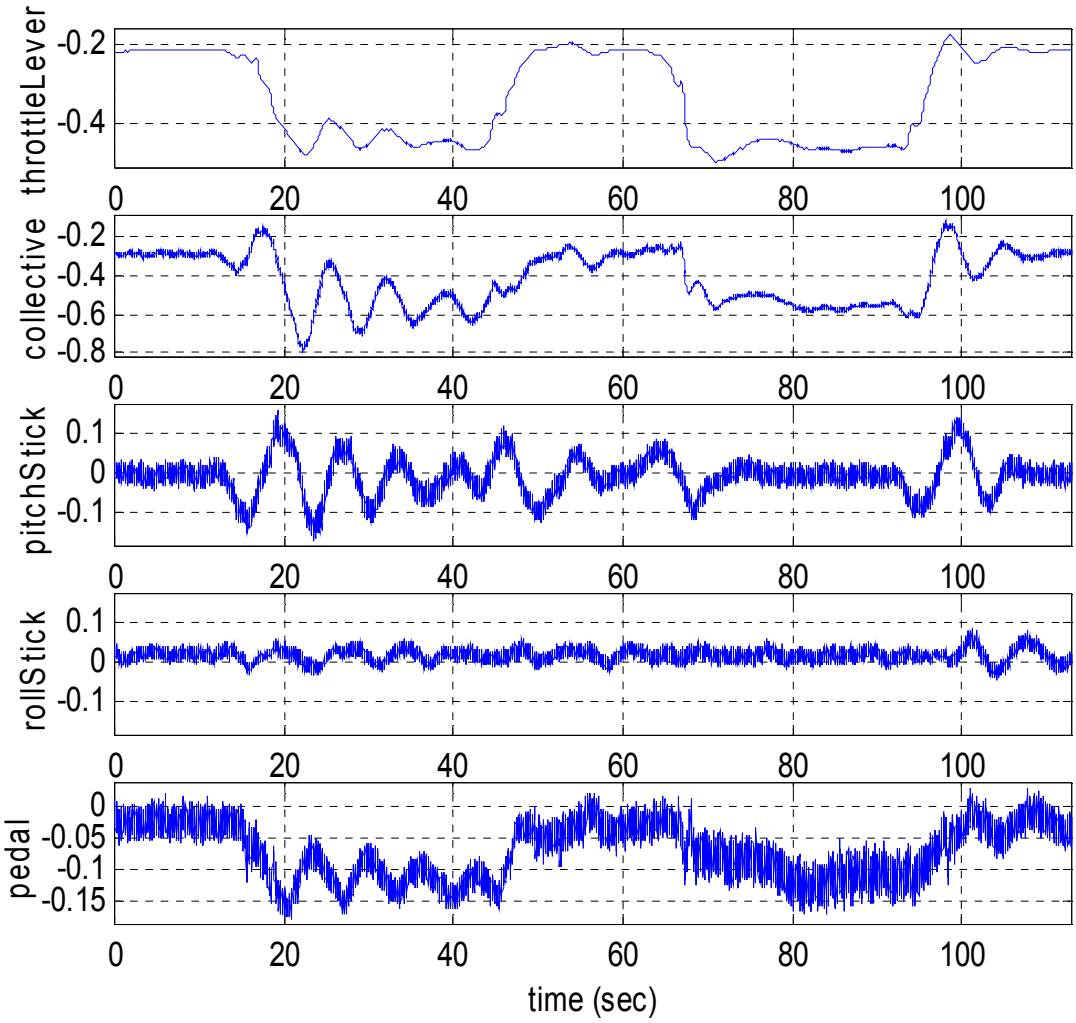


Figure 36. Software in the Loop Simulation Results for Hover - Flight Forward at 20ft/sec - Hover – Flight Backward at 20ft/sec – Hover: Actuator Commands

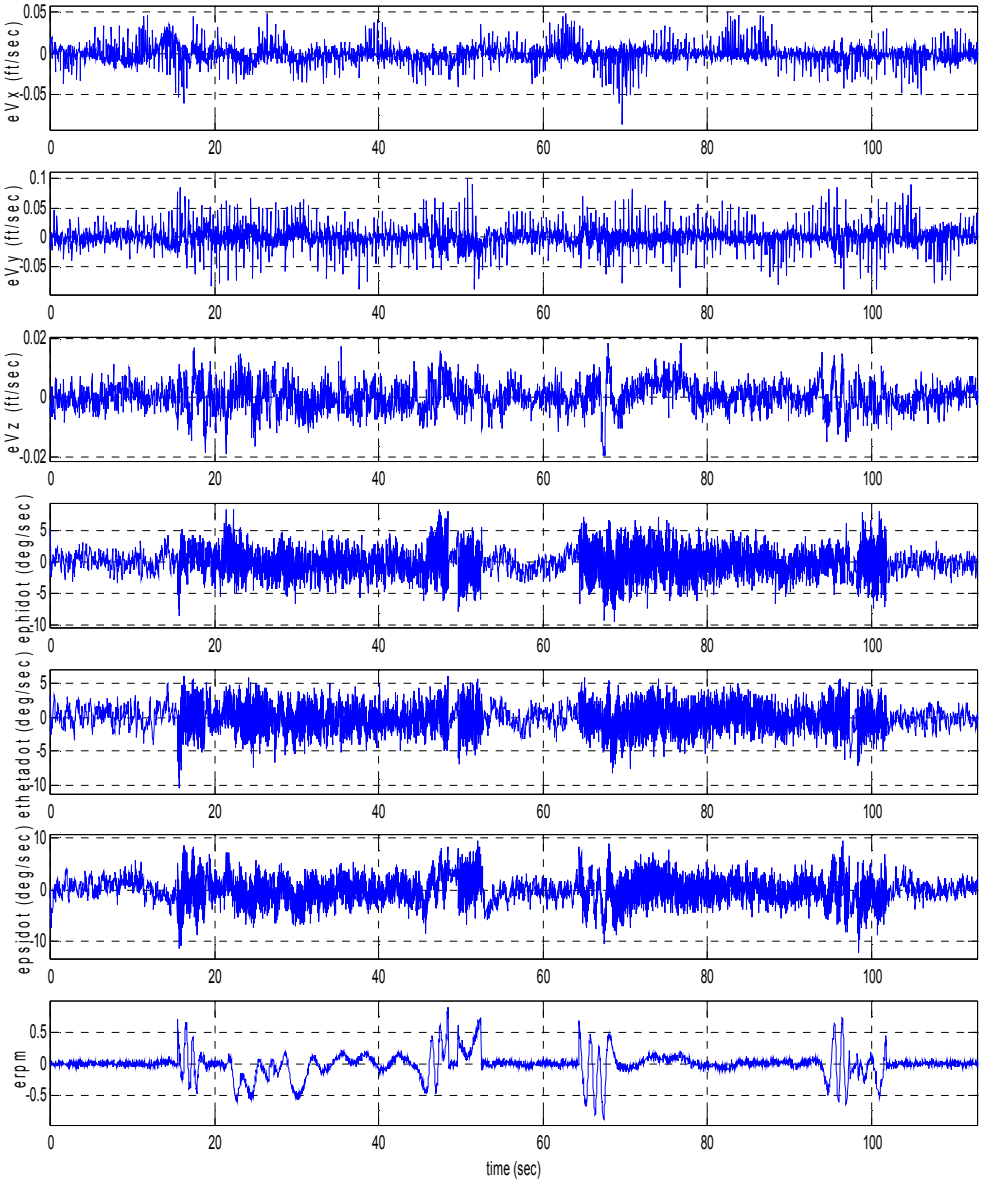
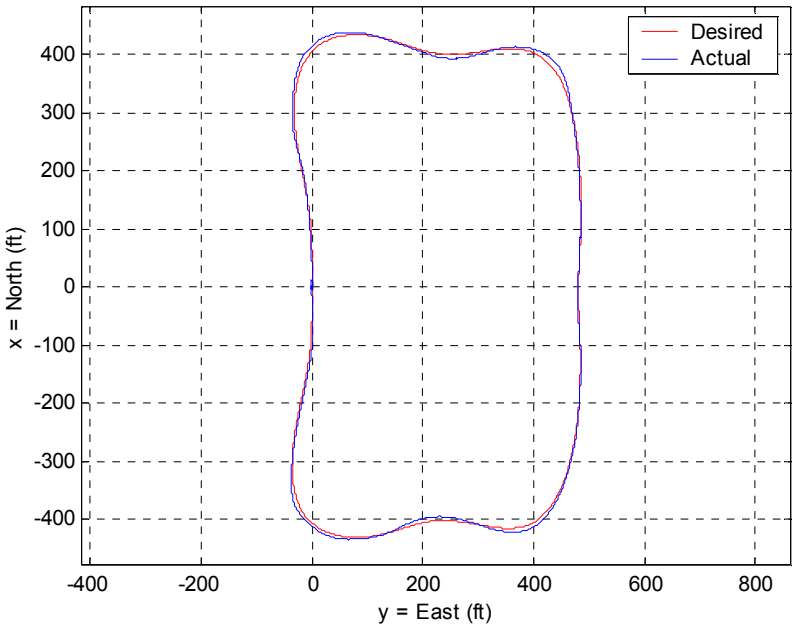
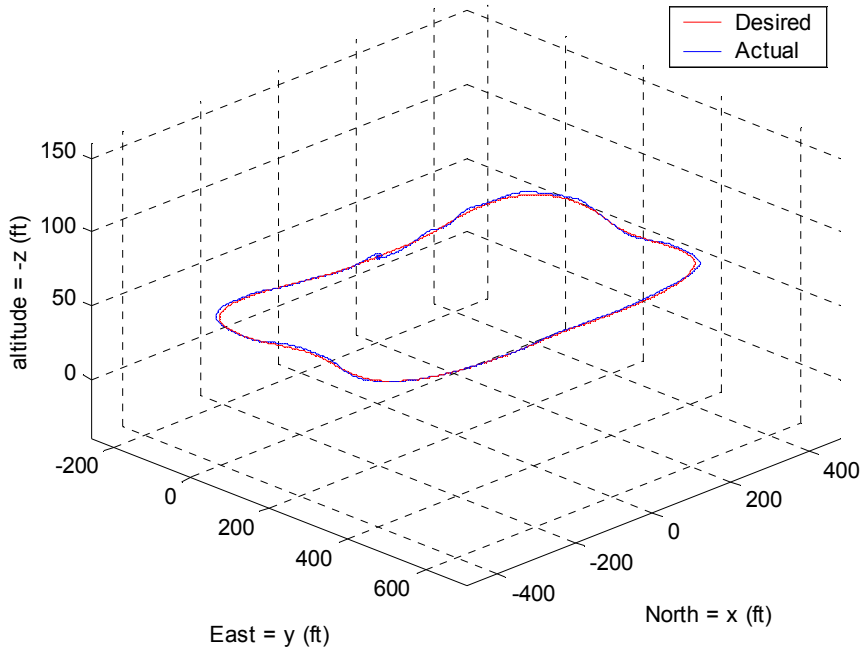


Figure 37. Software in the Loop Simulation Results for Hover - Flight Forward at 20ft/sec - Hover – Flight Backward at 20ft/sec – Hover: Plant Model Errors for Velocity, Euler Angle Rates and Main Rotor RPM

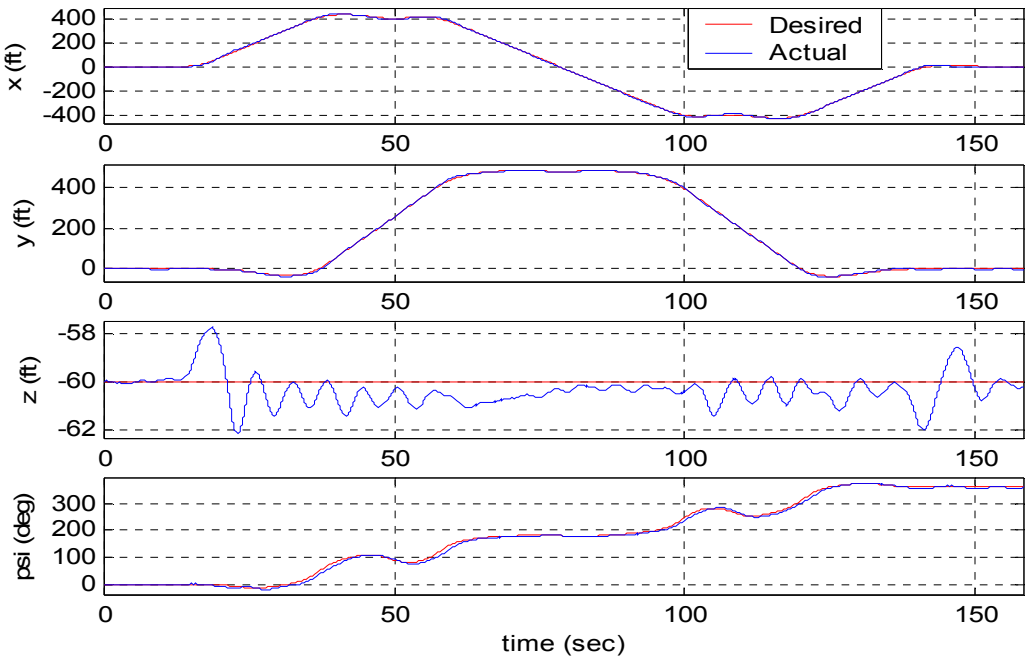


(a)

