DEVELOPMENT OF A PILOT-IN-THE-LOOP FLIGHT SIMULATOR USING NASA'S TRANSPORT CLASS MODEL

BY

KASEY ALAN ACKERMAN

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Mechanical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2014

Urbana, Illinois

Advisor:

Professor Naira Hovakimyan

Abstract

This thesis presents the development of an immersive flight simulator at the University of Illinois at Urbana-Champaign using NASA's Transport Class Model. Flight simulators are a common and useful tool for control system design as well as verification and validation, and are used extensively throughout the aviation industry to evaluate system performance. The simulation environment at the Illinois Simulator Laboratory uses a modular architecture composed of a Simulink dynamics model, X-Plane visualization, a Frasca 142 cockpit, and a customizable digital cockpit instrument panel, with communication between elements conducted over a local area network. The simulator is intended to streamline the design and evaluation process for a prototype flight control architecture integrating flight envelope protection and loss-of-control prevention systems with a robust adaptive control law, and the availability of a local simulator for control system development will serve to expedite the eventual transfer of this technology to alternate facilities.

The thesis documents the creation of a Simulink library that enables efficient switching of simulation modes and controller configurations, so that a control design can be rapidly evaluated and tested using batch simulations and flight simulator testing to obtain pilot performance feedback. Also documented here is the development of automated tools to improve the performance of the simulation during flight testing. Additionally, interfaces for improving the utility and usability of the simulator during testing are presented, as is the design of an environmental disturbance model to provide additional functionality for upcoming flight tests. Finally, preliminary results from simulator flight testing at the Illinois Simulator Laboratory are discussed.

To my parents, Jay and Susan Ackerman. It is only because of your love and support that I have had the opportunity to pursue my education, and for that I will be forever grateful.

Acknowledgments

First and foremost I would like to express my gratitude to my advisor, Prof. Naira Hovakimyan, for welcoming me into her research group and for the wonderful academic and professional opportunities that I have had as a result. I am deeply grateful for her guidance and support over the course of my Master's studies.

I would also like to thank my colleagues and friends from the ACRL for sharing their knowledge and insight with me, and also for welcoming me into their group. In particular, I am grateful to Enric Xargay, Niko Tekles, and June Chongvisal for their contributions to the development of the flight simulator software and the piloted testing at the Illinois Simulator Laboratory (ISL).

A special thanks goes to Ron Carbonari at the ISL, who has been instrumental in the development of the simulator environment and of the pilot interface, and is always a pleasure to work with. I would like to express my gratitude also to Don Talleur of the UIUC Institute of Aviation, whose insight and feedback during flight testing have been invaluable to the development of the software tools for the simulator and of the flight control system.

My thanks go out to all the great people at NASA Langley Research Center for their support and contributions to this research, and also for the amazing and rewarding opportunity to work at LaRC over the last two summers. Specifically I would like to thank Dr. Irene Gregory, Dr. Dave Cox, and Dr. Christine Belcastro, whose support and advice have made this research possible, as well as Paul Rothhaar, Carey Buttrill, and all the people at LaRC whom I have had the pleasure of working with.

I am truly grateful for the encouragement and support from many close friends who have helped me get to this point, both academically and personally. Finally, I would like to express my heartfelt thanks to my family, who have been a constant source of unconditional love and support over the years. I can't thank you enough!

This work was supported in part by the National Aeronautics and Space Administration.

Table of Contents

Abbreviations and Terminology

ACRL	Advanced Controls Research Laboratory
FCS	Flight Control System
FEP	Flight Envelope Protection
iReCoVeR	Integrated Reconfigurable Controller for Vehicle Resilience
ISL	Illinois Simulator Laboratory
LoC	Loss-of-Control
mex	Matlab executable
NASA	National Aeronautics and Space Administration
PITL	Pilot-in-the-Loop
RTW	Real-Time Workshop
TCM	Transport Class Model
UDP	User Datagram Protocol
UIUC	University of Illinois at Urbana-Champaign
V&V	Verification and Validation

Chapter 1

Introduction

As automated systems become more complex, conducting a comprehensive performance evaluation becomes very difficult. When complex automation is intended for use in a cyber-physical human system, e.g., dependent on the input or reaction of a human operator, the task of performance evaluation becomes even more challenging. Inputs to such systems are typically more complicated than what can be readily generated in a normal simulation environment, and these inputs are usually dependent on the state of the simulated system. This complexity poses an issue for evaluating the behavior of a system at all points in its operational space.

Verification and Validation (V&V), used to ensure that the system is designed correctly and functioning in the intended manner across its operational space, is a crucial step in the development of any safety-critical system, and many excellent software solutions exist for system V&V. A complement to these software tools is in-the-loop testing, a physical V&V technique where the physical object the system is designed to interact with is included in the feedback loop. When the physical object is a human, the process becomes human-in-the-loop testing, or pilot-in-the-loop (PITL) in the specific case of an aircraft system intended for use by a pilot. Human-in-the-loop testing is especially useful for evaluating subjective aspects of the system, such as the "feel" of the system performance to the human operator, that may not be readily determined through software methods. Since there will be human-machine interaction within the system, subjective criterion such as performance "feel" become important design considerations that are difficult to evaluate without the use of a simulator.

Simulator testing is common practice in many fields for the design of cyber-physical human systems, and is mandatory in many safety critical systems. Aviation is an excellent example of one such field; simulators are used both for software and hardware design testing, as well as pilot training and performance evaluation. The primary benefit of using a simulation for testing purposes is that it allows for system evaluation in a safe environment. In addition, simulation testing usually provides data collection capabilities beyond what would be available in the real system, allowing for advanced systems analysis.

Simulation testing is also useful in the process of flight control design, where control law iterations can be tested by a rated pilot whose feedback can be used to adjust the performance or functionality of the controller.

Typically an aircraft control system must meet a stringent set of safety and performance requirements, although many design decisions are also made using more subjective performance criterion. The ability to perform simulation testing and obtain pilot feedback at multiple stages of control law development can help streamline the design process.

To this end, the flight simulator at the Beckman Institute's Illinois Simulator Laboratory (ISL) [1] at the University of Illinois at Urbana-Champaign (UIUC) has been selected to develop a simulation environment which may be used to test and evaluate prototype flight control technologies. The Advanced Controls Research Laboratory (ACRL) at UIUC is working to develop the Integrated Reconfigurable Controller for Vehicle Resilience (iReCoVeR) flight control architecture, which combines a baseline gain-scheduled control and stability augmentation system with Flight Envelope Protection (FEP) and Loss-of-Control (LoC) prediction and prevention systems, fault detection and isolation, and a robust adaptive flight controller. This system is being developed under the National Aeronautics and Space Administration (NASA) Vehicle System Safety Technologies program, with the goal of preventing LoC incidents and accidents in transport class aircraft. This control architecture will require extensive software and PITL simulation prior to flight tests using a testbed flight vehicle.

Initial testing and verification of the control design is streamlined through the use of a local simulator that can run aircraft dynamics and controller design models using the same software environment in which the models are typically developed. The use of a single software environment provides portability between the design and test environments to enable rapid turnaround of design iterations and prompt feedback on performance of the system. Further, the use of a local simulator enables debugging of any software or hardware issues that could arise when transferring the simulation to a real-time environment for V&V work performed by research pilots, saving time and money during later development stages. Since the iReCoVeR architecture has been designed under a NASA program, it was envisioned that this technology would be transferred to NASA for flight testing in the future. It is therefore advantageous to have a flexible local simulator platform for preliminary design and V&V work prior to this technology transfer.

In order to provide such a platform for use with the iReCoVeR control architecture, a modular simulator architecture has been created at the ISL. The simulation environment is a networked system composed of a Frasca 142 cockpit, a projection display using X-Plane graphics software, a custom-designed digital cockpit instrument panel, and an aircraft dynamics model implemented in Matlab [2] and Simulink [3]. The simulation environment currently used has been tailored specifically to the iReCoVeR control system development for use with NASA's Transport Class Model (TCM) Simulink dynamics model, although the tools developed for the simulator were designed to be as portable as possible for use with future control designs and alternative dynamics models. The modularity of the simulator design allows for further reconfiguration of the simulation environment as needed for future testing.

This thesis documents the contributions made to the flight simulator environment at ISL, including the integration of the TCM dynamics with communication tools developed at the ACRL, and the design of a user-friendly interface for operating the simulation and recording data. Also detailed here are the simulation performance gains obtained by using compiled versions of Simulink subsystems, and the development of a environmental disturbance model that will enhance the set of available simulation scenarios for upcoming PITL testing. This thesis is also intended to be a reference for future use of the simulation environment and the tools developed and described here, as well as for adaptation of these tools for alternative applications. The remainder of this work is divided into the following chapters:

- Chapter 2 provides an overview of the ISL simulator architecture and the TCM Simulink model. The structure of the hardware and software that make up the simulation environment is described, as well as the internal structure and basic operation of the TCM simulation. A brief guide for operating the simulation is also provided.

- Chapter 3 discusses the network communication tools used to integrate the TCM dynamics into the flight simulator environment and modifications needed for applications to new system configurations. The development of the Input library for efficiently switching between simulation modes and configurations is detailed.

- Chapter 4 describes the need to develop a method for improving simulation performance during flight testing, and documents the automated compilation procedures created to substitute computationally heavy Simulink subsystems with binary executable functions, substantially enhancing simulation performance while preserving the original functionality of the model and its supporting utilities.

- Chapter 5 outlines the development and usage procedure for the operator interface of the TCM simulation. A Graphical User Interface (GUI) is developed to facilitate the initialization of the Simulink model in a user-specified configuration, and substantial modifications are made to the data acquisition procedure to reduce postprocessing workload and analysis time.

- Chapter 6 describes the flight testing conducted at ISL and the preliminary results obtained. Additionally, an environmental disturbance model is developed that uses additive methods to produce a wind disturbance of a user specified profile. The mathematical model used for generating disturbances is presented as well as its implementation in the Simulink model.

Chapter 2

Simulator System Overview

The development process for the iReCoVeR Flight Control System (FCS) can be significantly enhanced through the use of a local simulator platform where FCS design changes can be tested. The simulator allows for PITL testing which can serve not only as a V&V process, but can also be used to improve the control system design. If FCS design iterations can be transferred to the simulator quickly and without the time-consuming process of porting the software to a different hardware implementation, then the testing and development process can be greatly expedited.

The goal of this thesis is to develop a simulation environment that brings together several preexisting elements under a single architecture that facilitates the design and testing of flight control systems and pilot interfaces. Additionally, the tools developed to integrate the simulation elements should be generalizable so that they may be applied to enable similar functionality between different simulation elements. The primary benefit of this architecture is that it allows design and development work to occur in its native interface, then be immediately tested and verified using batch simulations, desktop flight simulation, or full-scale PITL testing without the need to implement the design in a different environment. This streamlining of the design and testing process can yield a significant benefit in terms of reduced design time and greater flexibility in evaluating design changes.

The FCS design work in this case is development of a flight controller with FEP and LoC prevention systems as part of the iReCoVeR control architecture. The FCS is designed for and implemented within the TCM Simulink model developed at NASA's Langley Research Center [4]. Since the TCM and its supporting tools are implemented in Matlab and Simulink, and also because the tools created for the iReCoVeR architecture are to be transferred to NASA at a future date, it is required that the dynamic model used to drive the simulation be implemented in Matlab and Simulink. The implementation of the dynamics model and FCS in Simulink is the primary motivation for the development of the communication tools and software utilities presented in this thesis.

Similarly, the PITL simulator hardware was also predetermined for this project. The simulator at ISL is comprised of a Frasca 142 [5] cockpit and three projectors providing a full 180° view with graphics driven by X-Plane 9 [6]. User Datagram Protocol (UDP) connections over a local area network are used to communicate between the computers which drive the aircraft dynamics, the graphical representation, and the cockpit controls. UDP communication tools must then be incorporated into the TCM simulation in order to be compatible with the existing infrastructure.

Another requirement on the development of the simulation environment is that the dynamics model must be capable of performing design work and batch simulation in an office or laboratory setting, and also capable of piloted testing at the ISL flight simulator facility, driving the aircraft dynamics and handling the data acquisition. It is with these requirements in mind that the simulation tools described in this thesis were developed, as well as with the intention of transferring the tools developed here to different dynamics models and simulator elements.

Previous work at the ACRL at UIUC has resulted in the development of the CNCT library [7], a set of tools to enable UDP communication between a Simulink model and the simulator hardware at ISL. The library contains blocks that can be placed into the Simulink model to send and receive via UDP the signals required to interact with the rest of the simulator. The CNCT library will be utilized to facilitate the communication with the TCM simulation, while newly developed tools will integrate the TCM simulink model into the simulator for design and testing. In the following sections the simulink model of the TCM and the ISL simulator hardware will be discussed, while new developments will be discussed in Chapters 3 through 6.

2.1 TCM Simulation Model

The Simulink model of the TCM aircraft was developed by NASA for research and development of flight control laws and other technologies to prevent LoC incidents and manage aircraft upset conditions that could potentially contribute to an LoC event [8]. The TCM Simulink model is divided into subsystems, with each subsystem represented by a "block," illustrated in Figure 2.1. Each block may have a set of inputs and outputs, with functions and operations performed on internal signals within the block. Blocks representing subsystems may be nested, and signals (variables passed between blocks and operators) are represented as lines with arrows showing the direction of signal dependency.

The internal structure of the white GTM_FullScale block in the center of Figure 2.1 is shown in Figure 2.2. This block contains the Simulink implementation of the mathematical model which governs all of the aircraft dynamics. Contained within the dynamics subsystem are blocks in which the aircraft equations of motion, actuator dynamics, engine dynamics, aerodynamics model, sensor models, and aircraft system models are

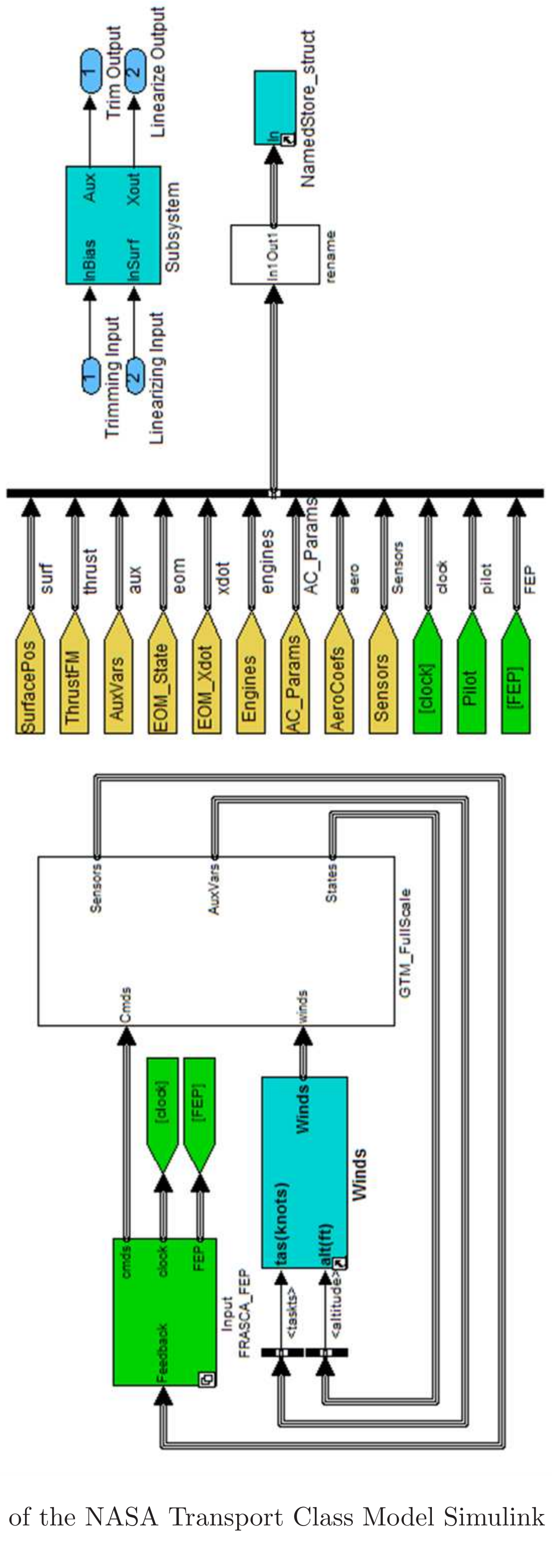


Figure 2.1: Top level of the NASA Transport Class Model Simulink dynamics model.

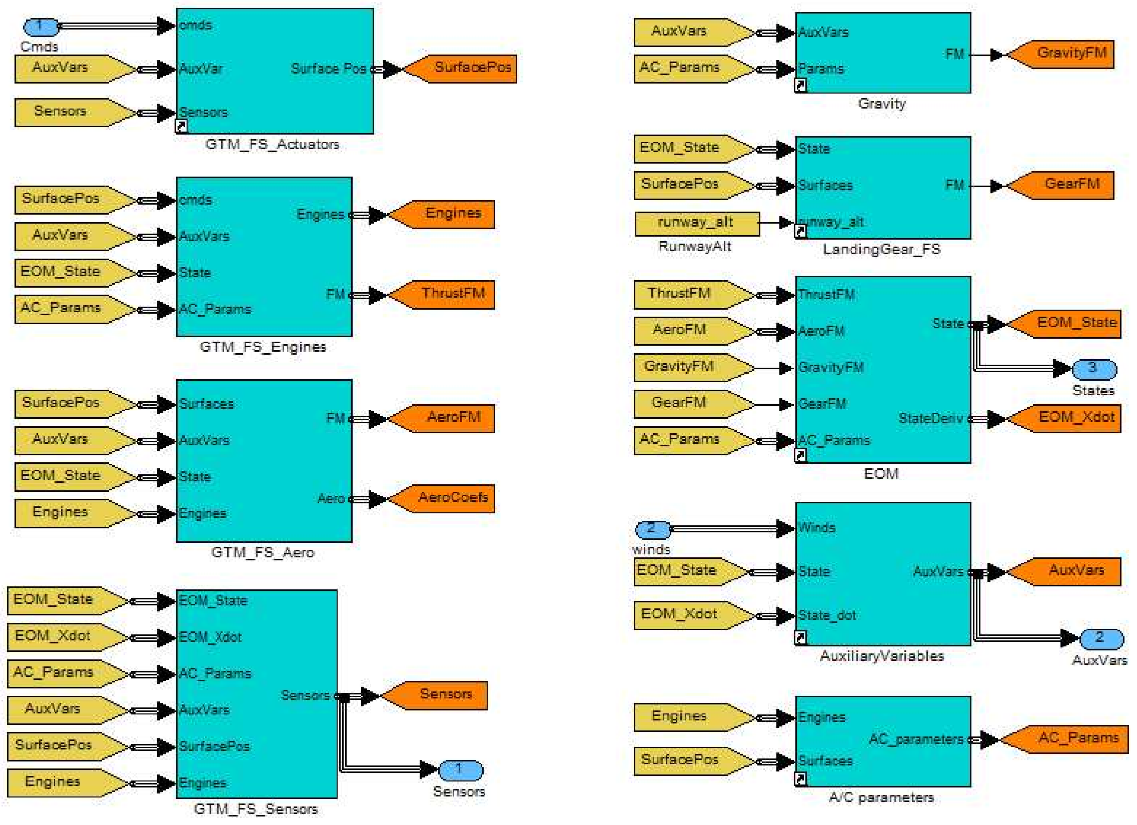


Figure 2.2: Dynamics portion of the NASA TCM Simulink model.

implemented. At the top left of Figure 2.2, the GTM_FS_Actuators block takes the control surface commands from the green Input block of Figure 2.1 and calculates the actual positions of the control surfaces using the mathematical actuator dynamics model. The surface position information is then available to other subsystems through the use of a global signal routing tag, shown in orange. Forces and moments generated from the engines, aerodynamics, gravity, and landing gear are calculated through mathematical models contained within the GTM_FS_Engines, GTM_FS_Aero, Gravity, and LandingGear_FS blocks, respectively. The forces and moments are routed to the equations of motion in the EOM block where along with the aircraft geometric and inertial information from the A/C parameters block, the aircraft accelerations and position derivatives are integrated to calculate position, angular rates, and velocities. Aerodynamic variables such as angle-of-attack, angle-of-sideslip, true and estimated airspeed, dynamic pressure, and others are calculated in the AuxiliaryVariables block, and the sensor dynamics used to provide feedback to the FCS and cockpit instruments are implemented in GTM_FS_Sensors. Additional detail pertaining to the functionality of the TCM dynamics model can be found in [8].

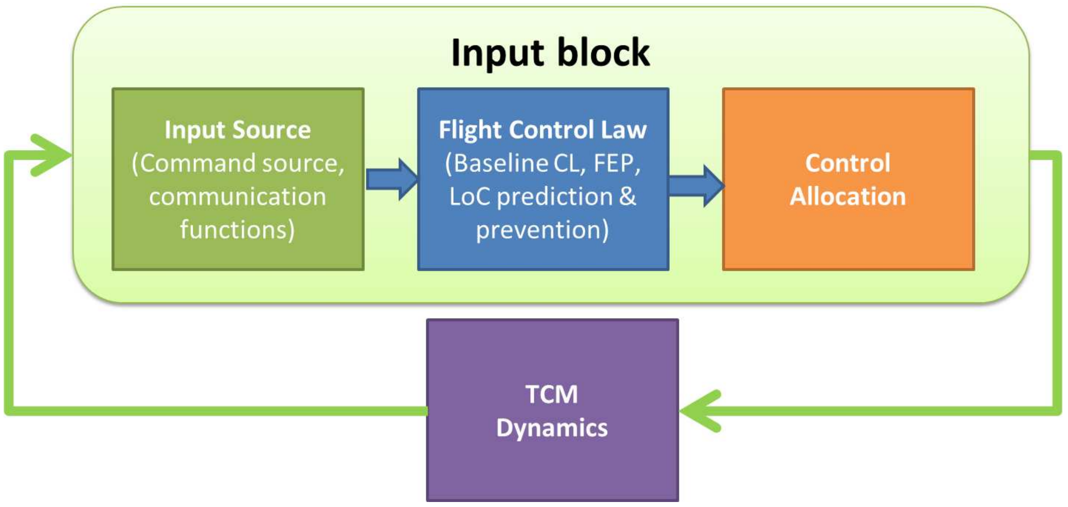

Input block
Input Source
(Command source, communication functions)
Flight Control Law
(Baseline CL, FEP, LoC prediction & prevention)
Control Allocation
TCM Dynamics

Figure 2.3: Schematic structure of the TCM simulation Input block.

Control surface command inputs to the aircraft dynamics block are generated in the Input block, which has the basic internal structure shown schematically in Figure 2.3. The Input block is where the core of the development work for the TCM is conducted and where the communication and flight control software tools are implemented. Depending on the simulation mode, the input source portion of the Input block either generates preprogrammed commands or receives pilot inputs, which are then sent to the FCS. The current implementation of the FCS consists of a C*U longitudinal control law and roll rate and angle-of-sideslip lateral-directional control law, augmented with FEP and LoC prediction and prevention systems [9] [10]. Pilot commands are modified by the FCS and transformed to control surface commands through the control allocation. Aircraft feedback signals are then utilized to update the X-Plane visualization and cockpit instrument panel through UDP communication software implemented in the command source system.

