Final Report
13 July 1999 - 30 November 2004
Contract Number F33615-99-C-1497

An Integrated Multi-Layer Approach to Software Enabled Control: Mission Planning to Vehicle Control

Tamás Keviczky, Ryan Ingvalson, Héctor Rotstein, Oreste Riccardo Natale and Gary J. Balas
University of Minnesota
Minneapolis, MN 55455

Andrew Packard
University of California
Berkeley, CA 94720

DARPA Software Enabled Control Program

Dr. John Bay, Program Manager
Information Exploitation Office
Defense Advanced Research Projects Agency
3701 N. Fairfax Drive
Arlington, Virginia
22203-1714

Dale Van Cleave, Technical Monitor
Information Directorate
Air Force Research Laboratory
Air Force Material Command
Wright-Patterson Air Force Base, Ohio
45433-7334

November 30, 2004

Contributing authors

Tamás Keviczky	University of Minnesota
Ryan Ingvalson	University of Minnesota
Héctor Rotstein	On Sabbatical Leave from Rafael, Israel and the
	Dept. of Electrical Engineering, Technion, Israel
Andrew Packard	University of California, Berkeley
Oreste Riccardo Natale	Università degli Studi del Sannio in Benevento
Gary J. Balas	University of Minnesota

Contents

Chapter 1

Introduction

1.1 DARPA SEC program

The “Software-Enabled Control” (SEC) program sponsored by the Defense Advanced Research Projects Agency (DARPA) of the United States represents the first large-scale, targeted effort toward integration of advances in computing and control of autonomous, uninhabited air vehicles (UAVs). The SEC vision was to enable advanced control systems that exploit information to significantly increases in the performance and reliability of these vehicles. As part of the SEC program, the University of Minnesota (UMN) / University of California, Berkeley (UCB) team developed a unified design framework to synthesize and simulate individual vehicle management systems.

On-line control customization for Uninhabited Air Vehicles (UAVs) was the focus of our efforts during the five year program. Advances in on-line control customization have enabled a dramatic increase in military effectiveness by increasing the level of autonomy in UAVs, probability of mission success and survivability, expanding the range of UAV missions while reducing air vehicle fatigue and life cycle costs. The benefits to the military include use of extremely aggressive maneuvering of UAVs to achieve mission directives, accommodation of goal changes in real-time, life-extending control, a reduced need for hardware redundancy while allowing more complex control strategies without increased software production and verification costs. A key component of our research was the integration of our algorithms into the Open Control Platform (OCP) software infrastructure.

During the five year program advances were made in the areas of modeling, receding horizon control (RHC), linear parameter-varying (LPV) control, fault detection, reconfiguration, anytime control algorithms and real-time control interfaces and algorithms. The following is a list of UMN/UCB accomplishments under this contract.

- Development of a high fidelity model of a high performance, fighter aircraft, based on NASA Technical Paper 1538 [1]. This model served as the baseline fixed-wing UAV for the SEC program and was released as part of the OCP distributions. The nonlinear simulation included a baseline flight controller and was fully integrated into the OCP environment. This work was part of our collaboration with Honeywell Technology Center and Northrop Grumman.

- Early in the SEC program it was important to seamlessly integrate the OCP environment with Matlab and Simulink. UMN led this integration effort which included the networking of Matlab/Simulink processes across computer platforms.

- Developed a quasi-LPV model of the high performance nonlinear aircraft model. A systematic approach to transforming nonlinear, fixed wing, aircraft equations of motion into a quasi-LPV model was developed that is applicable to all similar UAVs. Development of an accurate quasi-LPV model of a UAV was important for LPV controller synthesis and RHC/LPV optimization.

- Developed nonlinear receding horizon controllers (RHC) based on LPV control Lyapunov functions for the high fidelity fixed-wing UAV.

- Developed numerically efficient algorithms to solve the nonlinear aircraft RHC control problem using NPSOL. As part of this effort, the trade-off between accuracy of solution and computational expense was investigated to address real-time computational constraints and issues. This led to the real-time implementation of nonlinear RHC for the longitudinal axis of the high fidelity fixed-wing UAV model. This represented a factor of 400 speedup relative to previous simulation results.

- On-line, linear system-theoretic algorithms were developed to handle variable length preview information. The length and accuracy of preview information provided by the mission planner allows on-line adaptation of the flight control system. These algorithms make optimal use of sensor and command preview information which led to an increased responsiveness of the UAV vehicle.

- Theoretical results were derived on the stability and performance enhancement of receding horizon control with a disturbance rejection objective, under the hypotheses: an existing controller is available and the disturbance, over the upcoming horizon length, is known. These results showed how an existing controller, preview of disturbances and commands and on-line optimization could be used to improve the disturbance rejection and tracking accuracy properties of the design.

- Linear matrix inequalities (LMIs) were used extensively in the synthesis and analysis of controllers for the SEC program. To efficiently solve these equations, semi-definite programming software (SDP) solvers, e.g. the LMI Matlab Toolbox (LMILab), are used. Development of fast, efficient and accurate SDP solvers is an active research area in the applied math, optimization and the operations research community. Many freely available, public domain SDP solvers exist and most take advantage of sparsity in the LMI constraints, offering perhaps significant decreases in computation time required to solve SEC control problems. Each offer different algorithms to solve the LMI optimization. A Matlab based translation code was developed to translate LMI problems formulated in LMILab to other freely available SDP solvers. This allows LMI problems formulated within LMILab to be solved with six freely available SDP solvers. This code was made available to all researchers through a link on the UMN SEC webpage.

- Formulated an LMI search method for non-quadratic sum-of-squares polynomial Lyapunov functions that simultaneously stabilizes a switched linear system. For the given class of switched linear systems, it was shown that one needs to only search over homogeneous forms for stabilizing solutions. A possible application of this result is to prove stability of a system where an on-line supervisor can abruptly switch between controllers while the plant is running.

- Development of anytime control algorithms for linear systems with guaranteed computation time. These algorithms allow controllers to taken advantage of dynamic scheduling where allotted CPU time can vary. Anytime control algorithms incorporate trade-offs between performance and CPU time. A smooth switching scheme was used which ensures smooth transitions of states and outputs as well as guarantees stability.

- Development of decentralized receding horizon control schemes for cooperative control of multi-vehicle formations. A new framework is formulated based on constrained optimization techniques to address this challenging problem. Special emphasis is put on decentralization, which makes the proposed methods implementable in practice. The optimal control framework and the proposed approach holds the promise to enable control design in a systematic way for real-time decentralized collision-free vehicle formation maneuvers, where the design cycle and the development time are not prohibitive and scalable. This framework allows different maneuvering objectives to be achieved by changing appropriate terms in the cost function (e.g. formation keeping and formation joining). It is immediate to include vehicle input and state constraints as well as to use multi-input multi-output linear vehicle models. Controller synthesis and theoretical development make use of algorithms that rely on the most advanced results in the field of computational geometry, mathematical programming solvers, constrained optimal control, invariant set computation and hybrid systems. These techniques allow the formulation of constrained optimal control problems and the computation of their equivalent look-up tables, which are easily implementable in real-time on a specific hardware platform.

- Co-edited with Tariq Samad at Honeywell the IEEE Press volume entitled "Software Enabled Control: Information Technologies for Dynamical Systems." [2] This volume covers advances in software enabled control or "control/software co-design." Topics covered in this volume included Software Architectures for Real-Time Control, On-Line Modeling and Control, and Hybrid Dynamical Systems. It was published in March 2003.

- Developed a RHC programming interface (API) for the OCP. This provided a standard receding horizon control interface with the OCP for all investigators. The RHC API was optimized to work with the OCP running under Windows and QNX operating systems and implemented all required optimization routines to solve the RHC algorithms in real-time. As part of the RHC API, access to the real-time, adaptive resource manager (RT-ARM) was supported. This allowed implementation of the RHC algorithms as anytime processes. This was a major accomplishment and the first time RHC algorithms have been implemented in this manner. This implementation was also flight tested as part of the Boeing SEC fixed-wing final exam experiments.

- Developed, synthesized, implemented and tested observer-based fault detection (FD) filters and threshold functions on the Boeing T-33 final exam platform. The FD algorithms were used to detect actuator faults in real-time as part of the final exam scenario. The information derived from the FD filter was used to update the on-line model of the T-33 aircraft. That model is used by the RHC algorithms for tracking control and trajectory generation.

- Synthesized, implemented and tested on-line RHC trajectory tracking algorithms for the Boeing T-33 final exam platform. The RHC algorithms continuously re-optimized and tracked trajectories in real-time subject to the on-line T-33 model, environment, maneuvering capabilities and current operational constraints. Both linear and linear parameter-varying RHC algorithms were implemented and flight tested in real-time under QNX on the Boeing T-33 final exam platform.

The subsequent chapters of this final technical report summarize contributions that the University of Minnesota / University of California, Berkeley team accomplished as part of the DARPA SEC Fixed Wing Capstone Flight Demonstration. The flight tests were performed at NASA Dryden in Edwards, CA during the two-week period of June 14–25, 2004. Benchmark and hardware-in-the-loop simulation results are presented and evaluated together with data from the actual flight tests.

1.2 Publications

The following publications resulted from this contract.

Theses

- R. Bhattacharya, "Transformation of Linear Control Algorithms into Operationally Optimal Real-Time Tasks," Ph.D. Thesis, University of Minnesota, December 2002.
- K. Zou, "Application of Nonlinear Receding Horizon Control to the F-16 Aircraft," Master's Thesis, June 2002.
- R. Ingvalson, "H_∞ Fault Detection Filter Design for a Closed-loop System," Master's Thesis, May 2004.
- Z. Jarvis-Wloszek, "Matrix Representations of Polynomials: Theory and Applications," Master's Report, May 2001.
- S.M. Estill, "Real-time Receding Horizon Control: Application Programmer Interface Employing LSSOL," Master's Report, December 2003.

Journal Publications

- R. Bhattacharya and G.J. Balas, "Anytime control algorithm: A model reduction approach," *AIAA Journal of Guidance, Dynamics and Control*, accepted for publication, May 2003.
- Z. Jarvis-Wloszek, D.O. Philbrick, M.A. Kaya, A.K. Packard and G.J. Balas, "Control with disturbance preview and online optimization," *IEEE Transactions on Automatic Control*, vol. 49, no. 2, 2004, pp. 266-270.
- J. Bokor and G.J. Balas, "Detection filter design for LPV systems – A geometric approach," *Automatica*, vol. 40, pp. 511-518, March 2004.
- R. Bhattacharya, G.J. Balas, M.A. Kaya and A.K. Packard, "Nonlinear receding horizon control of the F-16 aircraft," *AIAA Journal of Guidance, Dynamics and Control*, vol. 25, no. 5, 2002, p. 924-931.
- R. Bhattacharya and G.J. Balas, "An algorithm for computationally efficient digital implementation of LTI controllers," *Automatica*, submitted for publication, October 2002.
- R. Bhattacharya and G.J. Balas, "Implementation of online control customization within the Open Control Platform, " Software-Enabled Control: Information Technology for Dynamical Systems, *IEEE Press*, Wiley-InterScience, T. Samad and G.J. Balas, Editors, ISBN 0-471-23436-2.

Books

- Software-Enabled Control: Information Technology for Dynamical Systems, *IEEE Press*, Wiley-InterScience, T. Samad and G.J. Balas, Editors, ISBN 0-471-23436-2, May 2003.

Referred Conference Publications

- H. Rotstein, R. Ingvalson, T. Keviczky and G.J. Balas, “Input-Dependent Threshold Function for an Actuator Fault Detection Filter,” *16th International Federation of Automatic Control World Congress*, Prague CZ, July 2005, submitted Oct 2004.
- F. Borrelli, T. Keviczky, G.J. Balas, G. Stewart, K. Fregene and D. Godbole, “Hybrid Decentralized Control of Large Scale Systems,” *Hybrid Systems: Computation and Control*, Zurich, Switzerland March 2005.
- F. Borrelli, T. Keviczky and G.J. Balas, “Collision-free UAV formation flight using decentralized optimization and invariant sets,” *IEEE 2004 Conference on Decision and Control*, Paradise Island, Bahamas, Dec. 2004.
- T. Keviczky, F. Borrelli and G.J. Balas, “Hierarchical Design of Decentralized Receding Horizon Controllers for Decoupled Systems,” *IEEE 2004 Conference on Decision and Control*, Paradise Island, Bahamas, Dec. 2004.
- T. Keviczky, F. Borrelli and G.J. Balas, “A Study on Decentralized Receding Horizon Control for Decoupled Systems,” *2004 American Control Conference*, Boston, MA.
- Y. Ketema and G.J. Balas, “Agent–localized sufficient conditions for formation stability,” *IEEE 2003 Conference on Decision and Control*, Maui, HI December 2003.
- T. Keviczky and G.J. Balas, “Software enabled flight control using receding horizon techniques,” *AIAA Guidance, Navigation and Control Conference*, August 2003, AIAA-2003-5671.
- T. Keviczky and G.J. Balas, “Receding horizon control of an F-16 aircraft: a comparative study,” *European Control Conference*, September 2003.
- J. Bokor, Z. Szabo and G.J. Balas, “Inversion of LPV systems and its application to fault detection,” *IFAC SAFEPROCESS Conference*, Washington D.C. November 2003.
- R. Bhattacharya and G.J. Balas, “An algorithm for computationally efficient digital implementation of LTI controllers,” *American Control Conference*, June 2003, vol. 2, pp. 1165-1170.
- Y. Ketema and G.J. Balas, “Formation stability with limited information exchange between vehicles,” *American Control Conference*, June 2003, vol. 1, pp. 290-295.
- R. Bhattacharya and G.J. Balas, “Implementation of control algorithms in an environment of dynamically scheduled CPU time,” *AIAA Guidance, Navigation and Control Conference*, August 2002, AIAA-2002-4758.
- J. Bokor and G.J. Balas, “Detection filter design within the LPV framework,” *Proceeding of 19th Digital Avionics Systems Conference*, Philadelphia, PA, October 2000, GA3/1-5, Volume 2.
- R. Bhattacharya, G.J. Balas, M.A. Kaya and A.K. Packard, “Nonlinear receding horizon control of F-16 aircraft,” *2001 American Control Conference*, Alexandria, VA, June 2001, pp. 518-522.

Chapter 2

Real-time distributed control: the Open Control Platform

The growing complexity of control applications requires a software infrastructure that supports the developer in leveraging from inter-process communication, operating systems, the implementation details of tasks scheduling, and low level device control software in a seamless manner. This enables the developer to concentrate all his design efforts on the overall system behavior. One of the ultimate goals of the DARPA Software Enabled Control program was the development of the Open Control Platform, which is a software technology supporting real-time distributed control application development and implementation. This chapter explores the main features and underlying technology of the OCP.

2.1 Software enabled control systems architecture

Before concepts like software enabled control systems were formalized, control research was proceeding down a path determined by old views of the computational and systems context. Normal assumptions were: highly constrained sensing and actuation, limited processing and communications resources, computational intractability of large or even moderate state spaces, poorly characterized and unpredictable switching effects, and target systems that operated independently and without interaction with other systems [3].

Control theory and engineering have a remarkably successful history of enabling automation, and information-centric control is by now pervasive. Yet today's controllers are conservative: being products of over-design, they often yield under-performance. Their designs are statically optimized for nominal performance, around simplified time-invariant models of systems dynamics and a well-defined operational environment. They also fail in unexpected circumstances: control vulnerabilities that arise in extreme environments are frequently ignored. Systems modifications (reconfigurations, damage, failure) may demand large changes in the controller, perhaps on-line during operation.

With the advent of networked sensors and actuators, distributed computing algorithms, and hybrid control, the term "systems dynamics" has taken on a whole new meaning. Whereas it used to bring to mind only ordinary differential equations with perhaps some parameter uncertainty, noise, or disturbances, we can now include dynamic tasking, sensor and actuator reconfiguration, fault detection and isolation, and structural changes in plant model and dimensionality. Consequently, the ideas of system identification, estimation, and adaptation must be reconsidered.

This new perspective of the world also requires new models for control software implementation, avoiding to think of the software as simply the language of implementation. Control code (particularly, embedded control code) is a dynamic system. It has an internal state, responds to inputs, and produces outputs. It has time scales, transients, and saturation points. It can also be adaptive and distributed. As a control engineer knows, if we take this software dynamic system and couple it to the plant dynamics through the sensor and actuator dynamics, we have a composite system whose properties cannot be decided from the subsystems in isolation.

Thus, when we put an embedded controller on a hardware platform, we have not only a coupled system with significant off-diagonal terms, but a distributed one as well. To borrow from the computer engineering terminology, we have a problem in control/software co-design. The control design is evolving through the development of hybrid optimal control, reachability analysis, multiple-model systems, and parameter-varying control. The software is being facilitated by distributed computing and messaging services, distributed object models, real-time operating systems, and fault detection algorithms.

2.2 Open Control Platform overview

The Open Control Platform (OCP) provides an open, middleware-enabled software framework and development platform for control technology researchers who want to demonstrate their technology in simulated or actual embedded system platforms [4]. The middleware layer of the OCP provides the software layer isolating the application from the underlying target platform. It provides services for controlling the execution and scheduling of components, inter-component communication, and distribution and deployment of application components onto a target system. The embedded system domain of particular interest to the SEC program is that of uninhabited aerial vehicles (UAVs), however the software architecture of the OCP leaves the possibility of applications in other domains open.

The goals of the OCP are the following:

- Provide an open platform for enabling control research and technology transition.
- Support dynamic configuration of components and services.
- Provide a mechanism enabling the transition between execution and fault management modes while maintaining control of the target systems.
- Allow for coordinated control of multiple target systems.
- Provide a software system infrastructure that is isolated from a particular hardware platform or operating system.

The major components of the OCP software are summarized in the list below.

- *A middleware framework based on RT-CORBA* [5]. Provides the mechanism for connecting application components together to control their execution.
- *A simulation environment.* Allows the embedded application to execute and be tested in a virtual world, reading simulated sensors and driving simulated actuators on plant models.
- *Tool integration support.* Provides linkages to useful design and analysis tools such as Matlab/Simulink and Ptolemy II, allowing controller designs realized in these tools for easier transition to embedded targets.

- *Controls API (Application Programming Interface).* Provides a controls-domain friendly look and feel to the OCP. This is accomplished by using familiar terminology and simplified programming interfaces.

A primary motivating factor in implementing a middleware-based architecture was the promise of isolating the application components from the underlying platforms. This allows for a more cost-effective path for implementing common software components that could be used across different product lines and could be re-hosted onto evolving embedded computing platforms.

2.3 The OCP infrastructure technology

2.3.1 RT CORBA core

The OCP middleware is written in C++. It includes an RT CORBA component [5], which leverages the ACE and TAO products developed by the distributed object computing (DOC) research team at Washington University. TAO provides real-time performance extensions to CORBA.

2.3.2 OCP features

Real time publish/subscribe Communications among the components which use the RT CORBA infrastructure make use of the so called CORBA Event Channel (EC). The power of the EC can be seen from its data distribution capabilities: the components transferring data, including data suppliers that publish data and data consumers that subscribe to data, can either be placed in the same computer or process, or distributed onto other processes or computers, all without a change to the components source code. Only configuration data, specifying where the components reside, needs to be updated to implement a working redistributed architecture.

Naming services The naming service provides a mechanism for getting a reference to a component. This allows an application to store and retrieve references to components that are independent of address space and are therefore portable across processes.

2.3.3 Middleware services

The OCP provides a set of services in addition to the standard services provided by the CORBA specification, which gives additional real-time performance enhancements as well as higher level services (e.g. event service and replication utilities, etc.). This forms the application interface to the underlying middleware and serves two purposes:

- It provides an interface that isolates the application from the details of the underlying RT CORBA implementation and simplifies the application interface to the lower level services.
- It provides a clean interface for extending the base features provided by the controls API.

Resource management The OCP's resource management component provides mechanism for controlling resource in a mode-specific way. This is an essential element for supporting modes in hybrid systems. The designer specifies quality of service (QoS) information, which is an input into the resource management component to control the run-time execution of the OCP. The OCP resource management component is an extension of the Honeywell Labs real-time adaptive resource management (RT-ARM) capability [6]. The resource management component is responsible for partitioning the system resources based on the mode of execution. It performs a sort of meta-scheduling task for the OCP with the assistance of the TAO's scheduling component [7, 8]. The resource management component adjusts rates of execution based on utilization information from the scheduling component and notifies application components when rates have been adapted. These components, that are scheduled with the adapted rates, can then modify their behavior based on the assigned rate (e.g. they can adjust controller gains).

Hybrid systems support A hybrid system is a system that combines both continuous and event driven elements. For example, in a typical flight controls systems, the lower levels of the architecture tend to be designed as continuous-time controllers; when moving to higher levels of the architecture, controllers tend to be of the event driven supervisory type. These controllers are designed to function in one or more modes. Adding mode support to the OCP addresses a set of new challenges especially when controlling a UAV:

1. The OCP must support stable operation of the continuous (physical) system during mode changes.
2. Mode changes require that the system be reconfigured at run-time.
3. To better utilize limited computing resources, the OCP must support adaptation of resources in a mode-specific way.

2.3.4 OCP Controls API

As described earlier, the OCP provides several advanced mechanisms such as dynamic scheduling and resource management. To help hide the complexity from the controls designer, the OCP includes a control designer abstraction layer above the RT CORBA implementation. This API allows the designer to focus on familiar tools and terminology while enabling the use of RT CORBA extensions. This help provides a consistent view of the system that is meaningful to the control designer. In order to accomplish this task, the Controls API has been generalized as a combination of a high level description language and a simple programming interface. The designer expresses the characteristics of the system in familiar terms to form a high level description of the system. This description is then processed to populate a component registry which is used by the OCP. In the following sections the terminology associated with the Controls API is briefly explained.

Signals

The distributed control application design starts from the definition of the signals flowing through the whole application. A signal represents a set of variable definitions (name, type and meaning), which is used to connect the components of the applications via their ports. Signals represent the types of data flowing between the components.

Components and their behaviors

The concept of a component is already a familiar term to the control system designer. A control system is conceptually made up of components which have hardware and software semantics associated with them. These components communicate by reading and writing signals between connected ports. The OCP Component is a software entity that combines one or more legacy components (algorithms or piece of code created prior the introduction of the OCP) into an executable entity. The OCP components could have a one to one mapping or could combine many legacy components. OCP components form the minimum encapsulated entities that are manipulated through the Controls API.

Components often model entities in the physical or modeled world like a continuous or discrete controller, a supervision controller, or a plant model. An OCP component is classified according to the following guidelines:

- Loosely coupled (i.e. can execute in parallel).
- Potential distribution boundary (i.e. component may reside on another processor in a multi-processor system).
- Component choices may also be driven by Quality of Service (QoS) characteristics.
- Tightly coupled and must run at the same rate as other components, based on model of execution.
- Hard Real-Time.
- Soft Real-Time.

Components consist of Ports and Behaviors. The following sections describe the relationships between components, ports, and behaviors.

Ports

Input ports and output ports provide a wrapper for specifying Quality of Service (QoS) information for a component. Besides the Port Name, which provide an entry point for the component, and a Data Type, which specifies the type of signal sent or received, a port defines the Execution Information needed to run the component:

- Execution category: periodic, aperiodic or anytime.
- Execution time: worse case execution time.
- Deadline: deadline from the start of the frame.
- Rate set: the valid set of rates that the entry point can execute in a particular Mode. The rate defines the frequency at which the port scans the signal buffer seeking for new data to input to the component, or vice-versa the rate at which the data are sent out of the component.

Behaviors

Behaviors provide a mechanism for allowing user customization of the types of activities conducted by a component. This allows the user to provide meaningful names to user defined code which will be called by the OCP. The behavior specification syntax also provides a convenient mechanism for specifying relationships between input and output ports. An OCP component may be statically provided by more than one behavior, but at run-time – according to the needs of the control designer – only a subset can be made active. More than one component can read data from the same input port, but an output port can be assigned only to one behavior.

Processes

Processes provide a mechanism for defining how components are deployed in the system. Components are mapped to specific processes of the embedded system. Instances of components defined to be part of a process are unique across all processes in the system. All components that will ever exist in the system must be specified in the processes section. Specifying a component in the processes section does not mean that the component has to be created at startup, but the dynamic behavior is achieved by creating and destroying components at runtime. QoS information must be specified by the components of each process. Component interconnections must be specified for each process.

A graphical representation of the internal organization of an OCP process built of several components is reported in Figure 2.1, while a whole control application in terms of processes running on different machines is reported in Figure 2.2.

2.4 A new concept in real-time control: the *anytime* task

Most control systems built today are resource-limited. This is especially true for embedded control systems in mobile platforms due to constraints of size, weight, space or power. Great effort is expended in engineering solutions that provide jitter-free periodic execution while meeting hard real-time deadlines in systems with high CPU utilization.

Mission-critical command and control is a resource-constrained yet multifunctional enterprise, requiring simultaneous consideration of a variety of activities such as closed-loop control, measurement and estimation, planning, communication, and fault management. On the same computational platform, a large number of tasks must execute.

Anytime or *incremental* algorithms are particularly well suited for implementing tasks that must adapt their resource usage and quality of service [9, 10]. In an anytime algorithm, the quality of the result produced degrades gracefully as the computation time is reduced. In particular, such algorithms may be interrupted anytime and will always have a valid result available. If more computation time is provided, the quality of the result will improve. Example of applications that could benefit from dynamic resource management include:

- Automatic target recognition (ATR) systems which rely on distributed pipelined processing [11].
- Intelligent mission planning software in which processing time is used to synthesize an improved control strategy [12].
- Integrated vehicle health monitoring software based on computationally intensive system identification methods to detect and recover from fault conditions.
- Real-time trajectory optimizers that can dynamically replan routes and trajectories of a fleet of UAV [13].

2.4.1 Resource optimization

At a given instant, the set of currently executing models and tasks have to adapt to both internal and external triggers to make optimal use of the available computational resources. Adaptation among the executing control tasks occurs according to the following steps:

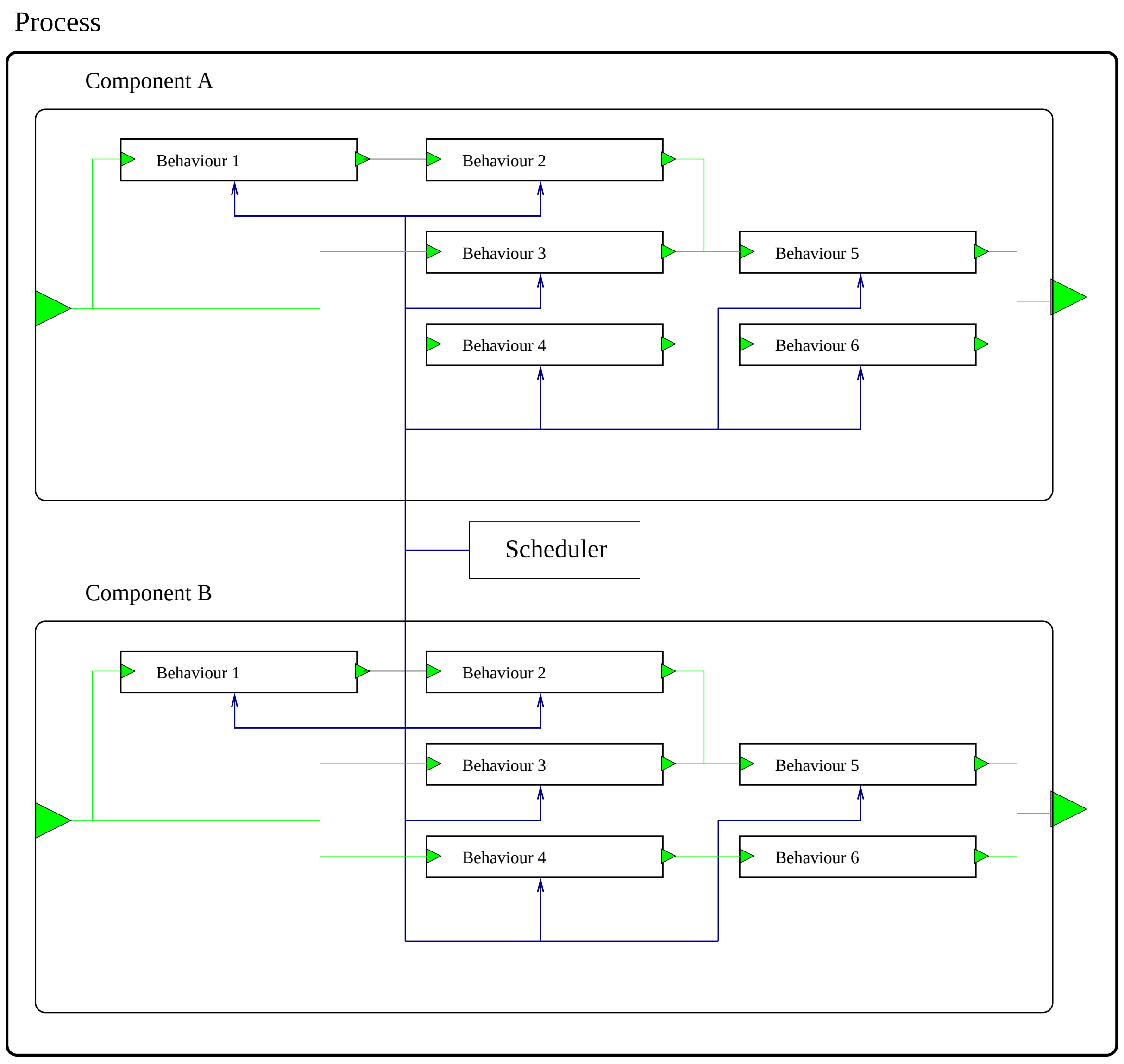


Figure 2.1: The internal logical structure of a single process.

1. Based on the computed or observed state, the criticality, completion deadlines, and computing requirements for the control tasks are determined. These values may be statically determined based on the mission mode or computed on-line by a higher-level planning system. The deadlines we refer to here generally correspond to response times derived from mission-level requirements, such as the need to compute a new trajectory prior to reaching a weather-system or a pop-up threat.

2. The task scheduler makes CPU computing resources available to tasks based on their criticality, computing requirements, and a schedulability analysis. Resources are measured in terms of CPU utilization and computed as execution rate $\times$ execution time per period.

3. Control tasks execute within their allotted time and are subject to preemption if they attempt

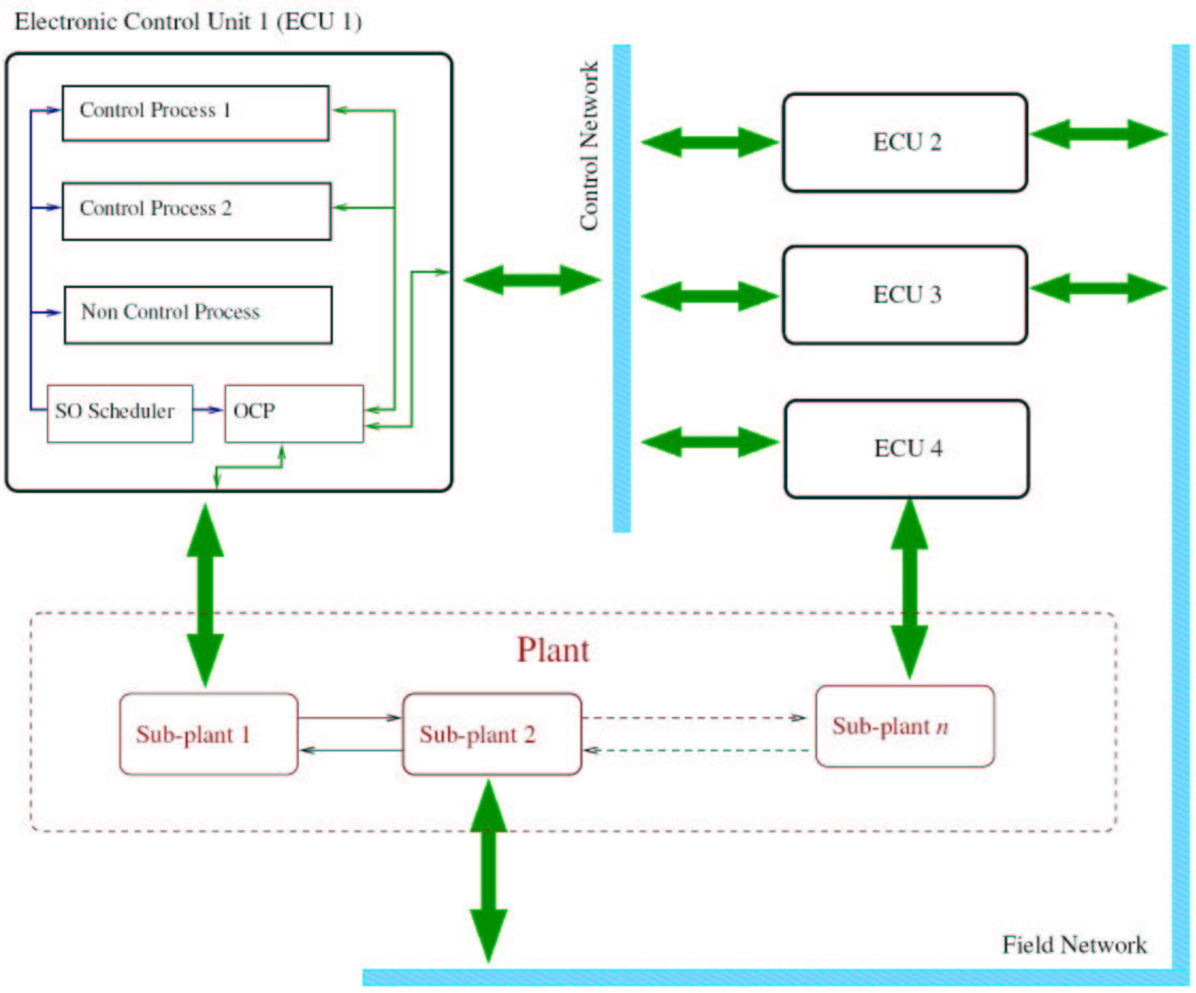


Figure 2.2: Software enabled control architecture functional diagram.

to consume more than their allowed resources. Tasks adapt to meet application constraints. The anytime scheduler provides tasks with the information necessary to adapt their computation to the resource available. The application tasks may need to balance the competing demands of deadlines and accuracy, given the resource made available to them. These can adapt their computation time to meet the deadlines, or adapt the deadlines to meet the assigned computation times, by including more or fewer algorithm iterations or sensor data sources, adjusting model fidelity, and using longer or shorter planning horizons. However, the analysis performed in step 1 ensures that the most critical tasks for the current operational scenario have the highest claim on resources.

In principle, anytime algorithms can make effective use of any amount of processing time that is available.

2.4.2 Anytime task scheduling

Service requirements for the anytime tasks are specified by an application-level policy task called the Anytime CPU Assignment Controller (ACAC). The ACAC is responsible for assigning a weight to each anytime task that indicates its relative CPU assignment. This can be based on deadlines, mission scenario, or other factors. Selection of the appropriate weight is essentially a control activity that can be used to optimize overall performance. Anytime tasks are modeled based on the following characteristics:

1. They are continually executing iterative algorithms that are not periodic. Examples include algorithms that continually refine their result (imprecise computation) and that produce new outputs based on new inputs.

2. Computation times and deadlines for each iteration of the algorithm are an order of magnitude larger than the basic periodic rate.

3. The computation time for each iteration is variable and data-dependent. Furthermore, it is possible for the algorithm to adapt its computation time based on the resource allocated.

Scheduling anytime tasks opens up some interesting issues. For example, since an anytime task may be continually executing, it cannot properly be modeled as a periodic task with properties as desired by rate-monotonic analysis (RMA). In particular, such tasks will all have to be modeled as having a worst case utilization of 1.0 thus rendering any RMA analysis meaningless. In addition, as mentioned earlier, the allocation of the CPU time is now a control function. However, the problem of designing control algorithms to assign computation time for various functional algorithms is poorly understood. In the short term, these “control” algorithms will necessarily be heuristic in nature.

Each anytime task is admitted to the system via a negotiator that can determine whether there is sufficient time in the schedule to accommodate the new task. The anytime scheduler then runs the anytime tasks for an amount of time proportional to its assigned weighting. All anytime task will run within a fixed periodic time block allocated by the system.

It is important that anytime algorithms coexist with the periodic tasks in the control system. In order to achieve this coexistence, the anytime task scheduler executes as a periodic task within the overall control system with period T and execution time C (see Figure 2.3). This periodic task is assumed to run at the system clock rate and can be modeled as a periodic task for rate monotonic analysis. The scheduler allocates a fixed fraction $\frac{C}{T}$ of the overall CPU time for the use of the anytime tasks, and this allocation is then subdivided based on the allocation of individual anytime tasks.

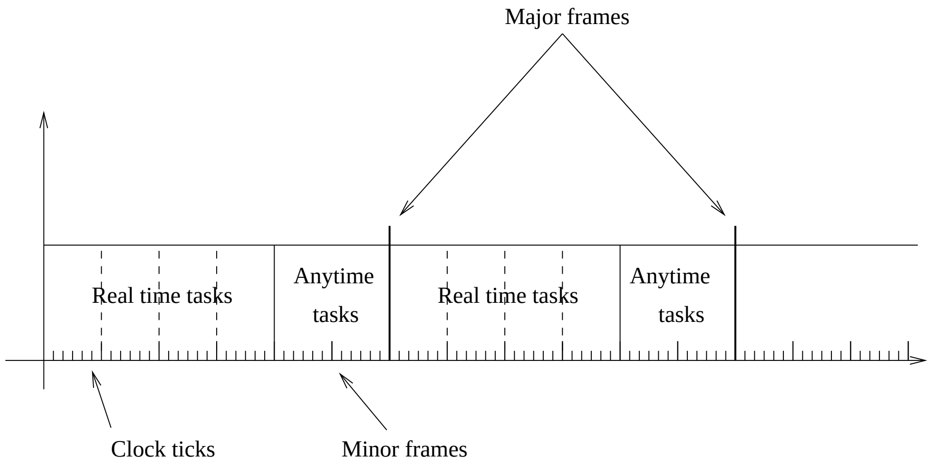


Figure 2.3: The coexistence between anytime tasks and real-time tasks. To achieve this coexistence the anytime scheduler is istantiated as a periodic real-time task and handled by the s.o. scheduler.

Chapter 3

System Identification

3.1 DemoSim description

This section provides a brief description of the Software Enabled Control (SEC) program DemoSim open vehicle simulation model, based on the manual included in OCP release 2.6.6.

DemoSim is a 6DOF flight model of a fixed wing vehicle equipped with an autopilot, which accepts guidance level input commands and performs waypoint tracking. The flight characteristics modeled are those of no actual military vehicle, although they model a flight test vehicle which was configured for SEC program flight demonstrations. The dynamic model and characteristics of the flight test vehicle were modeled in MATRIXx and autocoded into a Windows executable file, which could be distributed among the technology developer teams without compromising Boeing proprietary or otherwise sensitive models and data. The DemoSim open vehicle simulation model has been developed for researcher experimentation leading up to flight testing within a flight test vehicle specially configured for SEC experimentation.

The DemoSim executable allows for data-file driven experimentation with the model as well as control of DemoSim via the so-called UAV Control Interface, which could be used either in a Matlab/Simulink test environment, or within the Open Control Platform that houses the the final flight code implementation in C/C++. Due to the executable nature of the model, DemoSim represents a nonlinear dynamic black-box model of the test aircraft controlled by an autopilot.

The DemoSim input data files include the following:

- a required file specifying vehicle and environment initialization data
- a required file specifying a waypoint plan for the vehicle
- an optional file specifying time-triggered invocations of trajectory commands supported in the SEC open UAV Control Interface for data-file driven experimentation

From a timing perspective, DemoSim can be executed in one of two modes:

- pseudo real-time mode, where a second of simulated time will take approximately one second wall clock time
- "turbo" mode, where simulation time will advance as fast as the computational resources on the PC allow

For the purpose of data-file driven experimentation, DemoSim execution is controlled with commands and command line options in a DOS window. Simulation inputs are defined in ASCII input text files and with DOS command line options. Simulation outputs include the following:

- an ASCII log file capturing numerous parameters for every 10 milliseconds of simulated time
- messages logged to the DOS window which invoked the DemoSim execution, providing feedback on waypoint switching and invocations of trajectory commands defined in the SEC open UAV Control Interface

There are three files that can be used as inputs into the simulation to control DemoSim. Two of the files are required and the third is optional and indicated through the command line. The required files (`veh_init.dat` and `wp_plan.dat`) specify the initial state of the vehicle and an initial waypoint plan. The optional file (`traj_cmd.txt`) can be used to send trajectory control commands, as defined in the SEC open UAV Command Interface, at specific instances in simulated time.

Early DemoSim distributions simulated the behavior of a throttle-handling pilot by applying a quantizer-like effect on speed tracking. Due to numerous requests from the technology developer teams, this feature became optional in later versions allowing for more accurate modeling and evaluation of control algorithms.

The DemoSim open vehicle model could be controlled by the following allowable commands, which had to be either pre-specified in a structured ASCII text file for data-file driven simulations, or commanded in a Matlab data structure in “real-time” for use within the Simulink test environment.

MaintainCurrentHeadingSpeedAltitude This command has no parameters.

SetAndHoldAltitude This command has two parameters representing the commanded altitude in feet and specifying the desired altitude rate in feet per second.

SetAndHoldSpeed This command has two parameters. The first one is the unit specification, which could be selected from

- Millimach
- True Airspeed (ft/s)
- Ground Speed (ft/s)
- Knots

The second parameter is the actual commanded value.

SetAndHoldTurnRate This command has one parameter, which is the commanded turn rate in radians per second.

SetAndHoldHeading This command has two parameters. The first one is the desired heading in radians, while the second represents the direction of turn to obtain the desired heading chosen from the following list

- Left
- Right
- Shortest

FlyToWaypoint The only parameter of this command is the number of the desired waypoint to reach. The list of waypoints is specified in an ASCII input file.

To assist the technology developer teams in the testing and development of their control applications, a single and multi-vehicle MATLAB Simulink environment was provided by Boeing. This environment establishes interfaces to DemoSim and the UAV Control Interface. The `UAV_Control` Simulink block provides the interface to the UAV Control Interface application. The control inputs are passed on to the UAV Control Interface to be processed. The UAV Control Interface then applies the command to the DemoSim executable model, which is running as a separate process. The `DemoSimControl` Simulink block provides an interface for cycling the DemoSim model and returning the vehicle state. For each Simulink simulation step (20 Hz), the DemoSim model will be commanded to cycle 5 times (100 Hz). The output of this block is an array of type double and a size of 15 with the format shown in Table 3.1.

Data entry	**Variable**	**Units**
1	Current Waypoint Index	Integer
2	DemoSim Time	Milliseconds
3	Latitude	Radians
4	Longitude	Radians
5	Baro Corrected Altitude	Feet
6	WGS-84 Altitude	Feet
7	Velocity North	Feet Per Second
8	Velocity East	Feet Per Second
9	Velocity Up (Altitude Rate)	Feet Per Second
10	Ground Speed	Feet Per Second
11	True Airspeed	Feet Per Second
12	Heading	Degrees
13	Pitch θ	Degrees
14	Roll ϕ	Degrees
15	Yaw ψ	Degrees

Table 3.1: DemoSim output variables available in the Simulink test environment.

In addition to the signals available in Simulink, the DemoSim simulation posts data to a log file while it is executing. A new output file is created by each DemoSim execution and logs data every

10 milliseconds of simulation time. Thirty-eight different variables are logged, including signals that are not available on the final test platform. Table 3.2 summarizes the variables logged in the DemoSim output file.

Column	Variable	Units
1–15	same as in Table 3.1	
16	Next Waypoint Index	Integer
17	Calibrated Airspeed	Feet Per Second
18	Mach	Dimensionless
19	Angle of Attack α	Degrees
20	Sideslip β	Degrees
21	Roll Rate P	Radians Per Second
22	Pitch Rate Q	Radians Per Second
23	Yaw Rate R	Radians Per Second
24	North Wind Velocity	Feet Per Second
25	East Wind Velocity	Feet Per Second
26	Down Wind Velocity	Feet Per Second
27	Vehicle North Wind Estimate	Feet Per Second
28	Vehicle East Wind Estimate	Feet Per Second
29	Vehicle Down Wind Estimate	Feet Per Second
30	Flight Path Angle γ	Degrees
31	X-axis Acceleration	Feet Per Second Squared
32	Y-axis Acceleration	Feet Per Second Squared
33	Z-axis Acceleration	Feet Per Second Squared
34	Most Recent Set and Hold Altitude Command	(Baro Corrected Altitude) Feet
35	Most Recent Set and Hold Maximum Altitude Rate Command	Feet Per Second
36	Most Recent Set and Hold Heading Command	Degrees
37	Most Recent Set and Hold Speed Command	Last Commanded Speed Reference
38	Most Recent Set and Hold Turn Rate Command	Radians Per Second

Table 3.2: DemoSim log file output variables.

For more details about the DemoSim model the reader is referred to the documentation that comes with the latest OCP release.

3.2 Identification for RHC

This section is a summary of identification experiments related to the DemoSim open vehicle simulation model conducted at the University of Minnesota. The purpose of system identification is to provide a LTI MIMO description of DemoSim at a single representative flight condition that can be used for design and testing of control technologies developed for the DARPA SEC Fixed-Wing Demonstration.

Identification methods used in the experiments as well as modeling decisions that were made during the identification process are highlighted along with alternative subsystem models of different orders for certain input-output channels. This summary assumes that the reader is familiar with the DemoSim executable model.

3.2.1 Input-output selection

The T-33/UCAV final testbed will accept guidance-level commands from technology demonstrator software. The formulation of the Receding Horizon Control (RHC) scheme that was proposed by our team would suggest the selection of velocity, turn rate and flight path angle as the command signals. This selection of guidance-level commands is not unique. Since not all of these commands can be accommodated by the software on the final testbed, Table 3.3 lists the eventual command inputs that were selected (see corresponding explanation and remarks in the following subsection). The selection of the three output variables shown in Table 3.3 was governed by the objective of the proposed RHC scheme and its cost function formulation based on an LTI MIMO DemoSim model.

Inputs	Outputs
V_{cmd}	V
$\dot{\chi}_{cmd}$	χ
$\dot{h}_{cmd}$	γ

Table 3.3: Selected inputs and outputs of the LTI MIMO Demosim model

3.2.1.1 Remarks on input-output selection

1. It is important to keep in mind during the interpretation of identification results and the controller design/tuning process that velocity tracking will not be accomplished by automatic control in the final tesbed. The pilot will be cued by a numerical display about the desired groundspeed. Any time-delay, dynamics or other phenomena that could be associated with the pilot in the velocity loop were not included in the identification experiments.

2. DemoSim provides the ability to track either a desired heading or a desired turn rate for lateral guidance. However, turn rate is not available as a measurement on the testbed, which motivated the selection of heading as an output for this channel. There were two main reasons

for selecting turn rate as the command input for lateral guidance. One comes from the RHC problem formulation and the assumption that the typical trim flight conditions of the aircraft would be straight and level flight, or steady level turn. Both require a constant turn rate control command in steady state, which is more beneficial from certain numerical aspects of the optimization involved in the control solution. Another reason for choosing the turn rate tracking autopilot option of DemoSim was that the heading control loop of DemoSim interprets changes in the commanded heading as steps, which results in much more aggressive turning of the aircraft than might be desired. This induces significant cross coupling into other outputs and tends to saturate maximum bank angle maneuvering limits. Commanding desired turn rate gives us more control over how much we intend to approach these limits and at the same time reduces cross coupling to other outputs.

3. Even though commanding flight path angle of the aircraft would be more desirable from the aspect of the RHC formulation, DemoSim does not have this capability. Therefore, instead of commanding flight path angle, we have to be content with commanding a climb rate that corresponds to a desired flight path angle using some assumption or measurement on the aircraft's velocity. The climb rate – flight path angle input-output channel of DemoSim also has some peculiarities, which influenced the selection of command inputs for the identification process. Based on previous experiments with DemoSim and a confirmative answer from Boeing on the issue, we concluded that commanding altitude rates smaller than approx. 0.5 ft/s disables the altitude control loop and results in a diverging altitude trajectory. Using larger altitude rate commands avoids this problem, however there are significant steady state errors in altitude rate tracking that depend on the sign of the command signal (i.e. whether ascent or descent was commanded). These issues with the altitude rate tracking controller of DemoSim would make it very difficult to identify and control this channel directly. As an alternative approach, we decided to use the altitude tracking controller of DemoSim (by commanding sufficiently high altitude rates, since h_{cmd} and $\dot{h}_{cmd}$ have to be issued simultaneously). The RHC controller still provides a solution in terms of $\dot{h}_{cmd}$, but it is fed through an integrator to drive the altitude control loop of DemoSim. Based on these considerations, identification was performed assuming an altitude command input, and the input was artificially augmented with an integrator after the identification process.

3.2.2 Assumptions on cross-coupling

Based on extensive experimentation with DemoSim, some entries of the LTI MIMO DemoSim transfer matrix that represent cross-coupling terms between input and output channels were determined to be essentially zero or much less significant than other entries. Note that extra care was taken in these simulations to avoid any nonlinear effects that may stem from saturation of signals such as normal acceleration (n_z), longitudinal acceleration (n_x) and bank angle, or flight envelope limits. The absence of certain cross-coupling terms in our MIMO model does not mean that they are not present in DemoSim. However, based on their magnitude or nature (e.g. very low frequency relatively small altitude oscillation induced by step command on turn rate), any proposed RHC controller should be designed robust enough to handle such level of model mismatch. The assumption on the LTI MIMO DemoSim model structure was decided to be the following for identification purposes:

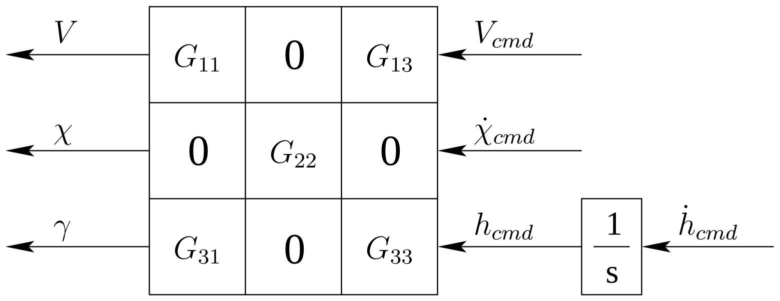


Figure 3.1: LTI MIMO DemoSim model structure assumed for identification

3.2.3 SISO identification of nonzero transfer matrix entries

The identification process was based on time-domain simulation data obtained by exciting DemoSim at the trim values of Table 3.4 corresponding to straight and level flight condition.

h_0	15000 ft
M_0	0.45
V_0	475.291 ft/s
$\bar{q}_0$	169.2628 psf

Table 3.4: Trim values of the identified models in terms of altitude, Mach number, ground speed and dynamic pressure.

The excitation signal was originally designed to be a composite of a doublet and a chirp signal in order to simultaneously capture steady-state behavior and emphasize the frequency region of interest for identification purposes, which was decided to be frequencies below approx. 1 rad/s. The crossover region of a control system using this model is assumed to be between 0.1 rad/s and 0.5 rad/s, approximately. The control signal sampling frequency is assumed to be 2 Hz. The frequency range of the sine sweep in the chirp signal was $\left[10^{-4}\text{ Hz} - 10^{-1}\text{ Hz}\right]$.

Initial identification experiments revealed that in some input-output channels the doublet response of DemoSim was nonlinear. This effect prompted the construction of a different excitation signal using a mixture of ramp, hold and chirp signals. This had the beneficial effect of avoiding saturation in variables such as longitudinal acceleration (n_x), when exciting the V_{cmd} input.

Based on these excitation signals and the simulated DemoSim responses, Empirical Transfer Function Estimates (ETFE) were created to serve as a basis for evaluation of identified models in the frequency domain. An ETFE is obtained as the ratio of the output Fourier transform and the input Fourier transform.

In light of the chosen DemoSim transfer matrix structure, the following approach was used to identify and construct an LTI MIMO prediction model. First, SISO transfer functions were identified corresponding to all nonzero input-output channels. Then balanced truncation was used to construct SIMO models for the velocity V and altitude h inputs based on the identified SISO subsystems. Finally, after checking that all subsystem poles were sufficiently different, a minimal

MIMO realization was constructed by using the identified SIMO and SISO subsystem matrices directly as building blocks.

The SISO models were identified from raw time domain data, as well as using the ETFEs in frequency domain and evaluating the obtained models in both time and frequency domains.

The time domain identification techniques that were used for the individual SISO channels included ARX, ARMAX, Box-Jenkins and the subspace-based MOESP methods. However, eventually the prediction error estimation of an Output Error (OE) model provided the most acceptable SISO models. The Output Error (OE) model has the following form

$$y(k) = \frac{B(q)}{F(q)} u(k - nk) + e(k) \tag{3.1}$$

In some instances "manual" fitting of second order transfer function responses was also performed. The identified discrete-time models had a sampling time of 0.5 seconds.

In the frequency domain, algorithmic (`magfit, fitmag`) and manual (`drawmag, bode`) fitting of transfer function responses to ETFEs was used to obtain alternate models.

Observations of Hankel singular values were also involved in making decisions about model orders of certain input-output channels, however in most cases these tests turned out to be not too decisive.

3.2.3.1 Remarks

1. Most of the identification experiments conducted at the University of Minnesota were performed before June 24, 2003 based on the DemoSim v0.2 open vehicle executable model, which was released on May 23, 2003. However, in order to accommodate some requests of the technology developer teams and correct bugs and modeling issues, a new release was distributed on September 8, 2003. This new release had completely different dynamic characteristics on some input-output channels, which rendered some of our previous modeling efforts futile and made a partial reidentification necessary. Since not all aspects of the updated and corrected DemoSim dynamics have changed, some identified subsystem models could be retained. The particular DemoSim version used to generate data is mentioned in the following sections, which discuss the identification process of each SISO channel.

2. The exact characterization of time-delays associated to each command input was based on later DemoSim versions within the Simulink testing environment. Time-delays were characterized as integer multiples of the sampling time and incorporated into the RHC prediction model. As flight tests later revealed, the dramatic time-delay mismatch in the velocity tracking channel between the hardware-in-the-loop simulations and the piloted aircraft was one of the most critical modeling error, which had a significant degrading effect on the controller performance of each team.