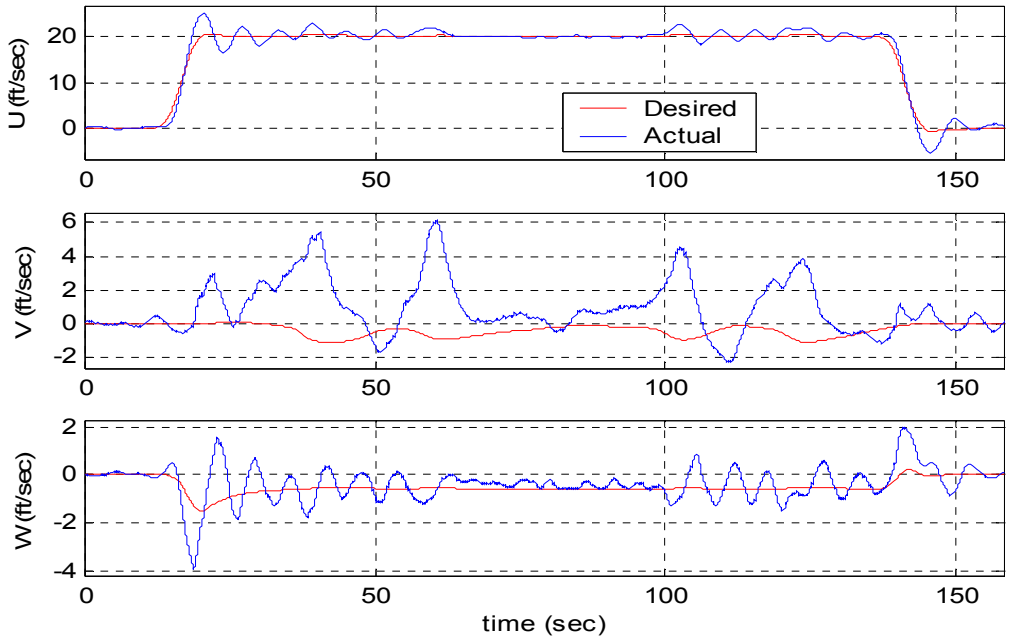


(b)

Figure 38. Software in the Loop Simulation Results for a Smooth Box at 20ft/sec: (a) Desired and Actual 2D Trajectory; (b) Desired and Actual 3D Trajectory

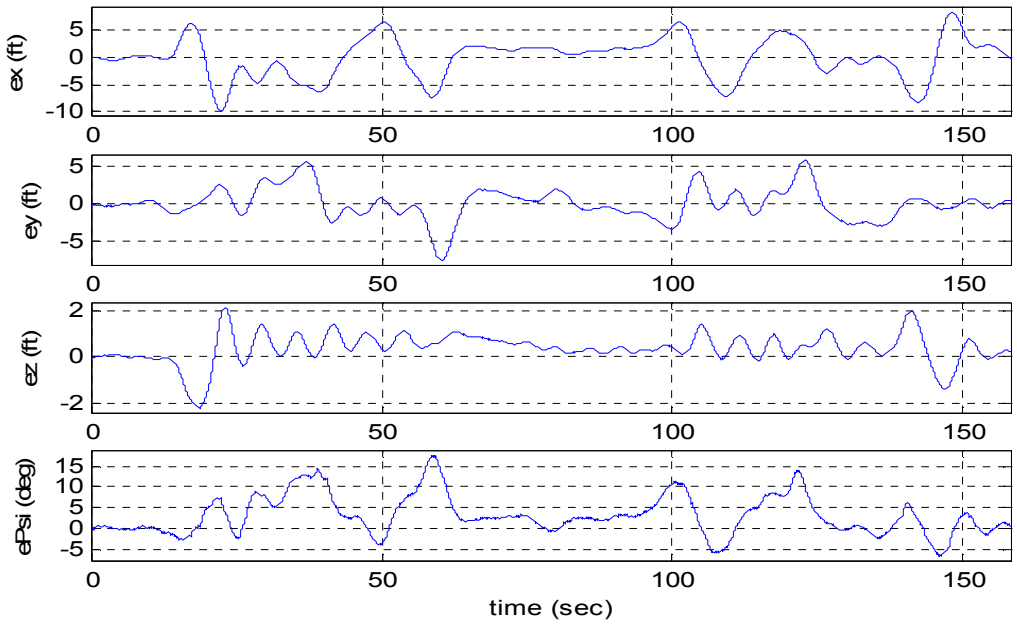


(a)

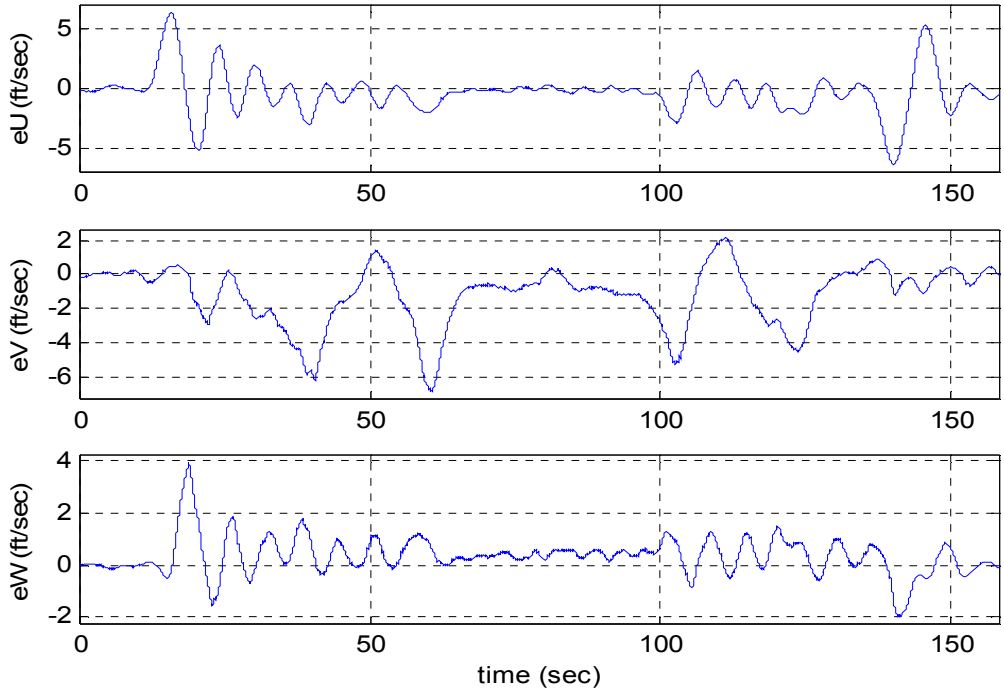


(b)

Figure 39. Software in the Loop Simulation Results for a Smooth Box at 20ft/sec: (a) Desired and Actual Position and Heading; (b) Desired and Actual Velocity in Body Frame



(a)



(b)

Figure 40. Software in the Loop Simulation Results for a Smooth Box at 20ft/sec: (a) Position and Heading Errors; (b) Velocity Errors in Body Frame

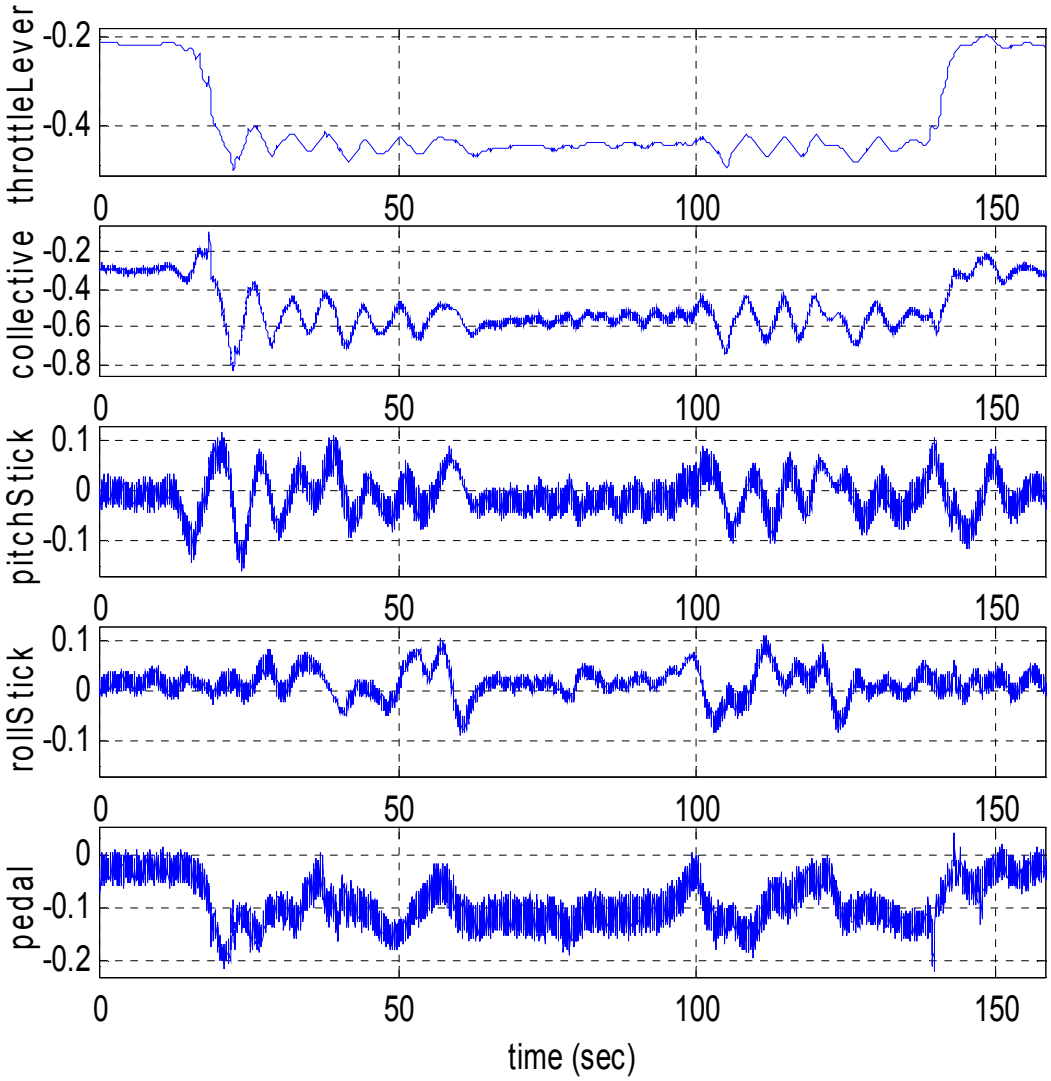


Figure 41. Software in the Loop Simulation Results for a Smooth Box at 20ft/sec: Actuator Commands

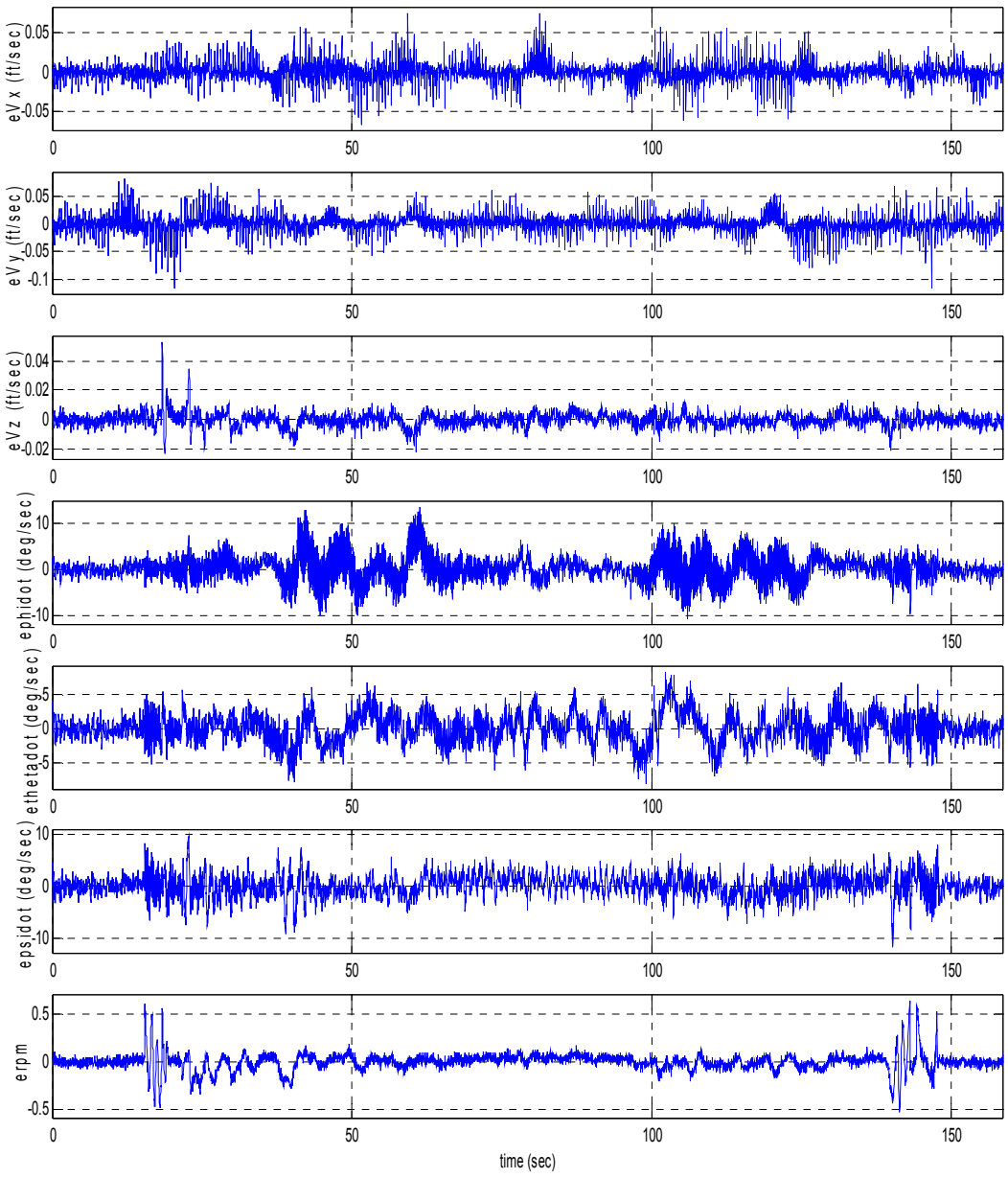


Figure 42. Software in the Loop Simulation Results for a Smooth Box at 20ft/sec: Plant Model Errors for Velocity, Euler Angle Rates and Main Rotor RPM

8.5 Flight Test Results

The same flight segments tested in software in the loop simulation were tried in actual flight. The flight test was performed on April 13, 2004 at the McKenna test range of Fort Benning, Georgia. At the moment of the test the wind was at 20mph (32ft/sec) from the West gusting to 28mph (45ft/sec). These gusty conditions were difficult to handle by the adaptive mode transition controller. Results are presented for the flight segments as follows:

- Hover pointing North. Results for this flight test are shown in Figures 43 to 47.
- Hover pointing North with heading changes to point East, West, and South. This test was interrupted when the vehicle started pointing to the wind (West), given that the vehicle was pitching too much - it is believed this was caused by the wind. Results for this flight test are shown in Figures 48 to 52.
- Hover - flight forward at 20ft/sec - hover – flight backward at 20ft/sec – hover. Results for this flight test are shown in Figures 53 to 57.
- A smooth box at 20ft/sec - move 400ft North, then move 400ft East, then move 800ft South, then move 400ft west, and finally move to initial position. This test was not completed. The wind made difficult for the controller to keep the vehicle following the desired trajectory. At some point, when the vehicle started pointing to the wind, the collective saturated and the longitudinal cyclic also saturated. Given that the adaptive mode transition controller does not have implemented any protection against saturation, it lost

control of the vehicle at that moment and the safety pilot took over to recover the vehicle.