To ensure that the simulation runs in real-time during flight simulation and desktop piloted simulation, a pacer block provided by NASA is used in the command source system to enforce that the simulation pace matches the pace of the computer system clock. Alternatively, the Simulation Pace block from Simulink's Aerospace Blockset can be used, although the accuracy observed using the NASA-provided pacer is better by several orders of magnitude. The pacer block is not used in batch simulation mode, as in this mode it is desirable for the simulation to run as fast as possible. It is worth noting that the pacer block can only delay the pace of the simulation. If the nominal simulation pace is slower than real-time, then additional measures are needed to speed up the simulation. Techniques to prevent the simulation from lagging real-time are especially important when conducting simulations using noisy signals, turbulent environmental conditions, or when computationally heavy control techniques are used. During flight testing at ISL it was discovered that the simulation was running slower that real-time during large environmental disturbances, which resulted in the need to compile portions of the simulation as discussed in Chapter 4.

Finally, simulation signals are collected and routed into the NamedStore_struct block on the right of Figure 2.1 for data acquisition and recording, detailed in Section 5.1.2. The final block in the TCM block diagram is the Subsystem block, which is utilized by the support functions to trim and linearize the TCM dynamics.

2.1.1 Running the Simulation

The software for the TCM simulation is divided into two main directories. The *libs* directory, where TCM subsystems are stored and linked from within the Simulink model, and the *gtm_design* directory, containing the simulation model and supporting utilities. At the top level of the *gtm_design* folder is the *gtm_designFS* TCM simulation and the setup functions used to initialize the model. Running the *setupTCM.m* script will start the setup GUI, where the user can specify the simulation mode and initial conditions. Clicking the *Setup Simulation* button calls the *TCM_init_fnc* script to add the file paths to the supporting functions and trim the model for simulation. The simulation can then be controlled using the Simulink *Play*, *Pause*, and *Stop* commands.

Supporting files for the TCM simulation are located in the *matfiles*, *mdls*, *mexfiles*, and *mfiles* directories, organized by file type. Flight test data is saved to the *data* directory, and signal bus definitions are saved to the *busdef* folder. Source code for the executable functions and communication tools are included in the *src* directory. Additional utilities are included at the same file level as the simulation model for compiling the aircraft dynamics and FCS, and will be discussed in detail in Section 4.3.

2.2 Flight Simulator Hardware

The flight simulator at ISL uses a modular architecture to provide a flexible simulation environment where components operate independently and communicate over a local area network using UDP. The schematic architecture of the simulator shown in Figure 2.4 provides a platform that can be applied to a variety of simulator hardware and software, and the tools developed here can be transferred directly to other flight simulators, visualization software, or aircraft dynamics models.

The Frasca cockpit used in the simulator is equipped with a yoke-wheel and pedals for aircraft attitude control as well as controls for throttle, spoiler deployment, flap deployment, and trim settings, and additional features are also available for use as needed for different aspects of the simulation. Pilot commands are read through an analog-to-digital card connected to a computer at the back of the cockpit and sent via UDP over

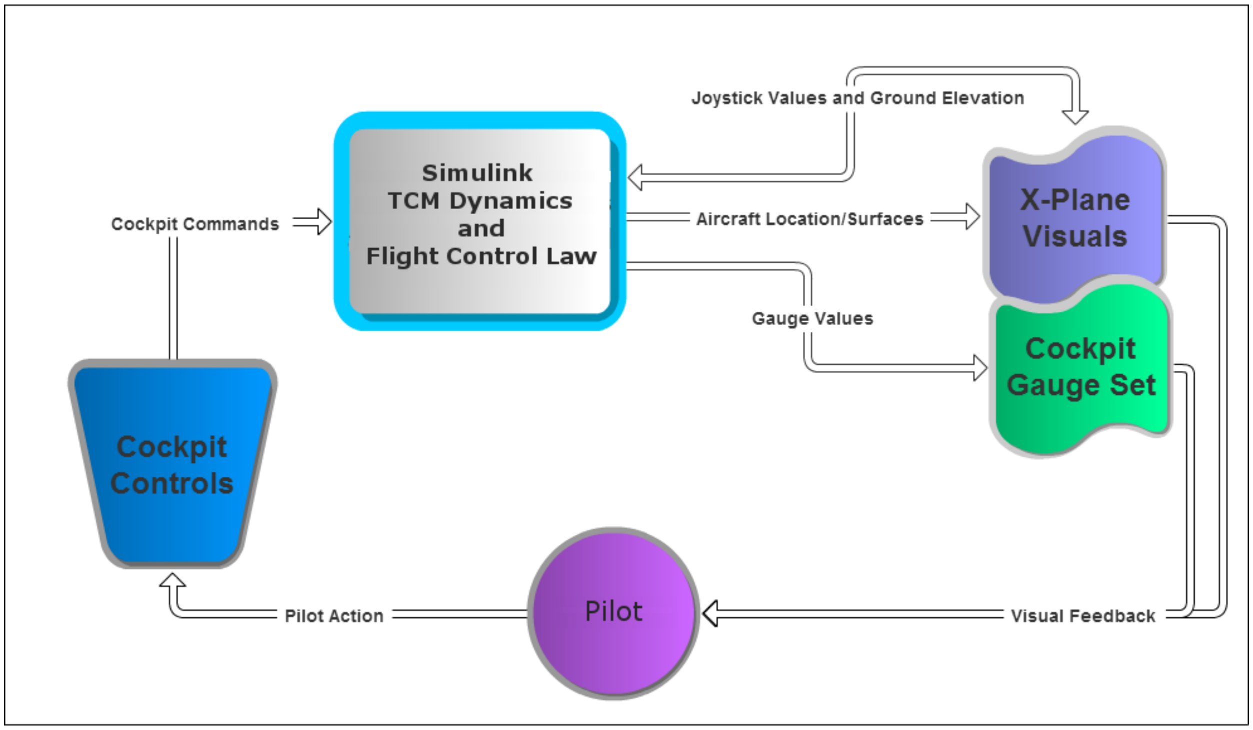


Figure 2.4: Schematic representation of the simulation architecture at the Illinois Simulator Laboratory.

the local area network to be used in the dynamics model driving the simulation.

The graphical representation is driven by four computers, all running X-Plane 9. There are three computers which drive the displays; one for each projector, and there is a main computer which runs a master copy of X-Plane. It is on this main computer that the aircraft dynamics built into X-Plane are normally used to provide the simulation experience. The aircraft position and orientation information is sent via UDP to the other three computers which then update the projected displays as new information is received. Since the simulator is also used for simulation and demonstration purposes, which do not use the TCM or any Matlab/Simulink dynamics model, it is desired to retain the architecture of the flight simulator so as not to interfere with the original functionality.

To make use of the existing X-Plane software at the simulator facility, a plugin provided by NASA with the TCM Simulink model is used to disable X-Plane's internal solver, forcing every element of the X-Plane dynamics to remain constant. The plugin is then used to overwrite the reference variables, or "datarefs," used by X-Plane to store the aircraft dynamic information such as position and orientation of the aircraft and the aircraft control surface positions. By writing new values to the datarefs, it is possible to control the position of the aircraft in space using information from an external source. The aircraft dynamics information is then produced by running the TCM dynamics model, with the aircraft and control surface positions sent

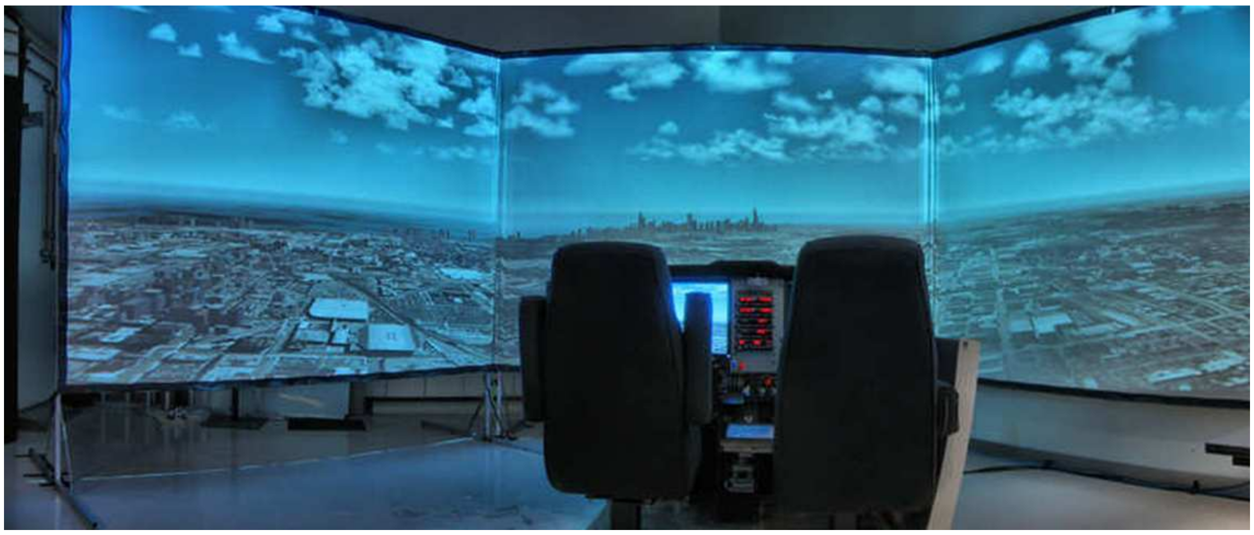


Figure 2.5: The flight simulator cockpit and visualization at the Illinois Simulator Laboratory.

via UDP to all four computers which drive the simulator displays. The plugin can be enabled or disabled to alternate between the use of X-Plane's dynamics models or an external source, minimizing the intrusiveness of this interface. When the simulation is driven from the TCM dynamics, pilot commands are routed to the computer running the Simulink model instead of to the main X-Plane computer. However, pilot command information can be sent to both computers at the same time to eliminate the need for switching between pilot command destinations.

The display panels in the simulator cockpit are used to present the pilot with a digital representation of instruments similar to those in a standard aircraft, along with additional information about the aircraft state including aerodynamic data, control surface deflections, and aircraft angular rates. The instrument panel was developed specifically for testing with the TCM model by members of the ACRL and ISL. Standard round "steam-gauge" style displays in the upper left and center portion of Figure 2.6 inform the pilot of the aircraft dynamic state. Clockwise from the upper left corner, these gauges display the airspeed, attitude, altitude, vertical speed, heading, and turn information respectively. The numeric displays in the upper right corner of Figure 2.6 give the pilot additional information about the aircraft dynamics and aerodynamic data. Many of these extra measurements are not available on real aircraft, but this information is useful for verification and evaluation of control system performance. The extra numeric displays can also be disabled during flight testing to avoid providing the pilot with normally unavailable information and artificially altering the results of a flight test session. The bar gauges in the lower right corner indicate the engine pressure ratio, throttle position, and spoiler and flaps setting, while the gear and brake indicators in the lower center alert the pilot when the gear or brake systems are active. The cockpit display is set up on a switch so that when the

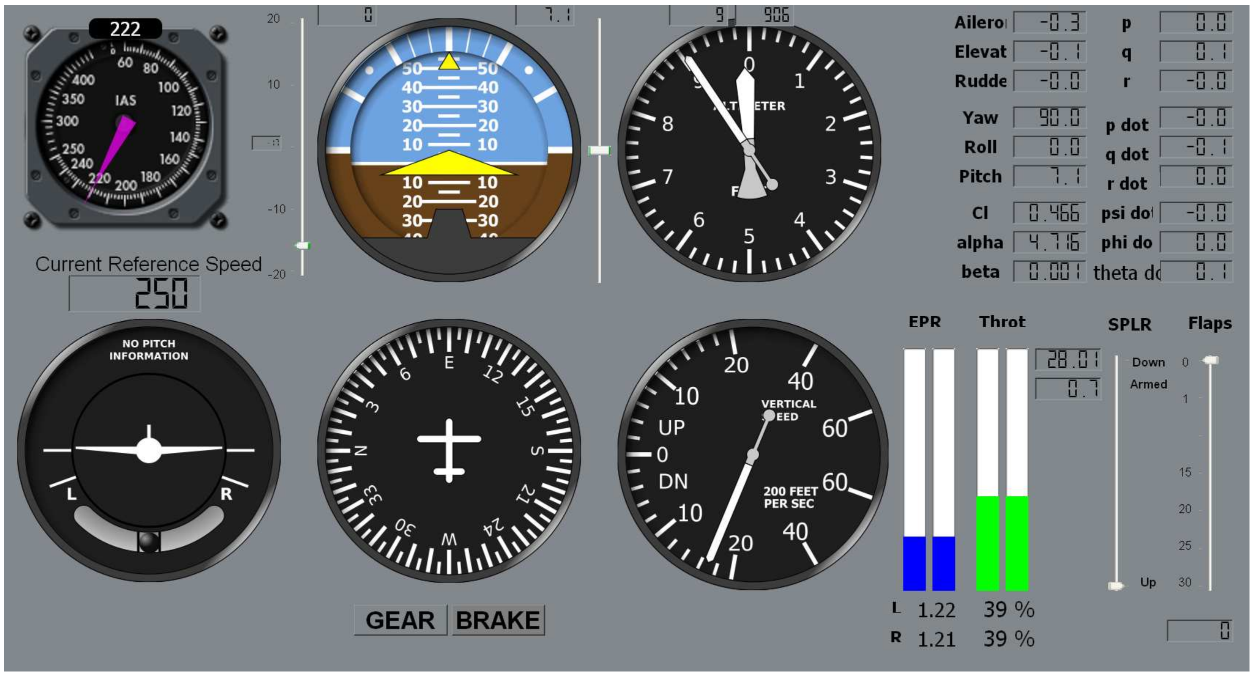


Figure 2.6: Simulator cockpit display instrument panel.

X-Plane dynamics are used, the built-in X-Plane cockpit can be displayed instead of the custom instrument panel. It is worth noting that this version of the instrument panel will soon be superseded by a newer version, which will incorporate a primary flight display augmented with information from the FCS. The new version will be more representative of the glass cockpit environments found on modern airliners, and is thus more applicable for research purposes.

Chapter 3

Integration of the TCM Simulation at ISL

In order to use the TCM dynamics to drive the simulator at ISL, the communication functions must be integrated into the TCM Simulink model. The communication functions send the aircraft dynamics information to the visualization software, receive the pilot inputs, and send the data to the instrument panel for feedback to the pilot. The CNCT Simulink library was developed to enable portability of the communication functions between Simulink models, however details of the CNCT library and the communication utilities will not be discussed here as they are not part of this work, but may be found in [7].

3.1 Implementation of the Communication Tools

The communication functions are implemented in the Input block of Figure 2.1. The communication functions must be external to the GTM_FullScale dynamics block and within the Input block so that the communication functionality may be altered with selection of different simulation modes. For controls development and simulation purposes, there are three modes in which the simulation can be operated. Batch simulation mode is a mode where the inputs to the controller or plant are predetermined and usually automated, and often requires multiple simulations to test or validate the result of a controller design change. The simulation should run quickly, ideally significantly faster than real-time, and space should be provided in the model for modification of the input commands. In desktop piloted simulation mode, the aircraft model can be flown using the X-Plane visualization software and a joystick without requiring the use of a full-blown simulator. This mode is useful for evaluating and visualizing the behavior of the aircraft, and also for inputting complex commands or commands that are more easily executed using visual cues. Flight simulation mode is used to provide the full flight simulator functionality when performing piloted testing. The use of simulation modes is discussed in more detail in Section 3.2. Flight simulation mode is the only mode which requires the use of the communication tools from the CNCT library, since desktop piloted simulation mode uses preexisting communication functions provided by NASA and batch simulations do not require any communication functionality. Thus only the communication implementation for the flight simulation mode is discussed here.

The portion of the Input block which contains the communication functions is the FRASCA_IN subsystem, shown in Figure 3.1. At the far left of the figure, the Frasca_To_Sim block is a C-Matlab executable (mex) s-function block which receives the UDP pilot command information sent from the Frasca cockpit. A C-mex function is a Matlab executable binary function written in the C programming language. The use of C-mex functions allows the use of the Winsock C libraries to facilitate UDP communication. The UDP information is read in as a character string which is then parsed in the C-mex function into an array of numeric values corresponding to the pilot commands. The array is then passed to the FRASCA_Baseline_FEP block which performs basic precalculations to process the pilot inputs into commands which can be sent to the FCS. The processed pilot commands are output from the block and collected into a bus structure which is then sent to the FCS via the FRASCA_CMD and PLT_CMD output blocks. The orange XPlane_POS_SURFS block in the center right of Figure 3.1 uses the global signal routing tags in the TCM dynamics block to select and sort the aircraft position and orientation information and the control surface position information that is routed to the large orange Sim_to_Xlink block. The Sim_to_Xlink block is a C-mex function used to send information to the Xlink plugin and thus to update the X-Plane visual representation.

Another important block in the FRASCA_IN subsystem is the Gauge_Feed block, which uses the global signal routing tags to collect information to be passed to the instrument panel display in the Frasca cockpit. The data are collected into a vector and routed to the SimToGauges C-mex s-function block, which then sends the information to the instrument panel via UDP. Some auxiliary functions have also been included within the FRASCA_IN subsystem to provide additional functionality to the simulation. The orange Graphical_GND_Height block at the top center of Figure 3.1 receives via UDP the altitude of the ground at the current aircraft location, which comes from X-Plane's navigation database. The ground height information can be used to automatically deploy landing gear in the event of incipient ground contact in the visualization through the FRASCA_ALT_Triggered_Gear block, or can be used to perform takeoffs and landings without the need to preprogram the runway height into the simulation.

3.1.1 Modifying the Communication Tools

There are several foreseeable situations for which a user would be required to alter the number or type of signals sent from or received by the simulation. For example, the pilot inputs from the cockpit may change as new functionality is added, or if new information is needed for display to the pilot through the cockpit display. When this occurs, it is then necessary to alter the structure and content of the communication functions. In order to change the signals sent from Simulink to either the X-Plane Xlink plugin or to the

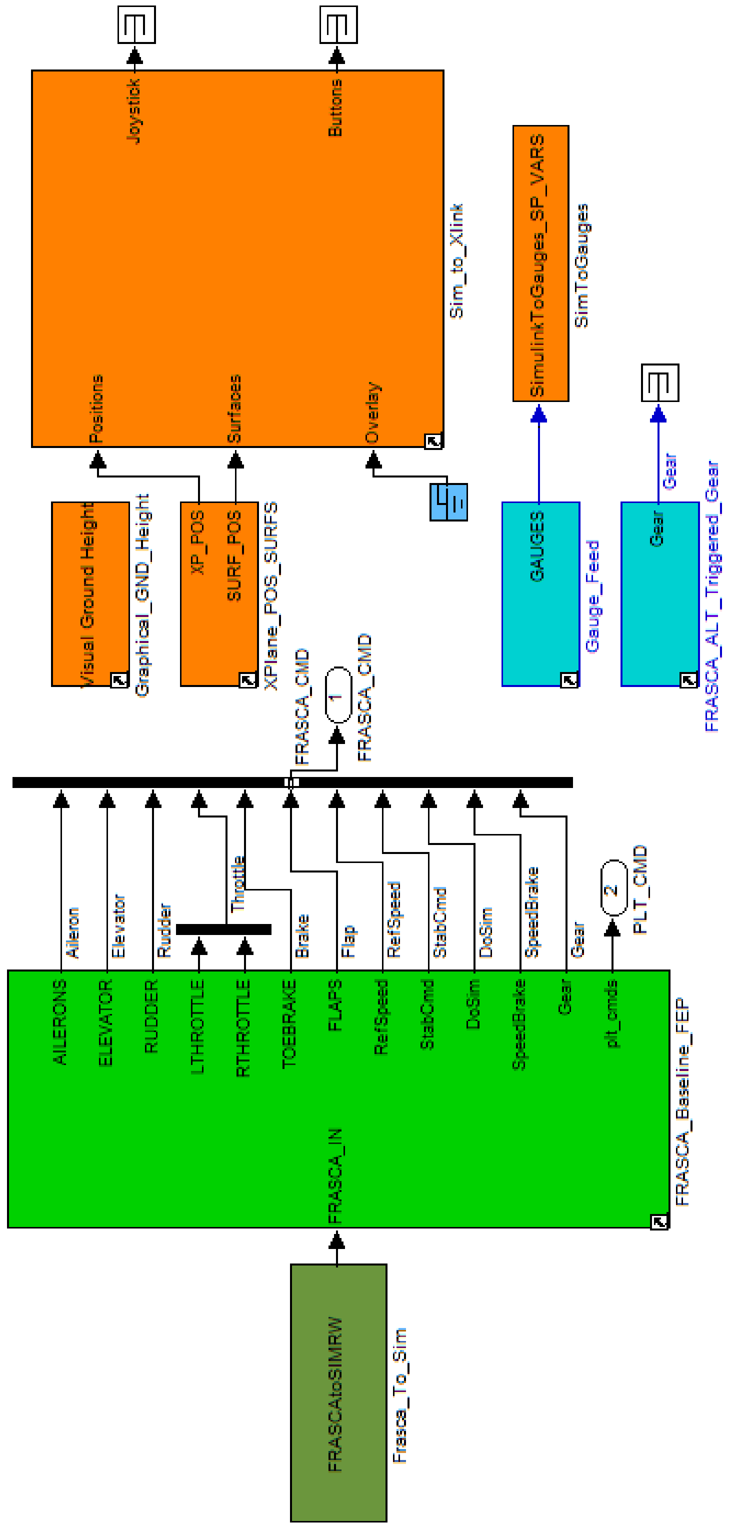


Figure 3.1: Simulink implementation of the CNCT library communication tools in the Input block of the TCM simulation.

cockpit display, the CNCT library blocks, the corresponding s-functions, and the data packet definition header files must all be modified in order to implement the changes.

To alter the data sent to the cockpit instrument panel, the first step is to augment the data packet header file *dataPacketGauges.h* with the new information. This file defines the *Gauge_dataPacket* structure, and the structure fields are the variables to be sent to the instruments. For example, to add the variables *myvar1* and *myvar2* to the structure with *myvar1* defined as a double and *myvar2* defined as a float, adding lines 3 and 4 in Listing 3.1 would define the variables as valid fields in the data packet structure. The second step following the variable definition, it is necessary to modify the corresponding C-mex s-function by editing the C++ file of the same name. Continuing with the example of the instrument panel information, the appropriate s-function is *SimulinkToGauges_SP_VARS.cpp*.

Listing 3.1: Example of syntax for adding variables to a data packet structure.

