POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Model-Based Design of a Generic Automatic Flight Control System for Helicopter Flight Simulation Training Devices

Tesi di Laurea Magistrale in Aeronautical Engineering
Ingegneria Industriale e dell'Informazione

Author: **Francesco Paolo De Simone**

Student ID: 927623
Advisor: Prof. Marco Lovera
Co-Advisor: Eng. Francesco Borgatelli
Academic Year: 2021-2022

Model-Based Design of a Generic Automatic Flight Control System for Helicopter Flight Simulation Training Devices.

Francesco Paolo De Simone. Master of Science Thesis in Aeronautical Engineering, Politecnico di Milano.

Abstract

Flight simulation technology has come a long way in recent years, and the field continues to develop at a rapid pace. As a provider of innovative software solutions for aviation training, *TXT e-solutions S.p.A.* is committed to incorporating the latest high-tech innovations into their products. One area of interest for the company is the use of Model-Based Design in the development of flight simulators, a growing field with ongoing research aimed at improving accuracy and reducing costs, development time, and maintenance time.

To investigate the potential of Model-Based Design, the company asked to analyze and improve a pre-existing model of a generic automatic flight control system. The technical methodologies and implementation issues that arose during this investigation are presented in this research work. The company provided input requirements for the model development, which are all analyzed and verified in this thesis.

The main outcome produced is a Simulink® model of a generic automatic flight control system that includes the most important autopilot components and their control logics. These components include Stability Augmentation System Mode (SAS), Attitude Hold Mode (ATT), Turn Coordinator Functionality (TC), Indicated Airspeed Hold Mode (IAS), Heading Hold Mode (HDG), Barometric Altitude Hold Mode (ALT), Radar Height Hold Mode (RHT), and Hover Hold Mode (HOV). An external auto-tuning Matlab® script is also produced to identify the optimal controllers parameters for the autopilot control system, based on specific project requirements in input. Additionally, the model was designed with high levels of generality and customisation, allowing quickly adaptation of the autopilot parameters and characteristics to the specific helicopter. The performances of the automatic flight control system were tested and validated against that achievable with a certified full flight simulator level D. Therefore, the results obtained demonstrate that the Model-Based Design approach is reliable even when dealing with more complex autopilot structures as the one presented in this work.

Sommario

La tecnologia della simulazione di volo ha fatto passi da gigante negli ultimi anni e il settore continua a svilupparsi a ritmo sostenuto. In qualità di fornitore di soluzioni software innovative per l'addestramento aeronautico, *TXT e-solutions S.p.A.* si impegna a incorporare le ultime innovazioni high-tech nei propri prodotti. Un'area di interesse per l'azienda è l'uso del Model-Based Design nello sviluppo dei simulatori di volo, un campo in crescita con ricerche continue volte a migliorare l'accuratezza e a ridurre i costi, i tempi di sviluppo e di manutenzione.

Per indagare le potenzialità del Model-Based Design, l'azienda ha chiesto di analizzare e migliorare un modello preesistente di un generico sistema di controllo automatico del volo. Le metodologie tecniche e i problemi di implementazione emersi durante lo studio sono presentati in questo lavoro di ricerca. L'azienda ha fornito alcuni requisiti in input per lo sviluppo del modello, i quali sono stati tutti analizzati e verificati in questa tesi.

Il risultato principale prodotto in questo lavoro è un modello Simulink® che include i componenti più importanti dell'autopilota e le loro logiche di controllo. Questi componenti includono lo Stability Augmentation System Mode (SAS), l'Attitude Hold Mode (ATT), il Turn Coordinator Functionality (TC), l' Indicated Airspeed Hold Mode (IAS), l' Heading Hold Mode (HDG), il Barometric Altitude Hold Mode (ALT), il Radar Height Hold Mode (RHT) e l' Hover Hold Mode (HOV). È stato prodotto anche uno script esterno di autotuning in ambiente Matlab®, al fine di identificare i parametri ottimali dei controllori del sistema di controllo dell'autopilota, sulla base di specifici requisiti di progetto in ingresso. Inoltre, il modello è stato progettato con alti livelli di generalità e personalizzazione, consentendo un rapido adattamento dei parametri e delle caratteristiche dell'autopilota a seconda dello specifico elicottero. Le prestazioni del sistema di controllo automatico di volo sono state testate e convalidate rispetto a quelle ottenibili con un simulatore di volo certificato full flight di livello D. Pertanto, i risultati ottenuti dimostrano che l'approccio Model-Based Design è affidabile anche quando ci si occupa di strutture di autopilota più complesse come quella presentata nel presente lavoro.

Contents

List of Figures

List of Tables

List of Abbreviations

AFCS	Automatic Flight Control System
AP	Autopilot
APCP	Autopilot Control Panel
ALT	Barometric Altitude Hold Mode
ASE	Automatic Stabilization Equipment
ATT	Attitude Hold System
CG	Center of Gravity
DOF	Degree of Freedom
EASA	European Union Aviation Safety Agency
FAA	Federal Aviation Administration
FSTD	Flight Simulator Training Device
FTR	Force Trim Release
GUI	Graphical User Interface
HDG	Heading Hold Mode
HOV	Hover Hold Mode
IAS	Indicated Airspeed Hold Mode
LHP	left-half plane
LHS	left-hand side
LTI	Linear Time Invariant
MBC	Multi Blade Coordinate
MIMO	Multiple Input Multiple Output
NED	North-East-Down
PI	Proportional–Integral Controller
PID	Proportional–Integral–Derivative Controller
RHP	right-half plane
RHT	Radar Height Hold Mode
SAS	Stability Augmentation System
SISO	Single Input Single Output
TC	Turn Coordinator
TF	Transfer Function
VNE	Velocity Never Exceed

Introduction and Thesis Objectives

According to the *Boeing Pilot & Technician Outlook 2020-2039*, the civil aviation industry will need to hire approximately 2.4 million new pilots and technicians by 2039 to meet the growing demand for air travel.

Flight simulators can play a crucial role in addressing this shortage by preparing pilots for the complexities of flying, both in civil and military applications. As aviation poses high levels of risk, the use of flight simulators allows pilots to practice and refine their skills in a safe and controlled environment. The synthetic environment is used during the initial phases of training for familiarization with the aircraft and then maintains its importance in subsequent phases, culminating in training for mission-critical and safety-critical procedures. Although simulation devices have different levels of complexity, realism, and fidelity compared to the real aircraft, they are linked by a common set of standards and requirements coming from the designated authorities (FAA and EASA). This suitability is also demonstrated through objective tests that link the behavior of the simulated aircraft within the synthetic environment with the real aircraft performances by comparing flight maneuvers accomplished in the simulator with the same maneuvers recorded in flight. By redacting the Qualification test guide (QTG), it is possible to demonstrate that the performance and handling qualities of a Flight Simulation Training Devices (FSTDs) are effectively representative of the specific craft, within prescribed limits imposed by the certification body.

Given the growing importance of this branch of the aviation industry, *TXT e-solutions S.p.A.*, a provider of engineering software solutions, has been at the forefront of this technology for years. The company has consistently pursued advancements in the latest flight simulation innovations, striving for continuous improvement. In the pursuit of replace traditional design strategies of flight simulators, *TXT e-solutions S.p.A.* has shown interest in the use of Model-Based Design throughout the development of flight simulators.

Indeed, most helicopter flight simulators of major aircraft manufacturers such as *Leonardo S.p.A.* or *Airbus S.a.S.* are realised using the C++ programming language. Therefore, for decades hand-coding approach has been the mainstream in industry being a versatile and

effective solution, allowing for high levels of modularity and reusability of recurring parts of code. However, the greatest disadvantage lies in the lack of a form of visual communication that enables the reader to automatically match the objects reproduced in the code with the physical components, which can be a problem in the case of particularly complex models, for which traceability can be difficult. To tackle this issue, Model-Based Design approach relies on a visual and intuitive modeling interface in which physical components are represented with graphical objects whose parameters and configuration properties are accessible and editable in a more user friendly environment enabling not only expert software developers to create models and design systems. However, this is just the most obvious benefit of this approach with respect to the traditional way of proceeding, as it not only preserve modularity and reusability features, but it also allows to save money enabling early verification of designs and thus potential error detection. Indeed, one of the main rule of thumb in projects development is that the earlier you step-back in a project the less it cost. Finally, as the purpose is still to produce a C++ code to integrate on an embedded processor, Model-Based Design provides the capability to automatically generate it.

Therefore, to investigate the potential of Model-Based Design, *TXT e-solutions S.p.A.* aim to realise a single generic helicopter automatic flight control system. The goal is to replace the traditional development workflow which address a specific autopilot design for each specific helicopter flight simulator in analysis. Instead, the pursuit of the company and thus the scope of this thesis, is to build a modular and customisable autopilot architecture which have to embrace common operating characteristics from different manufacturers and hence from different autopilot designs. The final structure of the automatic flight control system must be able to simulate a wide variety of helicopters by simply changing configurations and parameters according to the specific application.

Given this goal, a proof of concept of a basic generic helicopter autopilot provided of low level functionalities was developed in a previous work [32]. Therefore, in the present research, the company asked to analyze and improve this model in order to demonstrate that the Model-Based design approach is still valid even when dealing with more realistic and complex autopilot architectures. In particular, the improvement provided to the model must satisfy the requirements summarized below:

1. The model should be developed in Simulink®.
2. The current state of the generic autopilot should be studied and analyzed to adopt a design philosophy in line with the existing one.
3. The suitability of the model for C++ code generation by means of Simulink Em-

bedded Coder® should be preserved.

4. The current model should be improved with the autopilot upper modes and their control logics, along with a tuning methodology for their controllers.
5. The current level of modularity should be improved by developing an autopilot model in which each individual customized block fulfills a single operating function.
6. The level of generality of the autopilot model should be improved so that it can be easily adapted to different types of helicopters.
7. The results should be validated by comparing them with those of a full flight simulator level D.

To achieve these goals, several steps were necessary, which in turn were recorded and transcribed within the Chapters of this thesis according to the structure shown in the following:

- Chapter 1, based on private *TXT e-solutions* documents and from information gathered from aviation literature, provides important technical notions regarding two main topics. The first aims to briefly present which are the typical helicopter flight control inputs enabling the pilot to achieve and maintain manually controlled aerodynamic flight. In the second topic, instead, the attention is shifted to the description of a generic 4-axis automatic flight control system and its structure.
- Chapter 2 examines the principal Simulink® tools employed throughout the *Model-Based Design* of the generic autopilot. Emphasis is placed on the description of practical implementation procedures to be adopted in Simulink® in order to model systems with high levels of modularity, customisability and automation.
- Chapter 3 provides a detailed analysis of a collection of modular Simulink® blocks created from scratch throughout this work in order to ensemble the whole autopilot structure. Each block models a single operating autopilot function which is typically reused multiple times within different autopilot subsystems. To this purpose, emphasis is given to the implementation issues to address in order to achieve a customisation level that allows to reuse the blocks for different autopilot modes.
- Chapter 4 illustrates the customisable auto-tuning process developed to optimize the control systems gains of the assembled model of the generic autopilot. A twin-engine light utility helicopter dynamics is used as case-study model to test the tuning workflow.
- Chapter 5 outlines three different verification tests carried out to validate the whole

autopilot modeling, either from control logics and phisical modeling point of views as well as to demonstrate the tuning process effectiveness by comparisons with a certified full flight simulator level D.

1 | Flight Control System and Automatic Flight Control System

The first Section of this Chapter provides a brief description of the three essential flight controls that the pilot uses to maneuver a helicopter. In the second Section, the focus shifts to the structure of the automatic flight control system, which provides the pilot with additional assistance during flight.

The information contained in this Chapter are mainly taken from private *TXT e-solutions* documents and from notions gathered from aviation literature. Since the goal of this thesis is to fulfill the research requirements imposed at the beginning, the components described in this Chapter, either from the flight control system or from the autopilot, are restricted to the only necessary systems which are intended to be modeled in the automatic flight control system or that may be important for the understanding of this work.

1.1 Flight Control System

Fixed-wing aircraft and rotary-wing aircraft constitute two distinct types of flight vehicles that exhibit several differences, especially in their flight control surfaces. In the following, a brief description of the history of the helicopter flight control systems is provided.

The *Cierva* autogyro, developed in 1923, was the first rotary-wing aircraft capable of achieving significant forward velocities [34]. It consisted of a conventional airplane modified with rotating wings. Since it was an autogyro, its blades did not require power to rotate, and the pilot had no control over them. Consequently, the aircraft was operated using conventional airplane surfaces. One of the main challenges faced by rotary-wing inventors was the asymmetrical velocity field in the rotor during forward flight. This phenomenon is illustrated in Figure 1.1 for a rotor with anticlockwise rotation.

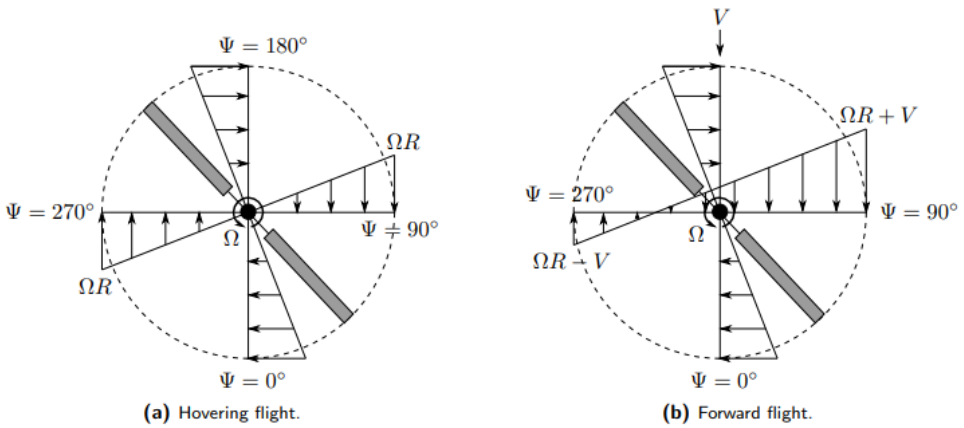

Figure 1.1: Profile of the velocity normal to the leading edge in hover and forward flight.

In the figure, the angular velocity of the rotor is denoted by W, its radius by R, the forward velocity of the vehicle by V, and the azimuth of one blade by Ψ. In hovering flight, the incident velocities are distributed symmetrically with respect to the rotor hub. However, during forward flight, the normal velocities on the advancing side ($0 < \Psi < 180$) are higher than those in the hover situation, while the velocities on the retreating side ($180 < \Psi < 360$) are smaller. Furthermore, a reverse flow region exists near the hub. This dissymmetry of velocities generates a lift gradient that produces a roll moment in the rotorcraft. In the *Cierva* autogyro this problem was solved by introducing hinges in the blade roots, which allowed for a flapping motion of the blades. This motion changed the local angles of attack and compensated for the asymmetrical lift distribution.

Due to Coriolis effects, when the rotor blades flap, a corresponding in-plane vibration appears at the blade root, usually known as lag (or lead-lag) motion. To deal with this vibration, lag hinges also need to be incorporated into the connection with the rotor hub. In addition to flapping and lag motion, the blades can be feathered about an axis parallel to the blade span. This rotation is very useful for controlling the entire helicopter, as it enables the pitch of the blades to be adjusted. Figure 1.2 illustrates the three fundamental blade motions and the hinge associated with each one.