3.2.3.2 $V_{cmd} \rightarrow V$ channel

Doublet commands produced significantly nonlinear responses in this channel, even though the magnitude of the excitation signal was relatively small (20 ft/s). The difference between acceleration and deceleration dynamics of DemoSim was apparent since the longitudinal acceleration and deceleration (n_x) limits were saturated due to the abruptly changing, step-like velocity commands. This phenomenon could not be captured by a single LTI model. A combined ramp, hold and chirp excitation signal was used instead to avoid nonlinear DemoSim behavior. The magnitude of the

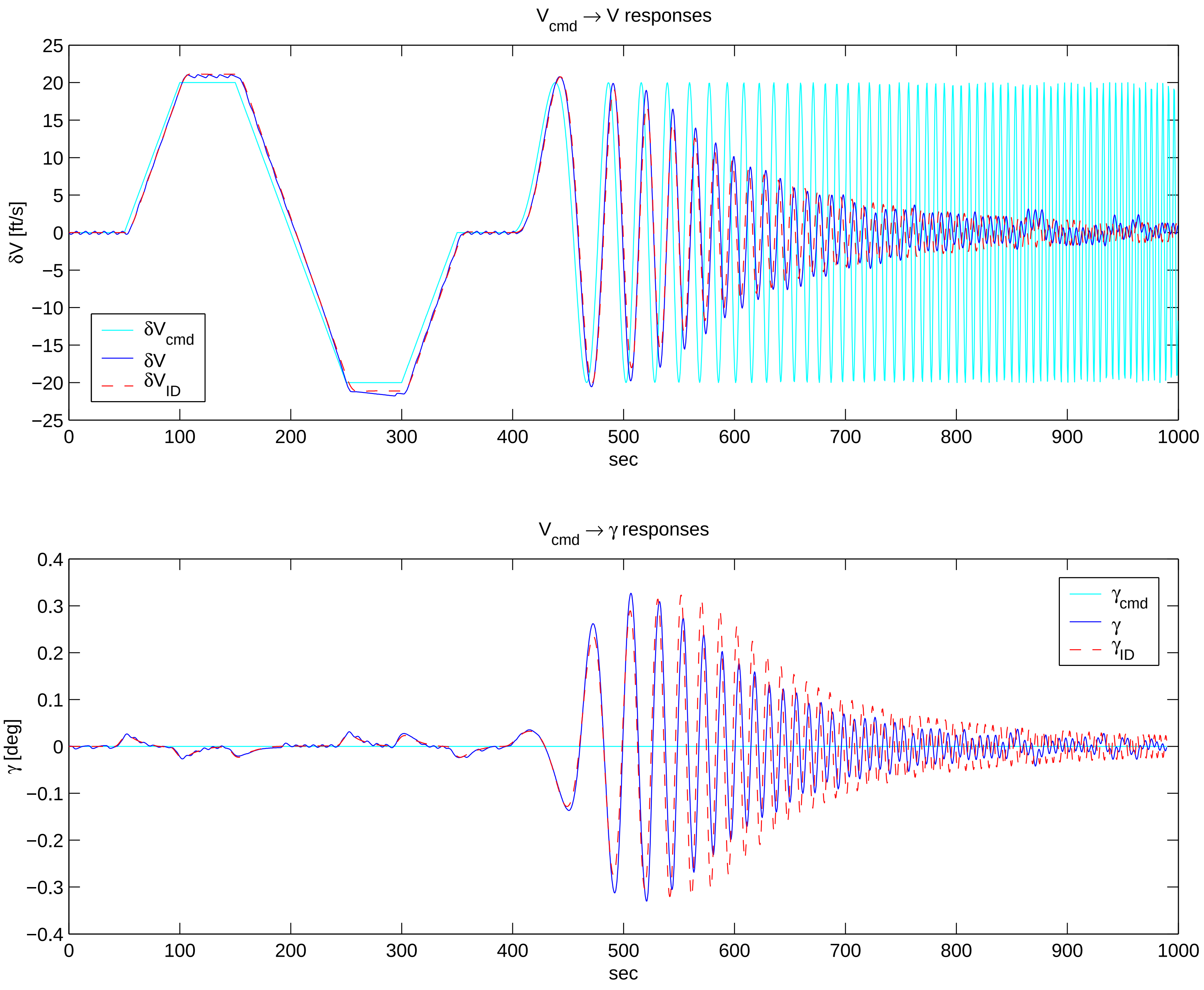


Figure 3.2: Time domain comparison of identified models with DemoSim responses for the V_{cmd} input channel

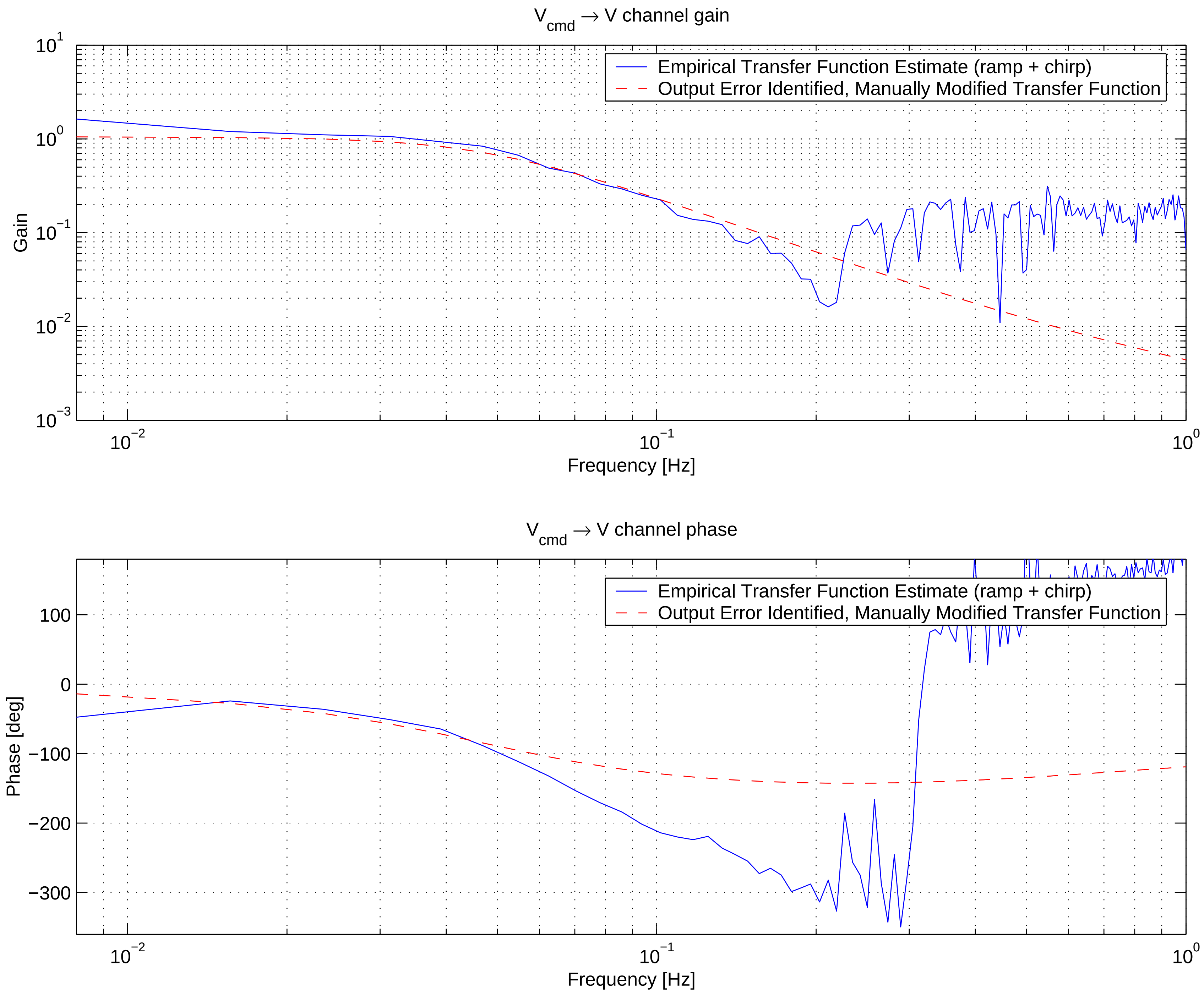


Figure 3.3: Frequency domain comparison between the identified $V_{cmd} \to V$ model and DemoSim ETFE

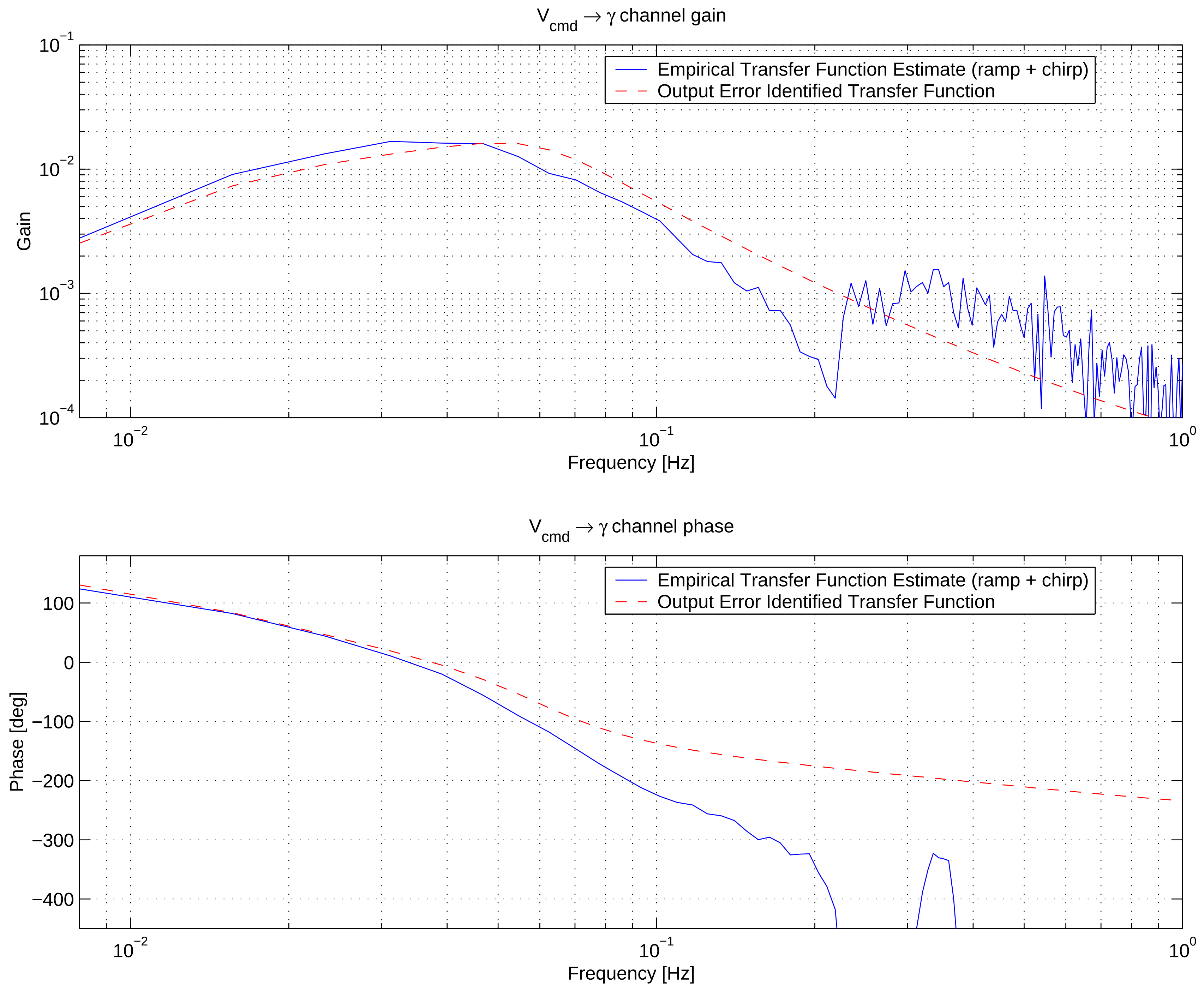


Figure 3.4: Frequency domain comparison between the identified $V_{cmd} \rightarrow \gamma$ model and DemoSim ETFE

combined ramp and chirp excitation signal used in this experiment was chosen to be 20 ft/s and the ramp slope as 0.4 ft/s^2. Input-output data of the May 23, 2003 DemoSim release formed the basis of the identification process for this SISO channel.

Although very accurate fitting of ETFE data could be obtained with relatively low order transfer functions, time response comparisons of these models indicated poor matching with the simulated DemoSim response. Significant overshoot or steady state errors prompted us to rely more on time domain identified OE models. However, frequency responses of these models did not match ETFE data very well (especially in phase above 0.05 Hz), which also shows up when comparing time responses with the chirp phase of the excitation signal, a second order OE identified model with time delay was chosen as a model candidate for this channel. The degree parameters and the time-delay of the identified OE model were chosen as $nb = 1, nf = 2, nk = 1$, where nb and nf denote the degrees of the B and F polynomials in equation (3.1), respectively. This model provided a good compromise for minimizing errors that occured between the identified model and DemoSim data.

However, the poles of the identified OE model were slightly modified manually to achieve better frequency domain fit to DemoSim data. The mismatch in Figure 3.3 is still apparent and the phase plot indicates that the numerator degree is probably too high. This mismatch was deemed acceptable in both frequency and time domains based on the following assumptions:

- The controller bandwidth over the velocity channel is expected to be significantly lower than in other channels on the real testbed.
- Since velocity commands will be implemented by a pilot, DemoSim can provide only an approximate description of the true system's behavior to these commands, which would render any further effort to achieve a more accurate velocity channel model questionable.

Further properties of the identified model are given in Section 3.2.6. Comparative plots between the identified model and DemoSim data are shown in Figures 3.2 and 3.3.

Remark

At the flight condition under investigation (see Table 3.4), the velocity output had a steady state bias of 9.759 ft/s, which was removed manually before applying any identification method. It is important to note that the value of this bias changes with flight condition, exhibiting a positive correlation between its magnitude and the speed of the aircraft.

3.2.3.3 $V_{cmd} \rightarrow \gamma$ channel

Due to significant changes in the dynamic response of this channel, input-output data of the updated September 8, 2003 DemoSim release was used as the basis of the identification process for this SISO channel.

An OE identified model of fourth order ($nb = 3, nf = 4, nk = 2$) was the simplest one that provided an acceptable match in both time and freq. domain comparisons. These are shown in Figures 3.2 and 3.4.

Note that for frequency domain comparisons, ETFE models were generated using the same excitation signal as in Section 3.2.3.2, however the identified OE model did not make use of the chirp response.

3.2.3.4 $\dot{\chi}_{cmd} \rightarrow \chi$ channel

Input-output data of the May 23, 2003 DemoSim release formed the basis of the identification process for this SISO channel. Although intuitively it seems natural that this channel should include an explicit integrator between the turn rate command and heading output, DemoSim experiments showed that the aircraft does not return exactly to its original heading after tracking a turn rate doublet command. This clearly indicates that the anticipated integral action is not present explicitly in this channel, it is only a manifestation of a pole very close to zero. Identification experiments enforcing an explicit integrator by using numerically differentiated heading output data confirmed this fact by showing apparent mismatch in time domain comparisons.

Finally, a combined doublet and chirp turn rate command signal was used in this experiment with magnitude of 0.263 deg/s. A second order OE model with delay was identified ($nb = 1, nf = 2, nk = 2$) using the time domain DemoSim response. The identified discrete-time model was converted to continuous-time using zero order hold transformation and evaluated by the comparisons shown in Figures 3.5 and 3.6.

Further properties of the identified model are given in Section 3.2.6.

3.2.3.5 $h_{cmd} \rightarrow V$ channel

Input-output data of the May 23, 2003 DemoSim release formed the basis of the identification process for this SISO channel. Time domain identified OE models using a combined doublet and chirp excitation signal of 50 ft magnitude did not seem to capture the higher frequency behavior of the system very well. Since the OE method essentially weighs all frequencies equally, some kind of frequency weighting of simulated DemoSim data could be used to remedy this problem.

Instead of pursuing a frequency-weighted time domain identification approach, we found that either a second order "manually" fitted transfer function or a fourth order "systematically" fitted transfer function achieved acceptable accuracy in both domains of evaluation. Comparative plots of these continuous-time transfer functions are shown in Figures 3.7–3.9. Eventually, the second order transfer function was chosen to reduce the complexity of the overall model.

Remarks

1. The velocity output had the same steady state bias as mentioned in Section 3.2.3.2, which was removed manually from DemoSim time response data.

2. A derivative (i.e. a zero at 0) was included explicitly in the "manually" identified second order transfer function.

Further properties of the identified models are given in Section 3.2.6.

3.2.3.6 $h_{cmd} \rightarrow \gamma$ channel

Input-output data of the May 23, 2003 DemoSim release formed the basis of the identification process for this SISO channel as well. Using the same excitation signal as in Section 3.2.3.5, two alternative models were obtained that both showed acceptable match with time and frequency domain DemoSim data.

A "manually" fitted second order transfer function including an explicit derivative was chosen as one alternative.

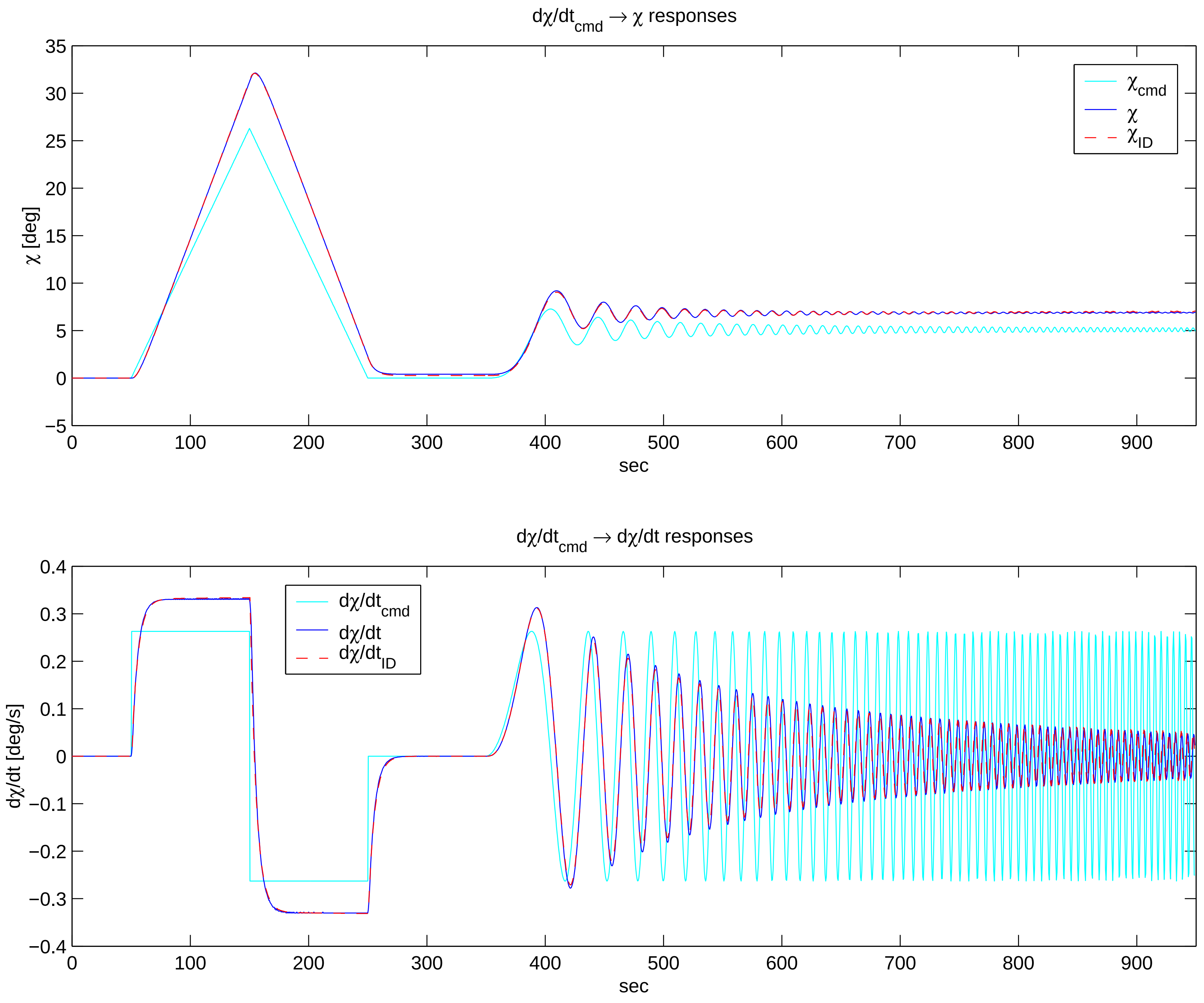


Figure 3.5: Time domain comparison of the identified model and DemoSim responses for the $\dot{\chi}_{cmd}$ input channel

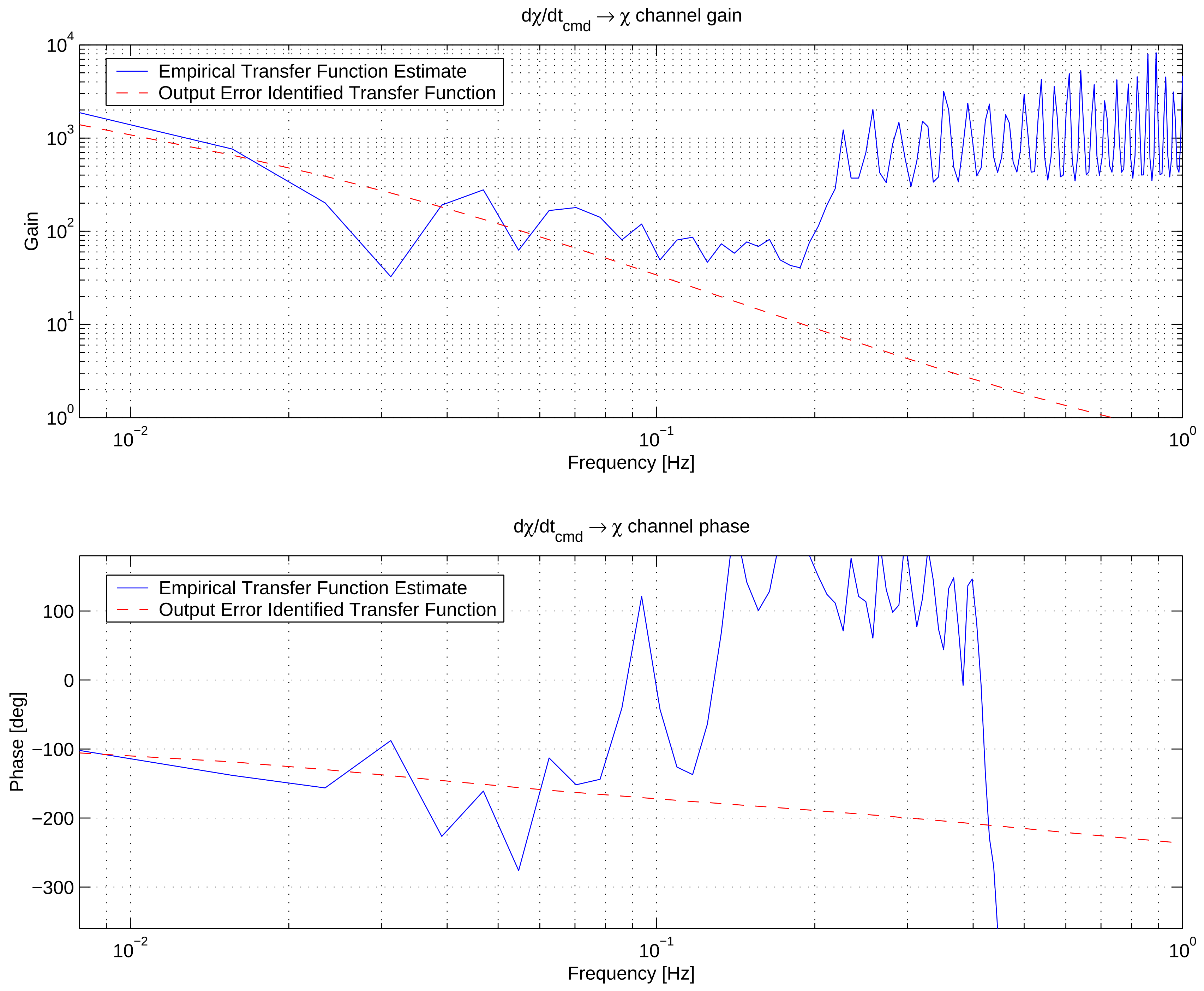


Figure 3.6: Frequency domain comparison between the identified $\dot{\chi}_{cmd} \rightarrow \chi$ model and DemoSim ETFE

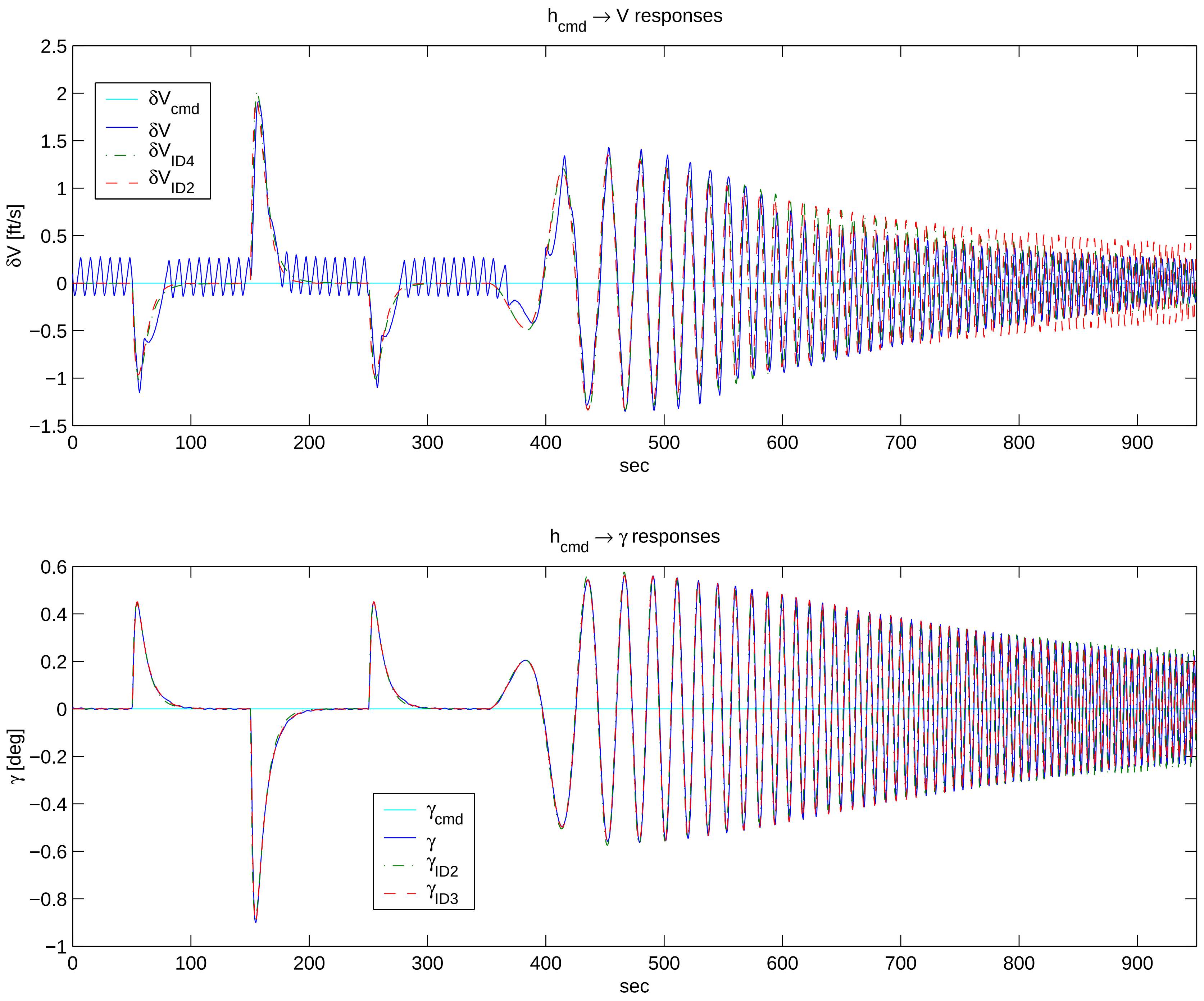


Figure 3.7: Time domain comparison between identified models of different order and DemoSim responses for the h_{cmd} input channel (number in subscript denotes model order)

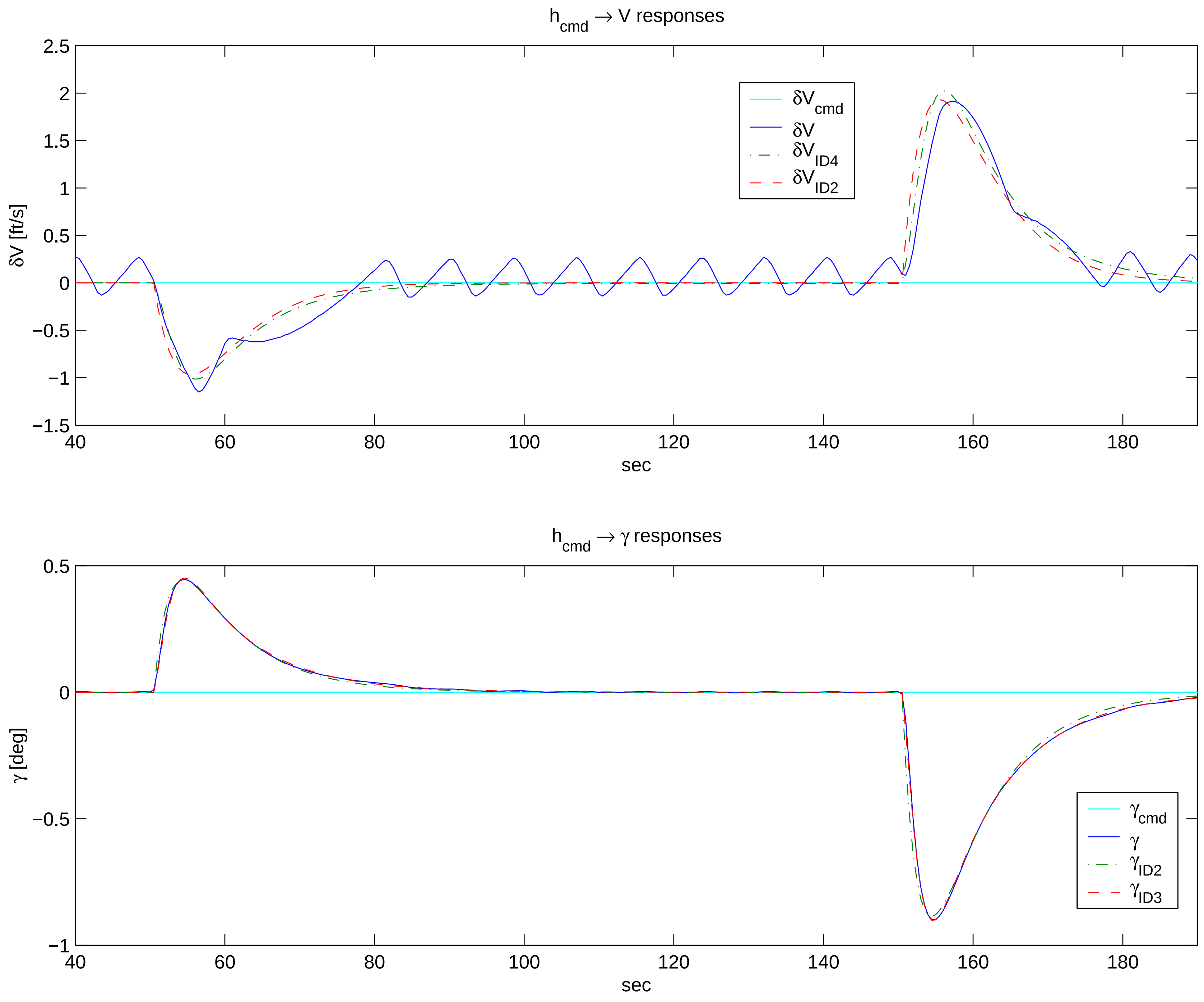


Figure 3.8: Enlarged portions of the time domain comparison shown in Fig. 3.7 for the h_{cmd} input channel

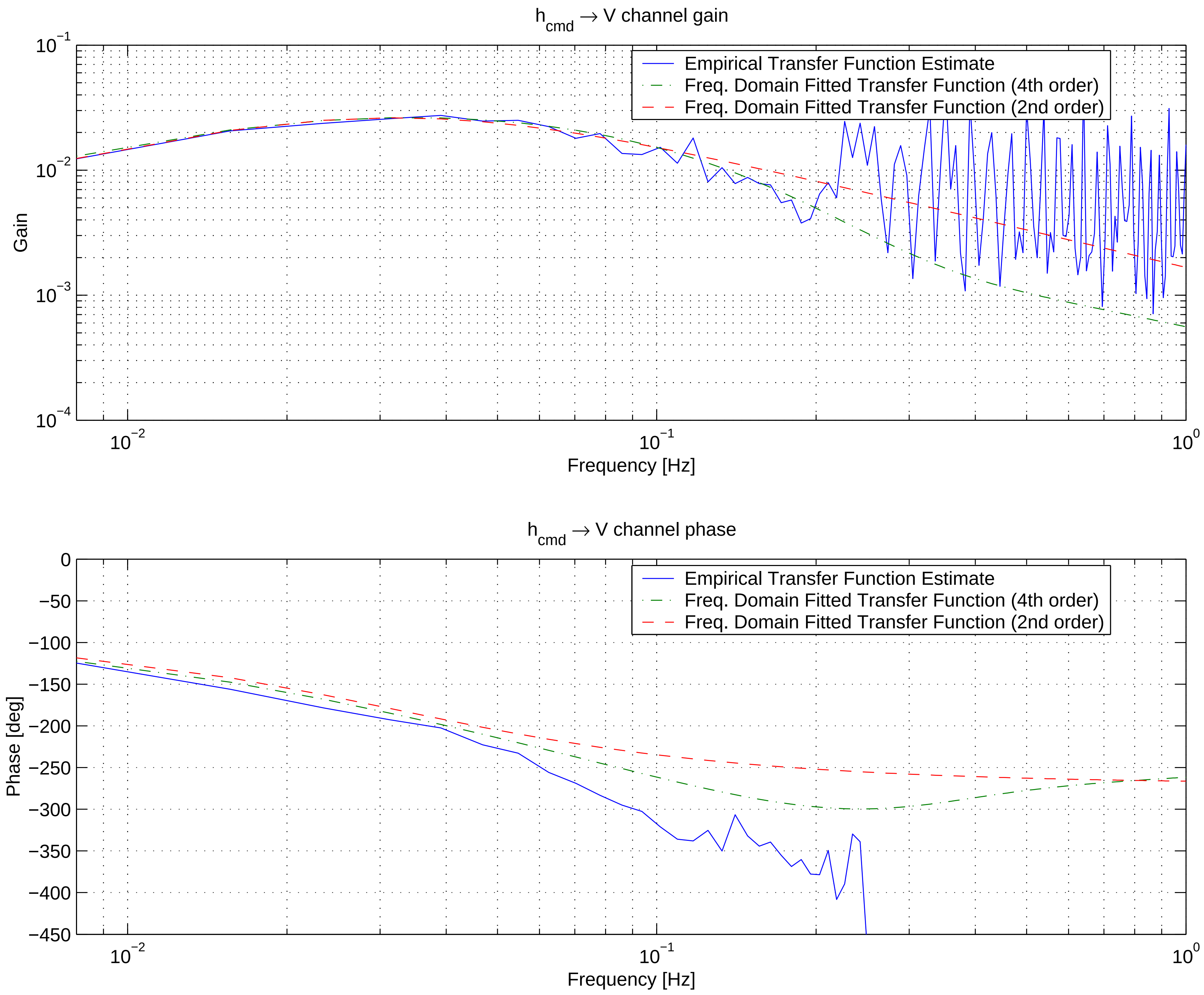


Figure 3.9: Frequency domain comparison between identified $h_{cmd} \rightarrow V$ models and DemoSim ETFE

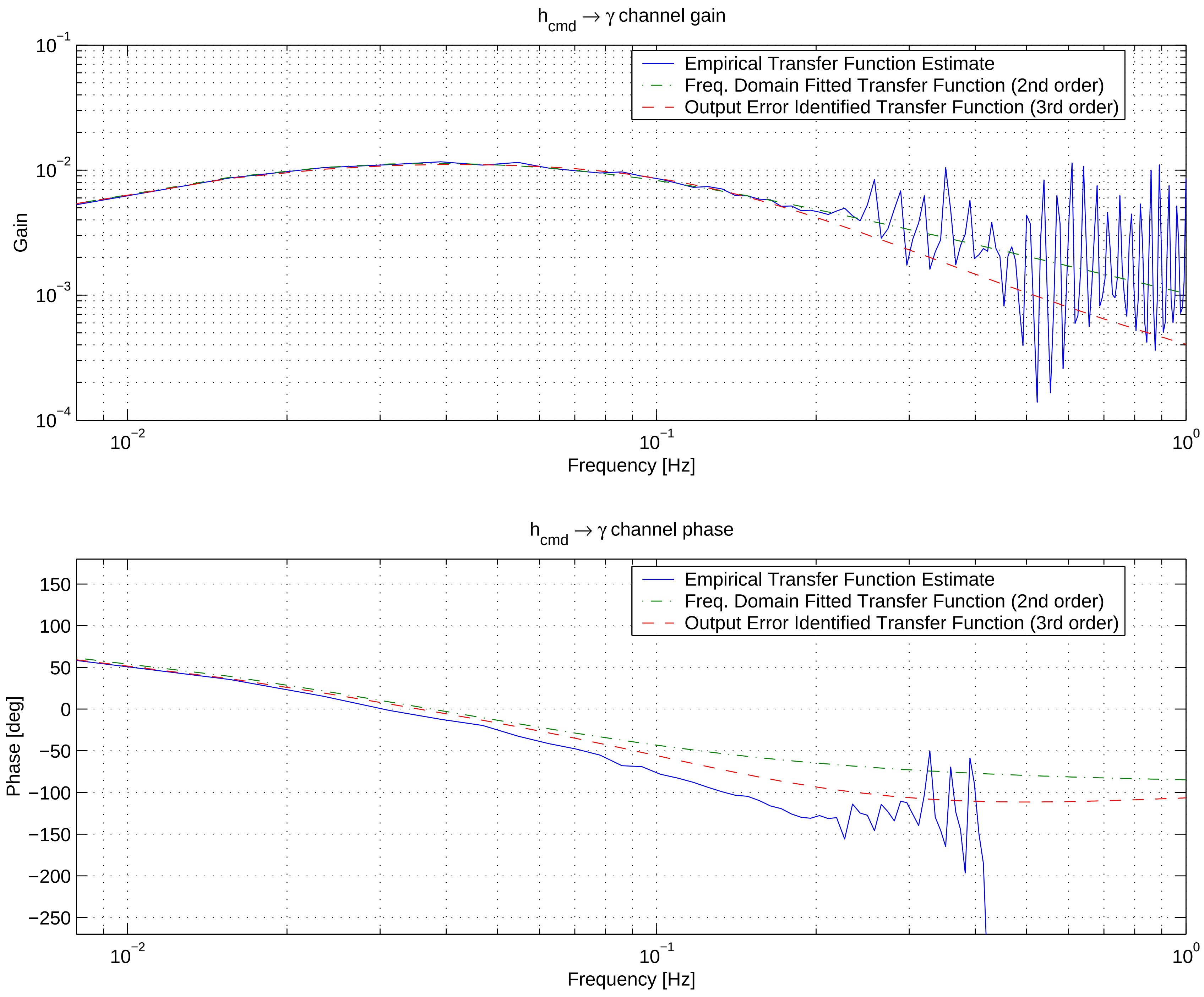


Figure 3.10: Frequency domain comparison between identified $h_{cmd} \to \gamma$ models and DemoSim ETFE

A third order OE identified model based only on the doublet portion of the input-output data was selected as the other alternative that provided somewhat better accuracy. Comparison of these models is shown in Figures 3.7, 3.8 and 3.10. Eventually, the third order OE model was chosen based on its better accuracy.

Further properties of the identified models are given in Section 3.2.6.

3.2.4 Creating reduced order SIMO models

After the identification of individual SISO subsystems, the next step in the modeling process was to merge subsystems that belong to one input channel and create SIMO models that can later be put together in a simple way to form the MIMO system.

SIMO models were created by merging the SISO systems as shown in (3.2) and applying balanced model reduction techniques.

$$\begin{bmatrix} V \\ \gamma \end{bmatrix} = \begin{bmatrix} G_{VV} \\ G_{\gamma V} \end{bmatrix} V_{cmd}, \quad \begin{bmatrix} V \\ \gamma \end{bmatrix} = \begin{bmatrix} G_{Vh} \\ G_{\gamma h} \end{bmatrix} h_{cmd} \tag{3.2}$$

The number of states to keep after truncation was decided after inspecting the diagonal elements of the Gramian corresponding to the balanced realization. The steady-state contribution of the truncated states were kept using the following technique.

Assume that the state matrices of the balanced realization are partitioned according to

$$\begin{array}{cc|c} \boxed{A_{11}} & A_{12} & \boxed{B_1} \\ A_{21} & A_{22} & B_2 \\ \hline \boxed{C_1} & C_2 & \boxed{D} \end{array}$$

and the states corresponding to A_{11}, B_1, C_1, D will be kept. The truncated, reduced order matrices are calculated according to the following formulas to account for the steady-state contribution of the truncated states

$$\begin{aligned} A_r &= A_{11} - A_{12}A_{22}^{-1}A_{21}, \quad & B_r &= B_1 - A_{12}A_{22}^{-1}B_2, \\ C_r &= C_1 - C_2A_{22}^{-1}A_{21}, \quad & D_r &= D - C_2A_{22}^{-1}B_2. \end{aligned} \tag{3.3}$$

Note that since the $\dot{\chi}_{cmd} \rightarrow \chi$ input-output channel was modeled as a completely decoupled SISO subsystem, only the V_{cmd} and h_{cmd} input channels are treated in this section.

3.2.4.1 V_{cmd} input channel

The balanced model reduction was performed using a scaling factor of 50 on the γ output channel. After inspecting the diagonal entries of the balanced Gramian, state matrices were truncated to keep only 4 states and account for the steady-state contribution of the truncated ones.

Although this truncated model had the correct steady-state gain, it introduced high frequency zeros. These zeros were removed from the model manually and one-one zero was added in each output channel, for better frequency domain match.

Figures 3.11-3.12 show a comparison of the bode plots between the reduced order SIMO models and the original identified SISO ones. Figure 3.13 evaluates the SIMO model in the time domain by comparing it to DemoSim output data.

Further properties of the reduced order SIMO models are given in Section 3.2.6.

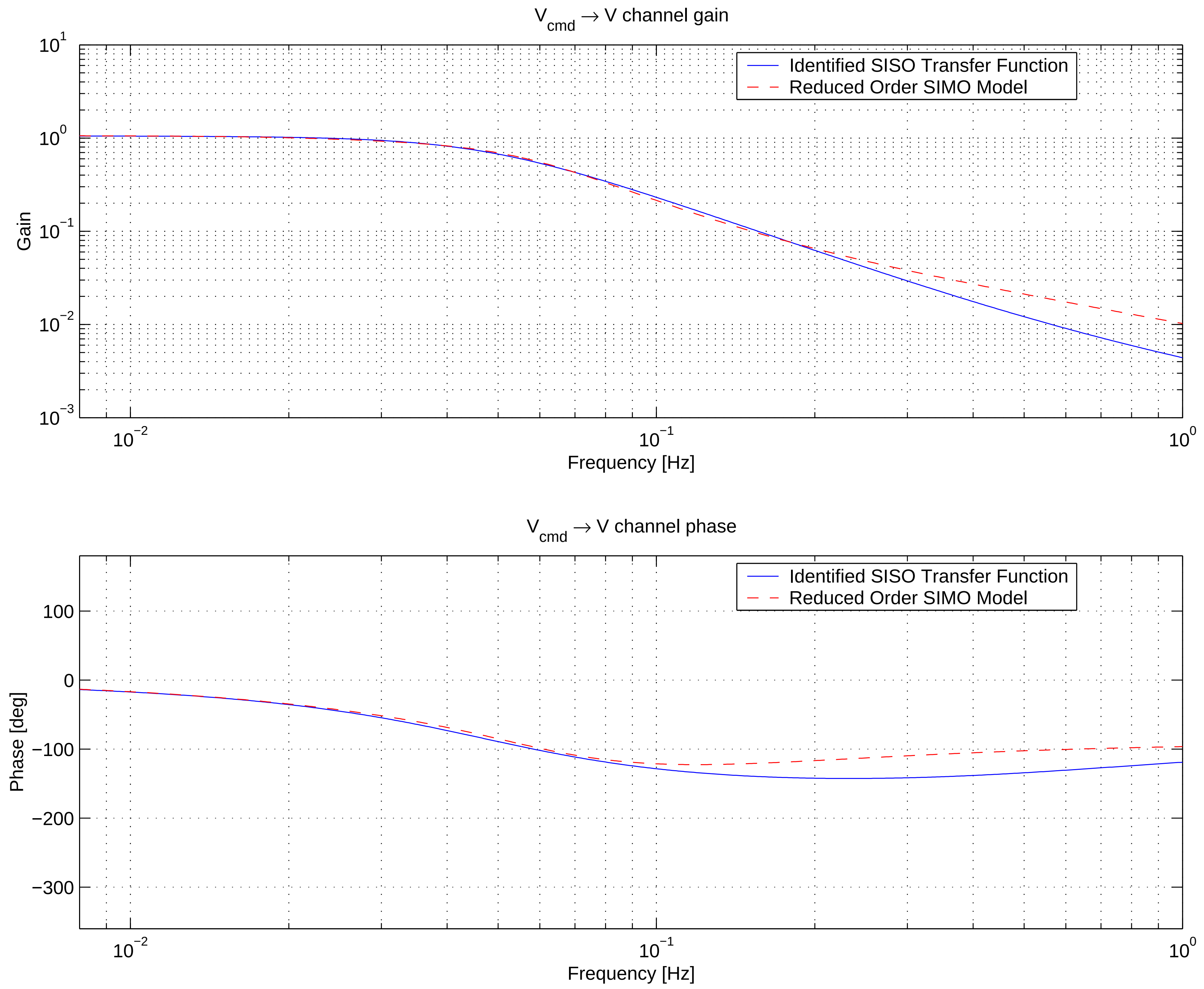


Figure 3.11: Frequency domain comparison between the reduced order V_{cmd} input SIMO model and the identified SISO $V_{cmd} \to V$ model

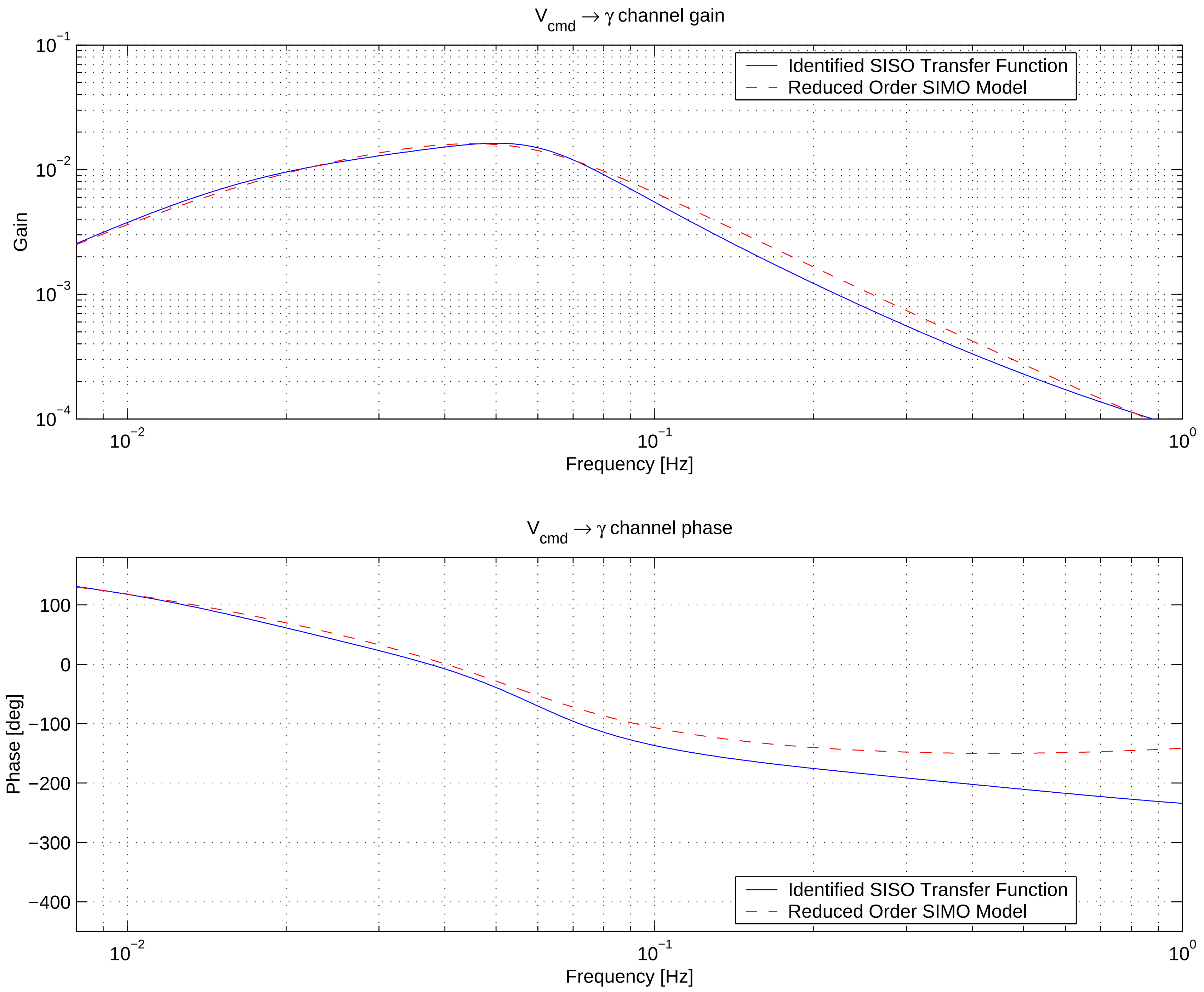


Figure 3.12: Frequency domain comparison between the reduced order V_{cmd} input SIMO model and the identified SISO $V_{cmd} \to \gamma$ model

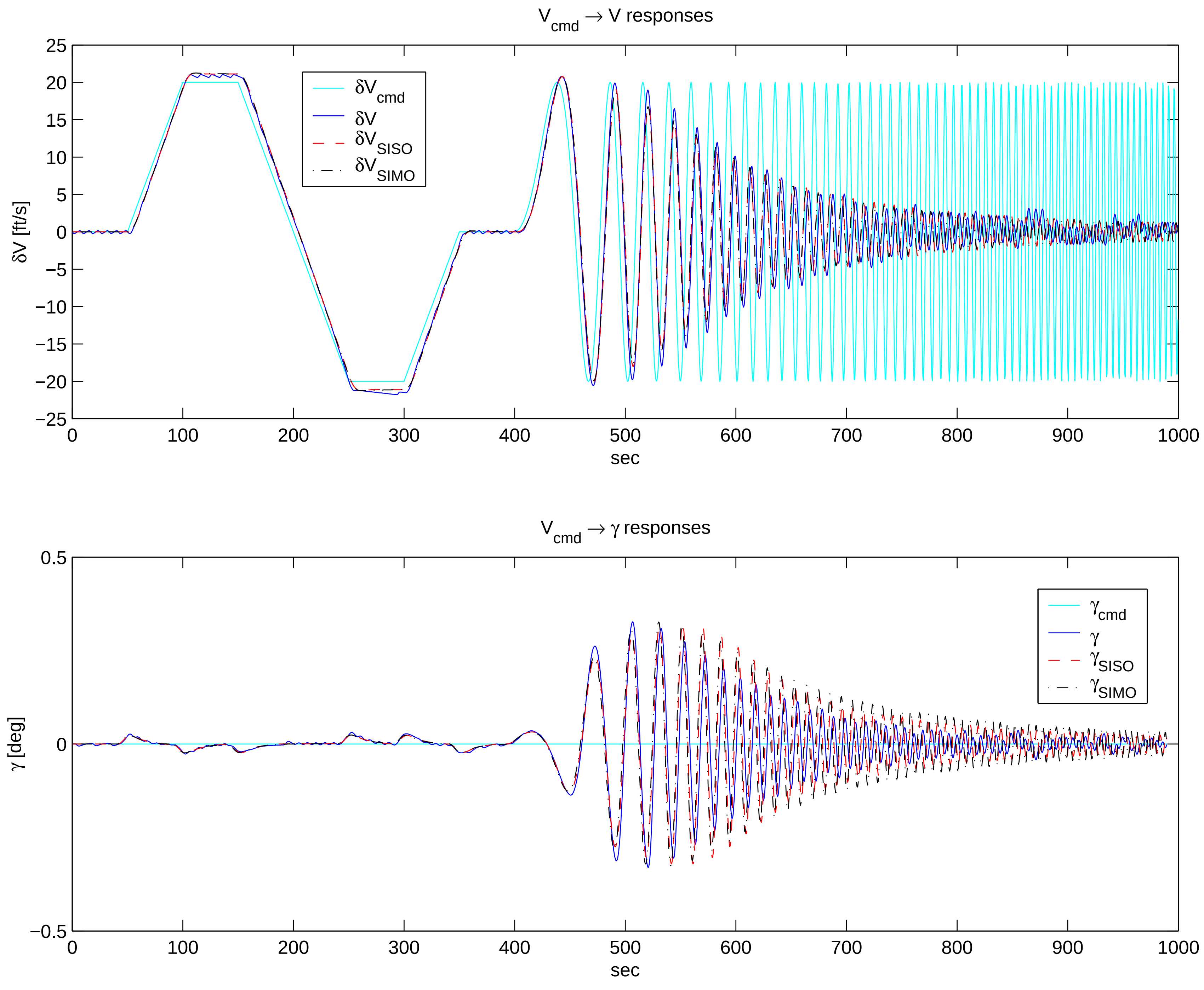


Figure 3.13: Time domain comparison between the identified SISO models, the reduced order SIMO system and DemoSim responses for the V_{cmd} input channel

3.2.4.2 h_{cmd} input channel

The balanced model reduction was performed exactly as decribed in the previous section. State matrices were truncated to keep only 2 states and account for the steady-state contribution of the truncated ones.

Although this truncated model had the correct steady-state gain, it introduced high frequency zeros in this case as well. These zeros were removed from the model manually.

Figures 3.14-3.15 show a comparison of the bode plots between the reduced order SIMO models and the original identified SISO ones. Figures 3.16-3.17 evaluate the SIMO model in the time domain by comparing it to DemoSim output data.

Further properties of the reduced order SIMO models are given in Section 3.2.6.

3.2.5 Building the MIMO model

Poles of the two reduced order SIMO and one SISO subsystems were determined to be sufficiently different from each other to enable building the identified MIMO model using simple diagonal augmentation and merging of the subsystem state matrices based on the corresponding input-output relationship structure. The h_{cmd} input channel was augmented with an integrator to form an $\dot{h}_{cmd}$ input. The combined and augmented MIMO model has 9 states: 4 comes from the V_{cmd} SIMO model, 2 comes from the h_{cmd} SIMO model, 2 comes from the $\dot{\chi}_{cmd}$ SISO model and 1 additional integrator on the h_{cmd} input.

3.2.6 Numerical results

The identified SISO models of Section 3.2.3 and the reduced order SIMO models of Section 3.2.4 are given here along with their poles and zeros.