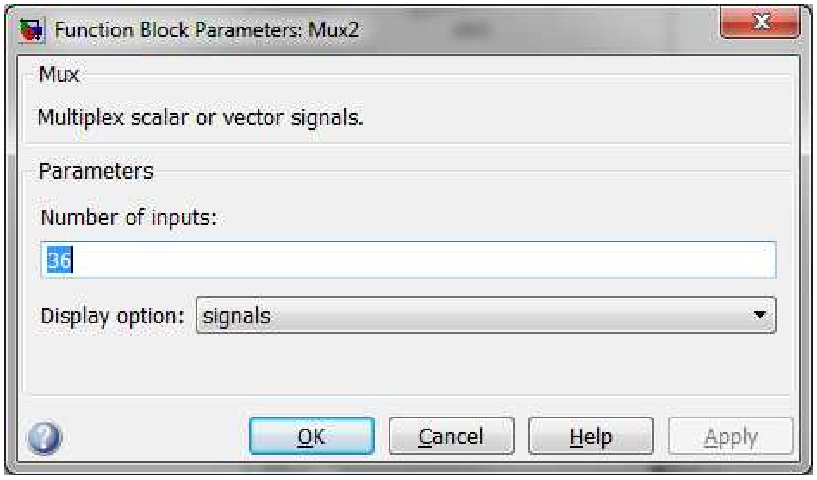
```
1 struct Gauge_dataPacket {
       ...
3      double myvar1; // Comment describing myvar1
       float  myvar2; // Comment describing myvar2
5      ...
  }
```

Now, if it is necessary just to add signals to the existing input vector, the line *ssSetInputPortWidth(S, 0, 36);* in the *static void mdlInitializeSizes(SimStruct *S)* function should be set to reflect the new number of input elements. To add the two variables *myvar2* and *myvar2*, the line should be changed to *ssSetInputPortWidth(S, 0, 38);*. The second input to the *ssSetInputPortWidth(·)* function is the index of the input, where the inputs are 0-indexed. To assign values to the new variables, adding lines 6 and 7 of Listing 3.2 to the *static void mdlOutputs(SimStruct *S, int_T tid)* function will assign the 37th and 38th element of the first s-function input to the the new structure variables, recalling that C++ is a 0-indexed language.

Listing 3.2: Example of syntax for assigning values to new variables in a data packet structure.

```
       ...
2      const real_T *u1 = ssGetInputPortRealSignal(S, 0);
       ...
4      Gauge_dataPacket data;
       ...
6      data.myvar1 = (double)u1(36);
       data.myvar2 = (float) u1(37);
8      ...
  }
```

Once the variables have been declared and assigned in the s-function, the Simulink model must then be modified to append the desired signals to the s-function input vector. Additional signals must be added to the vector multiplex (mux) block inside the Gauge_Feed block of Figure 3.1. Double-clicking the mux block


Function Block Parameters: Mux2

Mux

Multiplex scalar or vector signals.

Parameters

Number of inputs:

36

Display option: signals

Figure 3.2: Simulink dialog for modifying the number of input signals to the communication s-function.

brings up a dialog where the number of signals is specified, shown in Figure 3.2. Changing the number of inputs here to 38 will allow the addition of the two new variables. The global signal routing tags and bus selector in the Gauge_Feed block can be used to extract the desired signals and route them to the new inputs. To complete the process, the modified C-mex s-function needs to be recompiled. Running the *buildmex* script from the Matlab command prompt within the *gtm_designFS* directory will recompile all of the s-functions needed by the TCM simulation and copy the executables to the appropriate file path.

It is also possible to add a second input port to the s-function instead of concatenating additional signals to an existing port. In this case the additional port must be declared in the s-function and initialized appropriately. Declaration of a second input port is done in the C-mex s-function *SimulinkToXPlane2.cpp*, and is also briefly described in [7]. Once the C-mex function has been recompiled, the additional input port will be visible in the Simulink model for signal routing.

Alteration of the data sent to the X-Plane visualization is accomplished through the same process as for the cockpit display information. The data packet definition should be modified in the *dataPacket.h* header file, and the new variables should be assigned in the C-mex function *SimulinkToXPlane2.cpp*. The function should then be recompiled using the *buildmex* script and the XPlane_POS_SURF CNCT library block should be modified appropriately.

The same process is also followed to modify the information sent from the Frasca cockpit controls to the simulation. The header file *FRASCAdataPacket.h* and the C-mex function *FRASCA_To_SIMRW.cpp* should be modified and recompiled, and the FRASCA_Baseline_FEP CNCT library block modified to accept the new signal structure.

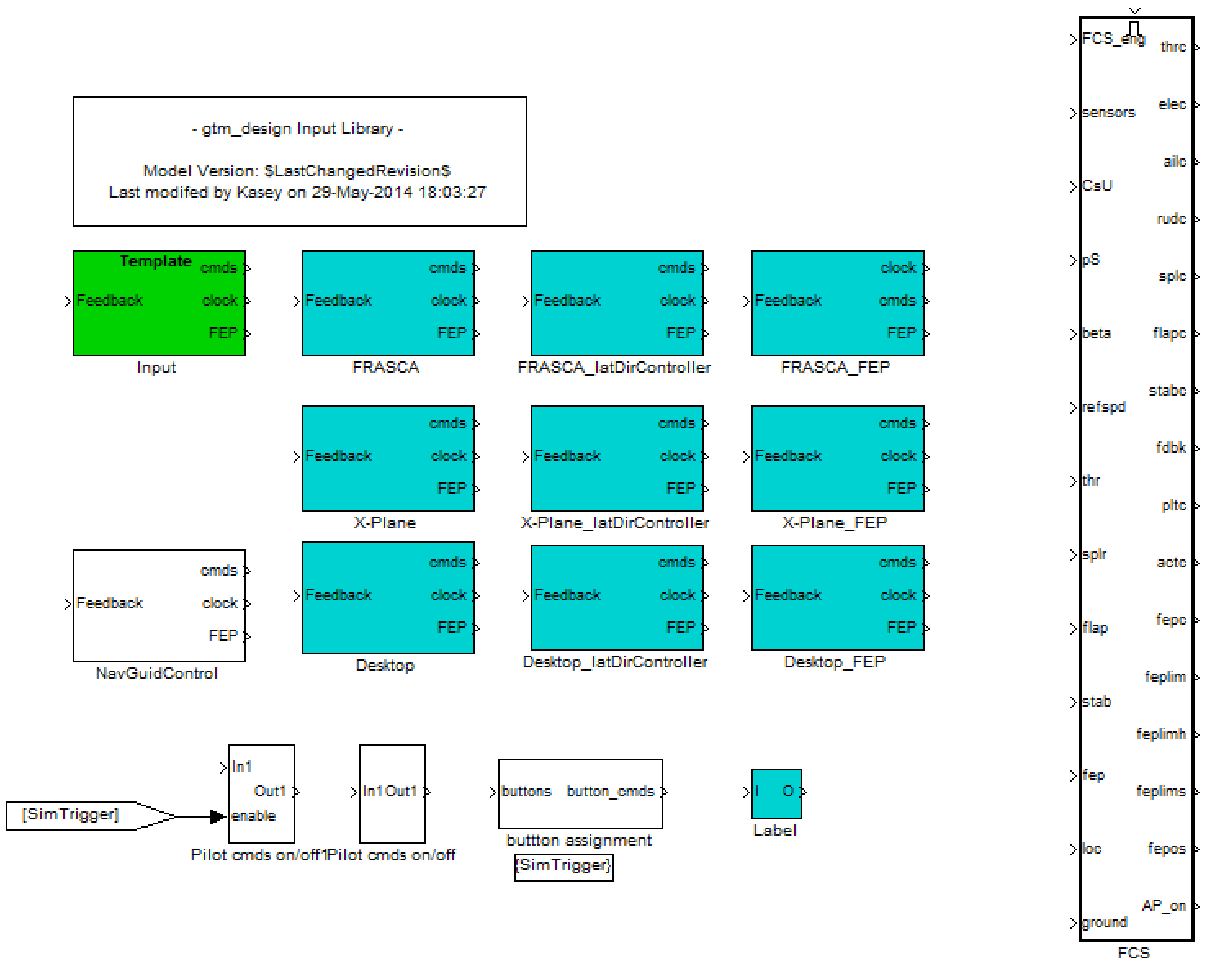


Figure 3.3: TCM Input library, containing various implementations of the communication tools and controller configurations, along with general utilities for the simulation.

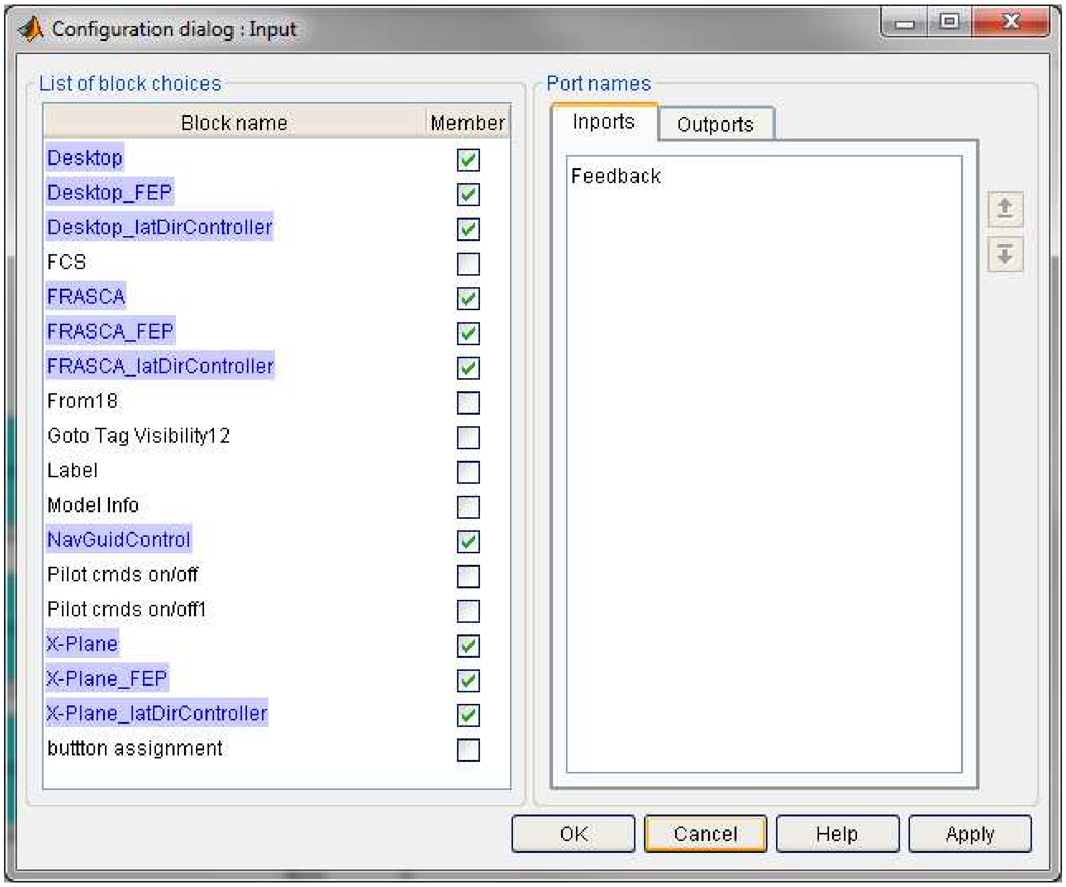
3.2 Input Library

The three distinct simulation modes of batch simulation, desktop piloted simulation, and flight simulator mode each require very different functionality in terms of input sources and communication tools. Additionally, the different modes must all be implemented in the same location in the TCM Simulink model. This poses a challenge to enable efficient switching between simulation modes. If multiple control systems are to be tested as well, then it is desirable to be able to quickly switch between control configurations during testing, which necessitates an additional dimension of functionality within the Input block. To facilitate efficient switching between simulation modes and controller designs, a library was created to store different iterations of the Input blocks, visible in Figure 3.3.

The Input blocks are the larger cyan blocks in the center of Figure 3.3, and are arranged in a grid in order of simulation mode and controller version. Vertically from bottom to top the label prefixes Desktop, X-Plane, and FRASCA correspond to the simulation modes batch simulation, desktop piloted simulation, and flight simulator mode, respectively. Horizontally, the label suffixes on each column denote different controller implementations. The left column of cyan blocks is open-loop operation, there is no flight control law and pilot inputs are passed directly to the control surfaces, a control mode referred to as stick-to-surface. The center column with the label suffix latDirController uses the lateral-directional portion of the baseline flight control law only, longitudinal control is still stick-to-surface and there is no envelope protection. The right column, with the label suffix FEP, implements the full baseline control law with the FEP system. In this mode, the FEP can be turned on or off, allowing the baseline controller to be used independently of the FEP. It is worth noting that the white NavGuidControl block in the lower left of Figure 3.3 is the default block provided by NASA with the TCM simulation, and all subsequent flight control and simulation developments were initially designed based on this template.

Switching between simulation modes and controller configurations is then accomplished by replacing the Input block in the TCM Simulink model with the appropriate block from the Input library. A brute-force approach to this would be to either delete the Input block from the TCM Simulink model and replace it with a new library block, or to replace the block in the Simulink model using the *replace_block(·)* command. A more elegant approach is to use a configurable subsystem to switch between the Input blocks. The configurable subsystem works not by replacing the block itself, but by altering the link to the Input library block. Switching between modes is accomplished by simply changing the target of the link in the Simulink model. The configurable subsystem block is visible as the green block labeled Input in Figure 3.3. Double-clicking this block opens the configuration dialog, shown in Figure 3.4, in which the user can specify which blocks to include in the configurable subsystem. It is critical to ensure that each block included in the configurable subsystem has an an input and output structure identical to all other included blocks. Otherwise, the configurable subsystem will not work. Changing the link target to switch between Input blocks can be done either by right-clicking the configurable subsystem block in the TCM Simulink model and selecting *Block Choice* and the desired block, as in Figure 3.5, or programmatically through the *set_param(·)* function, as used in the setup script for the TCM simulation. The code relevant to the block selection in the setup script is shown in Listing 3.3.

The Input library was implemented in the *UserContent_lib* Simulink library file. This file is the original location that research and design work for the TCM simulation was intended to be stored, and there are additional files in the library not directly related to the configurable subsystem. Since each column of the Input blocks in Figure 3.3 uses an identical controller implementation, the controller can either be


Configuration dialog : Input

List of block choices

Block name	Member
Desktop	✓
Desktop_FEP	✓
Desktop_latDirController	✓
FCS	
FRASCA	✓
FRASCA_FEP	✓
FRASCA_latDirController	✓
From18	
Goto Tag Visibility12	
Label	
Model Info	
NavGuidControl	✓
Pilot cmds on/off	
Pilot cmds on/off1	
X-Plane	✓
X-Plane_FEP	✓
X-Plane_latDirController	✓
buttton assignment	

Port names

Inports: Feedback

Figure 3.4: Dialog for choosing elements of the Input library to include in a configurable subsystem.

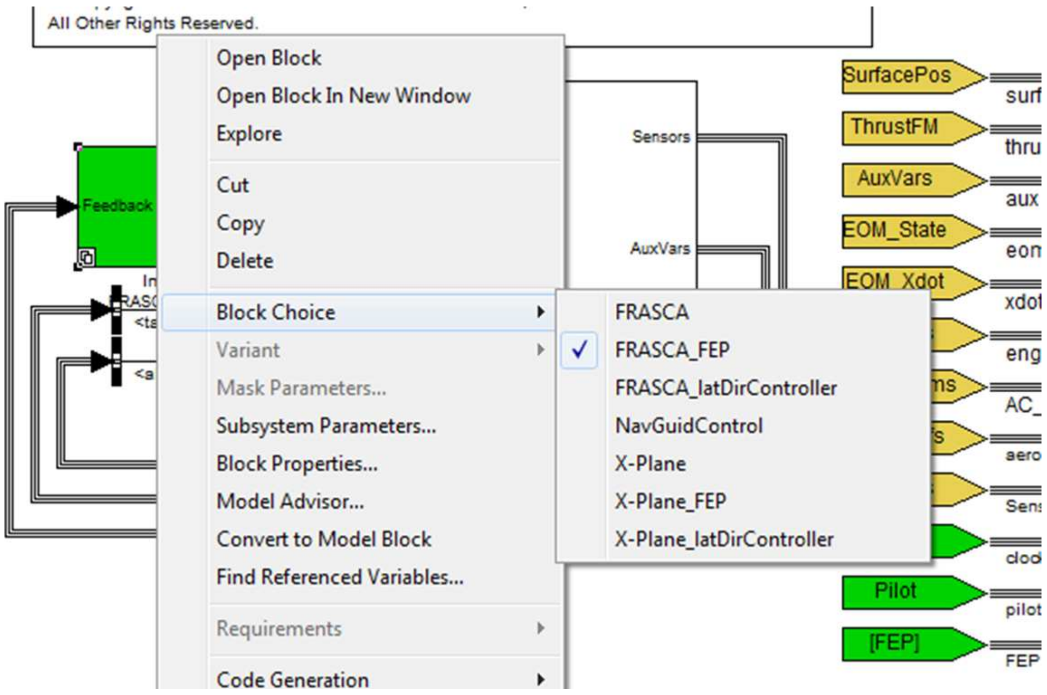


Figure 3.5: The configurable subsystem link may be changed using the right-click menu in Simulink.

Listing 3.3: Programmatic block selection for Input configurable subsystem in the setup function TCM_init_fnc.

```
SimType = IC.simtype;
...

switch Controller
    case 2
        blk_appnd = '_latDirController';
    case 3
        blk_appnd = '_FEP';
    otherwise
        blk_appnd = '';
end

...

config_subsys = [modelname, '/Input'];

...

switch SimType
    case 3
        set_param(config_subsys,'BlockChoice',strcat('FRASCA',blk_appnd));
    case 2
        set_param(config_subsys,'BlockChoice',strcat('X-Plane',blk_appnd));
    case 1
        set_param(config_subsys,'BlockChoice',strcat('Desktop',blk_appnd));
    otherwise
        error('Invalid simulation mode setting.')
end
...
}
```

implemented separately in each block, as is done for the latDirController blocks, or the controller can be linked to an external block, as is the case for the FEP blocks. When the controller is implemented separately in several different blocks, changes made to the controller in one are not automatically propagated to the other blocks using the same controller. This method is most beneficial if the desired controller function varies between blocks, or if there are minimal modifications required to the controller. In the case where the controller implementation is linked to an external source, changes made to the controller in any block can then be propagated back to the source, and the other blocks will update automatically when they load the link to the source.

The FCS block on the far right of Figure 3.3 is an example of an external source block. By placing it in the library model, it can be linked from within each of the FEP Input blocks, ensuring consistent functionality between the implementations. The FCS block is a compiled version of the flight control law, to be discussed further in Chapter 4. Similarly, other blocks in the Input library are source blocks, defined externally to the Input systems to enforce consistent behavior between simulation modes. Additionally, since the communication functions are also defined in an external library, modifications made to these functions are automatically loaded when the TCM Simulink model is opened.

By enabling simulation mode selection through a configurable subsystem, it is possible to efficiently switch between simulation modes and flight control law configurations. However, the use of a configurable subsystem necessitates a change in how the Input systems are modified. Since Simulink does not allow modifications within the configurable subsystem block used in the TCM Simulink model, any modification to the Input blocks must be done within the *UserContent_lib* library model. Once the changes are made and the library is saved, the changes will automatically propagate to the TCM Simulink model when the configurable subsystem link is updated.

Chapter 4

Improving the Simulator Performance

As new elements and features were added to the TCM simulation, it was noticed that the performance of the simulation was not adequate to perform PITL simulation in real-time. Dynamic models typically have a single integrator in the dynamics block to calculate positions and rates from the forces and moments imparted on the vehicle. Each element of the simulation that modifies the forces on the vehicle is an additional calculation that is required before the accelerations can be integrated.

After the addition of the FEP system, the number of calculations required exceeded the abilities of the computer that the simulation was implemented on at ISL. It was observed that the simulation was running at only two-thirds of real-time as observed in Figure 4.1. The lag ratio is computed as

$$lag\ ratio = \frac{t_{system\ clock} - t_{simulation\ clock}}{t_{system\ clock}} \tag{4.1}$$

The lag ratio will be zero when there is no difference between the system clock time and the simulation time, and approaches unity as the simulation pace approaches zero. The quantity $(1 - lag\ ratio)$ is then a characteristic of the severity of the simulation lag. The observed lag was obviously detrimental to PITL testing performance, and in fact invalidated several tests that had been conducted prior its discovery. To mitigate the observed lag issue, several steps were taken to improve the simulation performance.

As an initial step, the computer at ISL was upgraded to a level where the current version of the simulation could be run without delay. The upgrade was a precautionary measure taken with the knowledge that additional components would be added to the control system. While the computer upgrade was helpful, the improvement that produced the greatest benefit for reducing simulation lag was compiling particular elements of the TCM simulation. Compiled executables are able to run much faster than standard Simulink block implementations of the same function, comparable to the difference in execution time between an interpreted function to its compiled counterpart; there is usually an order of magnitude difference in the run time.