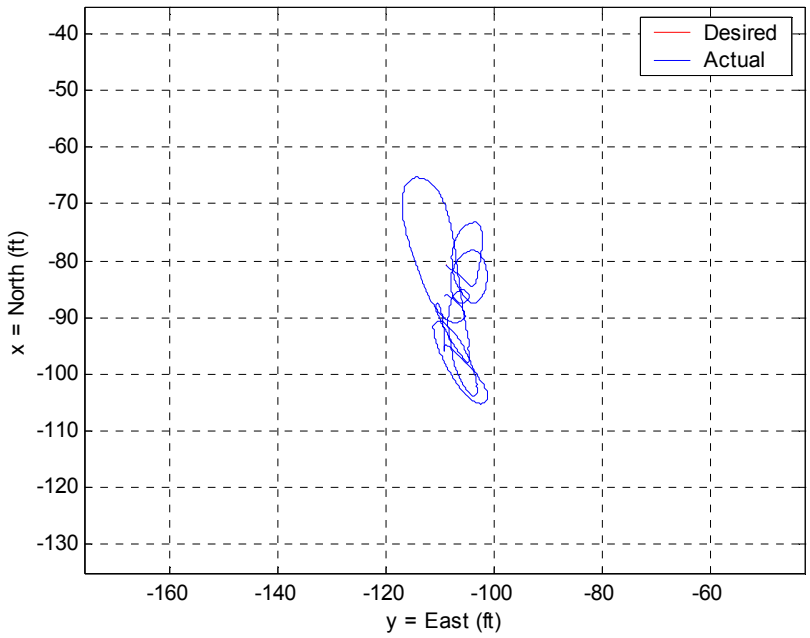
The same performance metrics used for the software in the loop simulation were computed for the flight test and the results are presented in Table 2.

Table 2. Performance Metrics for Flight Test

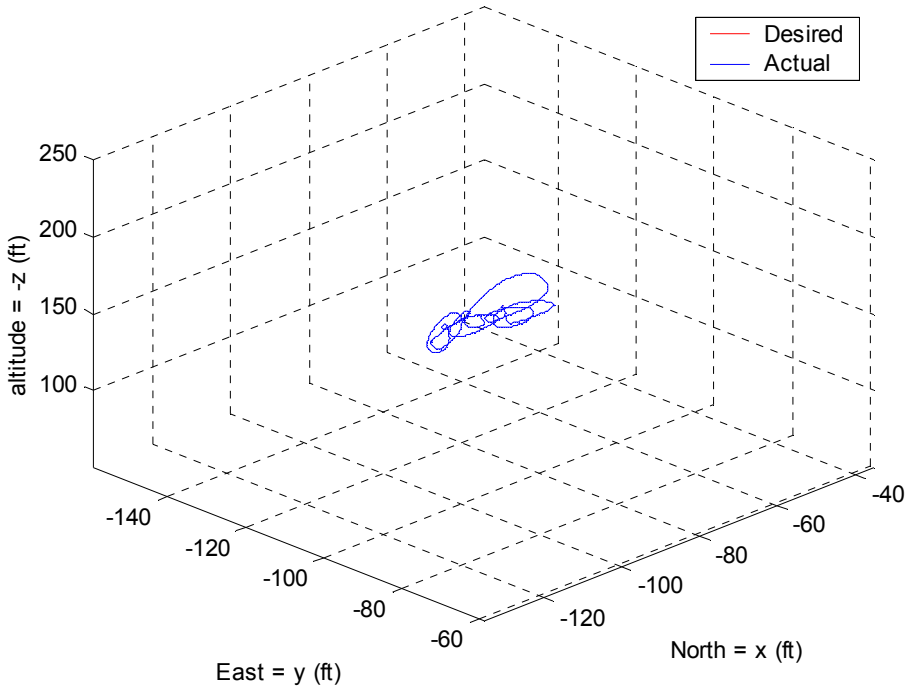
Flight Segment	Mean Magnitude of Position Error (ft)	Max Magnitude of Position Error (ft)	Mean Magnitude of Heading Error (deg)	Max Magnitude of Heading Error (deg)
Hover	10.7964	25.3326	5.5706	18.4194
Hover with heading changes	13.5464	25.3103	7.1445	32.4452
Hover - flight forward at 20ft/sec - hover – flight backward at 20ft/sec – hover	16.9902	45.4923	7.9236	25.5664

As it seen from the results, the tracking performance was not so good for this flight test. The reason for this is that the integral control was disabled almost all the time due to some logic in the Adaptive Mode Transition Control that was enabled to synchronize the integral control with the automatic trimming mechanism. When this logic is activated, the integral control is active only when the automatic trimming is enabled, that is, when the vehicle is in steady flight conditions. At the moment of the test, the gusty conditions made the vehicle move a lot so the conditions for automatic trimming were not met. However, the controller was able to keep the attitude of the vehicle within

reasonable values most of the time. The best results for this flight test were obtained for the flight segment in which the vehicle flew forward and backwards at 20ft/sec.

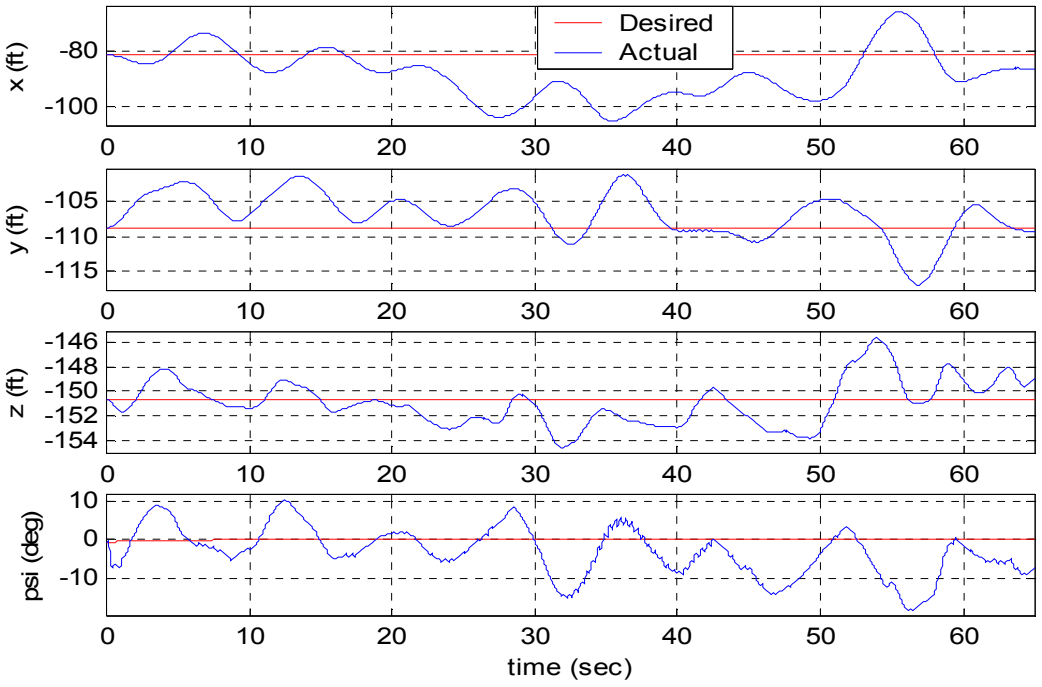


(a)

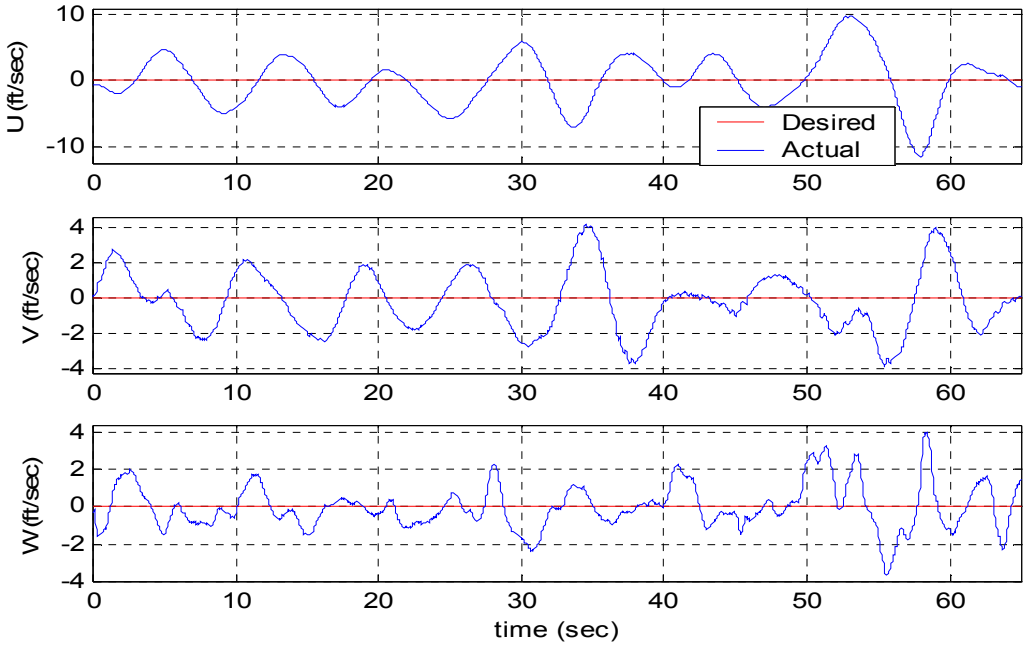


(b)

Figure 43. Flight Test Results for Hover: (a) Desired and Actual 2D Trajectory; (b) Desired and Actual 3D Trajectory

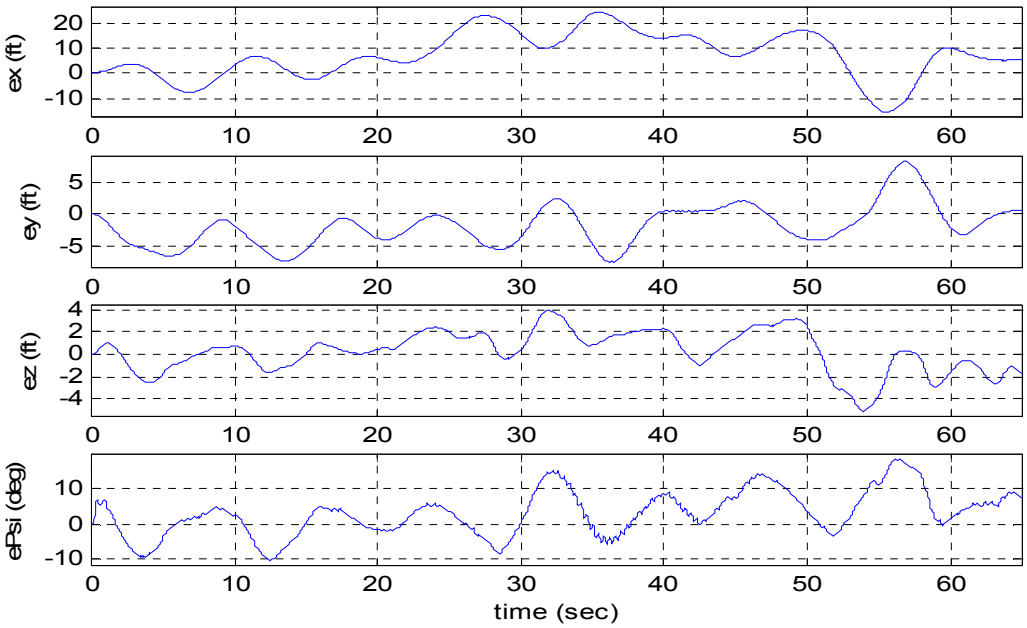


(a)