3.2.6.1 $V_{cmd} \to V$ model

The second order OE identified model after manual modification:

$$1.27\,\frac{0.018268s + 0.0748015}{s^2 + 0.45s + 0.09}$$

Corresponding poles and zeros:

$V_{cmd} \to V$ poles

real	imaginary	frequency	damping
−2.2500E−01	1.9843E−01	3.0000E−01	7.5000E−01
−2.2500E−01	−1.9843E−01	3.0000E−01	7.5000E−01

$V_{cmd} \to V$ zeros

real	imaginary	frequency	damping
−4.0947E+00	0.0000E+00	4.0947E+00	1.0000E+00

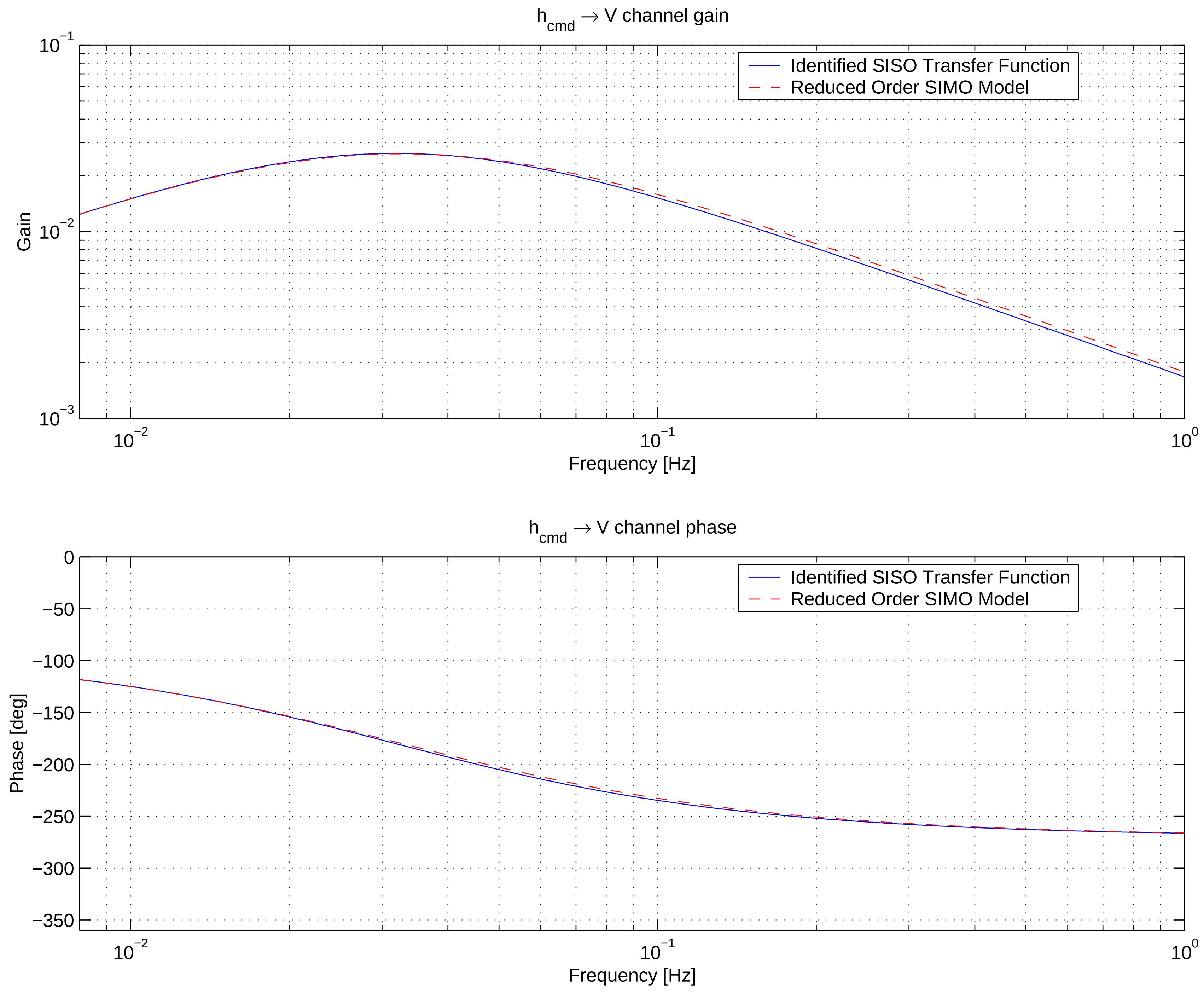


Figure 3.14: Frequency domain comparison between the reduced order h_{cmd} input SIMO model and the identified SISO $h_{cmd} \rightarrow V$ model

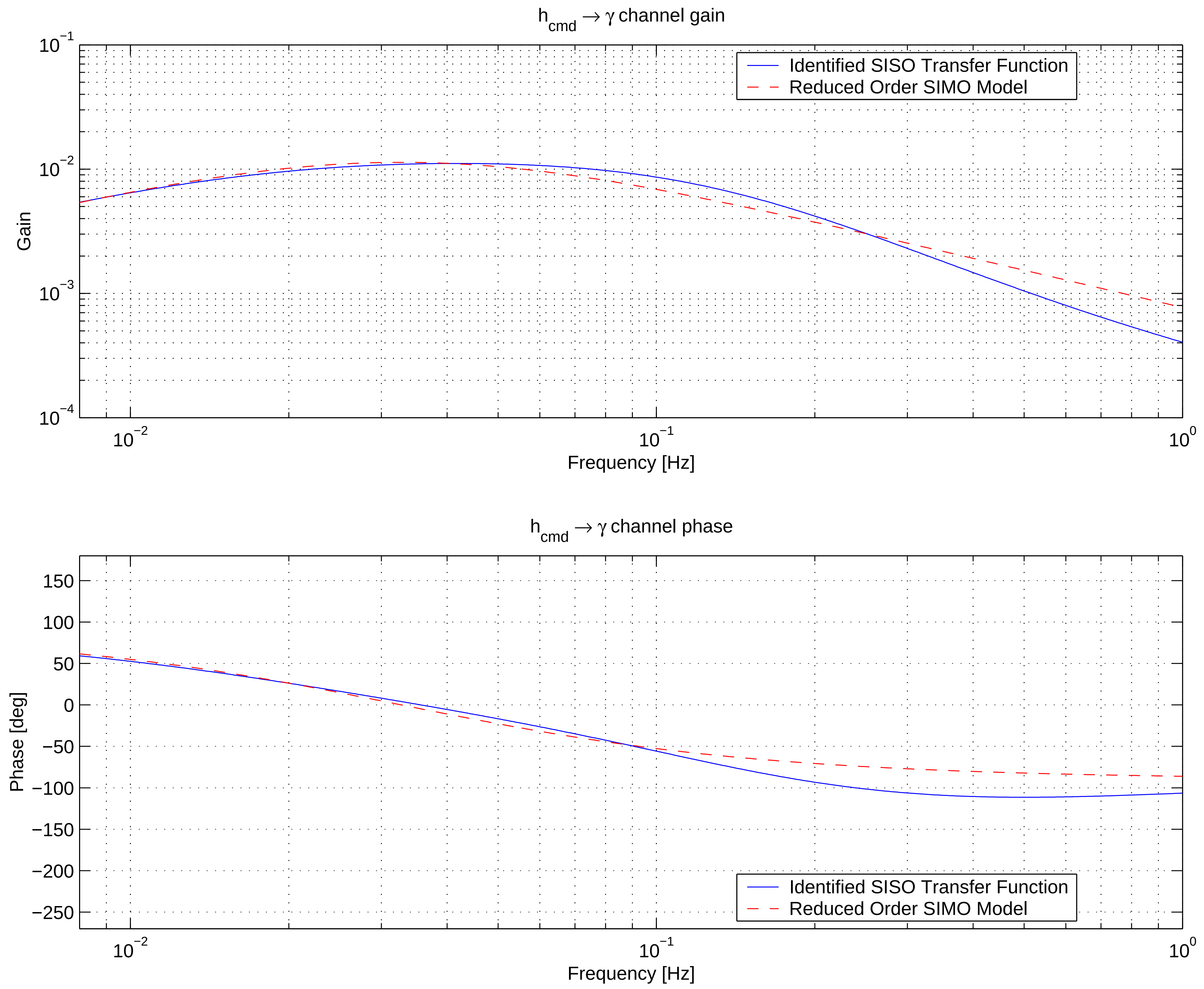


Figure 3.15: Frequency domain comparison between the reduced order h_{cmd} input SIMO model and the identified SISO $h_{cmd} \rightarrow \gamma$ model

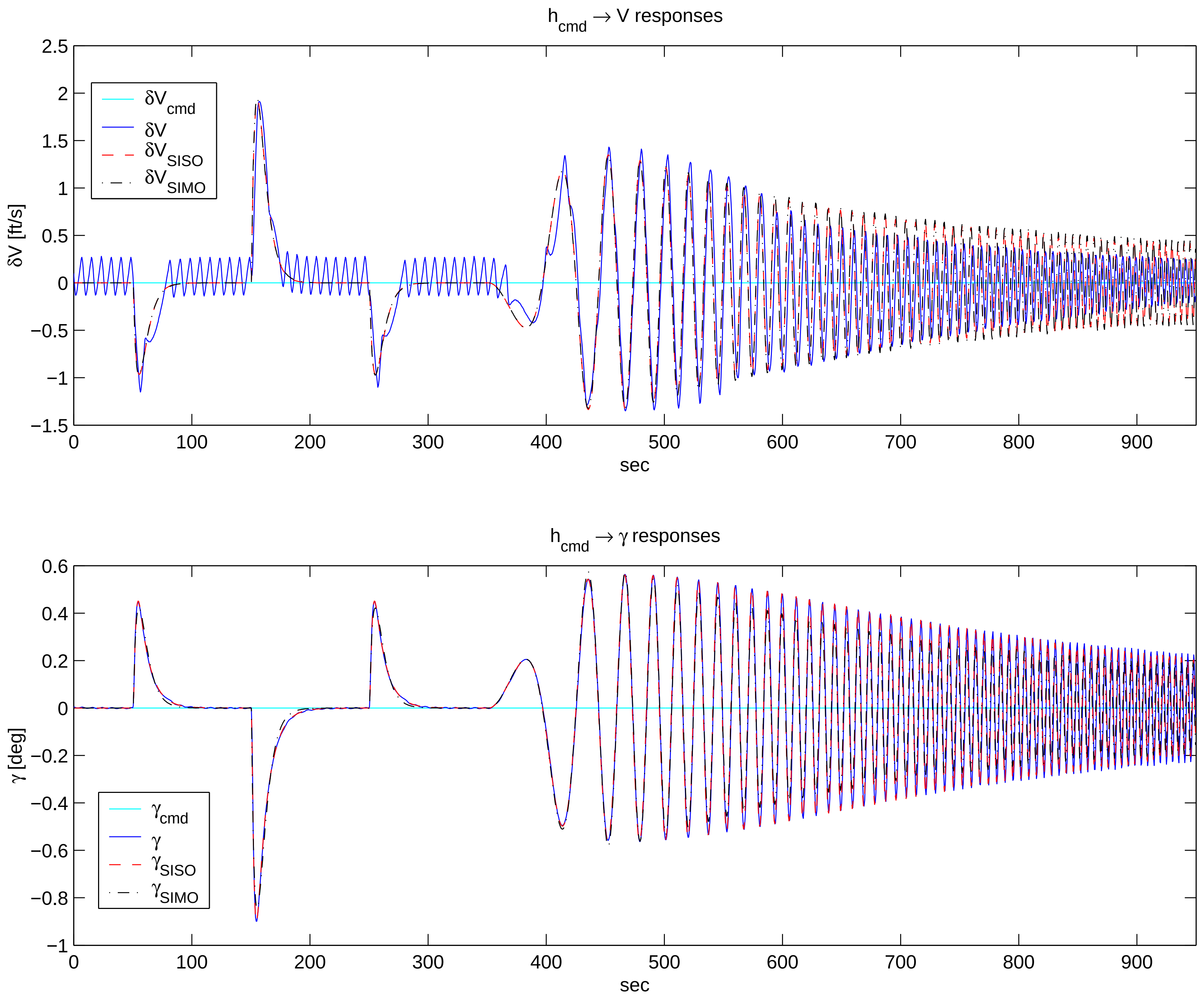


Figure 3.16: Time domain comparison between the identified SISO models, the reduced order SIMO system and DemoSim responses for the h_{cmd} input channel

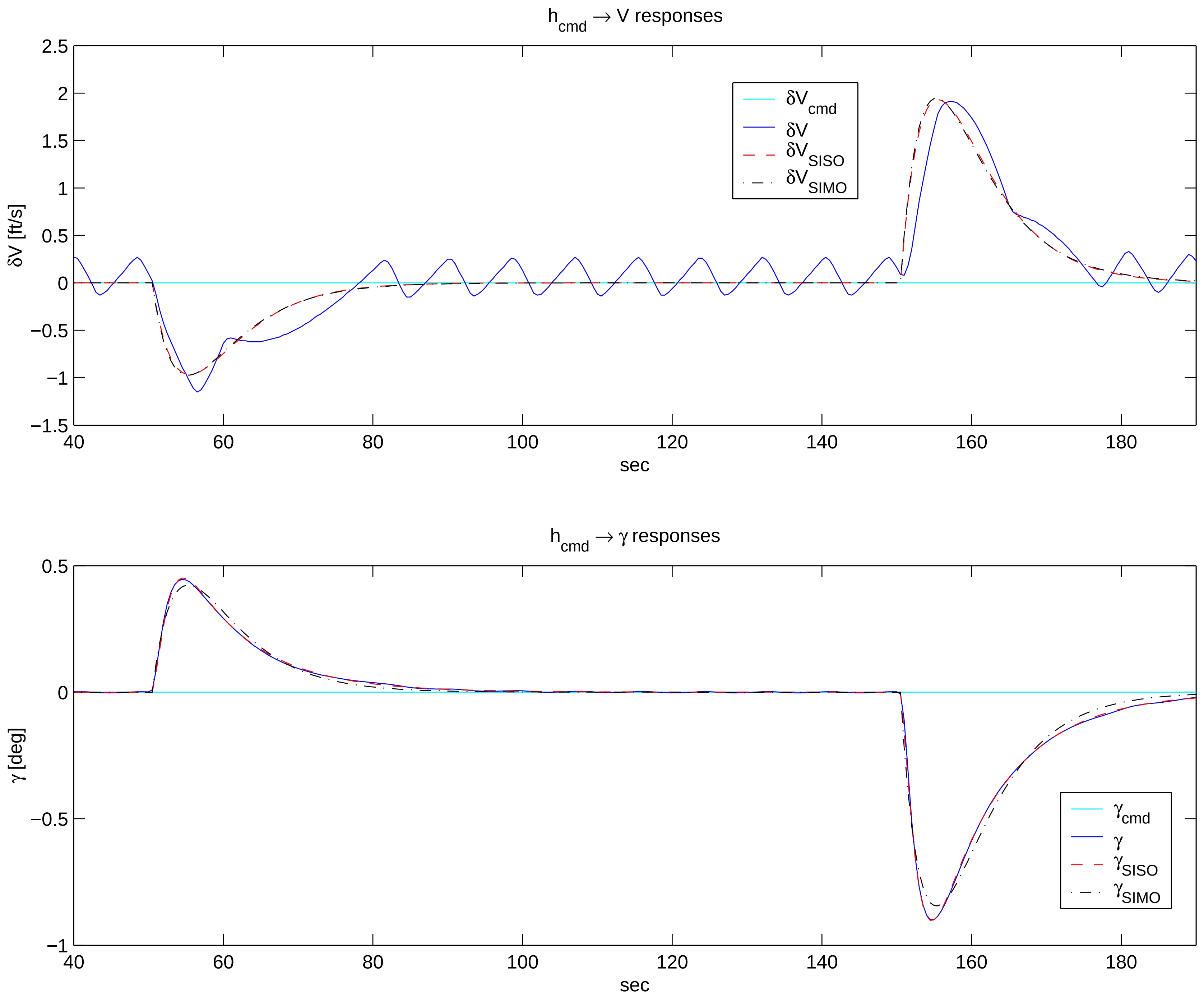


Figure 3.17: Enlarged portions of the time domain comparison shown in Fig. 3.16 for the h_{cmd} input channel

3.2.6.2 $V_{cmd} \rightarrow \gamma$ model

The fourth order OE identified model:

$$\frac{3.837\text{E}{-04}z^{-2} - 7.677\text{E}{-04}z^{-3} + 3.84\text{E}{-04}z^{-4}}{1 - 3.742z^{-1} + 5.276z^{-2} - 3.323z^{-3} + 0.7888z^{-4}}$$

Poles and zeros of the corresponding continuous-time model obtained using Zero Order Hold:

$V_{cmd} \rightarrow \gamma$ poles

real	imaginary	frequency	damping
−8.7204E−02	6.4796E−02	1.0864E−01	8.0268E−01
−8.7204E−02	−6.4796E−02	1.0864E−01	8.0268E−01
−1.5006E−01	−3.4132E−01	3.7284E−01	4.0246E−01
−1.5006E−01	3.4132E−01	3.7284E−01	4.0246E−01

$V_{cmd} \rightarrow \gamma$ zeros

real	imaginary	frequency	damping
−6.4883E−03	0.0000E+00	6.4883E−03	1.0000E+00
7.7638E−03	0.0000E+00	7.7638E−03	−1.0000E+00
3.8307E+00	0.0000E+00	3.8307E+00	−1.0000E+00

3.2.6.3 $\dot{\chi}_{cmd} \rightarrow \chi$ model

The second order OE identified model:

$$\frac{3.304z^{-2}}{1 - 1.909z^{-1} + 0.9085z^{-2}}$$

Poles and zeros of the corresponding continuous-time model obtained using Zero Order Hold:

$\dot{\chi}_{cmd} \rightarrow \chi$ poles

real	imaginary	frequency	damping
−1.9191E−01	0.0000E+00	1.9191E−01	1.0000E+00
8.1735E−05	0.0000E+00	8.1735E−05	−1.0000E+00

$\dot{\chi}_{cmd} \rightarrow \chi$ zeros

real	imaginary	frequency	damping
3.9371E+00	0.0000E+00	3.9371E+00	−1.0000E+00

3.2.6.4 $h_{cmd} \rightarrow V$ model

The second order identified model:

$$\frac{-0.0105s}{s^2 + 0.4s + 0.04}$$

Poles and zeros of the second order continuous-time model:

$h_{cmd} \rightarrow V$ poles

real	imaginary	frequency	damping
−2.0000E−01	−2.3853E−09	2.0000E−01	1.0000E+00
−2.0000E−01	2.3853E−09	2.0000E−01	1.0000E+00

$h_{cmd} \rightarrow V$ zeros

real	imaginary	frequency	damping
0.0000E+00	0.0000E+00	0.0000E+00	N/A

3.2.6.5 $h_{cmd} \rightarrow \gamma$ model

The third order OE identified model:

$$\frac{1.534\text{E}{-03}z^{-1} - 1.658\text{E}{-03}z^{-2} + 1.239\text{E}{-04}z^{-3}}{1 - 2.312z^{-1} + 1.769z^{-2} - 0.4512z^{-3}}$$

Poles and zeros of the corresponding continuous-time model obtained using Zero Order Hold:

$h_{cmd} \rightarrow \gamma$ poles

real	imaginary	frequency	damping
−1.0868E−01	0.0000E+00	1.0868E−01	1.0000E+00
−7.4155E−01	−3.0481E−01	8.0175E−01	9.2491E−01
−7.4155E−01	3.0481E−01	8.0175E−01	9.2491E−01

$h_{cmd} \rightarrow \gamma$ zeros

real	imaginary	frequency	damping
−3.7701E+00	0.0000E+00	3.7701E+00	1.0000E+00
−3.0046E−05	0.0000E+00	3.0046E−05	1.0000E+00

3.2.6.6 Reduced order SIMO model of the V_{cmd} input channel

The fourth order SIMO model has the following state matrices:

$$A = \begin{bmatrix} -6.998\text{E}{-01} & -2.617\text{E}{-01} & -4.338\text{E}{-02} & -2.283\text{E}{-03} \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \quad B = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix},$$

$$C = \begin{bmatrix} 6.367\text{E}{-02} & 8.935\text{E}{-02} & 3.507\text{E}{-02} & 2.41\text{E}{-03} \\ 2.575\text{E}{-04} & 2.575\text{E}{-03} & -4.279\text{E}{-06} & -1.214\text{E}{-07} \end{bmatrix}, \quad D = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

Poles of the fourth order continuous-time SIMO model:

$V_{cmd} \rightarrow \begin{bmatrix} V & \gamma \end{bmatrix}^T$ poles

real	imaginary	frequency	damping
−9.5568E−02	0.0000E+00	9.5568E−02	1.0000E+00
−1.9101E−01	0.0000E+00	1.9101E−01	1.0000E+00
−2.0660E−01	−2.8701E−01	3.5364E−01	5.8422E−01
−2.0660E−01	2.8701E−01	3.5364E−01	5.8422E−01

The model has no transmission zeros.

3.2.6.7 Reduced order SIMO model of the h_{cmd} input channel

$$A = \begin{bmatrix} -4.276\text{E}{-01} & -4.231\text{E}{-02} \\ 1 & 0 \end{bmatrix}, \quad B = \begin{bmatrix} 1 \\ 0 \end{bmatrix},$$

$$C = \begin{bmatrix} -1.116\text{E}{-02} & -1.9\text{E}{-19} \\ 4.85\text{E}{-03} & 1.514\text{E}{-07} \end{bmatrix}, \quad D = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

Poles of the second order continuous-time SIMO model:

$h_{cmd} \rightarrow \begin{bmatrix} V & \gamma \end{bmatrix}^T$ poles

real	imaginary	frequency	damping
−2.7207E−01	0.0000E+00	2.7207E−01	1.0000E+00
−1.5553E−01	0.0000E+00	1.5553E−01	1.0000E+00

The model has no transmission zeros.

3.3 Identification for FDI

The SISO LTI model used for the design of a fault detection filter was obtained using a process different from what was presented in the previous section. Identification was based on DemoSim simulation data in this case as well, although a faster sampling time was chosen. The model is described in Chapter 6 along with the techniques used for identification.

3.4 Characterization of flight envelope constraints

This section describes how the approximate nonlinear flight envelope limits were characterized and represented by a collection of linear constraints.

The most important constraints to be characterized and enforced by any chosen control approach were found to be the following:

- Flight envelope limits in terms of altitude and airspeed.
- Limits on maximum bank angle.
- Minimum and maximum vertical and longitudinal acceleration limits.
- Any other important limitations induced by the autopilot in DemoSim.

Figure 3.18 shows the flight envelope of the T-33 aircraft augmented with the UCAV avionics and autopilot. The actual flight envelope limits were restricted to the shaded area for the purpose of the flight test demonstration.

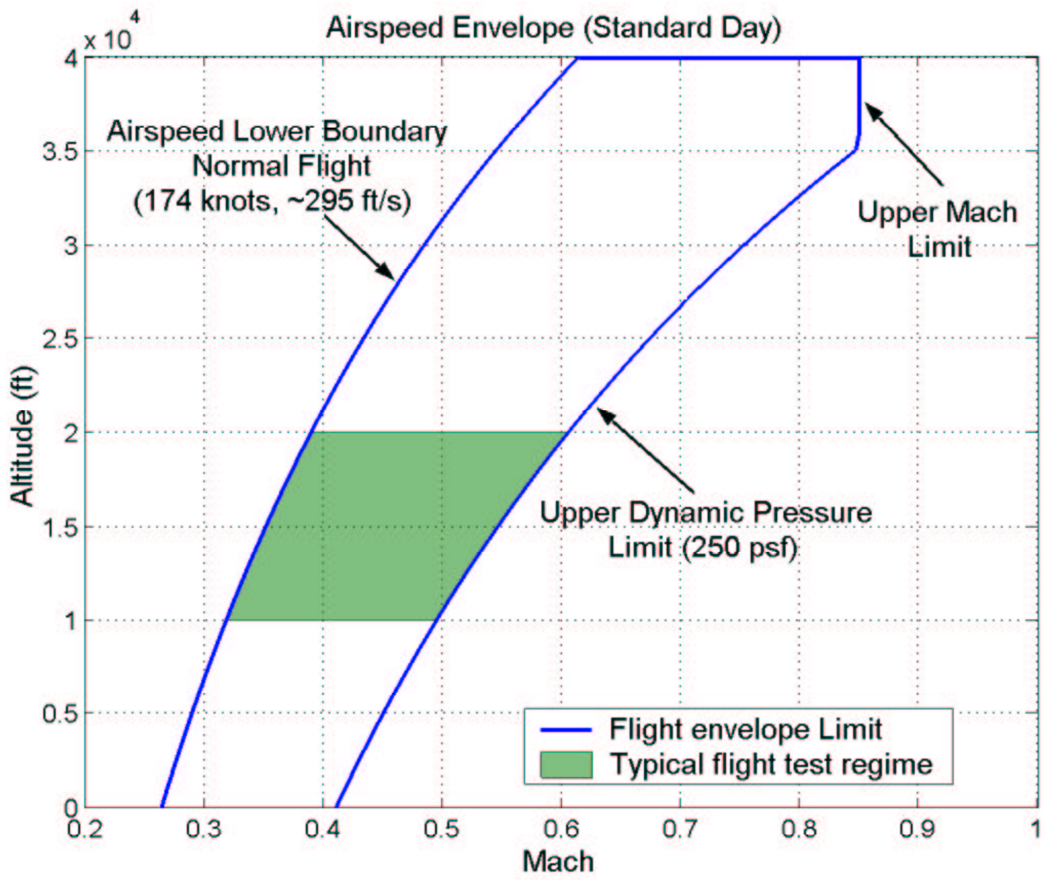


Figure 3.18: T-33/UCAV airspeed flight envelope in terms of altitude and Mach.

These flight regime constraints were respected by choosing a reference trajectory that lies between the minimum and maximum altitude limits. The constraints on minimum and maximum velocity were imposed by limiting the allowable deviations (± 100 ft/s) from the trim velocity value (approx. 500 ft/s). These limits were incorporated into the controller in terms of ground speed, whereas the actual limits were on true airspeed. (This caused issues given the very windy environment at Edwards AFB, as described in Section 8.3).

The main objective of characterizing the remaining three types of constraints was to arrive at formulas that could be approximated with linear expressions of available DemoSim model output variables and their derivatives. Consider the transformation of accelerations from `BODY` to `EARTH` coordinate frame given by the following expression

$$\begin{bmatrix} \dot{V} + g\sin\gamma \\ V\dot{\chi}\cos\gamma \\ -V\dot{\gamma} - g\cos\gamma \end{bmatrix} = \underbrace{\begin{bmatrix} \cos\alpha\cos\beta & \sin\beta & \sin\alpha\cos\beta \\ \sin\alpha\sin\mu - \cos\alpha\sin\beta\cos\mu & \cos\beta\cos\mu & -\cos\alpha\sin\mu - \sin\alpha\sin\beta\cos\mu \\ -\sin\alpha\cos\mu - \cos\alpha\sin\beta\sin\mu & \cos\beta\sin\mu & \cos\alpha\cos\mu - \sin\alpha\sin\beta\sin\mu \end{bmatrix}}_{T_{\mathtt{B}\to\mathtt{E}}} \begin{bmatrix} n_x g \\ n_y g \\ n_z g \end{bmatrix} \tag{3.4}$$

where the transformation matrix $T_{\mathtt{B}\to\mathtt{E}}$ is based on the angle-of-attack α, sideslip angle β and bank angle μ variables. The `BODY`-axis accelerations are denoted by n_x, n_y, n_z, whereas the velocity V, heading angle χ, and flight path angle γ variables are defined in `EARTH`-axis. The gravitational constant is denoted by g. Since the transformation matrix $T_{\mathtt{B}\to\mathtt{E}}$ is orthonormal, we can write equation (3.4) equivalently as

$$\begin{bmatrix} n_x g \\ n_y g \\ n_z g \end{bmatrix} = T^T_{\mathtt{B}\to\mathtt{E}} \begin{bmatrix} \dot{V} + g\sin\gamma \\ V\dot{\chi}\cos\gamma \\ -V\dot{\gamma} - g\cos\gamma \end{bmatrix} \tag{3.5}$$

Based on the limited information that was available about the autopilot incorporated into DemoSim, the following assumptions were reasonable to make

- Angle-of-attack can be considered zero $\alpha \approx 0$.
- The aircraft performs coordinated turns, therefore the side force and sideslip angles are approximately zero $n_y \approx 0, \beta \approx 0$. This means also that in level flight ($\gamma = 0$) we have

$$\frac{V\dot{\chi}}{g} = \tan\mu \tag{3.6}$$

Using these assumptions and the equations in (3.5) we obtain the following three equations that represent different types of constraints

1. Longitudinal acceleration constraint

$$n_x g = \dot{V} + g\sin\gamma \tag{3.7}$$

2. Turn constraint equation

$$0 = \cos\mu \cdot V\dot{\chi}\cos\gamma - \sin\mu\,(V\dot{\gamma} + g\cos\gamma) \tag{3.8}$$

3. Vertical acceleration constraint equation

$$n_z g = -\sin\mu \cdot V\dot{\chi}\cos\gamma - \cos\mu\,(V\dot{\gamma} + g\cos\gamma) \tag{3.9}$$

If we further assume that $|\mu| < \pi/2$, i.e. $\cos\mu > 0$ and $\dot{\gamma} > -g\cos\gamma/V$, (e.g. $\gamma < 10$ deg, $V = 200$ m/s $\longrightarrow \dot{\gamma} > -3$ deg/s), then using equation (3.8) we can express the sine and cosine of the bank angle μ using the following formulas

$$\cos\mu = \frac{a}{\sqrt{a^2+b^2}}, \qquad \sin\mu = \frac{b}{\sqrt{a^2+b^2}}, \tag{3.10}$$

$$a = \frac{V\dot{\gamma}}{g\cos\gamma} + 1, \qquad b = \frac{V\dot{\chi}}{g} \tag{3.11}$$

Substituting $\cos\mu$ and $\sin\mu$ into equation (3.9) we obtain

$$\sqrt{\left(\frac{V\dot{\chi}}{g}\right)^2 + \left(\frac{V\dot{\gamma}}{g\cos\gamma} + 1\right)^2} = -\frac{n_z}{\cos\gamma} \tag{3.12}$$

By studying the behavior of DemoSim and running extensive simulations, the following set of limitations on velocity, vertical acceleration and bank angle were determined

$$\begin{aligned} V_{max} &= 630 \text{ ft/s}, & V_{min} &= 340 \text{ ft/s}, \\ n_{z,max} &= 1.4 \text{ g}, & n_{z,min} &= 0.6 \text{ g}, \\ \mu_{max} &= 32 \text{ deg}, & \mu_{min} &= -32 \text{ deg}. \end{aligned} \tag{3.13}$$

Flight path angle γ was also limited to approximately ± 2 deg.

With these limits in mind and using the small angle approximation $\cos\gamma \approx 1$, we arrive at the nonlinear expressions that serve as the basis for characterizing the important constraints to be enforced by the controller. Using (3.6), (3.8), (3.12) and the limits in (3.13) we get the following inequalities in the three variables of $V, \dot{\chi}$ and $\dot{\gamma}$

$$V_{min} < V < V_{max}, \tag{3.14a}$$

$$\frac{g\tan\mu_{min}}{V_{min}} < \dot{\chi} < \frac{g\tan\mu_{max}}{V_{min}}, \tag{3.14b}$$

$$(n_{z,min}g)^2 < (V\dot{\chi})^2 + (V\dot{\gamma}+g)^2 < (n_{z,max}g)^2, \tag{3.14c}$$

$$\tan\mu_{min} < \frac{V\dot{\chi}}{V\dot{\gamma}+g} < \tan\mu_{max}. \tag{3.14d}$$

(The constraint on longitudinal acceleration (n_x) using equation (3.7) will not be used to formulate output constraints, instead a limit on the change of velocity command is the only way this aspect of the flight characteristics is incorporated in the controller design.)

Figure 3.19 represents the different types of constraints that were considered. Constraint type 1 represents bank angle μ limits based on the inequalities in (3.14d) and constraint type 2 shows vertical acceleration limitations imposed by (3.14c). Constraint type 3 was discovered after running extensive simulations with DemoSim. We could not explain this phenomenon using mathematical formulas, however its limiting effect could be approximated with straight lines very well in the given coordinate frame. This limitation is most probably induced by the autopilot in situations when the aircraft is trying to pitch down (n_z approaches its minimum), while turning hard (large $\dot{\chi}$) at the same time.

Since the coordinate axes each depend on two variables and both axes share velocity as one of the variables, the shape of nonlinear constraints can be visualized in a three-dimensional plot. The individual axes in this case are the three variables $V, \dot{\chi}$ and $\dot{\gamma}$. This three-dimensional characterization of constraints is illustrated in Figure 3.20. The "front" and "back" facets of the shape that

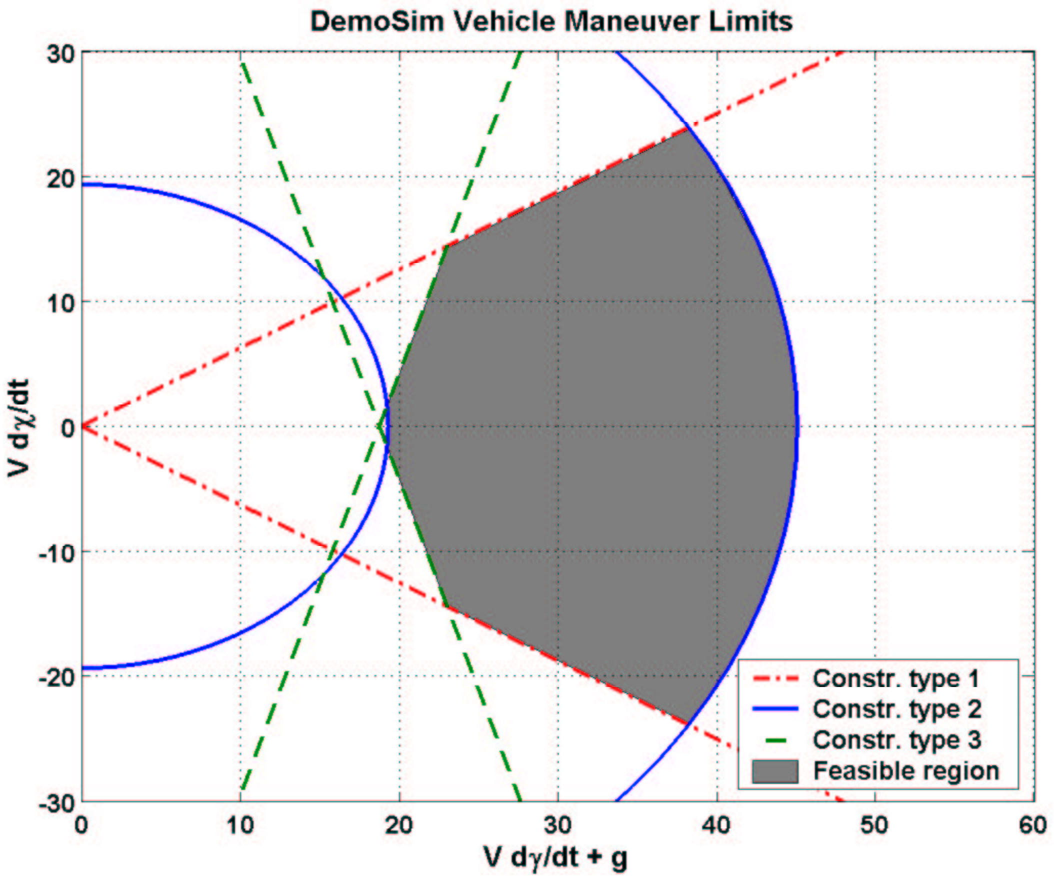


Figure 3.19: Characterization of nonlinear constraints (2D).

represent V_{min} and V_{max} limits are not shown in this and any subsequent plots for better visibility. Notice that the $\dot{\chi}_{min}$ and $\dot{\chi}_{max}$ limits (3.14b) become "active" in forming the boundary of this non-convex region near the edges at lower velocity and positive $\dot{\gamma}$ values.

The objective of representing these nonlinear constraints by linear expressions using the aforementioned three variables was achieved by finding a polyhedron that is an inner approximation of the non-convex feasible region shown in Figure 3.20. The resulting polytope, which is constructed from a small number of halfspaces compromising approximation accuracy is depicted in Figure 3.21. (Note that the bounding planes associated with V_{min}, V_{max} limits are not displayed. Hyperplanes representing $\dot{\chi}_{min}$ and $\dot{\chi}_{max}$ limits are also not shown, since other facets of the polytope rendered these redundant.) The number of halfspaces that characterize the polytope and represent linear constraints in terms of the three variables were kept small to allow the complexity of the problem to be maintained at a manageable size.

Using the halfspace representation of the inner polyhedral approximation to the non-convex constraint set, the original nonlinear maneuver limits can be characterized by the following linear inequality

$$A_z \begin{bmatrix} V \\ \dot{\chi} \\ \dot{\gamma} \end{bmatrix} \leq b_z \tag{3.15}$$

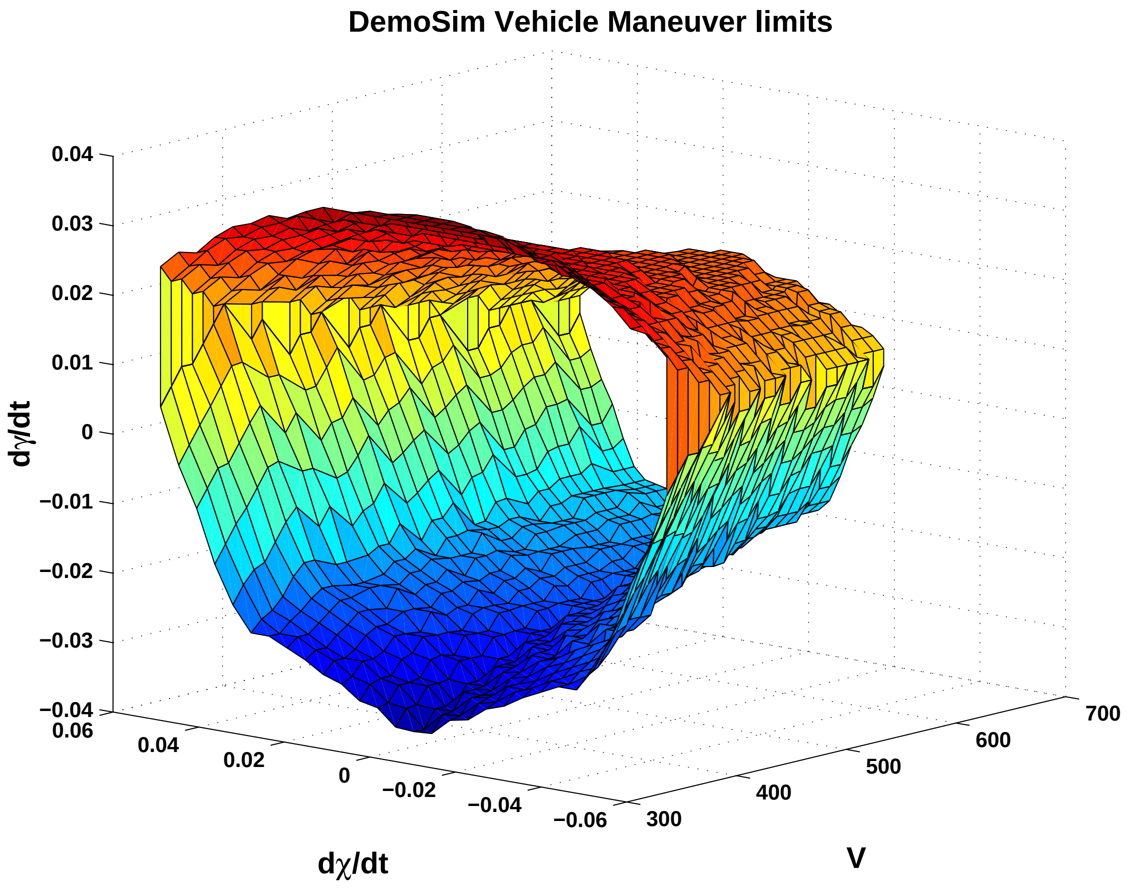


Figure 3.20: Characterization of nonlinear constraints (3D).

where $A_z \in \mathbb{R}^{10\times 3}$ and $b_z \in \mathbb{R}^{10}$. This inequality represents ten halfspace constraints, six of which are depicted in Figure 3.21. The remaining four are the maximum and minimum limits on velocity V and turn rate $\dot{\chi}$.

Acknowledgement

The formulas and derivation of the approximate nonlinear flight envelope constraints were obtained from George Papageorgiou at Honeywell Labs. His help and discussions on the subject is greatly appreciated.

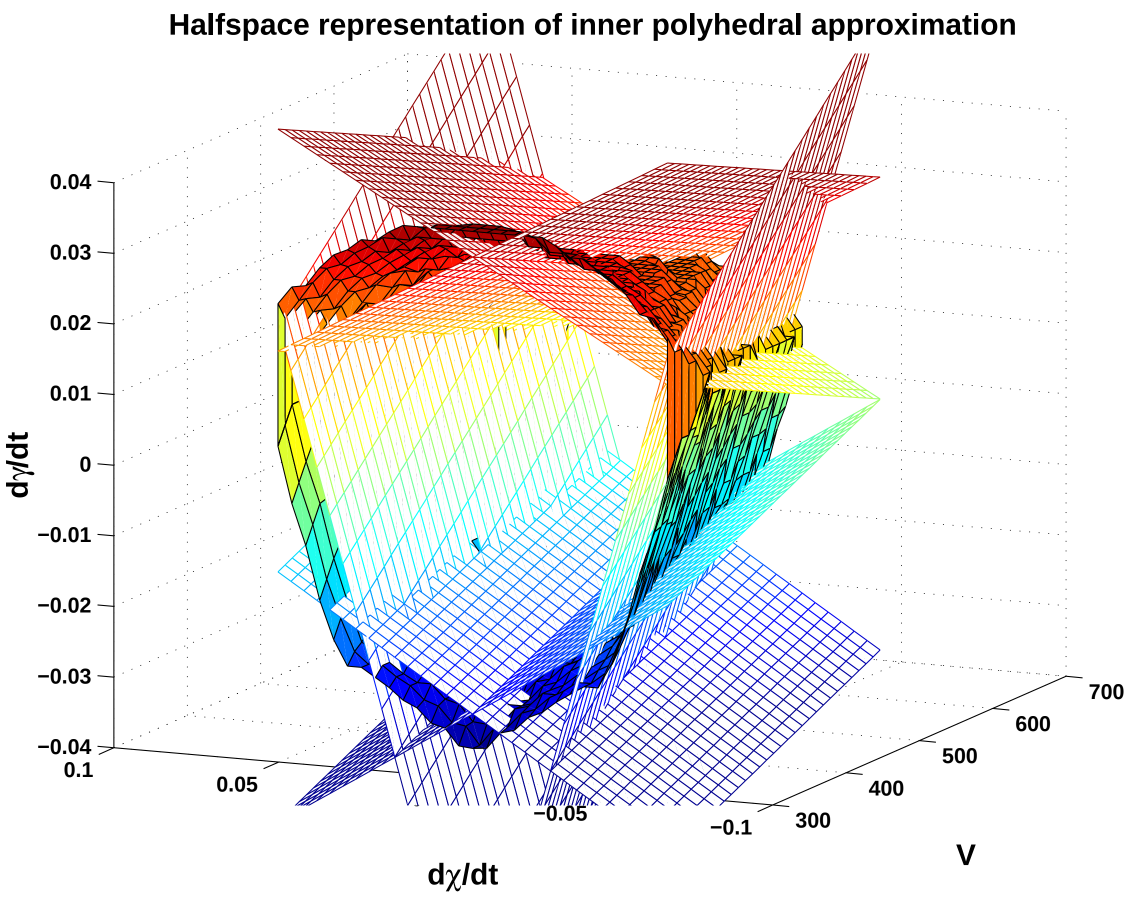


Figure 3.21: Inner polyhedral approximation of nonlinear constraints (3D).

Chapter 4

RHC API

4.1 Timing Architecture

Each time frame is divided into 3 intervals: `Pre`, `Opt`, and `Post`. These intervals allocate computation time to different parts of the RHC problem formulation and solution. From user supplied data (see Section 4.3), the RHC component can formulate, solve, and compute the proper control action to be taken.

The descriptions of `Pre`, `Opt` and `Post` are summarized in the following table.

Frame	Task Type	Description
`Pre`	Hard Real Time	Formulates the RHC problem from user supplied data. Computes candidate control action.
`Opt`	Anytime	Solve constrained optimization problem. If solution is infeasible, reformulate and solve relaxed problem.
`Post`	Hard Real Time	Process solution of RHC problem and return desired control action.

The hard-realtime function `Pre` executes 2 primary operations. It

- calls the warm-start function, which has two purposes:
  - initialize the decision variable v, based on data from the previous frame.
  - compute a candidate control action u_0.
- forms the linearly constrained, quadratic program using H, x^{est}, $d^{\text{est}}_{k:k+H-1}$, and the dynamic, constraint and cost matrices.

Although problem-dependent, the user can give worst-case (problem dependent) execution times for both of these steps, so these can be done in hard realtime fashion.

The anytime function `Opt` does one operation, calling LSSOL to solve the QP.

- If LSSOL returns infeasible, then the relaxed problem (see Section 4.4) is formed, and LSSOL is called again.
- `Opt` is terminated by the Scheduler. Cleanup will be a hard realtime task.

Finally, the hard real-time function `Post` is called.

- `Post` converts the most recent value of v into a control action, u_0^v.
- `Post` then makes a decision of which control action to use, u_0 or u_0^v. This value becomes the RHC component's output at the end of the frame.

The concepts described above can be found in the source code within the RHC_Core:

- In `LSSOL_RHC_Interface` the `PreOptimize()` method formulates the standard RHC problem formulation given user supplied data.
- In `LSSOL_RHC_Interface`, the `Optimize()` method attempts to solve the constrained optimizatino problem. If the solution is infeasible, the problem is reformulated and the relaxed problem is solved.
- In `LSSOL_RHC_Interface` the `PostOptimize()` method receives the solution from the `Optimize` frame. The solution is interpreted and a control action is computed.

4.2 Modification of LSSOL

The quadratic program solver LSSOL has been modified in order to allow recovery of intermediate iterates following premature termination from other processes. Section 10 of the LSSOL license from SBSI allows for modifications to the software.

Four additional arguments have been added to the call to LSSOL. These arguments are summarized in the following table.

Additional Arguments in LSSOL Modification		
Name	Dimension/Class	Notes
`iterflagstart`	2×1 `integer`	Elements are iteration numbers
`iterflagend`	2×1 `integer`	Elements are either 0 (iteration incomplete) or 1 (iteration complete)
`decisionvar1`	$(n_v + n_\epsilon) \times 1$ `double`	Solution pertaining to odd iteration numbers
`decisionvar2`	$(n_v + n_\epsilon) \times 1$ `double`	Solution pertaining to even iteration numbers

Upon completion of an odd-numbered iteration of LSSOL, the following sequence of operations occurs:

O1 The iteration number is written in the 1st element of `iterflagstart`.
O2 The decision variable is written into memory starting at `decisionvar1`
O3 The 1st element of `iterflagend` is set to 1
O4 The 2nd element of `iterflagend` is set to 0

Upon completion of an even numbered iteration of LSSOL, the following sequence of operations occurs:

E1 The iteration number is written in the 2nd element of `iterflagstart`.
E2 The decision variable is written into memory starting at `decisionvar2`
E3 The 2nd element of `iterflagend` is set to 1
E4 The 1st element of `iterflagend` is set to 0

Under the modest assumption that LSSOL completes one iteration before being interrupted, it is always possible to retrieve an uncorrupted decision variable value. Specifically

- If both entries of `iterflagend` are 0, then LSSOL was unable to complete the first phase, which consisted of
  - complete a single iteration,

  - write 1 (the iteration number) to 1st element of `iterflagstart`
  - write the decision variable to `decisionvar1`
  - write a 1 to 1st element of `iterflagend`

  In this case, the value of `decisionvar1` and `decisionvar2` must be considered suspect. An alternate solution must be used. One strategy, which we employ, is to use the control action from the previous time step. In our application, the decision variables represent deviations from the previous control action, hence a decision variable value of identically zero represents this action. Therefore, if `decisionvar1` is initialized to 0, it can be thought of as a valid value in this situation.

- If both entries of `iterflagend` are 1, then LSSOL was terminated between steps **O3/O4** or **E3/E4**. Both decision variables are uncorrupted. The most recent value can be determined by examining the `iterflagstart` values.
  - If the 1st entry of `iterflagstart` is larger than the 2nd entry of `iterflagstart`, then the value in `decisionvar1` is the most recent iterate from LSSOL.
  - If the 2nd entry of `iterflagstart` is larger than the 1st entry of `iterflagstart`, then the value in `decisionvar2` is the most recent iterate from LSSOL.
- If the 1st entry of `iterflagend` is 1 and the 2nd entry is 0, then LSSOL was terminated sometime after **O4**, but before completing **E3**. It is guaranteed that `decisionvar1` is an uncorrupted iterate.
- If the 1st entry of `iterflagend` is 0 and the 2nd entry is 1, then LSSOL was terminated sometime after **E4**, but before completing **O3**. It is guaranteed that `decisionvar2` is an uncorrupted iterate.

The concepts described above can be found in the source code within the RHC_Core:

- In `LSSOL_Optimizer.h` the `GetRecentIter()` method shows how the valid LSSOL iteration result is read.
- The modified LSSOL code is in `lssolsubs.c`. Search for the string “KEN” to find the modifications, which include function definitions, and about 30 lines of code implementing the readout logic.
- In `LSSOL_Optimizer.cpp` contains the actual call to `lssol2_` which is the modified version of LSSOL.

4.3 RHC Problem Formulation for Each Time Frame

The receding horizon control problem formulates and solves a quadratic optimization problem during each time frame. In the following subsections, we describe the standard problem formulation and its relaxed counterpart. We begin with two tables of notation whose meaning will be described in subsequent sections.

4.3.1 Notation and symbol definitions

Dimensions, Measurements, Dynamic Model Parameters		
Variable	Dimension/Class	Meaning
n_x	scalar `int`	State Dimension
n_u	scalar `int`	Input Dimension
n_y	scalar `int`	Output Dimension
n_d	scalar `int`	Known Signal Dimension
n_v	scalar `int`	Decision variable Dimension
H	scalar `int`	Horizon Length
x_0	$n_x \times 1$ `double`	Initial Condition
$d_{0:H-1}$	$n_d \times H$ `double`	preview of Known Signal
A	$n_x \times n_x$ `double`	Dynamic Model Parameters
B	$n_x \times n_u$ `double`	Dynamic Model Parameters
E	$n_x \times n_d$ `double`	Dynamic Model Parameters
Q	$n_x \times n_x$ `double`	Symmetric State Step Cost
R	$n_u \times n_u$ `double`	Symmetric Input Step Cost
M	$n_y \times n_y$ `double`	Symmetric Known Signal Step Cost
C	$n_y \times n_x$ `double`	Cost-Related Matrix for State
G	$n_y \times n_d$ `double`	Cost-Related Matrix for Signal
F	$n_v \times n_v$ `double`	direct quadratic Cost
K	$n_v \times 1$ `double`	direct linear Cost
Φ	$n_x \times n_x$ `double`	Symmetric State Terminal Cost
$\mathcal{U}$	$n_u \times n_v$ `double`	matrix for map from v to u

Constraints		
$a_{x,box}$	$n_x \times 1$ `double`	Lower Bound State Box Constraint
$b_{x,box}$	$n_x \times 1$ `double`	Upper Bound State Box Constraint
$a_{u,box}$	$n_u \times 1$ `double`	Lower Bound Input Box Constraint
$b_{u,box}$	$n_u \times 1$ `double`	Upper Bound Input Box Constraint
n_{L_x}	scalar `int`	# of Linear State Constraints
a_x	$n_{L_x} \times 1$ `double`	Lower Bound State Constraint
b_x	$n_{L_x} \times 1$ `double`	Upper Bound State Constraint
L_x	$n_{L_x} \times n_x$ `double`	State Constraint Matrix
n_{L_u}	scalar `int`	# of Linear Input Constraints
a_u	$n_{L_u} \times 1$ `double`	Lower Bound Input Constraint
b_u	$n_{L_u} \times 1$ `double`	Upper Bound Input Constraint
L_u	$n_{L_u} \times n_u$ `double`	Input Constraint Matrix
$n_{L_{G_u}}$	scalar `int`	# of Linear General u Constraints
a_{G_u}	$n_{L_{G_u}} \times 1$ `double`	Lower Bound General u Constraint
b_{G_u}	$n_{L_{G_u}} \times 1$ `double`	Upper Bound General u Constraint
L_{G_u}	$n_{L_{G_u}} \times (H \times (n_x + n_u))$ `double`	General u Constraint Matrix
$n_{L_{G_v}}$	scalar `int`	# of Linear General v Constraints
a_{G_v}	$n_{L_{G_v}} \times 1$ `double`	Lower Bound General v Constraint
b_{G_v}	$n_{L_{G_v}} \times 1$ `double`	Upper Bound General v Constraint
L_{G_v}	$n_{L_{G_v}} \times (H \times n_x + n_v)$ `double`	General v Constraint Matrix

4.3.2 Standard Formulation

In this section, we describe the constrained convex quadratic optimization problem that is formulated and solved at each time frame. Given the parameters described in section 4.3.1, consider the following minimization problem.

$$\min_{v} \sum_{k=0}^{H-1} x_k^T Q x_k + u_k^T R u_k + [Cx_k - Gd_k]^T M [Cx_k - Gd_k] + v^T F v + K^T v \; + \; x_H^T \Phi x_H$$

subject to

$$\begin{array}{rcll} a_{x,box} \leq & x_k & \leq b_{x,box} & k = 1, \ldots, H \\ a_x \leq & L_x x_k & \leq b_x & k = 1, \ldots, H \\ a_{u,box} \leq & u_k & \leq b_{u,box} & k = 0, \ldots, H-1 \\ a_u \leq & L_u u_k & \leq b_u & k = 0, \ldots, H-1 \\ a_{G_u} \leq & L_{G_u} \begin{bmatrix} x_{1:H} \\ u_{0:H-1} \end{bmatrix} & \leq b_{G_u} & \\ a_{G_v} \leq & L_{G_v} \begin{bmatrix} x_{1:H} \\ v \end{bmatrix} & \leq b_{G_v} & \end{array}$$

Here, the linear dynamics of the system are specified by the matrices A, B, and E such that

$$x_{k+1} = Ax_k + Bu_x + Ed_k \qquad \text{for } k = 0, \ldots, H-1$$

where the signal d represents both estimated disturbance and desired trajectories. The columns of $\mathcal{U}$ form a basis for the control input subspace. As a result, the control input signal u can then be written as linear combinations of the columns of $\mathcal{U}$.

$$u_{0:H-1} := \begin{bmatrix} u_0 \\ \vdots \\ u_{H-1} \end{bmatrix} = \mathcal{U} v$$

Discussion: The first four constraints are targeted towards typical operational constraints. The last two constraint types are general, allowing the user to specify general linear constraints on $x_1, x_2, \ldots, x_H, u_0, \ldots, u_{H-1}$ and on $x_1, x_2, \ldots, x_H, v$.

The subroutine `QPformulate` maps the system model and cost function matrices into the variables which parameterize the quadratic program, namely $(\bar{Q}, \bar{L}, \bar{Z}, \bar{a}, \bar{W}, \bar{b})$. The resulting problem can then be represented as a quadratic program of the form

$$\begin{array}{ll} \text{minimize} & \frac{1}{2} v^T \bar{Q} v + v^T \bar{L} + \bar{Z} \\ \text{subject to} & \bar{a} \leq \bar{W} v \leq \bar{b} \end{array}$$

Once formulated, the parameters $(\bar{Q}, \bar{L}, \bar{Z}, \bar{a}, \bar{W}, \bar{b})$ are passed to the quadratic program solver, LSSOL.

LSSOL will return a flag informing that the problem is infeasible. The subroutine `Reformulate` can be used to reformulate a relaxed problem with additional slack variables (see Section 4.4).

4.3.3 Argument list for `QPformulate`

The function prototype for `QPformulate()` is:

```
void QPformulate(double *x0pr, double *Apr, double *Bpr,
    double *Epr, double *Cpr, double *LXpr, double *LUpr,
    double *LGXUpr, double *LGXVpr, double *aUpr,
    double *bUpr, double *aLXpr, double *bLXpr,
    double *aLUpr, double *bLUpr, double *aXpr, double *bXpr,
    double *aLGXUpr, double *bLGXUpr, double *aLGXVpr,
    double *bLGXVpr, int rX, int ru, double *dpr, double *Qpr,
    double *Rpr, double *Mpr, double *Gpr, double *Phipr,
    double *Fpr, double *Kpr, double *Upr, int nx, int nu,
    int ny, int nd, int numDec, int allocV, int rowF, int rowK,
    int rLX, int rLU, int rLGxu, int rLGxv, int Hint,
    double *CostQuadpr, double *CostLpr, double *CostZpr,
    double *CCmatpr, int allocRowsCC, double *avecpr,
    double *bvecpr)
```

A detailed description of each argument follows:

`QPformulate`, Arguments 1-5, Initial condition, matrices in linear model

Name	Meaning	variable	# elements accessed starting from ptr
x0pr	Initial State	x_0	n_x
Apr	A-matrix	A	n_x^2
Bpr	B-matrix	B	$n_x n_u$
Epr	E-matrix	E	$n_x n_d$
Cpr	C-matrix	C	$n_y n_x$

`QPformulate`, Arguments 6-21, Constraint Data

Name	Meaning	variable	# elements accessed starting from ptr
LXpr	Linear map on x_k	L_x	$n_{L_x} n_x$
LUpr	Linear map on u_k	L_u	$n_{L_u} n_u$
LGXUpr	Linear map on x and u	L_{G_u}	$n_{L_{G_u}} H(n_x + n_u)$
LGXVpr	Linear map on x and v	L_{G_v}	$n_{L_{G_v}} H(n_x + n_v)$
aUpr	LowerBox on u_k	$a_{u,box}$	n_u or 0, see variable ru below
bUpr	UpperBox on u_k	$b_{u,box}$	n_u or 0, see variable ru below
aLXpr	Lower on $L_x x_k$	a_x	n_{L_x}
bLXpr	Upper on $L_x x_k$	b_x	n_{L_x}
aLUpr	Lower on $L_u u_k$	a_u	n_{L_u}
bLUpr	Upper on $L_u u_k$	b_u	n_{L_u}
aXpr	LowerBox on x_k	$a_{x,box}$	n_x or 0, see variable rX below
bXpr	UpperBox on x_k	$b_{x,box}$	n_x or 0, see variable rX below
aLGXUpr	Lower on $L_{G_u}[x;u]$	a_{G_u}	$n_{L_{G_u}}$
bLGXUpr	Upper on $L_{G_u}[x;u]$	b_{G_u}	$n_{L_{G_u}}$
aLGXVpr	Lower on $L_{G_v}[x;v]$	a_{G_v}	$n_{L_{G_v}}$
bLGXVpr	Upper on $L_{G_v}[x;v]$	b_{G_v}	$n_{L_{G_v}}$

`QPformulate`, Arguments 22-23, Constraint Data

Name	Notes
rX	should be 0 if no box constraints on x, otherwise should be n_x
ru	should be 0 if no box constraints on u, otherwise should be n_u

It is possible to count the number of linear constraints. Here, H is the optimization horizon, argument #45, still to be described. The total number of linear constraints T_{LC} is equal to

$$T_{LC} = H(rX + ru + n_{L_x} + n_{L_u}) + n_{L_{G_u}} + n_{L_{G_v}}$$

This enters into the sizes of some of the matrices to be defined.