The two elements of the simulation that were the largest contributors to the delay were the flight control

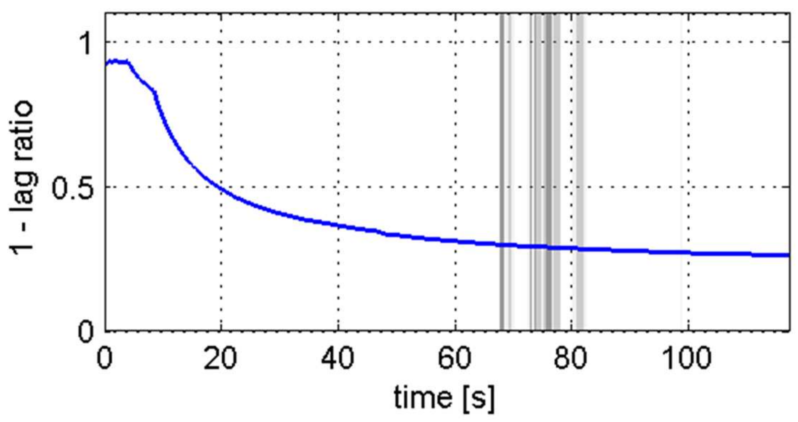


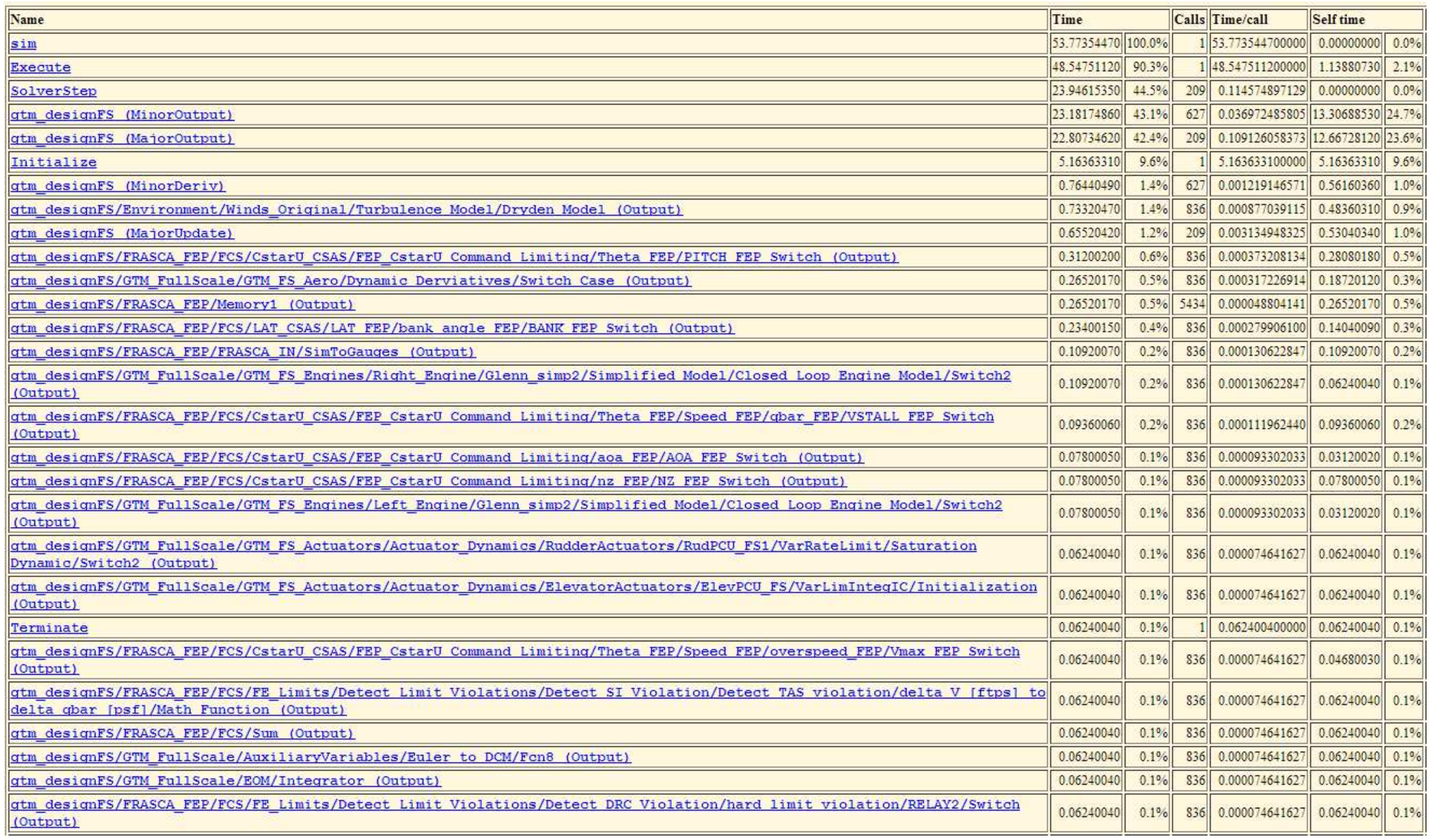
Figure 4.1: Simulation lag observed at ISL. This lag was observed without compiled dynamics or FCS, and prior to the installation of the upgraded computer.

law with the FEP system and the vehicle dynamics subsystem containing the acceleration integrator. To improve the simulation pace, the FCS and dynamics subsystems were replaced in the TCM Simulink model by compiled s-function versions. In the code generation and compilation of the Simulink subsystems, a standard programming process was adopted: source code files are created where changes can be made to alter the function of the Simulink block, the source block is then compiled into a binary form, and the block is run as a mex file. Simulink provides autocoder functionality to automatically generate code for individual blocks and compile it into binary executable form. To further facilitate the compilation process, and to make it more easily usable for the TCM simulation, scripts have been written to automate the procedure. The following sections detail the compilation of the flight dynamics block and flight control law block, respectively, along with the automated compilation procedure.

4.1 Compiling the Flight Dynamics Block

The flight dynamics integrator is one of the primary reasons the simulation was not able to run in real-time, though not due to poor design of the dynamics. Rather, the integrator is called frequently when computing the quantities needed by other simulation components during the next time step, and large disturbances or noisy signals can increase the integrator block execution time. As noted in Figure 4.2 the dynamics integrator is the nineteenth most frequently called block during the simulation, however it is called at most major and minor output steps, which means that simulation is highly dependent on the output of this integrator. In addition, several blocks in the FCS are called more even more frequently than the dynamics integrator, hence the need to also compile the FCS block at the same time.

Motivated by the desire to reduce simulation lag, there are several steps that must be taken prior to the



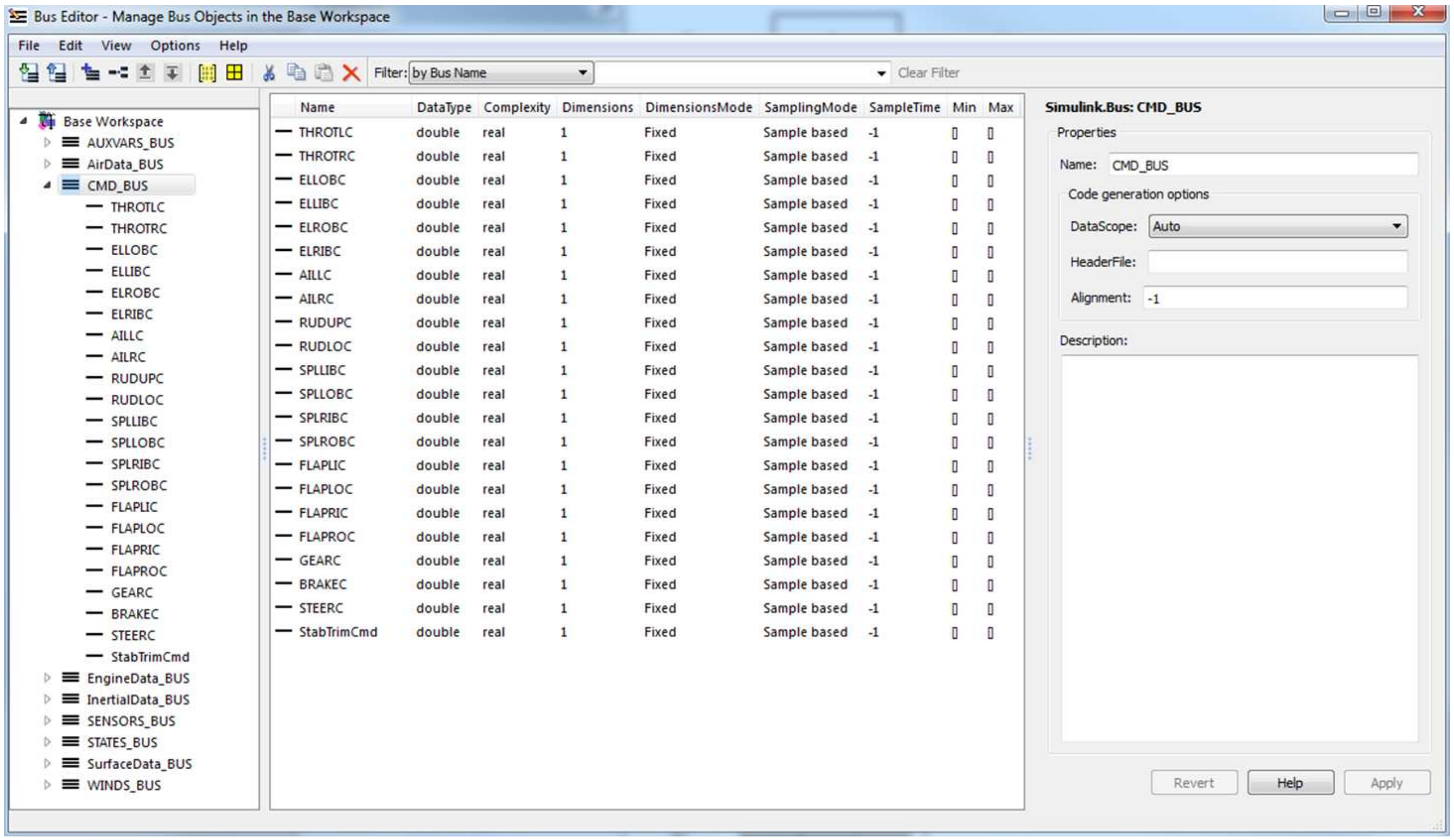
Name	Time		Calls	Time/call	Self time	
sim	53.77354470	100.0%	1	53.773544700000	0.00000000	0.0%
Execute	48.54751120	90.3%	1	48.547511200000	1.13880730	2.1%
SolverStep	23.94615350	44.5%	209	0.114574897129	0.00000000	0.0%
gtm_designFS (MinorOutput)	23.18174860	43.1%	627	0.036972485805	13.30688530	24.7%
gtm_designFS (MajorOutput)	22.80734620	42.4%	209	0.109126058373	12.66728120	23.6%
Initialize	5.16363310	9.6%	1	5.163633100000	5.16363310	9.6%
gtm_designFS (MinorDeriv)	0.76440490	1.4%	627	0.001219146571	0.56160360	1.0%
gtm_designFS/Environment/Winds_Original/Turbulence Model/Dryden Model (Output)	0.73320470	1.4%	836	0.000877039115	0.48360310	0.9%
gtm_designFS (MajorUpdate)	0.65520420	1.2%	209	0.003134948325	0.53040340	1.0%
gtm_designFS/FRASCA_FEP/FCS/CstarU_CSAS/FEP_CstarU Command Limiting/Theta FEP/PITCH FEP Switch (Output)	0.31200200	0.6%	836	0.000373208134	0.28080180	0.5%
gtm_designFS/GTM_FullScale/GTM_FS_Aero/Dynamic Derviatives/Switch Case (Output)	0.26520170	0.5%	836	0.000317226914	0.18720120	0.3%
gtm_designFS/FRASCA_FEP/Memory1 (Output)	0.26520170	0.5%	5434	0.000048804141	0.26520170	0.5%
gtm_designFS/FRASCA_FEP/FCS/LAT_CSAS/LAT FEP/bank angle FEP/BANK FEP Switch (Output)	0.23400150	0.4%	836	0.000279906100	0.14040090	0.3%
gtm_designFS/FRASCA_FEP/FRASCA_IN/SimToGauges (Output)	0.10920070	0.2%	836	0.000130622847	0.10920070	0.2%
gtm_designFS/GTM_FullScale/GTM_FS_Engines/Right_Engine/Glenn_simp2/Simplified Model/Closed Loop Engine Model/Switch2 (Output)	0.10920070	0.2%	836	0.000130622847	0.06240040	0.1%
gtm_designFS/FRASCA_FEP/FCS/CstarU_CSAS/FEP_CstarU Command Limiting/Theta FEP/Speed FEP/qbar_FEP/VSTALL FEP Switch (Output)	0.09360060	0.2%	836	0.000111962440	0.09360060	0.2%
gtm_designFS/FRASCA_FEP/FCS/CstarU_CSAS/FEP_CstarU Command Limiting/aoa FEP/AOA FEP Switch (Output)	0.07800050	0.1%	836	0.000093302033	0.03120020	0.1%
gtm_designFS/FRASCA_FEP/FCS/CstarU_CSAS/FEP_CstarU Command Limiting/nz FEP/NZ FEP Switch (Output)	0.07800050	0.1%	836	0.000093302033	0.07800050	0.1%
gtm_designFS/GTM_FullScale/GTM_FS_Engines/Left_Engine/Glenn_simp2/Simplified Model/Closed Loop Engine Model/Switch2 (Output)	0.07800050	0.1%	836	0.000093302033	0.03120020	0.1%
gtm_designFS/GTM_FullScale/GTM_FS_Actuators/Actuator_Dynamics/RudderActuators/RudPCU_FS1/VarRateLimit/Saturation Dynamic/Switch2 (Output)	0.06240040	0.1%	836	0.000074641627	0.06240040	0.1%
gtm_designFS/GTM_FullScale/GTM_FS_Actuators/Actuator_Dynamics/ElevatorActuators/ElevPCU_FS/VarLimIntegIC/Initialization (Output)	0.06240040	0.1%	836	0.000074641627	0.06240040	0.1%
Terminate	0.06240040	0.1%	1	0.062400400000	0.06240040	0.1%
gtm_designFS/FRASCA_FEP/FCS/CstarU_CSAS/FEP_CstarU Command Limiting/Theta FEP/Speed FEP/overspeed_FEP/Vmax FEP Switch (Output)	0.06240040	0.1%	836	0.000074641627	0.04680030	0.1%
gtm_designFS/FRASCA_FEP/FCS/FE_Limits/Detect Limit Violations/Detect SI Violation/Detect TAS violation/delta V [ftps] to delta qbar [psf]/Math Function (Output)	0.06240040	0.1%	836	0.000074641627	0.06240040	0.1%
gtm_designFS/FRASCA_FEP/FCS/Sum (Output)	0.06240040	0.1%	836	0.000074641627	0.06240040	0.1%
gtm_designFS/GTM_FullScale/AuxiliaryVariables/Euler to DCM/Fcn8 (Output)	0.06240040	0.1%	836	0.000074641627	0.06240040	0.1%
gtm_designFS/GTM_FullScale/EOM/Integrator (Output)	0.06240040	0.1%	836	0.000074641627	0.06240040	0.1%
gtm_designFS/FRASCA_FEP/FCS/FE_Limits/Detect Limit Violations/Detect DRC Violation/hard limit violation/RELAY2/Switch (Output)	0.06240040	0.1%	836	0.000074641627	0.06240040	0.1%

Figure 4.2: Diagnostic report showing the frequency and call time during a simulation. Several blocks in the aircraft dynamics and flight control systems are called very frequently, and have long call times which delay simulation execution.

compilation of the dynamics block. The structure of the dynamics block is fixed as shown in Figure 2.2, and the I/O structure, while alterable in the Simulink model, is essentially fixed since the model is generated by NASA and any developments made to the FCS must be compatible with the given I/O structure. Additionally, many of the existing functions that support the TCM model are written in a way that precludes compatibility with a compiled dynamics block. In particular, the trimming function that calculates control inputs that yield an equilibrium solution (zero accelerations) at a desired flight condition requires interaction with specific blocks and their handles. However, once a system is compiled it no longer contains blocks, so the original block handles do not exist and the trimming function is not able to operate. For simplicity and ease of use it is strongly desired to retain all of the preexisting functionality of the TCM simulation and its support tools.

To preserve the I/O structure of the dynamics, the buses that serve as the interface to the dynamics block need be defined to be compatible with an s-function block. Bus structures must be defined as bus objects in order for them to be recognized by the Simulink autocoder. Bus object definition is similar in nature to requiring a header file specifying the definition of a structure object in a conventional coding environment. The definition of the bus objects is done using the Simulink bus editor. As with normal programming environments, each structure must be specified individually. Figure 4.3 shows the Simulink bus editor environment, which may be accessed via the Simulink toolbar by clicking *Tools* $\rightarrow$ *Bus Editor*. Buses created in this interface can be exported to the Matlab workspace and saved as a *.mat* file. The buses must then be loaded when the simulation is started. Additionally, the bus type must be specified in the properties block of the bus object from which the signal originates, and individual bus objects must be defined for each unique bus which is an input to or output from the compiled block.

When the TCM simulation is trimmed to desired flight condition, it is set up so that input commands are delta commands to the flight control surfaces. That is, the trimmed configuration is stored in a data structure and the trimmed positions for the surface are implemented through constant value blocks that access the data structure fields. Incoming commands are then interpreted as deflections about the trim point. Since the trim positions need to modified each time the simulation is started at a different flight condition, the trim values cannot be hard coded in the source code generated from the block, but instead must be tunable parameters. Again, there is a standard analogy from conventional programming to help conceptually illustrate the role tunable parameters play in a Simulink s-function block: The autocoder generates a function that can by invoked by the Simulink interpreter at run time, and the tunable parameters are essentially input arguments passed to the function by the interpreter. However, the Simulink coder for the version of Matlab currently used does not support the use of data structure fields as tunable parameters. As a workaround, it is possible


Bus Editor - Manage Bus Objects in the Base Workspace
File Edit View Options Help
Filter: by Bus Name
Clear Filter
Revert
Help
Apply

Base Workspace
- AUXVARS_BUS
- AirData_BUS
- CMD_BUS
  - THROTLC
  - THROTRC
  - ELLOBC
  - ELLIBC
  - ELROBC
  - ELRIBC
  - AILLC
  - AILRC
  - RUDUPC
  - RUDLOC
  - SPLLIBC
  - SPLLOBC
  - SPLRIBC
  - SPLROBC
  - FLAPLIC
  - FLAPLOC
  - FLAPRIC
  - FLAPROC
  - GEARC
  - BRAKEC
  - STEERC
  - StabTrimCmd
- EngineData_BUS
- InertialData_BUS
- SENSORS_BUS
- STATES_BUS
- SurfaceData_BUS
- WINDS_BUS

Name	DataType	Complexity	Dimensions	DimensionsMode	SamplingMode	SampleTime	Min	Max
THROTLC	double	real	1	Fixed	Sample based	-1	[]	[]
THROTRC	double	real	1	Fixed	Sample based	-1	[]	[]
ELLOBC	double	real	1	Fixed	Sample based	-1	[]	[]
ELLIBC	double	real	1	Fixed	Sample based	-1	[]	[]
ELROBC	double	real	1	Fixed	Sample based	-1	[]	[]
ELRIBC	double	real	1	Fixed	Sample based	-1	[]	[]
AILLC	double	real	1	Fixed	Sample based	-1	[]	[]
AILRC	double	real	1	Fixed	Sample based	-1	[]	[]
RUDUPC	double	real	1	Fixed	Sample based	-1	[]	[]
RUDLOC	double	real	1	Fixed	Sample based	-1	[]	[]
SPLLIBC	double	real	1	Fixed	Sample based	-1	[]	[]
SPLLOBC	double	real	1	Fixed	Sample based	-1	[]	[]
SPLRIBC	double	real	1	Fixed	Sample based	-1	[]	[]
SPLROBC	double	real	1	Fixed	Sample based	-1	[]	[]
FLAPLIC	double	real	1	Fixed	Sample based	-1	[]	[]
FLAPLOC	double	real	1	Fixed	Sample based	-1	[]	[]
FLAPRIC	double	real	1	Fixed	Sample based	-1	[]	[]
FLAPROC	double	real	1	Fixed	Sample based	-1	[]	[]
GEARC	double	real	1	Fixed	Sample based	-1	[]	[]
BRAKEC	double	real	1	Fixed	Sample based	-1	[]	[]
STEERC	double	real	1	Fixed	Sample based	-1	[]	[]
StabTrimCmd	double	real	1	Fixed	Sample based	-1	[]	[]

Simulink.Bus: CMD_BUS

Properties

Name: CMD_BUS

Code generation options

DataScope: Auto

HeaderFile:

Alignment: -1

Description:

Figure 4.3: Simulink bus editor environment.

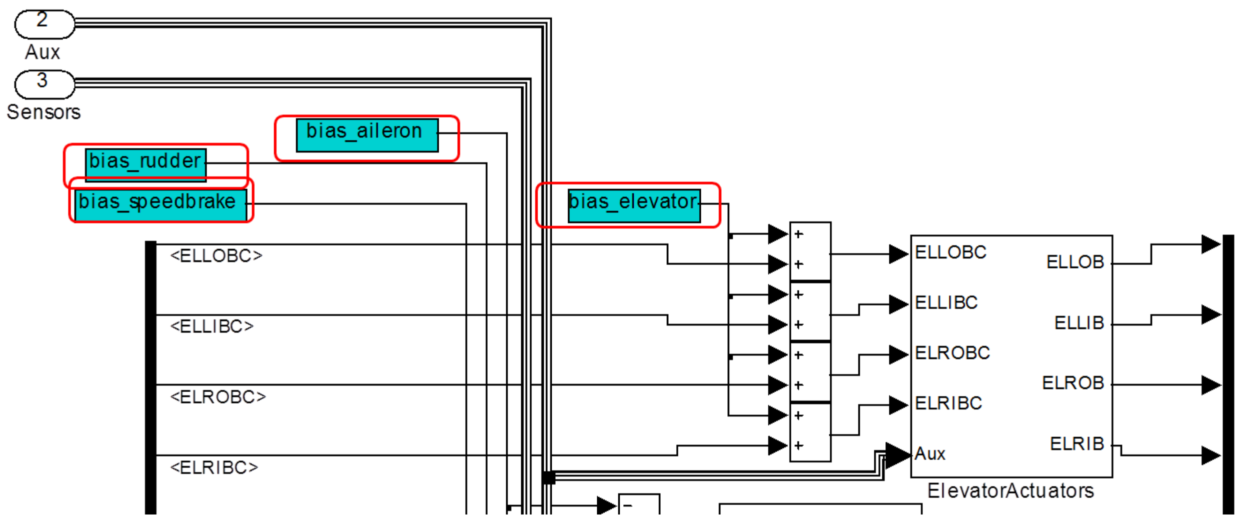


Figure 4.4: The circled constant block values were modified from calling a structure field to calling a standard variable to set the value of the block to be a tunable parameter in the compiled version of the dynamics block.

to define the values in the constant blocks as non-structure elements and pass the structure field to the parameter at run time. Figure 4.4 shows the implementation of this workaround in the TCM actuator system block.

With the bus objects defined and the tunable parameter workaround complete, it is possible to compile the dynamics block. The model in which the block to be compiled is implemented must be a functioning model; it must be able to run without any errors. Therefore all incoming bus signals must be defined and any variable used by the model must be defined and nonempty. The block can be compiled directly from the main simulation model if desired, but this is not good practice since the goal of compiling the dynamics block is to replace it completely. Unless a backup is made of the original block, it will be lost upon replacement, along with the possibility to easily make further modifications. In theory, the code generated by the autocoder could potentially be modified and recompiled, but this is far more trouble than it is worth. Instead, a reduced version of the original simulation is used which contains only the bare minimum elements necessary to run. The reduced model is shown in Figure 4.5, and is saved under the file name *gtm_designFS_DYN.mdl*. The inputs to the dynamics block GTM_FS are just the normal input buses as defined in the bus editor in the original simulation model, with all signals set to zero. These input signals are essentially just placeholders, as they are only used to define the correct signal structure. The global GoTo tags (SurfacePos, ThrustFM, etc., and APengage) are included to provide signal routing destinations and sources in order to suppress warnings and to prevent errors resulting from undefined signals. The input and output subsystem in the upper right corner Figure 4.5 is used by the TCM support functions required to initialize the model.