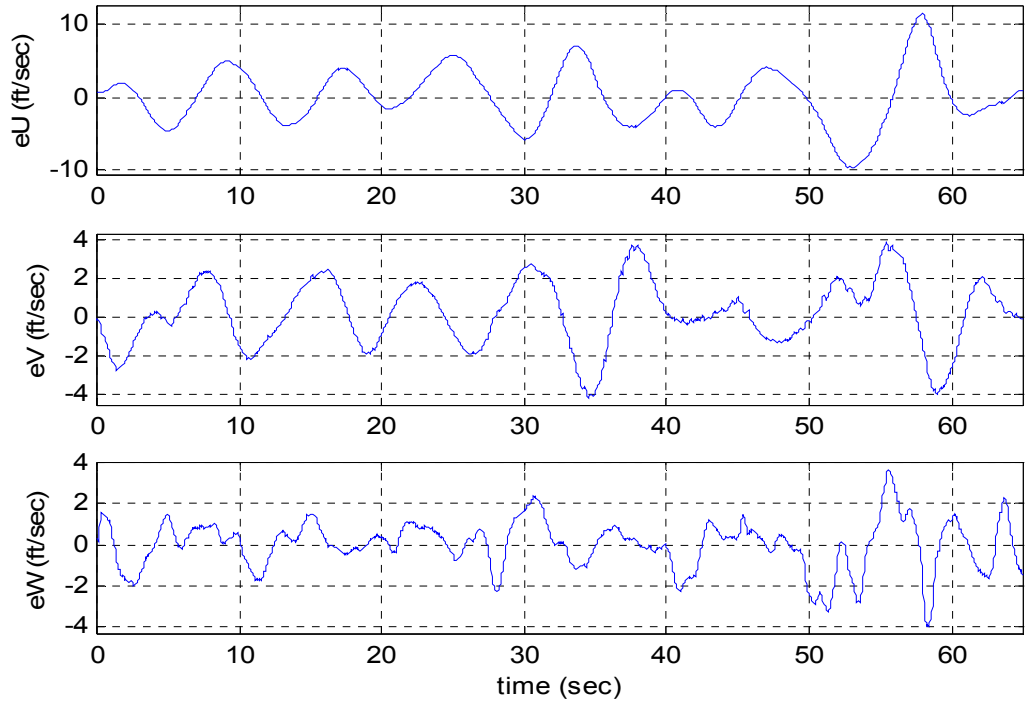


(b)

Figure 44. Flight Test Results for Hover: (a) Desired and Actual Position and Heading; (b) Desired and Actual Velocity in Body Frame



(a)



(b)

Figure 45. Flight Test Results for Hover: (a) Position and Heading Errors; (b) Velocity Errors in Body Frame

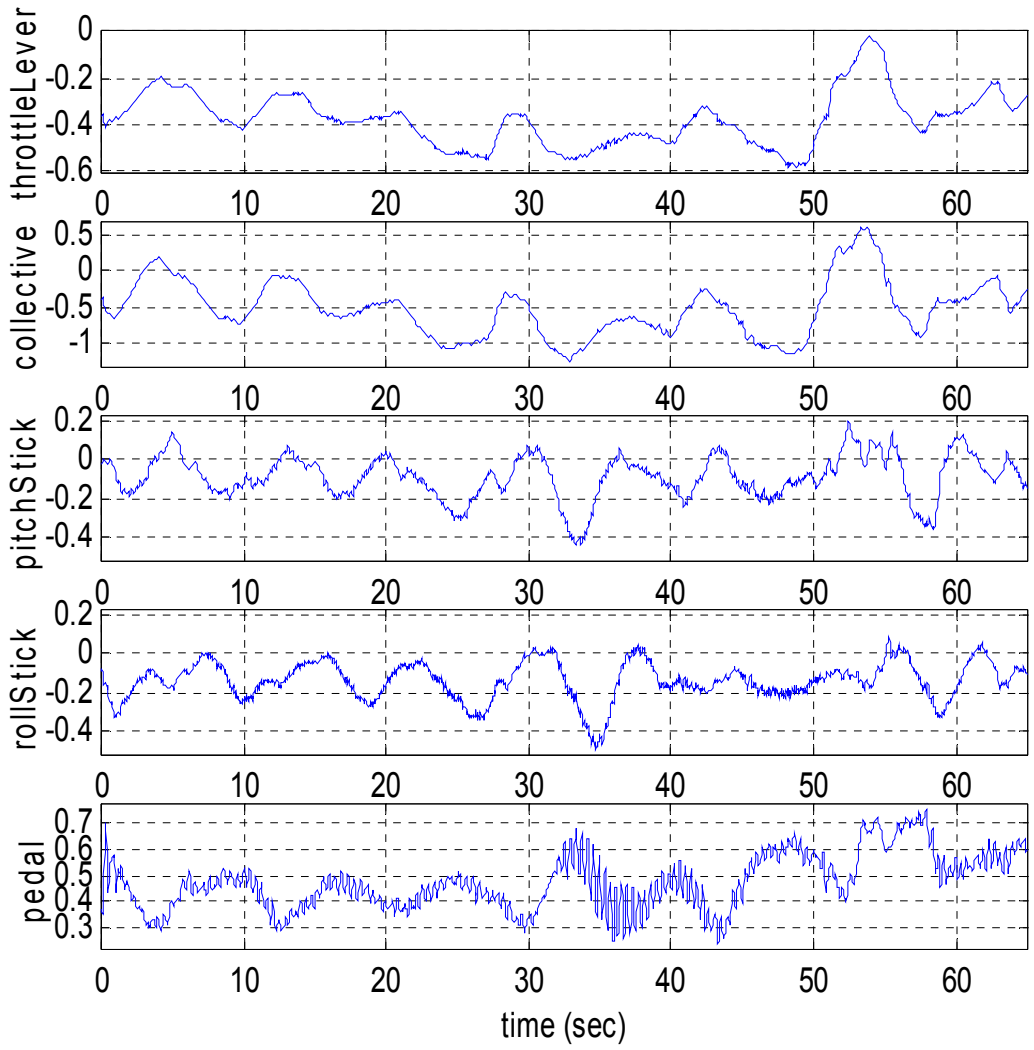


Figure 46. Flight Test Results for Hover: Actuator Commands

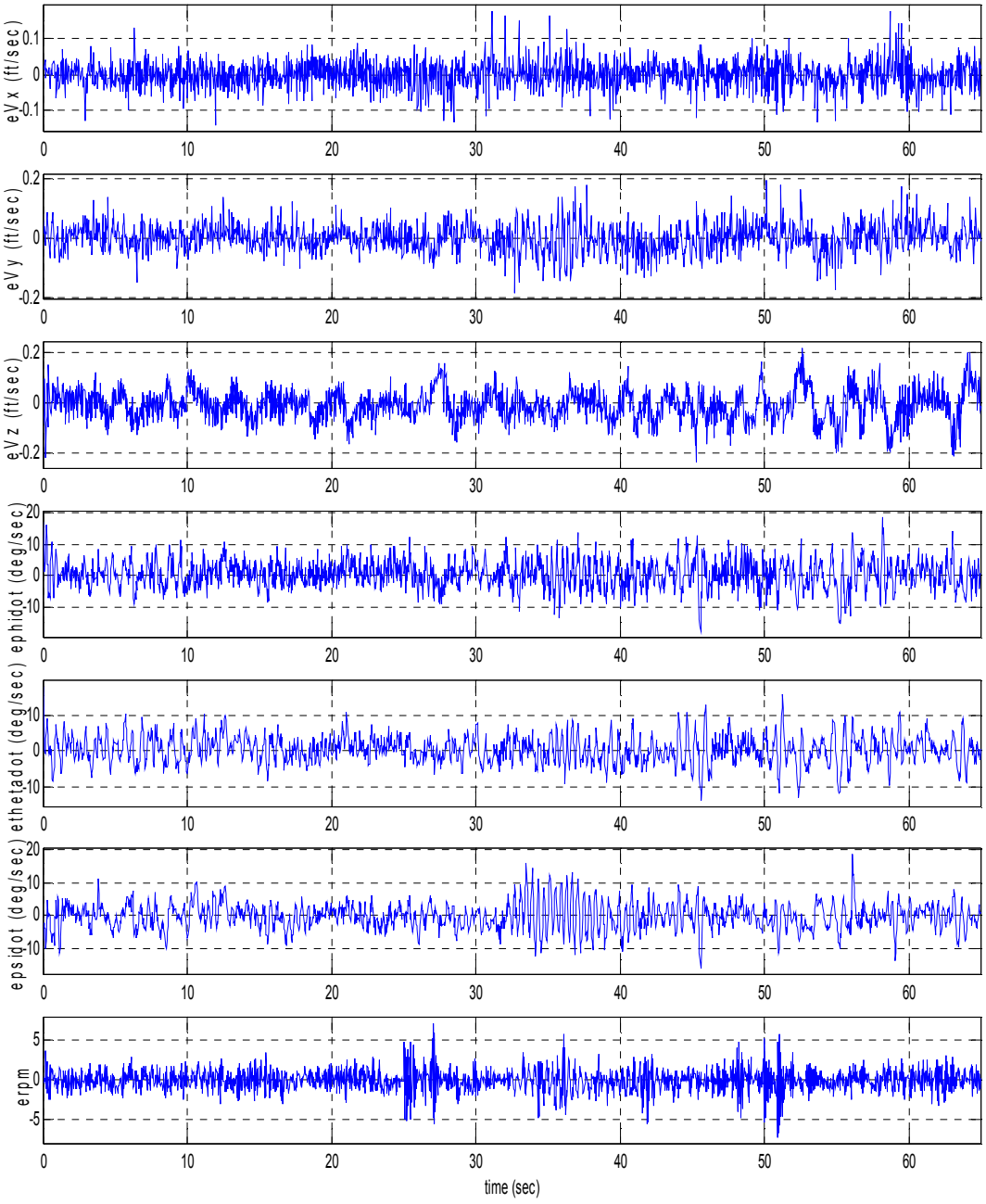
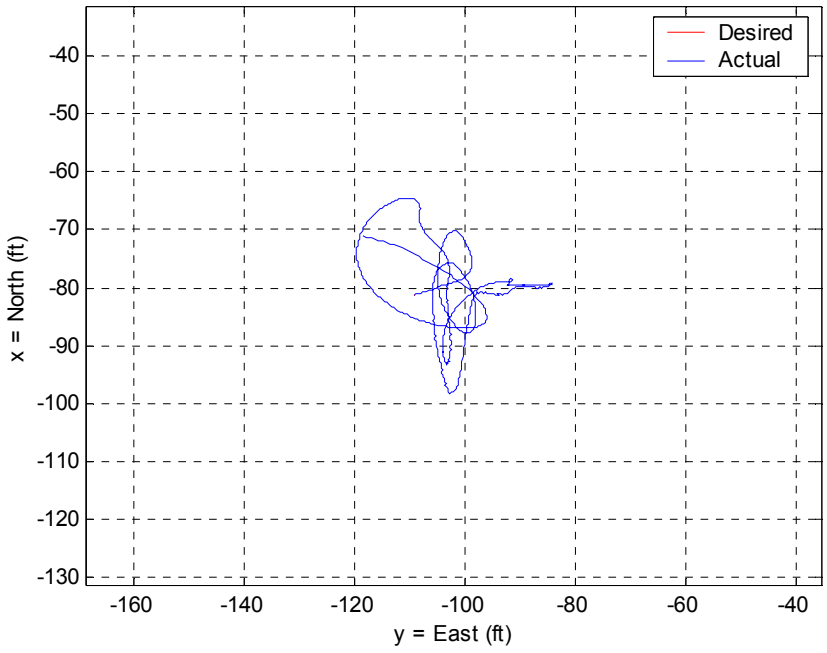
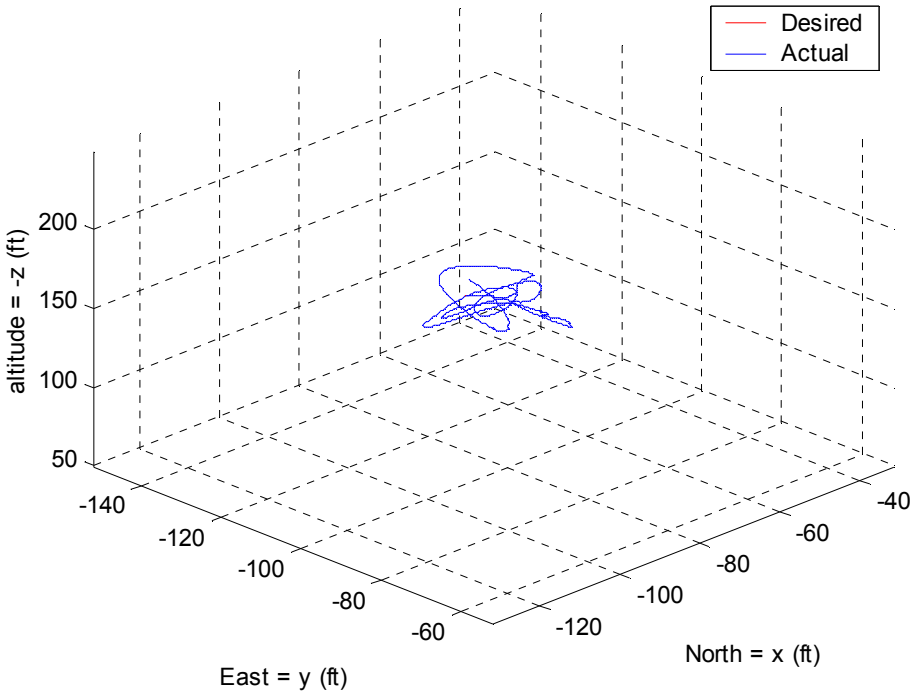


Figure 47. Flight Test Results for Hover: Plant Model Errors for Velocity, Euler Angle Rates and Main Rotor RPM