`QPformulate`, Arguments 24-31, Cost Function Weights

Name	Meaning	variable	# elements accessed starting from ptr
dpr	d_k on $[0, H-1]$	d	$n_d H$
Qpr	Q-matrix	Q	n_x^2
Rpr	R-matrix	R	n_u^2
Mpr	M-matrix	M	n_y^2
Gpr	G-matrix	G	$n_y n_d$
Phipr	Phi-matrix	Φ	n_x^2
Fpr	F-matrix	F	n_v^2
Kpr	K-matrix	K	n_v

`QPformulate`, Argument 32, Basis for input signal

Name	Meaning	variable	# elements accessed starting from ptr
Upr	U-matrix	U	$H n_u n_v$

`QPformulate`, Arguments 33-37, Dimensions

Name	Meaning	variable	Notes
nx	# of States	n_x	
nu	# of Inputs	n_u	
ny	# of Outputs	n_y	
nd	# of KnownExternal	n_d	
numDec	# of Decision Variables	n_v	

Arguments 38-45, Constraint Data Dimensions

Name	Meaning	variable	Notes
allocV			numDec + # of constraint relaxation slack variables (see Section 4.4
rowF	# of rows of F		Extraneous, must be set to n_v
rowK	# of rows of K		Extraneous, must be set to n_v
rLX	# of rows of L_x	n_{L_x}	
rLU	# of rows of L_u	n_{L_u}	
rLGxu	# of rows of L_{G_u}	$n_{L_{G_u}}$	
rLGxv	# of rows of L_{G_v}	$n_{L_{G_v}}$	
Hint	Optimization Horizon	H	

Discussion: Memory for the following variables must be allocated before calling `QPformulate`. `QPformulate` fills these arrays, which are then ready for an immediate call to LSSOL to solve the quadratic program.

Arguments 46-52, Resultant data for QP solver			
Name	Meaning	variable	# of Allocated Elements
CostQuadpr	quadratic cost	$\bar{Q}$	allocV*allocV
CostLpr	linear cost	$\bar{L}$	allocV
CostZpr	constant cost	$\bar{Z}$	1
CCmatpr	Linear Constraint Matrix	$\bar{W}$	$2T_{LC}$*allocV
allocRowsCC	Rows Allocated for CCmatpr		Extraneous. Must be $2T_{LC}$
avecpr	Constraint Lower Bound	$\bar{a}$	$2T_{LC}$
bvecpr	Constraint Upper Bound	$\bar{b}$	$2T_{LC}$

The concepts described above can be found in the source code within the RHC_Core. Specifically, in `LSSOL_RHC_Interface`, `QPformulate()` is called by methods within `PreOptimize()`.

4.4 Constraint Relaxation

If the contrained problem is infeasible, the constraints must be relaxed and the problem solved again. The subroutine `Reformulate.c` does this is an automatic manner, using user-specified weights which define the relative "cost" of relaxing individual constraints.

Associated with each linear constraint bound (lower bounds $a_{x,box}, a_x, a_{u,box}, a_u, a_{G_u}, a_{G_v}$, and corresponding upper bounds) is an index vector. For example, consider a_x. Let $a_{Sx,idx}$ be a column vector of the same dimension, containing nonnegative integer values. These are referred to as the *soft indices.* Consider the i'th element, associated with the i'th row of the constraints, namely

$$(a_x)_i \leq (L_x x_k)_i \quad k = 1, \dots, H$$

The intepretation is as follows:

- If $a_{Sx,idx}(i) = 0$, then this constraint is not relaxed in the reformulation.
- If $a_{Sx,idx}(i) > 0$, then this constraint is relaxed in the reformulation. A constraint relaxation variable, $\epsilon_{a_{Sx,idx}(i)}$ is introduced, and the constraint is modified to

  $$-\epsilon_{a_{Sx,idx}(i)} + (a_x)_i \leq (L_x x_k)_i \quad k = 1, \dots, H$$

  which clearly relaxes the constraint.

There are analogous *soft index vectors* for each of the constraints. The values in these vectors are all nonnegative. Elements that are 0 correspond to constraints that are not relaxed in the reformulation. Positive entries indicate the constraint is relaxed in the reformulation, and the value of the positive entry is the index of the associated slack variable.

Assume that the positive values of the soft indices consist of the numbers $1, 2, \dots, n_\epsilon$, where various soft index entries may have the same value (meaning they are relaxed with the same slack variable).

Let ϵ denote the $n_\epsilon \times 1$ slack variable. Let $\rho \in \mathbf{R}^{n_\epsilon}$ be a specified weight vector with positive entries. If the original problem is infeasible, the relaxed problem is

$$\min_{v,\epsilon} \quad \rho^T \epsilon + \frac{1}{2} v^T \bar{Q} v + v^T \bar{L} + \bar{Z}$$

subject to the relaxed constraints.

At this point, we have introduced the notation in the tables below.

Dimensions, Measurements, Dynamic Model Parameters		
Variable	Dimension/Class	Meaning
n_ϵ	scalar `int`	Relaxation Variable Dimension
ρ	$n_\epsilon \times 1$ `double`	nonnegative Relaxation weights

Constraint relaxation soft indices		
Variable	Dimension/Class	Associated with...
$a_{Sx,box,idx}$	$n_x \times 1$ `int`	$a_{x,box}$
$b_{Sx,box,idx}$	$n_x \times 1$ `int`	$b_{x,box}$
$a_{Su,box,idx}$	$n_u \times 1$ `int`	$a_{u,box}$
$b_{Su,box,idx}$	$n_u \times 1$ `int`	$b_{u,box}$
$a_{Sx,idx}$	$n_{L_x} \times 1$ `int`	a_x
$b_{Sx,idx}$	$n_{L_x} \times 1$ `int`	b_x
$a_{Su,idx}$	$n_{L_u} \times 1$ `int`	a_u
$b_{Su,idx}$	$n_{L_u} \times 1$ `int`	b_u
$a_{SG_u,idx}$	$n_{L_{G_u}} \times 1$ `int`	a_{G_u}
$b_{SG_u,idx}$	$n_{L_{G_u}} \times 1$ `int`	b_{G_u}
$a_{SG_v,idx}$	$n_{L_{G_v}} \times 1$ `int`	a_{G_v}
$b_{SG_v,idx}$	$n_{L_{G_v}} \times 1$ `int`	b_{G_v}

The function prototype for `Reformulate.c` is

```
 void Reformulate(double *CostLpr, int
numLinCons, int nv,
    double *Wpr, int neps, double *ConsMatpr,
    int nrowConsMat, int H, int nrowLx, int nrowLu, int nBx,
    int nBu, int nrowGxu, int nrowGxv,
    double *lowBoundHardpr, double *upBoundHardpr,
    double *lowBoundSoftpr, double *upBoundSoftpr,
    int *lowLxSoftInd, int *upLxSoftInd, int *lowLuSoftInd,
    int *upLuSoftInd, int *lowBxSoftInd, int *upBxSoftInd,
    int *lowBuSoftInd, int *upBuSoftInd, int *lowGxuSoftInd,
    int *upGxuSoftInd, int *lowGxvSoftInd, int *upGxvSoftInd)
```

A detailed description of each argument follows:

`Reformulate`, Arguments 1-16			
Name	Meaning	variable	Notes
costLpr	linear cost term	$\bar{L}$	from QPformulate, updated to reflect cost of relaxation
numLinCons	# of linear constraints	T_{LC}	
nv	# of orig decision variables	n_v	
Wpr	relaxation weighting	ρ	
neps	# of relaxation slack variables	n_ϵ	
ConsMatpr		$\bar{W}$	same as CCmatpr from QPformulate, updated to reflect relaxed constraints
nrowConsMat			<u>must</u> be $2T_{LC}$*allocV
H	Horizon	H	
nrowLx	# linear constraints on x	n_{L_x}	
nrowLu	# linear constraints on u	n_{L_u}	
nBx			same as rX
nBu			same as ru
nrowGxu	# linear constraints on x, u	n_{LG_u}	
nrowGxv	# linear constraints on x, v	n_{LG_v}	
lowBoundHardpr		$\bar{a}$	avecpr from QPformulate
upBoundHardpr		$\bar{b}$	bvecpr from QPformulate

Discussion: Memory for the following variables must be allocated before calling `Reformulate`. `Reformulate` fills these arrays, which are then ready for an immediate call to LSSOL to solve the quadratic program.

`Reformulate`, Arguments 17-18			
Name	Meaning	variable	Notes
lowBoundSoftpr			updated $\bar{a}$ to reflect relaxed constraints
upBoundSoftpr			updated $\bar{b}$ to reflect relaxed constraints

Discussion: The following variables represent the soft indices. They are nonnegative integers.

`Reformulate`, Arguments 19-30, soft indices		
Name	variable	Notes
lowLxSoftInd	$a_{Sx,idx}$	
upLxSoftInd	$b_{Sx,idx}$	
lowLuSoftInd	$a_{Su,idx}$	
upLuSoftInd	$b_{Su,idx}$	
lowBxSoftInd	$a_{Sx,box,idx}$	can be NULL if rX is 0
upBxSoftInd	$b_{Sx,box,idx}$	can be NULL if rX is 0
lowBuSoftInd	$a_{Su,box,idx}$	can be NULL if ru is 0
upBuSoftInd	$b_{Su,box,idx}$	can be NULL if ru is 0
lowGxuSoftInd	$a_{SG_u,idx}$	
upGxuSoftInd	$b_{SG_u,idx}$	
lowGxvSoftInd	$a_{SG_v,idx}$	
upGxvSoftInd	$b_{SG_v,idx}$	

After `Reformulate` is called, the resulting quadratic program (with $n_v + n_\epsilon$ total decision variables) can be called using variables

- CostQuadpr, CostLpr (updated), CostZpr
- ConsMatpr (updated) (equivalently CCmatpr)
- lowBoundSoftpr, upBoundSoftpr (both updated from) lowBoundHardpr and upBoundHardpr)

Chapter 5

Receding horizon control design

5.1 Control objective

The objective of the RHC control design was to track a time-stamped three-dimensional position reference trajectory in the presence of constraints that limit the maneuverability of the aircraft based on the identified guidance-level vehicle model. The considered set of constraints were characterized in Section 3.4.

In Chapter 3 the nonlinear black-box open vehicle executable model, called DemoSim, was used to identify linear time-invariant models and important constraints of the dynamic behavior between certain commands and output signals. This process provided a high-level, closed-loop model of the T-33 aircraft equipped with an autopilot to represent the most important characteristics of a UCAV-like unmanned vehicle.

5.2 DemoSim modeling

The inputs to the identified LTI DemoSim model are the following command signals: ground speed command V_{cmd} (i.e. total velocity w.r.t. ground), turn rate command $\dot{\chi}_{cmd}$ and altitude rate command $\dot{h}_{cmd}$. The model outputs are ground speed V, heading χ and flight path angle γ. A representation of these variables is depicted in Figure 5.1 using a fixed local coordinate frame. The variables ζ and η denote the local north and east coordinates, respectively.

The linear DemoSim dynamics was identified at the following input-output trim values

$$\begin{aligned} V_{\text{DStrim}} &= 505 \text{ ft/s}, \\ \dot{\chi}_{\text{DStrim}} &= 0 \text{ rad/s}, \\ \dot{h}_{\text{DStrim}} &= 0 \text{ ft/s}, \\ \chi_{\text{DStrim}} &= \frac{\pi}{2} \text{ rad}, \\ \gamma_{\text{DStrim}} &= 0 \text{ rad}. \end{aligned} \tag{5.1}$$

5.3 Prediction model

The prediction model was chosen to accommodate two important requirements. It has to provide a reasonably accurate description of the dynamic relationship between the control inputs of the test platform and the output signals of interest, which include position coordinates for tracking

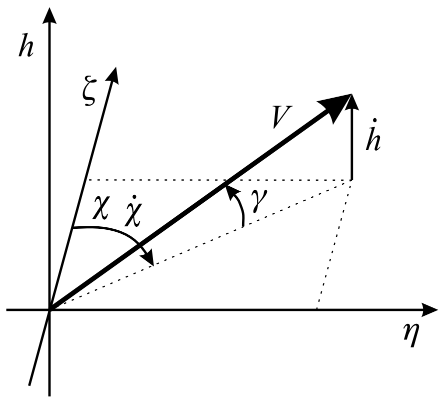


Figure 5.1: Representation of DemoSim model input and output variables in a local coordinate frame.

performance and other variables used for describing maneuvering constraints. At the same time, it has to be simple enough to limit the complexity of the optimization problem that is solved online in a receding horizon fashion.

These objectives were captured by constructing the prediction model from the identified LTI DemoSim dynamics and a flat-earth kinematic model. Using linearized kinematics that was updated every timestep, this approach resulted in a discrete-time linear time-invariant prediction model. Although this model was fixed throughout the prediction horizon of the optimization problem, the updates to the linearized kinematics part rendered the prediction model a linear parameter-varying system, which depended on the current velocity, heading and flight path angle values.

The continuous-time nonlinear prediction model, comprised of the LTI dynamics and the nonlinear kinematics, is depicted in Figure 5.2. The tilded input and output variables of the LTI DemoSim model represent deviations from the trim values:

$$V_{cmd} = V_{\text{DStrim}} + \tilde{V}_{cmd} \tag{5.2a}$$
$$\dot{\chi}_{cmd} = \dot{\chi}_{\text{DStrim}} + \tilde{\dot{\chi}}_{cmd} \tag{5.2b}$$
$$\dot{h}_{cmd} = \dot{h}_{\text{DStrim}} + \tilde{\dot{h}}_{cmd} \tag{5.2c}$$
$$V = V_{\text{DStrim}} + \tilde{V} \tag{5.2d}$$
$$\chi = \chi_{\text{DStrim}} + \tilde{\chi} \tag{5.2e}$$
$$\gamma = \gamma_{\text{DStrim}} + \tilde{\gamma} \tag{5.2f}$$

A schematic diagram of the linearized prediction model is shown in Figure 5.3. Note that the nonlinear kinematics is linearized around fixed V_0, χ_0, γ_0 values that represent current measurements. The inputs fed into the linearized kinematics model represent the differences between the true values predicted by the LTI DemoSim model (after addition of trim values) and the current measurements used for linearization:

$$\tilde{V}_0 = (V_{\text{DStrim}} + \tilde{V}) - V_0 \tag{5.3a}$$
$$\tilde{\chi}_0 = (\chi_{\text{DStrim}} + \tilde{\chi}) - \chi_0 \tag{5.3b}$$
$$\tilde{\gamma}_0 = (\gamma_{\text{DStrim}} + \tilde{\gamma}) - \gamma_0 \tag{5.3c}$$

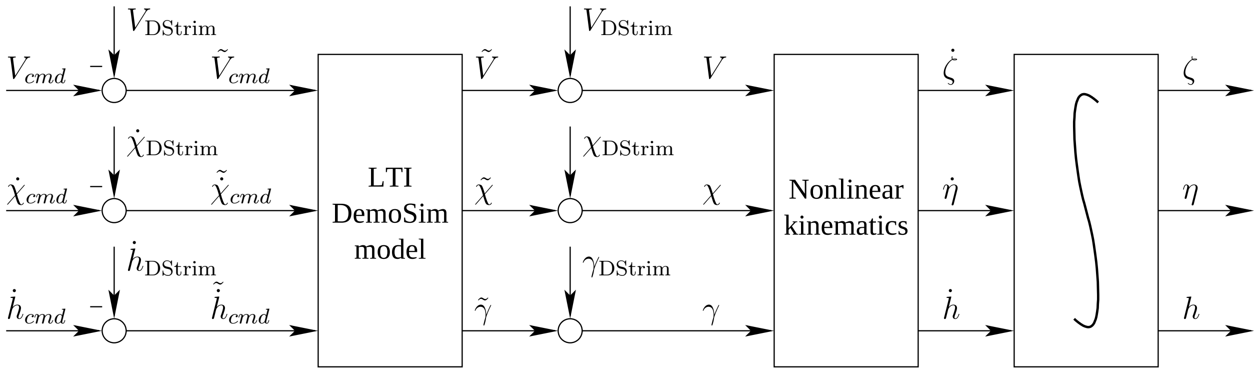


Figure 5.2: Nonlinear prediction model.

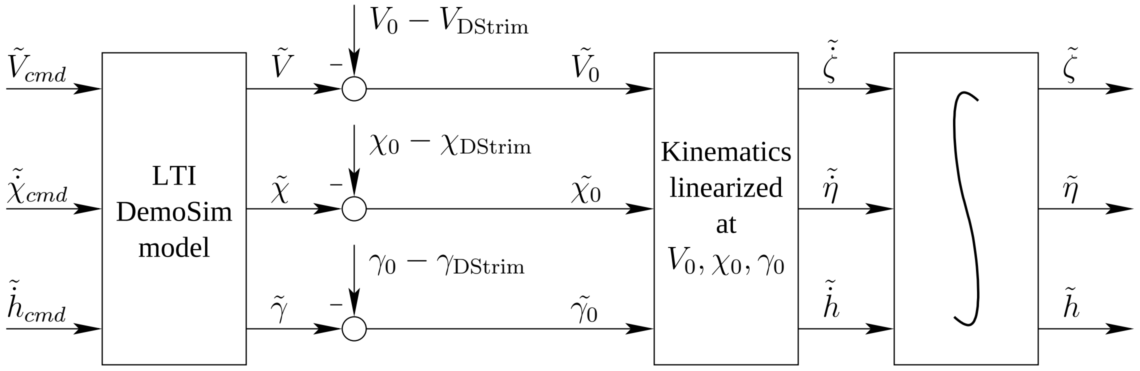


Figure 5.3: Linearized prediction model.

The linearized model was discretized using the discrete-time identified DemoSim dynamics and approximating the continuous-time nonlinear kinematics around fixed V_0, χ_0, γ_0 values with the following forward-Euler discretized linear system

$$\begin{bmatrix} \tilde{\zeta}(k+1) \\ \tilde{\eta}(k+1) \\ \tilde{h}(k+1) \end{bmatrix} = \tilde{A} \begin{bmatrix} \tilde{\zeta}(k) \\ \tilde{\eta}(k) \\ \tilde{h}(k) \end{bmatrix} + \tilde{B} \begin{bmatrix} \tilde{V}_0(k) \\ \tilde{\chi}_0(k) \\ \tilde{\gamma}_0(k) \end{bmatrix} \tag{5.4}$$

where

$$\tilde{A} = I_3, \quad \tilde{B} = T_s \cdot \begin{bmatrix} \cos\chi_0 \cos\gamma_0 & -V_0 \sin\chi_0 \cos\gamma_0 & -V_0 \cos\chi_0 \sin\gamma_0 \\ \sin\chi_0 \cos\gamma_0 & V_0 \cos\chi_0 \cos\gamma_0 & -V_0 \sin\chi_0 \sin\gamma_0 \\ \sin\gamma_0 & 0 & V_0 \cos\gamma_0 \end{bmatrix}. \tag{5.5}$$

The sampling time T_s was 0.5 seconds.

The addition of DemoSim trim values and the subtraction of the current measurement values at the output of the LTI DemoSim model were implemented by augmenting the LTI dynamics with extra states (integrators with zero inputs), which were subtracted from the model outputs. The state values were updated every timestep based on the difference between DemoSim trim values and current measurements. Let us denote the original discrete-time LTI DemoSim matrices with $A_{\text{DS}}, B_{\text{DS}}, C_{\text{DS}}, D_{\text{DS}}$. Using $A_{\text{DS0}}, B_{\text{DS0}}, C_{\text{DS0}}, D_{\text{DS0}}$ to denote the augmented dynamics that adjusts the output values based on current measurements (V_0, χ_0, γ_0) we have

$$\begin{aligned} A_{\text{DS0}} &= \begin{bmatrix} A_{\text{DS}} & 0 \\ 0 & I_3 \end{bmatrix}, \quad B_{\text{DS0}} = \begin{bmatrix} B_{\text{DS}} \\ 0 \end{bmatrix}, \\ C_{\text{DS0}} &= \begin{bmatrix} C_{\text{DS}} & -I_3 \end{bmatrix}, \quad D_{\text{DS0}} = D_{\text{DS}}. \end{aligned} \tag{5.6}$$

The estimated time-delays associated with the pilot model and data processing by the on-board avionics were accounted for by augmenting the discrete-time DemoSim dynamics with extra states. The command input time-delays were characterized as integer multiples of the sampling time T_s. Assuming a time-delay of n_d samples affecting each input channel, the state-space matrices of the DemoSim dynamics were augmented as

$$\begin{aligned} A_d &= \begin{bmatrix} A_{\text{DS0}} & B_{\text{DS0}} & 0 \\ 0 & 0 & I_{3(n_d-1)} \\ 0 & 0 & 0 \end{bmatrix}, \quad B_d = \begin{bmatrix} 0 \\ 0 \\ I_3 \end{bmatrix}, \\ C_d &= \begin{bmatrix} C_{\text{DS0}} & D_{\text{DS0}} & 0 \end{bmatrix}, \qquad D_d = 0. \end{aligned} \tag{5.7}$$

Since the nonlinear kinematics part of the prediction model was always linearized around the current measurements of V_0, χ_0, γ_0, the outputs of this augmented, modified LTI DemoSim model could be fed directly into the linearized kinematics model. In other words, the complete linear prediction model could be obtained by the following augmentation of the state-space matrices

$$\begin{aligned} A &= \begin{bmatrix} A_d & 0 \\ \tilde{B}C_d & \tilde{A} \end{bmatrix}, \quad B = \begin{bmatrix} B_d & 0 \\ \tilde{B}D_d & 0 \end{bmatrix}, \\ C &= \begin{bmatrix} 0 & I_3 \end{bmatrix}, \qquad D = \begin{bmatrix} 0 & I_3 \end{bmatrix}. \end{aligned} \tag{5.8}$$

Additional disturbance inputs were added in the formulas above (5.8), to model additive output disturbance that affects the plant.

The states associated with the DemoSim dynamics "half" of the prediction model were updated using a linear time-invariant observer that relied on the control inputs sent from the RHC controller to the plant $(V_{cmd}, \dot{\chi}_{cmd}, \dot{h}_{cmd})$ and measurements of ground speed V, heading angle χ and flight path angle γ. Note that flight path angle could not be measured directly on the test platform, so a conversion from ground speed and altitude rate measurements was performed to obtain flight path angle according to $\gamma = \arcsin \frac{\dot{h}}{V}$. The input-output scheme of the RHC controller and the observer are shown in Figure 5.4.

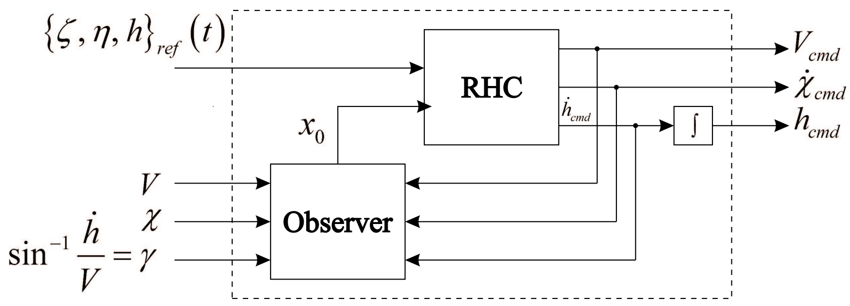


Figure 5.4: RHC controller and observer.

The T-33 test platform avionics provides GPS position measurements in terms of latitude λ, longitude Λ and altitude h using the WGS-84 system. These measurements were converted to north ζ, east η and altitude h values to be compatible with the coordinate frame of the reference trajectory. The north and east coordinates were obtained by the transformation of the geodetic measurements (GPS) into an NEU (north-east-up) local Cartesian coordinate frame that had its origin at the point in space where the RHC controller is engaged $(\lambda_0, \Lambda_0, h_0)$. This flat-earth coordinate frame for north-east navigation is depicted in Figure 5.5 as $K_{\zeta\eta}$. The geodetic GPS altitude measurements however were used directly, without any further conversion. This geodetic coordinate frame for altitude navigation is also illustrated in Figure 5.5 and denoted by K_h.

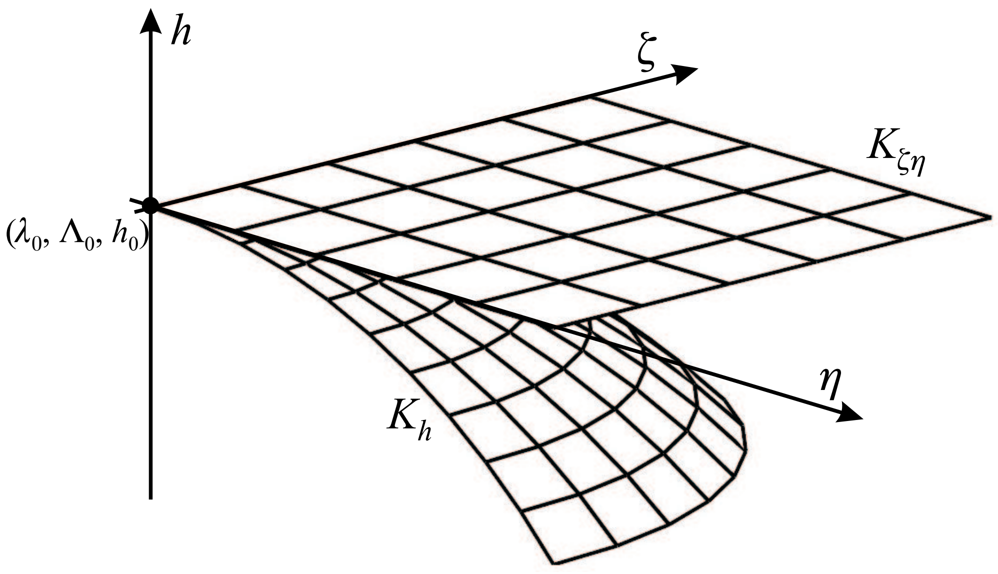


Figure 5.5: Navigation frames for north, east, and altitude coordinates.

This means that the current north-east local coordinates of the nonlinear prediction model were obtained from geodetic GPS measurements using the NEU frame conversion, whereas the current GPS altitude measurement was used directly as the current altitude coordinate. The states associated with the position integrators of the linearized prediction model were always initialized at zero (current position).

The output signals of the entire linear prediction model were assigned to three objective groups denoted by u, z and y. These correspond to hard actuator or other input constraints, maneuvering limits and tracking performance, respectively. The commanded input signals are denoted by r.

The prediction model in (5.8) has only y outputs for tracking performance. The outputs z used to define maneuvering limits were not included in the flight code and will be described in more detail in Section 5.7, including the definition of A_z, b_z linear constraint parameters obtained in Section 3.4.

$$y = \begin{bmatrix} \tilde{\zeta} \\ \tilde{\eta} \\ \tilde{h} \end{bmatrix}, \quad z = A_z \begin{bmatrix} V \\ \dot{\chi} \\ \dot{\gamma} \end{bmatrix}, \quad u = r = \begin{bmatrix} \tilde{V}_{cmd} \\ \tilde{\dot{\chi}}_{cmd} \\ \tilde{\dot{h}}_{cmd} \end{bmatrix}, \quad \Delta r = \begin{bmatrix} \Delta\tilde{V}_{cmd} \\ \Delta\tilde{\dot{\chi}}_{cmd} \\ \Delta\tilde{\dot{h}}_{cmd} \end{bmatrix} \tag{5.9}$$

The parameter-variance of the prediction model present in the $\tilde{B}$ matrix of the linearized kinematics model (5.5), is characterized by the nominal velocity V_0, heading χ_0 and flight path angle

γ_0, around which the kinematic model was linearized. Denoting this vector of parameters with $\varrho(k) = [V_0(k) \;\; \chi_0(k) \;\; \gamma_0(k)]^T$, the linearized discrete-time prediction models have the form

$$\begin{aligned} x(k+1) &= A_k x(k) + B_k r(k) \\ \begin{bmatrix} y(k) \\ z(k) \\ u(k) \end{bmatrix} &= C_k x(k) + D_k r(k) \end{aligned} \tag{5.10}$$

where the parameter dependency of the prediction model is indicated by the subscript k, meaning

$$\begin{aligned} A_k &= A(\varrho(k)), \quad B_k = B(\varrho(k)), \\ C_k &= C(\varrho(k)), \quad D_k = D(\varrho(k)). \end{aligned}$$

5.4 RHC problem formulation

The time-stamped three-dimensional position reference trajectory was specified in terms of north, east, and altitude coordinates in a local NEU coordinate frame relative to the point in space where the controller is engaged. The reference position vector elements were placed 0.5 seconds apart from each other in time. Denote the reference position trajectory values by $\zeta_{ref}(k), \eta_{ref}(k), h_{ref}(k)$ at time step k. The LTI prediction model-based RHC controller was required to track a "linearized" position reference trajectory $\tilde{\zeta}_{ref}(k), \tilde{\eta}_{ref}(k), \tilde{h}_{ref}(k)$. This was generated by subtracting the simulated output of the nonlinear kinematics from the original reference trajectory. The "nominal" simulated trajectory was calculated using fixed V_0, χ_0, γ_0 values based on the current measurements used for linearization. The linear reference trajectory was therefore obtained by

$$\tilde{\zeta}_{ref}(k) = \zeta_{ref}(k) - \zeta_0(k), \tag{5.11a}$$

$$\tilde{\eta}_{ref}(k) = \eta_{ref}(k) - \eta_0(k), \tag{5.11b}$$

$$\tilde{h}_{ref}(k) = h_{ref}(k) - h_0(k), \tag{5.11c}$$

where the "nominal" simulated trajectories were calculated by

$$\zeta_0(k) = \sum_{i=1}^{k} T_s \cdot \dot{\zeta}_0(V_0, \chi_0, \gamma_0), \tag{5.12a}$$

$$\eta_0(k) = \sum_{i=1}^{k} T_s \cdot \dot{\eta}_0(V_0, \chi_0, \gamma_0), \tag{5.12b}$$

$$h_0(k) = \sum_{i=1}^{k} T_s \cdot \dot{h}_0(V_0, \chi_0, \gamma_0). \tag{5.12c}$$

The state values that represent the constant additive output disturbance in the prediction model were updated every timestep based on the following disturbance filter

$$d(k+1) = 0.99 d(k) + 0.01 d_{in}(k) \tag{5.13}$$

where the input $d_{in}(k)$ was determined from the following error equation

$$d_{in}(k) = \underbrace{\begin{bmatrix} \zeta(k) \\ \eta(k) \\ h(k) \end{bmatrix}}_{\text{pos. meas.}} - \underbrace{\begin{bmatrix} \zeta_{ref}(k) \\ \eta_{ref}(k) \\ h_{ref}(k) \end{bmatrix}}_{\text{pos. ref.}} - \underbrace{\begin{bmatrix} \tilde{\zeta}(k|k-1) \\ \tilde{\eta}(k|k-1) \\ \tilde{h}(k|k-1) \end{bmatrix}}_{\text{pred. lin. output}} \tag{5.14}$$

(Note: Although the error equation in (5.14) was implemented in the flight code, it is incorrect. The correct error equation would be

$$d_{in}(k) = \underbrace{\begin{bmatrix}\zeta(k)\\ \eta(k)\\ h(k)\end{bmatrix}}_{\text{pos. meas.}} - \underbrace{\begin{bmatrix}\zeta_0(k|k-1)\\ \eta_0(k|k-1)\\ h_0(k|k-1)\end{bmatrix}}_{\text{nom. pos. pred.}} - \underbrace{\begin{bmatrix}\tilde{\zeta}(k|k-1)\\ \tilde{\eta}(k|k-1)\\ \tilde{h}(k|k-1)\end{bmatrix}}_{\text{pred. lin. output}} \tag{5.15}$$

where $\zeta_0(k|k-1), \eta_0(k|k-1), h_0(k|k-1)$ are position predictions based on the nonlinear kinematics model and $V_0(k-1), \chi_0(k-1), \gamma_0(k-1)$ measurements at time $k-1$.

Nominal simulation results show a marginal improvement in performance using the correct error equation. When trying to simulate the flight test environment by assuming orders of magnitude greater time-delays in the actual plant than what was assumed in the prediction model, the difference introduced by the corrected error equation is more eloquent, however still doesn't change the qualitative nature of the results.)

The optimization problem setup is based on the linear MPC formulation of [14] with some modifications. The studies in [15, 16] formed the basis of the RHC design for the final flight test. In most linear predictive controllers, the performance is specified by the following quadratic cost function to be minimized, which will also be adopted here:

$$\begin{aligned} J(k) = &\sum_{i=1}^{H_p} \|\hat{y}(k+i \mid k) - y_{ref}(k+i \mid k)\|_Q^2 + \\ &+ \sum_{i=0(\delta H_c)}^{H_c-1} \|\Delta r(k+i \mid k)\|_R^2 + \rho\varepsilon \end{aligned} \tag{5.16}$$

where $\hat{y}(k+i \mid k)$ is the i-step ahead prediction of the outputs based on data up to time k. H_p denotes the number of steps in the output prediction horizon. These predictions of the outputs are functions of future control increments $\Delta r(k+i \mid k)$ for $i = 0, \delta H_c, 2\delta H_c, \ldots, H_c - 1$. The integer number of samples H_c is called the control horizon, the control signal is allowed to change only at integer multiples of δH_c samples and is set to be constant for all $i \geq H_c$. This means that the future control signal has the form of a stairstep function with steps occuring at δH_c intervals. The reference signal y_{ref} represents the desired outputs, Q and R are suitably chosen weighting matrices. For our specific application, the optimization problem was specified using the following parameters

$$H_c = 1, \quad \delta H_c = 1, \quad H_p = 40,$$
$$Q = \begin{bmatrix}0.01 & 0 & 0\\ 0 & 0.01 & 0\\ 0 & 0 & 1\end{bmatrix}, \quad R = \begin{bmatrix}10 & 0 & 0\\ 0 & 5\cdot 10^6 & 0\\ 0 & 0 & 1\end{bmatrix}.$$

The slack variable ε and its weight ρ are used for softening constraints. The exact purpose of the slack variable and weight in the problem formulation will be clarified shortly.

In order to obtain the predictions for the signals of interest, a model of the process is needed. By using a linear model, the resulting optimization problem of minimizing $J(k)$ will be a quadratic programming (QP) problem, for which fast and numerically reliable algorithms are available. The linearized prediction model, developed in Section 5.3, is augmented with extra states to fit the formulation in this RHC scheme. Three integrators are added to convert the control changes Δr

into actual controls r, each one associated with the command inputs of velocity, turn rate and altitude rate. A simple disturbance model is incorporated to the state space description of the prediction model in equation (5.17), which assumes constant disturbances are acting on outputs. The constant disturbance estimates are obtained by filtering the difference between measured and predicted outputs, as described by equation (5.14).

The disturbance model also serves to mitigate the effect of model mismatch. The augmented linear prediction model has the following form

$$\begin{aligned}
\overbrace{\begin{bmatrix}\hat{x}(k+1)\\ \hat{d}(k+1)\\ r(k)\end{bmatrix}}^{\hat{\xi}(k+1)} &= \overbrace{\begin{bmatrix}A_k & 0 & B_k\\ 0 & I & 0\\ 0 & 0 & I\end{bmatrix}}^{\mathcal{A}_k}\overbrace{\begin{bmatrix}\hat{x}(k)\\ \hat{d}(k)\\ r(k-1)\end{bmatrix}}^{\hat{\xi}(k)} + \overbrace{\begin{bmatrix}B_k\\ 0\\ I\end{bmatrix}}^{\mathcal{B}_k}\Delta r(k) \\
\underbrace{\begin{bmatrix}\hat{y}(k)\\ \hat{z}(k)\\ \hat{u}(k)\end{bmatrix}}_{\hat{w}(k)} &= \underbrace{\left[\begin{array}{c|c|c} & I & \\ C_k & 0 & D_k\\ & 0 & \end{array}\right]}_{\mathcal{C}_k}\underbrace{\begin{bmatrix}\hat{x}(k)\\ \hat{d}(k)\\ r(k-1)\end{bmatrix}}_{\hat{\xi}(k)} + \underbrace{D_k}_{\mathcal{D}_k}\Delta r(k)
\end{aligned} \tag{5.17}$$

After creating the prediction model and formulating the RHC problem, the optimization problem was translated to the formulation used by the RHC API described in Section 4.3 of Chapter 4. A table showing the corresponding parameters of the two formulations and how the conversion was done will be described in Section 5.5.

In order to gain better insight to the RHC problem, a standard quadratic programming solution based on substitution is shown. Although this derivation does not represent the actual process that was used within the RHC API to solve the problem, some parts of this development were used in the flight code to obtain predictions of the system for the disturbance estimator.

By using successive substitution, it is straightforward to derive that the prediction model of inner-loop outputs (signals of interest) over the prediction horizon is given by equation (5.18).

Denote parts of the state matrices $\mathcal{C}_k$ and $\mathcal{D}_k$ in equation (5.18) that correspond to the predicted

---

$$\underbrace{\begin{bmatrix}\hat{w}(k+1\mid k)\\ \hat{w}(k+2\mid k)\\ \vdots\\ \hat{w}(k+H_c\mid k)\\ \hat{w}(k+H_c+1\mid k)\\ \vdots\\ \hat{w}(k+H_p\mid k)\end{bmatrix}}_{\mathcal{W}(k)} = \underbrace{\begin{bmatrix}\mathcal{C}_k\mathcal{A}_k\\ \mathcal{C}_k\mathcal{A}_k^2\\ \vdots\\ \mathcal{C}_k\mathcal{A}_k^{H_c}\\ \mathcal{C}_k\mathcal{A}_k^{H_c+1}\\ \vdots\\ \mathcal{C}_k\mathcal{A}_k^{H_p}\end{bmatrix}}_{\Psi_k}\hat{\xi}(k) + \underbrace{\begin{bmatrix}\mathcal{C}_k\mathcal{B}_k & \mathcal{D}_k & \cdots & 0\\ \mathcal{C}_k\mathcal{A}_k\mathcal{B}_k & \mathcal{C}_k\mathcal{B}_k & \mathcal{D}_k & \vdots\\ \vdots & \vdots & \ddots & \ddots\\ \mathcal{C}_k\mathcal{A}_k^{H_c-1}\mathcal{B}_k & \mathcal{C}_k\mathcal{A}_k^{H_c-2}\mathcal{B}_k & \cdots & \mathcal{C}_k\mathcal{B}_k\\ \mathcal{C}_k\mathcal{A}_k^{H_c}\mathcal{B}_k & \mathcal{C}_k\mathcal{A}_k^{H_c-1}\mathcal{B}_k & \cdots & \mathcal{C}_k\mathcal{A}_k\mathcal{B}_k\\ \vdots & \vdots & \ddots & \vdots\\ \mathcal{C}_k\mathcal{A}_k^{H_p-1}\mathcal{B}_k & \mathcal{C}_k\mathcal{A}_k^{H_p-2}\mathcal{B}_k & \cdots & \mathcal{C}_k\mathcal{A}_k^{H_p-H_c}\mathcal{B}_k\end{bmatrix}}_{\Theta_k}\underbrace{\begin{bmatrix}\Delta r(k\mid k)\\ \vdots\\ \Delta r(k+H_c-1\mid k)\end{bmatrix}}_{\Delta\mathcal{R}(k)} \tag{5.18}$$

$\hat{y}(k)$ outputs in $\hat{w}(k)$, with an additional y subscript

$$\mathcal{C}_k = \begin{bmatrix} \mathcal{C}_{ky} \\ \mathcal{C}_{kz} \\ \mathcal{C}_{ku} \end{bmatrix}, \quad \mathcal{D}_k = \begin{bmatrix} \mathcal{D}_{ky} \\ \mathcal{D}_{kz} \\ \mathcal{D}_{ku} \end{bmatrix}.$$

Consider only those predicted outputs that appear in the performance index

$$\begin{aligned} \hat{y}(k) &= \mathcal{C}_{ky}\hat{\xi}(k) + \mathcal{D}_{ky}\Delta r(k), \\ \mathcal{Y}(k) &= \left[\hat{y}\left(k+1 \mid k\right), \ldots, \hat{y}\left(k+H_p \mid k\right)\right]^T \end{aligned}$$

using only the corresponding $\mathcal{C}_{ky}$ and $\mathcal{D}_{ky}$ matrices in expression (5.18). The prediction for these outputs has the form

$$\mathcal{Y}(k) = \Psi_{ky}\hat{\xi}(k) + \Theta_{ky}\Delta\mathcal{R}(k) \tag{5.19}$$

Substituting the predicted output in (5.19) into the cost function of (5.16), we get a quadratic expression in terms of the control changes $\Delta\mathcal{R}(k)$:

$$J(k) = \Delta\mathcal{R}(k)^T\mathcal{H}_k\Delta\mathcal{R}(k) - \Delta\mathcal{R}(k)^T\mathcal{G}_k + const + \rho\varepsilon \tag{5.20}$$

where

$$\begin{gathered} \mathcal{H}_k = \Theta_{ky}^T Q_e \Theta_{ky} + R_e, \quad \mathcal{G}_k = 2\Theta_{ky}^T Q_e \mathcal{E}(k), \\ const = \mathcal{E}^T(k) Q_e \mathcal{E}(k) \end{gathered}$$

and $\mathcal{E}(k)$ is defined as a tracking error between the future target trajectory and the free response of the system, i.e. $\mathcal{E}(k) = \mathcal{Y}_{ref}(k) - \Psi_{ky}\hat{\xi}(k)$. Q_e and R_e are block diagonal matrices of appropriate dimensions with Q and R on the main diagonal, respectively. (These could be chosen parameter-dependent also.)

As in most applications, there are level and rate limits on control inputs. These are enforced as hard constraints

$$\underline{u} \le \hat{u}\left(k+1 \mid k\right), \ldots, \hat{u}\left(k+H_p \mid k\right) \le \overline{u} \tag{5.21}$$

$$\underline{\Delta r} \le \Delta r\left(k\right), \Delta r\left(k+\delta H_c\right), \ldots, \Delta r\left(k+H_c\right) \le \overline{\Delta r} \tag{5.22}$$

since the RHC algorithm has almost direct control over them (the optimization variables are the changes in control inputs), so there is no modeling uncertainty associated with this aspect of the prediction model. However, another type of constraint is also considered in this specific application example represented by certain maneuvering limits on the aircraft. The controller has to be versatile enough to handle these limits that might be system-state dependent or change according to different stages of a mission. The most important maneuvering constraints of the T-33 testbed were characterized in Section 3.4 based on the DemoSim open vehicle executable model. It is vital that these limits are treated as soft constraints, since disturbances and model mismatch can easily lead to infeasibility problems if hard constraints are put on these type of output signals.

The numerical values of the u output signal limits (representing hard constraints) were specified as

$$\underline{u} = \begin{bmatrix} -100 \text{ ft/s} \\ -0.035 \text{ rad/s} \\ -1000 \text{ ft/s} \end{bmatrix}, \quad \overline{u} = \begin{bmatrix} 100 \text{ ft/s} \\ 0.035 \text{ rad/s} \\ 1000 \text{ ft/s} \end{bmatrix}. \tag{5.23}$$

Hard constraints were put on the optimization variables as well, specifically

$$\underline{\Delta r} = \begin{bmatrix} -4 \text{ ft/s}^2 \\ -0.01 \text{ rad/s}^2 \\ -1000 \text{ ft/s}^2 \end{bmatrix}, \quad \overline{\Delta r} = \begin{bmatrix} 4 \text{ ft/s}^2 \\ 0.01 \text{ rad/s}^2 \\ 1000 \text{ ft/s}^2 \end{bmatrix}. \tag{5.24}$$

Constraint softening for the outputs z is accomplished by introducing an additional slack variable that allows some level of constraint violation if no feasible solution exists

$$\begin{aligned} &\underline{z} - \varepsilon \leq \hat{z}\,(k+1 \mid k)\,, \ldots, \hat{z}\,(k+H_p \mid k) \leq \overline{z} + \varepsilon \\ &0 \leq \varepsilon \end{aligned} \tag{5.25}$$

It is beneficial to use an ∞-norm (maximum violation) penalty on constraint violations (as shown in (5.16) and (5.25)), because it gives an "exact penalty" method if the weight ρ is large enough. This means that constraint violations will not occur unless no feasible solution exists to the original "hard" problem. If a feasible solution exists, the same solution will be obtained as with the "hard" formulation. Using the linear prediction model in (5.18), all of the constraints in (5.21) and (5.25) can be posed as linear constraints on the optimization variables $\Delta\mathcal{R}$ and ε. Finally, the QP to be solved at each time step has the following form

$$\begin{aligned} \min_{\Delta\mathcal{R},\, \varepsilon} \quad & \Delta\mathcal{R}^T \mathcal{H}_k \Delta\mathcal{R} + \Delta\mathcal{R}^T \mathcal{G}_k + const + \rho\varepsilon \\ \text{s. t.} \quad & \begin{bmatrix} \Omega_{k,hard} \\ \Omega_{k,soft} \end{bmatrix} \Delta\mathcal{R} \leq \begin{bmatrix} \omega_{k,hard} \\ \omega_{k,soft} \end{bmatrix} + \begin{bmatrix} 0 \\ \varepsilon \end{bmatrix} \\ & 0 \leq \varepsilon \end{aligned} \tag{5.26}$$

Note: The optimization yields $\Delta\tilde{V}_{cmd}, \Delta\tilde{\dot{\chi}}_{cmd}, \Delta\tilde{\ddot{h}}_{cmd}$ values, which were integrated to get $\tilde{V}_{cmd}, \tilde{\dot{\chi}}_{cmd}, \tilde{\dot{h}}_{cmd}$. The obtained $\tilde{\dot{h}}_{cmd}$ value was further integrated to get $\tilde{h}_{cmd}$. Trim values were added to arrive at $V_{cmd}, \dot{\chi}_{cmd}, h_{cmd}$ values, which could be directly implemented on the T-33 autopilot.

5.4.1 Remarks

The problem formulation in the preceding section is a natural extension of a fixed LTI model based RHC. The prediction at a certain time step is based on a linear model that best describes the plant at the actual flight condition, assuming that flight condition dependent linear models are available for prediction. A fixed LTI model is used over the entire prediction horizon but it is updated according to the values of the scheduling parameters $\varrho\,(k)$ every time the horizon is propagated and the optimization is resolved based on new measurement data. This approach leads to the QP problem in (5.26), and the state matrices describing the internal model change in each implementation cycle according to their current values: $\mathcal{A}_k$, $\mathcal{B}_k$, $\mathcal{C}_k$, $\mathcal{D}_k$. Usually a flight condition dependent description of the plant dynamics can be obtained either by freezing the scheduling parameters of a quasi-LPV model [17], or interpolating over a database of linearized models. In other cases the nonlinear prediction model is simple enough to lend itself to "online" linearization, while still retaining a reasonable prediction accuracy. This latter approach was followed in our control solution.

We note if an accurate prediction of the parameters that the linear models depend on is available, this would allow for the prediction model to vary over the prediction horizon. The optimization problem could still be formulated as a quadratic program using different state matrices of the internal model at each time step. Obtaining a reasonable prediction of the scheduling parameters is not always easy, one could experiment with solving the problem first with the fixed LTI model based RHC method and use the solution as the prediction for the scheduling parameters. Our investigations indicate, that this extra effort doesn't lead to significant improvement for the specific application example and horizon lengths considered. Moreover, even though the optimization problem complexity is retained, the additional computational overhead could undermine real-time implementation of these ideas if the parameter-dependent models are calculated using interpolation over a collection of linear systems.

Acknowledgement

The authors gratefully acknowledge discussions with George Papageorgiou at Honeywell Labs on the particular RHC formulation to be used in the final flight test.

5.5 Translation to the RHC API problem formulation

Denote the dimensions of input-output signals and states introduced in the previous section with n_r, n_y, n_z, n_u and n_ξ, respectively. The number of control changes to optimize over is denoted by $n_{\Delta\mathcal{R}} = n_r\lceil H_c/\delta H_c\rceil$. Let us denote the selection matrix that maps the control change optimization variables $\Delta\mathcal{R}$ into changes in control at each time step of the output horizon with $\mathcal{S}$

$$\begin{bmatrix} \Delta r_0 \\ \vdots \\ \Delta r_{H_p-1} \end{bmatrix} = \mathcal{S}\Delta\mathcal{R} \tag{5.27}$$

where $\mathcal{S} \in \mathbb{R}^{H_p \times n_r\lceil H_c/\delta H_c\rceil}$ represents the difference between the output horizon H_p and control horizon H_c, as well as any blocking technique implemented by choosing $\delta H_c > 1$. This means that in general the matrix $\mathcal{S}$ is structured as follows

$$\mathcal{S} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix} \tag{5.28}$$

where the '1'-s appear in row numbers that are integer multiples of δH_c, until the column space is spanned. The remaining rows are all zeros.

Tables 5.1 and 5.2 contain a summary of the conversion between the RHC API problem formulation and the corresponding parameters described in Section 5.4. The general linear constraint matrix L_{G_v} referred to in Table 5.2 is given as

$$L_{G_v} = \begin{bmatrix} I_{H_p} \otimes \mathcal{C}_{kz} & 0_{n_z H_p \times n_{\Delta\mathcal{R}}} & -1_{H_p\times 1} \\ -I_{H_p} \otimes \mathcal{C}_{kz} & 0_{n_z H_p \times n_{\Delta\mathcal{R}}} & 1_{H_p\times 1} \\ 0_{1\times n_\xi H_p} & 0_{1\times n_{\Delta\mathcal{R}}} & 1 \end{bmatrix} \tag{5.29}$$

RHC API notation	RHC parameters in Section 5.4
Q	$0_{n_\xi \times n_\xi}$
R	R
C	$\mathcal{C}_{ky}$
G	I_{n_y}
M	Q
F	$0_{(n_{\Delta\mathcal{R}}+1)\times(n_{\Delta\mathcal{R}}+1)}$
K	$[0_{1\times n_{\Delta\mathcal{R}}} \quad \rho]^T$
Φ	$0_{n_\xi \times n_\xi}$
U	$\left[\mathcal{S} \otimes I_{n_r} \quad 0_{n_r H_p \times 1}\right]$ **
A	$\mathcal{A}_k$
B	$\mathcal{B}_k$
E	$0_{n_\xi \times n_y}$
H	H_p
u_k	Δr_k
d_k	$y_{ref}(k)$

Table 5.1: Corresponding parameters in the RHC API problem formulation (*to be continued*).

5.6 RHC reconfiguration based on FDI output

The Fault Detection (FD) filter design is described in Chapter 6 along with a description of the test environment in the final flight demonstration. For detailed information about the experiment timetable and the activation time of the fault detection filter, refer to Appendices A-B.

For the purpose of describing the RHC reconfiguration process based on FDI output, it suffices to say that at some point in the flight experiment a simulated aileron actuator fault is inserted. This is implemented in the flight control software by corrupting the turn rate command output

** In the specific example at hand with the chosen parameters described in the previous section, we have

$$U = \begin{bmatrix} I_3 & 0_{3\times 1} \\ 0_{(n_r H_p - 3)\times 3} & 0_{(n_r H_p - 3)\times 1} \end{bmatrix}$$

RHC API notation	**RHC parameters in Section 5.4**
$a_{x,box}$	$-\infty$
$b_{x,box}$	$+\infty$
L_x	$\mathcal{C}_{ku}$
a_x	$\underline{u}$
b_x	$\overline{u}$
$a_{u,box}$	$\underline{\Delta r}$
$b_{u,box}$	$\overline{\Delta r}$
L_u	0
a_u	$-\infty$
b_u	$+\infty$
L_{G_u}	0
a_{G_u}	$-\infty$
b_{G_u}	$+\infty$
L_{G_v}	see in equation (5.29)
a_{G_v}	$\begin{bmatrix} 1_{1\times H_p} \otimes \underline{z}^T & 0 \end{bmatrix}^T$
b_{G_v}	$\begin{bmatrix} 1_{1\times H_p} \otimes \overline{z}^T & +\infty \end{bmatrix}^T$

Table 5.2: Corresponding parameters in the RHC API problem formulation (*continued*).

of the controller with the output of a fault model. The corrupted control signal is then sent to the healthy aircraft, however for the controller it appears as if a fault has occurred in the aircraft. The fault detection filter was designed to detect this artificially inserted fault when the aircraft reaches a certain segment of the reference trajectory designed to excite the lateral fault dynamics and facilitate the detection process.

The fault model is depicted in Figure 5.6, which illustrates how the turn rate control command was corrupted in the fault scenario.

The receding horizon controller was reconfigured when the threshold residual of the detection filter indicated a fault after the insertion of the simulated fault model at the appropriate segment of the flight experiment. The reconfiguration process involved two major changes to the RHC

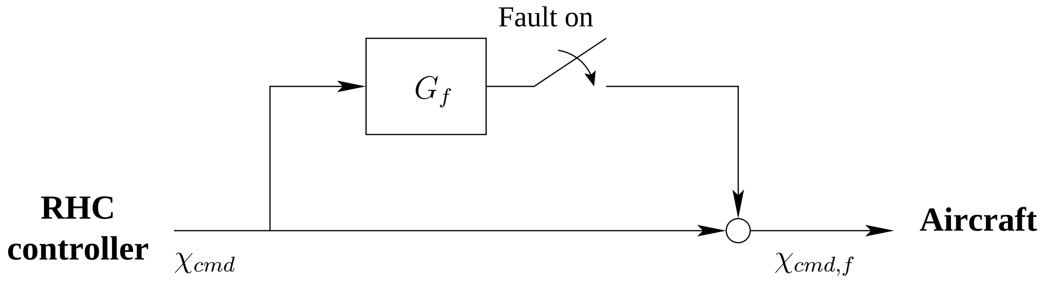


Figure 5.6: Fault injection diagram.

controller. The prediction model was updated and augmented with additional dynamics to reflect the faulty behavior of the aircraft. At the same time, more restrictive constraints were put on the turn rate control command, to counteract the corruption caused by the fault model and represent a less aggressive control strategy.

The numerical values of the modified u control signal limits (representing hard constraints) associated with the faulty model were specified as

$$\underline{u} = \begin{bmatrix} -100 \text{ ft/s} \\ -0.027 \text{ rad/s} \\ -1000 \text{ ft/s} \end{bmatrix}, \quad \overline{u} = \begin{bmatrix} 100 \text{ ft/s} \\ 0.027 \text{ rad/s} \\ 1000 \text{ ft/s} \end{bmatrix}. \tag{5.30}$$

A description of modifications to the prediction model based on incorporation of the fault dynamics follows. Based on modeling considerations described in Chapter 3, the turn rate ($\dot{\chi}$) input to heading (χ) output channel of DemoSim was identified as a single-input single-output (SISO) system, decoupled from other inputs and outputs. This means that the state matrices describing the identified DemoSim dynamics have the following structure:

$$A_{\text{DS}} = \begin{bmatrix} A_{VV} & 0 & A_{Vh} \\ 0 & A_{\chi\dot{\chi}} & 0 \\ A_{\gamma V} & 0 & A_{\gamma h} \end{bmatrix}, \quad B_{\text{DS}} = \begin{bmatrix} B_{VV} & 0 & B_{Vh} \\ 0 & B_{\chi\dot{\chi}} & 0 \\ B_{\gamma V} & 0 & B_{\gamma h} \end{bmatrix}, \tag{5.31}$$
$$C_{\text{DS}} = \begin{bmatrix} C_{VV} & 0 & C_{Vh} \\ 0 & C_{\chi\dot{\chi}} & 0 \\ C_{\gamma V} & 0 & C_{\gamma h} \end{bmatrix}, \quad D_{\text{DS}} = \begin{bmatrix} D_{VV} & 0 & D_{Vh} \\ 0 & D_{\chi\dot{\chi}} & 0 \\ D_{\gamma V} & 0 & D_{\gamma h} \end{bmatrix}.$$

Due to this decoupled structure, the faulty lateral DemoSim dynamics can be represented by the interconnection shown in Figure 5.7, where A_f, B_f, C_f, D_f stand for the state matrices of the SISO fault model. The faulty lateral dynamics can be described by the following state matrices based on the interconnection in Figure 5.7.

$$A_{\chi\dot{\chi},f} = \begin{bmatrix} A_{\chi\dot{\chi}} & B_{\chi\dot{\chi}}C_f \\ 0 & A_f \end{bmatrix}, \quad B_{\chi\dot{\chi},f} = \begin{bmatrix} B_{\chi\dot{\chi}}(1+D_f) \\ B_f \end{bmatrix}, \tag{5.32}$$
$$C_{\chi\dot{\chi},f} = \begin{bmatrix} C_{\chi\dot{\chi}} & D_{\chi\dot{\chi}}C_f \end{bmatrix}, \quad D_{\chi\dot{\chi},f} = \begin{bmatrix} D_{\chi\dot{\chi}}(1+D_f) \end{bmatrix}.$$

Using this representation, we can construct the modified prediction model incorporating the faulty lateral dynamics by replacing $A_{\chi\dot{\chi}}, B_{\chi\dot{\chi}}, C_{\chi\dot{\chi}}, D_{\chi\dot{\chi}}$ in the DemoSim state matrices with $A_{\chi\dot{\chi},f}, B_{\chi\dot{\chi},f}, C_{\chi\dot{\chi},f}, D_{\chi\dot{\chi},f}$. The obtained faulty state matrices of $A_{\text{DS},f}, B_{\text{DS},f}, C_{\text{DS},f}, D_{\text{DS},f}$ are then used to build the complete prediction model as described in (5.6)–(5.8).