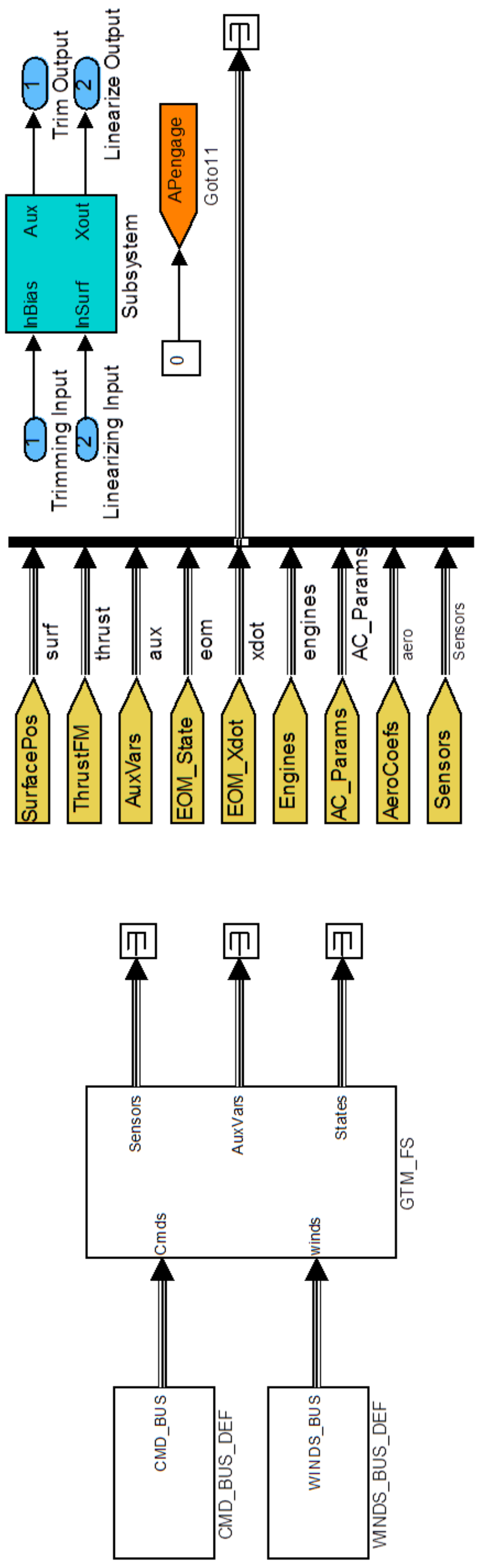


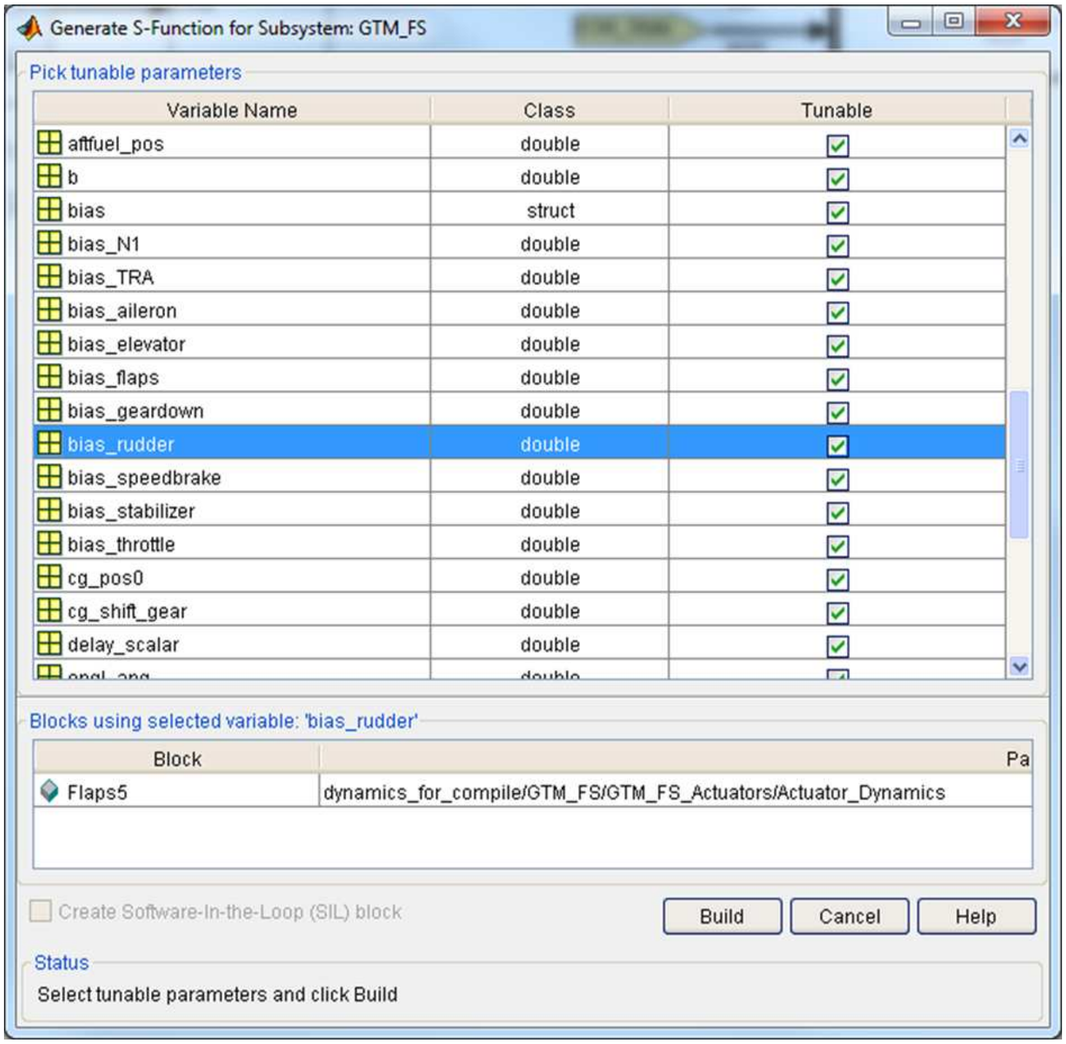
Figure 4.5: Reduced version of the TCM Simulink model used for compiling the dynamics block. The reduced version contains only the dynamics block GTM_FS and other necessary components.

Compiling the block can be accomplished by right-clicking on the GTM_FS block and selecting *Code Generation → Generate S-Function*, which will start the build process by which the block code is generated and compiled. The first window that pops up asks the user to select which parameters to make tunable, an example of which is given in Figure 4.6. Any constant-valued variable that is used in the block and its subsystem is automatically populated to the list of variables, including values for constant blocks, integrator initial conditions, saturation block upper and lower limits, etc. The *Pick tunable parameters* section of *Generate S-Function* window is where the user should select any variables that need to be tunable, as their values will be hard-coded if they are not selected. Clicking *Build* in the parameter selection window will call the autocoder to generate the source code files and the Matlab compiler to build the executable. The output of the build process is the code files (C source code and header files in the case of the TCM simulation since C was the target language specified in the Simulink configuration parameters), the Matlab executable file which is called by the Simulink interpreter, and a new untitled model containing the s-function version of the dynamics block. The s-function block has exactly the same functionality as the dynamics block from which it was compiled, however it runs much more efficiently since it is a binary executable. Once the mex file has been moved to a location in the path of the original model's working directory, the compiled block can be used to directly replace the original dynamics block. The tunable parameters selected during the build process are accessed through the compiled block's mask; double-clicking the block brings up the mask dialog where the variables for the tunable parameters can be altered. These default to the names of the variables at compile time. Changing the names of the altered variables back to their original structure field references restores the full functionality of the TCM support functions.

4.2 Compiling the Flight Control System Block

Similar to the integrator in the flight dynamics, there are several elements in the FCS that are called frequently during the simulation and require substantial time to execute at each time step, as observed in Figure 4.2. In order to speed up the simulation rate, it is desired to compile the FCS in the TCM Simulink model as well. The compilation process is similar to that for compiling the dynamics block and follows the same basic steps.

A model containing an editable version of the FCS has been saved under the *FCS_src* directory entitled *gtm_designFS_FCS.mdl*, containing the uncompiled Simulink implementation of the control system and FEP in the FCS block in Figure 4.7. The creation of the FCS source model is similar to the source model created in Figure 4.5, and serves the same function.



Variable Name	Class	Tunable
aftfuel_pos	double	✓
b	double	✓
bias	struct	✓
bias_N1	double	✓
bias_TRA	double	✓
bias_aileron	double	✓
bias_elevator	double	✓
bias_flaps	double	✓
bias_geardown	double	✓
bias_rudder	double	✓
bias_speedbrake	double	✓
bias_stabilizer	double	✓
bias_throttle	double	✓
cg_pos0	double	✓
cg_shift_gear	double	✓
delay_scalar	double	✓

Blocks using selected variable: 'bias_rudder'

Block	Pa
Flaps5	dynamics_for_compile/GTM_FS/GTM_FS_Actuators/Actuator_Dynamics

Select tunable parameters and click Build

Figure 4.6: Tunable parameter selection window. Any variables not selected will be hard coded as numeric values when the autocoder is invoked if they are not selected here.

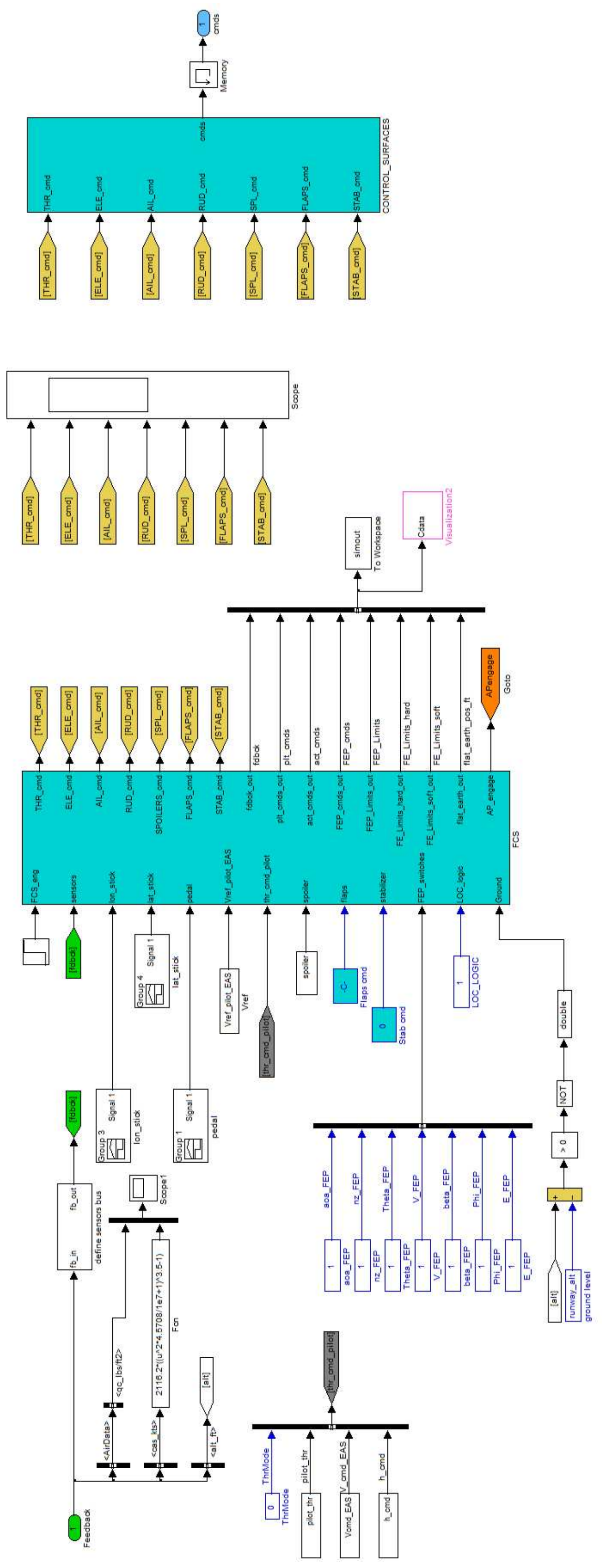


Figure 4.7: Flight control system Simulink source. The FCS block (center of diagram) is compiled for implementation in the TCM Simulink model.

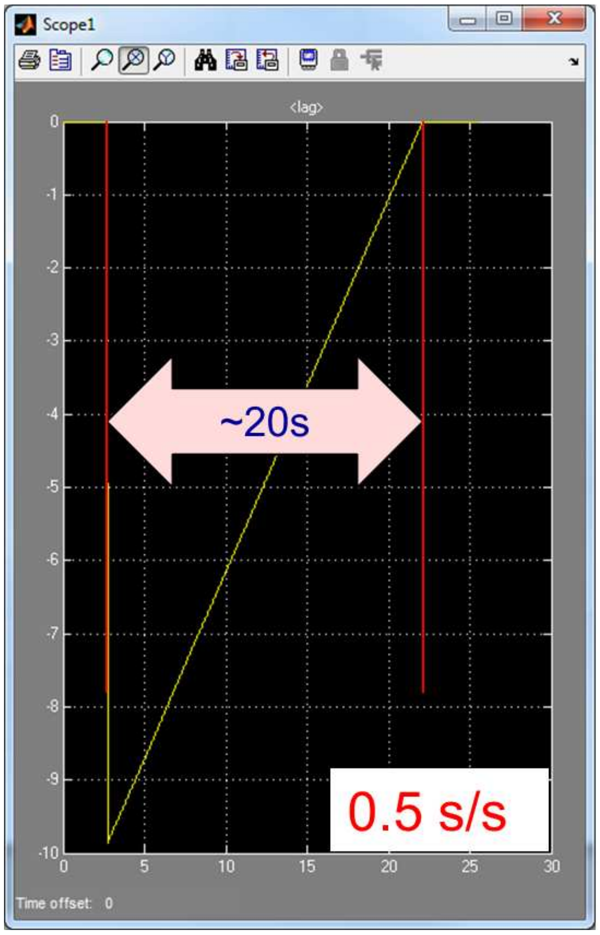
Again, the input and output bus signals of the FCS block need to be defined as bus objects using the Simulink bus editor. For the FCS, there are no parameters that need to be made tunable. None of the control system or FEP system parameters need to be modified each time the simulation is started and do not affect the trim condition of the model. It is therefore acceptable to allow the autocoder to hard code these values into the executable function. However, if it is desired to alter some set of parameters within the FCS subsystem, the selection of tunable parameters is done in a manner similar to that for the dynamics block. Right-clicking on the FCS block and selecting *Code Generation* $\rightarrow$ *Generate S-Function*, then clicking *Build* will generate the binary mex file with source code along with a Simulink model containing the compiled version of the FCS block. The s-function version of the block can then be implemented in the TCM simulation, replacing the original system, while all of the original functionality of the block will be retained.

The performance improvement from compiling both the dynamics and FCS can be observed in Figure 4.8. The performance test was conducted by artificially stopping the simulation for a short time while the computer system clock was still running. The simulation was then allowed to run and attempt to catch up to the real-time target. Using the original version of the simulation, for an approximately ten-second induced lag the recovery time was about twenty seconds, a lag recovery rate of roughly 0.5 *sec/sec*. After implementing the compiled versions of the two systems, the recovery time dropped to about 12.5 seconds with a recovery rate of roughly 0.8 *sec/sec*. The increase in lag recovery rate and reduction in lag recovery time represent a substantial improvement in simulation performance, and ensure that the simulation will run in real-time on the upgraded computer at the ISL.

4.3 Automating the Compilation Process

While compiling the dynamics and FCS elements yields significant improvements in simulation performance, it complicates the design procedure when adding or modifying features in either system. Where before it was possible to open the model, make a modification, and then run the simulation to verify the functionality, the procedure is now much more complicated. For example, if the modification is to the FCS (most of the design changes will be; it is usually desired to avoid modifications to the aircraft dynamics) the *gtm_designFS_FCS* source model must be opened so that modifications can be made and verified there. The compilation procedure outlined in Sections 4.1 and 4.2 must then be followed to generate a block which then needs to be copied over to the main *gtm_designFS* model.

Compiling subsystems in the above manner is a cumbersome procedure, and several steps must be



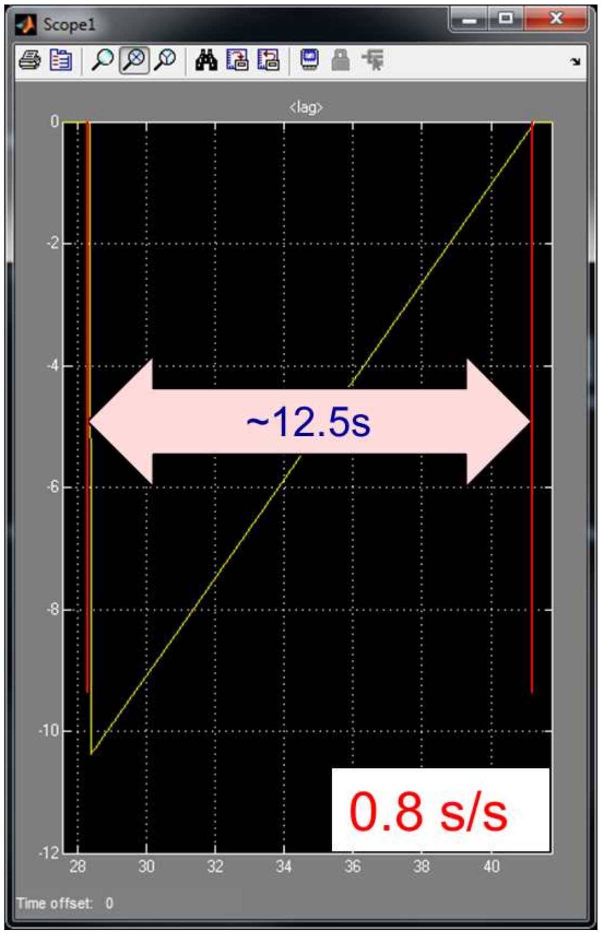


Figure 4.8: Simulation lag recovery for the for the TCM simulation without (left) and with (right) the compiled versions of the dynamics and FCS. An artifical lag was induced on the simulation, which was then allowed to recover to real-time. Recovery is significantly faster using the compiled blocks, indicating a large improvement in simulation performance.

followed each time a new modification is made. In order to streamline the compilation process, several of the intermediate steps have been automated. Two different functions have been created, one for compiling the dynamics block and one for compiling the FCS block. These functions handle the tasks of compiling the source blocks, saving a copy of the compiled version, copying the compiled version into its final location for the TCM simulation, and copying the executable files into the path of the TCM simulation. The automated compilation of the FCS is the simpler of the two processes and will be discussed first, followed by the automated compilation of the dynamics.

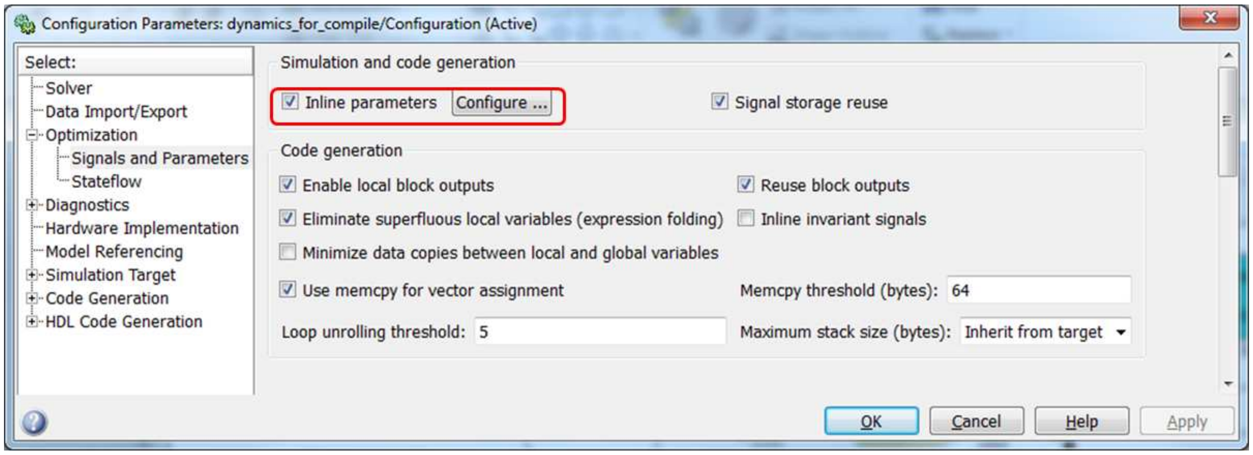
4.3.1 Automating FCS Compilation

The FCS is always implemented in the Input block of the simulation, and the library blocks need to use identical copies of the FCS to ensure consistent operation between simulation modes. To ensure that the FCS is consistent across the range of simulation modes, the process from Section 4.2 was used to generate a compiled version of the FCS, which was then copied into the library *UserContent_lib*, see Figure 3.3. The uncompiled version of the FCS was then deleted in each of the _FEP Input blocks and replaced with the compiled version using Simulink's drag-and-drop copy method. Since the compiled version is now a library block, instead of making a wholesale copy Simulink creates a link back to the library block. Thus, each time a change is made to the compiled FCS library block, the change is automatically propagated to all of of the linked versions.

The procedure for compiling the FCS has now been reduced to two steps. Changes are made to the Simulink source model *gtm_designFS_FCS*, then running the Matlab script *buildFCS* performs the remaining steps to compile the block and replace its library implementation and the executable functions used in the simulation. The full script can be viewed in Listing A.1.

When the *buildFCS* script is called, the variables in the Matlab workspace need to be cleared since the setup scripts for the FCS use many of the same variables as the main simulation. The user is prompted before the data in the workspace is deleted, in case anything needs to be saved. The script then clears the workspace, saves the current directory location, and changes the working directory to *FCS_src*. Modifiedd setup programs are needed for the _FCS version of the simulation, since this model does not have all of the functionality or components of the full simulation. The setup script *setupFS_FCS* is a modified version of the original setup script provided with the TCM simulation by NASA. All of the functions called within the original script have been copied to the *FCS_src* directory and appended with *_FCS* to distinguish them from the originals.