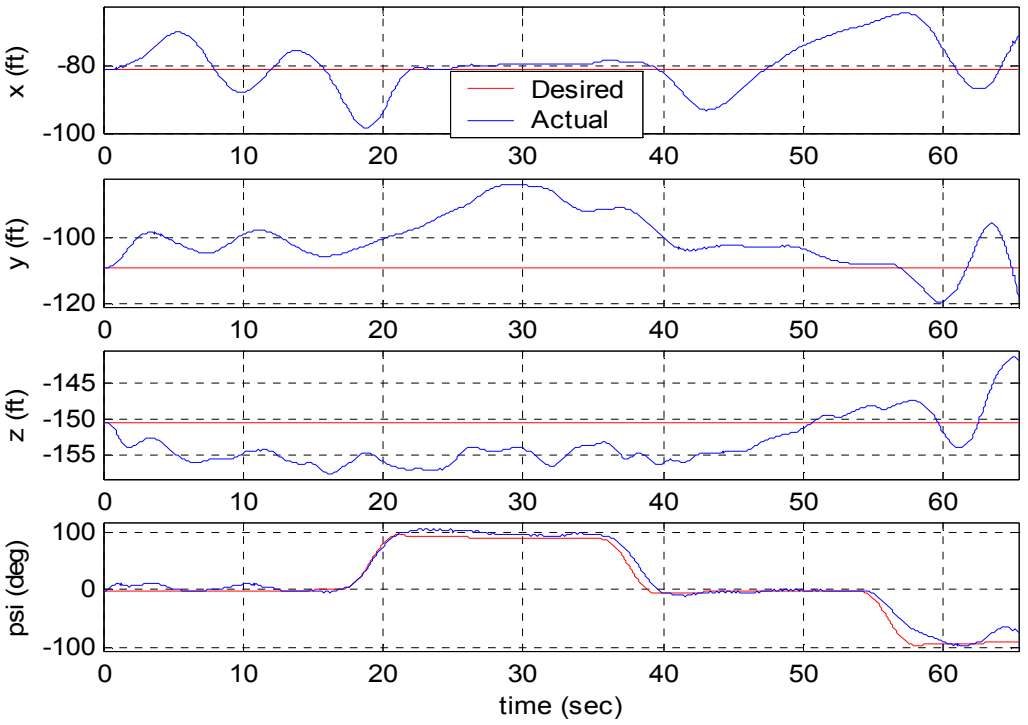


(a)

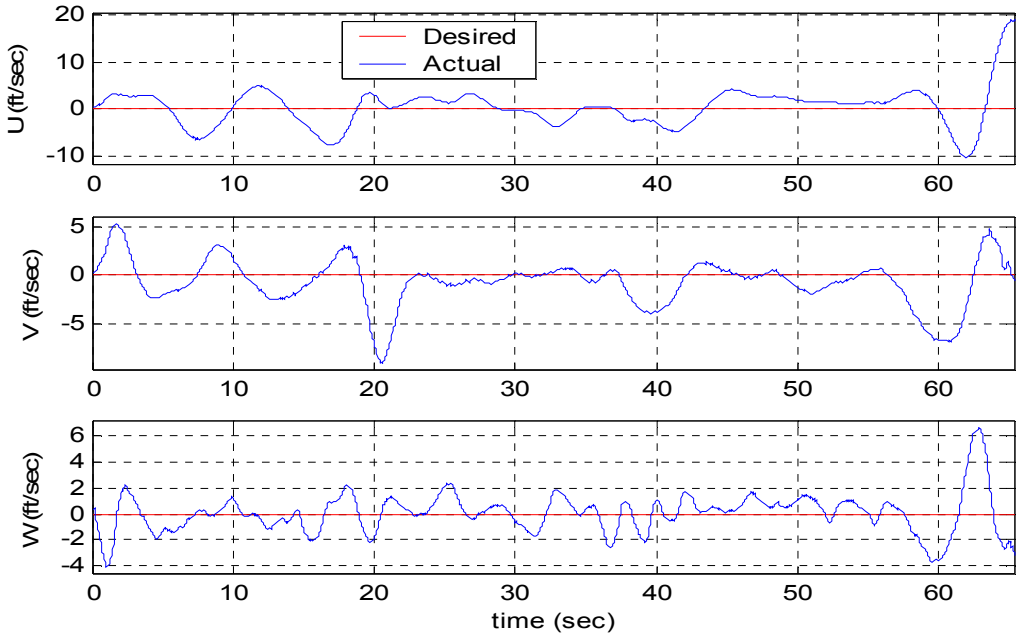


(b)

Figure 48. Flight Test Results for Hover with Heading Changes: (a) Desired and Actual 2D Trajectory; (b) Desired and Actual 3D Trajectory

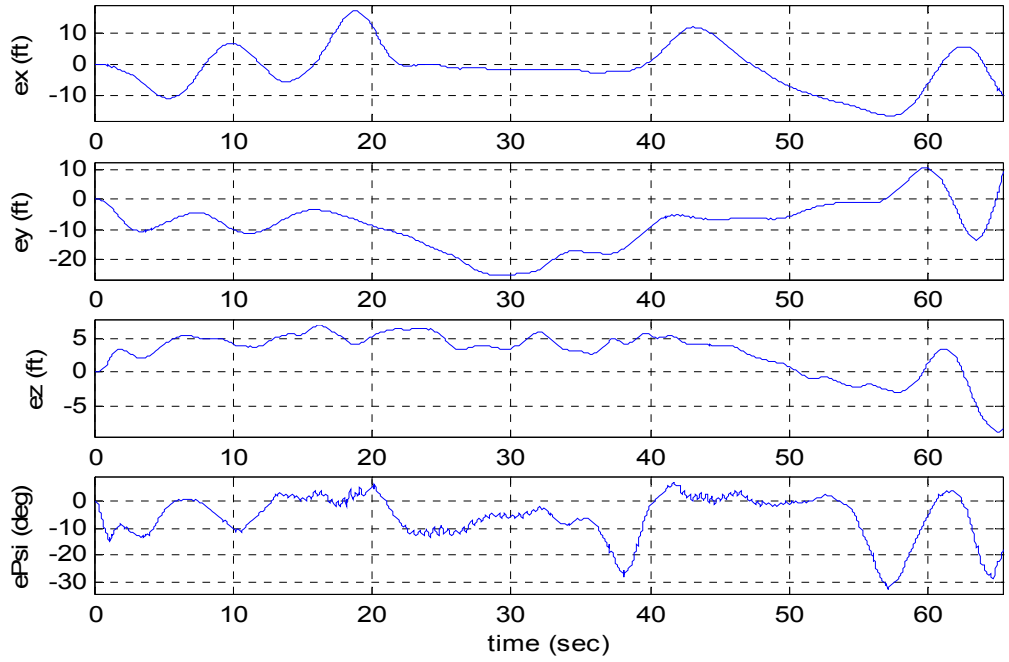


(a)

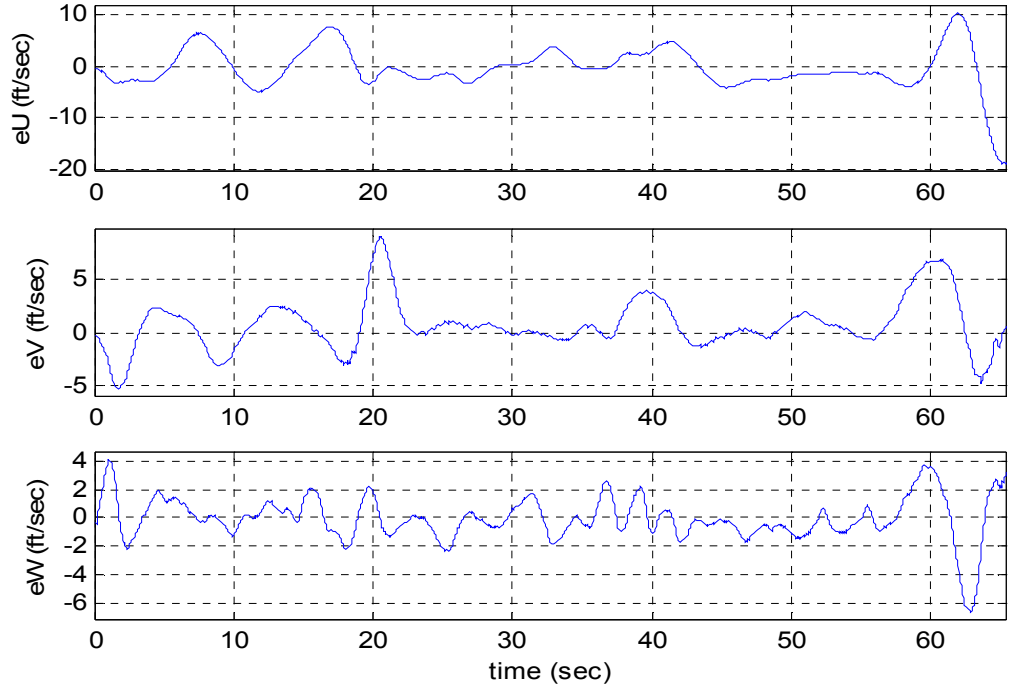


(b)

Figure 49. Flight Test Results for Hover with Heading Changes: (a) Desired and Actual Position and Heading; (b) Desired and Actual Velocity in Body Frame



(a)



(b)

Figure 50. Flight Test Results for Hover with Heading Changes: (a) Position and Heading Errors; (b) Velocity Errors in Body Frame