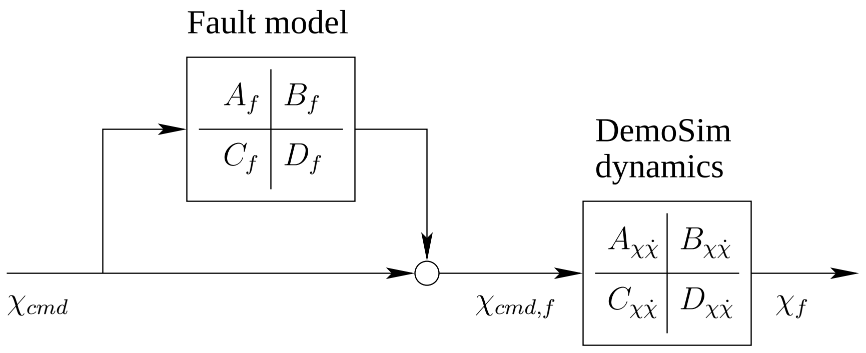


Figure 5.7: Lateral DemoSim dynamics after fault activation.

5.7 Flight envelope limits as output constraints

Flight envelope limits were characterized in Section 3.4 as a collection of linear inequalities in the following form

$$A_z \begin{bmatrix} V \\ \dot{\chi} \\ \dot{\gamma} \end{bmatrix} \leq b_z \tag{5.33}$$

This means that predictions for $V, \dot{\chi}, \dot{\gamma}$ variables are needed in order to implement the constraints of the form (5.33).

State matrices of the original linear discrete-time identified DemoSim model can be used to define the additional prediction model outputs that are needed for constraint specifications. The identified DemoSim model is described with state matrices A_{DS}, B_{DS}, C_{DS}, D_{DS} corresponding to $\tilde{V}, \tilde{\chi}, \tilde{\gamma}$ outputs that represent deviations from trim values. Specifying a true ground speed V output of the complete prediction model requires the addition of the trim value to the identified LTI DemoSim model output variable: $V = V_{\text{DStrim}} + \tilde{V}$. Since the identified model has χ and γ outputs, their derivatives could be approximated using the discrete-time state matrices in the following way. Assume that a discrete-time LTI system is described with state-matrices A, B, C, D. If the D matrix is all zero, then the derivative of the output signal y can be approximated by

$$\dot{y} \sim T_s^{-1} \Delta y = T_s^{-1} (y_{k+1} - y_k) = T_s^{-1} C(A - I)x_k + T_s^{-1} C B u_k \tag{5.34}$$

where T_s is the sampling time. Since the D_{DS} matrix of the identified DemoSim model is all zero, we can apply this method to formulate new $\dot{\chi}, \dot{\gamma}$ outputs using the original identified state matrices.

Partition the original $C_{\text{DS}}, D_{\text{DS}}$ identified DemoSim matrices with respect to the individual outputs in the following way

$$C_{\text{DS}} = \begin{bmatrix} C_{\text{DS},V} \\ C_{\text{DS},\chi} \\ C_{\text{DS},\gamma} \end{bmatrix}, \quad D_{\text{DS}} = \begin{bmatrix} D_{\text{DS},V} \\ D_{\text{DS},\chi} \\ D_{\text{DS},\gamma} \end{bmatrix}. \tag{5.35}$$

Then form the matrices

$$C_{\text{DS}}^o = \begin{bmatrix} C_{\text{DS},V} \\ T_s^{-1} C_{\text{DS},\chi}(A_{\text{DS}} - I) \\ T_s^{-1} C_{\text{DS},\gamma}(A_{\text{DS}} - I) \end{bmatrix} \quad D_{\text{DS}}^o = \begin{bmatrix} D_{\text{DS},V} \\ T_s^{-1} C_{\text{DS},\chi} B_{\text{DS}} \\ T_s^{-1} C_{\text{DS},\gamma} B_{\text{DS}} \end{bmatrix} \tag{5.36}$$

in order to create new augmented DemoSim matrices of

$$\begin{aligned} A^o_{\text{DS0}} &= A_{\text{DS0}}, & B^o_{\text{DS0}} &= B_{\text{DS0}}, \\ C^o_{\text{DS0}} &= \begin{bmatrix} C_{\text{DS0}} \\ C^o_{\text{DS}} \quad 0 \end{bmatrix}, & D^o_{\text{DS0}} &= \begin{bmatrix} D_{\text{DS0}} \\ D^o_{\text{DS}} \end{bmatrix}. \end{aligned} \tag{5.37}$$

which now have six outputs of the variables

$$\begin{bmatrix} \tilde{V}_0 & \tilde{\chi}_0 & \tilde{\gamma}_0 & \tilde{V} & \dot{\tilde{\chi}} & \dot{\tilde{\gamma}} \end{bmatrix}^T$$

including the newly defined last three. The matrices in (5.37) will replace the original A_{DS0}, B_{DS0}, C_{DS0}, D_{DS0} matrices of (5.6) in constructing the complete prediction model. This means that the matrices A_d, B_d, C_d, D_d are constructed exactly as in (5.7), but using the newly defined A^o_{DS0}, B^o_{DS0}, C^o_{DS0}, D^o_{DS0} from (5.37). The complete prediction model state matrices are then formed by

$$\begin{aligned} A^o &= \begin{bmatrix} A_d & 0 \\ \tilde{B}C_{d1} & \tilde{A} \end{bmatrix}, & B^o &= \begin{bmatrix} B_d & 0 \\ \tilde{B}D_{d1} & 0 \end{bmatrix}, \\ C^o &= \begin{bmatrix} 0 & I_3 \\ A_z C_{d2} & 0 \end{bmatrix}, & D^o &= \begin{bmatrix} 0 & I_3 \\ A_z D_{d2} & 0 \end{bmatrix}. \end{aligned} \tag{5.38}$$

where C_{d1}, D_{d1} correspond to the original three outputs and C_{d2}, D_{d2} to the newly created second three outputs of the matrices C_d, D_d. The complete prediction model of (5.38) describes the dynamic relationship between the control r and disturbance d inputs, and the tracking y and soft constraint z outputs as defined in (5.9)

$$\begin{aligned} x_{k+1} &= A^o x_k + B^o \begin{bmatrix} r_k \\ d_k \end{bmatrix} \\ \begin{bmatrix} y_k \\ z_k \end{bmatrix} &= C^o x_k + D^o \begin{bmatrix} r_k \\ d_k \end{bmatrix} \end{aligned} \tag{5.39}$$

The flight envelope limits can then be represented by lower and upper limits on the z output signals

$$\underline{z} = -\infty, \qquad \overline{z} = b_z - A_z \begin{bmatrix} V_{\text{DStrim}} \\ 0 \\ 0 \end{bmatrix} \tag{5.40}$$

to be used in the soft constraint formulation of (5.25).

Chapter 6

Fault detection filter design

This chapter describes the fault detection filter design process for the Fixed Wing Capstone Flight Demonstration. The contents of this chapter is borrowed from a publication[1] that will be submitted to a journal in the near future and in its format represents a separate entity within this technical report.

6.1 Introduction

6.1.1 The challenge

The *Software Enabled Control* (SEC) program [2] was a research initiative undertaken by DARPA and the U.S. Air Force Research Laboratory (AFRL) to exploit recent developments in software and computing technologies for applications to control systems. The program was geared toward uninhabitated air vehicles (UAVs), and culminated with a flight test during June 2004, where the main technologies were demonstrated in a "simulated" UAV, the T-33/UCAV. The group at the University of Minnesota (UMN), in collaboration with the University of California at Berkeley (UCB), implemented and flight tested a receding horizon control (RHC) algorithm for trajectory tracking. The open control platform (OCP) [18] provided the middleware interface to the T-33/UCAV for implementation of real-time adaptive control algorithms. The flight control system is a safety-critical component of the UAV and should include a level of fault detection and RHC reconfiguration in case a faulty condition has been declared.

The flight testbed to be used was a T-33 jet aircraft. This aircraft was chosen because its avionics package was the same as the X-45 uninhabited combat air vehicle (UCAV), see Figure 6.2.[2] This provided the SEC program with a single aircraft characterized by realistic UAV dynamics *as well as* being fitted with an avionics interface of a true UAV.

To understand the challenge involved in designing a fault detection (FD) filter for the SEC flight test, consider the block diagram of the system illustrated in Figure 6.1. From this block diagram, the signals available for control and FD were the control signal u and the measurement signal y. Hence the *system* of interest is an input/output *black-box* for which a nonlinear model DemoSim was developed specifically for the SEC program by Boeing, who provided the project management for the SEC program, and was provided to the SEC groups. As could be expected, the resulting

[1]R. Ingvalson, H. P. Rotstein, T. Keviczky and G. J. Balas, "Input-Dependent Threshold Function for an Actuator Fault Detection Filter"

[2]The T-33 photo in this figure was obtained from an online photo database. The UCAV photo was obtained from the Boeing Company's corporate web site: www.boeing.com.

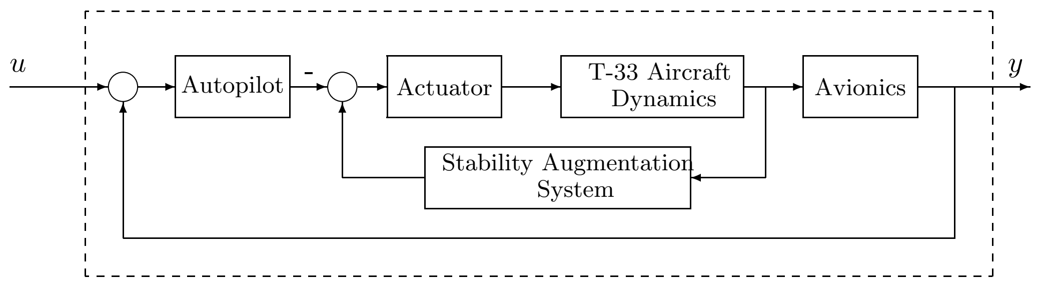


Figure 6.1: Fault detection setup in the SEC experiment. Notice that only the signals u and y are available for processing

Figure 6.2: The T-33 jet aircraft (*left*) and the X-45 UCAV (*right*)

model and system involve complexities that cannot be ignored when doing control and FD design: unknown time-delays, non-linear behavior in the form of various limiters at the autopilot level, plant variations, etc.

The signals internal to the closed-loop system formed by the autopilot, actuators, aircraft dynamics and avionics are not available for design or use in DemoSim or on the actual flying platform, see Figure 6.1. The reason for this limited information is a combination of proprietary and confidential information. Needless to say, the limited information greatly constrains the FD problem. Indeed, almost every approach to FD assumes that either an inexact, but decent, model of the internal dynamics of the system of interest exists *or* that learning from examples is possible. See, e.g. the comprehensive books [19] and [20]. In the SEC case, a model was available but the internal dynamics were not. Because of this, evaluation of the effect of an actuator fault could not be performed. Moreover, faults due to changes in the internal dynamics could not be simulated since the flight simulation for design and testing, provided to the SEC researchers, did not have access to internal dynamics. The challenge was to develop an FD system that could be designed and tested both in hardware-in-the-loop and flight test given a limited set of information and actuation. As discussed in this chapter, the challenge was addressed by using a number of tools, including $\mathcal{H}_\infty$ fault detection, a full nonlinear aircraft simulator for fault models estimation and development of a new threshold function as a detection tool.

The constraints imposed by the SEC final testbed, as artificial as they may seem, are appearing with increasing frequency in practice for a number of reasons. From a system's engineering viewpoint, it is desirable to see each one of the components of a given project as independent as possible. This greatly facilitates the development of a project in terms of responsibilities and performance demonstration, but tends to perturb the information flow, especially if a new sub-systems is added as an after-thought. Moreover, different blocks, e.g. the autopilot and the flight controls, may be developed by different groups on possibly different processors, with the corresponding communication and proprietary concerns. Finally, software re-usability driven by economical and certification considerations, where entire sub-components are taken off-the-shelf, may further restrict the access to relevant signals.

With all of these limitations, it remained the responsibility of the SEC UMN researchers to develop a FD algorithm for the T-33/UCAV flight testbed. The strategy employed for addressing this *closed-loop* FD problem is based primarily on the foundations of robust control theory, specifically that of $\mathcal{H}_\infty$ optimization. Also necessary for this problem was an input-dependent threshold function, for which a novel approach is developed using ideas from the model invalidation literature. The FD algorithm and threshold function were a primary component of the OCP, and once completed they were integrated into the OCP, where the algorithm could be tested by simulation and ultimately flight tested as part of the SEC Capstone Demonstration. This chapter focuses on the work done on the FD algorithm, up to and including the simulation of the UMN/UCB SEC Capstone Demonstration, integrated with the FD algorithm.

6.1.2 Organization of the chapter

A brief background to FD is discussed in the next section, which is followed by a more detailed introduction to the FD SEC problem, Section 6.2. Here, the setup and design of the FD problem are described. Section 6.3 contains the theoretical contribution of the chapter: a function that can be implemented on-line to decide if a fault has occurred. Since the function may provide a conservative criterion, a number of tuning parameters is also introduced, and their effect on the overall performance is discussed.

Section 6.4 contains further modeling details in the SEC problem, and shows the considerations involved in the design of the FD filter and the tuning of the threshold function. Section 6.5 illustrates the performance of the FD scheme in simulations and also in an actual flight test. Finally, Section 6.6 contains the conclusions and suggestions for future work.

Fault detection is understood as the ability to recognize unexpected changes in the functioning of a system, usually resulting from physical failures or breakdowns. Research efforts in fault detection started during the early 1970's and a relatively large body of knowledge is now available. For a review of many results and pointers to the literature, the reader is referred to the books [20] and [19].

Arguably the most straightforward method of dealing with faults is to use redundant subsystems. For instance, a typical commercial aircraft navigation system may have triple-redundant inertial references plus double-redundant air data systems as a navigation sensing suite. A voting scheme can then be implemented to check the performance of the individual sensors and detect abnormal behavior. As another example, the Segway human transporter has a suite of 9 gyros for stabilization/equilibrium purposes.

Hardware redundancy is expensive and usually limited to high-end applications. Hence the drive to replace hardware redundacy by "analytic" redundancy whereby additional knowledge of the systems involved is used instead of actual redundancy. Earlier work in FD concentrated on the conditions for detecting and identifying faults for highly idealized models (see e.g. [21]) and their application in design. The focus of this work is on the trade-off between the ability to detect faults and the level of noise in the measurements. Recently, paralleling the developments in control theory, the effect of uncertainty in the models has been recognized as a major factor affecting the detection of faults. This has given rise to the idea of *robust* fault detection, namely, the ability to detect faults in the presence of model uncertainty. Initially, robustness has been addressed indirectly by first designing a fault detection filter and then applying threshold filters, based upon the assumed uncertainty level, to the *residuals*, signals generated by the FD filters [22, 23]. In particular, [22] attempted to estimate the smallest size of the failure which is detectable in spite of sensor noise and model uncertainty. In later work, model uncertainty was explicitly taken into account and several robust FD filters were proposed [24, 19, 25, 26]. Preliminary applications of these techniques have also been reported in the literature, e.g. [27].

An FD scheme usually consists of two stages: construction of a filter for generating residuals and a decision stage for analyzing the residuals and deciding if a fault has actually occurred. Relatively little has been done in combining robust FD filters with the synthesis of a *robust* threshold strategy. For example, in [28] the optimal threshold function is investigated, where optimality is understood in terms of false-alarm and miss-detection rates. This approach provides a practical solution when the basic trade-off is with measurement noise, but becomes less convenient when measurement noise is small as compared to model uncertainty. The main trade-off for the flight demonstration is between fault detection and model uncertainty. Thus, robust techniques were required for both, the filter design and threshold strategy.

When model uncertainty is large, the residuals generated by any FD filter, due to this uncertainty, are significant; hence, the design of a threshold function capable of handling model uncertainty becomes critical. This chapter presents a solution to this problem using energy-like arguments. The proposed approach has similarities with the work in [29], where a *model invalidation* argument is used to decide whether a fault has occurred or not. The main disadvantage of the model invalidation approach is that it cannot be implemented on-line since it involves the solution of an optimization problem of monotonically increasing size. Exploiting the special structure of the SEC problem, an alternative criterion may be formulated for the fault detection threshold filter which dramatically reduces the computation cost and hence can be implemented in real-time.

6.2 The $\mathcal{H}_\infty$ fault detection filter

As stated earlier, the FD filter design was accomplished using techniques well known to the robust control community. The basic setup for the FD problem under consideration is shown in Figure 6.3. One familiar to robust control will immediately recognize it as a typical "uncertain" system. The block G_{nom} represents the nominal model of the system, and the block F is the filter to be designed. The W-blocks and the Δ-blocks represent the uncertain aspects of the system, and are known as the *weighting functions* and *perturbation matrices*, respectively. The primary difference in our approach is that we solved an *open-loop* $\mathcal{H}_\infty$ optimization problem. Whereas, control designers typically use $\mathcal{H}_\infty$ to solve *closed-loop* problems. The proposed FD problem is open-loop because the design block, the filter F, does not lie in a feedback path. Even though this is not a typical $\mathcal{H}_\infty$ problem, it is still well suited for it, because the basic assumptions for $\mathcal{H}_\infty$ methods are easily satisfied for our proposed FD problem. For example, one assumption requires that all system interconnection transfer functions (see Figure 6.4) are stable and proper which is satisfied for the given problem formulation.

As seen in Figure 6.3 there are two Δ blocks – also known as *uncertainty* matrices – in consideration. In robust control theory, these uncertainty blocks are unknown and, in general, complex-unstructured matrices. They are used to model parametric uncertainties, or can also be used to

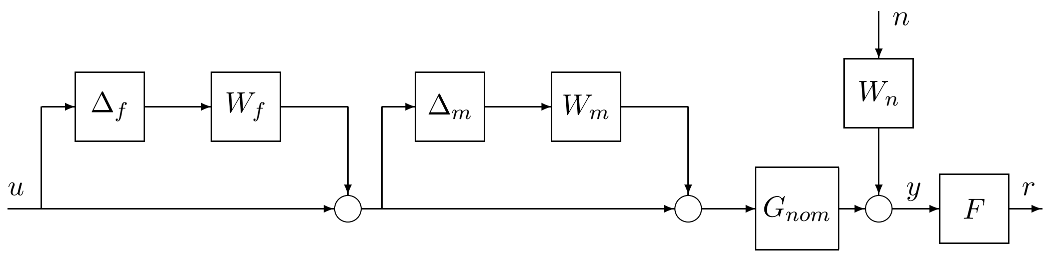


Figure 6.3: Uncertain representation of the fault detection system

represent uncertainties that are more difficult to define, such as system modeling errors. Additionally, it is usually assumed that each Δ is norm-bounded by 1, i.e. $\|\Delta\| < 1$. With this assumption, the weighting functions W must be scaled appropriately to represent the magnitude of the signals. Although, many robust control methods approach uncertainty in a similar manner, there are certain robust techniques that make additional assumptions about the uncertainty blocks. For instance, in μ-synthesis the Δ-blocks are assumed to have a *structured* form, specifically a block-diagonal structure. $\mathcal{H}_\infty$ methods, on the other hand, assume unstructured uncertainties.

6.2.1 $\mathcal{H}_\infty$ problem formulation

The first block Δ_f is associated with the perturbations due to the fault model. The second block Δ_m represents the uncertainty involved in describing the physical system via a mathematical model, i.e. modeling errors. Lastly, the block W_n is included to encompass any high-frequency errors that may enter the system, such as noise. The weighting functions W_f and W_m allow the introduction of *a priori* knowledge on the nature of the fault and the plant uncertainty. As mentioned in the introduction, the objective of the fault detection algorithm is to detect a fault during a closed-loop operation. These two uncertainty blocks, see Figure 6.3, represent an input-output effort at modeling plant uncertainty and fault dynamics. This setup clearly highlights that if there is no separation between the frequency associated with the faults and the frequency of the model uncertainty, then one cannot distinguish between them using input-output signals. Therefore, in the proposed approach it is assumed that the faults and the modeling errors do not share the exact same frequency characteristics.

Following the idea of $\mathcal{H}_\infty$ norm-based fault detection (see, e.g. [27]), the configuration in Figure 6.3 is re-drawn as shown in Figure 6.4. In this figure, the uncertainty blocks Δ_f and Δ_m have been removed and two new fictitious signals f and d are included. The objective of the $\mathcal{H}_\infty$ fault detection design is to synthesize a filter F such that the transfer matrix $\mathcal{T}_{rw}$ between the *extended* input:

$$w = \begin{bmatrix} f \\ u \\ d \end{bmatrix} \tag{6.1}$$

and the *residual error* $\tilde{r}$ are small in an $\mathcal{H}_\infty$ sense. This would imply, in particular, that the *residual* r of the FD filter tracks the fault $\hat{f} = W_f f$. Notice that the particular uncertainty model selected also covers disturbances at the plant input.

One of the advantages of this formulation is that standard tools for $\mathcal{H}_\infty$ control design can be used to design the filter. For other alternative approaches to $\mathcal{H}_\infty$ fault detection, see Chapter IX in [19].

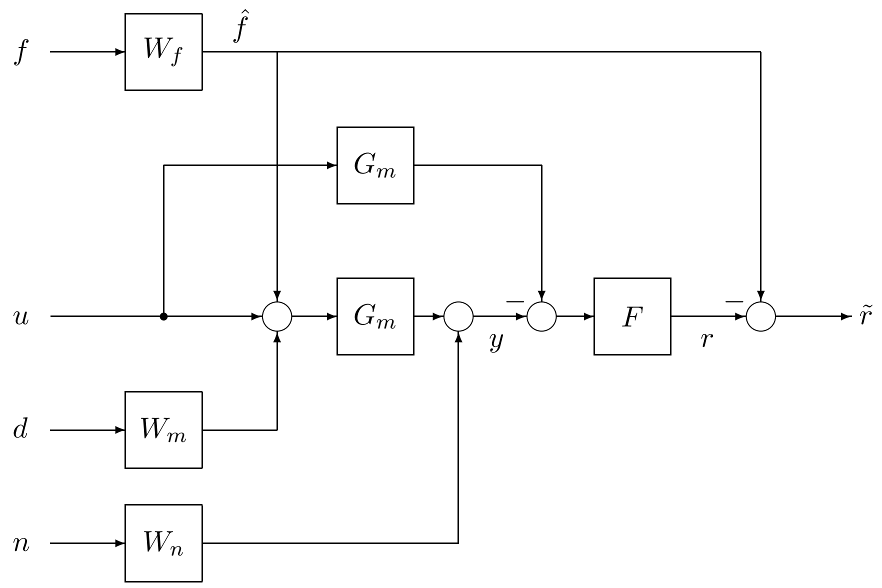


Figure 6.4: Setup for the $\mathcal{H}_\infty$ fault detection filter design. If the transfer function from f to $\tilde{r}$ is "small," then the residual r tracks the fault signal $\hat{f}$.

6.2.2 $\mathcal{H}_\infty$ design trade-offs

In order to better understand the trade-off involved in the $\mathcal{H}_\infty$ optimization problem, one should look closely at the effect of the system inputs on the residual r. These relations can be obtained from the system interconnection, shown in Figure 6.4. From this figure, the residual is related to the inputs by

$$\begin{aligned} r &= F\left[G_m\left(W_f f + W_m d\right) + W_n n\right] - W_f f \\ &= \left(FG_m - I\right) W_f f + FG_m W_m d + FW_n n \end{aligned} \tag{6.2}$$

Equation (6.2) shows the trade-off involved in the proposed fault detection problem. To track the fault, the filter F should invert the plant G_m in the bandwidth of the fault as determined by W_f. At the same time, the filter should be small enough to attenuate the effect of noise $W_n n$ and plant uncertainty (and input disturbance) $G_m W_m d$. If there is adequate frequency separation for these two, then a good solution can be obtained by approximating G_m^{-1} in the frequency band of interest and then "rolling-off" to prevent disturbance and noise affect the residuals.

Notice that if there is significant overlap in the bands of W_f and $G_m W_m$ or W_n, then no FD filter will be able to isolate a fault adequately. This should be a major concern in the design of a FD filter for a closed-loop system. In many cases of practical importance, however, separation will hold approximately and the $\mathcal{H}_\infty$ FD filter produces the best compromise in terms of the $\mathcal{H}_\infty$ norm. This observation motivates the topic of the next section: design of fault decision criterion.

6.3 Is there a fault?

In the absence of a clear frequency separation, plant uncertainty will result in a non-negligible residual even if no fault is present. The use of a simple thresholding strategy will hence give rise to a large number of false alarms or, if the threshold value is increased, miss-detections. This section describes an input-dependent threshold function that exploits the additional information assumed for the system. From Figure 6.3,

$$r = F\left(\left[G_m\left(I+\Delta_m W_m\right)\left(I+\Delta_f W_f\right) - G_m\right]u + W_n n\right) \tag{6.3}$$

Following the model invalidation paradigm (see, e.g. [30]) a fault will not be declared if there exist Δ_m stable and n such that $\|\Delta_m\|_\infty \le 1$, $\|n\|_2 \le 1$, and:

$$r = F\left(G_m \Delta_m W_m u + W_n n\right). \tag{6.4}$$

Namely, there exists an uncertainty and a noise consistent with the problem that can "explain" the observed data.

As shown in [29] in the context of fault detection, given $(u(\tau), r(\tau))$ for $\tau = 0, \cdots, t$, the problem of verifying the existence of a norm bounded uncertainty Δ_m subject to (6.4) can be transformed into an optimization problem with a linear matrix inequality constraint. This fact has interesting consequences for *off-line* fault detection, but involves the solution of a monotonously increasing optimization problem and hence cannot be used for on-line computations. The objective of this section is to present an alternative, albeit weaker, criterion suitable for real-time applications. Consider the projection operator:

$$(T_{t_1}^{t_2} u)(\tau) = \begin{cases} u(\tau) & t_1 \le \tau < t_2 \\ 0 & \text{elsewhere} \end{cases} \tag{6.5}$$

with the simplifying notation $T^t = T_0^t$. Then (6.4) can be replace by the stronger condition:

$$\left\|T^t r\right\|_2^2 \le \left\|T^t F G_m \Delta_m W_m u\right\|_2^2 + \left\|T^t F W_n n\right\|_2^2 \tag{6.6}$$

for each time instant t. Given a causal operator G, one has $\left\|T^t G u\right\|_2 \le \left\|G T^t u\right\|_2$, and so (6.6) implies

$$\left\|T^t r\right\|_2^2 \le \left\|F G_m \Delta_m W_m T^t u\right\|_2^2 + \left\|F W_n T^t n\right\|_2^2 \tag{6.7}$$

Given the assumptions $\|\Delta_m\|_\infty \le 1$, $\|n\|_2 \le 1$,

$$\left\|T^t r\right\|_2^2 \le \|F G_m W_m\|_\infty^2 \left\|T^t u\right\|_2^2 + \|F W_n\|_\infty^2 \tag{6.8}$$

Since for a given design the transfer matrices above are constant, (6.8) can be re-written as:

$$\left\|T^t r\right\|_2^2 \le \alpha^2 \left\|T^t u\right\|_2^2 + \beta^2 \tag{6.9}$$

where:

$$\begin{aligned} \alpha &\doteq \|F G_m W_m\|_\infty \\ \beta &\doteq \|F W_n\|_\infty . \end{aligned}$$

Condition (6.9) can be used for fault detection in real-time applications. Indeed, one can compute the *threshold signal*:

$$f(t) = \left\|T^t r\right\|_2^2 - \alpha^2 \left\|T^t u\right\|_2^2 - \beta^2 \tag{6.10}$$

for each time t and declare a fault if $f(t) > 0$ at some time instant t.

Notice that if (6.6) holds for each time t, then (6.4) will also hold true. The opposite is not necessarily true for time-invariant uncertainties Δ_m, and in general the former condition will be much more restrictive. In addition to the above, the use of the triangular inequality and the norm-bounding properties makes (6.9) a sufficient, but in general far from necessary condition for (6.4). Hence (6.10) must be relaxed to make it useful in practice.

6.3.1 Relaxing the threshold condition

In addition to the gap between (6.4) and (6.7), there are good reasons to relax the constraint on $T^t r$ associated with (6.7) by introducing new design parameters. Indeed, during the fault detection filter design stage, the weighting functions W_m, W_n may be modified to achieve desirable behaviors of the filter not necessarily captured by the $\mathcal{H}_\infty$ formulation (e.g. a roll-off rate). This is especially true for the noise signal n, which is often stochastic in nature and hence can only be approximately modeled in the $\mathcal{H}_\infty$ design. Moreover, being a worst-case criteria, $\mathcal{H}_\infty$ and the threshold strategy defined above are ill-suited for trading-off false-alarm and miss-detection rates, which are central in any fault detection design.

Two design parameters are introduced in (6.9) in an attempt to compensate the difficulties mentioned above. First, the *running-norm*

$$\left\|T^t u\right\|_2^2 = \sum_{\tau=0}^{t-1} \|u(\tau)\|^2$$

is modified by introducing the *forgetting factor* $\kappa < 1$:

$$S^t u(t)^2 \doteq \sum_{\tau=0}^{t-1} \|\kappa^{t-1-\tau} u(\tau)\|^2$$

This exponential decay on the influence of "old" data can be used for both norms in (6.10). The usage of the forgetting factor has two main consequences:

1. The threshold strategy (6.10) tends to become insensitive to faults if W_m is relatively large as compared to the actual plant/model mismatch observed in practice, especially when the function is computed over long time intervals. This may give rise to fault miss-detection.

2. At some points during the operation of the system, one may want to allow for relatively large plant/model mismatch, not captured by the uncertainty bound W_m. In these instances, (6.10) may result in a *false-alarm* since the unduly mismatch effectively behaves as a fault. This difficulty may be overcome by temporarily ignoring the value achieved by f while the large mismatch is present, and then allowing the exponential weight to bring f back to negative.

Second, the noise level β is replaced by a tuning parameter $\overline{\beta}$ that can be used to reduce the false-alarm rate. This parameter can be tuned by analyzing $(u(t), r(t))$ data records under *benign* conditions, e.g. operating points where model/plant mismatch is small.

6.4 Fault detection for the SEC program

The fault detection design for the SEC program is based on the simulation *DemoSim* of the T-33/UCAV aircraft provided by Boeing to the SEC research groups. DemoSim is a black-box simulation of the T-33/UCAV augmented aircraft. As mentioned in the introduction, although one can provide inputs, modify a few functioning parameters, and observe output logs there is no access to the internal signals, dynamics, or logic of the simulator. As a consequence of and in addition to standard model uncertainty, one needs to address the fact that DemoSim's autopilot implementation and internal discrete logic (saturation levels, limiters) are unknown. To make things more difficult from a fault detection viewpoint, the inputs to DemoSim are actually autopilot commands and hence only guidance-level (i.e. kinematic) control of the vehicle is possible.

Input Channels	Output Measurements
V_{cmd} – Velocity (ft/sec)	V_{meas} – Velocity (ft/sec)
$\dot{\chi}_{cmd}$ – Heading Rate (deg/sec)	χ_{meas} – Heading (deg)
$\dot{h}_{cmd}$ – Altitude Rate (ft/sec)	γ_{meas} – Flight Path Angle (deg)

Table 6.1: Input and output signals for the T-33/UCAV Testbed

The set of input and output signals used for control are presented in Table 6.1. The fault detection scheme described in the previous sections was applied to the single-input, single-output lateral-directional dynamics subsystem from $\dot{\chi}_{cmd}$ to χ_{meas}. This transfer function is referred to as the χ-channel. This subsystem was selected since χ_{meas} would be essentially decoupled from the other two reference inputs when control commands are restricted to lie within tolerable limits. Thus, the FD problem could be simplified to a single-input, single-output (SISO) problem.

A linear model was identified from I/O data obtained using DemoSim's χ-channel[3]. The model identified was based upon the response of DemoSim to a 0.5 deg/sec step command. A third-order autoregressive exogenous model was identified using the Matlab System Identification Toolbox. The identified discrete time transfer function with sampling period of 0.1 seconds was

$$G_m = \frac{2.48 \cdot 10^{-3} z^3}{(z - 0.98)(z^2 - 1.89z + 0.90)}$$

This is the model used in all subsequent analysis and design. The frequency response of this model is shown in Figure 6.5.

6.4.1 Fault model

DemoSim does not provide a way of internally simulating a fault and hence it was necessary to simulate a fault by either corrupting the input or output channel of DemoSim in such a way that the resulting output resembled a faulty system. There are many ways in which this can be done, the approach used involved insertion of a multiplicative input fault, as shown in Figure 6.6. In this figure, u is the actual or *true* command to be fed to DemoSim, and $\hat{u}$ is the "corrupted" command which will produce the "faulty" output. The "no fault" scenario corresponds to the case when the "Fault On" switch is open, $\hat{u} = u$.

Since only the lateral motion was being considered in this FD problem, it was necessary to only look at faults which would have strong coupling to this channel. One such fault would be an aileron actuator fault. Hence, W_f was designed such that the overall system (i.e. with the multiplicative fault input) would behave as if a true aileron actuator fault occurred.

Physically, an aileron actuator fault may result in changing the dynamics of the actuator, e.g. a change in the damping or natural frequency of the actuator. These changes could result from actual damage, i.e. faults, to the physical system – such as a loss of hydraulic pressure or damage to the aileron control surface.

[3]For identification and simulation, it was actually the numerically differentiated output of DemoSim that was used. That is, the model identified was based on a $\dot{\chi}_{cmd} \rightarrow \dot{\chi}_{out}$ response of DemoSim.

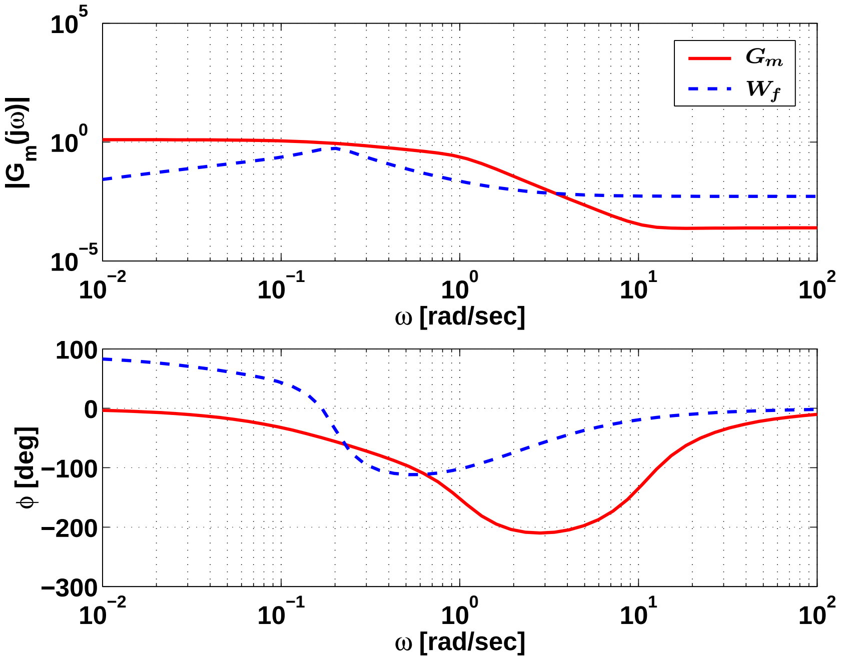


Figure 6.5: Bode plots of the plant model G_m and the fault model W_f

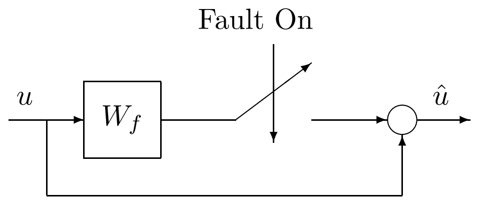


Figure 6.6: Multiplicative fault input

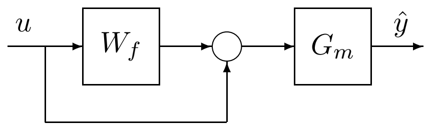


Figure 6.7: Fault identification setup

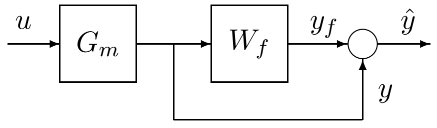


Figure 6.8: Equivalent representation for fault ID setup

6.4.1.1 Identification of W_f

Identification of the fault filter, W_f, was performed using a Boeing 747 aircraft simulation.[4] This simulation allowed individual parameters of the actuators to be varied. From the simulation, a nominal response *and* a faulty response was generated which was not possible with DemoSim. The fault filter W_f was identified using these responses as described in the following paragraphs. W_f was generalized and scaled according to the dynamics of DemoSim's χ-channel relative to the Boeing 747 simulation.

Denote the faulty system as $\hat{G}_m$. Assuming a multiplicative input fault, as in Figure 6.6, the response $\hat{y}$ of the true faulty system $\hat{G}_m$ was approximated by

$$\hat{y} = G_m(1 + W_f)u \tag{6.11}$$

It now remains to identify W_f, such that the above equation is a good approximation to the faulty system. The block diagram for this equation is given in Figure 6.7. Assuming that all systems are linear, we can redraw Figure 6.7 with G_m on the input side, as in Figure 6.8. From this figure, it is clear that $\hat{y} = y + y_f$, where y_f is the response of W_f to input y. Thus, the relationship can be written as $z = \hat{y} - y$, and since both y and $\hat{y}$ are know from the simulation, the frequency response of W_f can be calculated as

$$W_f = \frac{\text{FFT}\,(\hat{y} - y)}{\text{FFT}\,y} \tag{6.12}$$

where FFT denotes the discrete-time Fast Fourier Transform.

Since a true fault in a physical system usually cannot be characterized precisely by one response $\hat{y}$ – as (6.12) would suggest – it was necessary to generate a family of faulty responses, $\hat{y}$. This collection of faulty responses needs to be a good representation of the entire set of possible faulty responses. An individual weighting function W_f was determined for each faulty response, to this collection of weighting functions a hand-fit upper bound $\overline{W}_f$ was determined. The $\overline{W}_f$ upper bound was used as the final weight.

[4]In the remainder of this section, the Boeing 747 simulation will be referred to as "the simulation". Any references to DemoSim will be explicit.

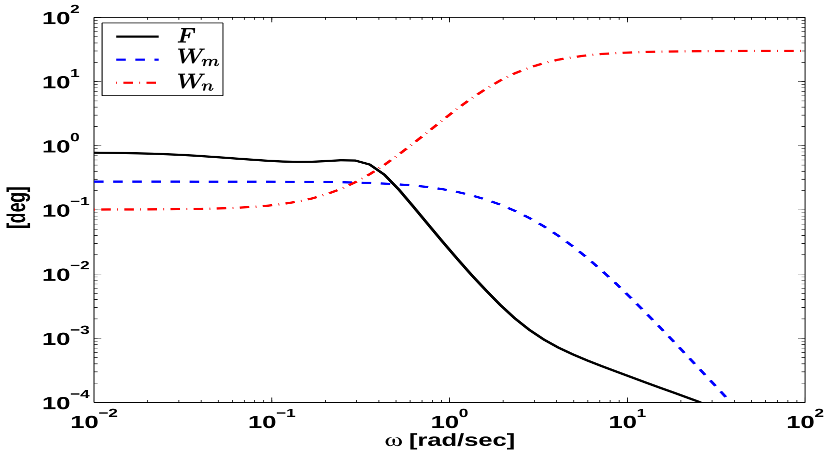


Figure 6.9: $\mathcal{H}_\infty$ Filter design and weighting functions

6.4.1.2 Generation of nominal and faulty responses

As mentioned earlier, the fault being simulated is an aileron actuator fault. In the simulation, this actuator is modeled as a second-order system with a nominal natural frequency of $\omega_m = 16.4$ rad/sec and damping ratio $\zeta_m = 0.67$. Using these parameters, the *nominal* response y of (6.12) was generated with the simulation.

The collection of the faulty cases was chosen with natural frequency $\omega_f \in [15, 3]$ rad/sec and damping ratios from ranges $\zeta_f \in [0.7, 1.5]$. This was done to cause the actuator to exhibit a slower response; and thus, more characteristic of a truly faulty or degraded actuator. A number of these fault cases were simulated to generate faulty responses $\hat{y}$, with (6.12) used to generate a frequency response for each fault case. The upper bound fit $\overline{W}_f$ to these responses was the weight used in design. Future reference to W_f will be understood to refer to this upper bound.

Since the dynamics of the Boeing 747 simulation were not the same as the dynamics of DemoSim it was necessary to shift the frequency of W_f so that it's response lied within the bandwidth of DemoSim's linear model G_m. The gain was also adjusted, and was later used as a parameter to vary the intensity of the fault. The resulting transfer function was

$$W_f = \frac{2.6 \cdot 10^{-3} s(s+10)(s+5)(s+0.3)}{(s+0.9)(s+0.2)(s^2+0.416s+0.64)} \,.$$

The frequency response of W_f is shown Figure 6.5.

6.4.1.3 Filter design results

The characteristics of the final $\mathcal{H}_\infty$ filter design and the expected performance are briefly discussed here. Figure 6.9 shows the results of the $\mathcal{H}_\infty$ optimization of the fault detection filter. Included in the plot are the $\mathcal{H}_\infty$ optimal filter F, as well as the weighting functions W_p and W_n used in the final design.

From Figure 6.9, it is clear that F is essentially a low-pass filter. This in not surprising since the problem has been formulated as a disturbance rejection problem. The roll-off also occurs at a frequency where expected, around 0.4 rad/sec. This is near the frequency in which the noise W_n

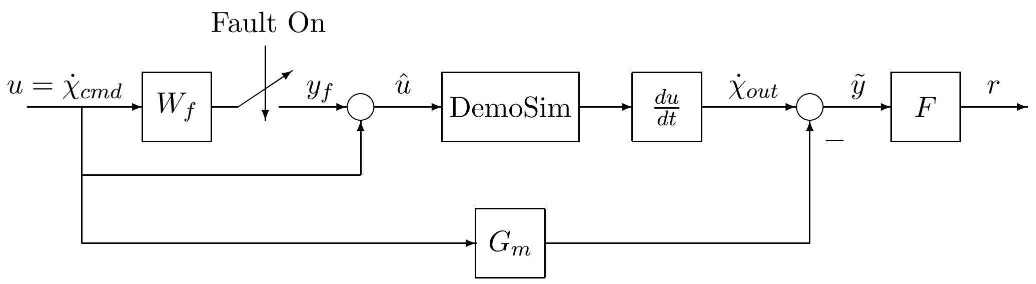


Figure 6.10: FD filter test simulation diagram

begins to increase. Further discussion of the performance of this filter is presented in subsequent sections.

6.5 Simulation results

Section 6.5.1 contains a summary of the simulation results obtained from integrating the FD filter with DemoSim. These results demonstrate the performance of the FD filter by issuing an *a priori* command signal.

Following these results, Section 6.5.2 presents the simulation results of integrating the FD filter into the UMN/UCB final SEC Capstone Demonstration. In this simulation, the commands to DemoSim were generated by the RHC algorithm developed for the T-33/UCAV testbed by UMN researchers and described in Chapter 5. Section 6.5.2 also contains an overview of the entire UMN/UCB SEC Capstone Flight Demonstration Experiment as well as a discussion of the Capstone simulation environment.

6.5.1 Simulation of the FD Filter and DemoSim

Based on the $\mathcal{H}_\infty$ design interconnection shown in Figure 6.4, a simulation environment was constructed to test the fault detection filter. The block diagram for the simulation is shown in Figure 6.10.

In this simulation, 0.5 deg/sec $\dot{\chi}$ step commands are issued to DemoSim – a positive step at $t = 0$ sec was followed by a negative step at $t = 100$ sec. The fault was turned on 70 seconds after the positive step command. The simulation results are shown in Figure 6.11. These plots show the command u, the faulty command $\hat{u}$, the model "error" $\tilde{y}$, and the residual r and fault signal y_f, respectively.

According to the simulation experiment setup, the filter's response to the positive step shows the performance in the presence of *no fault*. Whereas, the negative step shows the filter's performance in the presence of a fault. Also, note from Figure 6.5 that W_f rolls off at low frequency; thus, the fault will only affect the output of DemoSim during the *transient* phase of the input command u. This is clear from the plot of $\hat{u}$ in Figure 6.11, since no change is seen immediately in this signal when the fault is turned on at 70 sec. It is only after the second step, that a change is noticed in the command – note the sinusoidal behavior of $\hat{u}$ after the negative step.

From the plot of the residual in Figure 6.11, it is quite clear that it is excited more during the negative step than the positive step, due to the effect of the fault. Also, plotted on the same axes is the output y_f of W_f, see Figure 6.10. In the $\mathcal{H}_\infty$ optimization, one of the performance objectives

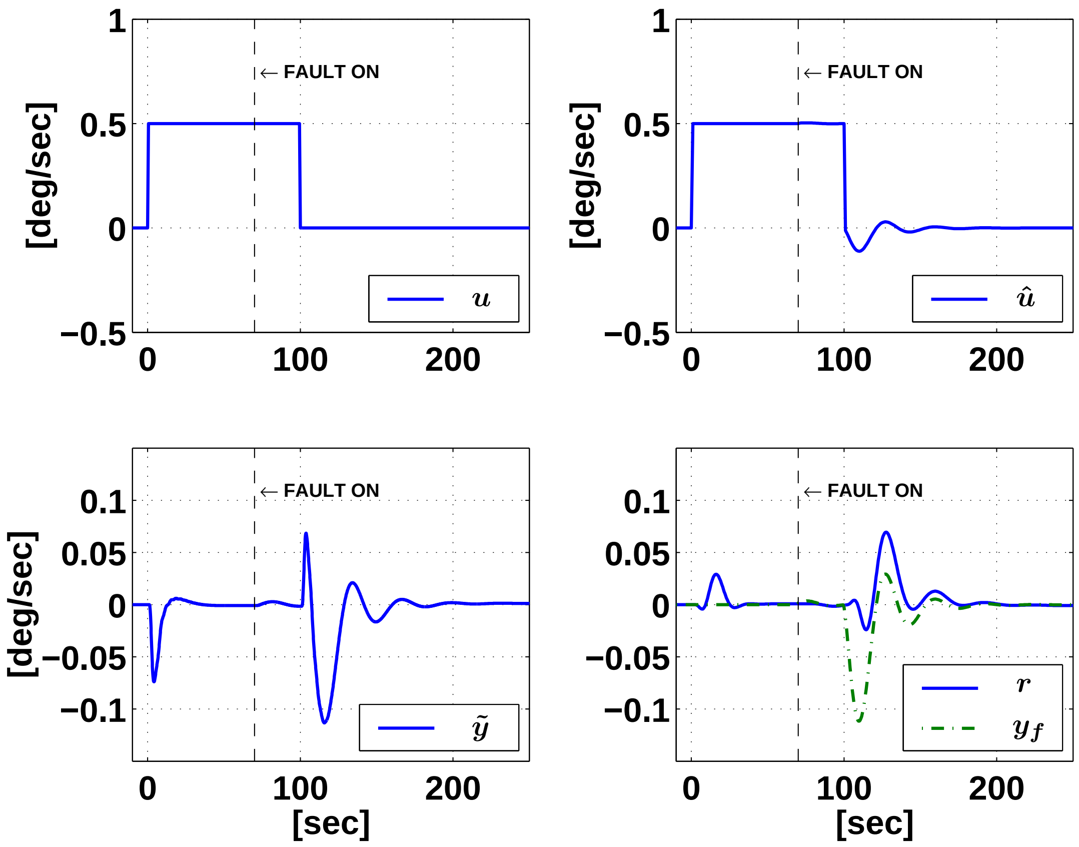


Figure 6.11: Simulation results

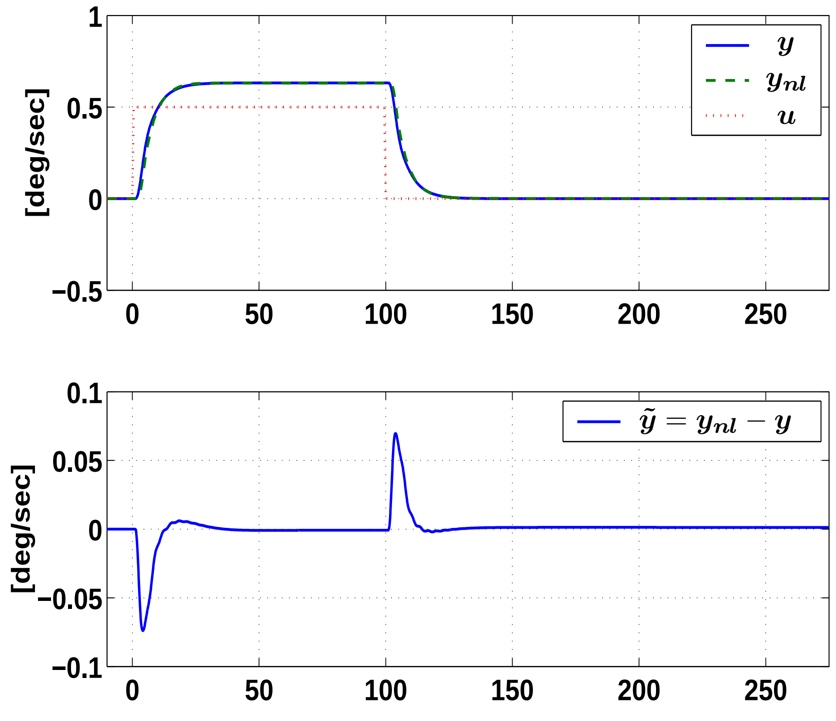


Figure 6.12: $\dot{\chi}$ Responses and model mismatch

was such that r was to track the fault y_f. Though, the tracking error is not always small, the filter's response converges toward the response y_f as we desire.

One of the reasons why the residual tracking error is not as small as we would like is due to the *model mismatch* inherent in the system. To understand the mismatch between the linear model and DemoSim, a simulation identical to that shown in Figure 6.11 was performed, *without fault.* The results of this "model mismatch run" are shown in Figure 6.12. The *non-faulted* $\dot{\chi}$ responses of the nonlinear DemoSim and of the linear model of DemoSim G_m – y_{nl} and y, respectively – are shown in the first plot of Figure 6.12. Shown in the second plot is the output error $\tilde{y} = y_{nl} - y$, also know as the *model mismatch*[5]. The signal $\tilde{y}$ is the signal that is fed to the filter F. From the plot of $\tilde{y}$ one can see that the signal does not have much high frequency content, therefore it was difficult to filter out its effect, because its frequency band lied in the same range as that of the fault. In other words, the system lacked sufficient frequency separation between the model mismatch and the fault. This trade-off forced a design which had poor tracking performance, i.e. of the filter residual r and the fault signal $\hat{f}$ (see Section 6.2 for definitions), in order to have greater ability to detect the fault.

Knowing that $\tilde{y}$ (see Figure 6.10) is a measure of the model mismatch in the system, one can gain insight into how the filter responds when there is model mismatch and no fault by looking at the response to the positive step command. That is, $\tilde{y}(t)$ for $t < 70$ sec is precisely the model mismatch in the system, and its effect is seen as a small jump in the residual around time $t = 10$ sec, see the plot of r in Figure 6.11.

As mentioned earlier, one of the main trade-offs in the filter design was between the fault

[5]Note that $\tilde{y}$ is referred to as the "model mismatch" in this instance *only.* This is because the fault has been turned *off*, and in this situation $\tilde{y}$ is truly a measure of the model mismatch, but in general it is not.

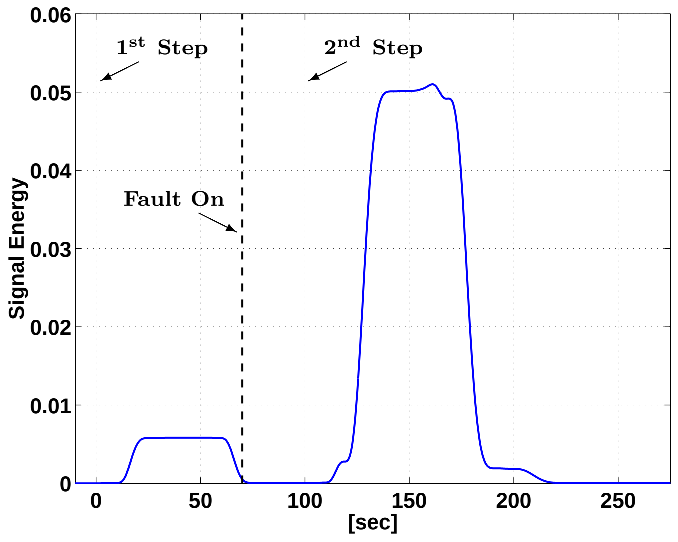


Figure 6.13: Running integral of residual energy

tracking accuracy of the filter and the extent to which the model mismatch is rejected. Shown in Figure 6.13 is a 5 sec running integral of the energy of the residual r (see Figure 6.11). Clearly, from Figure 6.13, the energy in the residual is much greater after the second step than the first step. If we denote the peak energy level after the first step as E_1 and the peak energy level after the second step as E_2. Then we can define the difference $\Delta E = E_2 - E_1$. While designing and testing the filters, this difference ΔE was *worse*, i.e. *smaller*, for filters which had better fault tracking. Thus, verifying our proposition that better tracking leads to poorer model mismatch rejection. The final filter design exhibited the greatest separation in terms of ΔE. The simulation results of Section 6.5.1 are based upon this design.

6.5.2 Simulation of the full UMN/UCB SEC Capstone Demonstration

Further verification of the FD algorithm was obtained by simulation of the full flight experiment demonstration, that is, by incorporating all UMN/UCB developed technologies: the RHC controller, the RHC API, and the FD algorithm. Herein we will focus our attention on the results and performance of the combined FD algorithm and threshold function. But first, let us present a brief overview of the UMN/UCB SEC Capstone Flight Demonstration Experiment.

6.5.2.1 UMN/UCB SEC Capstone Demonstration overview

Outlined below are the events of the SEC UMN/UCB flight demonstration experiment plan. The actual flight test took place in June 2004. Figure 6.14 is a diagram of the flight demonstration test range and the UMN/UCB flight plan. The main part of the flight experiment is divided into three segments: *Phase I* – pop-up threat avoidance, *Phase II* – target engagement, and *Phase III* – fault insertion and detection.

Prior to the main three phases is the *Initialization Phase*. It is a set of conditional procedures that the pilot must undertake to ensure a safe transition of control from the autopilot to the RHC algorithm. The main objective is that the T-33/UCAV must be flying within tolerable limits *and* at the proper altitude and speed at the time the RHC is engaged. Since the flight plan and trajectory

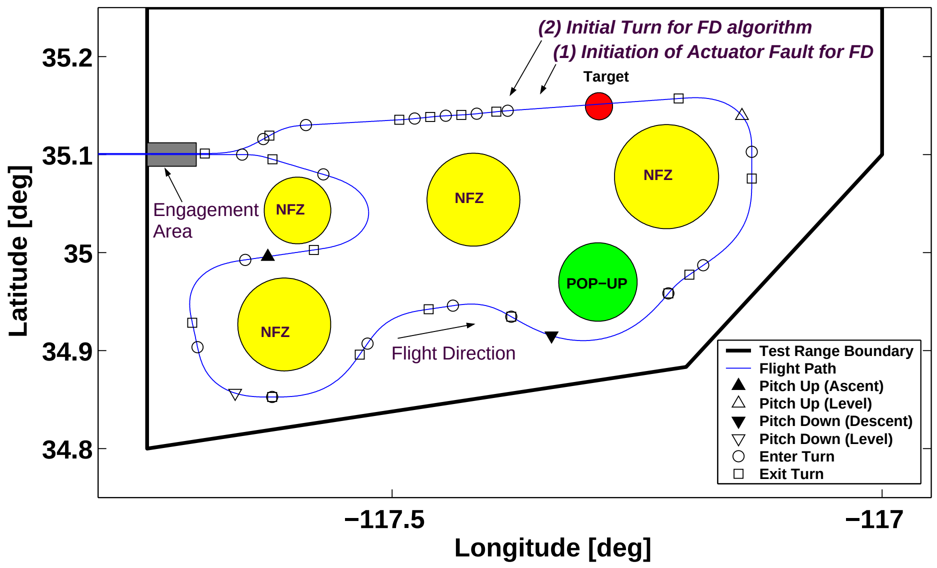


Figure 6.14: UMN/UCB SEC Capstone Demonstration experiment plan

are defined relative to the point the RHC is engaged, care must also be taken that the engagement happens in a timely manner so that the trajectory does not defy the bounds of the test range. For this purpose an *Engagement Area* was defined near the ingress point of the test range for the T-33/UCAV. Thus, if the aircraft is flying with a westerly track the flight experiment will proceed if the RHC is engaged in this *Engagement Area*, see Figure 6.14.