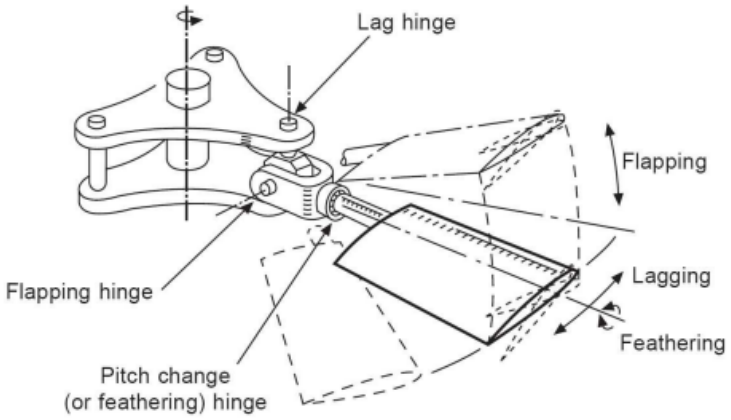

Figure 1.2: Fundamental rotor blade motions [4].

After the development of the *Cierva* autogyro, it became apparent that the effectiveness of the traditional fixed wing aircraft control surfaces was inadequate, especially when flying at low airspeeds. Therefore, a new component, called *direct control* system, was developed. It consists of a gimbal mounted rotor that can be tilted longitudinally and laterally using a control stick. As the rotor thrust vector is almost always perpendicular to its tip-path plane (the plane described by the tips of the rotating blades), the rotorcraft is accelerated in the direction of the tilt, generating roll and pitch moments around its CG.

The *direct control* was effective in controlling the autogyro, but it proved impractical for helicopters. Consequently, a new system was spontaneously developed by several engineers to account for the unequal aerodynamics on the rotor during forward flight and to provide roll and pitch control in those kind of systems. This solution, known as cyclic pitch, was initially proposed by *G. A. Crocco* in 1906. As the name indicates, it makes use of the pitch motion of the blades to achieve pitch and roll maneuvers in helicopters. In modern times, this invention continues to serve as one of the primary control systems for a helicopter.

1.1.1 Cyclic Pitch

The cyclic pitch system, which allows for the control of the pitch angle of helicopter blades, is reliant on the swashplate assembly. A typical architecture of this assembly can be seen in Figure 1.3.

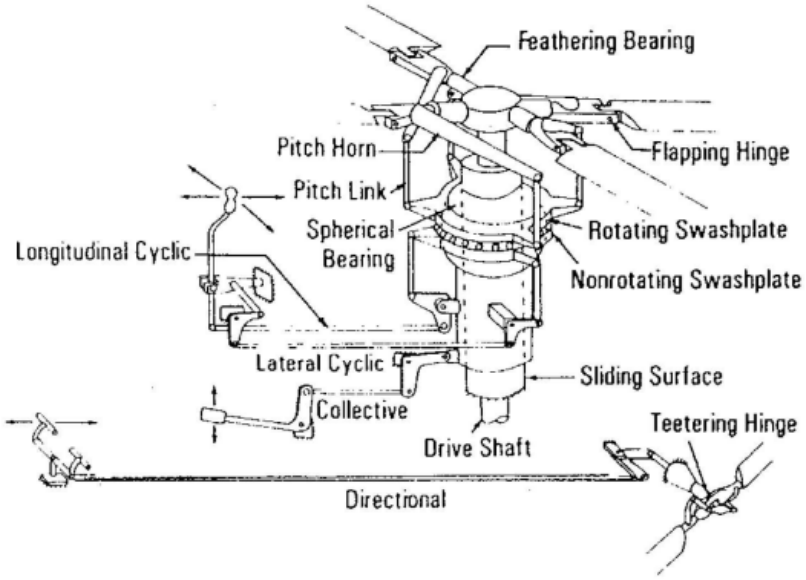

Figure 1.3: Generic helicopter flight control system. [34].

The swashplate comprises two main parts, namely the stationary swashplate and the rotating swashplate. The stationary (outer) swashplate, located on the main rotor mast, is connected to the cyclic and collective controls using a series of pushrods. These pushrods make up the so-called flight control actuation chain, which can vary between different types of helicopters, distinguishing between mechanical controls, hydro-mechanical controls and fly-by-wire controls. The fixed swashplate can tilt in all directions and move vertically. The rotating (inner) swashplate, on the other hand, is mounted on the stationary swashplate via a bearing and is allowed to rotate with the main rotor mast. An anti-rotation link is in place to prevent the inner swashplate from rotating independently of the blades, which could result in torque being applied to the actuators. The outer swashplate typically has an anti-rotation slider as well to prevent it from rotating. Both swashplates move up and down as a single unit, with the rotating swashplate linked to the pitch horns via pitch links.

By focusing exclusively on the cyclic pitch control mechanism, this system is responsible for tilting the swashplate and subsequently the rotor tip-path plane by cyclically adjusting the pitch of the helicopter blades. The tilt of the thrust vector, which is almost perpendicular to this plane, enables the pilot to steer the helicopter in the desired direction. The control for this tilt is managed using the so-called cyclic stick, which is usually located between the pilot's legs and which is illustrated in Figure 1.3. By moving this stick forward or backward, the pilot can tilt the tip-path plane longitudinally, causing the helicopter to pitch its nose down or up, respectively. Similarly, moving this command

sideways results in the tilt of the tip-path plane laterally, causing the helicopter to roll to the right or left.

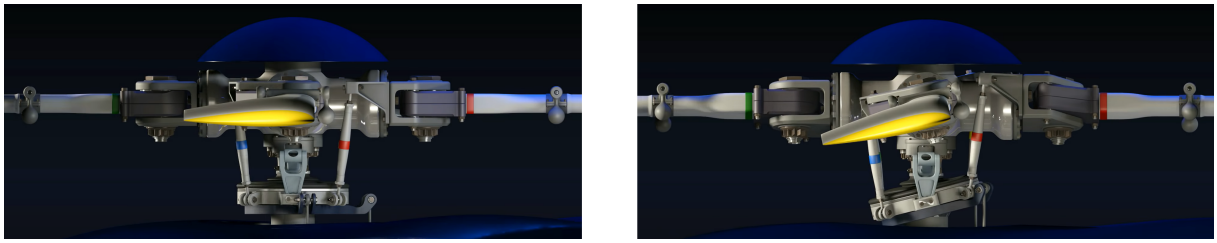

Figure 1.4: Example Cyclic Adjustment.

1.1.2 Collective Pitch

While the cyclic control is responsible for changing the pitch of the rotor blades in a cyclical manner, the collective control allows for an equal pitch adjustment across all blades. This is made possible by the vertical movement of the swashplate, which moves along a sliding surface around the rotor shaft. The collective lever, located on the left side of the pilot, operates a series of mechanical linkages that control the movement of the sliding surface, as depicted in Figure 1.3. Raising the collective lever increases the pitch of the blades, which generates more lift and increases rotor thrust. Figure 1.5 shows what happens to the rotor blades when this lever is raised. Conversely, lowering the collective lever reduces the thrust produced. This type of control is primarily used for vertical maneuvers, including climbing and descending.

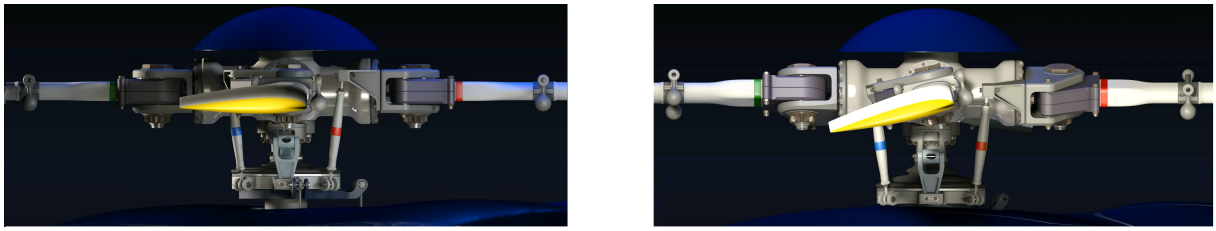

Figure 1.5: Example Collective Adjustment.

1.1.3 Anti-Torque Pedals

The anti-torque pedals are the last and third set of flight controls that enable a pilot to maneuver a helicopter. Indeed, as the name of these flight controls suggest, the main rotor, mounted on the fuselage, generates a torque moment that produces an equal and opposite moment on the helicopter body, causing it to spin, as per the principle of conservation of the angular momentum. In a single main rotor helicopter, a small auxiliary rotor is

usually mounted on the tail boom at the rear of the fuselage in order to generate thrust to counteract this moment. The collective pitch of the auxiliary rotor's blades is controlled by the pedals, as depicted in Figure 1.3. The left pedal increases the yaw moment of the helicopter to the left, while the right pedal is used to turn the helicopter to the right. Besides balancing the main rotor torque, the pedals also enable directional control by altering the helicopter's heading. Unlike the main rotor, the tail rotor has a small radius and a high rotational speed, which makes its flapping motion negligible, resulting in the absence of a cyclic pitch system.

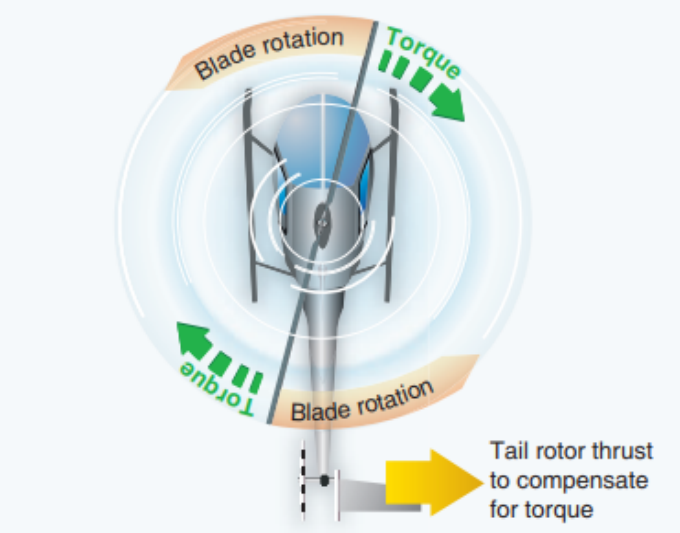

Figure 1.6: Anti-Torque Solution.

1.2 Automatic Flight Control System

Helicopters are associated with intrinsic instabilities and inter-axis couplings, which make controlling them a challenging task. This is because pilots need to constantly act on the different controls simultaneously. To address this issue, mechanical stabilizing systems were implemented in rotary-wing aircraft in the early years [34].

Initially, control forces in small helicopters were modest, allowing the pilot to directly act on the control mechanisms, as shown in Figure 1.3. However, with the increasing size of helicopters, control forces also increased, necessitating the installation of hydraulic boosted actuators to help the pilot handle the helicopter by providing extra strength in the control deflections. With the development of electronics, hydraulic control actuators began to incorporate an electro-hydraulic servo valve, which can be operated by electric signals from a flight computer. This made it possible to integrate automatic control laws, leading to the implementation of SAS with limited authorities of around 10 % of the

actuator stroke to improve the flying qualities.

Following the incorporation of this system and the further development of flight control laws, full authority flight controllers were introduced, and the first helicopter autopilots were implemented. Instead of controlling the control deflections directly, the pilot sends their intentions to the flight computer on board, which computes the required actions of the actuators to perform the desired maneuvers. The references commanded to the computer are normally supplied in terms of angular rates or attitude angles that correspond to the displacement of the conventional control interfaces in the cockpit. This type of configuration serves as the basis for fly-by-wire systems, which eliminate the mechanical linkages between the cockpit and the actuators, reducing the weight of the vehicle and the complexity of its mechanical systems.

In general, an *Automatic Flight Control System* (AFCS) in a helicopter is a collection of control systems designed to provide various levels of automatic control of flight by automatically changing the orientation of the flight controls along three or four axes of motion. A 4-axis AFCS operates in all four axes of aircraft control: pitch, roll, yaw, and collective.

Different nomenclatures may be found in manuals dealing with AFCS, even among different helicopters of the same manufacturer. However, despite the name assigned to the AFCS architecture, the AFCS is typically structured in two control loops:

- the innermost loop is the helicopter *Automatic Stabilization Equipment* (ASE), which include pitch, roll and yaw rate stabilisation by means of *Stability Augmentation System* (SAS), pitch, roll and yaw attitude hold by means of *Attitude Hold* (ATT), Turn Coordination by means of *Turn Coordinator* (TC). ASE operates in the pitch, roll and yaw axes.
- the outermost loop is the helicopter *Autopilot* (AP) which in turn enables the pilot to engage modes in all four axes of aircraft control. Those modes differs from the previous because they allow to control and to hold "external" conditions [31] reducing the pilot's workload and enhancing his situational awareness for a better managing of critical situations. The most important among these systems are the *Indicated Airspeed Hold* (IAS), the *Barometric Altitude Hold* (ALT), the *Radar Height Hold* (RHT), the *Heading Hold* (HDG), etc. The AP modes function through the ASE control systems and axes and may also operate in the collective axis. However, AP modesyy does not have to operate on every axis but may work on a single one up to all four axes. Transition of modes on axes, as well as mode engagement and disengagement decisions, are provided by the AP control logics.

The pilot can access these modes primarily through the switches and buttons on the *Autopilot Control Panel* (APCP).

Another common AFCS framework provides the following division:

1. *Lower Modes*, which basically contain SAS and ATT and thus control the aircraft angular rates and attitude. These systems grant a certain degree of stability and maneuverability of the aircraft, and therefore, the pilot hardly disengages those aids during flight.

2. *Upper Modes* which in practice contain the modes that allow control of the primary flight parameters (e.g. *Indicated Airspeed*, *Barometric Altitude*, *Radar Height*, etc.). In order to complete their tasks, those modes rely on the *Lower Modes* and especially on the ATT, which is usually a prerequisite for their functioning. However, the category of *Upper Modes* is basically coincident with the definition of *Autopilot* provided in the previous AFCS framework.

3. *Functionalities* which in practice differ from the previous categories because they provide automatic increment of helicopter controllability, stability and maneuverability. Indeed, one peculiarity of these systems is that they are always engaged during flight, but they provide control actions only in specific flight conditions. Among these functionalities, there are the *Turn Coordinator* (TC) and the *Auto-Trim*.

Since the framework adopted in this research is the second one, as it eases the achievement of greater modularity in the AFCS, the term "autopilot" will be used with the same meaning of "automatic flight control system" to shorten the notation.

1.2.1 Autopilot Control Panel and Stick Grip Control Switches

The *Autopilot Control Panel* provides controls for mode selection and arming, and mode status display for the AFCS. It is also used for pre-flight testing. The APCP is usually located in the center of the inter-seat console between the pilots and is equipped with multiple push buttons and rotary/push knobs, which are typically labeled with corresponding mode. Pressing a button triggers the activation of the specific mode, but it does not guarantee engagement in the AFCS due to control logics that may or may not allow engagement depending on various factors. Alternatively, pressing the button of an armed mode triggers its disengagement. Figure 1.7 shows an example configuration of the APCP.

Figure 1.7: APCP Example Configuration.

In addition, helicopters furnished with modern automatic flight control systems usually have cyclic and collective grips with several switches and buttons. The configuration and number of these additional grip controls may vary depending on the specific helicopter, but the fundamental cyclic and collective grips are always equipped with

- the *Force Trim* switch (also called *Force Trim Release* (FTR). As already said before, control forces in small helicopters were modest enough to allow the pilot to directly act on the control mechanisms via cyclic, collective, and pedals. Later on, as helicopters got bigger, so did the forces and hence the previously cited solutions were introduced. However, the drawback was that the pilot lost the feel from the real control displacement applied to the swashplate or to the tail rotor [34].