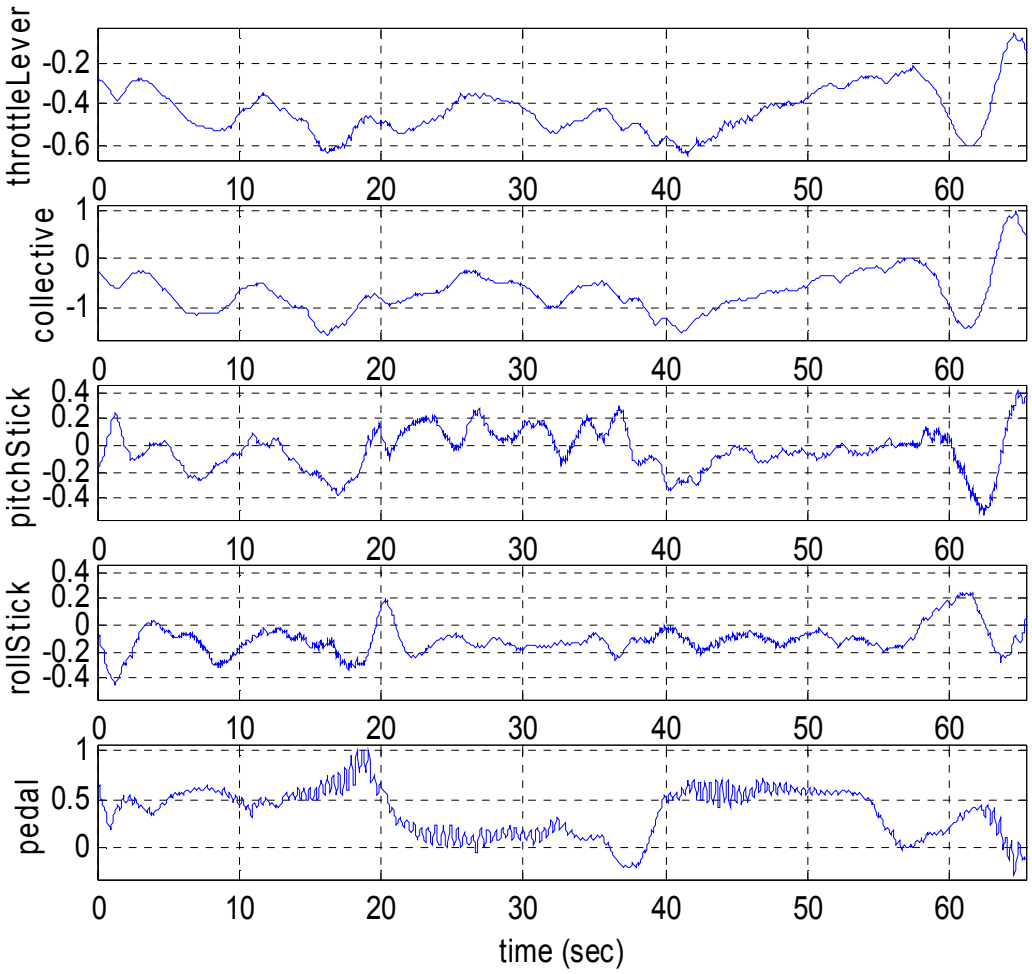


Figure 51. Flight Test Results for Hover with Heading Changes: Actuator Commands

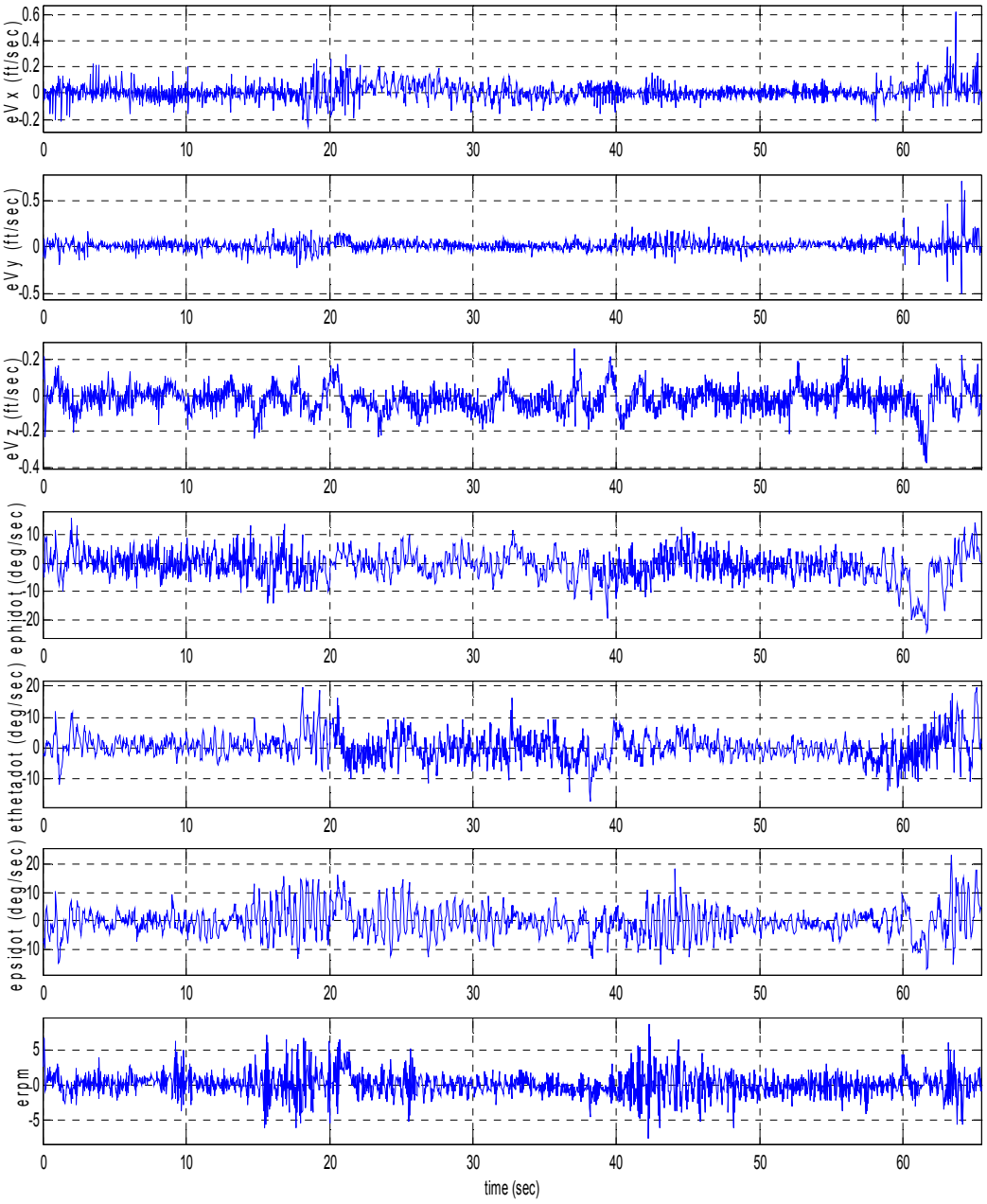
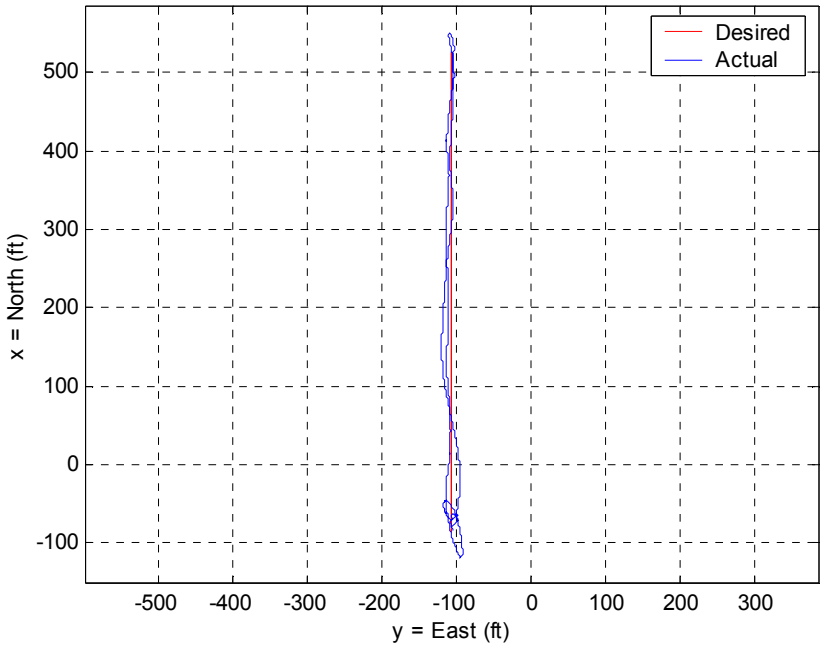
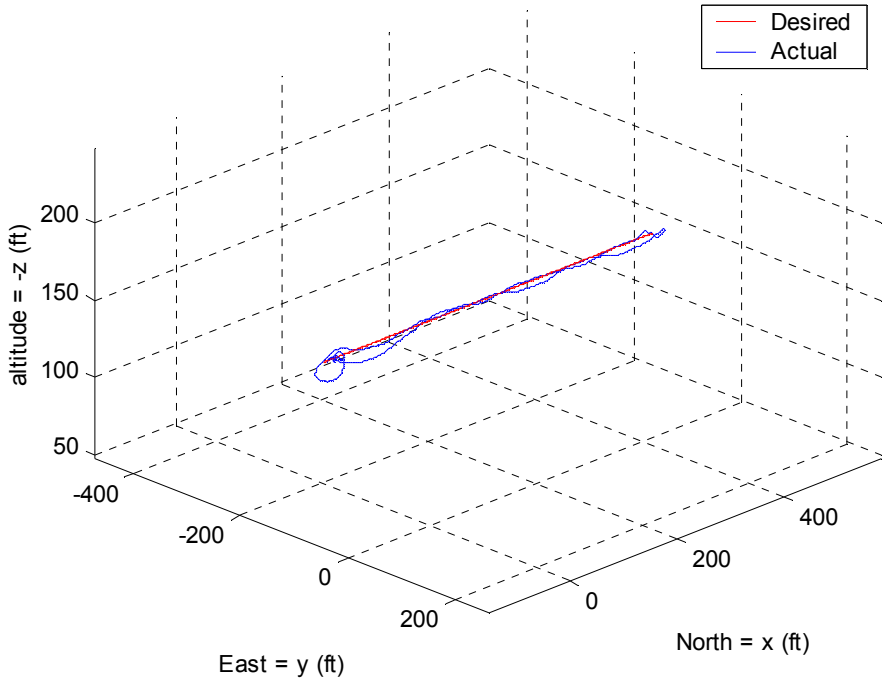


Figure 52. Flight Test Results for Hover with Heading Changes: Plant Model Errors for Velocity, Euler Angle Rates and Main Rotor RPM

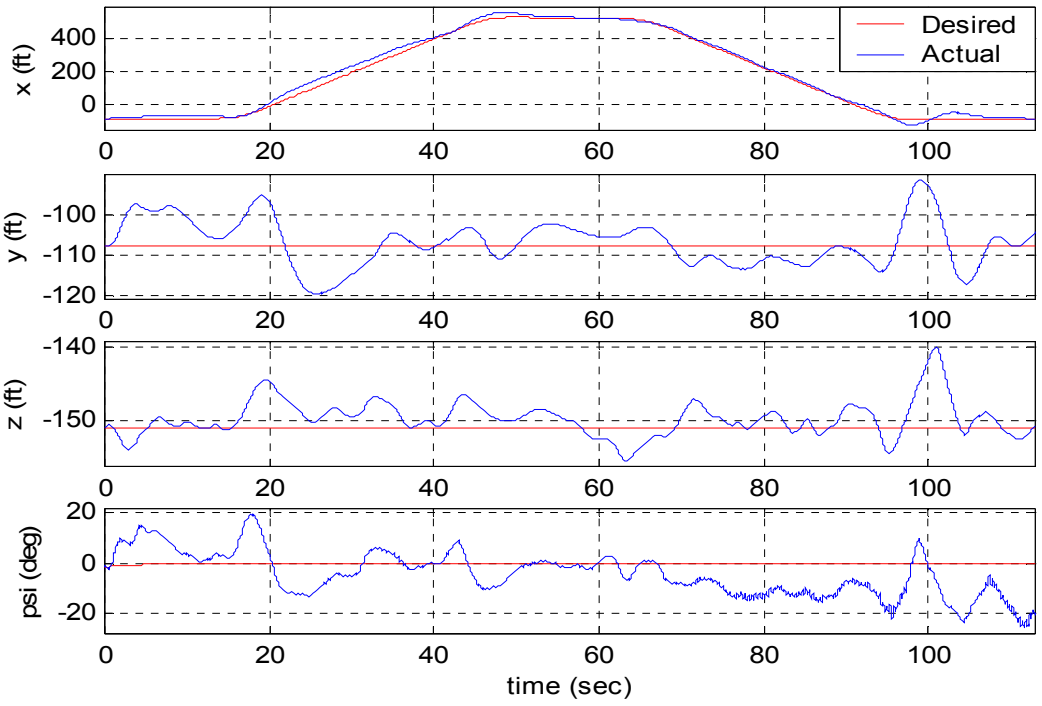


(a)

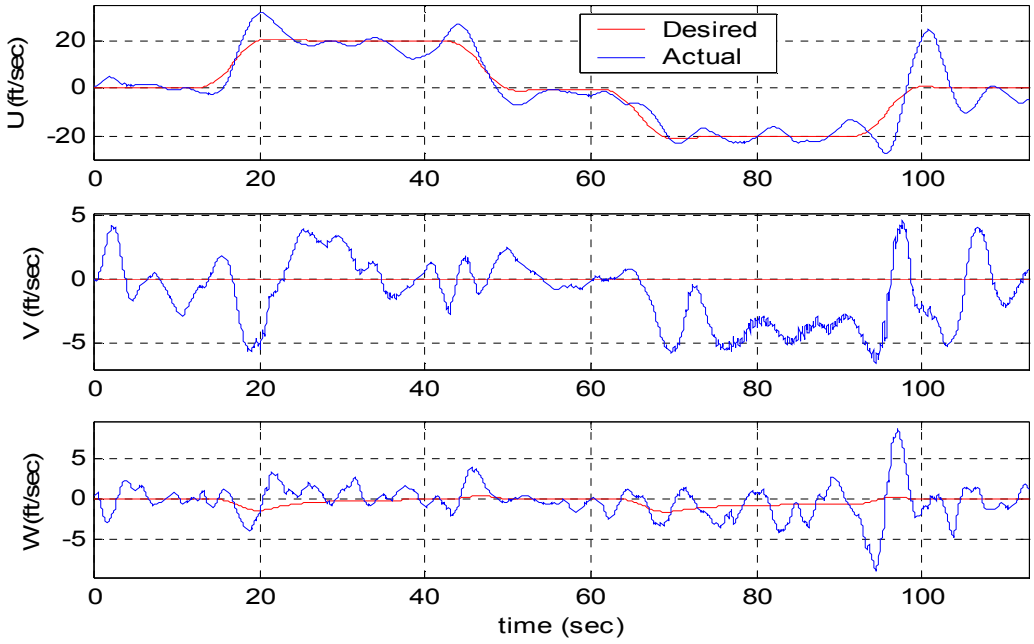


(b)

Figure 53. Flight Test Results for Hover - Flight Forward at 20ft/sec - Hover – Flight Backward at 20ft/sec – Hover: (a) Desired and Actual 2D Trajectory; (b) Desired and Actual 3D Trajectory

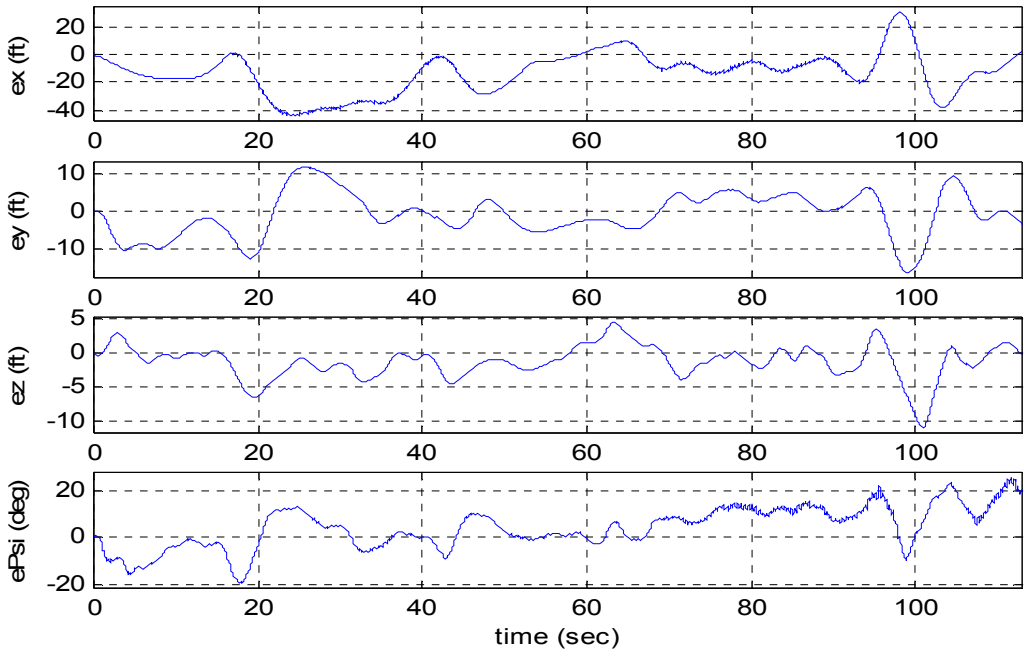


(a)

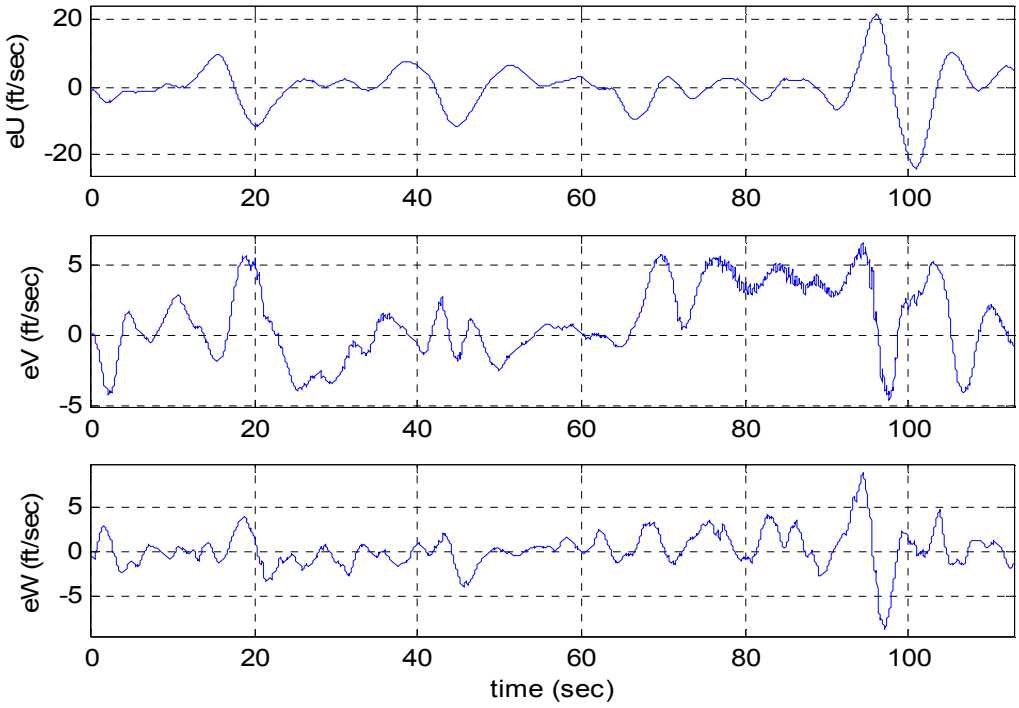


(b)