Phase I of the flight demonstration experiment involves trajectory tracking and obstacle avoidance while proceeding toward a predefined target. In this phase the reference trajectory flies around no-fly-zone obstacles (NFZs, see Figure 6.14), and also flies over a potential pop-up obstacle, which may or may not appear depending upon the decision of the flight experiment manager. In the case where the pop-up is inserted, the flight path will be replanned to avoid the pop-up, and the T-33/UCAV will continue toward the target.

The next stage, *Phase II*, begins after the aircraft passed the pop-up location. Whether or not the pop-up threat is inserted, the T-33/UCAV will continue its way towards the target for engagement. Target engagement occurs when the T-33/UCAV flies over the target. This event signifies the end of *Phase II*.

The last phase, *Phase III*, is the flight segment of most interest to the fault detection component. This is the part of the experiment where the aileron actuator fault is inserted,[6] and the fault detection algorithm, implementing the threshold strategy of Section 6.3, is used to detect the fault.

Within 30 seconds of target engagement, the fault is to be turned on, i.e. the input command u will be corrupted, as in Figure 6.6, to simulate an aileron actuator fault (see Section 6.4.1). Immediately during and after the target flyover, the aircraft should be in straight and level flight. Thus, a fault should not be detected until the onset of a turn. This FD part of the flight demonstration will be discussed later.

See Chapter 8 for further details on the complete UMN/UCB flight demonstration experiment results.

[6]See Section 6.4.1 for details on how this is accomplished.

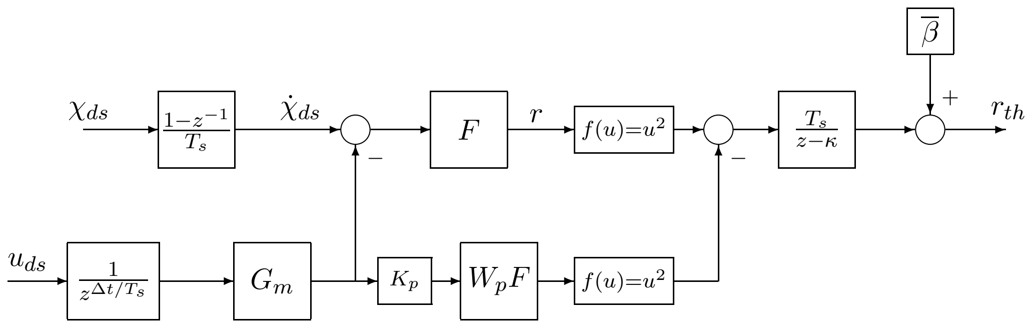


Figure 6.15: Simulation setup for the fault detection algorithm

6.5.2.2 Simulation environment of the UMN/UCB Capstone Demonstration

Figure 6.14 shows an overview of the UMN/UCB flight test scenario including the initiation of the fault. This simulation environment very closely follows what was flown during the actual flight demonstration in June of 2004. There are two notable differences. First, the T-33/UCAV was replaced with the nonlinear, high-fidelity DemoSim executable model. Second, the RHC and the FD algorithms have been implemented in a Matlab/Simulink environment which interfaces to the OCP and DemoSim.

The simulation implementation of the FD algorithm, the $\mathcal{H}_\infty$ filter (see Section 6.4) and the threshold function (see Section 6.3), is outlined in the discrete-time block diagram of Figure 6.15. The inputs to the algorithm are χ_{ds} and u_{ds}, DemoSim's χ output and $\dot{\chi}$-channel control command, respectively. The output is r_{th}, denoting the *threshold* residual, which is in comparison to the $\mathcal{H}_\infty$ FD residual r. Table 6.2 is a description of the signals and blocks of this figure. It contains both the constant parameters and the design parameters of the algorithm. The Table also includes the final values used for the design parameters.

Constant	**Description**
F	$\mathcal{H}_\infty$ Fault Detection Filter
G_m	Identified LTI Model of DemoSim
W_p	The $\mathcal{H}_\infty$ Model Uncertainty Weighting Function
T_s	Algorithm Sampling Period
Δt	χ-Channel Input Time Delay of DemoSim

Design Parameter	**Description**	**Value**
κ	Forgetting Factor	0.95
$\overline{\beta}$	System Noise Level	1.9×10^{-3}
K_p	Model Uncertainty Tuning Gain	0.5

Table 6.2: FD algorithm constants and design parameters

As mentioned earlier, the FD algorithm, as well as the RHC algorithm, was implemented and tested in a Matlab/Simulink environment. This environment is called the UAV Control Interface, and it was a tool provided with the OCP. It allowed the SEC researchers a simple means for implementing and testing their algorithms within the OCP framework, without the need to generate native OCP code (C++) for all of the algorithms. The UAV Control Interface was only used by SEC researchers for testing of algorithms. All final flight code was ultimately coded into C++ and merged into the OCP[7] for the flight test.

6.5.2.3 Capstone Demonstration simulation results

Testing the performance of the FD algorithm illustrated in Figure 6.15, required a trajectory that would cause the RHC controller to issue $\dot{\chi}$ commands within the range validity for the FD algorithm. This restriction was that $u \leq |0.5|$ deg/s.[8] Based upon the expected – constant and nominal – velocity, a way-point trajectory was designed such that the heading rate $\dot{\chi}$ would fall in this range. The trajectory for the FD segment of the flight is a series of S-turns with a turn rate of $\leq |0.2|$ deg/s. The onset of these turns is shown in Figure 6.14.

For clarity to the reader, it is worthwhile to mention again the inputs to the FD algorithm: χ_{ds} and u_{ds}. The input χ_{ds} is the χ-channel output of DemoSim. The input u_{ds} is the *nominal* χ-channel RHC controller command. The FD algorithm is trying to detect the cases when DemoSim exhibits a faulty response. If one recalls, this faulty response is obtained by issuing a corrupted command $\hat{u}_{ds}$ instead of the nominal command u_{ds}. The corrupted signal is obtained by a multiplicative fault input such that $\hat{u}_{ds} = (1 + W_f)u_{ds}$, see Figure 6.6 and Section 6.4.1 for more information. But $\hat{u}_{ds}$ can be viewed as an auxiliary/internal variable and is invisible to the FD algorithm. For the FD algorithm u is a $\dot{\chi}$ command in deg/s.

The Figures 6.16 and 6.17 are from a run of the full simulation with the RHC controller, RHC API, and the FD algorithm. The time scales have been shifted relative to the time when the fault was turned on t_{on}. This was timed to start just before a series of turns which will excite the fault for detection.[9] The time that the turns begin t_{turn} is 20 sec. The $\dot{\chi}$ commands and response of DemoSim for the full simulation run are shown in Figure 6.16.

The simulated output of the threshold generator (displayed in Figure 6.15) is shown in Figure 6.17. This plot is the threshold function's residual r_{th}. Since the trajectory was designed such that DemoSim would be in straight and level flight at t_{on}, no activity is expected form the FD algorithm until it is excited. This is designed to occur at t_{turn}, when the trajectory begins its series of turns. From this figure this appears to have happened – albeit, with a time delay. Some of this delay is clear from Figure 6.16, where one can notice that there is a significant delay – about 8 sec – in the $\dot{\chi}$ response of DemoSim. The rest of the delay may be a result of the threshold function still damping out (i.e. by means of the "forgetting factor") the remaining transients in the system from previous maneuvers. From Figure 6.16 one can see this, because both $\dot{\chi}_{ds}$ and u_{ds} have *not* completely reached steady state prior to the series of turns.

Nonetheless, the threshold function did accomplish its goal, and the fault is detected. At $t = 75.5$ sec it is quite clear that the threshold function has responded to the fault. If one recalls, the criteria for detection of a fault is that $r_{th} > 0$. This occurs at 75.5 sec, thus $t_{\text{detect}} = 75.5$ sec.

[7]For the RHC algorithm this was done via the RHC API.

[8]This range was determined from stand alone testing of the FD algorithm and DemoSim. This limitation was in effect, a direct result of the fact that the inner dynamics of DemoSim were unavailable to the SEC researchers causing an inability to identify an accurate model of DemoSim, and hence, resulting in significant model mismatch.

[9]Recall there is no response to the fault in steady-state since the fault weight W_p rolls off at low frequencies. Hence, a fault will only appear in the output during a transient, and thus the need for the series of turns to excite the system.

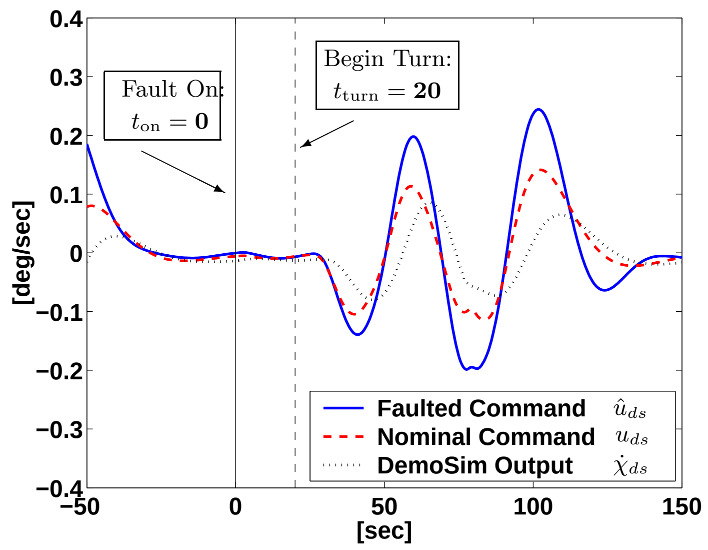


Figure 6.16: $\dot{\chi}$ Simulation signals

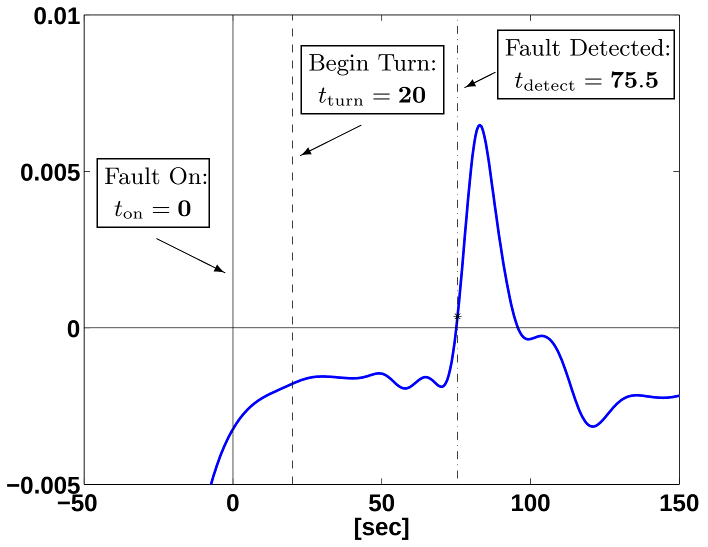


Figure 6.17: Simulation threshold r_{th}

6.6 Conclusion

The focus of this chapter was the FD algorithm design, implementation, and testing. The FD design discussed herein was done using well known $\mathcal{H}_\infty$ design methods. Additionally, a novel approach to thresholding the residual of an FD algorithm was presented, along with the theory, design, and testing of this new uncertainty-conscious, input-dependent threshold function.

The effectiveness of this combined algorithm – $\mathcal{H}_\infty$ FD filter and threshold function – was tested in a simulation with a full non-linear aircraft simulator coupled with other SEC UMN/UCB technologies: the RHC guidance controller and the RHC API. The results of the combined FD algorithm has shown promising results based upon successful detection of an aileron actuator fault. It is hoped that the results and insights of the input-dependent threshold function will provide a basis for further research and investigation into robust fault detection.

Chapter 7

RHC and FD integration within the RHC API / OCP framework

7.1 Introduction

This chapter describes the integration of the Receding Horizon Controller (RHC) and the Fault Detection filter (FD) using the RHC Application Program Interface (RHC API) implemented under the Open Control Platform (OCP) software infrastructure. In the appendix the Unified Modeling Language notations used in this chapter are briefly reviewed.

7.2 Control software requirements analysis

7.2.1 Actors identification

The use case diagram in Figure 7.1 shows the main actors and collection of scenarios identified in this work. The external rectangle depicts the whole *Aerial Vehicle Control* system, with which only the actor *Pilot* interacts. Internally, the actors *Receding Horizon Controller*, *Fault Detection Filter* and *OCP* interact with each other within the *Experiment Supervision System.* The identified actors are described in the following.

The Pilot The *Pilot* actor represents the two pilots of the Lockheed T-33 aircraft involved in the experiment. The *Pilot* interacts with the Aerial Vehicle Control system to activate and deactivate the controller during the flight. The discrete events that can be issued by the *Pilot* are listed in Table 7.1 and are sent to *RHC Controller* actor which reacts according to the *Experiment Execution* use case.

The Fault Detection Filter The Fault Detection Filter has been described in Chapter 6 and it is implemented by the *Fault Detection Filter* actor (*FD* actor in the following). Table 7.2 shows the name and the meaning of the input / output signals to the *FD* actor, while Table 7.4 lists the discrete events issued by the *FD* actor and sent to *RHC Controller* actor.

The Fault Simulator The *Fault Simulator* actor generates the fault signal that is used to corrupt the turn rate control command sent to the aircraft. This actor is used to simulate a fault in the aileron actuator. Table 7.3 shows the name and the meaning of the input / output signals of the *Fault Simulator* actor.

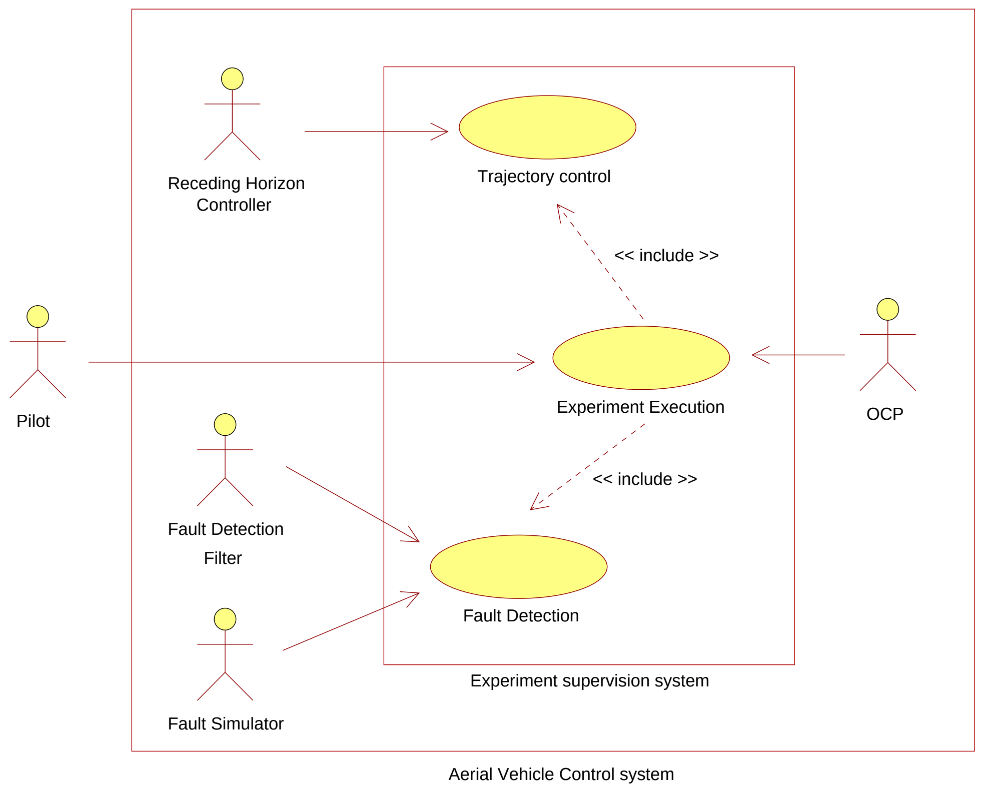


Figure 7.1: Flight experiment supervision system Use Case diagram. The picture shows the actors identified and the subsystems with which they interact.

Discrete event
`CMD_ON` (button)
`CMD_OFF` (button)
`START` (button)
`RESET` (button)
`SND_OBST` (button)
`FAULT_ON` (button)

Table 7.1: Discrete events sent by the *Pilot* to the *RHC Controller*.

Signal name	**Type**	**I/O**
`sigTurnRate_OutputSetAndHoldTurnRate`	`OCPSignal`	I
`inputUAV_State_Heading[0]`	`float`	I
`faultDetected`	`bool`	O

Table 7.2: Input / output signals of the FD actor.

The Receding Horizon Controller The Receding Horizon Controller actor (*RHC Controller* actor in the following) is the component of the Aerial Vehicle Control system which issues commands to the airplane's autopilot to follow a predefined or dynamically defined trajectory. The control strategy has been described in Chapter 5. Table 7.5 shows the name of the input / output signals to the RHC actor, and Table 7.6 reviews all the discrete events which can change the internal discrete state of the *RHC Controller* actor.

OCP This actor represents all the functionalities provided by the OCP and the RHC_API (described in Chapter 4).

7.2.2 Use cases description

This section describes the three use cases depicted in Figure 7.1.

7.2.2.1 Experiment execution (Experiment 1)

This use case covers two scenarios, each of them corresponding to a different flight experiment configuration called *UMNUCB Experiment 1* and *UMNUCB Experiment 2*. A detailed description of the two experiments can be found in the appendices.

7.2.2.2 Tracking control

This use case is *included* in the *Experiment Execution* use case. Once the controller has been engaged, it's role is to compute the control inputs that fly the aircraft along the assigned trajectory as described in Experiment Phase I to Experiment Phase III in the appendices.

Signal name	**Type**	**I/O**
`sigTurnRate_OutputSetAndHoldTurnRate`	`OCPSignal`	I
`faultedSignal`	`float`	O

Table 7.3: Input / output signals of the *Fault Simulator* actor.

Discrete event
Fault detected

Table 7.4: Discrete events sent by the *FD* to the *RHC Controller*.

Signal name	Type	I/O
inputUAV_State_CommonState[0].time_	float	I
refTrajNorth	Matrix	I
refTrajEast	Matrix	I
refTrajAlt	Matrix	I
inputUAV_State_CommonState[0].latitude	float	I
inputUAV_State_CommonState[0].longitude	float	I
inputUAV_State_CommonState[0].theWGS84_Altitude	float	I
inputUAV_State_CommonState[0].velocityUp_	float	I
inputUAV_State_Heading[0]	float	I
inputUAV_State_GndSpeed[0]	float	I
sigSpeed_OutputSetAndHoldSpeed	OCPSignal	O
sigHeading_OutputSetAndHoldHeading	OCPSignal	O
sigAltitude_OutputSetAndHoldAltitude	OCPSignal	O
sigTurnRate_OutputSetAndHoldTurnRate	OCPSignal	O

Table 7.5: Input / output signals of the RHC controller.

Discrete event	Sender
CMD_ON (button)	*Pilot*
CMD_OFF (button)	*Pilot*
START (button)	*Pilot*
RESET (button)	*Pilot*
SND_OBST (button)	*Pilot*
FAULT_ON (button)	*Pilot*
Fault detected	*FD Filter*
Engagement conditions met	*RHC Controller*
Target reached	*RHC Controller*

Table 7.6: Incoming discrete events of the *RHC Controller* actor (note that the last two listed events are internally generated).

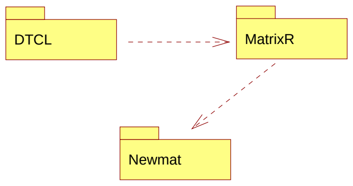


Figure 7.2: Relationships between the designed and reused packages.

7.2.2.3 Fault detection

This use case is *included* in the *Experiment Execution* use case.

7.3 Static structure design

Given the requirements expressed informally in the previous section, the static structure of the control software has been designed to satisfy the following needs

- perform numerical integration of discrete time linear systems used as prediction models and state observers;
- simplify coding of matrix computations;
- enable the control systems to react to discrete events;
- provide input / output interfaces to text files;
- implement control algorithms within the OCP RHC_API

according to the following principles

- address each requirement by integrating their solution into the static code structure to enhance the possibility of reuse;
- make use of libraries already available.

Figure 7.2 shows the relationships between the designed and reused packages. Section 7.3.1 explains the purpose of the *Newmat* library and how it has been extended to fit the needs of this work. Section 7.3.2 deals with the library designed to directly implement the control algorithms. The integration of the control algorithms within the OCP framework is discussed in section 7.3.3.

7.3.1 The *Newmat* library

In order to expedite C++ coding of the control algorithms that rely heavily on matrix algebra and manipulations, we made use of the 8.0 version of the Newmat C++ library. This package is intended for scientists and engineers who need to manipulate a variety of types of matrices using standard matrix operations [31]. Emphasis is on the kind of operations needed in statistical calculations such as least squares, linear equation solve and eigenvalues.

It supports among other matrix types *Matrix*, *RowVector*, *ColumnVector*. Only one element type (float or double) is supported. The package includes the operations $*$, $+$, $-$, Kronecker product, Schur product, concatenation, inverse, transpose, conversion between types, submatrix, determinant, Cholesky decomposition, QR triangularisation, singular value decomposition, eigenvalues

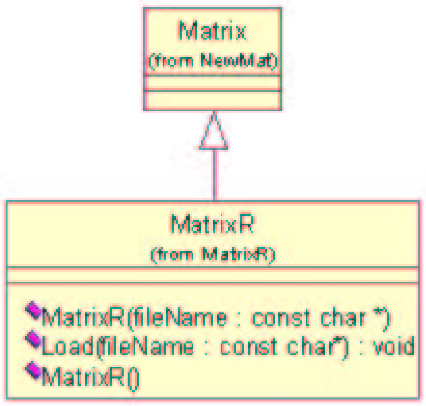

Matrix
(from NewMat)
MatrixR
(from MatrixR)
MatrixR(fileName : const char *)
Load(fileName : const char*) : void
MatrixR()

Figure 7.3: The extension of the class Matrix (from the package Newmat) to add standard text file input / output capabilities.

of a symmetric matrix, sorting, fast Fourier transform, printing and an interface with Numerical Recipes in C.

It is intended for matrices with dimensions ranging from 10-by-10 to the maximum size your machine will accommodate in a single array. The number of elements in an array cannot exceed the maximum size of an int. The package will work for very small matrices but becomes rather inefficient. In this version of the package some of the factorisation functions are not optimised for paged memory and so become inefficient when used with very large matrices. A lazy evaluation approach to evaluating matrix expressions is used to improve efficiency and reduce the use of temporary storage.

This package has been extended to provide the possibility to read and write matrix elements from standard text files. Figure 7.3 shows the derived class with the added methods.

7.3.2 The *Discrete Time Control Library* (DTCL)

Figure 7.4 shows the class diagram of the DTCL library. There is a main class, the *GeneralSystem* class, which has been designed to provide a framework for derivation of the other classes. This class exposes the *oneStep()* method, which is responsible for computing one step of the algorithm implemented by the class.

The *LTISystem* class, derived from *GeneralSystem*, implements a general discrete time linear time-invariant system. When the overloaded method *oneStep(input : const Matrix&)* is called, the standard difference equations

$$\begin{aligned} x_{k+1} &= Ax_k + Bu_k, \\ y_k &= Cx_k + Du_k, \\ x(0) &= x_0, \end{aligned} \tag{7.1}$$

are calculated, where the value of the variable u_k is equal to *input* value. The class provides through the class *MatrixR* the input / output capabilities using standard text files.

The *MPCController_api* implements the actual controller, whose algorithm has been described in Chapter 5. This class is associated with a state observer, realized as an *LTISystem* object, and with four helper classes used to collect all the data needed for computations.

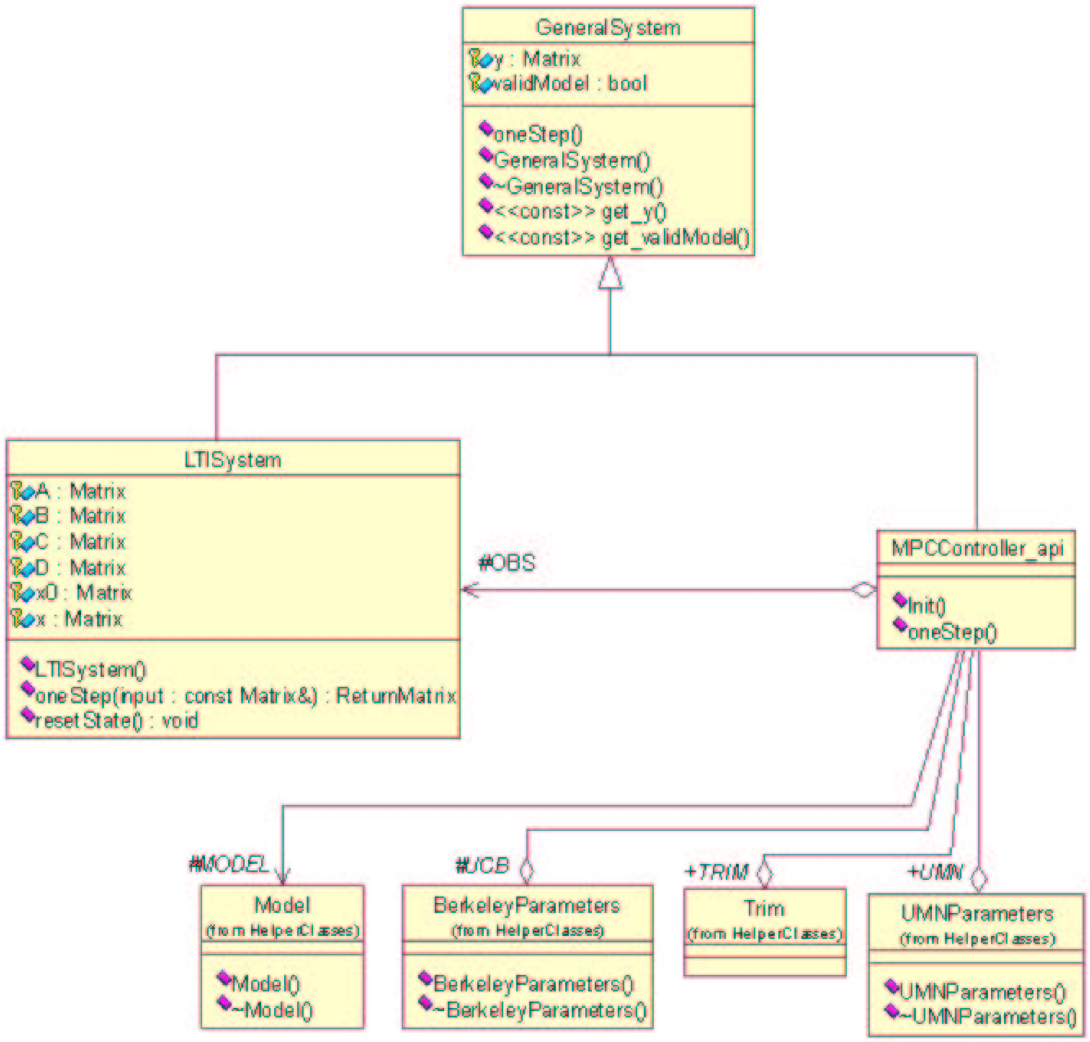


Figure 7.4: Class diagram of the DTCL library.

7.3.3 Integration within the OCP framework

The RHC_API provides the *RHC_Component* class which is directly interfaced with the OCP. The methods *Pre()*, *Opt()*, and *Post()* represent the three main components of the flight controller code. These methods have different scheduling mechanisms in the real-time system and are invoked by the Anytime scheduler as described in Chapter 4.

The *RHC_Component* implements both the *FD* and the *RHCController* actors by associations to a set of *LTISystem* classes and to the *MPCController_api* class (through a proper interface class), respectively.

7.4 Dynamic behavior specifications

The *RHC_Component* has an internal discrete state machine used to interact with the *Pilot* as well as with the *FD* and the *RHCController* actors implementation as specified by the use case diagram described in Section 7.2.2. The main state-chart is depicted in Figure 7.6, while the corresponding sub-state-charts are depicted in Figure 7.6, Figure 7.8, and Figure 7.9, respectively.

7.5 Implementation issues

7.5.1 Memory allocation in real-time implementation

The programming paradigms used for the implementation of the described algorithms (such as object oriented programming in C++ under the QNX real-time operating system) raised an interesting issue which, if present, is usually not relevant for non-real-time applications. In this section, the fundamental problem and the adopted solution is discussed.

The memory allocated by the operating system to a process ready for execution can be divided into two logical parts: the *stack* and the *heap*[1]. The stack memory is used to store all the data whose storage size does not change during the process execution. Typical examples of this type of data are the code itself and the *statically* allocated data (i.e. the variables local to a process and its non recurrent procedures belonging to types predefined by the compiler).

The heap instead, is used to store data whose size *changes* during the execution. In object oriented programming, class objects are often created and destroyed at run time. When an object is created, the necessary space is taken from the heap; when the object is destroyed, the memory is released to be used again by some other object.

A consequence of this process is that memory fragmentation problems can arise when memory allocation and release occurs frequently. If the memory becomes too fragmented, it can lead to the lack of enough contiguous memory space to be allocated for an object, even if the sum of all the free space available is much more than needed. When this occurs the object cannot be created, the program encounters an *out of memory* error message (or similar one) and its execution is stopped by the operating systems.

In the real-time application described in this report, a lot of objects of considerable size are dynamically created and destroyed at each execution cycle (two times a second). After a certain number of steps (it could be even couple of minutes of execution), the process was terminated due to the described dynamic heap memory allocation error.

[1]The technical reasons for this memory classification are not addressed in this report, the reader is referred to [32] for more details.

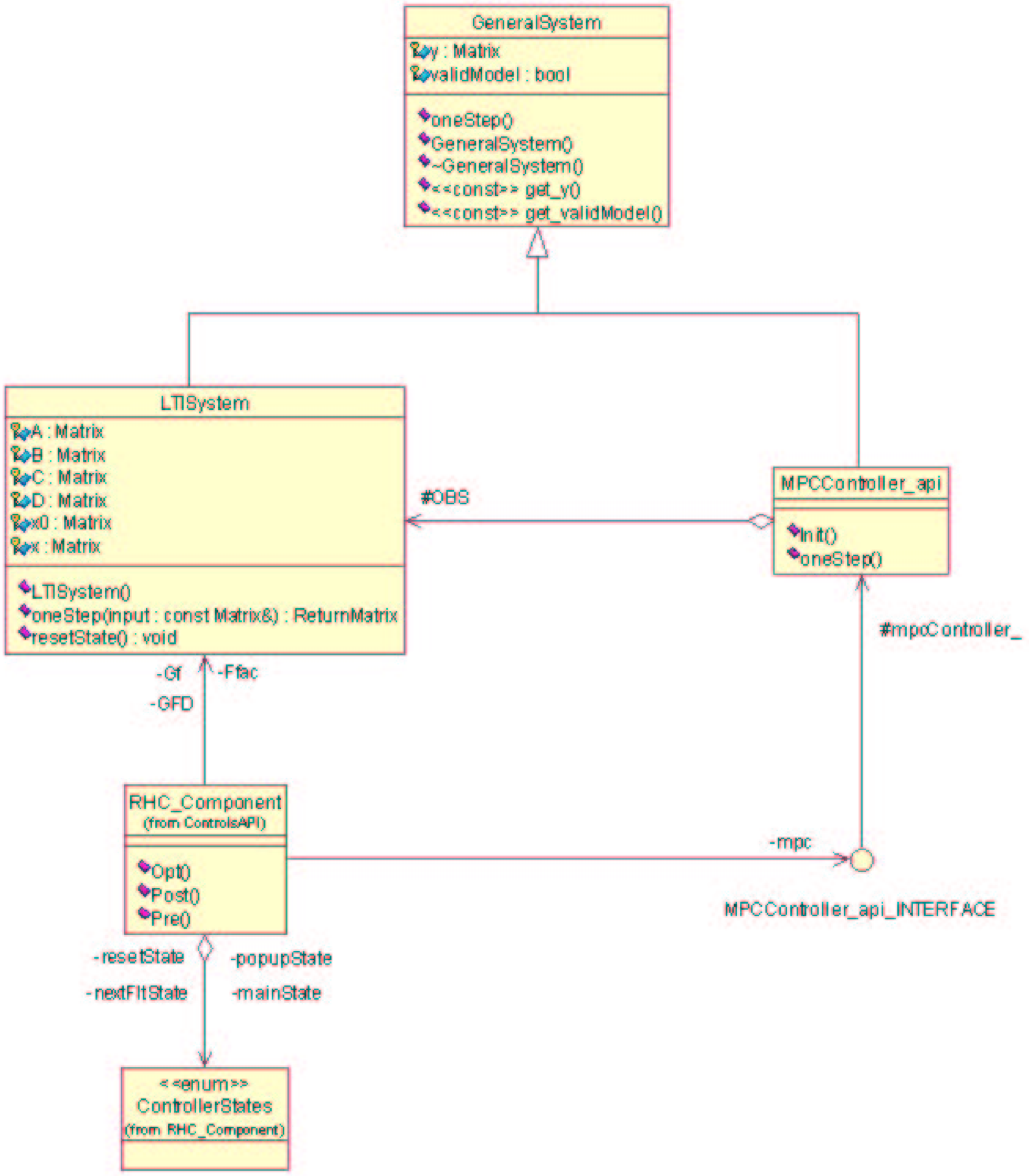


Figure 7.5: Static structure of the control system implementation within the OCP.

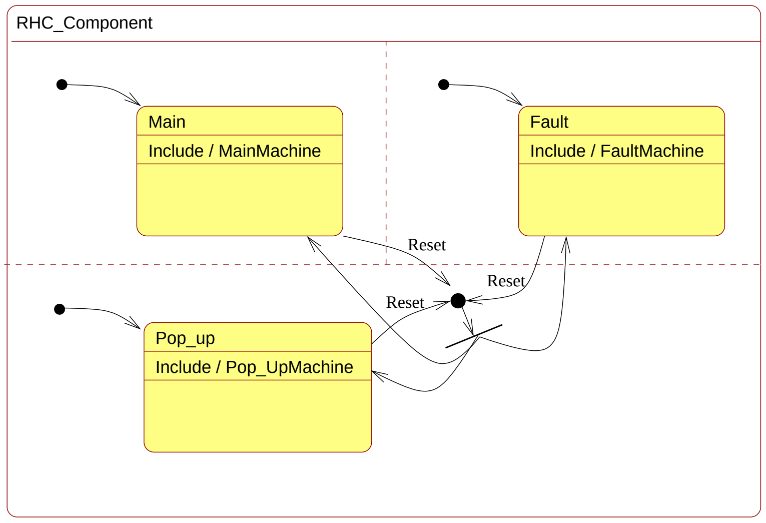


Figure 7.6: State-chart for the RHC Component.

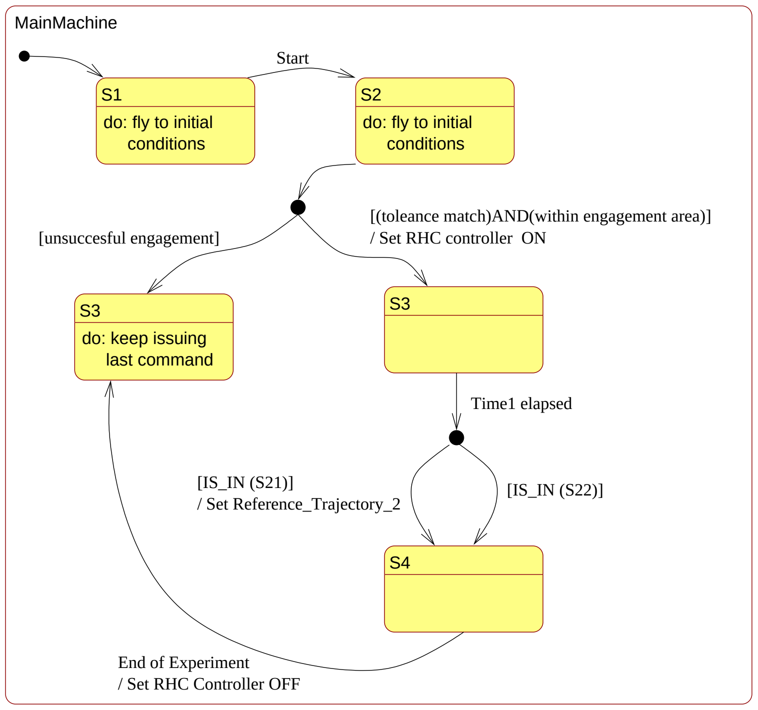

MainMachine
Start
S1
do: fly to initial conditions
S2
do: fly to initial conditions
[unsuccesful engagement]
[(toleance match)AND(within engagement area)]
/ Set RHC controller ON
S3
do: keep issuing last command
S3
Time1 elapsed
[IS_IN (S21)]
/ Set Reference_Trajectory_2
[IS_IN (S22)]
S4
End of Experiment
/ Set RHC Controller OFF

Figure 7.7: Main state-chart for the experiment execution.

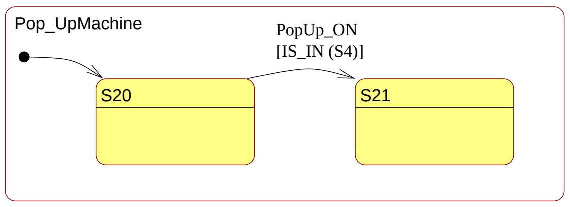


Figure 7.8: State-chart for the Popup threat handling.

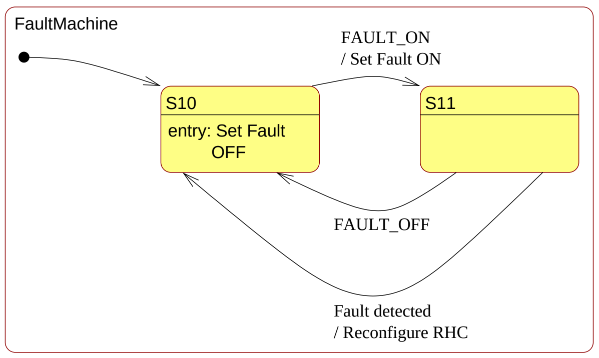


Figure 7.9: State-chart for the FD subsystem.

The solution adopted in order to get around this issues has been to code all variables as local and allocate objects dynamically in an initialization phase only, which occurs during startup at the beginning of the flight experiment. No dynamic allocation function call has been used during normal control execution. Although this solution satisfied real-time flight code requirements and is very simple in principle, it has the disadvantage of making the resulting code less intuitive and readable.

7.5.2 Frame overrun detection and handling

Due to the particular structure of flight code implementation within the OCP real-time environment, extra care had to be taken to ensure that execution does not violate the tripartite process structure of the Pre-Opt-Post framework for RHC implementation, described in Chapter 4. This meant that any frame-overruns that occur from abnormal code execution, or algorithmic anomalies had to be detected and appropriate contingency measures had to be put in place to maintain control of the aircraft and avoid a situation from which the code execution and the controller cannot recover.

This critical element of the flight code was accomplished using a simple state machine that caught any frame overruns that might have occurred in the three-process structure. Depending on which process was responsible for the frame-overrun, appropriate actions could be taken in the flight code to prevent an unrecoverable execution error. For instance, if the optimization solver could not converge and produce any control outputs to implement in the available time-frame, then a guaranteed safe, baseline solution was sent to the autopilot for implementation. This contingency measure was able to overcome accidental frame-overruns and prevent software crashes. Although this ad-hoc solution might lead to decreased performance, it maintains control of the aircraft at all times.

Chapter 8

Description and presentation of results

The contents of this chapter consists of two main parts. Section 8.1 includes a general description of the final flight test infrastructure and the most important elements of the final experiment. A basic summary of key features of the experiment controller interface and a brief timeline of events in the UMN/UCB experiment scenario are also provided.

In light of this description, results are presented in Section 8.2 based on Matlab/Simulink simulations using the DemoSim open vehicle executable model and the RHC control techniques described in Chapter 5. High fidelity hardware-in-the-loop simulations and the recorded final flight test data are provided as well. Finally, conclusions are drawn in Section 8.3.

8.1 Final flight test infrastructure and experiment description

The flight tests took place at Edwards Air Force Base in the Mojave desert in June 2004. A brief description of the flight demonstration scenario is provided next in order to aid in the interpretation of results provided in this chapter. For more detail, refer to Appendices A-B. Figure 8.1 shows a bird's eye view of the test range and the experimental scenario elements.

The timeline of events starts with engaging the RHC controller in a pre-specified area near the ingress point once the starting conditions are met. The controller tracks a time-stamped position reference trajectory while respecting constraints on the vehicle dynamics. At a certain point along the reference trajectory, a pop-up threat can be invoked by a ground operator, which results in a switch to an alternative reference trajectory that avoids the threat. After the target is reached with a specified heading, a simulated fault is inserted into the system, which is detected shortly thereafter at a trajectory segment designed specifically for this purpose. After detection, the fault is removed and the aircraft returns to the egress point.

In a second, more ambitious scenario, the fault is not removed from the system after detection, and the RHC controller is reconfigured to adapt to the faulty vehicle dynamics. Constraints are also adjusted to restrict the aircraft's maneuvers. Section 8.2.2 presents high-fidelity simulation results for this scenario.

Flight envelope and maneuvering limits were not included in the flight code but they were implemented as soft output constraints for testing in high-fidelity simulations. Results are shown in Section 8.2.3.

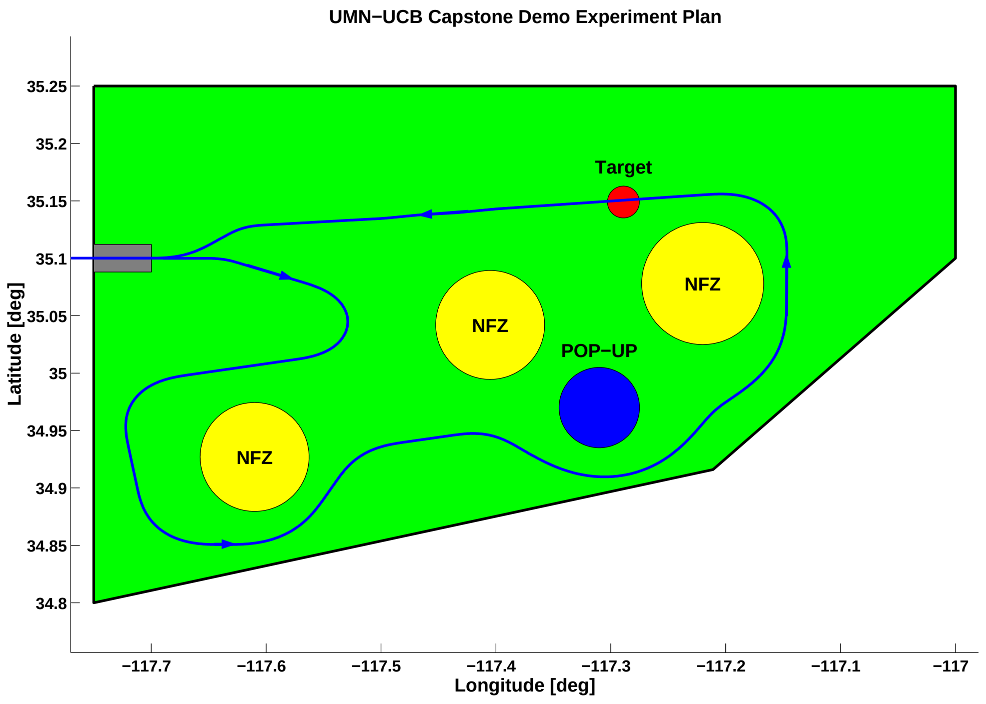


Figure 8.1: Illustration of the reference trajectory in the flight test area with the target, pop-up threat and no-fly-zone locations of the experiment scenario.

8.2 Data analysis

8.2.1 Simulation results in the Matlab/Simulink environment using DemoSim

The receding horizon guidance controller was tested in simulation with different wind conditions and showed excellent robustness. Figure 8.2 shows the resulting position trajectories under different wind conditions using the RHC controller described in Chapter 5. Tracking performance is illustrated by coordinate-wise tracking errors depicted in Figure 8.4.

Figure 8.3 shows how the true airspeed was adjusted according to wind conditions, to achieve the necessary ground speed required for accurate tracking of the position reference. Constraint enforcement under various wind conditions is also demonstrated in this figure by the saturating turn rate command. The main limitation of good performance at higher wind velocities was posed by flight envelope constraints.

Figure 8.5 shows the threshold residual of the fault detection filter after the fault has been inserted. The detection time is almost exactly the same in varying wind conditions.

8.2.2 Reconfiguration based on fault detection

As mentioned in earlier chapters, a second more ambitious experiment was constructed to test RHC controller reconfiguration based on fault detection and to provide a longer, more versatile reference trajectory for evaluating the guidance controller. Experiment #2 is conducted exactly the same way as Experiment #1 until the point of fault insertion, where the fault is not removed from the system after detection. Instead, the prediction model and constraints of the RHC controller are updated as

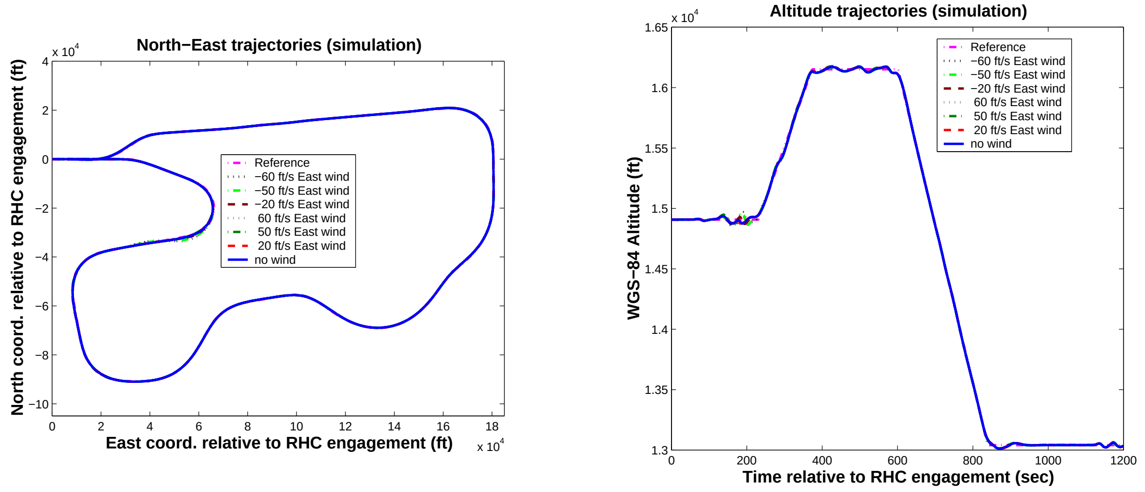


Figure 8.2: Simulated north, east and altitude tracking performance under different wind conditions.

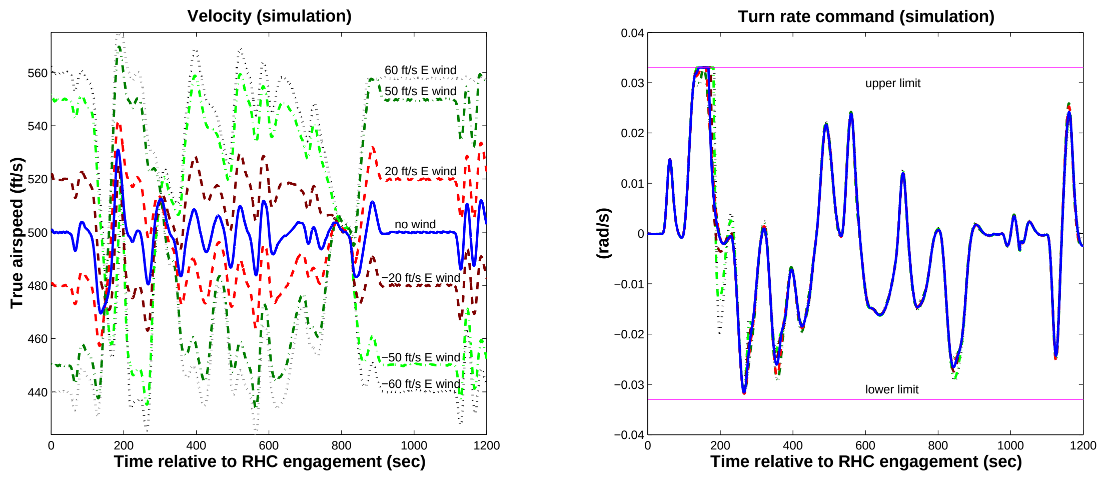


Figure 8.3: True airspeed and turn rate command simulation results under different wind conditions.

described in Section 5.6. More details of Experiment #2 can be found Appendix B. Unfortunately, this experiment could not be flight tested due to difficulties with asset scheduling at the base and shortage of time. This section presents simulation results conducted in the Matlab/Simulink environment using DemoSim.

Figure 8.6 shows position trajectories of the Experiment #2 simulation indicating good tracking performance even after reconfiguration of the RHC controller.

Figure 8.7 illustrates one effect of reconfiguration by plotting the turn rate command output of the RHC controller, which was corrupted by the output of the fault simulator, and the faulty command sent to DemoSim. Note that bounds on turn rate command became more restrictive after fault detection, as part of the controller reconfiguration. Note also that constraints were formulated for the RHC controller output and not the actual autopilot command, which includes the corruption by the fault simulator. This means that although the RHC controller restricted its

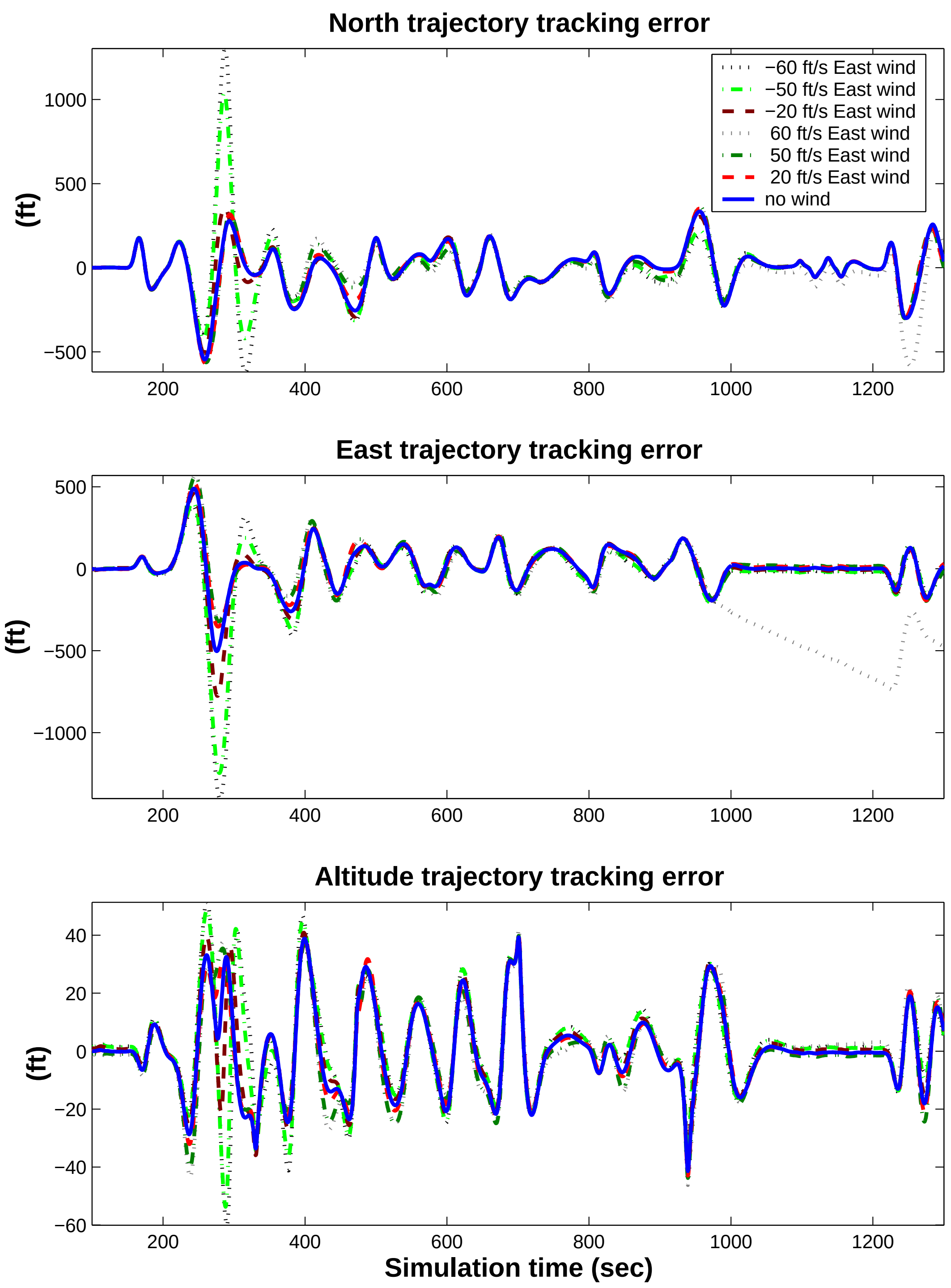


Figure 8.4: North, east and altitude tracking errors under different wind conditions.

turn rate commands to respect the specified tighter limits, the actual transient commands, due to the additive effect of the fault model, could reach values larger than the constraint limits. This is simply a consequence of the chosen fault modeling, and does not have any practical significance.

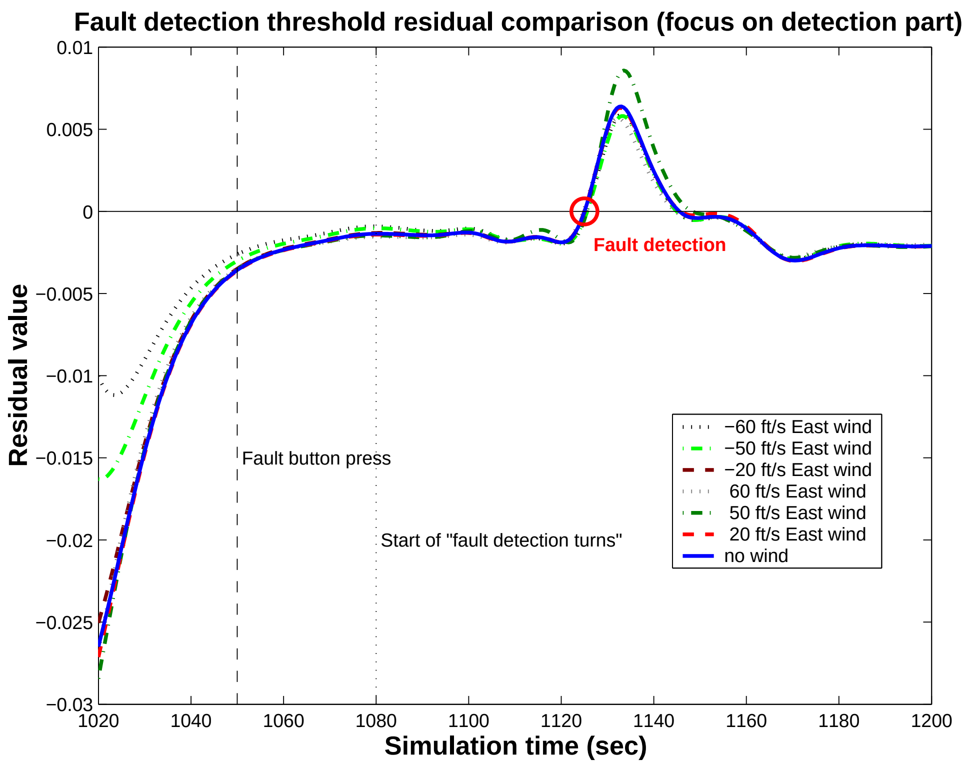


Figure 8.5: Threshold residual and fault detection under different wind conditions.