  To address the issue, control centering and artificial feel system were introduced inside the pilot controls [34]. Indeed, in addition to the artificial force gradient, the control should have a definite "detent" position that requires some force to initially move or "breakout". This detent can be achieved through the use of a cartridge with two preloaded springs opposing each other. When a breakout occurs, a sensor perceives this movement and updates the control status to the so-called *Out of Detent* state. As a result, both the gradient and detent work together to ensure that if the control is moved and then released, it will return to its original trim position.

  However, since during flight, a pilot often necessitate to apply large changes in flight controls in order to modify the helicopter flight condition, it would be desirable to avoid fighting against the retention of the spring as it would make it difficult for the pilot to control the aircraft smoothly. Moreover, the control used by the pilot should keep its new position in these cases. As a result, to address these necessities, a pilot presses and hold the *Force Trim* button present either in the cyclic and in collective

sticks, which in turn relax the retention of the spring. Once the pilot has maneuvered the helicopter and releases the button, the force gradients are re-established with the new location as the zero force point.

FTR buttons are typically located on both the cyclic and collective sticks, as depicted in Figure 1.12. However, the cyclic FTR button not only removes forces in the cyclic, but also in the pedals.

Furthermore, if the cyclic FTR button is hold, the operation of the modes engaged in the pitch, roll and yaw axes are inhibited; the same applies for the collective FTR button with respect to the modes engaged at that moment in the collective axis of the AFCS.

Additionally, pressing the cyclic FTR button resets the specific reference datum of each AFCS mode operating on the pitch, roll, and yaw axes to their current value, which is sensed at the moment in which the button is released. The same applies to the collective FTR button with respect to the mode operating on the collective axis. However, it should be noted that if the pilot operates against the spring retention (i.e. without pressing the FTR) in either the cyclic or the collective, the reference datum remains unchanged, and when the maneuver is complete, the artificial feel system returns the helicopter to its previous trim condition.

- the *Beep Trim* switch. When a pilot wants to maneuver to a new flight condition, he may disengage the trim system by means of FTR or he may "beep" the controls to the new flight condition. Indeed, a *Beep Trim* is a 4-way hat switch that can be moved and held in forward, backward, and sideways positions. It is typically present either on the cyclic and on the collective sticks and when used it moves the trim system to center about a new position.

  This new position is set on the basis of the modes engaged on the AFCS at that moment, since primarily the movement of the *Beep Trim* assigns an increment or decrement to the reference datum of specific modes, which is proportional to the amount of time that the switch is pressed in a certain direction. Forward and backward movements of the cyclic *Beep Trim* vary the reference datum of the pitch mode engaged in that moment in the AFCS. Left and Right movements of the cyclic *Beep Trim* does the same for the current AFCS roll mode. Instead, forward and backward movements of the collective *Beep Trim* act on the reference datum of the mode active on the collective axis, while sideways movements does the same for the yaw mode engaged. Each AFCS mode has its own *Beep Trim* functioning rate, which specifies the rate of variation of the specific reference datum per second.

Figure 1.8: Collective and Cyclic AFCS Grip Controls.

1.2.2 PID Controllers

The PID control algorithm has been widely used in various applications, including helicopter control systems, for several decades. The algorithm works by continuously measuring the difference between a desired setpoint and the current value of a process variable, and utilizing this error signal to compute an appropriate control action that can assist in bringing the process variable closer to the setpoint.

In modern helicopter applications, the PID control algorithm is still extensively employed as a component of the automatic flight control system to provide adjustments on cyclic, collective, and pedal inputs to enable stable flight. In the present work, therefore, the development of the AFCS also rests on PID controllers.

One of the advantages of using PID controllers in helicopter applications is that they can be easily tuned and customized to suit specific helicopter models and flight conditions. This means that helicopter manufacturers and operators can optimize the AFCS to provide the best possible performance and safety for their particular application.

In general, PID controllers are most effective when the loop to be controlled is linear and symmetric. However, when the system is non-linear and asymmetric, limitations regarding the use of PID controllers arise. This is unfortunately often the case in many helicopter flight phases, as rotorcrafts are non-linear systems that exhibit different behavior in different operating regions. The sub-optimality resulting from the use of PID control can vary from being imperceptible to causing severe damage to the system. One common approach to address this problem is to apply *Gain Scheduling*, a control strategy that involves linearizing the problem around different operating points to create a family of PID controllers (one for each point) and tuning each of them to be optimal for the

corresponding linearization.

PID Algorithm

In standard industrial regulator, the control variable u is obtained as a sum of three components: a term proportional to the error e, a term proportional to the integral of e and the last proportional to the derivative of e; indeed PID stands for *Proportional Integral Derivative*.

The error is defined as $e = y_{sp} - y$, i.e. the difference between the reference y_{sp} (often called setpoint) and the measured process variable y. The most basic algorithm formulation of a PID controller is discribed by [1]:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt} = K_p \left(e(t) + \frac{1}{T_i} \int_0^t e(\tau)d\tau + T_d \frac{de(t)}{dt} \right). \quad (1.1)$$

The proportional gain K_p, the integral gain K_i and the derivative gain K_d are the control parameters of a standard PID regulator (see Figure 1.9). The time constants T_i and T_d, called integral time and derivative time, respectively, are sometimes used instead of the integral and derivative gains.

An industrial PID regulator might involve only the action of one or two components, not necessarily the combination of all, since every term plays a different role in the control loop mechanism.

- The proportional term amplify the instantaneous error, reducing the system response time without introducing a phase shift. However, the stability is reduced when increasing K_p because the amplitude of oscillations are also enhanced. In general, with constant disturbances and a purely proportional regulator, the steady state error is non-zero but, in some cases, it can be cancelled by simply adding a proper offset to $u(t)$.
- The integral term takes into account the history of the error, it increases indefinitely when the steady state error approaches a finite value different from zero and therefore it is used to remove the residual error (in particular when the disturbances are affected by step changes). Indeed, The precision improvement of the integral term is often coupled with the action of the proportional one.
  On the other hand, the integral component adds a phase shift of -90° that reduces the phase margin, it increases the system response time by narrowing the bandwidth and might lead to the *integral windup* issue.

- The derivative term aims to control the error by predicting its future state. It introduces a phase shift of $+90°$ that increases the phase margin of the plant and reduces the system response time due to a broader bandwidth. The last effect represents also a drawback since high frequency noise in measures are amplified by the derivative controller and a low-pass filter is therefore needed.

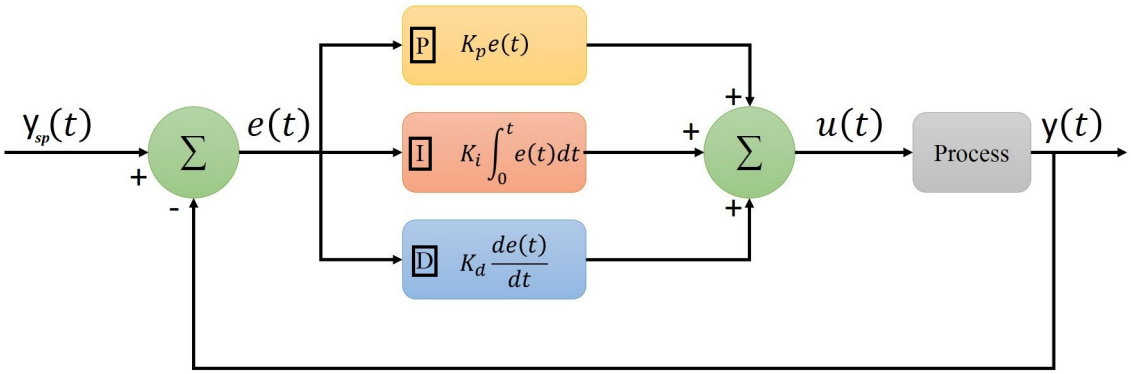

Figure 1.9: Block diagram of a feedback PID control system

The Laplace transform of the control law given in Equation 1.1 is:

$$U(s) = K_p E(s) + \frac{K_i}{s} E(s) + K_d s E(s) = K_p \left(1 + \frac{1}{T_i s} + T_d s\right) E(s), \tag{1.2}$$

when considering also the low-pass filter in the derivative term, the control law becomes:

$$U(s) = K_p \left(1 + \frac{1}{T_i s} + \frac{T_d s}{\frac{T_d}{N} s + 1}\right) E(s), \tag{1.3}$$

with N/T_d representing the bandwidth of the filter.

When using the PID model of Equation 1.3, it follows that a step change in the setpoint Y_{sp} (recall that $E = Y_{sp} - Y$) produces a sharp spike in the control variable U, ideally a Dirac impulse when $N \to +\infty$.

This problem can be solved considering the two-way (or 2-DOF) PID architecture, characterized by the addition of the parameters (b, c) to the standard PID model:

$$U = K_p \left((bY_{sp} - Y) + \frac{1}{T_i s}(Y_{sp} - Y) + \frac{T_d s}{\frac{T_d}{N} s + 1}(cY_{sp} - Y) \right), \tag{1.4}$$

b and c take values in the range $[0, 1]$ and are known as the setpoint proportional weight

and setpoint derivative weight, respectively. When $b = c = 1$, the basic PID control law is retrieved.

The formulation given in Equation 1.4 allows to decouple and differentiate the control transfer function response to the setpoint and the measured variable, indeed:

$$U = G_c^{sp} Y_{sp} - G_c^{out} Y, \quad \text{with} \quad \begin{cases} G_c^{sp} = K_p \left(b + \dfrac{1}{T_i s} + \dfrac{T_d s}{\frac{T_d}{N} s + 1} c \right), \\ G_c^{out} = K_p \left(1 + \dfrac{1}{T_i s} + \dfrac{T_d s}{\frac{T_d}{N} s + 1} \right). \end{cases}$$

Moreover, the weights b and c allow to place freely the zeros of the closed-loop system in order to adjust the transient response. However, in most industrial regulators $c = 0$ and $b \neq 0$, with this choice the derivative action applied to the setpoint dynamics is cancelled out.

Integral Windup

The *windup* phenomena happens when the output of the regulator $u(t)$ does not correspond to the real actuator action $m(t)$. The actuator is a device with physical limitation, its action $m(t)$, mathematically, is a non-linear function of $u(t)$ of the type:

$$m(t) = \begin{cases} u_{min} & \text{if } u(t) \leq u_{min}, \\ u(t) & \text{if } u_{min} < u(t) < u_{max}, \\ u_{max} & \text{if } u(t) \geq u_{max}. \end{cases}$$

The actuator saturation occurs whenever the control variable reaches the physical limits, then the feedback loop is broken and the plant evolves with a constant input. Meanwhile, the integrator continues to integrate the error variable and, since it is not asymptotically stable, values far different from the ones really needed are attained. Once the error gets to values that would produce a regular control signal, the integrator needs still time to decrease its output and thus also the PID controller before coming back to a working state. For this reason, this phenomena is often referred to as *integral windup*.

A simple approach to avoid this issue consists of an interruption of the integral action whenever the actuator saturation occurs; as soon as the actuator input and output matches again the integrator returns to operate. In Figure 1.10 is shown the block diagram of the PID controller that embeds the evaluation of $|m(t)| - |u(t)|$ and the conditional

interruption of the integrator.

This anti-windup method is known as *conditional integration* and is widely spread in industrial PID because is a sensible precaution, however attention must be paid to the way in which the saturation is identified in order to avoid useless obstacles in the control action.

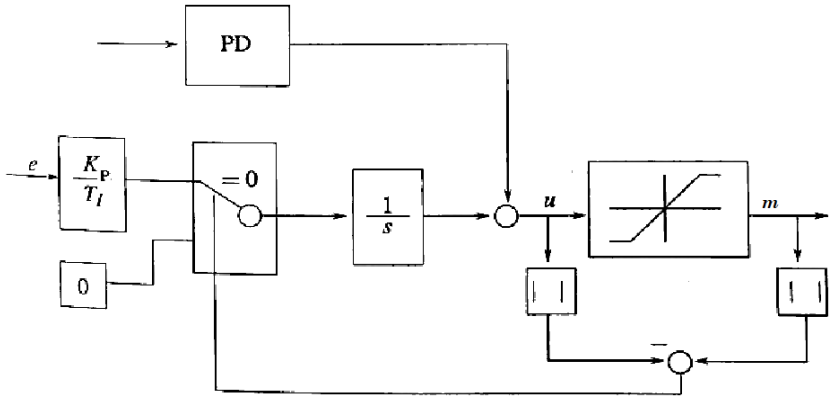

Figure 1.10: Block diagram of a PID controller with conditional integration

1.2.3 Lower Modes

Lower modes play a crucial role in maintaining the stability and maneuverability of helicopters. These modes are designed to address the inherent instabilities and inter-axis couplings that are typical of helicopter dynamics. Without them, flying a helicopter would be an extremely challenging task, as pilots would need to constantly manage the various controls simultaneously to maintain stability and control. Therefore, these lower modes represent the primary means by which pilots can effectively fly and control helicopters, making it possible to perform a wide range of complex maneuvers and tasks with greater ease and safety.

SAS - Stability Augmentation System

Conventional helicopters have an inherent instability, especially in pitch and roll, which, as already said, can lead to pilot fatigue during flight due to the prolonged control he has to apply. To overcome this issue, the *Stability Augmentation System* is used to improve the stability of the aircraft by introducing independent control actions in the three axes based on the pitch, roll, and yaw angular rates.

In its basic form, the SAS control law is a negative *Proportional* rate feedback. However, since SAS control actions feed *series actuators* with limited stroke authorities (usually

between ∓10% and ∓20% of the maximum stroke; specific values different from one helicopter to another), the high proportional gains required would lead to quick saturation in the actuators. To avoid this problem, the SAS control law may also include the *Integral* term (PI controlled SAS).

However, in this latter configuration, the SAS fights against the pilot's actions as well as disturbances. Indeed, the SAS must provide short-term aircraft angular rate stabilization to mitigate the effects of external disturbances, such as turbulence, but it should not provide any long-term stabilization on a preset aircraft attitude in order to avoid interfering with the pilot's hands-on and feet-on flight controls.

To address this issue, the solution illustrated in Figure 1.11 can be implemented, where the pseudo-attitude hold contribution is removed by cutting out this branch when the pilot is maneuvering. It implies that if a specific pilot control is *out of detent* or if the *force trim* buttons of the cyclic or collective grips are held, the integral SAS action associated with the stabilization control action of one or more specific axes is inhibited.