Figure 54. Flight Test Results for Hover - Flight Forward at 20ft/sec - Hover – Flight Backward at 20ft/sec – Hover: (a) Desired and Actual Position and Heading; (b) Desired and Actual Velocity in Body Frame



(a)



(b)

Figure 55. Flight Test Results for Hover - Flight Forward at 20ft/sec - Hover – Flight Backward at 20ft/sec – Hover: (a) Position and Heading Errors; (b) Velocity Errors in Body Frame

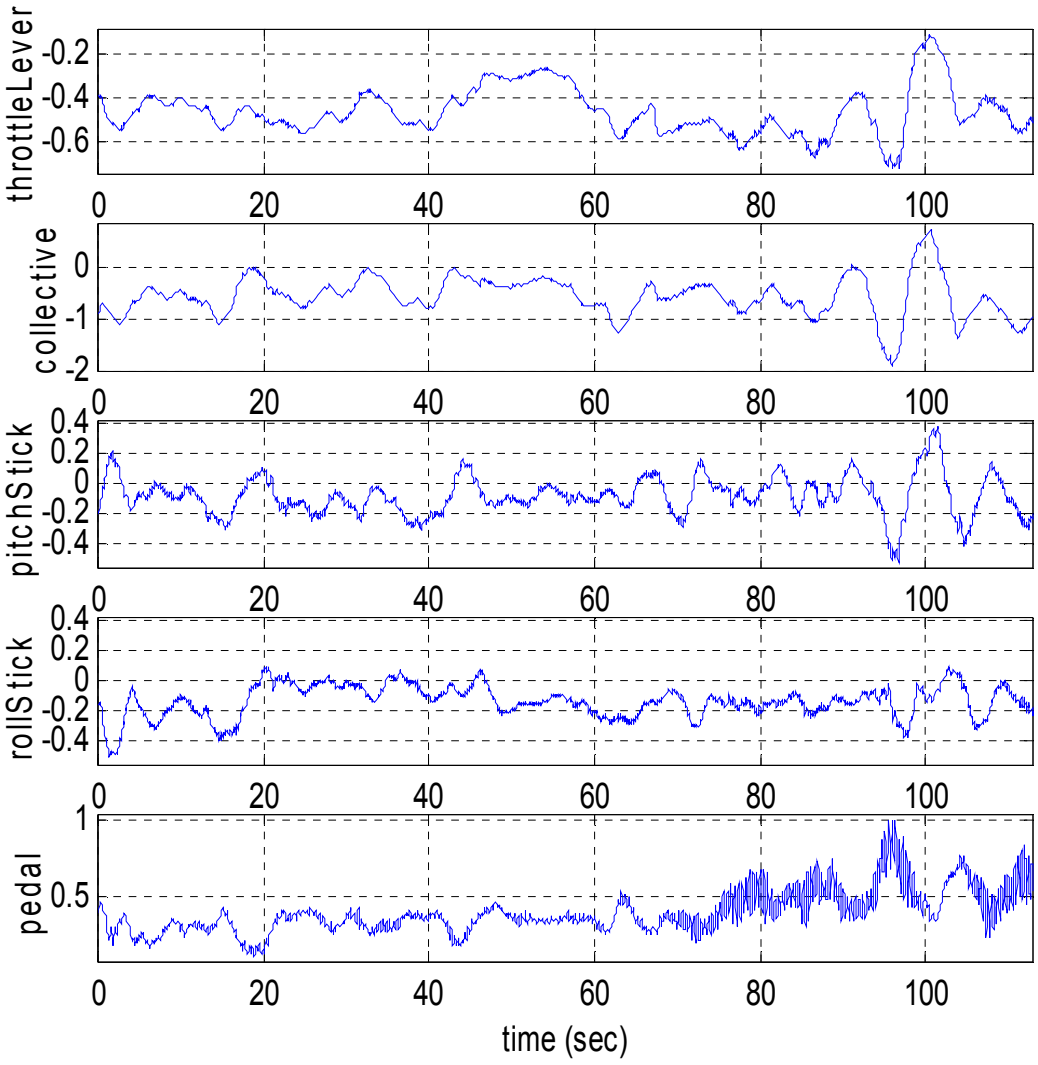


Figure 56. Flight Test Results for Hover - Flight Forward at 20ft/sec - Hover – Flight Backward at 20ft/sec – Hover: Actuator Commands

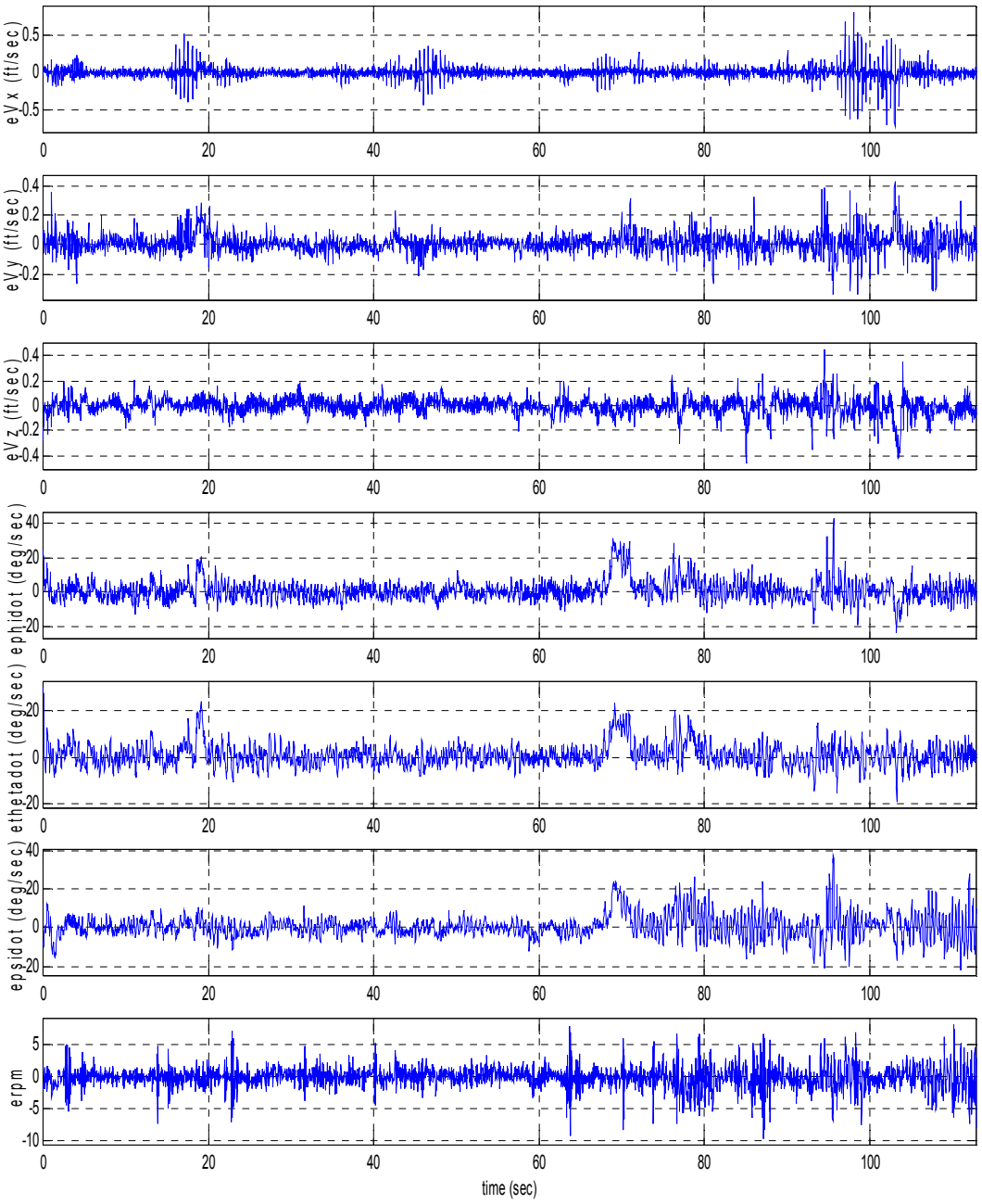


Figure 57. Flight Test Results for Hover - Flight Forward at 20ft/sec - Hover – Flight Backward at 20ft/sec – Hover: Plant Model Errors for Velocity, Euler Angle Rates and Main Rotor RPM

CHAPTER 9

CONCLUSION AND FUTURE RESEARCH

In this research, a new approach to the adaptive mode transition control problem and a hierarchical architecture to implement it were developed. The architecture was applied to the control of a helicopter UAV. A strict sequence of software in the loop simulations, hardware in the loop simulations and flight tests were used for validation and verification of the algorithms implemented.

The main contributions of this research are:

- Development of a hierarchical architecture for the implementation of the adaptive mode transition control, flexible enough to be able to accommodate future enhancements and more intelligence at the highest level of the hierarchy.
- Development of a new approach to the adaptive mode transition control problem addressing main concerns from previous accomplishments in this area.
- Exploitation of new software technologies including the OCP and the hybrid controls API to show how they enable the implementation of advanced control algorithms for UAVs.
- Implementation of the architecture and verification of its performance in software in the loop simulation, hardware in the loop simulation and through flight testing.

Some open issues related to this work that need to be addressed by further research. follow:

- Development of a theoretical framework for the adaptive mode transition control methodology. It is anticipated that the theory of linear parameter varying systems could help to address this subject.
- Based on the framework mentioned above it would be possible to improve the adaptive mode transition control methodology to guarantee the robust stability and performance of the controller.
- Enhancements in the higher level of the adaptive mode transition control architecture to enable intelligent mission planning and coordination with other agents in a multi-agent system.
- Application of the adaptive mode transition control architecture for the control of other kinds of large scale complex systems like industrial processes.
- Development of an adaptive mode transition control system with dynamic structure used to implement reconfigurable controllers. This could have applications in the area of fault tolerant control.
- Integration of the adaptive mode transition control architecture with schemes like envelope reshaping and protection to guarantee that the control does not exceed the safety operational limits of the dynamic system under control.
- Control of vehicles with changing center of gravity or mass.

PUBLICATIONS

G. Vachtsevanos, L. Tang, G. Drozeski, and L. Gutierrez, “Intelligent Control of Unmanned Aerial Vehicles for Improved Autonomy and Reliability,” submitted to 5th IFAC Symposium on Intelligent Autonomous Vehicles, Lisbon, Portugal, July 5-7 2004.

L. B. Gutierrez, G. Vachtsevanos, and B. Heck, “A Hierarchical Architecture for the mode transition control of unmanned aerial vehicles,” in *Proceedings of the AIAA Guidance, Navigation, and Control Conference and Exhibit*, Austin, Texas, August 11-14, 2003.

L. B. Gutierrez, G. Vachtsevanos, and B. Heck, "A Hierarchical/Intelligent Control Architecture for Unmanned Aerial Vehicles," in *Proceedings of the 11th Mediterranean Conference on Control and Automation MED'03*, Rhodes, Greece, June 18-20, 2003.

L. B. Gutierrez, G. Vachtsevanos, and B. Heck, “An approach to the adaptive mode transition control of unmanned aerial vehicles,” in *Proceedings of the 2003 American Control Conference, Denver*, Colorado, June 4-6, 2003.

L. B. Gutierrez, G. Vachtsevanos, and B. Heck, "A Hierarchical/Intelligent Control Architecture for Unmanned Aerial Vehicles," in Proceedings of the 21st Digital Avionics Systems Conference, (Irvine, CA), pp. 8.B.3-1/8.B.3-10, October 27-31, 2002.

REFERENCES

[1] J. S. A. Shamma, M., "Analysis of gain scheduled control for nonlinear plants," *IEEE Transactions on Automatic Control*, vol. 35, pp. 898-907, 1990.

[2] J. S. A. Shamma, M., "Gain scheduling: potential hazards and possible remedies," *IEEE Control Systems Magazine*, vol. 12, pp. 101-107, 1992.

[3] R. A. Nichols, R. T. Reichert, and W. J. Rugh, "Gain scheduling for H-infinity controllers: a flight control example," *IEEE Transactions on Control Systems Technology*, vol. 1, pp. 69-79, 1993.

[4] W. J. Rugh, "Analytical framework for gain scheduling," *IEEE Control Systems Magazine*, vol. 11, pp. 79-84, 1991.

[5] R. A. Hyde and K. Glover, "The application of scheduled H-infinity controllers to a VSTOL aircraft," *IEEE Transactions on Automatic Control*, vol. 38, pp. 1021-1039, 1993.

[6] D. J. Leith and H. E. Leithead, "On incorporating non-equilibrium plant dynamics into gain-scheduling design," in *Proc. UKACC International Conference on Control*, 1998.

[7] D. J. Stilwell and W. J. Rugh, "Interpolation of observer state feedback controllers for gain scheduling," *IEEE Transactions on Automatic Control*, vol. 44, pp. 1225-1229, 1999.