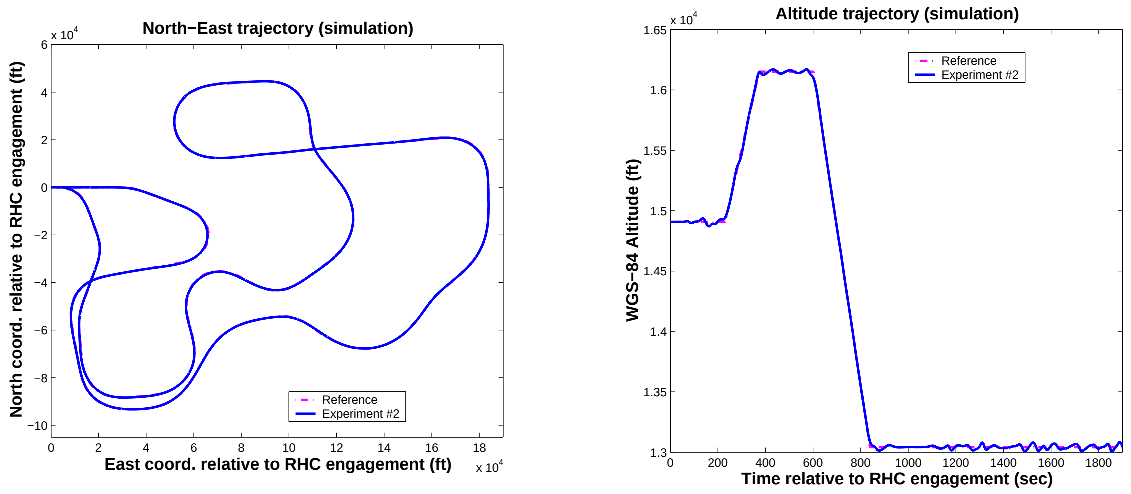


Figure 8.6: Simulated north, east and altitude tracking performance of Experiment #2 involving RHC reconfiguration.

If the fault model represents an actual physical system component, then soft output constraints could be specified on the corrupted control signal as well.

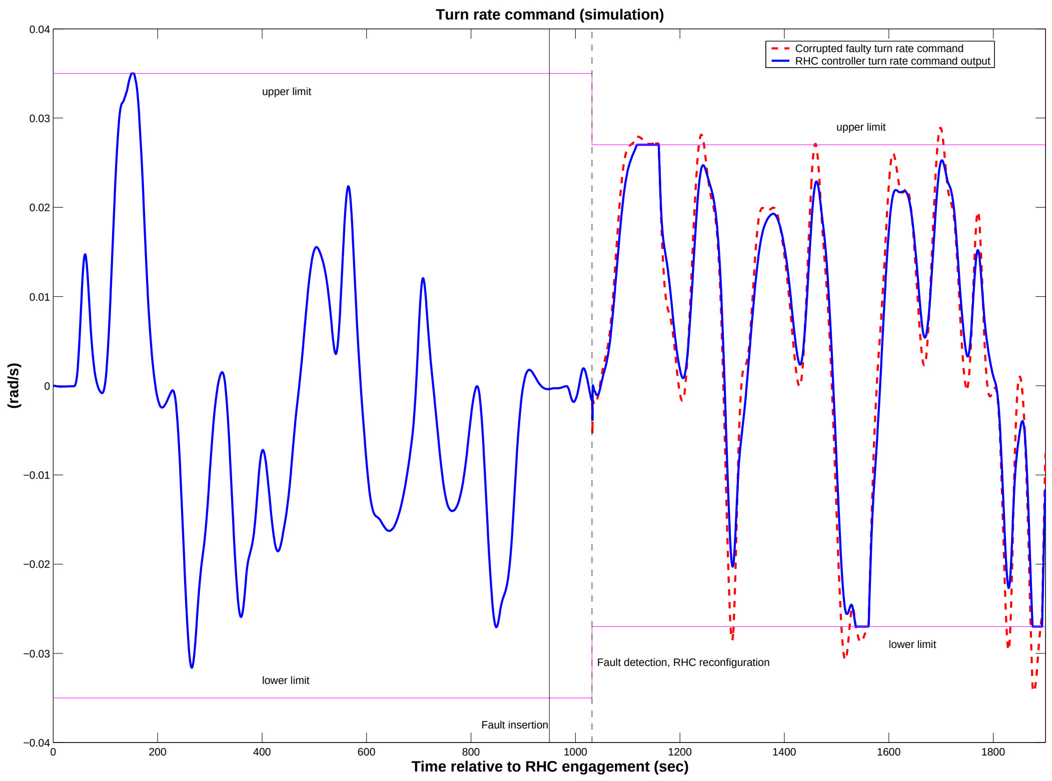


Figure 8.7: Commanded and corrupted (faulty) turn rate commands in Experiment #2. RHC reconfiguration includes updating turn rate command constraints after fault detection.

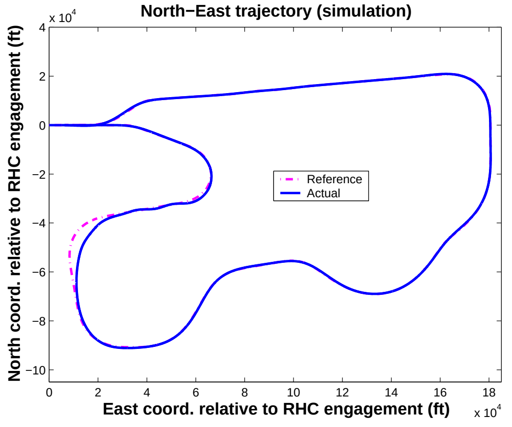
8.2.3 Flight envelope limits as output constraints

This section presents simulation results using maneuvering and flight envelope limits characterized in Section 3.4 as soft output constraints in the RHC formulation according to the description provided in Section 5.7.

Figure 8.8 shows north, east and altitude position trajectories, which indicate poorer tracking accuracy compared to the performance obtained without output constraint enforcement. The reason is that the inner polyhedral approximation of such constraints results in more restrictive limits on bank angle and consequently on turn rate command than before. The reference trajectory includes tighter turns than what these new linearized approximate limits allow, so the guidance controller is unable to perform very precise tracking given these restrictions on maneuverability. On the other hand, the controller tries to keep up with the time-stamped position reference by velocity control and accelerating to “catch up” with the reference point that “got ahead” of the aircraft in the tighter turns.

The component-wise trajectory tracking errors depicted in Figure 8.9 reflect this deviation from the reference trajectory due to restrictions on aircraft maneuverability. These error plots also indicate how tracking performance is regained by speed control. Ground speed, turn rate and roll angle are reported in Figure 8.10 showing a much more active velocity control profile than without maneuvering restrictions.

Maneuvering and flight envelope constraint enforcement is illustrated in Figure 8.11 by plotting the aircraft trajectory in the output constraint space defined by $V, \dot{\chi}, \dot{\gamma}$. These plots show the non-



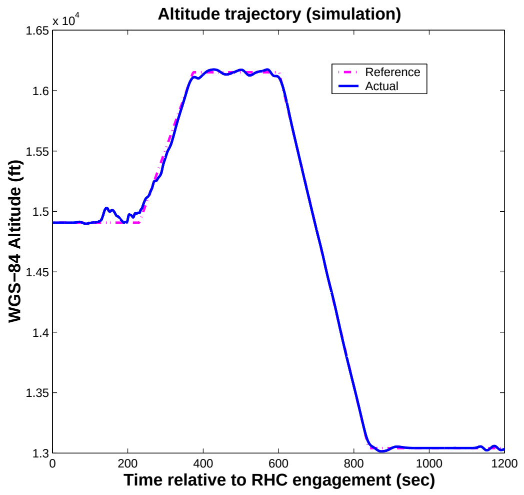


Figure 8.8: North, east and altitude tracking performance with linearized maneuvering limits as soft constraints.

linear feasible region from different viewing angles together with its inner polyhedral approximation as introduced in Section 3.4. The aircraft trajectory was plotted in terms of the constraint-space coordinate axes and overlayed on these images to show that the soft output constraints are respected. Slight violations can be noticed on the facets of the constraint-polyhedron corresponding to bounds on bank angle. These violations are possible due to the soft constraint formulation.

8.2.4 Hardware-in-the-loop simulations

The Boeing Company was responsible for coordinating and executing the flight tests, which took place in the second half of June, 2004. Boeing was also responsible for integrating the individual flight experiments and controller code developed for DemoSim and OCP by the different teams participating in the final demo. Preparations for the flight tests culminated in a final system check and testing phase of code development, which included hardware-in-the-loop testing of flight experiments in a realistic, high-fidelity simulation environment at the Boeing facilities, in Saint Louis.

The hardware-in-the-loop simulation environment relied on high-fidelity nonlinear closed-loop aircraft dynamics represented by the DemoSim model. It also made use of the actual avionics pallet flown in the flight tests with simulated sensors and electronic command data processing. Due to proprietary reasons and the International Trade of Arms Regulations (ITAR), only Boeing employees could perform these simulations. After the flight test, recorded data from hardware-in-the-loop simulations were cleared for public distribution.

Three hardware-in-the-loop simulations were run as part of the final pre-flight system check of our experiment. In the following, we will refer to hardware-in-the-loop simulations as "HIL" for brevity. The first HIL simulation was conducted by Boeing employees without inserting the fault at the specified segment of the scenario, which provided data for evaluating the false alarm characteristic of the detection filter (i.e. would it detect a fault if it wasn't inserted). The second HIL simulation was interrupted due to some problems with the Boeing test facilities that was beyond our control and occurred soon after the fault was initiated in our simulation. Data was recorded only up to a few seconds after fault detection, however it could be evaluated only up to

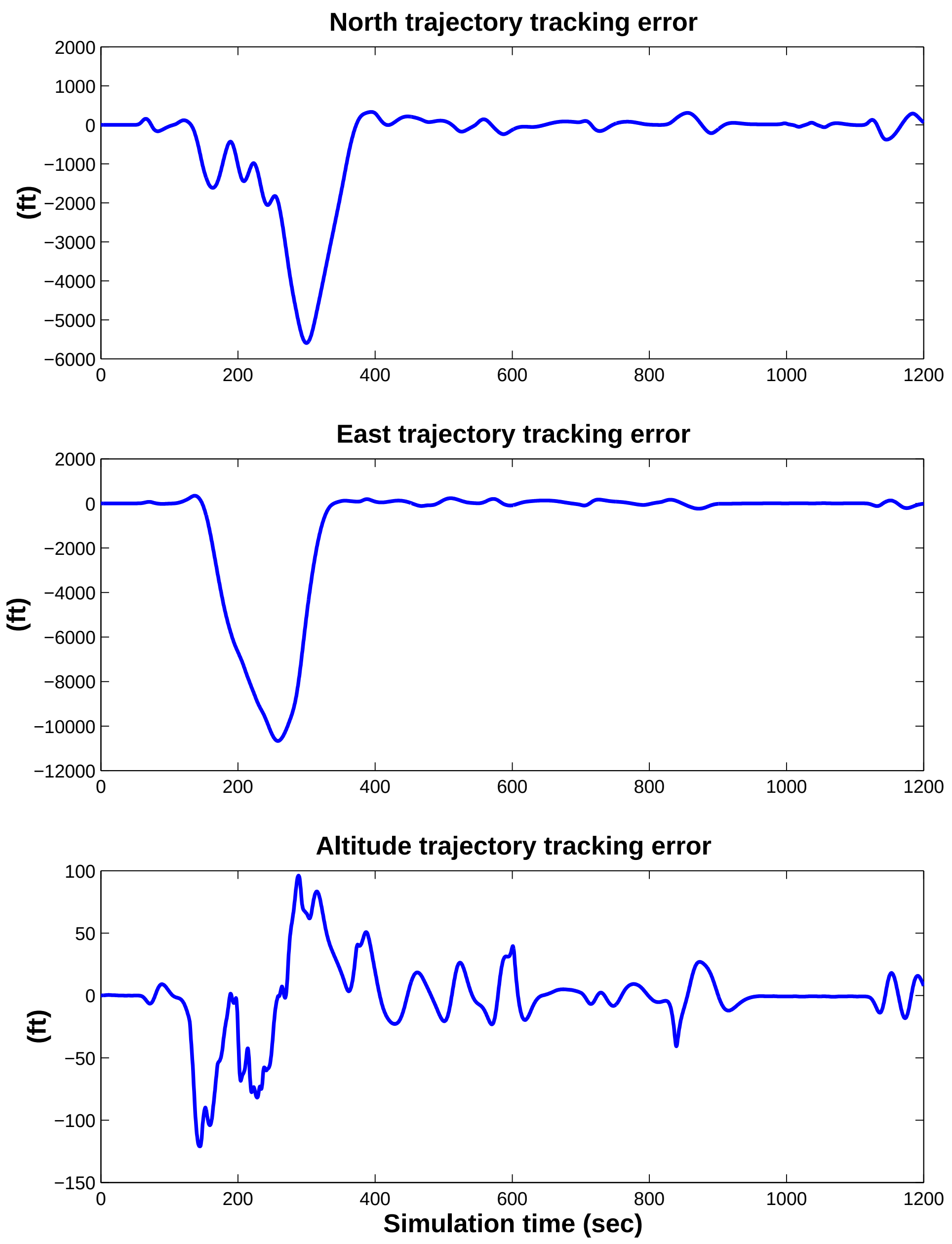


Figure 8.9: Coordinate-wise position tracking errors with limits on aircraft maneuverability.

the time instant of detection. The third and final HIL test run contains a full simulation of our experiment scenario including the initiation of the fault. (Note that only Experiment #1 was tested in HIL simulations.)

For comparison purposes, the figures used for illustration in this subsection depict HIL simulation data as well as simulation results conducted in our lab at the University of Minnesota using OCP and DemoSim. This latter simulation will be referred to as "UMN" results.

Figure 8.12(a) shows the evolution of the threshold residual value near the point where the fault was inserted using the "FAULT-ON" button of the experiment controller interface and the

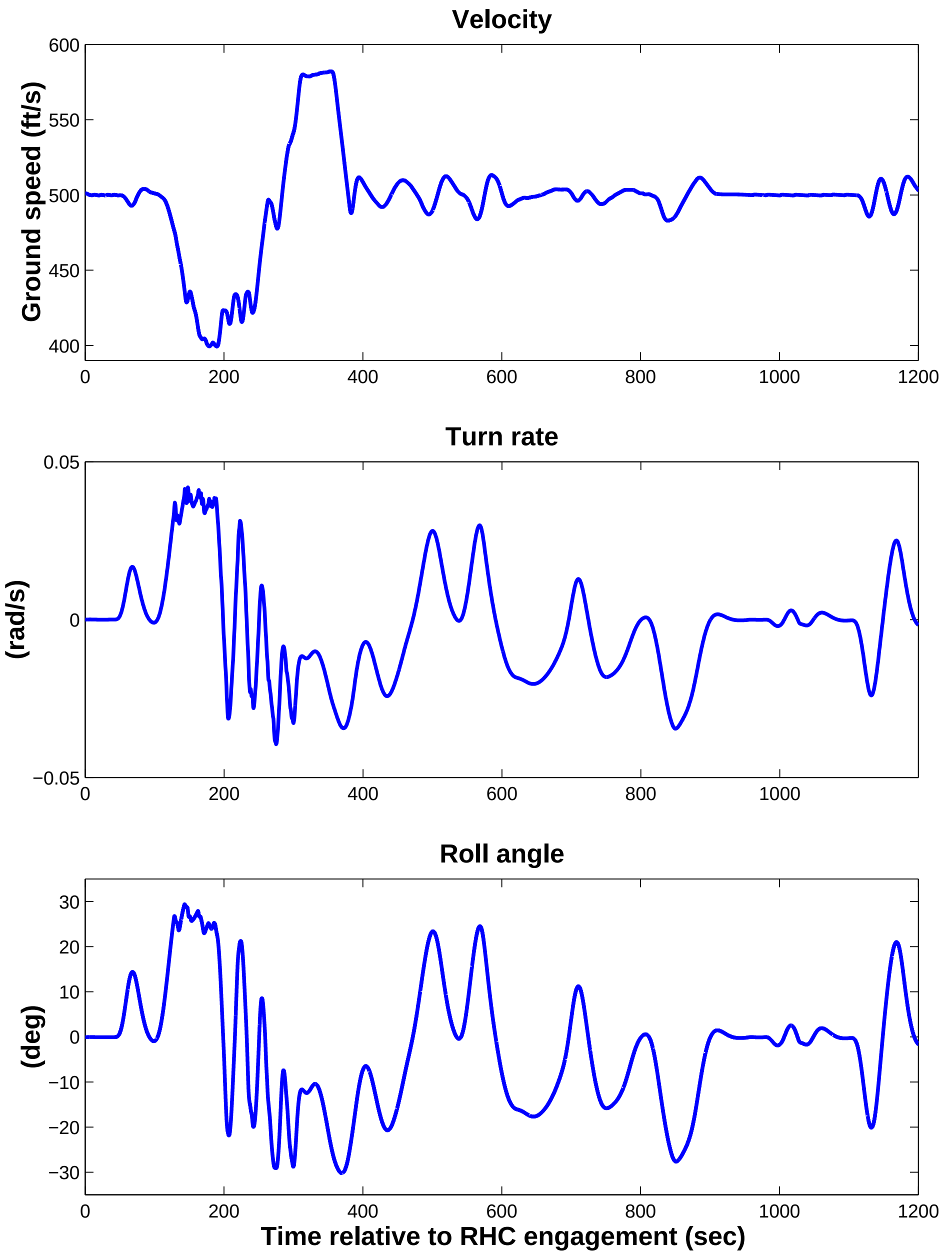
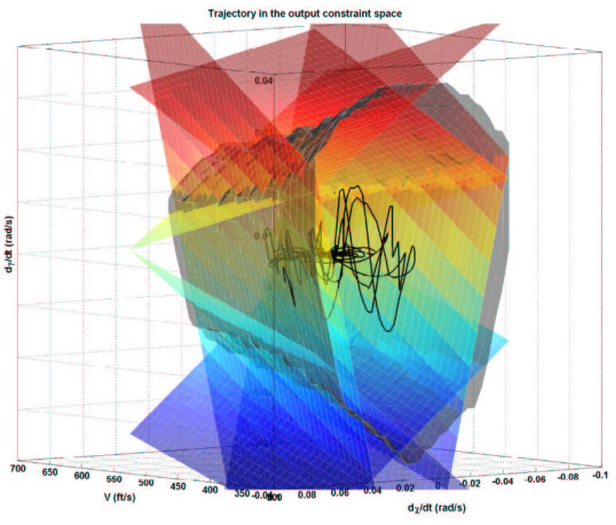


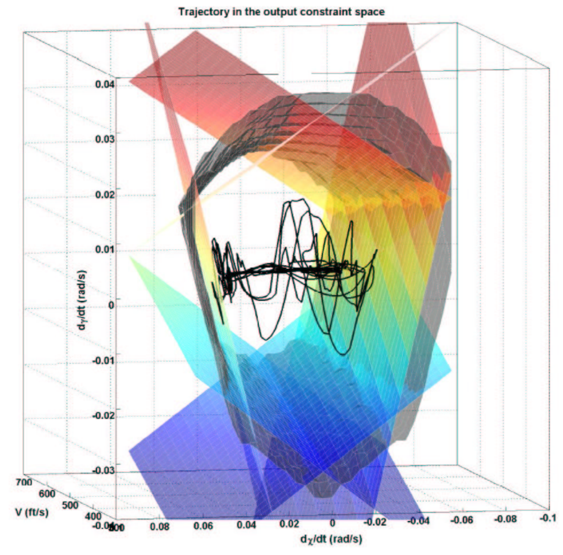
Figure 8.10: Ground speed, turn rate and roll angle time histories with limits on aircraft maneuverability.

aircraft reached the "fault detection" segment of the reference trajectory. The threshold value rises above zero near 1025 seconds after controller engagement signalling detection of the fault. The result looks remarkably similar to our UMN simulation. Figure 8.13(a) depicts the evolution of the threshold residual during the entire simulation. Figures 8.12(b)-8.13(b) show the same plots for test run #1, where the fault was not inserted. Results in this simulation indicate that the filter would not trigger a false alarm if the fault is not inserted in the HIL simulation.

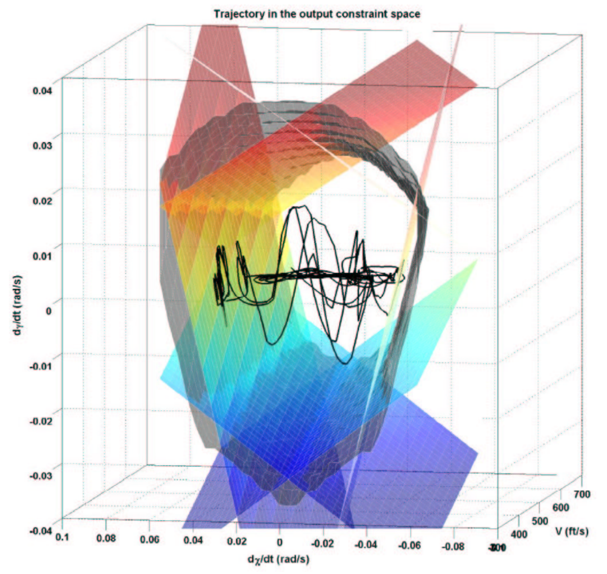
Figure 8.14 shows a comparison of tracking performance in the north-east coordinate frame and



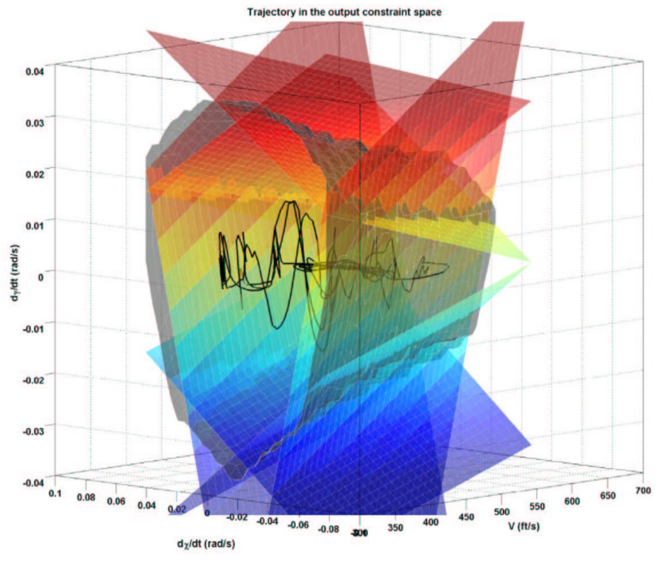
(a)



(b)



(c)

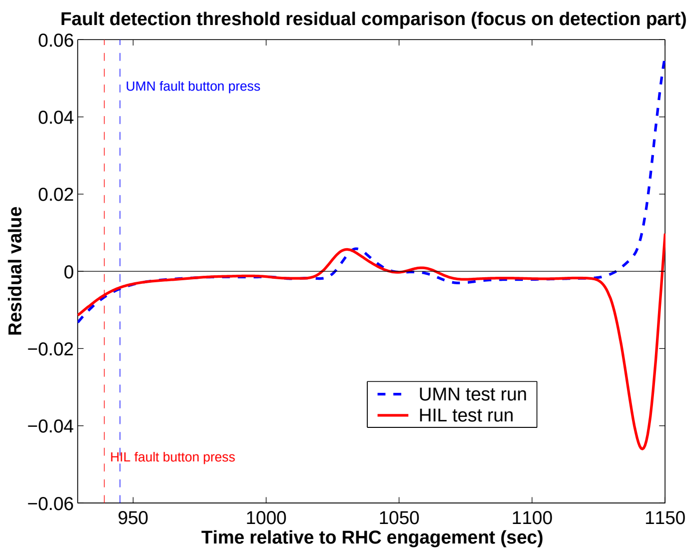


(d)

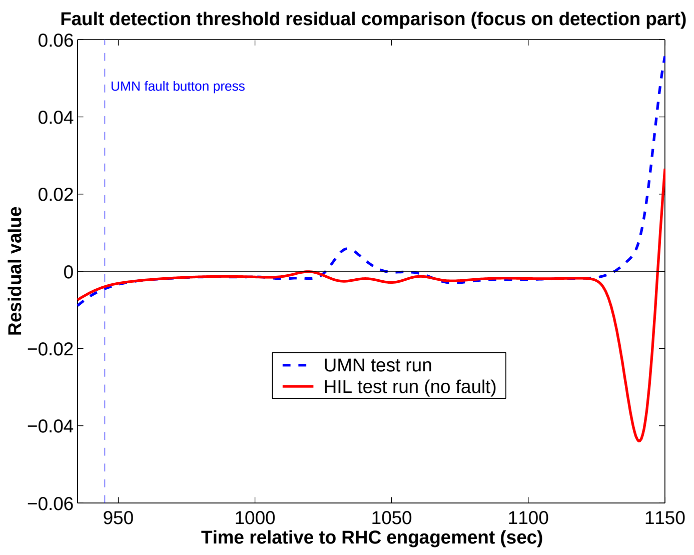
Figure 8.11: Output constraints.

in terms of altitude. Tracking errors are depicted in Figures 8.15-8.16 exhibiting similar trends between the in-house UMN and the high-fidelity HIL simulation results.

Figure 8.17 illustrates the velocity, heading and roll angle evolution during the simulation. The “jagged”, sawtooth-like nature of the velocity signal is due to the pilot model embedded into DemoSim, which represents pilot reaction to speed commands by a quantizer, which is probably acting on speed tracking error. This pilot quantization effect could be switched off in the Matlab/Simulink/DemoSim environment, so this phenomenon was not present in the results presented

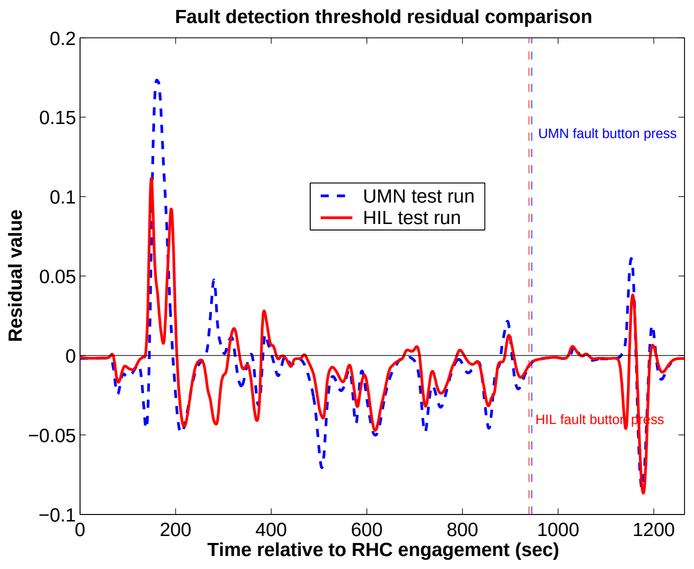


(a) With fault insertion.

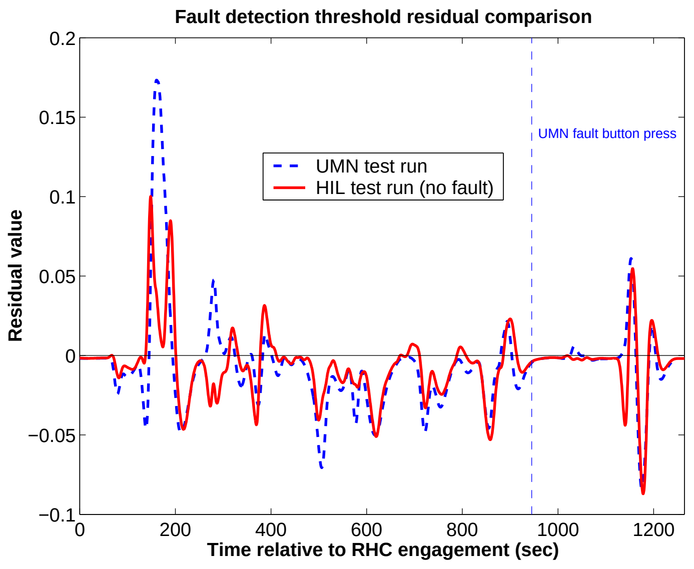


(b) No fault inserted.

Figure 8.12: Hardware-in-the-loop threshold residual comparison plots (focus on fault detection).



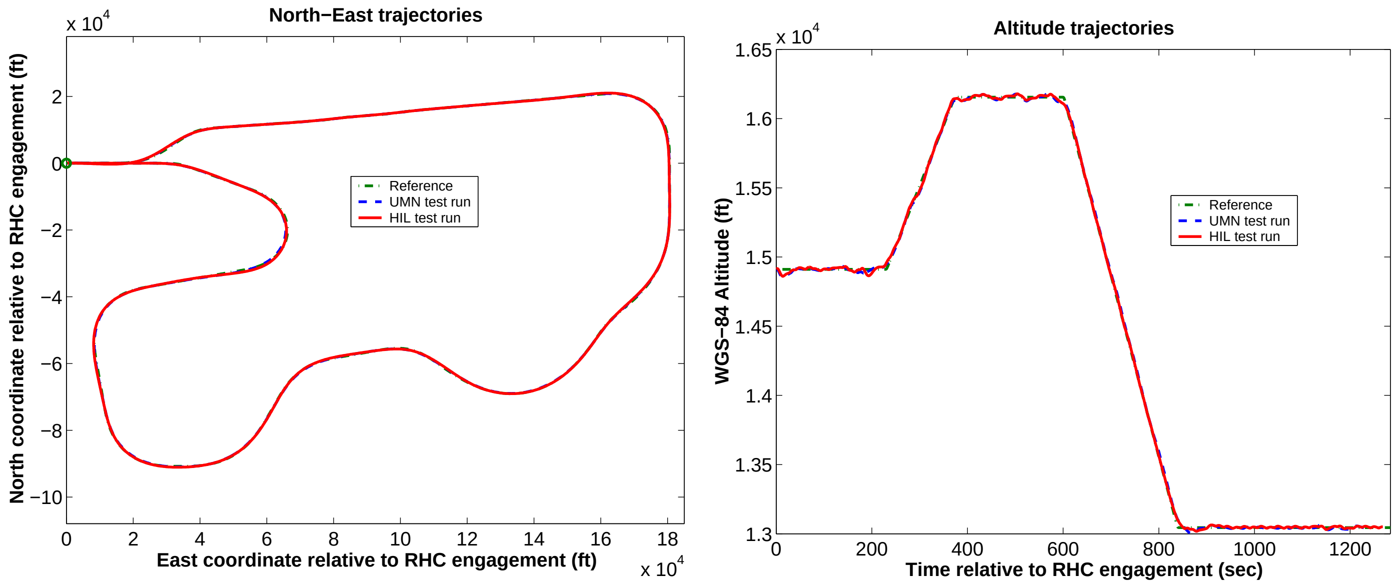
(a) With fault insertion.



(b) No fault inserted.

Figure 8.13: Hardware-in-the-loop threshold residual comparison plots (entire simulation).

in previous sections. The HIL test environment was constructed in such a way that this quantization effect could not be switched off. In order to obtain a fair comparison, pilot quantization was turned on during all UMN experiments presented in this section. The effect of this peculiar modeling phenomenon can be observed on the command signals as well, which are depicted in Figure 8.18. Enforcement of constraints can be observed by the saturating turn rate command.



(a) (b)

Figure 8.14: Hardware-in-the-loop north, east and altitude tracking performance comparison.

8.2.5 Experimental flight test results

Several universities and aerospace companies were involved in the two-week long flight demonstrations of the DARPA SEC project, each having their own experimental flight scenario to be tested. Due to difficulties with asset scheduling at the base, our team eventually had only two flight tests that could be evaluated.

The top two plots in Figure 8.19 show simulated tracking performance in the north-east coordinate frame and in terms of altitude under different wind conditions. The bottom two plots in Figure 8.19 illustrate the flight test results. The main reason for deviations from the reference trajectory and degraded tracking performance during flight test was that automatic speed control was not available on the test platform. Airspeed was controlled using manual adjustments to the throttle by the pilot, who was cued either verbally by a Weapons System Officer from the back seat or by a three-state LED indicator whether to increase, decrease or maintain the velocity of the aircraft based on commanded and actual speed measurements. The dead-zone of the “maintain speed” status indicator was approximately 30 ft/s wide. Another factor influencing the outcome of experiments was that flight tests were conducted in the presence of strong winds (25-30 knots).

Figure 8.20 suggests an average delay of 50-100 seconds in the velocity command channel, which was not modeled in the RHC controller. The controller was tested only up to 10-20 samples (5-10 seconds) of unmodeled additional delay compared to the prediction model and showed acceptable degradation of performance.

8.3 Lessons learned

As soon as details of the flight test environment was determined by Boeing, one of the main practical conclusions of the flight tests could already be anticipated. As foreseen already in early stages of

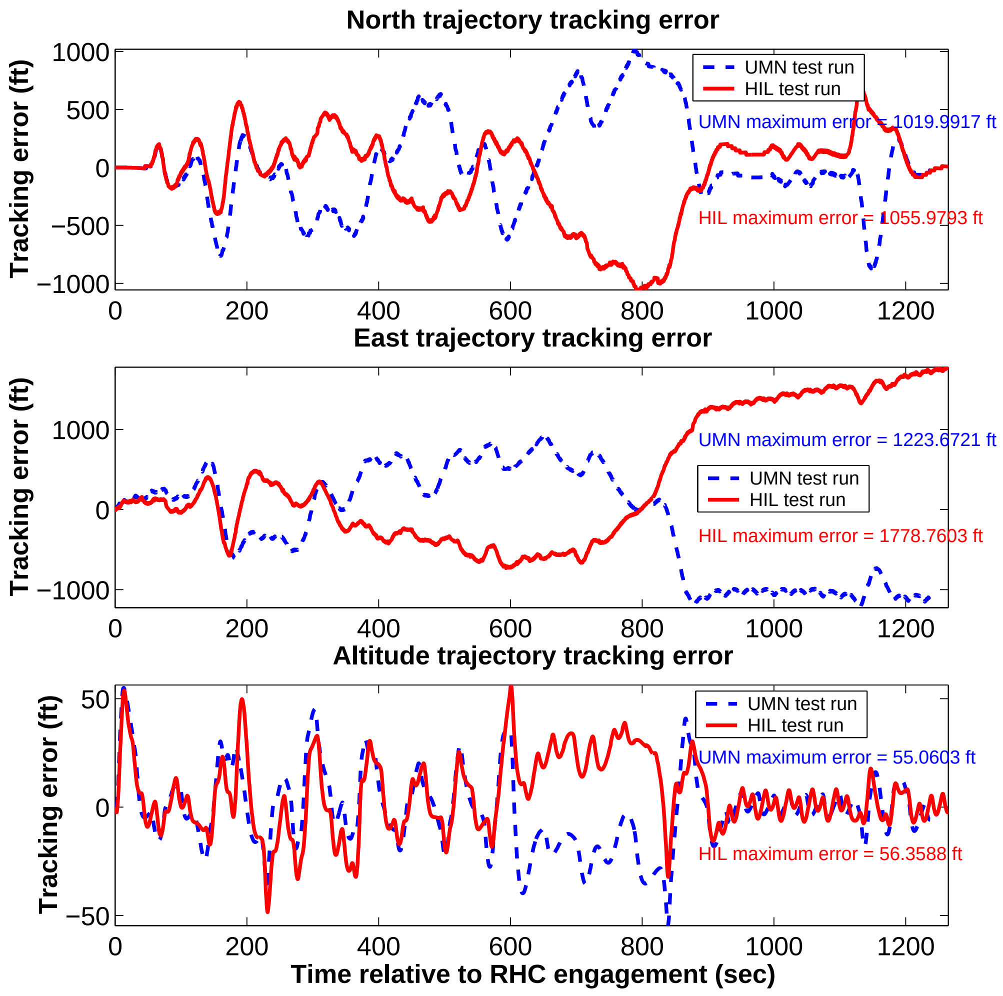


Figure 8.15: Comparison of coordinate-wise hardware-in-the-loop tracking errors.

preparations, the flight tests showed that automatic speed control is essential for tracking time-stamped position reference trajectories. Despite the absence of automatic throttle control, which is an essential part of every autonomous unmanned air vehicle, the RHC guidance controller showed acceptable tracking results in a windy environment.

At the cost of a few unsuccessful test runs, the first days of flight tests revealed that the avionics package provided altitude measurements in terms of barometric altitude as opposed to the WGS-84 system, which was assumed for control design and was also the units of commands sent to the autopilot. This meant that the engagement tolerances defined to trigger the execution of the RHC guidance controller had to be relaxed significantly to account for the possibly very large mismatch between barometric and WGS-84 GPS altitude. This in turn resulted in significant transients during the engagement procedure at the beginning of each test run. Strong winds resulted in saturation of flight envelope speed limits, which were practically determined by true airspeed as opposed to ground speed, which was used to formulate constraints in the control design process.

During the first half of our experiment, intense maneuvering and deviations from the reference trajectory were significant in both flight tests. This prevented the input-dependent threshold

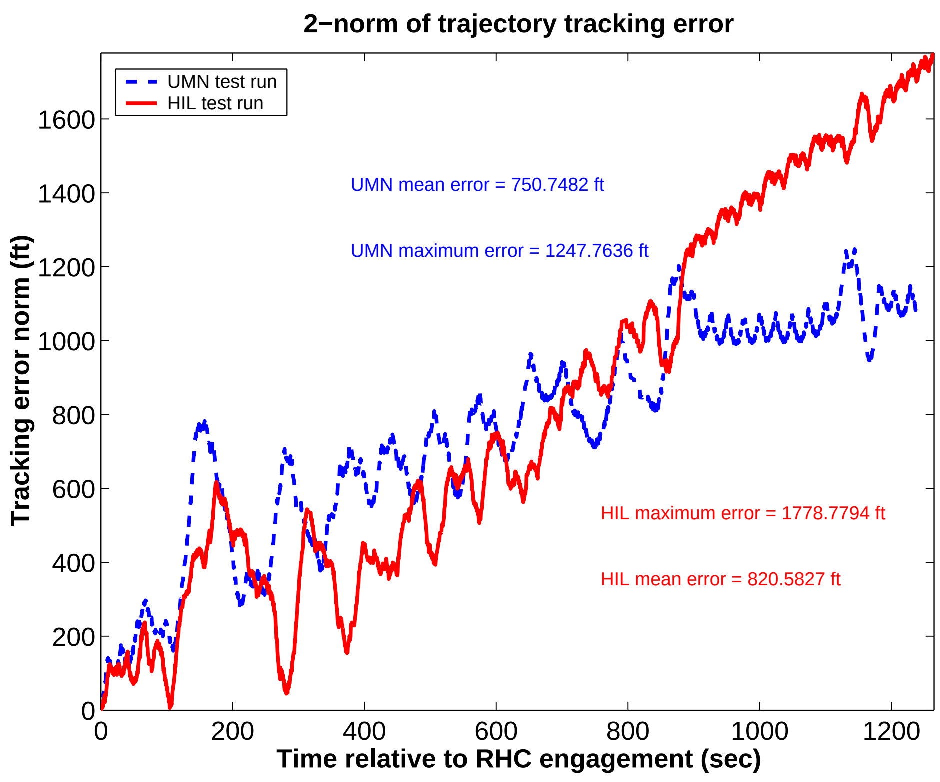


Figure 8.16: Comparison of the 2-norm of hardware-in-the-loop position tracking errors.

residual from sinking below zero by the start of the detection segment, which resulted in a false alarm immediately after the fault was inserted. Thus, flight testing of the fault detection filter were unsuccessful due to the input-dependence of the threshold residual, which relied heavily on excellent RHC tracking performance. This latter was hindered by difficulties in speed control as mentioned in the previous paragraph.

During the final pre-flight preparation phase we learned that close and frequent interaction with people responsible for conducting the final flight test experiment is essential. It is crucial for both influencing the development of the final test software environment and for clarifying elements of the experiments for all parties involved. Deficiencies in the communication of flight experiment details between all “players” of the flight test has even prevented one team from obtaining a single successful flight test that could be evaluated.

In conclusion, the RHC-based guidance system showed amenable robustness properties inspite of the dramatic difference between the velocity tracking behavior of the prediction model and the real system. Performance analysis of flight test data and simulation results with varying wind conditions suggest that the RHC guidance law would have excellent tracking performance using an autonomous speed control system. Further high-fidelity simulations showed that output constraints could be also accommodated using soft constraint formulation. This, along with the observed successful reconfiguration experiments, suggests that the proposed approach provides a high-performance, versatile guidance technology for future unmanned aircraft systems.

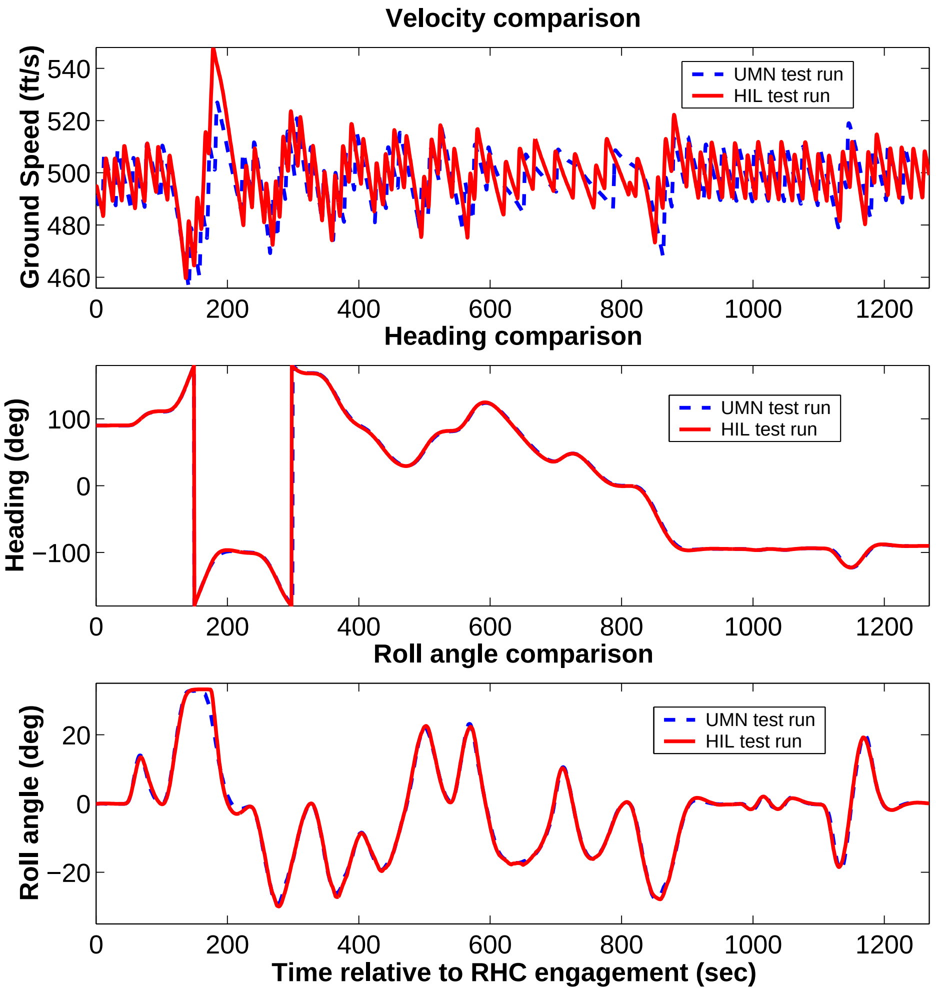


Figure 8.17: Ground speed, heading and roll angle comparison.

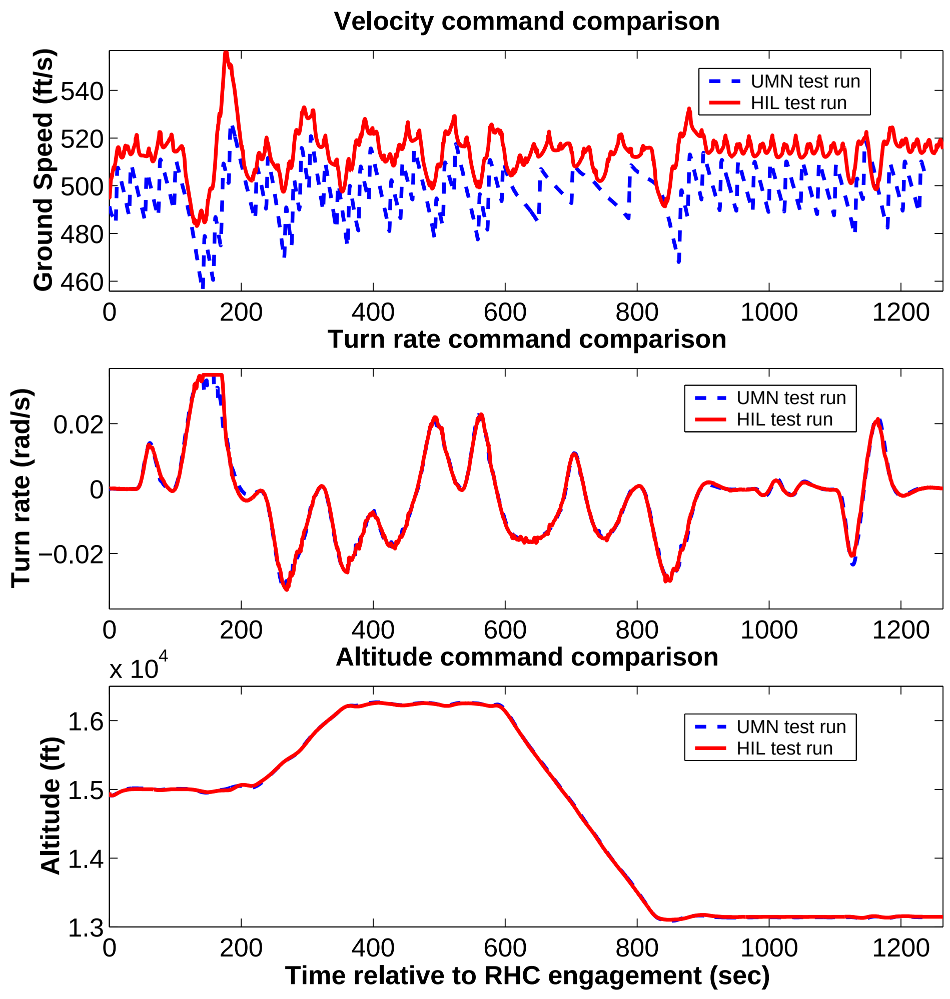


Figure 8.18: Comparison of command signals.

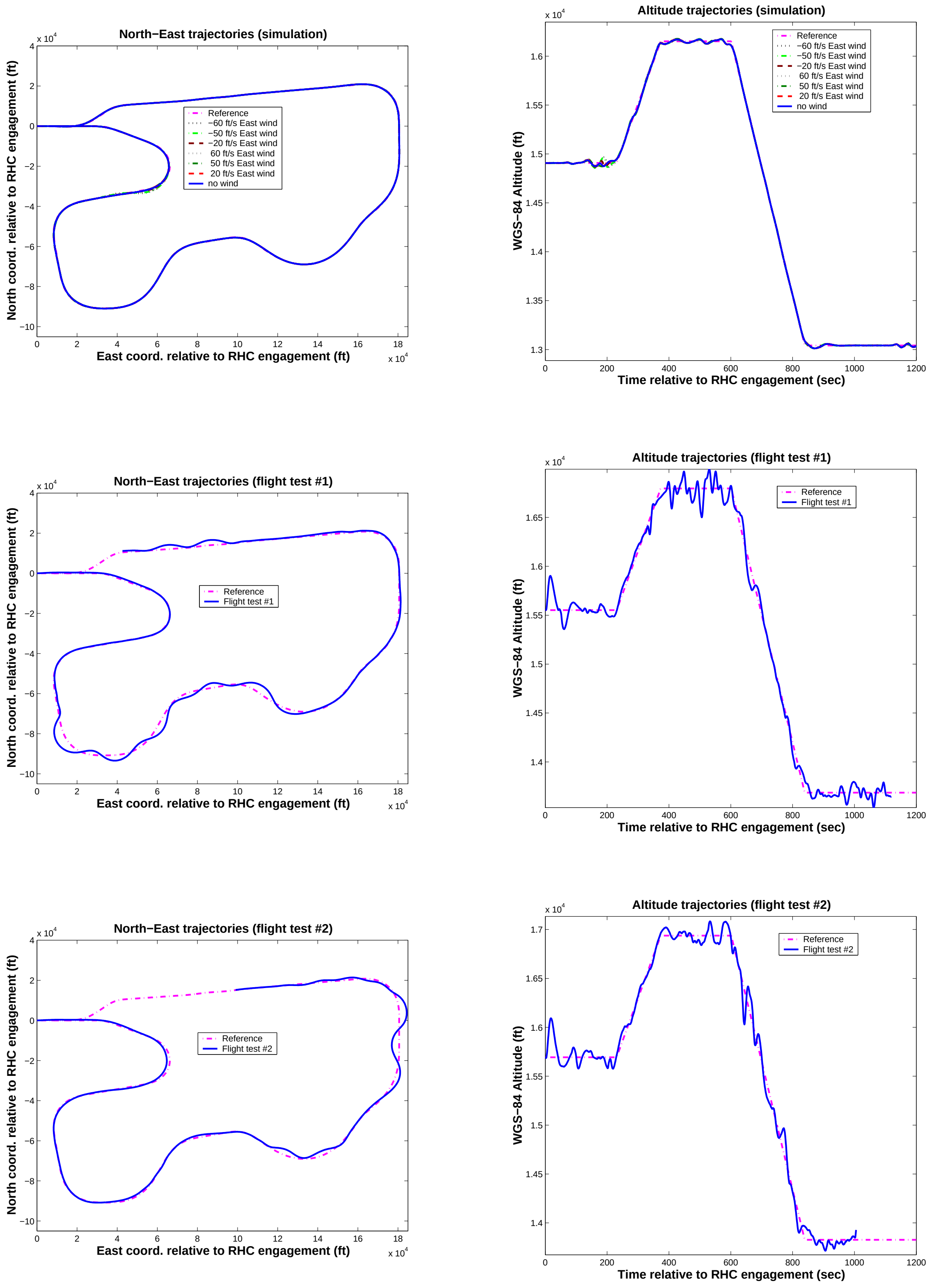


Figure 8.19: North, east and altitude tracking performance. The top two plots show simulation results under different wind conditions. The middle two plots show results of flight test #1. The bottom two plots represent flight test #2 data.

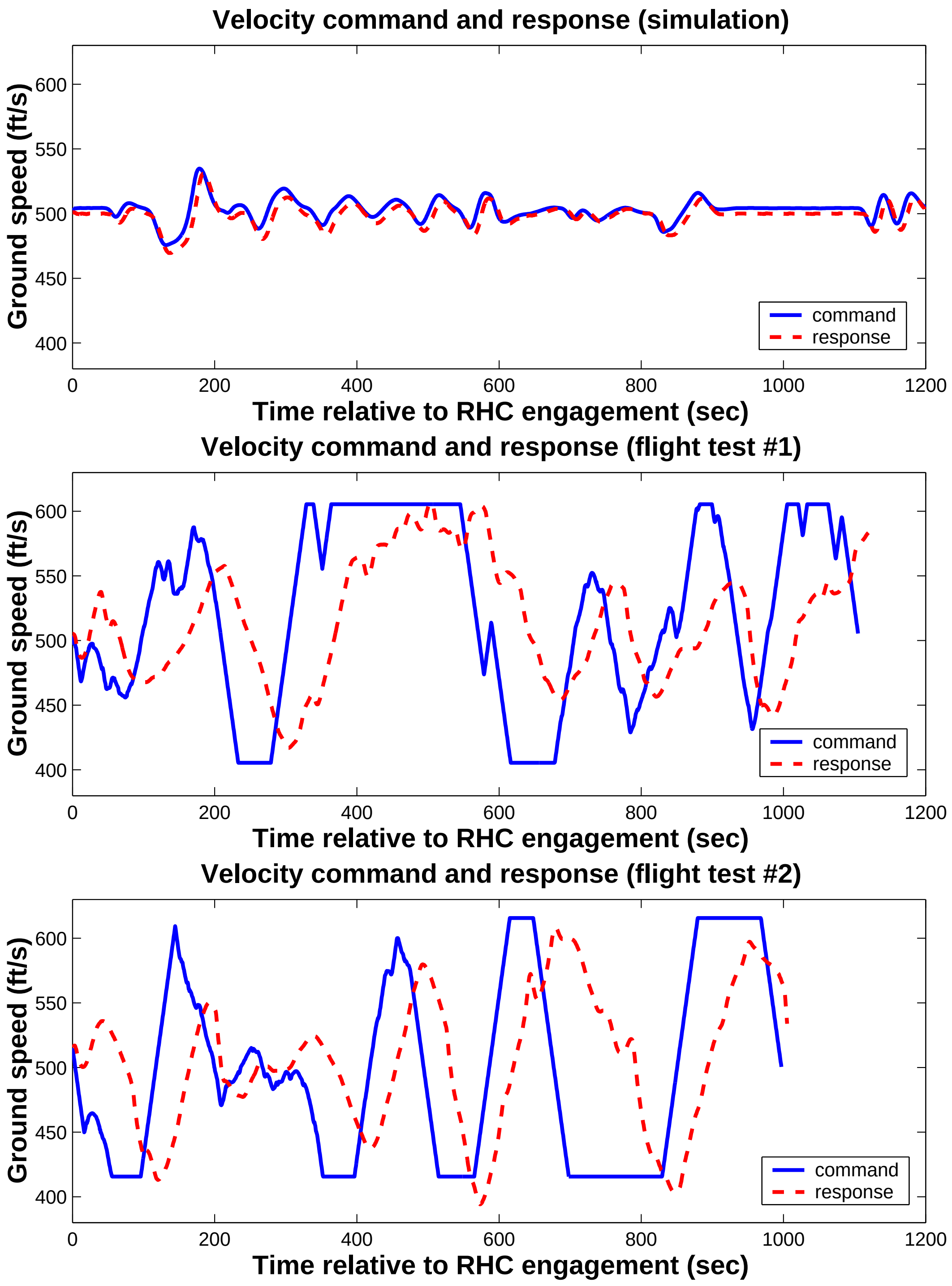


Figure 8.20: Commanded and actual ground speed in simulations and during flight tests #1, #2.

Appendix A

DARPA SEC Capstone Demo Description Experiment #1 Manual

DARPA SEC Capstone Demo Description Experiment #1

University of Minnesota

June 15, 2004

This document describes the Experiment Controller set-up for *Experiment #1* of the University of Minnesota / UC Berkeley SEC Capstone Flight Demonstration. The first few sections contain an informal description of the anticipated sequence of events during the flight test. This will be used to specify expected actions on the pilot cue-cards. The last section provides a description of the experiment controller logic using Sequential Functional Charts (SFC) [33]. In this document, the controller software used by our team will be referred to as UMNUCBC.

UMN/UCB Capstone Demonstration Overview

A summary of the anticipated events planned for the UMN/UCB Capstone Flight Demonstration Experiment is provided as a one-page overview at the end of this document. A more detailed description is provided in the following sections.

A.1 T-33/UCAV actions

Approach to Ingress Point

After take-off, and any testing required by the Boeing Demo Description Document, the pilot will be required to approach the Ingress Point of the test range from the west, flying straight and level at a constant altitude of 15000 ft (WGS84) and ground speed of 500 ft/s, keeping the same latitude as the Ingress Point location. In the meantime, UMNUCBC is issuing constant "SetAndHold" commands to fly to the same flight condition, which are ignored by the autopilot at this stage.

Controller engagement procedure

While en route to the Ingress Point on an approach trajectory described above, the pilot shall press the CMD-ON button to hand over control of the aircraft to the autopilot. The constant "SetAndHold" commands that are issued by UMNUCBC for the autopilot are implemented to continue to fly at the initial flight condition. The command shall approximately correspond to the actual flight condition of the aircraft.

When reaching the longitude corresponding to the Ingress Point, the pilot shall press the START button to engage the RHC controller, if the following criteria are met:

- The autopilot has reached a steady state tracking status.
- Aircraft autopilot is maintaining the specified flight condition up to the tolerances defined in Table A.1.