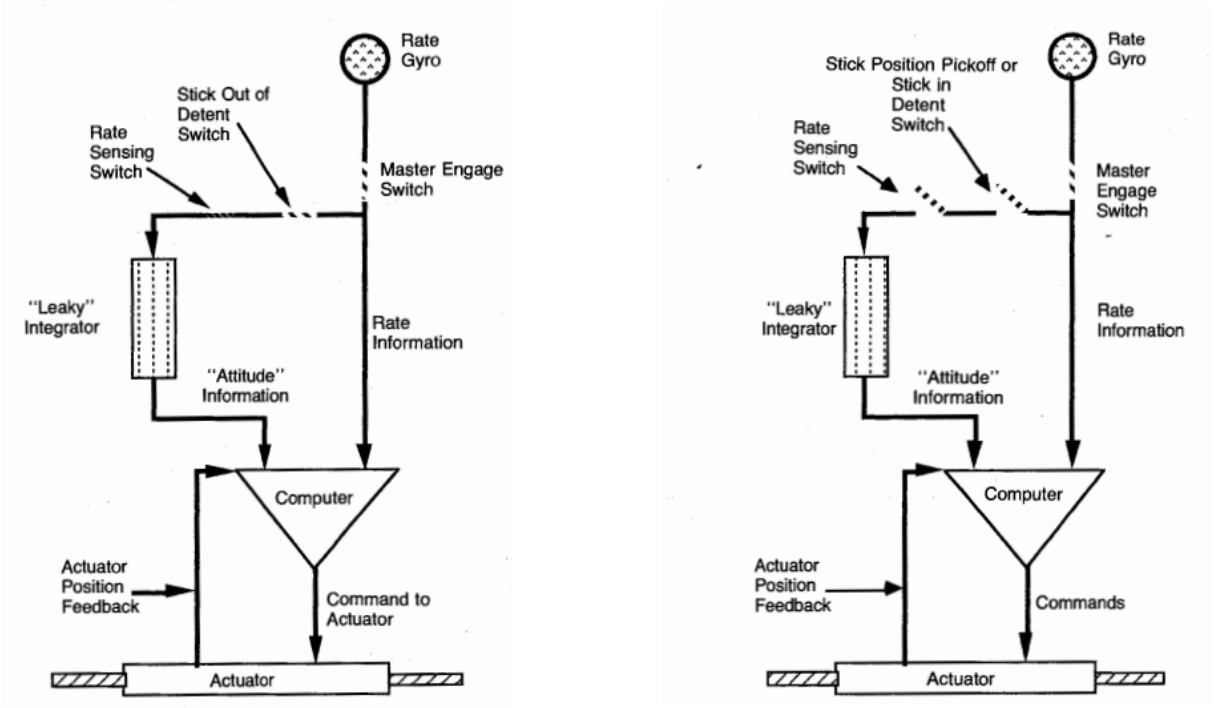

Figure 1.11: SAS block diagram (steady condition on the left, maneuver condition on the right) [31]

Moreover, the SAS can allow stability augmentation functions to be used separately, even if automatic control is not required or available due to an autopilot malfunction [31], as shown in the AFCS workflow diagram in Figure 1.11.

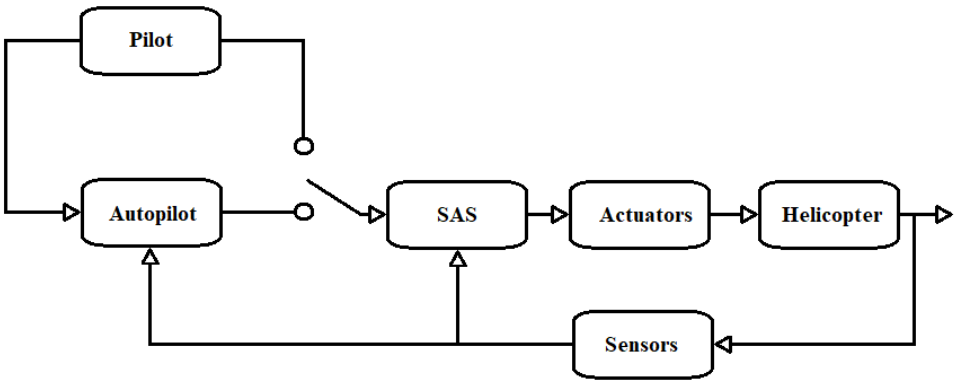

Figure 1.12: AFCS Workflow Diagram.

ATT - Attitude Hold System

The other basic mode which is always present in modern AFCS architectures, is the *Attitude Hold System*. This mode furnish another essential feature, as it provides the capability to acquire and hold a pitch, roll and yaw attitude reference; the ATT mode operates on the pitch, roll and yaw axis simultaneously and independently.

The ATT mode is commonly fed by a vertical gyro that senses the helicopter's attitude. The measured value is compared with a fixed reference attitude, and based on these inputs, a PID controller generates the driver signal for the *series actuators*, which, as already said, have limited stroke authorities. Nevertheless, even if the ATT mode is correctly engaged in the AFCS, it provides null control action if the pilot is flying manually. To adjust the reference attitude datum, *force trim* or *beep trim* can be employed, as discussed in Section 1.2.1. Obviously, references both in pitch attitude and in roll attitude are limited to specific thresholds characteristic of the helicopter's performance.

When the helicopter is in cruise condition and its *Indicated Airspeed* exceeds a certain threshold specific to the helicopter, the ATT mode is disengaged only for the yaw axis. In such cases, the yaw axis may be managed by the *Turn Coordination function* which automatically start to operate when the same *Indicated Airspeed* threshold is exceeded and if a minimum *Roll Angle* value is overpassed.

The ATT mode is the default mode of operation for the AFCS, automatically engaging when the AP button on the APCP is pressed. This mode is a prerequisite for every *upper mode* that operates on the pitch, roll, and yaw axis. Whenever an *upper mode* operating on these axes is manually disengaged via the APCP, the ATT mode automatically engages on the unengaged axis. In certain cases of automatic disengagement of some *upper modes*, the ATT mode also engages as a backup.

1.2.4 Upper Modes

Section 1.2 has already analyzed the most important features of the *upper modes*, but there are some additional notes that need to be provided to further inspect these systems. The *upper modes* are a collection of systems located at the outermost loop of the helicopter AFCS. These modes can actually fly the helicopter and perform certain functions selected by the pilot in the APCP. With some of these modes engaged, the pilot no longer directly controls the aircraft but selects the flight conditions that he wants the autopilot to maintain, while monitoring the functioning through displays and indicators in the cockpit.

For a 4-axis AFCS, the *upper modes* controlling the pitch, roll, or yaw axes always rely on the functioning of the ATT to successfully perform their task. However, the same does not apply to the collective channel, as the collective *upper modes* directly feed the collective *series actuator*. Modes that rely on the ATT do not feed any actuator, but their control law generates a $\delta Attitude$, which is summed with the current value of the attitude to provide the reference value for the ATT. The ATT subsequently evaluates the control action for the specific *series actuator* to chase the reference value of the specific *upper mode*.

Therefore, the control law behind a generic *upper mode* (valid also for collective modes) aims to hold the aircraft at a reference external condition by comparing this setpoint value to the current value of the same parameter. These inputs are provided to a PID controller which in turn calculates the necessary control action. The reference values of the *upper modes* can be adjusted using *force trim* or *beep trim* and if the pilot manually flies the aircraft, the *upper mode* functioning is inhibited. These same properties are present, as already seen, in the ATT; indeed the structural behaviour of an *Attitude Hold* is very similar to the one of a generic *upper mode*.

Another very important consideration regarding the *upper modes* refers to their control logics. Apart from the obvious use of switch and buttons in the APCP for the manual engagement of an *upper mode* in the AFCS, there are several non-trivial control logics that determine the automatic engagement or disengagement of a mode during flight. While these control logics may vary among different AFCS, the fundamental ones usually remain the same among different AP.

In conclusion, the facilities that can be provided by the *upper modes* are limited only by the sensors, computing capabilities and the control authority of the specific helicopter AP [31]. However, certain modes are more commonly available in different rotorcraft AFCS than others. Therefore, in line with the imposed requirements for the current research,

the following sections present these *upper modes* and their principal characteristics.

IAS - Indicated Airspeed Hold

The IAS mode provides the capability to capture and to hold an *Indicated Airspeed* reference between a minimum and a maximum *Indicated Airspeed* (usually between 30/40 *kt* and VNE); the mode operates on the pitch axis.

The initial datum in the IAS mode is the *Indicated Airspeed* existing at the time of engagement. The *Indicated Airspeed* reference can be modified using the cyclic *force trim* which releases the pitch *trim actuator* and synchronizes the position reference of the cyclic control with the current *Indicated Airspeed* reference. Setpoint adjustments may be provided also by means of forward/backward movements of cyclic *beep trim*.

IAS mode can be engaged/disengaged pressing the IAS push button on the APCP. Automatic engagement of IAS mode may be triggered by the ALT transition from pitch to collective axis. Automatic disengagement occurs in case of engagement of an incompatible mode (in the AFCS developed in this research, only incompatible mode is HOV).

ALT - Barometric Altitude Hold

The ALT mode provides the capability to capture and to hold a *Barometric Altitude* reference, between a minimum and a maximum value (usually upper value is the maximum operational altitude of the aircraft).

The mode may operate on the collective or on the pitch axes. If *Indicated Airspeed* is lower than a threshold value (typically $60kt$), the mode is engaged on the collective axis. Otherwise, if pitch axis is free, meaning that no other *upper modes* are controlling the pitch channel, then the ALT may operate on the pitch axis. If *Indicated Airspeed* drops below the threshold value while ALT is engaged on the pitch axis, and if collective axis is available, ALT is transferred to collective axis and IAS mode is automatically engaged in pitch axis. The viceversa applies if the *Indicated Airspeed* surpasses the threshold value while ALT is engaged on the collective axis, and pitch axis is free.

The initial datum in the ALT mode is the *Barometric Altitude* existing at the time of engagement. The *Barometric Altitude* reference can be modified using the collective *force trim* which releases the collective *trim actuator* and synchronizes the position reference of the collective control with the current *Barometric Altitude* reference. Setpoint adjustments may be provided also by means of forward/backward movements of collective *beep trim*.

ALT mode can be engaged/disengaged pressing the ALT push button on the APCP. Automatic engagement of ALT mode may be triggered by automatic disengagement of RHT due to exceeding of maximum *Radar Height*. Automatic disengagement occurs in case of engagement of an incompatible mode (in the AFCS developed in this research, incompatible mode are RHT and HOV).

RHT - Radar Height Hold

The RHT mode provides the capability to capture and to hold a *Radar Height* reference between a minimum and a maximum value (usually up to $2500ft$); the mode operates on the collective axis.

The initial datum in the RHT mode is the *Radar Height* existing at the time of engagement. The *Radar Height* reference can be modified using the collective *force trim* which releases the collective *trim actuator* and synchronizes the position reference of the collective control with the current *Radar Height* reference. Setpoint adjustments may be provided also by means of forward/backward movements of collective *beep trim*.

RHT mode can be engaged/disengaged pressing the RHT push button on the APCP. Automatic engagement of RHT mode may be triggered by HOV engagement. Automatic disengagement occurs in case of engagement of an incompatible mode (in the AFCS developed in this research, only incompatible mode is ALT). If the maximum *Radar Height* threshold is exceeded ALT mode operates as a backup mode.

HDG - Heading Hold

The HDG mode provides the capability to capture and to hold a *Heading Angle* up to VNE.

The mode may operates on the roll or on the yaw axes (in the latter case coincides with the ATT on the yaw axis). Below the same *Indicated Airspeed* threshold for the functioning of the *Turn Coordinator* (usually around $40kt$ and $60kt$ for light utility helicopters), the HDG is engaged or transitioned on the yaw axis. Otherwise, if the threshold is exceeded, HDG is engaged or transitioned on the roll axis and is assisted by the *Turn Coordinator* which provides control action on the yaw axis.

The initial datum in the HDG mode is the *Heading Angle* existing at the time of engagement. The *Heading Angle* reference can be modified using the pedal/cyclic *force trim* (respectively the first case for HDG on yaw axis and the second in case of HDG engaged on the roll axis) which releases the yaw/roll *trim actuator* and synchronizes the position

reference of the yaw/roll control with the current *Heading Angle* reference. Setpoint adjustments may be provided also by means of sideways movements of collective/cyclic *beep trim* (respectively the first case for HDG on yaw axis and the second in case of HDG engaged on the roll axis).

HDG mode can be engaged/disengaged pressing the HDG push button on the APCP. Automatic disengagement occurs in case of engagement of an incompatible mode (in the AFCS developed in this research, only incompatible mode is HOV).

HOV - Hover Hold

The HOV mode provides the capability to capture and to hold a *Lateral Groundspeed* and *Longitudinal Groundspeed* references under certain thresholds in terms of *Lateral Groundspeed*, *Longitudinal Groundspeed* and *Indicated Airspeed*.

The mode operates on the pitch and roll axis and, when engaged, the RHT (or ALT as backup) and HDG modes are simultaneously engaged by the system to control height and heading respectively. The collective Axis is Under Control of either RHT or ALT modes, while Yaw axis is under control of HDG mode.

The initial datum in the HOV mode is the *Lateral Groundspeed* and *Longitudinal Groundspeed* existing at the time of engagement. The *Lateral Groundspeed* and *Longitudinal Groundspeed* references can be modified using the cyclic *force trim* which releases the pitch and roll *trim actuators* and synchronizes the position reference of the pitch and roll controls with the current *Lateral Groundspeed* and *Longitudinal Groundspeed* references. Setpoint adjustments may be provided also by means of forward/backward/left/right movements of cyclic *beep trim*.

HOV mode can be engaged/disengaged pressing the HOV push button on the APCP. Automatic disengagement occurs in case of engagement of an incompatible mode (in the AFCS developed in this research, incompatible modes are IAS and HDG).

1.2.5 Functionalities

The AFCS provides different control *Functionalities* that automatically increase the helicopter controllability, stability and maneuverability. Each *Functionality* offers automatic control actions only in specific flight conditions that are characteristic of the respective *Functionality*.

TC - Turn Coordinator

To turn a helicopter while in a hover the pilot uses the pedals to control the tail rotor by adjusting the amount of sideways thrust it creates: this kind of maneuver is called *flat turn*. On the other hand, when turning the helicopter while in forward flight, the pilot tilts the main rotor disk in the desired direction using the cyclic control, and the fuselage follows; in this second configuration the pedals do not contribute to turning the helicopter, but instead are used to balance the torque and maintain longitudinal trim for coordinated flight. Indeed, pilots typically aim for *coordinated turns* which are turns with no lateral acceleration, as those motions are undesirable and poorly tolerated by the human body.

Therefore, within the AFCS, the *Turn Coordination Functionality* provides control of the aircraft lateral acceleration to ensure coordinated flight. The function operates through the yaw axis and may furnish turn coordination both in forward flight and in turns. The control law also in this case may be managed by a PID controller having the *Lateral Acceleration* as parameter to control and where the reference is always set to 0. The control action is fed to the yaw *series actuator*.

This function is automatically activated when the *Indicated Airspeed* at which the pilot is flying exceeds a determined threshold (usually around $40kt$ and $60kt$ for light utility helicopters) and a fixed value of *Roll Angle* is surpassed (usually between 0°and 3°). When these conditions are met, the system provides turn coordination if the autopilot is holding a roll attitude via a specific mode (e.g. HDG-Roll or ATT-Roll), and the pilot is flying feet-off. However, turn coordination is also automatically provided when the pilot manually maneuvers the helicopter in the roll axis without using the pedals.

2 | Simulink Tools for Flexible and Efficient System Design

This chapter examines the principal tools available in Simulink® for designing a system having high levels of modularity, generality, and automation, as in the case of the development of the current AFCS model. Modularity refers to the property of a system or program consisting of distinct units, each with a specific functionality, capable of interacting with each other. On the other hand, genericity refers to the ability of a system to offer a high degree of configurability to the end user, enabling customization of the model according to specific needs and requirements. In conclusion, automatization in Simulink refers to a suite of optimization tools that can automate the process of maximizing system performance in accordance with pre-established metrics and objectives. Additionally, Simulink offers the capability to automatically generate code from system models, reducing the time and effort required for software development and minimizing the risk of introducing errors into the code.

The subsequent paragraphs provide a concise overview of the essential Simulink® tools utilized for the *Model-Based Design* of the present AFCS, but which in general can be applied for any project with the above-mentioned requirements.

2.1 Custom Libreries

In Simulink®, a *Custom Library* refers to a collection of custom blocks that users can create and store in their personal library. These custom blocks are designed to perform specific functions or tasks that are frequently encountered during the development of a model or a project. Indeed, these blocks are designed with features that encourage their reusability, modularity, and scalability. The use of *Configuration Masks* can further enhance the purpose of these blocks by allowing customization of block parameters. This enables users to modify the behavior of custom blocks and provide a visual representation of their function.

It is important to exercise caution when adding a *Custom Library* to the Simulink® *Library Browser*, as this process can be complex and requires the user to follow specific steps. In order to achieve a result similar to the one shown in Figure 2.1, users must carefully follow the guidelines outlined in the reference [14].