[8] P. Apkarian, P. Gahinet, and G. Becker, "Self-scheduled H-infinity Control of Linear Parameter-varying Systems: a Design Example," *Automatica*, vol. 31, pp. 1251-1261, 1995.

[9] P. Apkarian and P. Gahinet, "A convex characterization of gain-scheduled H-infinity controllers," *IEEE Transactions on Automatic Control*, vol. 40, pp. 853-864, 1995.

[10] P. Apkarian and R. J. Adams, "Advanced gain-scheduling techniques for uncertain systems," *IEEE Transactions on Control Systems Technology*, vol. 6, pp. 21-32, 1998.

[11] A. Packard, K. Zhou, P. Pandey, and G. Becker, "A collection of robust control problems leading to LMIs," in *Proc. 30th IEEE Conference on Decision and Control*, 1991.

[12] S. Boyd, L. E. Ghaoui, E. Feron, and V. Balakrishnan, *Linear matrix inequalities in systems and control theory*, vol. 15. Philadelphia: SIAM, 1994.

[13] P. Gahinet, A. Nemirovski, A. J. Laub, and M. Chilali, *LMI Control Toolbox*: The MathWorks, Inc., 1995.

[14] P. Bergsten, M. Persson, and B. Iliev, "Fuzzy gain scheduling for flight control," in *Proc. 26th Annual Conference of the IEEE Industrial Electronics Society*, 2000.

[15] P. Korba, R. Babuska, H. B. Verbruggen, and P. M. Frank, "Fuzzy gain scheduling: controller and observer design based on lyapunov method and convex optimization," *IEEE Transactions on Fuzzy Systems*, vol. 11, pp. 285-298, 2003.

[16] V. Gavrilets, I. Martinos, B. Mettler, and E. Feron, "Flight test and simulation results for an autonomous aerobatic helicopter," in *Proc. 21st Digital Avionics Systems Conference*, 2002.

[17] V. Gavrilets, I. Martinos, B. Mettler, and E. Feron, "Control Logic for Automated Aerobatic Flight of Miniature Helicopter," in *Proc. AIAA Guidance, Navigation and Control Conference*, 2002.

[18] B. Mettler, V. Gavrilets, E. Feron, and T. Kanade, "Dynamic Compensation for High-Bandwidth Control of Small-Scale Helicopter," in *Proc. American Helicopter Society Test and Evaluation Technical Specialists Meeting*, 2002.

[19] R. A. DeCarlo, S. H. Zak, and G. P. Matthews, "Variable structure control of nonlinear multivariable systems: a tutorial," *Proceedings of the IEEE*, vol. 76, pp. 212-232, 1988.

[20] K. D. Young, V. I. Utkin, and U. Ozguner, "A control engineer's guide to sliding mode control," *IEEE Transactions on Control Systems Technology*, vol. 7, pp. 328-342, 1999.

[21] J. Guldner and V. I. Utkin, "The chattering problem in sliding mode systems," in *Proc. Fourteenth International Symposium on Mathematical Theory of Networks and Systems*, Perpignan, France, 2000.

[22] W.-S. Lin and C.-S. Chen, "Robust adaptive sliding mode control using fuzzy modelling for a class of uncertain MIMO nonlinear systems," in *IEE Proceedings-Control Theory and Applications*, vol. 149, 2002, pp. 193-201.

[23] C. W. Tao, M.-L. Chan, and T.-T. Lee, "Adaptive fuzzy sliding mode controller for linear systems with mismatched time-varying uncertainties," *IEEE Transactions on Systems, Man and Cybernetics, Part B*, vol. 33, pp. 283-294, 2003.

[24] K.-K. D. Young, "Design of variable structure model-following control systems," *IEEE Transactions on Automatic Control*, vol. AC-23, pp. 1079-1085, 1978.

[25] A. J. Calise and F. Kramer, "A variable structure approach to robust control of VTOL aircraft," in *Proc. 1982 American Control Conference*, Arlington, VA, USA, 1982.

[26] K.-K. D. Young, "Controller design for a manipulator using theory of variable structure systems," *IEEE Transactions on Systems, Man and Cybernetics*, vol. SMC-8, pp. 101-109, 1978.

[27] J. J. Slotine and S. S. Sastry, "Tracking control of non-linear systems using sliding surfaces, with application to robot manipulators," *International Journal of Control*, vol. 38, pp. 465-492, 1983.

[28] S. N. Singh, M. Pachter, P. Chandler, S. Banda, S. Rasmussen, and C. Schumacher, "Input-output invertibility and sliding mode control for close formation flying of multiple UAVs," *International Journal of Robust and Nonlinear Control*, vol. 10, pp. 779-797, 2000.

[29] G. A. Dumont and M. Huzmezan, "Concepts, methods and techniques in adaptive control," in *Proc. 2002 American Control Conference*, 2002.

[30] N. M. Filatov and H. Unbehauen, "Survey of adaptive dual control methods," *IEE Proceedings-Control Theory and Applications*, vol. 147, pp. 118-128, 2000.

[31] B. Wittenmark, "Adaptive Dual Control Methods: An Overview," in *Proc. 5th IFAC Symposium on Adaptive Systems in Control and Signal Processing*, Budapest, Hungary, 1995.

[32] P. R. Wahi, R.; Chowdhury, F.N., "A survey of recent work in adaptive flight control," in *Proc. 33rd Southeastern Symposium on System Theory*, 2001.

[33] A. J. Calise and R. T. Rysdyk, "Nonlinear adaptive flight control using neural networks," *IEEE Control Systems Magazine*, vol. 18, pp. 14-25, 1998.

[34] E. N. Johnson, A. J. Calise, H. A. El-Shirbiny, and R. T. Rysdyk, "Feedback Linearization with Neural Network Augmentation applied to X-33 Attitude Control," in *Proc. AIAA Guidance, Navigation and Control Conference*, Denver, CO, 2000.

[35] R. T. Rysdyk and A. J. Calise, "Adaptive nonlinear control for tiltrotor aircraft," in *Proc. 1998 IEEE International Conference on Control Applications*, 1998.

[36] A. J. Calise, S. Lee, and M. Sharma, "Direct Adaptive Reconfigurable Control of a Tailless Fighter Aircraft," in *Proc. AIAA Guidance, Navigation and Control Conference*, Boston, MA, 1998.

[37] A. Verma, K. Subbarao, and J. L. Junkins, "A novel trajectory tracking methodology using structured adaptive model inversion for uninhabited aerial vehicles," in *Proc. 2000 American Control Conference*, Chicago, IL, 2000.

[38] H. Nakanishi and K. Inoue, "Development of autonomous flight control systems for unmanned helicopter by use of neural networks," in *Proc. 2002 International Joint Conference on Neural Networks*, 2002.

[39] S. K. Kannan and E. N. Johnson, "Adaptive trajectory based control for autonomous helicopters," in *Proc. 21st Digital Avionics Systems Conference, 2002*, 2002.

[40] S. K. Kannan and E. N. Johnson, "Adaptive Flight Control for an Autonomous Unmanned Helicopter," in *Proc. AIAA Guidance, Navigation and Control Conference*, 2002.

[41] E. N. Johnson and A. J. Calise, "Pseudo-Control Hedging: A New Method for Adaptive Control," in *Proc. Workshop on Advances in Guidance and Control Technology*, Arsenal, Alabama, 2000.

[42] E. N. Johnson and A. J. Calise, "Neural network adaptive control of systems with input saturation," in *Proc. 2001 American Control Conference*, 2001.

[43] K. S. Narendra and J. Balakrishnan, "Adaptive control using multiple models," *IEEE Transactions on Automatic Control*, vol. 42, pp. 171-187, 1997.

[44] J. D. Boskovic and R. K. Mehra, "Stable multiple model adaptive flight control for accommodation of a large class of control effector failures," in *Proc. 1999 American Control Conference*, 1999.

[45] F. Rufus, G. Vachtsevanos, and B. Heck, "Real-time adaptation of mode transition controllers," *Journal of Guidance Control and Dynamics*, vol. 25, pp. 167-175, 2002.

[46] F. Rufus, G. Vachtsevanos, and B. Heck, "Adaptive Mode Transition Control of Nonlinear Systems Using Fuzzy Neural Networks," in *Proc. 8th IEEE Mediterranean Conference on Control and Automation*, Patras, Greece, 2000.

[47] F. Rufus, B. Heck, and G. Vachtsevanos, "Software-enabled adaptive mode transition control for autonomous unmanned vehicles," in *Proc. 19th Digital Avionics Systems Conferences*, 2000.

[48] F. Rufus, "Intelligent approaches to mode transition control," in *Electrical and Computer Engineering*. Atlanta: Georgia Institute of Technology, 2000.

[49] J. B. Rawlings, "Tutorial overview of model predictive control," *IEEE Control Systems Magazine*, vol. 20, pp. 38-52, 2000.

[50] D. Q. M. Mayne, H., "Receding horizon control of nonlinear systems," *IEEE Transactions on Automatic Control*, vol. 35, pp. 814-824, 1990.

[51] Y. L. Huang, H. H. Lou, J. P. Gong, and T. F. Edgar, "Fuzzy model predictive control," *IEEE Transactions on Fuzzy Systems*, vol. 8, pp. 665-678, 2000.

[52] S. Piche, B. Sayyar-Rodsari, D. Johnson, and M. Gerules, "Nonlinear model predictive control using neural networks," *IEEE Control Systems Magazine*, vol. 20, pp. 53-62, 2000.

[53] H. J. Kim, D. H. Shim, and S. Sastry, "Nonlinear model predictive tracking control for rotorcraft-based unmanned aerial vehicles," in *Proc. 2002 American Control Conference*, 2002.

[54] R. Giroux, R. Gourdeau, M. Pelletier, and R. Hurteau, "A linear quadratic tracker for VTOL-UAV trajectory control with multivariable constraints," in *Proc. AIAA Guidance, Navigation, and Control Conference and Exhibit*, Denver, CO, 2000.

[55] J. S. Jang and C. J. Tomlin, "Autopilot design for the Stanford DragonFly UAV - Validation through hardware-in-the-loop simulation," in *Proc. AIAA Guidance, Navigation, and Control Conference and Exhibit*, Montreal, Canada, 2001.

[56] B. S. Heck, L. M. Wills, and G. J. Vachtsevanos, "Software technology for implementing reusable, distributed control systems," *IEEE Control Systems Magazine*, vol. 23, pp. 21-35, 2003.

[57] B. S. Heck, L. M. Wills, and G. J. Vachtsevanos, "Software enabled control: background and motivation," in *Proc. 2001 American Control Conference*, 2001.

[58] D. P. Schrage and G. Vachtsevanos, "Software-enabled control for intelligent UAVs," in *Proc. 1999 IEEE International Symposium on Computer Aided Control System Design*, 1999.

[59] L. Wills, S. Kannan, S. Sander, M. Guler, B. Heck, J. V. R. Prasad, D. Schrage, and G. Vachtsevanos, "An open platform for reconfigurable control," *IEEE Control Systems Magazine*, vol. 21, pp. 49-64, 2001.

[60] L. Wills, S. Kannan, B. Heck, G. Vachtsevanos, C. Restrepo, S. Sander, D. Schrage, and J. V. R. Prasad, "An open software infrastructure for reconfigurable control systems," in *Proc. 2000 American Control Conference*, 2000.

[61] L. Wills, S. Sander, S. Kannan, A. Kahn, J. V. R. Prasad, and D. Schrage, "An open control platform for reconfigurable, distributed, hierarchical control systems," in *Proc. 19th Digital Avionics Systems Conferences*, 2000.

[62] J. L. Paunicka, B. R. Mendel, and D. E. Corman, "The OCP - an open middleware solution for embedded systems," in *Proc. 2001 American Control Conference*, 2001.

[63] J. L. Paunicka, D. E. Corman, and B. R. Mendel, "A CORBA-based middleware solution for UAVs," in *Proc. Fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. ISORC - 2001*, 2001.

[64] E. Frazzoli, M. A. Dahleh, and E. Feron, "Robust hybrid control for autonomous vehicle motion planning," in *Proc. 39th IEEE Conference on Decision and Control*, 2000.

[65] J. Theocharis and G. Vachtsevanos, "Adaptive fuzzy neural networks as identifiers of discrete-time nonlinear dynamic systems," *Journal of Intelligent and Robotic Systems: Theory and Applications*, vol. 17, pp. 119-168, 1996.

[66] G. F. Franklin, J. D. Powell, and M. Workman, *Digital Control of Dynamic Systems*, 3rd ed. Menlo Park, CA: Addison Wesley, 1998.

[67] L. B. Gutierrez, G. Vachtsevanos, and B. Heck, "A Hierarchical Architecture for the mode transition control of unmanned aerial vehicles," in *Proc. AIAA Guidance, Navigation, and Control Conference and Exhibit*, Austin, Texas, 2003.

[68] J. Bryson, Arthur E., *Dynamic Optimization*. Menlo Park, CA: Addison Wesley, 1999.

[69] M. Guler, S. Clements, N. Kejriwal, L. Wills, B. Heck, and G. Vachtsevanos, "Rapid prototyping of transition management code for reconfigurable control systems," in *Proc. 13th IEEE International Workshop on Rapid System Prototyping*, 2002.

[70] M. Guler, S. Clements, L. Wills, B. Heck, and G. Vachtsevanos, "Generic transition management for reconfigurable hybrid control systems," in *Proc. 2001 American Control Conference*, 2001.

[71] M. Guler, S. Clements, L. M. Wills, B. S. Heck, and G. J. Vachtsevanos, "Transition management for reconfigurable hybrid control systems," *IEEE Control Systems Magazine*, vol. 23, pp. 36-49, 2003.

[72] E. N. Johnson and S. Mishra, "Flight Simulation for the Development of an Experimental UAV," in *Proc. AIAA Modeling and Simulation Technology Conference*, Monterey, CA, 2002.