Measurement	Tolerance
Groundspeed	±15 ft/s
Heading	±0.1 deg
WGS-84 Altitude	±100 ft

Table A.1: Engagement tolerances

Corner	Latitude	Longitude
NW	35.112°	−117.75°
NE	35.112°	−117.6998°
SW	35.088°	−117.6998°
SE	35.088°	−117.75°

Table A.2: Engagement area definition

- The aircraft is within the designated "Engagement Area", described in Table A.2 by the four corner coordinates of the rectangular region.

If these criteria are not met, the pilot shall not press the START button. Instead, the pilot shall disengage by pushing the CMD-OFF button, then take control over the aircraft and repeat procedure from the approach to Ingress Point.

Remark: It is important to engage the RHC controller within the Engagement Area, otherwise the reference trajectory, which starts relative to the position of engagement could violate test range area limitations. Engagement outside the specified area is not allowed.

If the engagement procedure is unsuccessful after pushing the Start button (e.g. because the aircraft leaves the Engagement Area without satisfying flight condition tolerances), a disengagement message is displayed and the experiment has to be repeated from the beginning.

Experiment Phase I

After a successful engagement, UMNUCBC starts tracking a reference trajectory, which leads to the vicinity of a predefined target. Currently the target has predefined parameters described in Table A.3.

There is a window of opportunity within $0 - 450$ seconds after the engagement to initiate a pop-up threat along the route to the target by pushing the SND OBST button. After this time-frame, initiating the pop-up threat will have no effect.

Parameter	**Value**
Latitude	$35.149°$
Longitude	$-117.289°$
Radius	$0.014°$

Table A.3: Target definition

Parameter	**Value**
Latitude	$34.97°$
Longitude	$-117.31°$
Radius	$0.04°$

Table A.4: Pop-up threat definition

Experiment Phase II

After 450 seconds have passed since the time of engagement, the controller will track an approach trajectory to the target, which is dependent on whether the pop-up threat obstacle has been sent using the SND OBST button in Phase I. Pushing the SND OBST button and sending the pop-up threat obstacle in this phase of the experiment will not have any effect on the reference trajectory. When the aircraft has finished avoiding the pop-up threat, a message is displayed and it continues flying towards the target. This second phase of the experiment ends with flying over the target. The pop-up threat location is predefined according to Table A.4.

Experiment Phase III

After passing the target the aircraft should be flying straight and level. The pilot is expected to push the FAULT-ON button when the *"Target reached"* message is displayed or the aircraft exits the target circle, but no later than 30 seconds after the target is reached. This initiates the corruption of controller signals with the output of the fault simulator, and the observing of the Fault Detection filter output. As soon as the fault is detected, the fault simulator is switched off automatically.

The trajectory in this phase of the experiment leads back the aircraft to the vicinity of the engagement area, where the pilot should disengage by pushing the CMD-OFF button and taking control over the aircraft to leave the flight test area, either for landing or a rerun.

Disengagement can be initiated based on the pilot's decision, once the aircraft is back in the engagement area, or after the status display: "Experiment ended. Disengage controller!".

If the controller is deviating "significantly" from the reference trajectory after the initiation of the fault, the pilot can also take control by pushing CMD-OFF.

Reset functionality

The RESET button should be pushed only if the aircraft is in an approach to the Ingress Point. The procedure described in the "Approach to Ingress Point" section should be repeated after the RESET button is pushed.

A.2 Explanation of SFC diagrams, further remarks

General remarks

Goal of the SFC design is to simplify the experiment control logic by relying on pilot to execute actions and button-presses in the right order, according to the specified sequence and criteria.

Main Sequential Functional Chart (SFC 1)

This SFC defines the experiment control logic around the Receding Horizon Controller (RHC) from engagement until the end of the experiment.

The notation $t/X_i/T_i$ represents the value of a boolean variable that becomes true when the time T_i elapses after the activation of the state X_i, i.e. it represents a timer which starts when the state X_i is activated. This notation is used to define a time window during which the Pop-Up Threat indicator button value can be set.

After the experiment has ended by completing the trajectory, UMNUCBC stops sending Set-AndHold commands (i.e. the last ones issued remain valid) at the end of the trajectory and a disengagement message is displayed. The pilot shall disengage by taking control (CMD-OFF) and exit from the test area. The controller could be restarted by following the Approach to Ingress Point procedure and pushing the RESET button.

Fault Simulator / Detection button (SFC 2)

The fault simulator and the fault detection filter are started when the application starts up and stopped if the engagement procedure was unsuccessful or when the aircraft reaches the end of the reference trajectory ("End of experiment" condition). This SFC represents two logical states that determine whether the output of the fault simulator should be added to the controller output for implementation and whether to look at the output of the fault detection filter. The transition between these two states can be initiated by pressing the FAULT-ON and FAULT-OFF buttons. Once the fault has been initiated, the value of the FD flag, which is set based on the output of the fault detection filter, could also lead to cancelling the fault (i.e. reconfiguration after fault detection).

Pop-up threat (SFC 3)

This SFC for pop-up threat is used to set a flag, which indicates which trajectory to follow from two alternatives: one that avoids the threat and the other (default) that goes through it. The two trajectories are similar except for the segment near the pop-up threat. The two trajectories are loaded from file during initialization, and the marching trajectory window, which specifies the references for the next horizon in the RHC controller, fills up the `RefTrajEast`, `RefTrajNorth`, `RefTrajAlt` variables based on the trajectory indicated by the pop-up flag. (Note that since the circular no-fly-zones are defined as non-threat obstacles, the `ObstacleAvailable` method is called at the beginning of the experiment even before the pop-up threat obstacle is sent using the

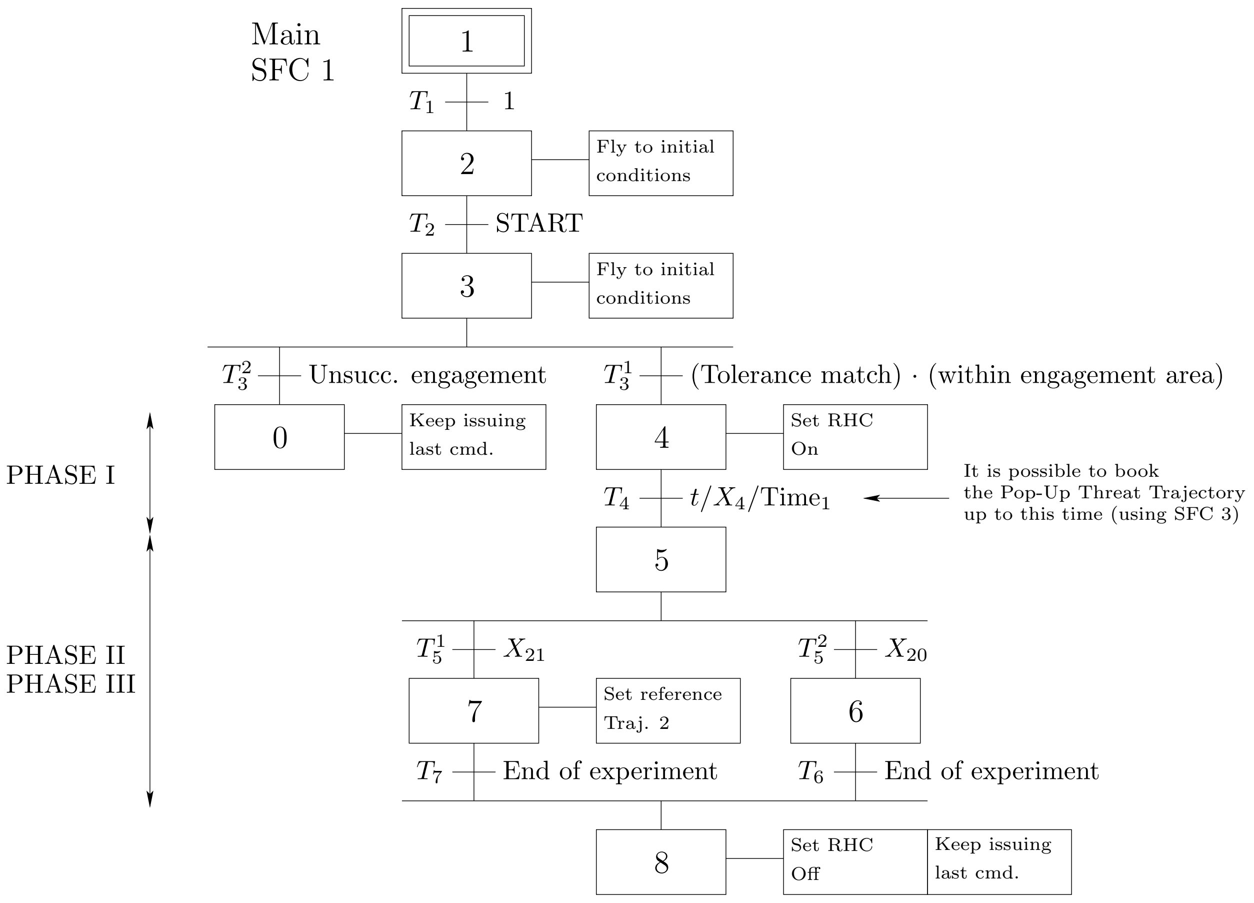


Figure A.1: Main SFC (SFC 1).

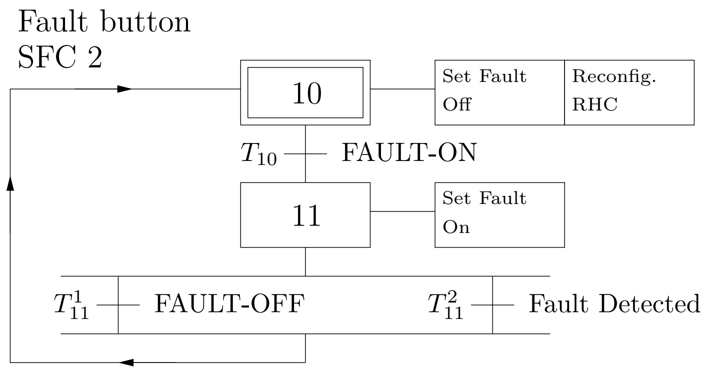


Figure A.2: Fault on/off button (SFC 2).

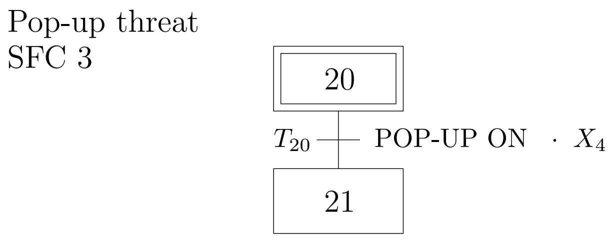


Figure A.3: Pop-up threat SFC (SFC 3).

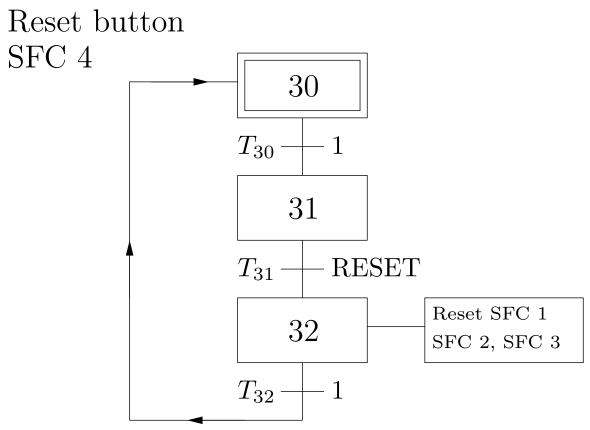


Figure A.4: Reset button (SFC 4).

SND OBST button. Since a call to the `ObstacleAvailable` method sets the `popup_ON_button` flag true, its value is always reset to false in the X_3 state. The SND OBST button should only be used in state X_4 to send the pop-up threat obstacle, so the condition of being in state X_4 is used as a necessary requirement for transition T_{20}. This avoids switching reference trajectories just because the non-threat obstacles are displayed in the beginning of the experiment.)

Reset button (SFC 4)

Once the controller is started (START button press), pressing the RESET button resets the SFC 1, the SFC 2, and the SFC 3 to the their initial phases, thus reinitializing the whole controller. The pilot will be instructed to press the button only if the aircraft is in the Approach to Ingress Point phase.

Status displays related to transitions of SFC diagrams

The messages listed in Table A.6 are displayed when the corresponding state-transitions occur.

Further information

The location and radius of circular no-fly-zones are given in Table A.5. These objects are defined as non-threat obstacles in the Experiment Controller.

The list of user-defined buttons is the following

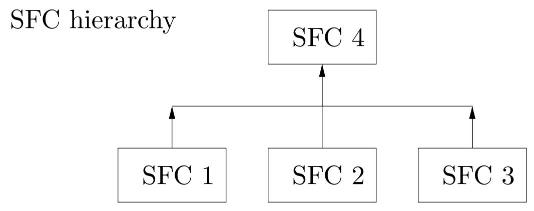


Figure A.5: Hierarchy of the SFCs. SFC 4 is the only one capable of resetting all the other SFCs, by pressing of the reset button.

- START
- FAULT-ON
- FAULT-OFF

NFZ #	**Latitude**	**Longitude**	**Radius**
1	34.927°	−117.61°	0.047°
2	35.042°	−117.405°	0.047°
3	35.078°	−117.22°	0.053°

Table A.5: No-fly-zone area definitions

SFC Transitions	**Status Display Message**
State 1 (initialization)	University of Minnesota / UC Berkeley SEC flight demonstration.
T_1	Commands initialized.
T_2	Experiment started. Flying to initial condition.
T_3^1	Controller engaged. Phase I started.
T_3^2	RHC engagement unsuccessful. Disengage controller!
T_4	UAV assigned to target. Phase II started.
T_5^1	Avoiding pop-up threat.
T_5^2	No pop-up threat invoked.
N/A	Threat avoided. Continuing towards target.
N/A	Target reached.
T_6	Experiment ended. Disengage controller!
T_7	Experiment ended. Disengage controller!
T_{10}	Fault initiated. Phase III started.
T_{11}^1	Fault removed.
T_{11}^2	Fault detected. Controller reconfigured.
T_{20}	Pop-up threat inserted.
T_{31}	Experiment is reset.
T_{32}	Controller is reinitialized.

Table A.6: Status display messages

UMN/UCB Capstone Demonstration Overview

1. Approach to Ingress Point
   - Fly straight and level at 15K ft, 500 ft/s ground speed directly approaching from West.
2. Engagement of Controller
   - En route to Ingress Point press CMD-ON button.
   - The "SetAndHold' commands issued by UMNUCBC for the autopilot are implemented to fly to initial condition.
   - Press START button at Ingress Point longitude to engage
     - IF
       - Aircraft autopilot maintaining flight condition up to ± 15 ft/s in ground speed, ± 0.1 degrees in heading angle and ± 100 ft in altitude.
     - ELSE
       - Do NOT press START button.
       - Disengage controller by pressing CMD-OFF button.
       - Pilot in control of aircraft, return to Approach to Ingress Point (1).
     - END
3. UMNUCBC tracking reference trajectory
   - Between 0-450 seconds, can initiate pop-up threat by use of SND OBST button.
4. UMNUCBC tracking reference trajectory
   - No POP-UP: track original reference trajectory to target.
   - POP-UP: track modified reference trajectory around Pop-Up to target.
   - Fly over target.
5. Initiate actuator fault
   - After target fly over (i.e. within 0-30 seconds after *"Target reached."* message is displayed), initiate actuator fault with FtSm-ON button. This automatically:
     - Engages fault simulator.
     - Activates fault detection logic.
     - Detects fault, reconfigures UMNUCBC system.
     - Disengages fault simulator.
6. Track return trajectory to Egress Point.
   - Press CMD-OFF button to disengage UMNUCBC at Egress Point.
   - Return control to Pilot.
7. Return to Ingress Point
   - New test or land based on pilot/team decision.

Appendix B

DARPA SEC Capstone Demo Description Experiment #2 Manual

DARPA SEC Capstone Demo Description Experiment #2

University of Minnesota

June 15, 2004

This document describes the Experiment Controller set-up for *Experiment #2* of the University of Minnesota / UC Berkeley SEC Capstone Flight Demonstration. Section B.1 contains an informal description of the anticipated sequence of events during the flight test. This will be used to specify expected actions on the pilot cue-cards. Section B.2 provides a description of the experiment controller logic using Sequential Functional Charts (SFC) [33]. In this document, the controller software used by our team will be referred to as UMNUCBC.

Note that the second experiment plan differs from the first UMN/UCB demo in that Experiment Phase III is extended and contains more events that increase the length of the overall test run. Besides changes in the underlying algorithm, the most significant difference is the reconfiguration of the RHC controller after a successful fault detection (as opposed to a complete fault removal in Experiment #1), and the ability to invoke multiple pop-up threats in the path of the "faulty" aircraft, while it's "returning" to the Egress Point. Experiment #2 has the exact same sequence of events in Phase I and Phase II as in Experiment #1 to minimize changes to the overall mission setup. Those sections that remain unchanged from the aspect of experiment control, are marked with a bar on the right margin of the document.

UMN/UCB Capstone Demonstration Experiment #2 Overview

A summary of the anticipated events planned for the UMN/UCB Capstone Flight Demonstration Experiment #2 is provided as a two-page overview at the end of this document. A more detailed description is provided in the following section.

B.1 T-33/UCAV actions

Approach to Ingress Point

After take-off, and any testing required by the Boeing Demo Description Document, the pilot will be required to approach the Ingress Point of the test range from the west, flying straight and level at a constant altitude of 15000 ft (WGS84) and ground speed of 500 ft/s, keeping the same latitude as the Ingress Point location. In the meantime, UMNUCBC is issuing constant "SetAndHold" commands to fly to the same flight condition, which are ignored by the autopilot at this stage.

Controller engagement procedure

While en route to the Ingress Point on an approach trajectory described above, the pilot shall press the CMD-ON button to hand over control of the aircraft to the autopilot. The constant

Measurement	**Tolerance**
Groundspeed	±15 ft/s
Heading	±0.1 deg
WGS-84 Altitude	±100 ft

Table B.1: Engagement tolerances

Corner	**Latitude**	**Longitude**
NW	35.112°	−117.75°
NE	35.112°	−117.6998°
SW	35.088°	−117.6998°
SE	35.088°	−117.75°

Table B.2: Engagement area definition

“SetAndHold” commands that are issued by UMNUCBC for the autopilot are implemented to continue to fly at the initial flight condition. The command shall approximately correspond to the actual flight condition of the aircraft.

When reaching the longitude corresponding to the Ingress Point, the pilot shall press the START button to engage the RHC controller, if the following criteria are met:

- The autopilot has reached a steady state tracking status.
- Aircraft autopilot is maintaining the specified flight condition up to the tolerances defined in Table B.1.
- The aircraft is within the designated “Engagement Area”, described in Table B.2 by the four corner coordinates of the rectangular region.

If these criteria are not met, the pilot shall not press the START button. Instead, the pilot shall disengage by pushing the CMD-OFF button, then take control over the aircraft and repeat procedure from the approach to Ingress Point.

Remark: It is important to engage the RHC controller within the Engagement Area, otherwise the reference trajectory, which starts relative to the position of engagement could violate test range area limitations. Engagement outside the specified area is not allowed.

If the engagement procedure is unsuccessful after pushing the Start button (e.g. because the aircraft leaves the Engagement Area without satisfying flight condition tolerances), a disengagement message is displayed and the experiment has to be repeated from the beginning.

Parameter	Value
Latitude	35.149°
Longitude	−117.289°
Radius	0.014°

Table B.3: Target definition

Pop-up threat name	Latitude	Longitude	Radius
POP-UP1	34.97°	−117.29°	0.04°
POP-UP2a	35.105°	−117.597°	0.035°
POP-UP2b	35.04°	−117.608°	0.02°

Table B.4: Pop-up threat definitions

Experiment Phase I

After a successful engagement, UMNUCBC starts tracking a reference trajectory, which leads to the vicinity of a predefined target. Currently the target has predefined parameters described in Table B.3.

There is a window of opportunity within $0-450$ seconds after the engagement to initiate a pop-up threat along the route to the target by pushing the SND OBST button and selecting POP-UP1. After this time-frame, initiating the pop-up threat will have no effect.

Experiment Phase II

After 450 seconds have passed since the time of engagement, the controller will track an approach trajectory to the target, which is dependent on whether the pop-up threat obstacle POP-UP1 has been sent using the SND OBST button in Phase I. Pushing the SND OBST button and sending the pop-up threat obstacle called POP-UP1 in this phase of the experiment will not have any effect on the reference trajectory. When the aircraft has finished avoiding the pop-up threat, a message is displayed and it continues flying towards the target. This second phase of the experiment ends with flying over the target. The location of the POP-UP1 threat is predefined according to Table B.4.

Experiment Phase III

After passing the target the aircraft should be flying straight and level. The pilot is expected to push the FAULT-ON button when the *"Target reached"* message is displayed or the aircraft leaves the target circle, but no later than 30 seconds after the target is reached. This initiates the corruption of controller signals with the output of the fault simulator, and the observing of the

Fault Detection filter output. As soon as the fault is detected, the RHC controller is reconfigured and continues tracking the reference trajectory with the "faulty" aircraft model. The trajectory in this phase of the experiment leads back the aircraft to the vicinity of the engagement area if no other pop-up threats are invoked.

En route to the Egress Point, two different pop-up threats can be sent to obstruct the path of the "returning" aircraft. These are called POP-UP2a and POP-UP2b with parameters described in Table B.4. The threat called POP-UP2a can be invoked only within 0-110 seconds after the *"Target reached"* message is displayed.

After 110 seconds have passed since the target was reached, the controller will continue to track a trajectory to the Egress Point, which is dependent on whether the pop-up threat obstacle POP-UP2a has been sent using the SND OBST since the target. Pushing the SND OBST button and sending the pop-up threat obstacle called POP-UP2a outside the specified time interval will not have any effect on the reference trajectory.

The pop-up threat called POP-UP2b can only be invoked if POP-UP2a was already sent, and the aircraft is avoiding it. The threat called POP-UP2b can be invoked only within 0-400 seconds after the *"Avoiding POP-UP2a"* message is displayed. After 400 seconds have passed since the aircraft began to avoid POP-UP2a, the controller will continue to track a trajectory to the Egress Point, which is dependent on whether the pop-up threat obstacle POP-UP2b has been sent using the SND OBST. Pushing the SND OBST button and sending the pop-up threat obstacle called POP-UP2b outside the specified time interval will not have any effect on the reference trajectory.

Figure B.1 illustrates the alternative mission trajectories and the approximate "decision" points until which the corresponding pop-up obstacles have to be invoked. These approximate location of the "deadline" points are indicated by black arrows. The figure depicts the aircraft trajectory at the end of the experiment in case all the pop-up threats were invoked. Please note again that POP-UP2b can be invoked only after the aircraft starts avoiding POP-UP2a.

Independent of the number of pop-up threats invoked during the experiment, the reference trajectory leads the aircraft back to the Engagement Area, where the pilot should disengage by pushing the CMD-OFF button and taking control over the aircraft to leave the flight test area, either for landing or a rerun.

Disengagement can be initiated based on the pilot's decision, once the aircraft is back in the engagement area, or after the status display: "Experiment ended. Disengage controller!".

If the controller is deviating "significantly" from the reference trajectory after the initiation of the fault, the pilot can also take control by pushing CMD-OFF.

Reset functionality

The RESET button should be pushed only if the aircraft is in an approach to the Ingress Point. The procedure described in the "Approach to Ingress Point" section should be repeated after the RESET button is pushed.

B.2 Explanation of SFC diagrams, further remarks

General remarks

Goal of the SFC design is to simplify the experiment control logic by relying on pilot to execute actions and button-presses in the right order, according to the specified sequence and criteria.

Main Sequential Functional Chart (SFC 1)

This SFC defines the experiment control logic around the Receding Horizon Controller (RHC) from engagement until the end of the experiment.

The notation $t/X_i/T_i$ represents the value of a boolean variable that becomes true when the time T_i elapses after the activation of the state X_i, i.e. it represents a timer which starts when the state X_i is activated. This notation is used to define time windows during which the pop-up threat indicator values can be set.

After the experiment has ended by completing the trajectory, UMNUCBC stops sending Set-AndHold commands (i.e. the last ones issued remain valid) at the end of the trajectory and a disengagement message is displayed. The pilot shall disengage by taking control (CMD-OFF) and exit from the test area. The controller could be restarted by following the Approach to Ingress Point procedure and pushing the RESET button.

Fault Simulator / Detection button (SFC 2)

The fault simulator and the fault detection filter are started when the application starts up and stopped if the engagement procedure was unsuccessful or when the aircraft reaches the end of the reference trajectory ("End of experiment" condition). This SFC represents three logical states. These determine whether the output of the fault simulator should be added to the controller output for implementation, whether to interpret the output of the fault detection filter or to reconfigure the RHC controller. The transition between the first two states can be initiated by pressing the FtSm-ON and FtSm-OFF buttons. Once the fault has been initiated, the value of the fault detection flag, which is set based on the output of the fault detection filter, can lead to the reconfiguration of the RHC controller when the fault is detected. Note that after the detection of the fault, the fault simulator cannot be removed, it continues corrupting the turn rate output of the controller until the end of the experiment. Reconfiguration is performed by switching to a second instance of the RHC controller with modified prediction model and constraints.

Pop-up threat (SFC 3)

This SFC for pop-up threat is used to indicate which trajectory to follow from six alternatives, depending upon which pop-up threats have to be avoided. The alternative trajectories are similar up to the points of avoiding the pop-up threats. The trajectories are loaded from file during initialization, and the marching trajectory window, which specifies the references for the next horizon in the RHC controller, fills up the `RefTrajEast`, `RefTrajNorth`, `RefTrajAlt` variables based on the trajectory indicated by SFC 3. (Note that since the circular no-fly-zones are defined as non-threat obstacles, the `ObstacleAvailable` method is called at the beginning of the experiment even before any pop-up threat obstacle is sent using the SND OBST button. Within the `ObstacleAvailable` method, the location of the received obstacle is used to determine which pop-up threat was invoked and athe corresponding flag is set to true. The SND OBST button should only be used in states $X_4, X_8, X_9, X_{12}, X_{14}$ to send the corresponding pop-up threat obstacle, so the condition of being in these states is used as a necessary requirement for transitions $T^1_{200}, T_{202}, T_{203}$. This avoids switching reference trajectories just because the non-threat obstacles are displayed in the beginning of the experiment or any other unforeseen events such as sending obstacles at incorrect times.)

Reset button (SFC 4)

Once the controller is started (START button press), pressing the RESET button resets the SFC 1, the SFC 2, and the SFC 3 to the their initial phases, thus reinitializing the whole controller. The pilot will be instructed to press the button only if the aircraft is in the Approach to Ingress Point phase.

Status displays related to transitions of SFC diagrams

The messages listed in Table B.6 and B.7 are displayed when the corresponding state-transitions occur.

Further information

The location and radius of circular no-fly-zones are given in Table B.5. These objects are defined as non-threat obstacles in the Experiment Controller.

The list of user-defined buttons is the following

- START
- FAULT-ON
- FAULT-OFF

NFZ #	Latitude	Longitude	Radius
1	34.927°	−117.617°	0.04°
2	35.054°	−117.417°	0.047°
3	35.078°	−117.22°	0.053°
4	35.183°	−117.677°	0.057°
5	35.1°	−117.46°	0.02°

Table B.5: No-fly-zone area definitions

SFC Transitions	**Status Display Message**
State 1 (initialization)	University of Minnesota / UC Berkeley SEC flight demonstration.
T_1	Commands initialized.
T_2	Experiment started. Flying to initial condition.
T_3^1	Controller engaged. Phase I started.
T_3^2	RHC engagement unsuccessful. Disengage controller!
T_4	UAV assigned to target. Phase II started.
T_5^1	Avoiding POP-UP1 threat.
T_5^2	POP-UP1 threat was not invoked.
N/A	Threat avoided. Continuing towards target.
T_6	Target reached.
T_7	Target reached.
T_{10}^1	Avoiding POP-UP2a threat.
T_{10}^2	POP-UP2a threat was not invoked.
T_{11}^1	Avoiding POP-UP2a threat.
T_{11}^2	POP-UP2a threat was not invoked.
T_{13}	Experiment ended. Disengage controller!
T_{15}	Experiment ended. Disengage controller!
T_{16}^1	Avoiding POP-UP2b threat.
T_{16}^2	POP-UP2b threat was not invoked.
T_{17}^1	Avoiding POP-UP2b threat.
T_{17}^2	POP-UP2b threat was not invoked.

Table B.6: Status display messages

SFC Transitions	**Status Display Message**
T_{18}	Experiment ended. Disengage controller!
T_{19}	Experiment ended. Disengage controller!
T_{20}	Experiment ended. Disengage controller!
T_{21}	Experiment ended. Disengage controller!
T_{100}	Fault initiated. Phase III started.
T_{101}^{1}	Fault removed.
T_{101}^{2}	Fault detected. Controller reconfigured.
T_{200}^{1}	POP-UP1 threat inserted.
T_{202}	POP-UP2a threat inserted.
T_{203}	POP-UP2b threat inserted.
T_{301}	Experiment is reset.
T_{302}	Controller is reinitialized.

Table B.7: Status display messages (cont'd)

UMN/UCB Capstone Demonstration Experiment #2 Overview

1. Approach to Ingress Point
   - Fly straight and level at 15K ft, 500 ft/s ground speed directly approaching from West.
2. Engagement of Controller
   - En route to Ingress Point press CMD-ON button.
   - The "SetAndHold' commands issued by UMNUCBC for the autopilot are implemented to fly to initial condition.
   - Press START button at Ingress Point longitude to engage
     - IF
       - Aircraft autopilot maintaining flight condition up to ±15 ft/s in ground speed, ±0.1 degrees in heading angle and ±100 ft in altitude.
     - ELSE
       - Do NOT press START button.
       - Disengage controller by pressing CMD-OFF button.
       - Pilot in control of aircraft, return to Approach to Ingress Point (1).
     - END
3. UMNUCBC tracking reference trajectory
   - Between 0-450 seconds, can initiate pop-up threat POP-UP1 by use of SND OBST button.
4. UMNUCBC tracking reference trajectory
   - No POP-UP: track original reference trajectory to target.
   - POP-UP: track modified reference trajectory around Pop-Up to target.
   - Fly over target.
5. Initiate actuator fault
   - After target fly over (i.e. within 0-30 seconds after *"Target reached."* message is displayed), initiate actuator fault with FtSm-ON button. This automatically:
     - Engages fault simulator.
     - Activates fault detection logic.
     - Detects fault, reconfigures UMNUCBC system.
6. Track return trajectory to Egress Point.
   - Between 0-110 seconds after the target, can initiate pop-up threat POP-UP2a by use of SND OBST button.
     - No POP-UP2a: track original reference trajectory to Egress Point.
     - POP-UP2a: track modified reference trajectory around Pop-Up to Egress Point.

    - Between 0-400 seconds after POP-UP2a avoidance started, can initiate second pop-up threat POP-UP2b by use of SND OBST button.
- Aircraft eventually returns to Engagement Area after avoiding pop-ups.
- Press CMD-OFF button to disengage UMNUCBC at Egress Point.
- Return control to Pilot.

7. Return to Ingress Point

    - New test or land based on pilot/team decision.

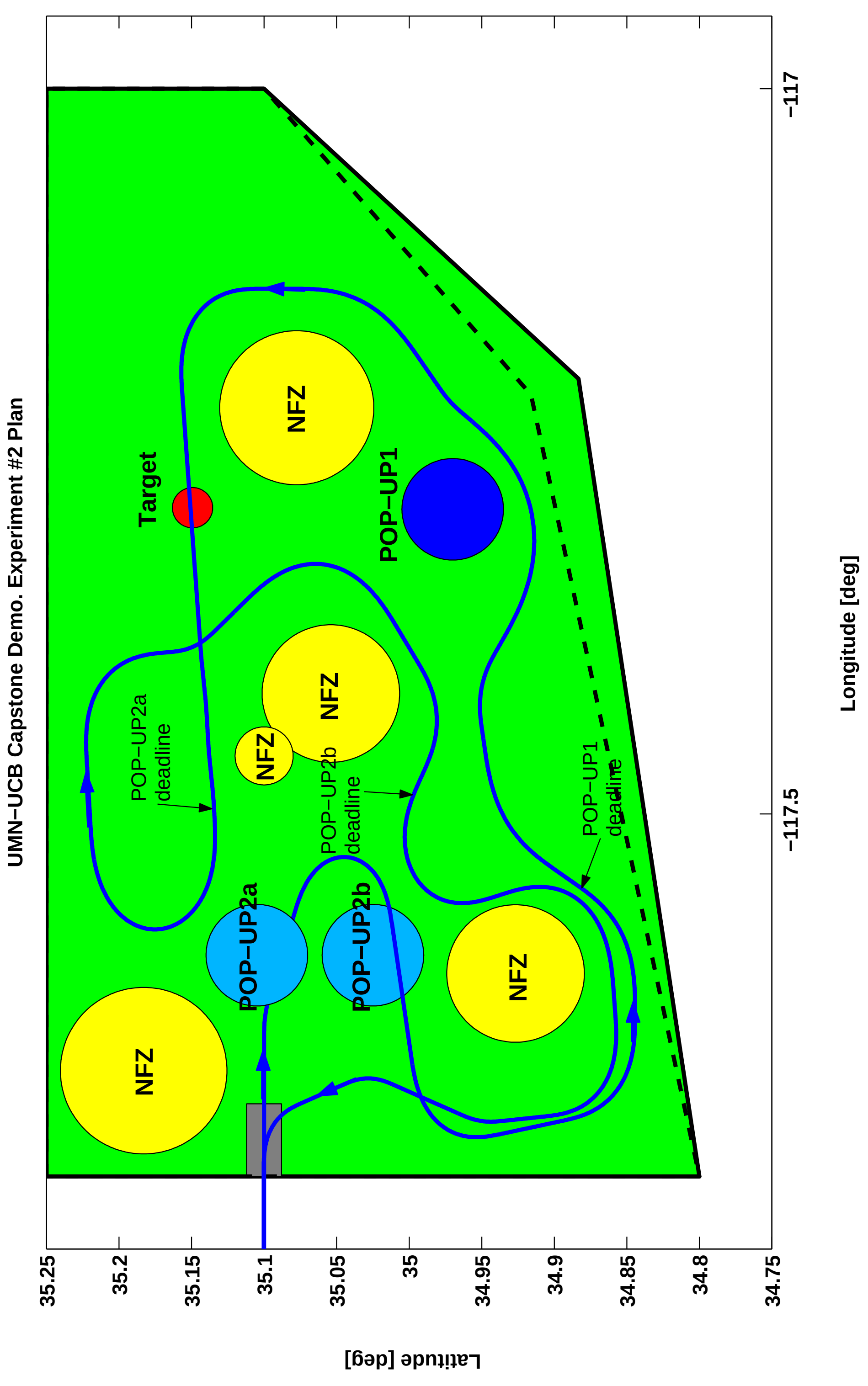


Figure B.1: Illustration of the reference trajectory in the flight test area with our desired target, pop-up threat and no-fly-zone locations. The displayed trajectory corresponds to invoking all the pop-up threats. Approximate locations of timeouts associated with invoking threats are indicated with black arrows.

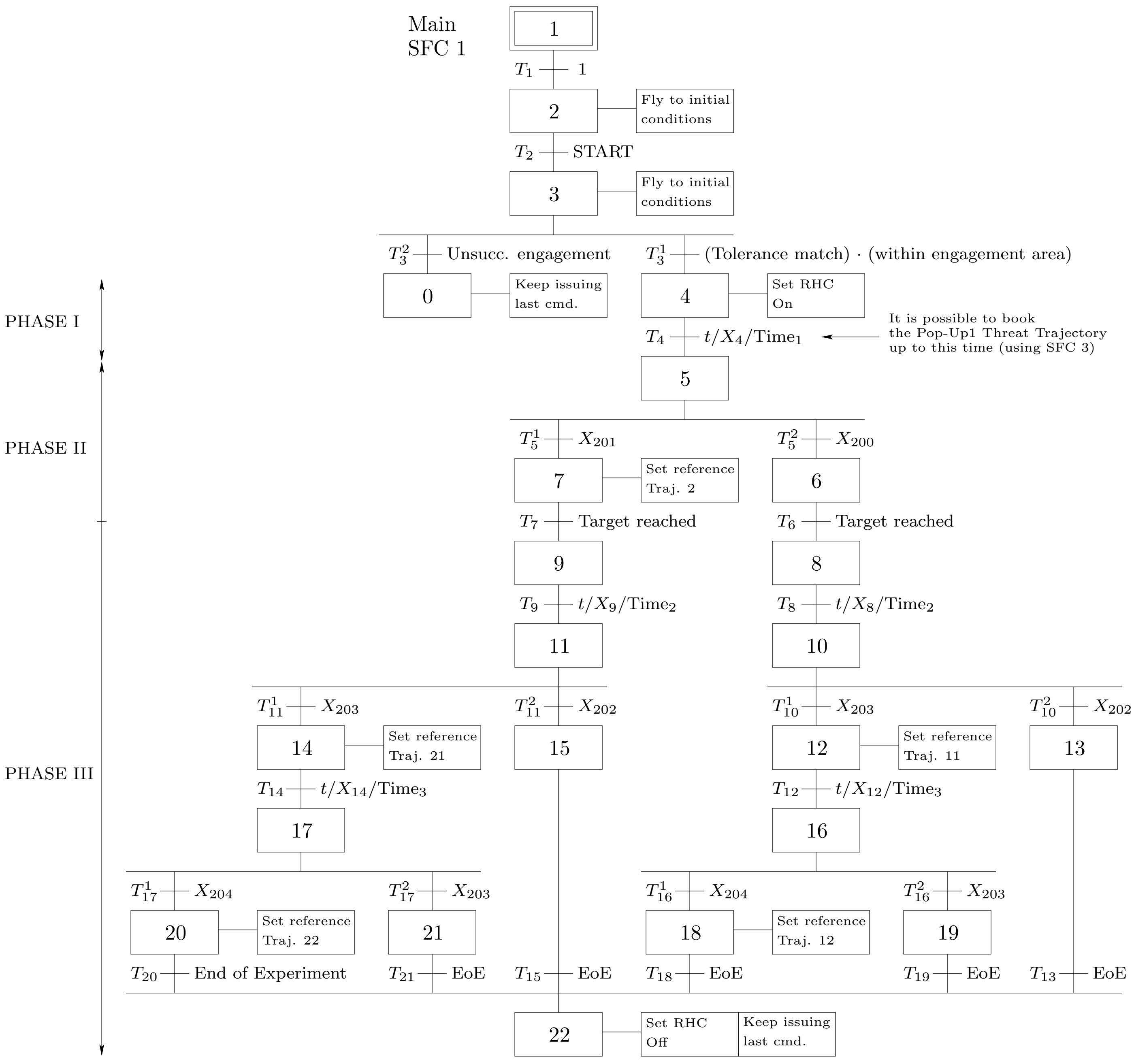


Figure B.2: Main SFC (SFC 1).

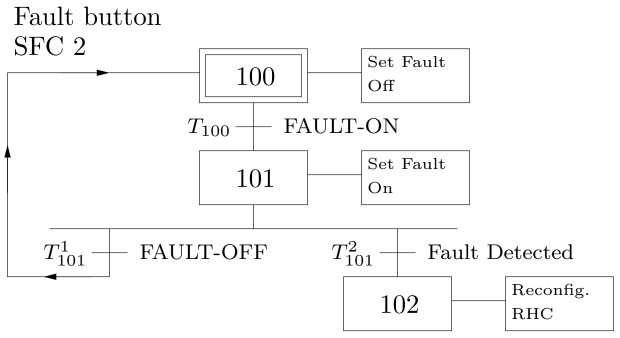


Figure B.3: Fault on/off button (SFC 2).

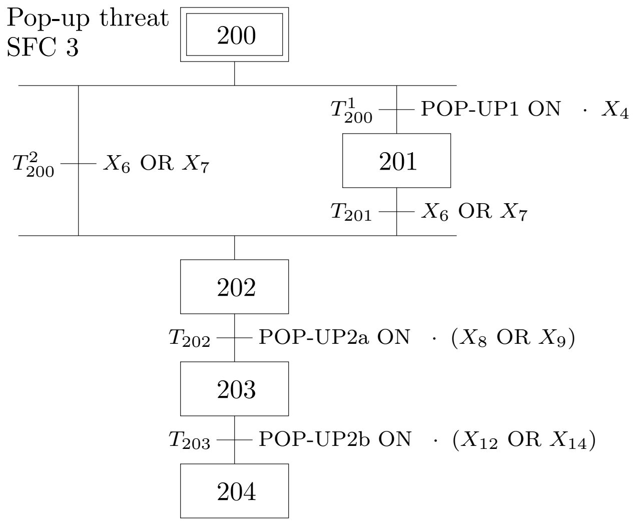


Figure B.4: Pop-up threat SFC (SFC 3).

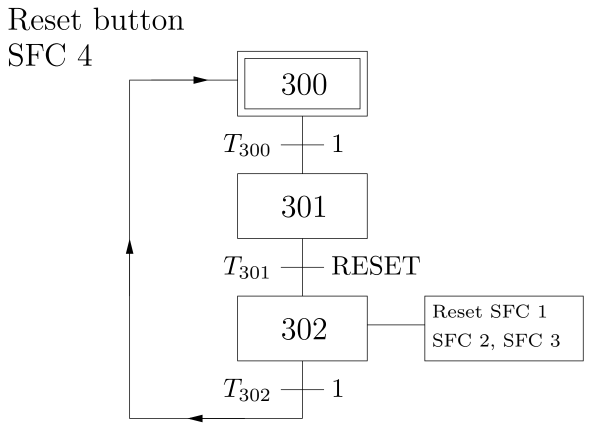


Figure B.5: Reset button (SFC 4).

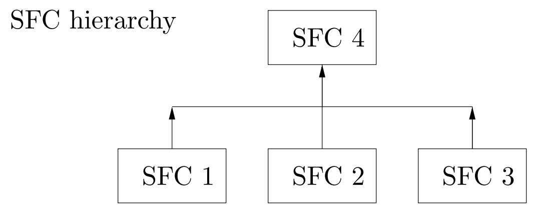


Figure B.6: Hierarchy of the SFCs. SFC 4 is the only one capable of resetting all the other SFCs, by pressing of the reset button.

Appendix C

UML essentials

In this appendix a brief explanation of the UML notations used in Chapter 7 is reviewed, however the reader is referred to [34] for a more detailed introduction to the UML and to [35] for a complete discussion of the notation, while [36] discusses the design of real-time systems using the UML.

The UML notation supports nine diagrams to reflect the various aspect of a software system that needs to be specified formally. In a system specification it is not mandatory to use all the diagrams, so in the following only a brief description of the notation associated with diagrams used in this report is given:

- The use case diagram (described in Section C.1).
- The class diagram (described in Section C.2).
- The state-chart diagram (described in Section C.3).

C.1 Use case diagrams

The use case diagram is a technique for capturing the functional requirements of a system. Use cases work by defining the typical interactions between the users of a system and the system itself, providing a narrative of how a system is used. A *scenario* is a sequence of steps describing an interaction between a user and a system, and a use case is a set of scenarios tied together by a common user goal.

An *actor* initiates a use case. An actor is depicted as a stick figure on a use case diagram. The system is depicted as a box. A use case is depicted as an ellipse inside the box. Communications associations connect actors with the use cases in which they participate. Relationships among use cases are defined by means of *include* and *extend* relationships. The notation is depicted in Figure C.1.

C.2 Class diagrams

In a *class diagram*, classes are depicted as boxes (see Figure C.2) and the static relationships between them are depicted as arcs. The following three main types of relationships between classes are supported:

Associations An association between two classes, which is referred as a binary association, is represented as a line joining two class boxes. An association has a name and optionally a

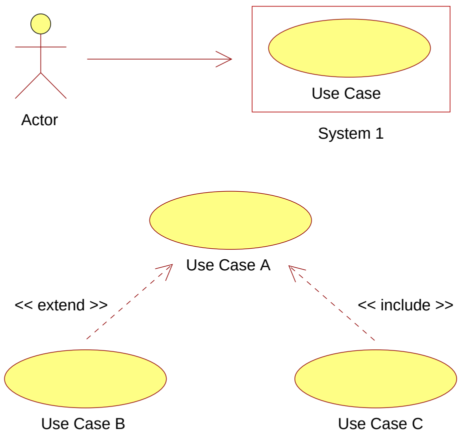


Figure C.1: UML notation for use case diagram.

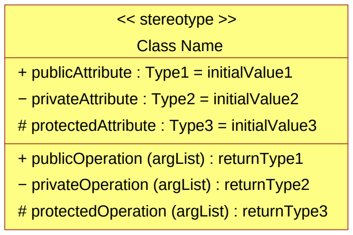

<< stereotype >>
Class Name
+ publicAttribute : Type1 = initialValue1
− privateAttribute : Type2 = initialValue2
# protectedAttribute : Type3 = initialValue3
+ publicOperation (argList) : returnType1
− privateOperation (argList) : returnType2
# protectedOperation (argList) : returnType3

Figure C.2: A general class. A class has a name, sets of attributes and operations; optionally, it can be stereotyped.

small arrowhead to depict the direction along which the association name should be read. On each end of the association line joining the classes there is the multiplicity of the association, which indicates how many instances of one class are related to an instance of the other class. Optionally, a stick arrow may also be used to depict the direction of *navigability*[1]. The notation is represented in Figure C.3, and a simple example is reported in Figure C.4.

Aggregation and composition hierarchies These are *whole/part* relationships. The composition relationship (shown by a black diamond) is a stronger form of *whole/part* relationship than the aggregation relationship (shown by a hollow diamond). The diamond touches the aggregate/composite class box. A composition is a relationship stronger than the aggregation, namely an object of a class that participates as a component in a composition association cannot participate in any other composition association (see Figure C.5).

[1]In an association between two classes, *navigability* is used informally to suggest the direction along which the association name should be read (like in Figure C.4), however at code implementation, the arrow indicates which of the two classes is referenced by the other one (for instance by mean of a *pointer*, or in database design, by a *key* field).

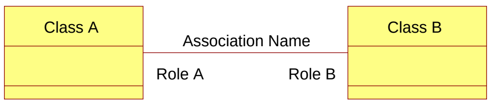


Figure C.3: Association between two classes.

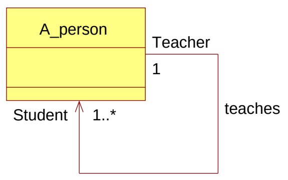


Figure C.4: Association with role multiplicity example. A person with a role of teacher teaches one or more persons with the role of students.

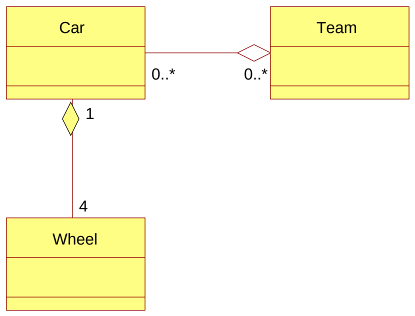


Figure C.5: Composition and aggregation example. A car is made up of four wheels, and a wheel can belong only to one car; also a car can participate in a team of vehicles, but can be shared by more than one team at the same time.

Generalization/specialization hierarchy This is an *is-a* relationship. A generalization is depicted as an arrow joining the subclass (child) to the superclass (parent), with the arrowhead touching the superclass box (see Figure C.6).

Visibility refers to whether an element of the class is visible from outside the class as depicted in Figure C.2. Representing visibility is optional on a class diagram. *Public* visibility, denoted with a $+$ sign, meaning that the element is visible from outside the class; *private* visibility, denoted with a $-$ sign, means that the element is visible only from within the class that defines it and is thus hidden from other classes; *protected* visibility, denoted with a $\sharp$ sign, means that the element is visible from within the class that defines it and within all subclasses derived from it.

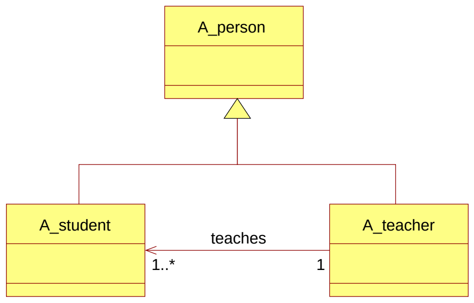


Figure C.6: Generalization example. This diagram represents the same concepts expressed in Figure C.4 in a more expressive way. *A_student* is a particular case of *A_person* taught by an other particular case of *A_person*: *A_teacher*.

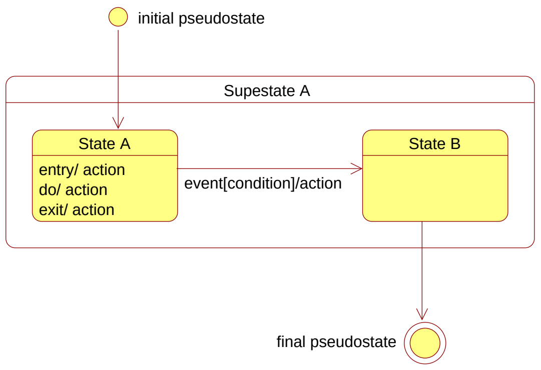


Figure C.7: State-chart diagram notations.

C.3 State-chart diagram

In the UML notation, a state transition diagram is referred to as state-chart diagram, often called simply state-chart. Within UML, states are represented by rounded boxes and transitions are represented by arcs among them. The initial state of a state-chart is represented by an arc originating from a small black circle. Optionally, a final state may be depicted by a small black circle inside a larger white circle. A state-chart may be hierarchically decomposed such that a *superstate* is decomposed into *sub-states*. On the arc representing the state transition, the notation *event [condition] / action* is used. The *event* causes the state transition. The optional *condition* must be true, when the event occurs, for the transition to take place. The optional *action* is performed as a result of a transition. Optionally, associated with a state, there may be

entry actions performed when the state is entered;

activities performed for the duration of the state;

exit actions performed on exit from the state.

The notation is represented in Figure C.7.

Bibliography

[1] L. Nguyen, M. Ogburn, W. Gilbert, K. Kibler, P. Brown, and P. Deal, "Simulator study of stall/post-stall characteristics of a fighter airplane with relaxed longitudinal static stability," NASA Langley Research Center, Hampton, Virginia, Tech. Rep. 1538, 1979.

[2] T. Samad and G. J. Balas, *Software-Enabled Control: Information Technology for Dynamical Systems.* Wiley Interscience – IEEE Press, 2003.

[3] H. Gill and J. Bay, *Software-Enabled Control.* Wiley Interscience – IEEE Press, 2003, ch. The SEC vision, pp. 3–8.

[4] J. L. Paunicka, B. R. Mendel, and D. E. Corman, *Software-Enabled Control.* Wiley Interscience – IEEE Press, 2003, ch. Open Control Platform: a software platform supporting advances in UAV control technology, pp. 38–62.

[5] Object Management Group, "Realtime CORBA joint revised submission," OMG Document orbos, Tech. Rep. 99-02-12, Mar. 1999.

[6] D. Rosu, K. Schwan, S. Yalmanchili, and R. Jha, "On adaptive resource allocation for complex real-time applications," in *Proceedings of the IEEE Real-Time Systems Symposium*, Dec. 1997.

[7] C. D. Gill, D. C. Schmidt, and R. Cytron, "Multi-paradigm scheduling for distributed real-time embedded computing," in *IEEE Proceedings Special Issue on Embedded Systems*, 2002.

[8] B. S. Doerr, T. Venturella, R. Jha, C. D. Gill, and D. C. Schmidt, "Adaptive scheduling for real-time, embedded information systems," in *Proceedings of the 18th IEEE/AIAA Digital Avionics Systems Conference*, Oct. 1999.

[9] M. Agrawal, D. Cofer, and T. Samad, *Software Enabled Control.* Wiley Interscience – IEEE Press, 2003, ch. Real-Time adaptive resource management for multimodel control, pp. 85–103.

[10] M. Agrawal, D. Coffer, and T. Samad, "Real-time adaptive resource management for advanced avionics," *IEEE Control Systems Magazine*, vol. 23, no. 1, pp. 76–88, Feb. 2003.

[11] R. Jha, M. Muhammad, S. Yalamanchili, K. Schwan, D. Rosu, and C. deCastro, "Adaptive resource allocation for embedded parallel applications," in *Proceedings of the 3rd International Conference on High Performance Computing*, Dec. 1996.

[12] D. Musliner, R. Goldman, M. Pelican, and K. Krebsbach, "SA-CIRCA: Self adaptive software for hard real-time environments," *IEEE Intelligent Systems*, vol. 14, no. 4, pp. 23–29, 1999.

[13] W. Koenig, D. Cofer, D. Godbole, and T. Samad, “Active multi-models and software enabled control for unmanned aerial vehicles,” in *Proceedings of the Association of Unmanned Vehicle Systems International*, July 1999.

[14] J. M. Maciejowski, *Predictive Control with Constraints.* Prentice Hall, 2002.

[15] T. Keviczky and G. J. Balas, “Receding horizon control of an F-16 aircraft: a comparative study,” in *European Control Conference*, 2003.

[16] ——, “Software enabled flight control using receding horizon techniques,” in *AIAA Guidance, Navigation, and Control Conference*, 2003.

[17] M. Huzmezan and J. M. Maciejowski, “Reconfiguration and scheduling in flight using quasi-lpv high-fidelity models and *MBPC* control,” in *Proc. of the American Control Conf.*, 1998.

[18] J. L. Paunicka, B. R. Mendel, and D. E. Corman, *Software-Enabled Control.* Wiley Interscience – IEEE Press, 2003, ch. Software Architectures for Real-Time Control.

[19] J. Chen and R. J. Patton, *Robust Model-Based Fault Diagnosis for Dynamic Systems.* Kluwer Academic Publishers, 1999.

[20] J. J. Gertler, *Fault Detection and Diagnosis in Engineering Systems.* Marcel Dekker, New York, 1998.

[21] M. A. Massoumnia, “A geometric approach to the synthesis of failure detection filters,” *IEEE Trans. on Automatic Control*, vol. 31, pp. 839–846, 1986.

[22] A. Emami-Naeini, M. M. Akhter, and S. M. Rock, “Effect of model uncertainty on failure detection: The threshold selector,” *IEEE Trans. on Automatic Control*, vol. 33, pp. 1106–1115, 1988, effectUncertaintyFaultDetection.pdf.

[23] J. J. Gertler, “Survey of model-based failure detection and isolation in complex plants,” *IEEE Control Systems Magazine*, vol. 8, no. 6, pp. 3–11, 1988.

[24] W. H. Chung and J. L. Speyer, “A game theoretic fault detection filter,” *IEEE Trans. on Automatic Control*, vol. 43, pp. 143–161, 1998.

[25] R. Mangoubi, B. Appelby, and J. Farrell, “Robust estimation in fault detection,” in *Proc. of IEEE Conf. on Decision and Control*, 1992, pp. 2317–2322.

[26] H. Niemann and J. Stoustrup, “Filter design for failure detection and isolation in the presence of modeling errors and disturbances,” in *Proc. of IEEE Conf. on Decision and Control*, Dec. 1996, pp. 1155–1160.

[27] A. Marcos, S. Ganguli, and G. Balas, “Application of h-infinity fault detection and isolation to a boeing 747-100/200,” in *AIAA Guidance, Navigation, and Control Conference*, 2002.

[28] J. Stoustrup, H. Niemann, and A. la Cour Harbo, “Optimal threshold functions for fault detection and isolation,” in *Proc. of the American Control Conf.*, 2003.

[29] D.-S. Shim and M. Sznaier, “A caratheodory-fejer approach to simultaneous fault detection and isolation,” in *Proc. of the American Control Conf.*, 2003, pp. 2979–2984.

[30] R. Smith, G. Dullerud, S. Rangan, and K. Poolla, "Model validation for dynamically uncertain systems," *Mathematical Modelling of Systems*, vol. 3, no. 1, pp. 43–58, 1997.

[31] R. B. Davies, "Newmat library," http://www.robertnz.net, 2002.

[32] A. Tanenbaum, *Modern Operating Systems.* Prentice Hall, 2001.

[33] R. David and H. Alla, *Petri Nets and Grafcet: Tools for Modelling Discrete Event Systems.* Prentice Hall, 1992.

[34] M. Fowler and K. Scott, *UML Distilled (3^{th} Ed.).* Addison Wesley Technology Series, 2004.

[35] G. Booch, I. Jacobson, and J. Rumbaugh, *The Unified Modeling Language user guide*, ser. Technology Series. Addison Wesley, 1995.

[36] H. Gomaa, *Designing concurrent, distributed, and real-time applications with UML.* Addison Wesley, 2000.