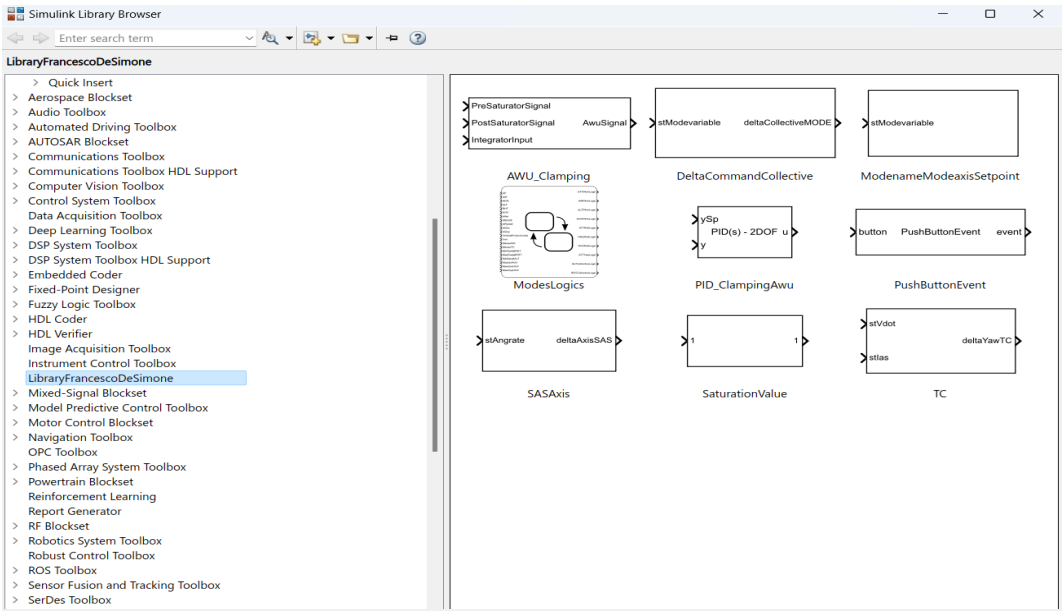

Figure 2.1: Custom Library

In the present work, the *Custom Library* shown in Figure 2.1 has been created and all blocks in the library were designed with the intention of achieving generality, allowing them to be reused multiple times throughout the assembly of the model. Indeed, throughout the development of the AFCS, various modifications have been tested and custom blocks have evolved accordingly. The *Custom Library* allows the user to keep track of these modifications and ensure that they are reflected in the model being developed. It is therefore considered good practice not to set the *Library Link* to *Disable Link* when building the model.

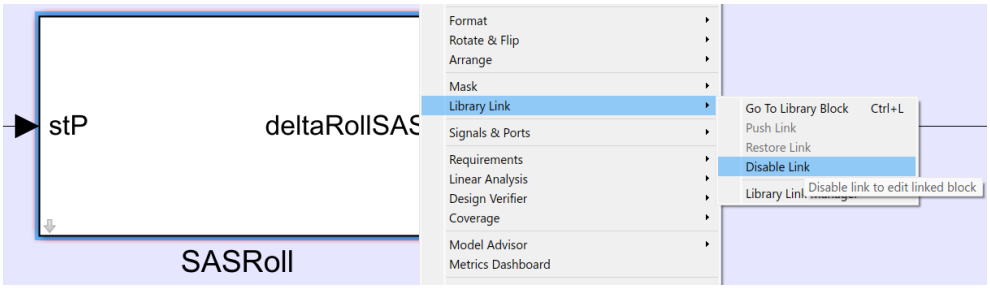

Figure 2.2: Library Link

Instead, the *Disable Link* function may be used for testing changes locally in the model without affecting the real structure of the library block. In this regard, it should be noted that changes made while the link is disabled will not be preserved in the library block, even if the user decides to select *Restore Link*.

2.2 Block Masks

In Simulink®, a mask is a GUI that can be added to a subsystem made of different blocks in order to allow end-users to customize its structure, behavior and parameters without dealing with the individual components that compose it.

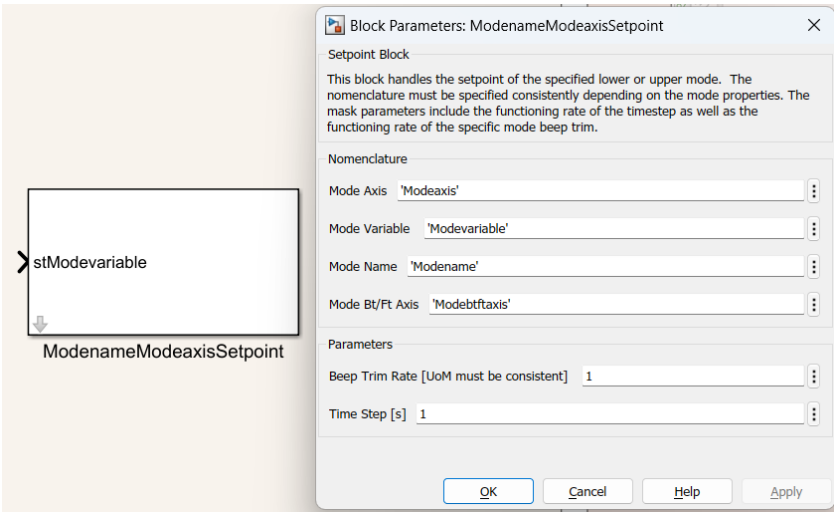

Figure 2.3: Example of a Masked Block and its GUI

Clearly, in order to have this type of automation, the price to be paid is the upstream definition of the characteristics that this mask must have. This is where the *Mask Editor* [15] comes in, as this tool provides a graphical interface for editing masks, allowing users to add and arrange parameters, create custom user interfaces, and specify the behavior of the block.

The example displayed in Figure 2.4 illustrates the structure of the *Parameters & Dialog* interface within the *Mask Editor*. This interface represents the most significant component among the other panels within the editor, as it enables the user to define the parameters of the block that are customizable through the mask. Additionally, the interface provides users with the ability to customize the GUI of the mask, allowing for the selection of various controls such as text fields, checkboxes, and drop-down menus to modify and add parameters.

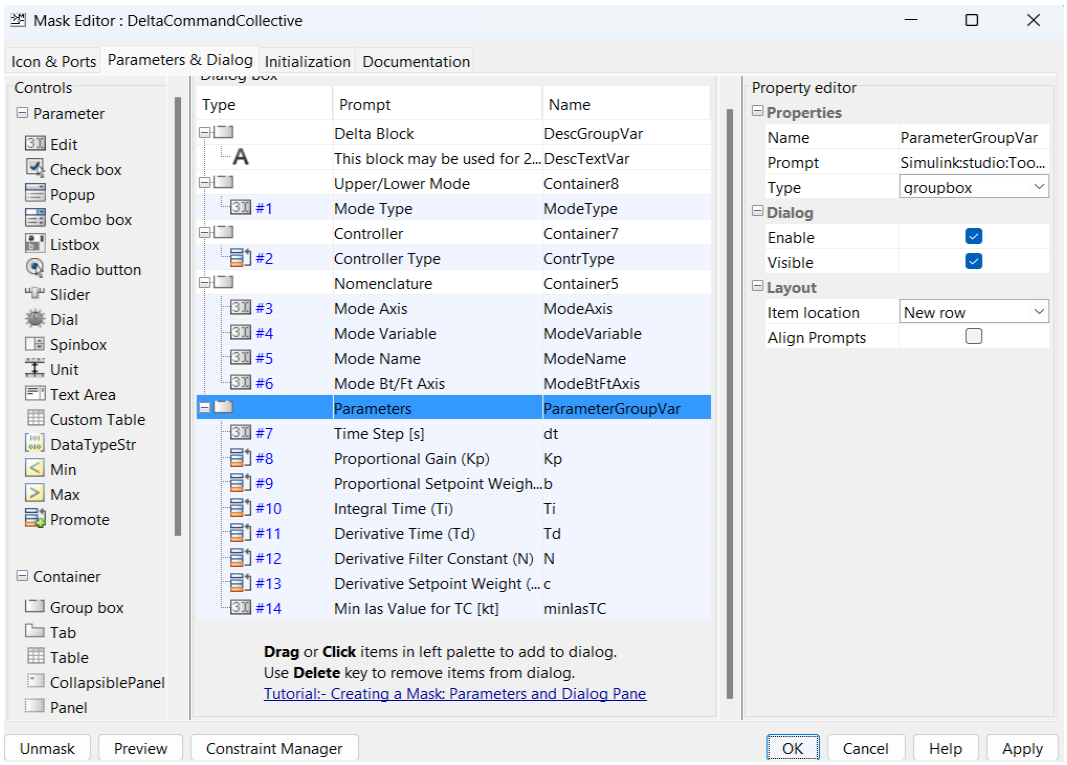

Figure 2.4: Example of a Mask Editor Interface

In addition to the *Parameters & Dialog* interface, the *Initialization* section within the tabbed panes holds significant importance and plays a pivotal role in enhancing the reusability and generality of the masked block. This section provides the ability to initialize the masked block using MATLAB® code. It is important to note that all parameters defined within the *Parameters & Dialog* interface are also available within this section. This allows the initialization of the masked block to be customized based on the properties of these parameters, which are defined directly by the end-user. Such flexibility enables the subsystem to vary in every aspect and can be tailored to meet the needs of the designer.

On the other hand, the *Icon & Ports* section is primarily utilized to create a custom icon for the block, which is displayed in the Simulink® model. This section also allows users to define the block's input and output ports and specify their properties, such as dimensions and data types.

Finally, the *Documentation* pane serves as a platform to describe the behavior and purpose of the masked block in detail. This section enables users to add comprehensive information, including *Description*, which can be crucial while using the MATLAB® command's *set_param*.

2.3 Stateflow Charts

Stateflow® is a MATLAB® tool that provide a graphical programming environment based on finite state machines, i.e. dynamic systems that transition from one operating mode (state) to another. With Stateflow® is possible to describe how MATLAB algorithms and Simulink models respond to input signals, events, and time-based conditions; all those features enable the system designer to easily develop supervisory control logics and task management logics while having models that remain clear and concise, even as the complexity of the system increases. The Stateflow® environment also supports testing and debugging phases, considering various simulation scenarios, with methods for activity animation and integrity control systems. Finally, this toolbox may generate code from their own state machine.

Stateflow®, following an approach similar to the typical Simulink® block diagrams, employs a state transition diagram composed of various objects organized in hierarchy. To optimize designer needs, multiple strategies and tools may be employed; however, it will be just explained in the following, which are the most important graphical objects of this toolbox, as well as the most adopted components in the realisation and development of the AFCS model.

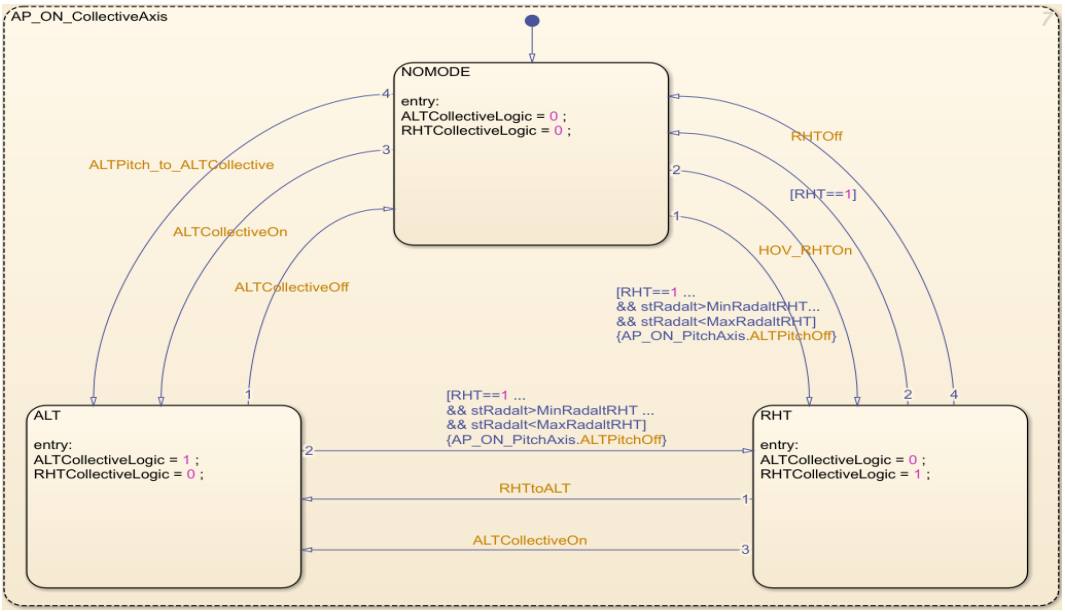

Figure 2.5: Example of Stateflow® Chart

- *States* are a graphical Stateflow® object that may be active or inactive depending on the logics of the diagram. Usually, a state contain a specific set of actions that are executed when it become active. Those actions, depending on designer's needs,

may be very simple as updating existing variables or creating new ones, or they may be more complex and invoke, for example, *MATLAB® Functions*.

When defining a state actions it is common practice to define when the state action occur, by specifying one or more of the following [21] :

1. *entry* (abbr. *en*), if it only needs to be performed in the time step in which the state changes from inactive to active.
2. *during* (abbr. *du*), if it is to be executed for each time step in which the state is active, starting with the time step after the state has been activated
3. *exit* (abbr. *ex*), if it only needs to be performed in the time step in which the state changes from active to inactive.

It is possible to organize states in a hierarchical structure wherein a state referred to as a *Substate* can only become active when its parent state is active. States that have substates are commonly referred to as *Superstates*.

Once the hierarchy has been defined, it is important to recognise that each superstate imposes a decomposition, which is unique for each of its substates, and that defines their *Decomposition* of execution [24]. The decomposition can be either *Exclusive (OR)* or *Parallel (AND)*, identified in the graph by solid and dashed borders, respectively. Exclusive decomposition is employed to represent mutually exclusive operating modes, where only one substate can be active at each time step. On the other hand, when a state has parallel decomposition, all substates operate at the same time step, but it is crucial to define the proper *Execution Order* among those substates.

For example, in Figure 2.5, a superstate and three substates are shown. The superstate has a parallel execution with respect to the other substates that are not shown in the Figure, but that make up the overall Stateflow® block: this is recognisable by the dotted lines and the number in the top right-hand corner that identifies the order of execution of that state. As for its substates, they have an exclusive execution, as shown by the solid lines on their border; also, by chance in this example, all these substates are characterised by the entry state action.

- *Transitions* are Stateflow® objects that determine the activity or inactivity of a state by connecting the latter, with one or more arrows (transitions), into or out of itself, with one or more states.

  To establish a criteria for switching between states, each transition arrow can be

accompanied by one or more transition actions. These actions are identified by label syntax and definition order, as demonstrated in Figure 2.6.

```
trigger[condition]{condition_action}/{transition_action}
```

Figure 2.6: Transition Label [25]

There are four main types of transition actions that can be classified as follow:

1. *Events (or Trigger)* are objects that can activate actions in a parallel state within a Stateflow® chart or in another Stateflow® chart. There are two types of events, implicit and explicit. Focusing solely on the explicit events, i.e. those declared by the user, they can be classified into three distinct categories [22] :

   (a) *Input* events, when the event transmitted to a chart came from outside the Stateflow® block.

   (b) *Local* events, when the event may occur anywhere within the chart, but is visible only within the parent object and its descendants. The level of hierarchy visibility of those events may be specified through the *Model Explorer* GUI; Visibility can be assigned according to specific needs, from upper superstates to lower substates, providing a high level of flexibility in modelling the logics.