The compilation script then uses the Matlab Real-Time Workshop (RTW) to compile the FCS block


Configuration Parameters: dynamics_for_compile/Configuration (Active)

Select:
- Solver
- Data Import/Export
- Optimization
  - Signals and Parameters
  - Stateflow
- Diagnostics
- Hardware Implementation
- Model Referencing
- Simulation Target
- Code Generation
- HDL Code Generation

Simulation and code generation

☑ Inline parameters [Configure ...]   ☑ Signal storage reuse

Code generation

☑ Enable local block outputs   ☑ Reuse block outputs
☑ Eliminate superfluous local variables (expression folding)   ☐ Inline invariant signals
☐ Minimize data copies between local and global variables
☑ Use memcpy for vector assignment   Memcpy threshold (bytes): 64
Loop unrolling threshold: 5   Maximum stack size (bytes): Inherit from target

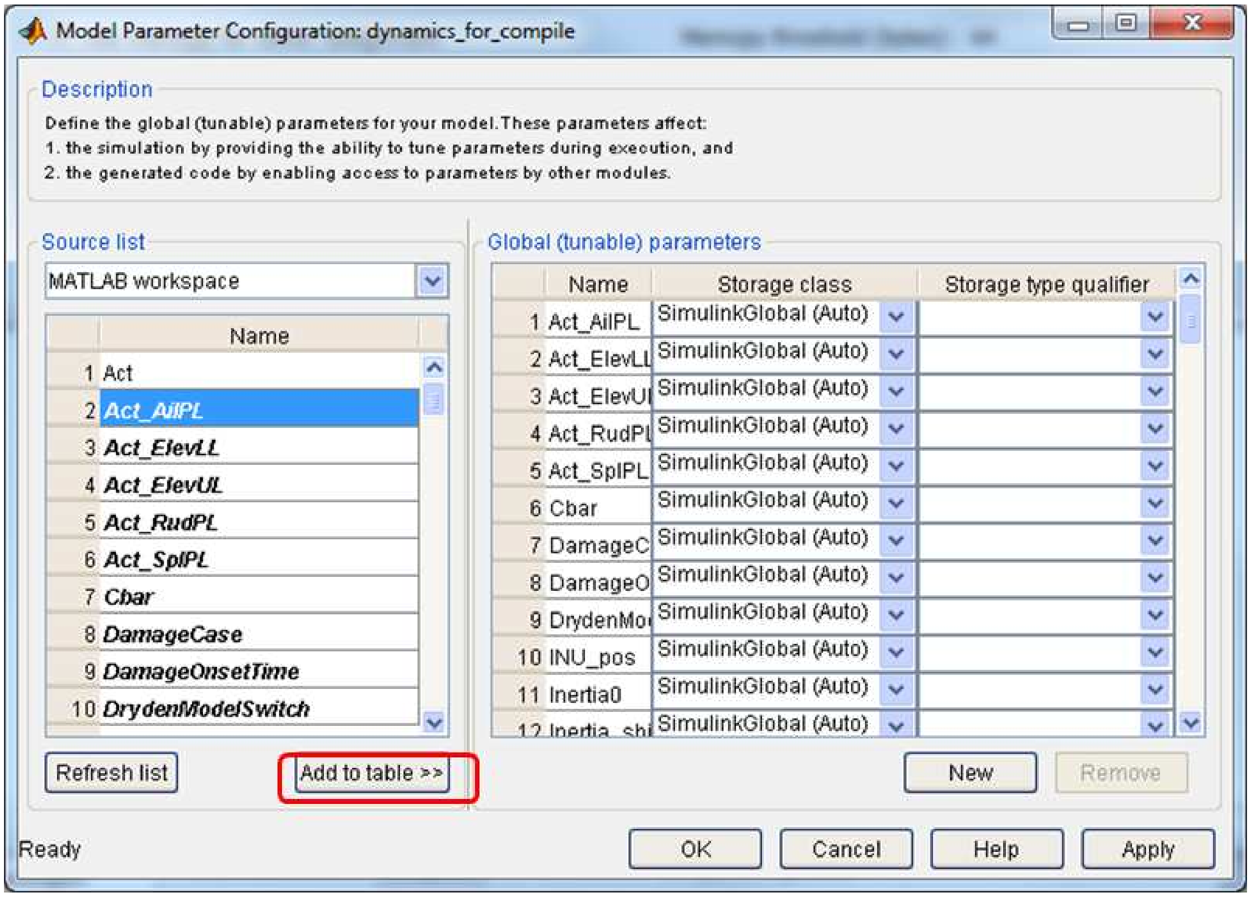
Figure 4.9: Selecting the use of inline parameters in the Simulink configuration parameters menu.

through the *rtwbuild(·)* function, see line 38 of Listing A.1. The compiled block is then saved under the filename *FCS_sfcn* and used to replace the version in the *UserContent_lib* library. The Matlab executable is then copied to the *mexfiles* directory where all the binaries used by the TCM simulation are stored. Finally, the user is prompted to run the main setup file *setupTCM* to refresh the library links in the simulation. The prompt signifies the completion of the process outlined in Section 4.2.

4.3.2 Automating the Dynamics Compilation

Additional setup work is required to automate the compilation process for the TCM dynamics. The dynamics block requires tunable parameters, which were selected manually in the procedure in Section 4.1. Unfortunately, it is not possible to programmatically select tunable parameters in the same way. Instead, RTW requires the use of inline parameters. To specify inline parameters, after running the *setupFS_DYN* script, open the source model *dynamics_for_compile* and select *Simulation* → *Configuration Parameters*. Then select *Optimization* → *Signals and Parameters* and ensure *Inline parameters* is checked, as in Figure 4.9. Click *Configure* to open the dialog to select the parameters which should be inlined and made tunable, shown in Figure 4.10. Confirming these changes sets the selected variables as inlined tunable parameters in the generated code and in the compiled version of the dynamics block.

Additionally, since the trimming function relies on the uncompiled version of the dynamics, it is necessary to swap these two versions in and out depending on whether the simulation is being trimmed or executed. To facilitate the block swapping, a storage model *gtm_designFS_sfun_model* was created to store the unaltered version of the dynamics, the version with the modified parameter names, and the s-function version. Figure 4.11 shows the contents of the storage model, along with the internal structure of each block.



Description

Define the global (tunable) parameters for your model. These parameters affect:
1. the simulation by providing the ability to tune parameters during execution, and
2. the generated code by enabling access to parameters by other modules.

Source list

MATLAB workspace

	Name
1	Act
2	*Act_AilPL*
3	***Act_ElevLL***
4	***Act_ElevUL***
5	***Act_RudPL***
6	***Act_SplPL***
7	***Cbar***
8	***DamageCase***
9	***DamageOnsetTime***
10	***DrydenModelSwitch***

Global (tunable) parameters

	Name	Storage class	Storage type qualifier
1	Act_AilPL	SimulinkGlobal (Auto)	
2	Act_ElevLL	SimulinkGlobal (Auto)	
3	Act_ElevUl	SimulinkGlobal (Auto)	
4	Act_RudPl	SimulinkGlobal (Auto)	
5	Act_SplPL	SimulinkGlobal (Auto)	
6	Cbar	SimulinkGlobal (Auto)	
7	DamageC	SimulinkGlobal (Auto)	
8	DamageO	SimulinkGlobal (Auto)	
9	DrydenMo	SimulinkGlobal (Auto)	
10	INU_pos	SimulinkGlobal (Auto)	
11	Inertia0	SimulinkGlobal (Auto)	
12	Inertia_shi	SimulinkGlobal (Auto)	

Ready

Figure 4.10: Inlined parameters are selected using the model parameter configuration menu.

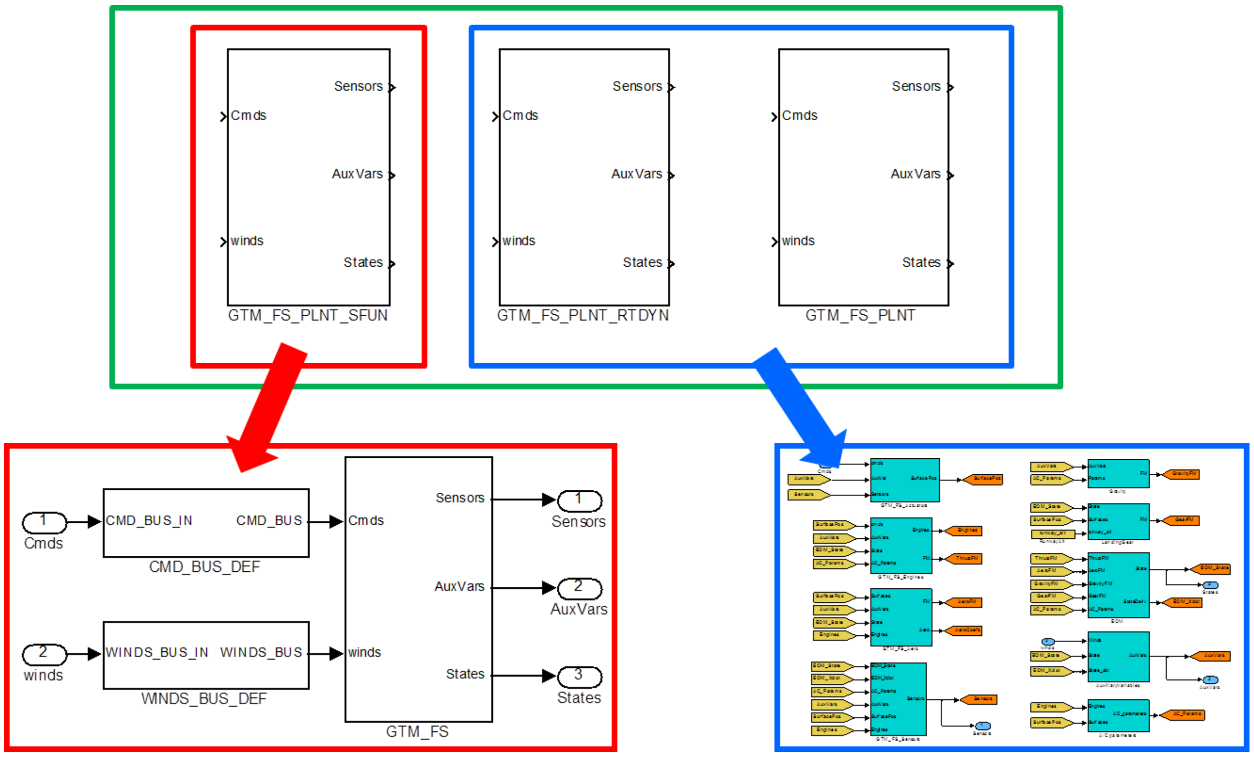


Figure 4.11: Structure of the TCM dynamics storage model. The compiled version of the dynamics and its internal structure are highlighted in red, while the uncompiled versions with and without modifications to the trim condition variables are highlighted in blue.

The script *buildDYN* automates the compilation of the dynamics block, and the full Matlab source code is provided in Listing A.2. Since the _DYN version of the simulation also uses the same variables as the full simulation, the Matlab workspace variables must be cleared. The user is prompted to proceed or quit to save data. The current working directory is saved and the working directory is changed to *DYN_src*. Here, the *setupFS_DYN* function is called, this time as an overloaded modification of the original TCM setup file from NASA. The setup file runs the scripts to load required variables into the model workspace, then opens the *gtm_designFS_DYN* model with the original version of the dynamics. Here the model can be trimmed and the trim information is stored into modified constant parameters, shown in Listing 4.1. If given an optional argument of 1, the setup function initializes and loads the required parameters into the model workspace and copies the latest version of the dynamics block with modified constants into the model used for compiling, shown Figure 4.5. Otherwise, the compiled version of the dynamics is used to update and replace the dynamics block in the *gtm_designFS_DYN* model. The latter option (an input argument not equal to 1 passed to the setup function) may be useful for verification of functionality, but is not used during the automated compilation process.

Listing 4.1: Modification to setup script for compiling dynamics.

```
    MWS.bias_aileron = MWS.bias.aileron;
    MWS.bias_elevator = MWS.bias.elevator;
    MWS.bias_flaps = MWS.bias.flaps;
    MWS.bias_geardown = MWS.bias.geardown;
    MWS.bias_N1 = MWS.bias.N1;
    MWS.bias_NfR = MWS.bias.NfR;
    MWS.bias_rudder = MWS.bias.rudder;
    MWS.bias_rudhys = MWS.bias.rudhys;
    MWS.bias_speedbrake = MWS.bias.speedbrake;
    MWS.bias_stabilizer = MWS.bias.stabilizer;
    MWS.bias_throttle = MWS.bias.throttle;
    MWS.bias_TRA = MWS.bias.TRA;

    MWS.Act_ElevUL = MWS.Act.ElevUL;
    MWS.Act_ElevLL = MWS.Act.ElevLL;
    MWS.Act_AilPL = MWS.Act.AilPL;
    MWS.Act_RudPL = MWS.Act.RudPL;
    MWS.Act_SplPL = MWS.Act.SplPL;

if exist('myopt') && ~isempty(myopt) && myopt==1
    loadmws2([],'gtm_designFS_DYN');
    loadmws2(MWS,'gtm_designFS_DYN');
    bdclose('gtm_designFS_DYN');
    open_system('dynamics_for_compile');
    replace_block('dynamics_for_compile', 'Name', 'GTM_FS', 'gtm_designFS_sfun_model/
        GTM_FS_PLNT_RTDYN','noprompt');
else
    appendmws(MWS,'gtm_designFS_DYN')
  replace_block('gtm_designFS_DYN', 'Name', 'GTM_FullScale', 'gtm_designFS_DYN_sfun_model/
      GTM_FS_PLNT_SFUN', 'noprompt');
    sim('gtm_designFS_DYN',0.0);
end
```

The *rtwbuild(·)* function is then used to compile the dynamics block, and the compiled block is saved. The number of parameters in the s-function mask of the compiled block needs to be determined next so that desired parameters can be replaced automatically. A list of old and new parameter names is specified, and the list of parameters is parsed and the old parameter names replaced by their new versions. Next, the compiled block with updated parameters is saved and copied to the storage model, replacing the older version of the block. The executable file is then copied to the *mexfiles* directory and the user is prompted to run *setupTCM* to import the newest version of the compiled dynamics into the simulation.

Using the two functions *buildFCS* and *buildDYN*, it is thus possible to automate the procedure of compiling the different elements of the simulation to reduce the burden on the user and improve the simulation performance. Under the new simulation structure using compiled systems, modifications to either system need to be made in the Simulink source files, as all access to the internal structure of the block is lost once it is compiled to s-function form. However, great care has been taken to simplify the process so that the user is able to make a modification in the source models, then quickly verify that there are no errors by running the modified model. The user can then automatically compile the systems and changes will be propagated to the main TCM simulation where functionality can be verified. It is worth noting that if the input and output structure of the block is changed, the automated process will fail unless additional steps are taken. First, the bus definitions must be updated to reflect the new structure, as do the inputs to the block in the source model. Then if the number of inputs or outputs has changed, the new connections will need to be made (or the excess connections deleted) manually in the source model, the library blocks or storage model, and in the TCM simulation itself. It is then necessary that the user follow the manual compilation procedure (as outlined in Sections 4.1 and 4.2) the first time the block is compiled after changing the number of input or output signals.

Chapter 5

Interface Development

In order to efficiently conduct control design, flight testing, and data acquisition, there are three primary interfaces that need to be considered in the design of the simulator environment: the interface between the simulator components, the interface between the simulator and the pilot, and the interface between the simulator software and the operator. Each of these interfaces must be properly designed in order for the simulator to function effectively.

The interface between the simulator components must be defined so that the communication between each pair of components is compatible. Each element of the simulator is connected to the local area network, over which communication is enabled through UDP. The messages sent over the network are defined by the header files discussed in Section 3.1.1, while message targets are specified by the IPv4 address and port information defined by the *islnetports.h* header file. Messages communicated between two components are always defined by a single data packet header file for that component pair, so proper definition of the pairwise header files ensures congruous information transfer between Simulink and X-Plane, Simulink and the cockpit display, and between the Frasca cockpit controls and Simulink. The same communications strategy is also used for the desktop piloted simulation: pilot commands are defined through a data packet header file to transfer joystick commands from X-Plane to the Simulink model, however communication is done only on the host computer rather than a network.

The interface between the pilot and the simulator is defined by the Frasca cockpit flight controls and by the feedback from both the X-Plane visualization and the cockpit display panel. It is important to construct a cockpit environment where information is presented to the pilot in a clear, intuitive, and veridical manner, and that the control inputs provide functionality with which the pilot is familiar. As the Frasca cockpit controls are similar to those found in real aircraft, pilot familiarity is not a major concern provided the control inputs are mapped appropriately in the simulation. Information about the world environment is generated through the X-Plane graphics and the three projectors to provide a representative viewpoint for the pilot, and aircraft information is presented to the pilot through a flat panel display running the digital instrument panel from Figure 2.6. The instrument panel display has been quite useful for testing conducted to date, however upcoming flight tests require the use of an augmented glass cockpit-style display, which is more representative of cockpit environments found in transport class aircraft. The new panel design will allow simulator testing using new visual cues in an effort to increase pilot situational awareness during aircraft component failure or aircraft upset conditions in order to prevent LoC.

Additionally, a user-friendly interface has been developed to enable efficient use of the TCM simulation software by the simulator operator. This interface provides the user with a convenient environment to setup the simulation in the the desired mode and with user-specified initial conditions, as well as providing the ability to efficiently record and store data during flight testing.

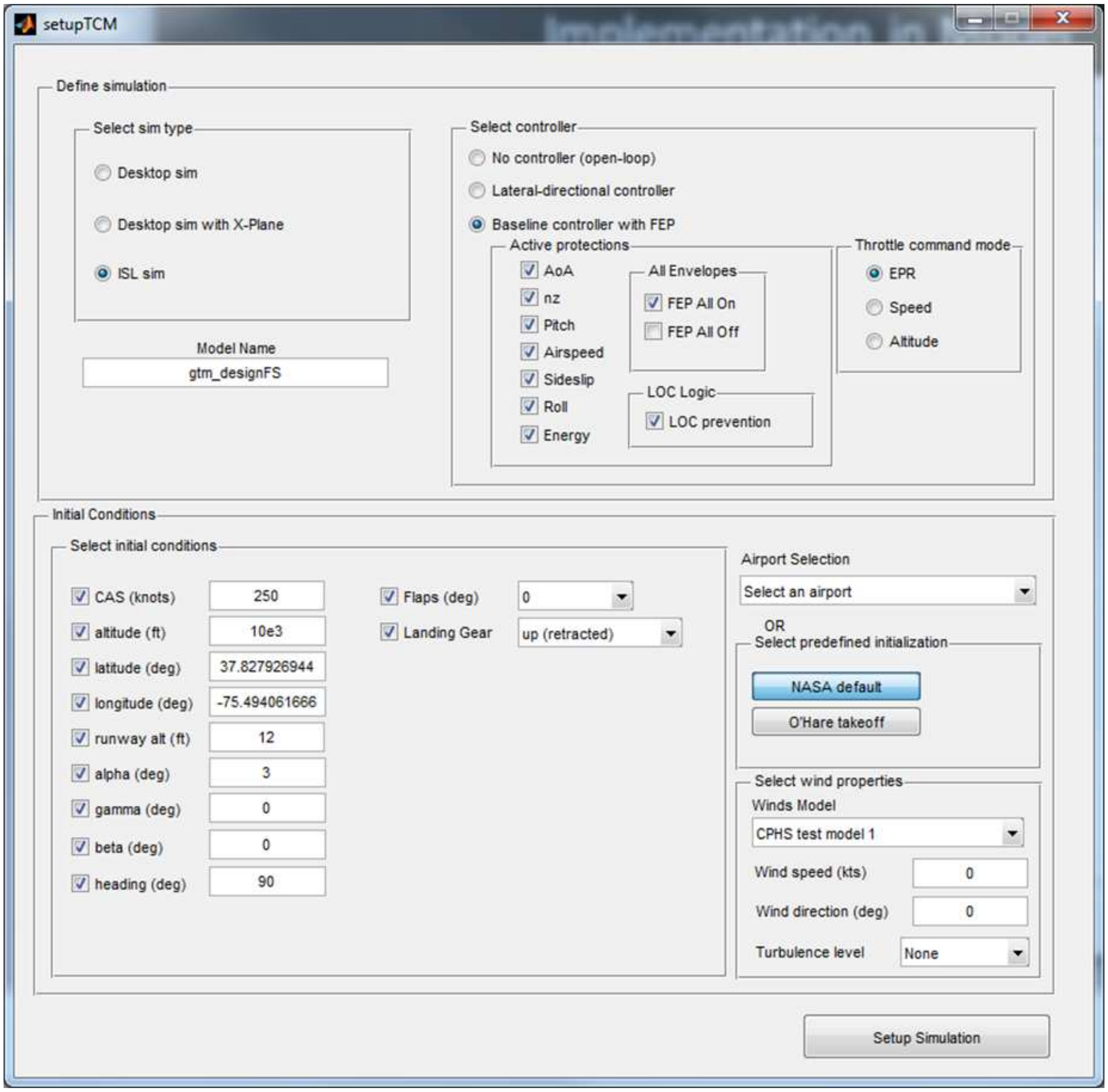
5.1 Simulation User Interface

5.1.1 Simulation Setup

Simulation setup is conducted through the use of a Graphical User Interface (GUI), shown in Figure 5.1, which was built using the Matlab GUIDE software. The various fields of the GUI allow the user to select the simulation mode and controller configuration, Simulink model name, simulation initial conditions, and environmental conditions.

The upper half of the GUI in Figure 5.1 defines the simulation mode and controller configuration. The *Select sim type* option allows the user to choose between batch simulation, desktop piloted simulation, and flight simulator mode by checking the option *Desktop sim*, *Desktop sim with X-Plane*, or *ISL sim*, respectively. The *Model Name* option is used to specify the name of the Simulink model to be used for the simulation, useful if there are multiple simulation models. The *Select Controller* section of the GUI lets the user choose between multiple controller configurations, as well as to select whether flight envelope protection is enabled and which protections are active. The throttle operation mode can also be set to either a pass-through mode to command engine pressure ratio directly, or as a speed or altitude command using the *EPR*, *Speed*, and *Altitude* options the *Throttle command mode* section of the GUI.

Initial conditions of the simulation and environmental parameters are set under the *Initial Conditions* section in the lower half of Figure 5.1. Selecting either of the predefined initial condition sets *NASA default* or *O'Hare takeoff* will change all of the variables to the values defined in the set. Selecting a location through the *Airport Selection* dropdown menu will change the initial position information by setting the appropriate latitude, longitude, and runway altitude. Individual parameters may be altered through the text entry boxes in the *Select initial conditions* portion of the GUI. Environmental conditions are set through the wind speed, direction and turbulence level options in the *Select wind properties* panel. Finally, clicking on the *Setup*


setupTCM

Define simulation

Select sim type
- Desktop sim
- Desktop sim with X-Plane
- [✓ ISL sim]

Model Name: gtm_designFS

Select controller
- No controller (open-loop)
- Lateral-directional controller
- [✓ Baseline controller with FEP]

Active protections
- [✓ AoA]
- [✓ nz]
- [✓ Pitch]
- [✓ Airspeed]
- [✓ Sideslip]
- [✓ Roll]
- [✓ Energy]

All Envelopes
- [✓ FEP All On]
- FEP All Off

LOC Logic
- [✓ LOC prevention]

Throttle command mode
- [✓ EPR]
- Speed
- Altitude

Initial Conditions

Select initial conditions

[✓ CAS (knots)]	250
[✓ altitude (ft)]	10e3
[✓ latitude (deg)]	37.827926944
[✓ longitude (deg)]	-75.494061666
[✓ runway alt (ft)]	12
[✓ alpha (deg)]	3
[✓ gamma (deg)]	0
[✓ beta (deg)]	0
[✓ heading (deg)]	90
[✓ Flaps (deg)]	0
[✓ Landing Gear]	up (retracted)

Airport Selection: Select an airport

OR

Select predefined initialization
- NASA default
- O'Hare takeoff

Select wind properties

Winds Model	CPHS test model 1
Wind speed (kts)	0
Wind direction (deg)	0
Turbulence level	None

Setup Simulation

Figure 5.1: Graphical user interface used to set up and initialize the TCM Simulink model.