   (c) *Exit* events, when the event occurs within the Stateflow® chart but their effects are transmitted outside of it.

   It should be noted that events, unlike the other three transition actions described in the following, need to be well defined in the *Symbols Pane* prior to being used. Furthermore, as depicted in Figure 2.5, events can be identified as distinct from other transitions by the absence of brackets and their orange color if visibility is properly configured. It is evident that events must be triggered somewhere, and one possible means of doing so is through *Condition Actions* or *Transition Actions*, also illustrated in Figure 2.5.

2. *Conditions* are Boolean expressions that determine the occurrence of a transition. In transition label syntax, conditions must be enclosed in square brackets. Logical operators can be employed to define conditions, and more complex conditions can be specified by invoking *Graphical Functions*, *Truth Table Functions*, *MATLAB® functions*, or *Simulink® functions* that return a numeric

value.

3. *Conditions Actions* are actions specified in curly braces that are executed when the condition guarding the transition becomes true, but before the destination of the transition is determined to be valid.

4. *Transition Actions* are actions that differ from the previous ones in that they are specified in curly brackets preceded by a forward slash and their execution occurs when the entire transition path is considered valid. Thus, in the case where the transition is part of a transition path that consists of several segments, the transition action is executed if the path reaches the destination state or an end junction.

- *Default Transitions* are a special type of transition that lacks a source. It serves to resolve ambiguity among two or more neighboring exclusive (OR) states by indicating which state to enter firstly, as shown in the state at the top of the Figure 2.5.

- *Junctions* are representative of decision points, as they facilitate the creation of transition paths consisting of multiple transition segments. Such paths can be established, for instance, from a single source to multiple destinations or from several sources to a singular destination. In such scenarios, any intermediate transitions necessitate the inclusion of a connective junction as either a source or destination. In the Stateflow® environment, connective junctions can be utilized as the foundation for constructing fundamental programming objects, such as for loops, while loops, if-else statements, and similar constructs. To assist with the development of these common flow chart patterns, the Stateflow® platform includes the *Pattern Wizard* tool, an integrated feature designed specifically for this purpose [17].

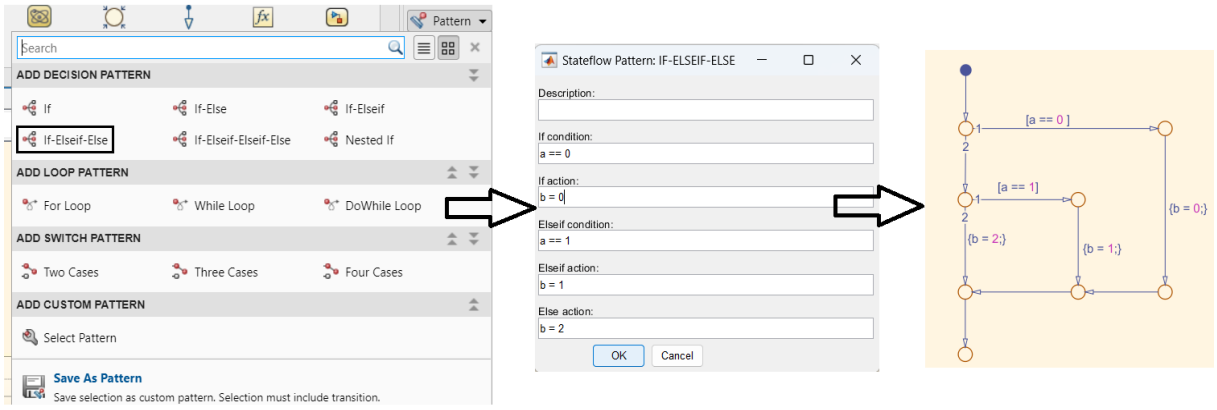

Figure 2.7: Example of usage of Stateflow® Pattern Wizard

- *MATLAB® Functions* are objects within Stateflow® that enable the creation of algorithms in the *MATLAB Editor* while storing them inside the Stateflow® chart itself. These functions are particularly useful when graphical representation using charts is too restrictive, making it easier to directly implement MATLAB® code. It is important to note that *MATLAB® Function* may access data not only from the specified input and output data, but also from their parent. Therefore, it is important to check the data properties to verify that the data is gathered and utilized as expected. This can be done by opening in the *Model Explorer* the Stateflow® containing the *MATLAB® Function* of interest: here, data properties can be viewed and modified if necessary.

Scope

Location where data resides in memory, relative to its parent.

Setting	Description
`Local`	Data defined in the current chart only.
`Constant`	Read-only constant value that is visible to the parent Stateflow object and its children.
`Parameter`	Constant whose value is defined in the MATLAB® base workspace or derived from a Simulink block parameter that you define and initialize in the parent masked subsystem. The Stateflow data object must have the same name as the MATLAB variable or the Simulink parameter. For more information, see Share Parameters with Simulink and the MATLAB Workspace.
`Input`	Input argument to a function if the parent is a graphical function, truth table, or MATLAB function. Otherwise, the Simulink model provides the data to the chart through an input port on the Stateflow block. For more information, see Share Input and Output Data with Simulink.
`Output`	Return value of a function if the parent is a graphical function, truth table, or MATLAB function. Otherwise, the chart provides the data to the Simulink model through an output port on the Stateflow block. For more information, see Share Input and Output Data with Simulink.
`Data Store Memory`	Data object that binds to a Simulink data store, which is a signal that functions like a global variable. All blocks in a model can access that signal. This binding allows the chart to read and write to the Simulink data store, sharing global data with the model. The Stateflow object must have the same name as the Simulink data store. For more information, see Access Data Store Memory from a Chart.
`Temporary`	Data that persists during only the execution of a function. You can define temporary data only for graphical functions, truth tables, or MATLAB functions in charts that use C as the action language.

Figure 2.8: Data memory allocation in Statflow® [23]

One crucial data property is the *scope* field, which indicates the allocation of data with respect to its parent and whose classification is shown in Figure 2.8. Additionally, users can modify data directly in the Stateflow® environment using the *Property Inspector* tab, while another possibility for managing few data in charts is through the *Symbols Pane* interface.

2.4 Code Generation

The *Model-Based Design* approach offers significant advantages for the development of complex projects, including the ability to automate processes through the automatic generation of code from system models. This approach greatly reduces development time and minimizes the risk of errors. Indeed, in traditional system design, the need of manual

programming is typically onerous and laborious, but the use of Simulink®, as a *Model-Based Design* software, enables the creation of models that can be translated into *C* or *C++* code documents and executed on various platforms, including embedded processors.

Aligned with the methodology taken in the previous thesis [32], the current research necessitates the development of a Simulink® model that is capable of being exported as *C++* code. This is vital to enable the future integration of the AFCS model into the Flight Simulator of *TXT e-solutions*. In this regard, the current work builds upon the practical methodology highlighted in [32] and aspires to construct a Simulink® model equipped with all the essential features to be exported utilizing the Embedded Coder® tool of Simulink®. To ensure that the model can be exported without any issues, the same system design philosophy utilized in [32] is followed as closely as possible. Although this approach may sometimes impose limitations on subsystem design, it provides a guarantee about the safe export of the model.

Given the design strategy just outlined, the research goal exclusively focuses on developing the Simulink® model. Therefore, this section does not describe the features of the Embedded Coder®, as it is not actually used in the present work but implicitly imposes restrictions on it.

2.4.1 Time Step Issue

The time step is a parameter that determines the accuracy and efficiency of a simulation, and it is dependent on the hardware platform (e.g. industrial control systems such as programmable logic controllers (PLCs), Embedded systems such as microcontrollers and field-programmable gate arrays (FPGAs), ecc.) on which the simulation software is installed. Different hardware platforms operate at different frequencies, which means that the time step may need to be adjusted to ensure that the simulation is accurate and efficient for a specific application. Therefore, it is crucial that this parameter is easily manageable directly in the simulation software.

In Simulink® environment, the time step pertains to the magnitude of time for which the Simulink® model is executed and processed during each simulation step. Given its relevancy, this parameter can be easily modified and accessed by the user through the *Solver* section in the *Model Settings* interface of Simulink®. Although this operation is straightforward remaining within the boundaries of Simulink®, during the transition to the generated code, the management of the time step results much more restrictive.

Indeed, it is worth considering a simple Simulink® model, as depicted in Figure 2.9. It is required that this model be highly customizable for the end-user and must also be

exportable to *C++* code. To meet these requirements, the model's parameters must be defined as *Tunable*, allowing them to be exported as *Tunable* as well.

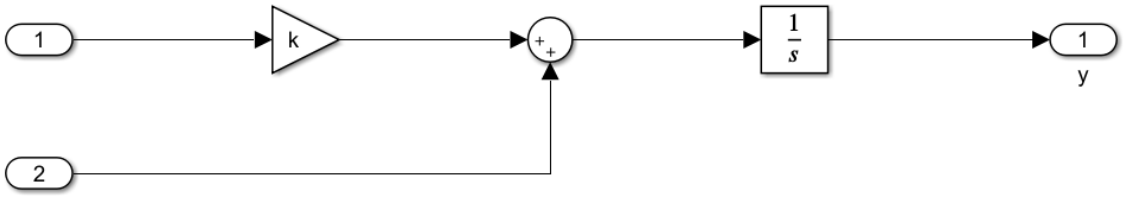

Figure 2.9: Example Model for the time step issue

The simple Simulink® model shown in Figure 2.9 has only two parameters, namely the *Gain* value k and the simulation time step dt. The latter parameter should be defined within the *Solver*, after specifying that the simulation solver will have a constant simulation time step (*Fixed-step*). This setting is fundamental as in practical applications different constant time step amplitudes may be used depending on specific hardware. Following this, the numerical values for these parameters must be defined within the *MATLAB® Workspace*.

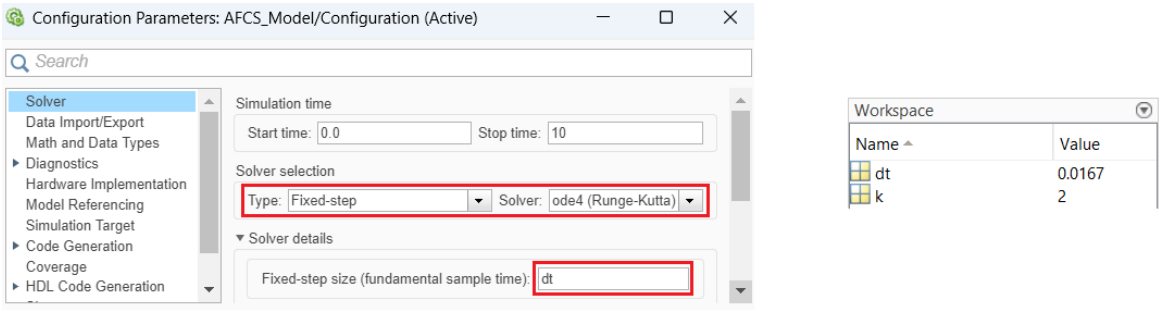

Figure 2.10: Simulink® settings for *Solver* type and time step value

Prior to generating the code, it is necessary to configure the tunability of the parameters and their storage class in the *Optimization* panel, as well as properly set up the *Code Generation* panel.

The *Inline* and *Tunable* behaviors [11] are available for the parameters, and the user must choose one. If the inlined behavior is selected, then the time step is replaced inline within the code, which makes it impossible to parameterize it: the programmer would have to modify it within the *C++* solution, which is not feasible. Indeed, it is important to note that the generated software should be managed as if it were being used by a

customer who does not have a MATLAB/Simulink license, as it is purely a development tool and not part of the final product. To achieve this, the time step and other custom parameters in the system must be defined as *Tunable*.

Additionally, the configuration of how these *Tunable* parameters must be exported in the *C++* code is also necessary. Various solutions may be applied, but the most convenient approach in this specific case may be to set the storage class to *ImportedExtern* [26] [16]. Doing so allows the declaration of the *Global variable* in the generated file *filenameprivate.h* when the code is exported. However, the assignment of the values of those variables must be set externally using, for example, an external library. This approach preserves the code structure and enables the user to go back to the Simulink® model, change the parameter values, perform some tests, and then return to the exported version of the *C++* solution with an updated parameters list. Clearly, these parameters must be properly allocated in an external file when saved from Matlab®/Simulink®.

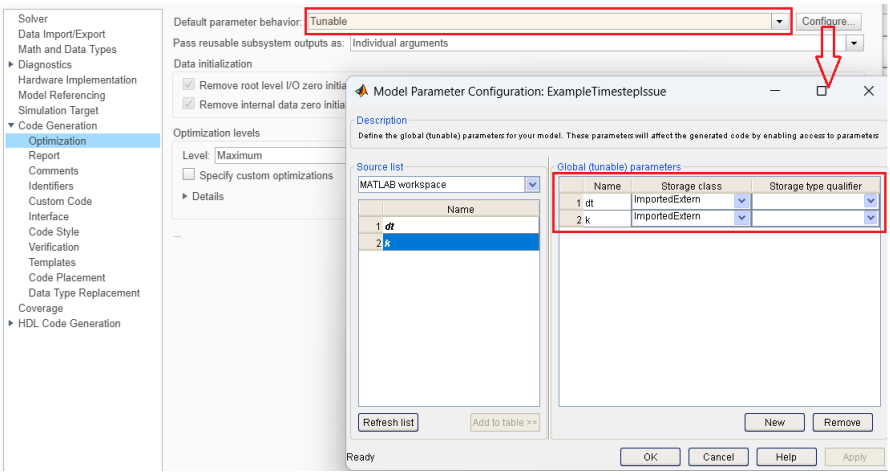

Figure 2.11: Simulink® settings for parameters tunability and storage class

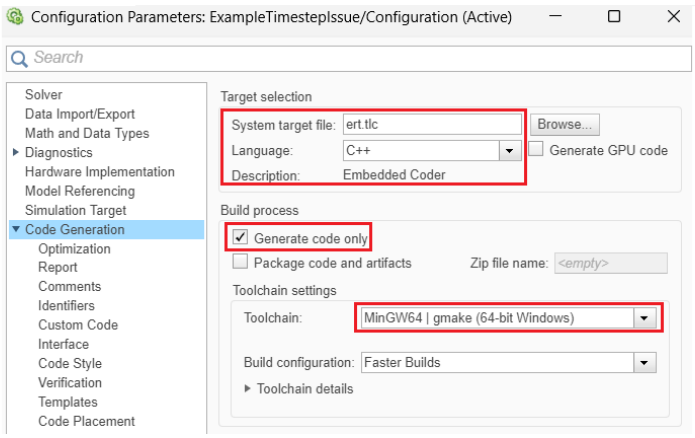

Figure 2.12: Simulink® settings for *C++* code generation

Finally, the *C++* solution can be generated using the *Embedded Coder*® [13]. However, as illustrated in Figure 2.13, an error occurs when attempting to generate code with the aforementioned features.

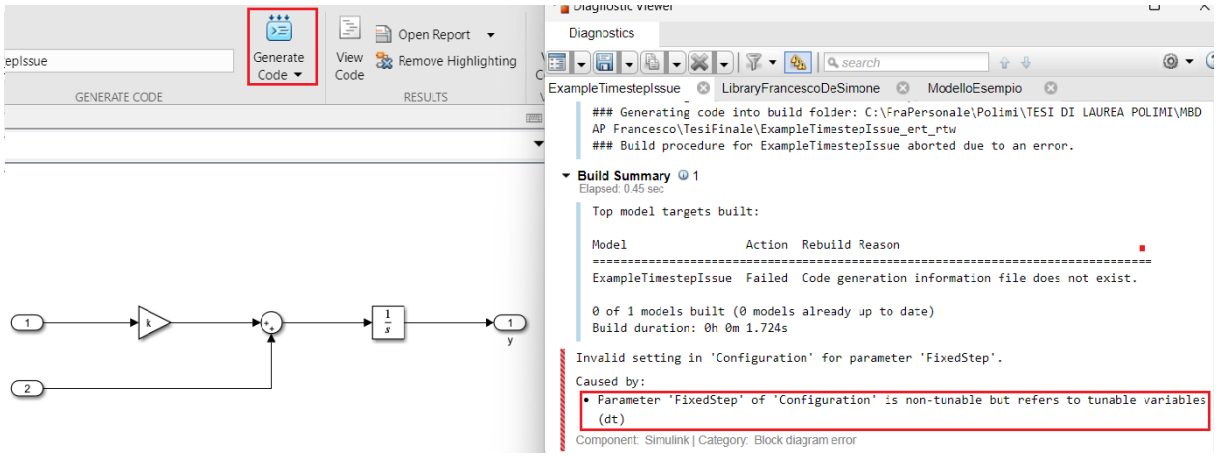

Figure 2.13: Code Generation Error

2.4.2 Custom Solver Algorithm

The error is linked to the presence of a tunable parameter as a time step value in the *Fixed-step Solver*. To overcome this problem, one practical solution is developed in the previous work [32] and is also employed in the present thesis.

In practice, when simulating a dynamic system, i.e. computing its states at successive time steps over a specified time span, Simulink® employs the *Solver*. The *Solver* applies a numerical method to solve the set of ordinary differential equations that represent the model and determines the time of the next simulation step. The *Solver* also satisfies the accuracy requirements specified by the user while solving this initial value problem.

If the AFCS Simulink® model does not incorporate *Library Browser* blocks that use the specified *Solver* time step for their task, the *Solver*'s time step can be replaced with an arbitrary numerical value. In such a scenario, when generating the code, only the file *ert_main.cpp* would employ the *Solver* time step information. However, this file, which defines the necessary statements for initializing, executing, and terminating the model, is not essential for integrating the generated code within a real flight simulator system as it already has its own configurations file, among which, also the flight simulator time step value is present.

However, developing a control system without using any component that requires the usage of the *Solver* is impossible. Therefore, every time the Simulink® model needs a

block that employs the usage of the *Solver*, the block is rebuilt from scratch. Indeed, the mask contents of various *Library Browser* blocks can be accessed by the designer, making it easier to create a customized version of them. The custom version is required to perform the same task as the standard block while also having the time step as a custom parameter, assignable from an external environment.

In order to rebuild the standard *Library Browser* blocks that cannot be used in the AFCS model, it is essential to choose a numerical algorithm for performing integration. This algorithm may be employed to rebuild standard Simulink® blocks like PID controllers, filters, and transfer functions.

In general, various numerical integration schemes are available in the literature, each with its own distinct properties that may be more or less suitable for a given application. For this thesis, the same algorithm utilized in the previous work [32] is adopted to perform numerical integration since it satisfies the same application requirements.

Runge-Kutta 4-th Order

Consider the following initial value problem:

$$\begin{cases} \dot{x}(t) = f(t, x(t)), \quad t \in [t_0, T], \\ x(t_0) = x_0, \end{cases}$$

a numerical solution is able to generate a sequence of values $\{x_n\}_{n=0}^{N}$ such that, at time t_n, x_n is an approximation of the exact solution:

$$x_n \approx x(t_n) \quad \forall n.$$

The well-known family of *Runge-Kutta* methods are single-step, that is, the algorithm calculate x_{n+1} through a continuos function that involves the step-size $h_n := t_{n+1} - t_n$ and the pair of values (t_n, x_n).
The idea of the German mathematicians Runge and Kutta was to develop new schemes that were able to extend the classic *forward Euler* method, providing also greater accuracy. Since the advent of digital computers, researchers contributed to build the theory around Runge-Kutta methods and to widen this family of numerical solvers. Indeed, early works proposed explicit Runge–Kutta methods, but implicit ones have been studied and now are included in the same class.

The recursive equations of a Runge-Kutta method with s stage is:

$$\begin{cases} x_{n+1} = x_n + h_n \sum_{i=1}^{s} b_i k_i, & n = 0, \dots, N-1, \\ k_i = f(t_n + h_n c_i, x_n + h_n \sum_{j=1}^{s} a_{ij} k_j), & i = 1, \dots, s, \end{cases} \tag{2.1}$$

by a_{ij}, b_i and c_i are denoted, respectively, the elements of a matrix $A \in \mathbb{R}^{s \times s}$, vector $\boldsymbol{b} \in \mathbb{R}^s$ and vector $\boldsymbol{c} \in \mathbb{R}^s$; generally expressed via the *Butcher tableau* as in the following grid.

$$\begin{array}{c|c} \boldsymbol{c} & A \\ \hline & \boldsymbol{b}^T \end{array}$$

When the method is explicit and the step-size h is fixed, the system (2.1) is substituted by:

$$\begin{cases} x_{n+1} = x_n + h \sum_{i=1}^{s} b_i k_i, & n = 0, \dots, N-1, \\ k_1 = f(t_n + h c_1, x_n), & \\ k_i = f(t_n + h c_i, x_n + h \sum_{j=1}^{i-1} a_{ij} k_j), & i = 2, \dots, s; \end{cases} \tag{2.2}$$

therefore, for the computation, is necessary to define just the lower-triangular part of matrix A.

One of the most famous and commonly used Runge-Kutta method has four stage (RK4) because it uses four function evaluations every step. It is characterized by the following Butcher tableau:

$$\begin{array}{c|cccc} 0 & & & & \\ 1/2 & 1/2 & & & \\ 1/2 & 0 & 1/2 & & \\ 1 & 0 & 0 & 1 & \\ \hline & 1/6 & 2/6 & 2/6 & 1/6 \end{array}$$

and the resulting algorithm is:

$$\begin{aligned} k_1 &= f(t_n, x_n), \\ k_2 &= f(t_n + \frac{h}{2}, x_n + \frac{h}{2}k_1), \\ k_3 &= f(t_n + \frac{h}{2}, x_n + \frac{h}{2}k_2), \\ k_4 &= f(t_n + h, x_n + hk_3), \\ x_{n+1} &= x_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4), \\ t_{n+1} &= t_n + h. \end{aligned}$$

It is important to highlight that RK4 satisfies a property that holds for all Runge-Kutta algorithms and guarantees the consistency of the methods, that is:

$$\sum_{i=1}^{s} b_i = 1.$$

Moreover, RK4 has order of accuracy 4, indeed the truncation error is $\mathcal{O}(h^4)$. It has been proven [2] that one, two, three, four stages yield methods of order one, two, three, and four, respectively. But above order 4, it is no longer possible to obtain order s with just s stages, hence the addition of another stage does not produce an effective improvement in terms of accuracy; this feature makes RK4 especially convenient.
However, higher order method are still remarkable when considering stability region. The latter is defined as the set of points in the complex plane for which the stability function $R(z)$ has magnitude bounded by 1: $\{z \in \mathbb{C} : |R(z)| \leq 1\}$. The expression of stability function for RK4 is:

$$R(z) = 1 + z + \frac{1}{2}z^2 + \frac{1}{6}z^3 + \frac{1}{24}z^4$$

and the stability region is shown in figure 2.14.

In the present analysis, the function f depends linearly only by the state vector $\boldsymbol{x}$ and is embedded in a linear time-invariant SISO system of the form:

$$\begin{cases} \dot{\boldsymbol{x}} = A\boldsymbol{x} + Bu \\ y = C\boldsymbol{x} + Du \end{cases}.$$

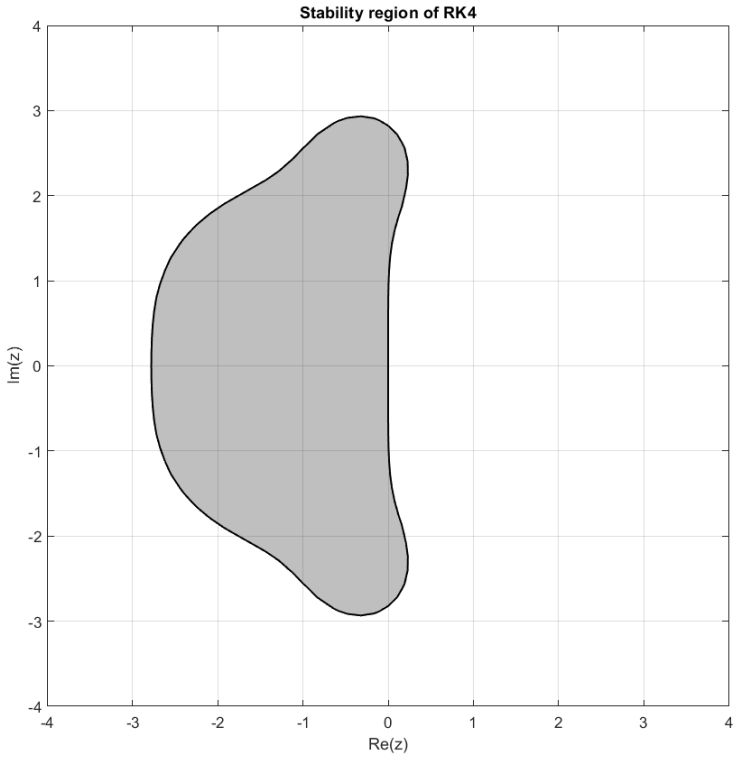

Figure 2.14: Stability region of explicit Runge-Kutta methods with order 4

The application of RK4 to this system produce the following algorithm:

$$
\begin{aligned}
\boldsymbol{k}_1 &= A\boldsymbol{x}_n + Bu_n, \\
\boldsymbol{k}_2 &= A(\boldsymbol{x}_n + \frac{h}{2}\boldsymbol{k}_1) + Bu_{n+\frac{1}{2}}, \\
\boldsymbol{k}_3 &= A(\boldsymbol{x}_n + \frac{h}{2}\boldsymbol{k}_2) + Bu_{n+\frac{1}{2}}, \\
\boldsymbol{k}_4 &= A(\boldsymbol{x}_n + h\boldsymbol{k}_3) + Bu_{n+1}, \\
\boldsymbol{x}_{n+1} &= \boldsymbol{x}_n + \frac{h}{6}(\boldsymbol{k}_1 + 2\boldsymbol{k}_2 + 2\boldsymbol{k}_3 + \boldsymbol{k}_4), \\
y_{n+1} &= C\boldsymbol{x}_{n+1} + Du_{n+1},
\end{aligned}
$$

where $u_{n+\frac{1}{2}}$ is approximated with the mean value between u_n and u_{n+1}. Lastly, the step-size h need to be chosen so that, $\forall\lambda \in \sigma(A)$, $h\lambda$ belongs to the stability region $\{z \in \mathbb{C} : |R(z)| \leq 1\}$.

The algorithm can be stored in a *MATLAB function* , as shown in Figure 2.15, allowing for easy access to the numerical integration scheme whenever it is needed. More details regarding the application of this algorithm are provided in Section 3.3.2.

```
function [st, out] = RK4(AMat, BMat, CMat, DMat, st0, in0, in)

inmed = (in0 + in)/2; % mid-steptime input

k1 = AMat*st0 + BMat*in0;
k2 = AMat*(st0 + k1*dt/2) + BMat*inmed;
k3 = AMat*(st0 + k2*dt/2) + BMat*inmed;
k4 = AMat*(st0 + k3*dt) + BMat*in;

st = st0 + dt/6*(k1 + 2*k2 + 2*k3 + k4); % state approximation
out = CMat*st + DMat*in; % output evaluation
end
```

Figure 2.15: RK4 *MATLAB function*

Code Generation Problem: Solved

Regarding the error in Figure 2.13, the Simulink® model used to generate the code has a standard *Integrator* block that employ the time step information declared in the *Solver* at each step. To resolve the code generation issue, it's necessary to create a custom version of this block that uses the externally provided time step value and performs the integration using the RK4 Method explained earlier. Details on creating the custom block are provided in the following section. For now, it's just worth noting that all code generation settings remain the same, except for the ability to set randomly both the time step size and the *Solver* numerical scheme.

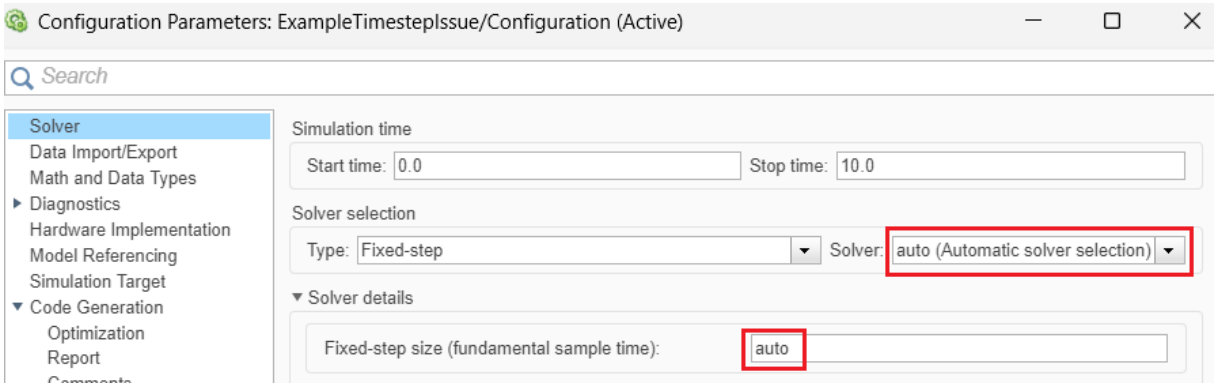

Figure 2.16: New Simulink® settings for *Solver* type and time step value

Finally, requesting code generation provides a solution without any errors. As shown in Figure 2.17, this solution allows declaring the time step variable without specifying its numerical value, which must be provided from an external source, as requested in the *Storage Class* definition.

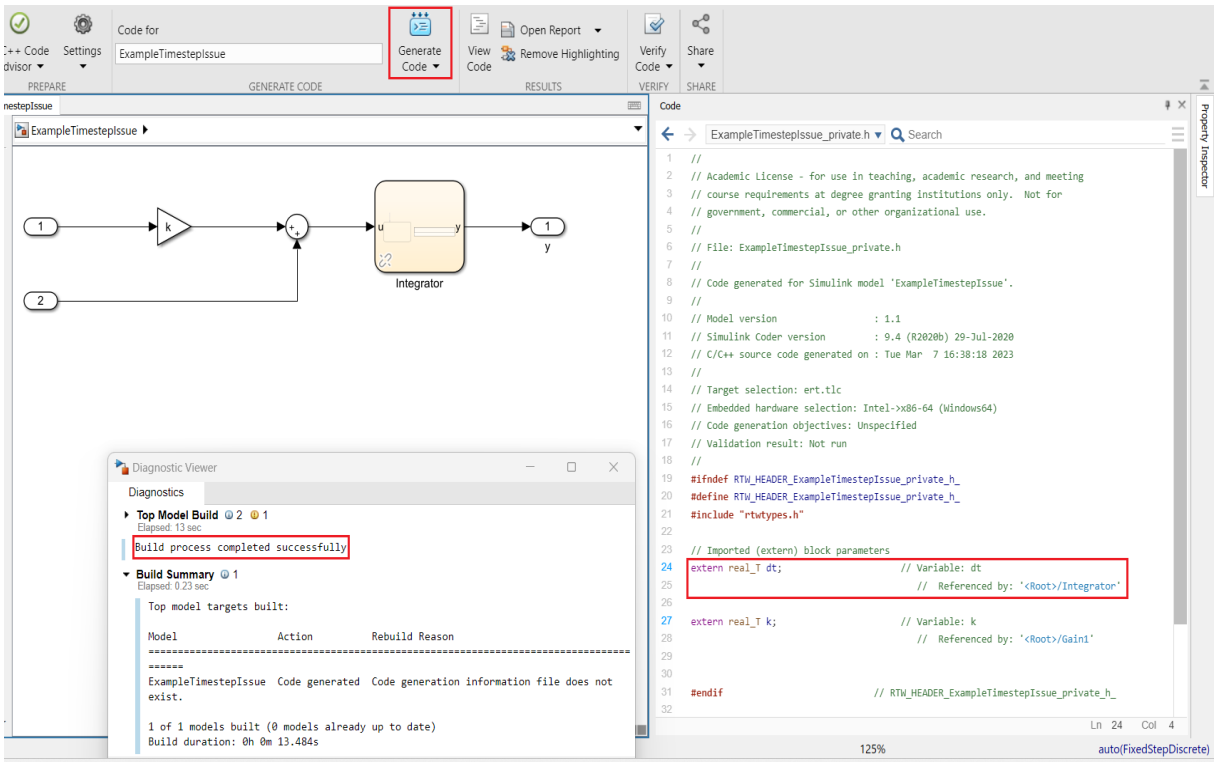

Figure 2.17: Succesful Code Generation: Tunable Variable Declaration

The custom RK4 algorithm that performs the integration is stored in the *filename.cpp* file. As depicted in Figure 2.18, the numerical scheme uses the custom time step to carry out the integration, effectively resolving the issue associated with the *Tunable* time step.