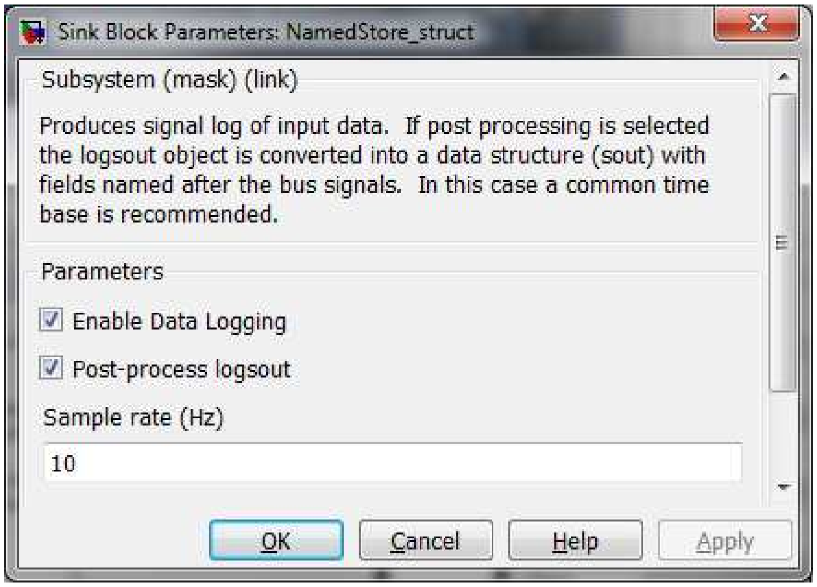
Simulation button will send the user-specified information to the TCM setup script to trim the aircraft and prepare the model for simulation.

The GUI relies on callback functions which are executed based on user actions in the GUI. Each element of the GUI has a callback function that runs anytime a parameter of that element changes, and there are also callbacks that run when the GUI is opened or closed. The GUIDE software autocodes the basic functions of the GUI when it is created, however the callbacks may be manually edited to perform the desired operations. A listing of the manually edited callbacks for the TCM simulation is provided in Listing B.1. Each element of the GUI is an object that can be accessed through a handle, and element parameters can be queried or specified using the *get(·)* and *set(·)* commands. Handle names can be specified in the object properties menu in the GUIDE software.

When the GUI script is first run, the startup callback *setupTCM_OpeningFcn*, lines 2-64 of Listing B.1, is executed to set the default values of the GUI parameters. Setting up default parameter values is done as a shortcut for simulation setup; the default configuration is automatically set to enable the user to immediately setup the simulation if desired. Selecting options within the GUI executes that element's callback function. For example, selecting the *NASA default* option in the initial conditions panel uses the *nasa_def_Callback* function in lines 115-146 of Listing B.1 to set the string parameter of the initial condition text entry fields to the set of prespecified values that starts the simulation off with the aircraft in flight over NASA's Wallops Flight Facility. Other GUI objects that have callbacks that edit handle values of related objects are the *O'Hare takeoff* and *Airport Selection* option, which alter the initial condition values, as well as the elements in the *Active protections* panel, which all interact with the *FEP All On* and *FEP All Off* options.

Selecting the *Setup Simulation* button runs the *setup_sim_Callback* function in lines 68-111 of Listing B.1. The *setup_sim_Callback* function queries the values of the GUI objects to collect the simulation setup information, which is then stored in the data structure *IC*. The structure is then passed to the Matlab base workspace and the TCM initialization script is called. The initial condition information is used to trim the aircraft to the specified state using the NASA-supplied trimming support utility, and the simulation type and controller configuration options are used to specify the link to the appropriate Input block used by the configurable subsystem discussed in Section 3.2. Upon successful completion of the model setup, the GUI is closed and the Simulink model is ready for operation.

The use of a GUI provides an efficient and convenient utility for the simulation operator to setup the simulation for flight testing and for switching between simulation modes. Additional space has been allotted in the GUI for expansion of functionality as the development and testing work progresses.


Sink Block Parameters: NamedStore_struct

Subsystem (mask) (link)

Produces signal log of input data. If post processing is selected the logsout object is converted into a data structure (sout) with fields named after the bus signals. In this case a common time base is recommended.

Parameters

☑ Enable Data Logging

☑ Post-process logsout

Sample rate (Hz)

10

Figure 5.2: Data logging block parameter selection dialog. Data logging in the simulation can be enabled or disabled along with the postprocessing functions, and the data sampling rate can be specified.

5.1.2 Data Acquisition

Data recording and storage from a set of flight tests or simulations for later postprocessing and analysis is a necessary function for the simulation environment, especially if the flight test objectives are to compare multiple simulation runs using different pilots or FCS configurations. The data acquisition process for the TCM simulation is facilitated by the development of a data acquisition interface which is a significant modification of the data logging functionality provided with the original TCM simulation. Data are collected into a single bus structure using the global signal routing tags in the model, then routed into the NamedStore_struct block on the far right of Figure 2.1. The original TCM simulation used Simulink's built in data logging functionality with a postprocessing function to store the information in a data structure in the Matlab workspace. The data structure could then be saved manually if desired, and the logging functionality could be turned on or off as well.

In order to present the flight test data to the simulation operator in an efficient manner for postprocessing after a series of flight tests or batch simulations, it was necessary to modify the existing data collection function. Data recording may be toggled by double-clicking on the NamedStore_struct block and checking or unchecking the *Enable Data Logging* field, as shown in Figure 5.2. The data can be output in its raw form from the Simulink data logging utility if desired by disabling the *Post-process logsout* option, and the data sample rate can also be altered in the NamedStore_struct parameter dialog to increase resolution or to reduce disk memory usage.

The data from a flight test session may be recorded and saved using the Simulink model *Play*, *Pause*, and *Stop* commands. Clicking the *Play* button will start the simulation and start recording data. Data will be recorded until either a *Pause* or *Stop* command is given, or until the simulation stop time has elapsed. Once the simulation is paused or stopped, the output data stream is processed and stored in a structure, then saved in a time-stamped file in the simulation *data* directory. If the simulation was paused and resumed, the output data stream is bookmarked at the time of the pause. Upon the next *Pause* or *Stop* command, the output data stream is again processed, but only the data from the last bookmark is stored and saved in a new time-stamped file. Bookmarking and selective output of the data stream allows the user to control the data output using preexisting Simulink functionality to reduce data postprocessing and analysis time by automatically saving only the desired data into an easily identifiable file. Additionally, it is desirable to be able to modify the output structure that is recorded as testing needs dictate. Therefore, the data recording utility should also be able to parse the data output dynamically.

The data acquisition functionality is obtained through Matlab code implemented as block property callbacks, which may be accessed by right-clicking on a block and selecting *Block Properties*. The initialization callback saves the initial configuration at setup time and stores it in the variable *SetupInfo*. The initialization callback is provided in Listing 5.1. When the simulation is started, the time bookmark flag is set to -1 so that data will be recorded from the start of the simulation. When the simulation is paused or stopped, the pause or stop callback function is invoked to store the data output from the previous bookmark time. The identical pause and stop function callbacks are given in Listing 5.2. The function checks to ensure that data logging and postprocessing are enabled, then processes the data log variable *logsout* using the NASA provided utility *CreatSimOut*. The data structure is then processed through the function *MakeTestOutput*, which strips out data recorded prior to the last bookmark time and reduces the data using the specified sampling rate. The current time is then appended to the standard filename and the modified structure is saved under the time-stamped filename. The source code for the *MakeTestOutput* function and its helper function *getfields*, which dynamically parses a structure and returns an array of variable names that define a new structure to be saved, are provided in Listings C.1 and C.2.

Listing 5.1: Initialization callback for NamedStore_struct block.

```
root_mws = grabmws(bdroot);
SetupInfo = root_mws.SetupInfo;
clear root_mws;
```

Listing 5.2: Pause and Stop function callback for NamedStore_struct block.

```
if strmatch(get_param(gcb,'enable_logging'),'on','exact'),
  if get_param(bdroot,'SimulationTime')>lasttime,
    if strmatch(get_param(gcb,'do_stopfcn_callback'),'on','exact'),
        if exist('logsout','var'),
            sout = CreateSimOut(logsout);
            [fstruct,fname]=MakeTestOutput(sout,lasttime);
        fstruct.SetupInfo = SetupInfo;
            save(fname, 'fstruct');
        end,
      clear logsout;

      %latch current time
      lasttime =get_param(bdroot,'SimulationTime');
    end,
  end,
end
```

Chapter 6

Piloted Testing

Ongoing PITL testing at ISL has been crucial to the development and evaluation of both the simulation environment and the controls design work for the iReCoVeR flight control system. Pilot and simulator operator feedback have been employed to improve the simulator hardware, software, and flight control systems. Feedback regarding functionality and ease of use is critical to developing a simulation environment that is representative of a real aircraft and realistic enough to ensure that meaningful results can be obtained through flight tests. Operator feedback is also important for development of the software utilities so that they enable the operator to conduct testing, data acquisition, and analysis in an efficient and effective manner.

Early flight tests at ISL focused primarily on system shakedown and performance tests, and included control input mapping and command shaping to properly interpret the pilot control inputs from the Frasca cockpit, as well as testing the cockpit instrument panel and visualization operation and verifying communication system operation. Having verified the functionality of the simulator system, testing moved to refinement of system components. Pilot feedback was utilized to refine the control mapping, along with the layout and information content of the instrument panel.

Recent testing work has concentrated on evaluation of the flight control system, specifically the FEP and LoC prediction and prevention systems. FEP and LoC prevention system testing has been conducted using environmental disturbance models which lead to potential aircraft upset and LoC conditions. Environmental disturbance situations tested at ISL include a simulated microburst based on the wind speed profile and flight path of Delta 191 during the microburst leading to the accident at the Dallas-Fort Worth airport in 1985, and also a generalized gust disturbance model that uses a sequence of "$(1 - \cos)$" profiles to generate a wind profile. A typical $(1 - \cos)$ profile can be seen in Figure 6.1, and implementation of the general gust disturbance model is discussed in Section 6.1. Flight testing using the FEP and LoC prevention systems has yielded useful data and has shown benefits for mitigation of aircraft upset situations and for LoC prevention [11]. However, flight tests have highlighted the need for an improved pilot interface.

Automated systems such and the FEP and LoC prediction and prevention systems pose a risk to pilot situational awareness in the cockpit. Actions taken by automated systems may not be noticed or understood

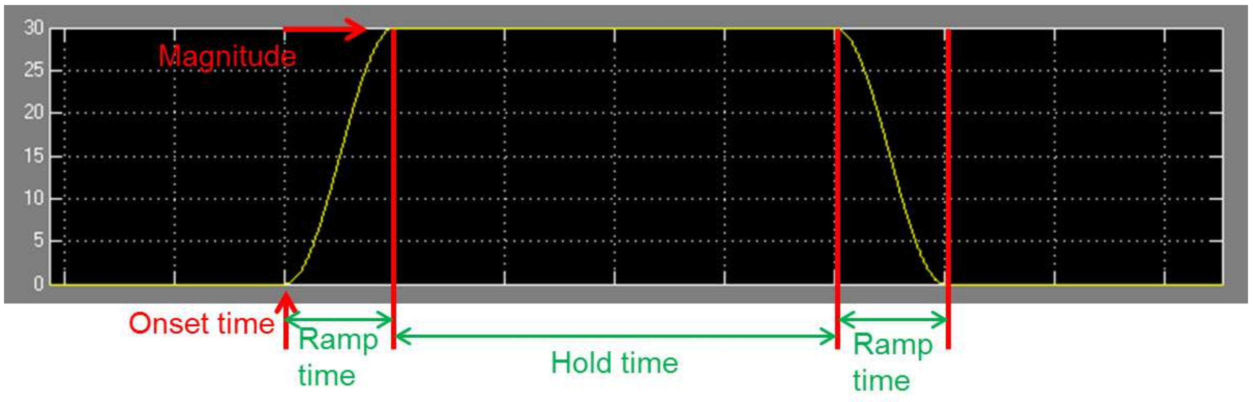


Figure 6.1: "$(1 - \cos)$" gust profile, named for the rise and falloff of the signal which follows the profile $1 - \cos(\omega t)$.

by the pilot, especially if the system is opaque to the pilot or utilizes information not available to the pilot. Whenever automated system actions are not clearly visible there exists the threat that the pilot may not fully understand the state of the aircraft, and therefore may not be able to respond appropriately, unless the actions of the automated systems are communicated clearly. Research into both the technical and human factors elements of cockpit interface design will be a major part of upcoming testing at ISL. Future research work will involve the design and evaluation of a new cockpit display based on the glass cockpit environment found in modern transport class aircraft, augmented with information from the FEP and other automated control systems in an effort to enhance pilot situational awareness to prevent LoC during aircraft upset conditions.

6.1 Environmental Disturbance Modeling

The gust disturbance model implemented in the TCM simulation is built on top of the environmental disturbance model provided with the TCM Simulink model, and is used to generate a wind gust scenario whose profile is defined as a summation of $(1 - \cos)$ profiles. The new gust disturbance model allows for the definition of test scenarios consisting of multiple independent gusts, occurring separately or simultaneously, which are repeatable, flexible, and can be tuned and modified to produce a desired scenario.

A scenario is defined by three sets of gusts: a set of tailwind (TW) gusts which initiate in the direction parallel to the projection of the longitudinal axis of the airplane in the horizontal plane, a set of side gusts (SG) which initiate in the direction perpendicular to the projection of the longitudinal axis of the airplane in the horizontal plane, and a set of downdraft (DD) gusts in the vertical direction with positive direction defined as toward the ground. The sets of gusts are independent and may each have an arbitrary nonnegative number of gusts. Each gust is defined by its onset time T_s, rise time T_r, hold time T_h, wind speed k, and turbulence level l. The meaning of the gust parameters is illustrated in Figure 6.1, except for the turbulence level which is an integer value between 1 and 4 inclusive, with 1 corresponding to no turbulence and 2, 3, and 4 corresponding to light, moderate, and severe turbulence respectively.

The set of scenarios is stored in a data structure, and each scenario is itself a structure composed of three 6-column matrices TW, SG, and DD, and a *gust_heading_hold* parameter which can be 0 or 1. The heading hold parameter specifies whether a wind gust, once initiated in a given direction, will maintain a constant bearing regardless of which direction the aircraft subsequently turns, or whether the gust direction rotates with the aircraft in the horizontal plane. The first column of each matrix is the gust number, used for convenience to keep track of the number of gusts, and the remaining columns are the values of the five parameters respectively. The number of rows in each matrix, denoted n_{TW}, n_{SG}, and n_{DD}, are the number of gusts that will initially impact the aircraft in each direction.

A single $(1 - \cos)$ gust profile is defined as

$$G(t) = g(t, T_s, T_r, k) - g(t, T_s + T_r + T_h, T_r, k), \tag{6.1}$$

where

$$g(t, T_s, T_r, k) = \frac{k}{2}\left[1 - \cos\left(\text{sat}_0^{\pi}\left[(t - T_s)\frac{2\pi}{\text{sat}_{0.01}^{100}\left[2T_r\right]}\right]\right)\right], \tag{6.2}$$

and **sat** is the saturation function. Let G_{TW}, G_{SG}, and G_{DD} be gusts in the body coordinate frame Tail, Side, and Down, and define the column vectors $r_1(\psi) = \begin{pmatrix} \cos\psi & \sin\psi & 0 \end{pmatrix}^{\mathsf{T}}$, $r_2(\psi) = \begin{pmatrix} -\sin\psi & \cos\psi & 0 \end{pmatrix}^{\mathsf{T}}$, and $r_3 = \begin{pmatrix} 0 & 0 & 1 \end{pmatrix}^{\mathsf{T}}$, where ψ is the aircraft heading angle. Now let $R(\psi)$ be the rotation matrix

$$R(\psi) = \begin{pmatrix} r_1(\psi) & r_2(\psi) & r_3 \end{pmatrix}, \tag{6.3}$$

then the i^{th} gust in the inertial North, East, Down frame used by the default TCM environment model is

$$\begin{pmatrix} G_{N_i} \\ G_{E_i} \\ G_{D_i} \end{pmatrix} = R(\psi) \begin{pmatrix} G_{TW_i} \\ G_{SG_i} \\ G_{DD_i} \end{pmatrix}, \tag{6.4}$$

and the total gust in the inertial frame is given by

$$\begin{pmatrix} G_{N_i} \\ G_{E_i} \\ G_{D_i} \end{pmatrix} = \sum_{i=1}^{n_{TW}} r_1(\psi) G_{TW_i} + \sum_{i=1}^{n_{SG}} r_2(\psi) G_{SG_i} + \sum_{i=1}^{n_{DD}} r_3 G_{DD_i}. \tag{6.5}$$

Using the method in Equations 6.1 through 6.5 to define disturbances allows for flexible definition of disturbance scenarios, and can approximate nearly any desired disturbance profile, given enough gust definitions, in a manner similar to series approximation. However, series approximation using this gust generation technique could become tedious due to the manual definition of gust events currently employed. The method described here is therefore better suited for simple or mildly complex profiles. The above method is also not particularly realistic in that the gust direction rotates in the horizontal plane with the aircraft. A more realistic mechanism is to hold the gust heading constant at its onset time. Then each gust will have the desired initial effect, but the individual gust heading will be constant through its duration. Note that this is only a concern in the TW and SG directions, as the downward gusts are not affected by aircraft rotation in the above model.

Define

$$\psi_H \in \mathbb{R}^{n_{TW}+n_{SG}}. \tag{6.6}$$

Then if the *gust_heading_hold* option is set to 1, for each $i \in [1, 2, \ldots, n_{TW} + n_{SG}]$, if gust i is in progress (which can be checked by either $t > T_{s_i}$ or $G_i(t) \neq 0$), let

$$\psi_{H_i} = \psi_{H_i}, \tag{6.7}$$

otherwise let

$$\psi_{H_i} = \psi. \tag{6.8}$$

Then the total gust can be calculated as

$$\begin{pmatrix} G_{N_i} \\ G_{E_i} \\ G_{D_i} \end{pmatrix} = \sum_{i=1}^{n_{TW}} r_1\left(\psi_{H_i}\right) G_{TW_i} + \sum_{i=1}^{n_{SG}} r_2\left(\psi_{H_{n_{TW}+i}}\right) G_{SG_i} + \sum_{i=1}^{n_{DD}} r_3 G_{DD_i}. \tag{6.9}$$

The method using the heading hold option in Equation 6.9 is a more realistic implementation that can be used to generate environmental disturbances used for flight testing. The disturbance model architecture described here provides an environment where multiple scenarios can be created, stored, and imported into the model to provide user-defined repeatable disturbance events.

Associated with each gust is a turbulence level. The turbulence is generated using the Dryden turbulence model, which takes an integer value argument to create white noise turbulence of the intensity specified by the argument, from no turbulence to severe. The overall turbulence level during each scenario is calculated similarly to the wind gust speed. Define the turbulence level for each gust as

$$l_i \in \{1, 2, 3, 4\} \tag{6.10}$$

and the turbulence level for the i^{th} gust as

$$L_i = g(t, T_s, T_r, l_i) - g(t, T_s + T_r + T_h, T_r, l_i). \tag{6.11}$$

Then the overall turbulence level is calculated by

$$L = \left\lfloor \mathrm{sat}_1^4 \left[\max_i \{L_{TW_i}, L_{SG_i}, L_{DD_i}\} \right] \right\rfloor , \tag{6.12}$$

where $\lfloor \cdot \rfloor$ denotes the floor function. Then the turbulence level is scaled with the magnitude of the gust it accompanies and occurs simultaneously with the gust.

Implementation of the disturbance model is done in the Winds block of Figure 2.1. Within the Winds block, the disturbance model sends gusts in the North, East, Down inertial frame and passes the turbulence level variable to the Dryden turbulence model, illustrated in Figure 6.2. The scenario variable is imported through constant-value blocks and passed to a Matlab function block which performs the calculations for the disturbance model.

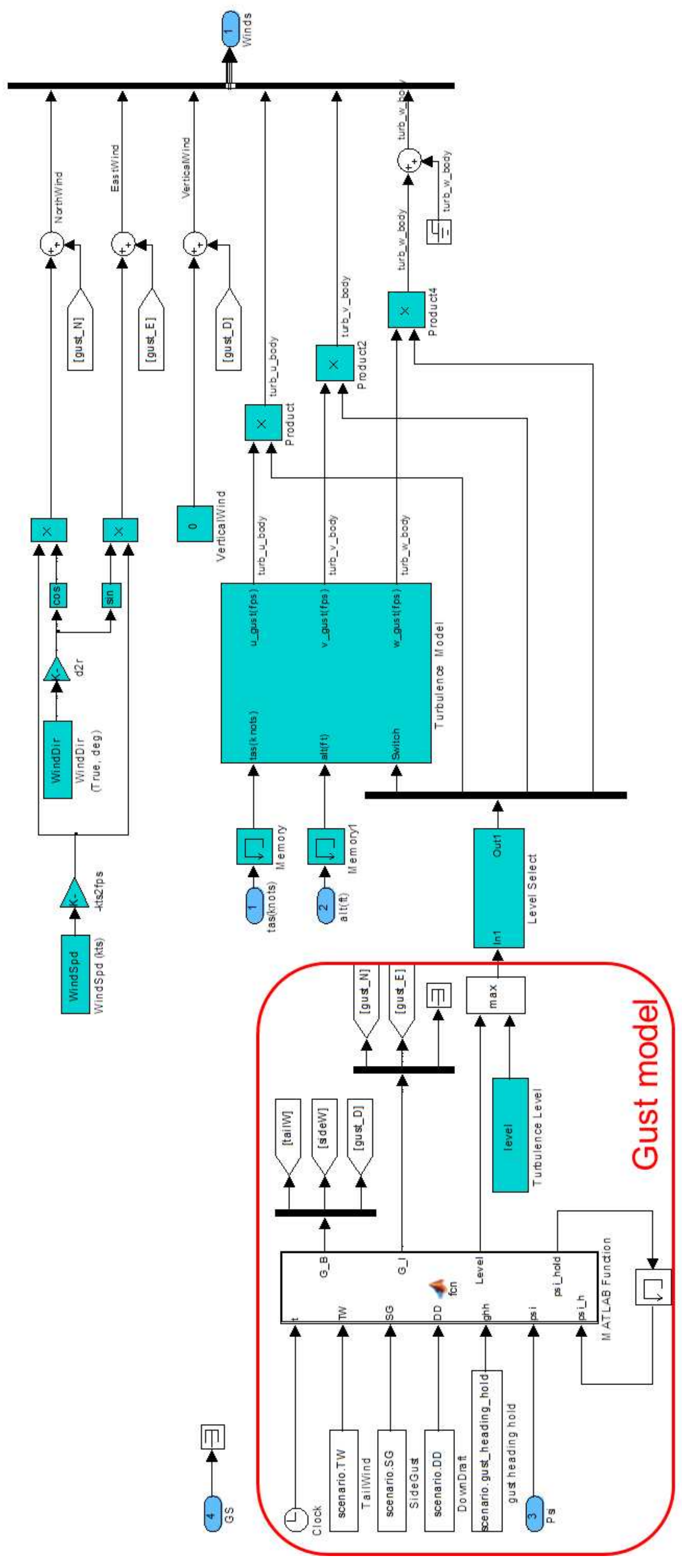


Figure 6.2: Implementation of the general gust disturbance model in the TCM simulation Winds block.

Chapter 7

Conclusion

This thesis has presented the work done to implement NASA's Transport Class Model within a modular flight simulator environment with independent simulator components communicating over a local area network. A simulation framework was developed to enable control design and testing in both a laboratory and flight simulator setting using a single Simulink model. The simulator development was accomplished through the creation of an Input library using a configurable subsystem to switch between simulation modes and flight control law configurations. A graphical user interface was developed to assist the simulation operator with configuring the simulation and setting initial conditions for both flight testing and batch simulations. The thesis has also documented the process of compiling subsystems of the TCM Simulink model to improve the simulation performance. The subsystems are compiled using Matlab's Real-Time Workshop utility, and tools were developed to automate the process of compiling the aircraft dynamics and flight control systems, while preserving the operability of the existing support functions for the TCM. An environmental disturbance model was also designed to allow for the creation of repeatable disturbance scenarios for flight testing, which uses additive gust profiles with optional turbulence to define the overall disturbance.

Initial flight testing at ISL has been beneficial for improvement of the simulator and the software tools presented in this thesis. Testing has also yielded positive results concerning the development of the iReCoVeR flight control system, the current version consisting of a baseline gain-scheduled control and stability augmentation system with flight envelope protection and LoC prediction and prevention systems. Future work at ISL will involve the creation and evaluation of a pilot interface designed to enhance pilot situational awareness during environmental disturbance events, with the goal of preventing LoC events during aircraft upset. Opportunities for improving the simulator environment will continue to be investigated, and the results obtained through the use of the simulator software and through flight testing will translate to expedited development of flight control systems.

Appendix A

Compiled Simulink Block Source Code

Listing A.1: **buildFCS.m**: Matlab script for automated compiling of the flight control system Simulink block.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% build_FCS.m
%
% Generates mex file for TCM FCS developed by Niko Tekles.
%
% Inputs:
%   None
%
% Outputs:
%   None
%
% Builds the FCS block in gtm_designFS_FCS/NavGuidControl with an
% s-function target. Saves masked s-function block and replaces instances
% used by gtm_designFS_FCS and gtm_designFS.
% Replaces mex file in mexfiles directory with new version.
%
% Author: Kasey Ackerman
%         kaacker2@illinois.edu
% Created:  2 Apr 2014
% Modified: 2 Apr 2014 by K.Ackerman
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% We need to clear the data from the workspace, confirm before proceeding
fprintf(1,'Building FCS mex file...\n\nWARNING: this will erase all data from the workspace
    \n\n[y/N]  ');
proceed = input(' ','s');
if (~strcmp(proceed,'y'))&&(~strcmp(proceed,'Y'))
    error('build_FCS_abort','build_FCS function build aborted by user');
```

```
end
clear all;

% store current directory location and change to FCS_src
homedir = pwd;
cd ./FCS_src;

% Run setup file and build FCS block
setupFS_FCS;
rtwbuild('gtm_designFS_FCS/NavGuidControl/FCS'); % _FCS
save_system('untitled','FCS_sfcn');
bdclose('untitled');

% Replace FCS block in model used for compiling
open_system('FCS_sfcn');
replace_block('gtm_designFS_FCS/NavGuidControl', 'Name', 'S-Function', 'FCS_sfcn/FCS','
    noprompt');
save_system('gtm_designFS_FCS');
bdclose('gtm_designFS_FCS');
eval(['cd ',homedir]);

% Replace FCS block in UserContent_lib
open_system('UserContent_lib')
set_param('UserContent_lib','Lock','off')
replace_block('UserContent_lib/FCS', 'Name', 'FCS_sfcn', 'FCS_sfcn/FCS','noprompt');
save_system('UserContent_lib');
bdclose('UserContent_lib');
bdclose('FCS_sfcn');

% Move mex file to main mex file storage directory
archtype  = computer('arch');
processor = archtype((end-1):end);
copyfile(['./FCS_src/FCS_sf.mexw',processor],['./mexfiles/FCS_sf.mexw',processor]);

fprintf(1,'FCS build complete. Please run <setupTCM> to refresh library links.\n\n');
clear all;
return;
```

Listing A.2: **buildDYN.m**: Matlab script for automated compiling of the TCM aircraft dynamics block.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% build_DYN.m
%
% Generates mex file for modified version of TCM dynamics block.
%
% Inputs:
%   None
%
% Outputs:
%   None
%
% Builds the GTM_FS block in gtm_designFS_DYN with an s-function target.
% Saves masked s-function block and replaces instances used by
% gtm_designFS_DYN_sfun_model and gtm_designFS_sfun_model.
% Replaces mex file in mexfiles directory with new version.
%
% Author: Kasey Ackerman
%          kaacker2@illinois.edu
% Created:  2 Apr 2014
% Modified: 2 Apr 2014 by K.Ackerman
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% We need to clear the data from the workspace, confirm before proceeding
fprintf(1,'Building FCS mex file...\n\nWARNING: this will erase all data from the workspace
    \n\n[y/N] ');
proceed = input(' ','s');
if (~strcmp(proceed,'y'))&&(~strcmp(proceed,'Y'))
    error('build_DYN_abort','build_DYN function build aborted by user');
end
clear all;

% store current directory location and change to FCS_src
homedir = pwd;
cd ./DYN_src;

% Run setup file and build GTM_FS block
setupFS_DYN(1);
rtwbuild('dynamics_for_compile/GTM_FS');
```

```
save_system('untitled','GTM_FS_sfcn');
bdclose('untitled');
save_system('dynamics_for_compile');
bdclose('dynamics_for_compile');

% Get the number of parameters in the GTM_FS s-function mask
open_system('GTM_FS_sfcn');
try
    i = 1;
    while 1
        eval(['get_param(''GTM_FS_sfcn/GTM_FS'',''sfcnParam',num2str(i),''');']);
        i = i+1;
    end
catch
    num_params = i-1;
end

% These are the parameters that had to be changed to allow these variables
% to be inlined
%                 replace this        with this
params2change = {'Act_AilPL',         'Act.AilPL';...
                 'Act_ElevLL',        'Act.ElevLL';...
                 'Act_ElevUL',        'Act.ElevUL';...
                 'Act_RudPL',         'Act.RudPL';...
                 'Act_SplPL',         'Act.SplPL';...
                 'bias_N1',           'bias.N1';...
                 'bias_TRA',          'bias.TRA';...
                 'bias_aileron',      'bias.aileron';...
                 'bias_elevator',     'bias.elevator';...
                 'bias_flaps',        'bias.flaps';...
                 'bias_geardown',     'bias.geardown';...
                 'bias_rudder',       'bias.rudder';...
                 'bias_speedbrake',   'bias.speedbrake';...
                 'bias_stabilizer',   'bias.stabilizer';...
                 'bias_throttle',     'bias.throttle'};

% Change the values of the parameters to their original version (right
% column above)
for i = 1:num_params
    prm = eval(['get_param(''GTM_FS_sfcn/GTM_FS'',''sfcnParam',num2str(i),''');']);
    for j = 1:size(params2change,1)
        if strcmp(prm, params2change{j,1})
            eval(['set_param(''GTM_FS_sfcn/GTM_FS'',''sfcnParam',num2str(i),''',
                params2change{j,2});'])
        end
    end
end
save_system('GTM_FS_sfcn');

% Replace GTM_FS block in plant model storage: gtm_designFS_sfun_model
eval(['cd ',homedir]);
addpath('./mdls');
rehash path;
open_system('gtm_designFS_sfun_model');
replace_block('gtm_designFS_sfun_model/GTM_FS_PLNT_SFUN', 'Name', 'GTM_FS', 'GTM_FS_sfcn/
    GTM_FS','noprompt');
save_system('gtm_designFS_sfun_model');
bdclose('gtm_designFS_sfun_model');
bdclose('GTM_FS_sfcn');

% Move mex file to main mex file storage directory
archtype  = computer('arch');
processor = archtype((end-1):end);
copyfile(['./DYN_src/GTM_FS_sf.mexw',processor],['./mexfiles/GTM_FS_sf.mexw',processor]);

fprintf(1,'DYN build complete. Please run <setupTCM> to import compiled dynamics into TCM
    simulation.\n\n');
clear all;
return;
```

Appendix B

TCM Setup GUI Callback Source Code

Listing B.1: Modified callback functions for TCM setup graphical user interface. This is not a complete listing of the callback code, only the modified functions have been listed.

```
% --- Executes just before setupTCM is made visible.
function setupTCM_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to setupTCM (see VARARGIN)

% Startup commands
% Set initial values for the GUI
set(handles.cas_select,'Value',1);
set(handles.alt_select,'Value',1);
set(handles.lat_select,'Value',1);
set(handles.lon_select,'Value',1);
set(handles.runwayalt_select,'Value',1);
set(handles.alpha_select,'Value',1);
set(handles.gamma_select,'Value',1);
set(handles.beta_select,'Value',1);
set(handles.heading_select,'Value',1);
set(handles.flaps_select,'Value',1);
set(handles.gear_select,'Value',1);

set(handles.isl_sim,'Value',1);
set(handles.nasa_def,'Value',1);
set(handles.ohare_def,'Value',0);
set(handles.desktop_sim,'Value',0);
set(handles.desktopxp_sim,'Value',0);
set(handles.isl_sim,'Value',1);
set(handles.model_name,'String','gtm_designFS');
set(handles.fepcon,'Value',1);
set(handles.aoafep,'Value',1);
set(handles.nzfep,'Value',1);
set(handles.thetafep,'Value',1);
set(handles.vfep,'Value',1);
set(handles.betafep,'Value',1);
set(handles.phifep,'Value',1);
set(handles.efep,'Value',1);
set(handles.throtIsEPR,'Value',1);
set(handles.locprev,'Value',1);
set(handles.FEPAllOn,'Value',1);
set(handles.FEPAllOff,'Value',0);

set(handles.airport,'Value',1);
set(handles.cas_input,'String','250');
set(handles.alt_input,'String','10e3');
set(handles.lat_input,'String','37.827926944');
set(handles.lon_input,'String','-75.494061666');
set(handles.runwayalt_input,'String','12');
set(handles.alpha_input,'String','3');
set(handles.gamma_input,'String','0');
set(handles.beta_input,'String','0');
set(handles.heading_input,'String','90');
set(handles.flaps_input,'Value',1);
set(handles.gear_input,'Value',1);
set(handles.windSpd,'String','0');
set(handles.windDir,'String','0');
set(handles.turbLev,'Value',1);
set(handles.windsModel,'Value',1);

% Choose default command line output for setupTCM
handles.output = hObject;
% Update handles structure
guidata(hObject, handles);


% --- Executes on button press in setup_sim.
function setup_sim_Callback(hObject, eventdata, handles)
% hObject    handle to setup_sim (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Save user input to base workspace
IC.mdl_name = get(handles.model_name,'String');
IC.cas = str2num(get(handles.cas_input,'String'));
IC.alt = str2num(get(handles.alt_input,'String'));
IC.lat = str2num(get(handles.lat_input,'String'));
IC.lon = str2num(get(handles.lon_input,'String'));
IC.runwayalt = str2num(get(handles.runwayalt_input,'String'));
IC.alpha = str2num(get(handles.alpha_input,'String'));
IC.gamma = str2num(get(handles.gamma_input,'String'));
IC.beta = str2num(get(handles.beta_input,'String'));
IC.heading = str2num(get(handles.heading_input,'String'));
flaps_array = get(handles.flaps_input,'String');
IC.init_flaps_pos = get(handles.flaps_input,'Value');
IC.flaps = str2num(flaps_array{IC.init_flaps_pos});
IC.gear = get(handles.gear_input,'Value')-1;

IC.simtype = find( [ get(handles.desktop_sim,'Value'); get(handles.desktopxp_sim,'Value');
    get(handles.isl_sim,'Value')] );
IC.contype = find( [ get(handles.OLCon,'Value'); get(handles.latCon,'Value'); get(handles.
    fepcon,'Value') ] );

IC.FEP.aoa = get(handles.aoafep,'Value');
IC.FEP.nz = get(handles.nzfep,'Value');
IC.FEP.theta = get(handles.thetafep,'Value');
IC.FEP.v = get(handles.vfep,'Value');
IC.FEP.beta = get(handles.betafep,'Value');
IC.FEP.phi = get(handles.phifep,'Value');
IC.FEP.e = get(handles.efep,'Value');
IC.speedcontrol = find( [get(handles.throtIsEPR,'Value'), get(handles.throtIsAlt,'Value'),
    get(handles.throtIsSpd,'Value')] )-1;
IC.LOClogic = get(handles.locprev,'Value');

IC.windSpd = str2num(get(handles.windSpd,'String'));
IC.windDir = str2num(get(handles.windDir,'String'));
IC.turbLev = get(handles.turbLev,'Value');
IC.windsModel = get(handles.windsModel,'Value');

```

```matlab
assignin('base','IC',IC);

% Setup the model and close the GUI
TCM_init_fnc;
close(handles.TCMgui)


% --- Executes on button press in nasa_def.
function nasa_def_Callback(hObject, eventdata, handles)
% hObject    handle to nasa_def (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% Hint: get(hObject,'Value') returns toggle state of nasa_def

% Change the initial conditions to the NASA default values
set(handles.airport,'Value',2);

set(handles.cas_select,'Value',1);
set(handles.alt_select,'Value',1);
set(handles.lat_select,'Value',1);
set(handles.lon_select,'Value',1);
set(handles.runwayalt_select,'Value',1);
set(handles.alpha_select,'Value',1);
set(handles.gamma_select,'Value',1);
set(handles.beta_select,'Value',1);
set(handles.heading_select,'Value',1);
set(handles.flaps_select,'Value',1);
set(handles.gear_select,'Value',1);

set(handles.cas_input,'String','250');
set(handles.alt_input,'String','10e3');
set(handles.lat_input,'String','37.827926944');
set(handles.lon_input,'String','-75.494061666');
set(handles.runwayalt_input,'String','12');
set(handles.alpha_input,'String','3');
set(handles.gamma_input,'String','0');
set(handles.beta_input,'String','0');
set(handles.heading_input,'String','90')
set(handles.flaps_input,'Value',1);
set(handles.gear_input,'Value',1);
```

```
% --- Executes on button press in ohare_def.
function ohare_def_Callback(hObject, eventdata, handles)
% hObject    handle to ohare_def (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% Hint: get(hObject,'Value') returns toggle state of ohare_def

% Change the initial conditions to setup for a takeoff from O'Hare
% International Airport
set(handles.airport,'Value',3);

set(handles.cas_select,'Value',1);
set(handles.alt_select,'Value',1);
set(handles.lat_select,'Value',1);
set(handles.lon_select,'Value',1);
set(handles.runwayalt_select,'Value',1);
set(handles.alpha_select,'Value',1);
set(handles.gamma_select,'Value',1);
set(handles.beta_select,'Value',1);
set(handles.heading_select,'Value',1);
set(handles.OLCon,'Value',1);
set(handles.flaps_select,'Value',1);
set(handles.gear_select,'Value',1);

set(handles.cas_input,'String','0');
set(handles.alt_input,'String','667');
set(handles.lat_input,'String','41.9691');
set(handles.lon_input,'String','-87.9253');
set(handles.runwayalt_input,'String','667');
set(handles.alpha_input,'String','0');
set(handles.gamma_input,'String','0');
set(handles.beta_input,'String','0');
set(handles.heading_input,'String','90');
set(handles.flaps_input,'Value',4);
set(handles.gear_input,'Value',2);


% --- Executes on button press in aoafep.
```

```
function aoafep_Callback(hObject, eventdata, handles)
% hObject    handle to aoafep (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% Hint: get(hObject,'Value') returns toggle state of aoafep
set(handles.FEPAllOn,'Value',0);
set(handles.FEPAllOff,'Value',0);


% --- Executes on button press in thetafep.
function thetafep_Callback(hObject, eventdata, handles)
% hObject    handle to thetafep (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% Hint: get(hObject,'Value') returns toggle state of thetafep
set(handles.FEPAllOn,'Value',0);
set(handles.FEPAllOff,'Value',0);


% --- Executes on button press in vfep.
function vfep_Callback(hObject, eventdata, handles)
% hObject    handle to vfep (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% Hint: get(hObject,'Value') returns toggle state of vfep
set(handles.FEPAllOn,'Value',0);
set(handles.FEPAllOff,'Value',0);


% --- Executes on button press in phifep.
function phifep_Callback(hObject, eventdata, handles)
% hObject    handle to phifep (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% Hint: get(hObject,'Value') returns toggle state of phifep
set(handles.FEPAllOn,'Value',0);
set(handles.FEPAllOff,'Value',0);


% --- Executes on button press in efep.
```

```
function efep_Callback(hObject, eventdata, handles)
% hObject    handle to efep (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% Hint: get(hObject,'Value') returns toggle state of efep
set(handles.FEPAllOn,'Value',0);
set(handles.FEPAllOff,'Value',0);


% --- Executes on button press in betafep.
function betafep_Callback(hObject, eventdata, handles)
% hObject    handle to betafep (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% Hint: get(hObject,'Value') returns toggle state of betafep
set(handles.FEPAllOn,'Value',0);
set(handles.FEPAllOff,'Value',0);


% --- Executes on button press in locprev.
function locprev_Callback(hObject, eventdata, handles)
% hObject    handle to locprev (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% Hint: get(hObject,'Value') returns toggle state of locprev
set(handles.FEPAllOn,'Value',0);
set(handles.FEPAllOff,'Value',0);


% --- Executes on selection change in airport.
function airport_Callback(hObject, eventdata, handles)
% hObject    handle to airport (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% Hints: contents = cellstr(get(hObject,'String')) returns airport contents as cell array
%        contents{get(hObject,'Value')} returns selected item from airport

% Change the location and runway information to match the selected airport
switch get(handles.airport,'Value')
    case 2 % Wallops
```

```
        lat          = '37.827926944'; % deg
        lon          = '-75.494061666'; % deg
        heading      = '90';
        runwayAlt    = '12';
    case 3 % O'Hare
        lat          =  '41.9691';
        lon          = '-87.9253';
        heading      = '90';
        runwayAlt    = '667';
    otherwise
        lat          = '0';
        lon          = '0';
        runwayAlt    = '0';
        heading      = '0';
end
if get(handles.airport,'Value')>=2
    set(handles.lat_input,'String',lat);
    set(handles.lon_input,'String',lon);
    set(handles.runwayalt_input,'String',runwayAlt);
    set(handles.heading_input,'String',heading);
end


% --- Executes on button press in FEPAllOn.
function FEPAllOn_Callback(hObject, eventdata, handles)
% hObject    handle to FEPAllOn (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% Hint: get(hObject,'Value') returns toggle state of FEPAllOn

% Turn on protection for all flight envelopes
if get(handles.FEPAllOn,'Value')
    set(handles.aoafep,'Value',1);
    set(handles.nzfep,'Value',1);
    set(handles.thetafep,'Value',1);
    set(handles.vfep,'Value',1);
    set(handles.betafep,'Value',1);
    set(handles.phifep,'Value',1);
    set(handles.efep,'Value',1);
    set(handles.locprev,'Value',1);
    set(handles.FEPAllOff,'Value',0);
end


% --- Executes on button press in FEPAllOff.
function FEPAllOff_Callback(hObject, eventdata, handles)
% hObject    handle to FEPAllOff (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% Hint: get(hObject,'Value') returns toggle state of FEPAllOff

% Turn off protection for all flight envelopes
if get(handles.FEPAllOff,'Value')
    set(handles.aoafep,'Value',0);
    set(handles.nzfep,'Value',0);
    set(handles.thetafep,'Value',0);
    set(handles.vfep,'Value',0);
    set(handles.betafep,'Value',0);
    set(handles.phifep,'Value',0);
    set(handles.efep,'Value',0);
    set(handles.locprev,'Value',0);
    set(handles.FEPAllOn,'Value',0);
end
```

Appendix C

Data Acquisition Function Source Code

Listing C.1: **MakeTestOutput** function. This function processes a data structure to remove all data before a flagged time, and resamples the data at a user-specified rate.

```
function [sout,filename] = MakeTestOutput(sout,lasttime)
% Kasey Ackerman <kaacker2@illinois.edu>
% Advanced Controls Research Laboratory
% University of Illinois at Urbana-Champaign

ctime = clock;
filename = sprintf('data/sout_%02.f%02.f%02.f_%02.f%02.f%02.f',...
                    ctime(1),ctime(2),ctime(3),ctime(4),ctime(5),ctime(6));
fn = getfields(sout,'sout',[]);

names = [];
for ii=1:length(fn)
    if strcmp('struct',class(eval(char(fn(ii)))))
        names = getfields(eval(char(fn(ii))),char(fn(ii)),names);
    end
end
names2=names;
purgevals = [];
for ii=1:length(names)
    if strcmp('struct',class(eval(char(names(ii)))))
        names2 = getfields(eval(char(names(ii))),char(names(ii)),names2);
        purgevals = [purgevals;ii];
    end
end

names2(purgevals)=[];
timeloc = find(sout.clock.time>lasttime,1);

for ii = 1:length(names2)
```

```
    eval([char(names2(ii)),'=',char(names2(ii)),'(',num2str(timeloc),':end);']);
end
```

Listing C.2: **getfields** function. Helper function to **MakeTestOutput**. This function processes a data structure to obtain the structure field names and return an array of names that define a new structure with the same fields.

```
function [sout,filename] = MakeTestOutput(sout,lasttime)
function names = getfields(obj,objname,names)
% Kasey Ackerman <kaacker2@illinois.edu>
% Advanced Controls Research Laboratory
% University of Illinois at Urbana-Champaign

fn = fieldnames(obj);

for ii=1:length(fn)
namestemp{ii} = sprintf('%s.%s',objname,char(fn(ii)));
end
names = [names;namestemp'];
```

References

[1] University of Illinois at Urbana-Champaign, “Beckman Institute Illinois Simulator Laboratory,” http://www.isl.uiuc.edu/.

[2] The Mathworks, Inc., “Matlab,” http://www.mathworks.com/products/matlab/.

[3] The Mathworks, Inc., “Simulink,” http://www.mathworks.com/products/simulink/.

[4] National Aeronautics and Space Administration, “Langley Research Center,” http://www.nasa.gov/centers/langley/home/.

[5] Frasca International, “Frasca flight simulation,” http://www.frasca.com/.

[6] Laminar Research, “X-plane,” http://www.x-plane.com/.

[7] S. Pelech, “Integration of the GTM T2 model into a full sized simulator for human-in-the-loop testing,” M.S. thesis, University of Illinois at Urbana-Champaign, May 2013.

[8] R. M. Hueschen, “Development of the Transport Class Model (TCM) aircraft simulation from a sub-scale Generic Transport Model (GTM) simulation,” NASA, Technical Memorandum 217169, August 2011.

[9] N. Tekles, E. Xargay, R. Choe, N. Hovakimyan, I. M. Gregory, and F. Holzapfel, “Flight envelope protection for NASA’s Transport Class Model,” in *AIAA Guidance, Navigation and Control Conference*, National Harbor, MD, January 2014.

[10] N. Tekles, “Flight envelope protection, loss-of-control prevention, and upset recovery systems for NASA’s Transport Class Model,” M.S. thesis, Technical University of Munich, March 2014.

[11] J. Chongvisal, N. Tekles, E. Xargay, D. Talleur, A. Kirlik, and N. Hovakimyan, “Loss-of-control prediction and prevention for NASA’s Transport Class Model,” in *AIAA Guidance, Navigation and Control Conference*, National Harbor, MD, January 2014.