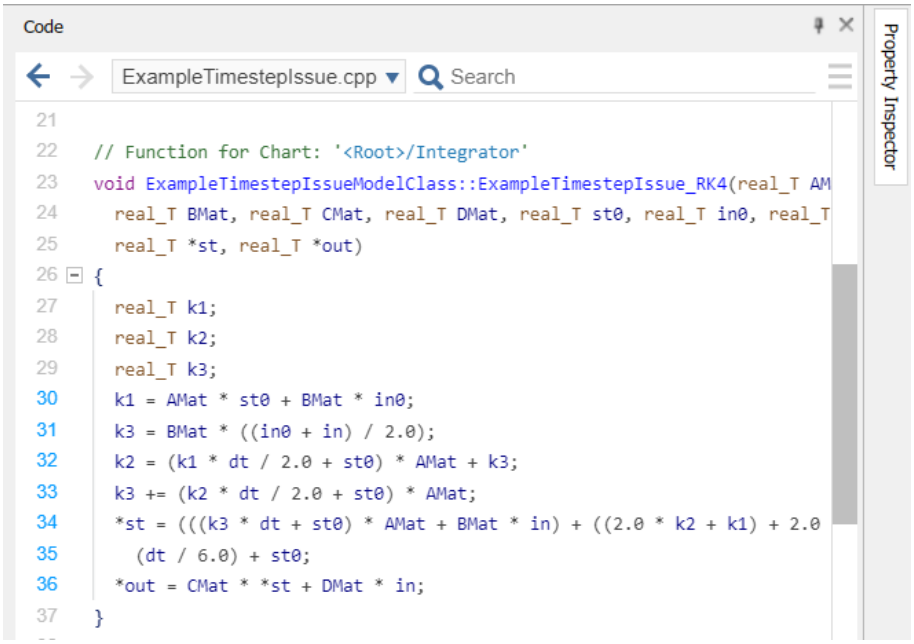

Figure 2.18: Succesful Code Generation: RK4 Algorithm

3 | AFCS Modeling in Simulink

The modeling phase in the model-based approach is a critical component of the development process. The phase involves creating a complex model that represents the real system using several blocks connected to each other on different hierarchical layers. It also involves identifying the system's inputs, outputs, states, and transitions, as well as its internal and external interfaces. The model must ensure that the real behavior, structure, and interactions with the environment are as closely represented as possible. Clearly, the accuracy of the model increases with a more detailed modeling, but this decreases its efficiency during simulation and increases its complexity. Therefore, finding the right trade-off among these considerations is crucial during the modeling phase. Above all, accurately representing the system's entry requirements and constraints are key features of successful modeling.

Now, by focusing on this application, the primary aim of this thesis is to develop an AFCS model with a flexible structure that can be quickly adapted to any helicopter system architecture. In order to meet those requirements, the development of the AFCS model must include:

- the creation of custom blocks with the highest possible level of system modularity, allowing for the reuse of small but key pieces of models in future AFCS developments.
- the modeling of generic blocks with the highest possible level of customization in order to concede the parametric definition of helicopter AFCS variables.

Therefore, this chapter presents the structure of each *Custom Library* block, its main purpose, and the Simulink® tools utilized to build it up.

3.1 Push Button Block

Generally, in a helicopter's Autopilot Control Panel, such as the one depicted in Figure 1.7, there exist numerous push buttons and rotary knobs, as well as other triggering devices or switches, depending on the particular helicopter under analysis. Given that push buttons constitute the most fundamental triggers that are invariably present in the

APCP, this study only consider such types of switches as they enable the most basic function of the APCP, i.e., the engagement or disengagement of flight modes.

The purpose of the *Push Button Block* is to enable the detection of a logical event that occurs when a button is pressed in the helicopter's APCP. As the act of pressing the button is not instantaneous, but rather occurs over a finite period of time, the input from the user can be considered a Boolean history where a value of 1 is assigned when the button is pressed and 0 otherwise. However, it is only necessary to detect the event of the button being pressed and not the entire history, since this event is the key factor for the engagement or disengagement of modes in the *Modes Logics Stateflow*. Thus, the *Push Button Block* has been designed to produce an output that reflects only the button press event, with a value of 0 at every time step except for those when the input switches from 0 to 1, but not the vice versa.

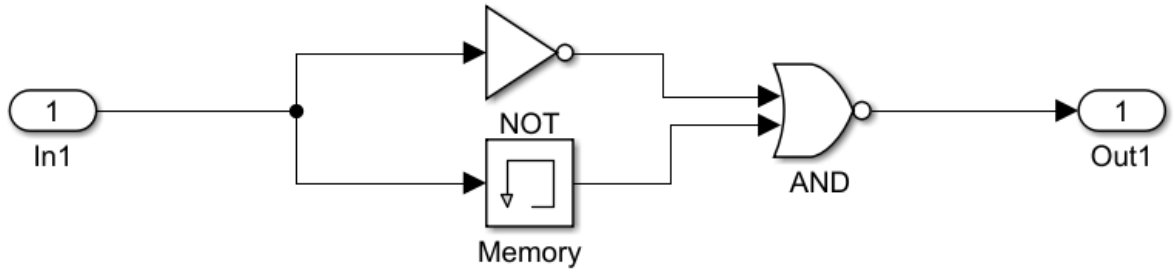

Figure 3.1: Push Button Block Structure

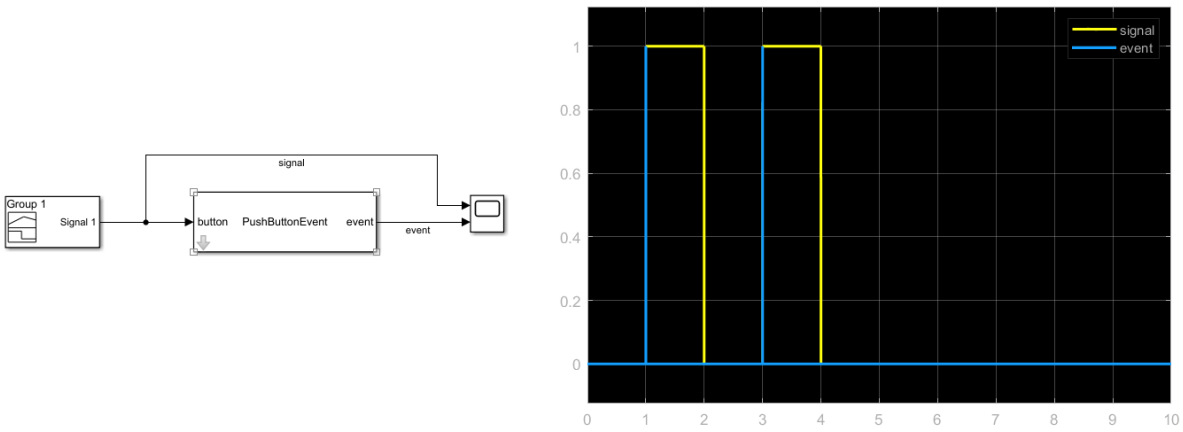

Figure 3.2: Push Button Block Example Application

3.2 Modes Logics Stateflow®

Autopilot upper mode transitions may differ depending on the helicopter model. Examining flight manuals from various manufacturers, common operating characteristics of the logics were identified along with some distinguishing features. Generally, the logics that identify those transitions can be categorized into two main groups:

- Manual engagement/disengagement, in which the pilot selects the status of an AP mode manually through the APCP
- Automatic engagement/disengagement, in which the software automatically switches the status of an AP mode based on certain factors, such as:
    1. the flight conditions in which the helicopter is operating (e.g., airspeed, altitude, attitude limitations, etc.)
    2. the compatibility of different modes to be simultaneously activated (e.g., the autopilot cannot engage two upper modes that operate on the same axis)
    3. the disengagement of a specific mode (e.g. some modes have a backup mode)
    4. the loss of an actuator on a specific axis
    5. the loss of a specific sensor data
    6. The presence of a *Reset Function* in the AP, which can trigger the deactivation of the current modes and the activation of the traditional upper modes (IAS, HDG, ALT) all at once.

In creating the automatic engagement/disengagement logics, it has been decided not to consider the loss of an actuator or specific sensor data as a discriminating factor. These reasons are considered secondary and handled differently, depending on the selected helicopter type. Additionally, the presence of the *Reset Function* is not taken into account since it is optional and varies depending on the helicopter model. Instead, the design objective is to allow the transition logics to cater for various helicopter models without being specific to any of them. To this end, logics are created with the ability to parameterise certain discriminating input parameters that may vary according to the type of helicopter considered.

To develop a dynamic system capable of transitioning from one operating mode to another, state machines are considered the optimal tool. As a result, the *Modes Logics Block* has been developed using Stateflow®, as illustrated in Figure 3.3.

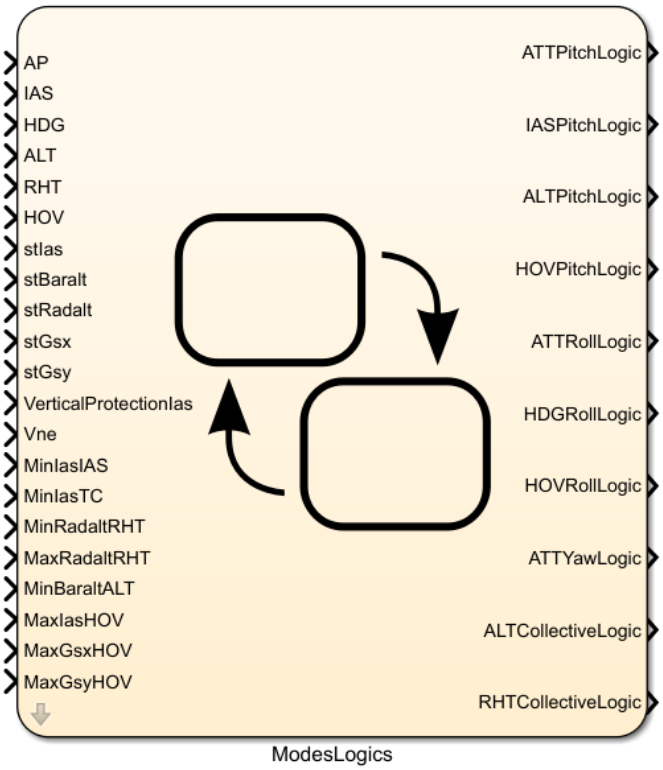

Figure 3.3: Modes Logics Stateflow® Block

By accessing the mask of this block, the Symbols Pane interface depicted in Figure 3.4 can be accessed. Within this pane, it is possible to clearly view and manage:

- Inputs, consisting of Boolean values associated with the activation/deactivation of an APCP button, measurements of certain helicopter dynamical parameters, and static parameters that define mode thresholds.
- Outputs, which are Boolean values dependent on the status of a specific mode on a particular axis.
- Local events occurring within the state that are triggered elsewhere.

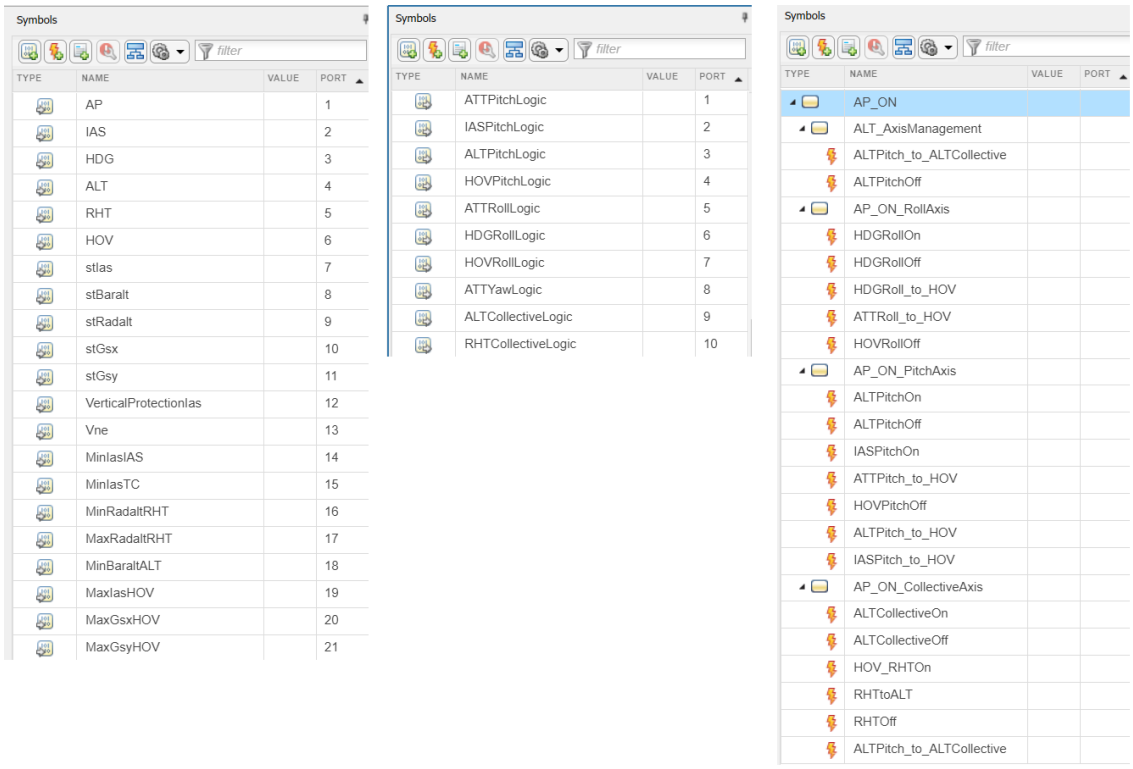

Figure 3.4: Modes Logics Stateflow® : inputs (left), outputs (middle), events (right)

Instead, regarding the core structure of the *Modes Logics Block*, it is possible to divide the state transition diagram in three sections:

1. in Figure 3.5, the AP's power-off status. When the helicopter is switched on, first the AFCS is switched off and only after pressing the AP button on the APCP can the AFCS be switched on. In addition, if the power-on state of the AP is activated, pressing the AP button turns the AFCS off.

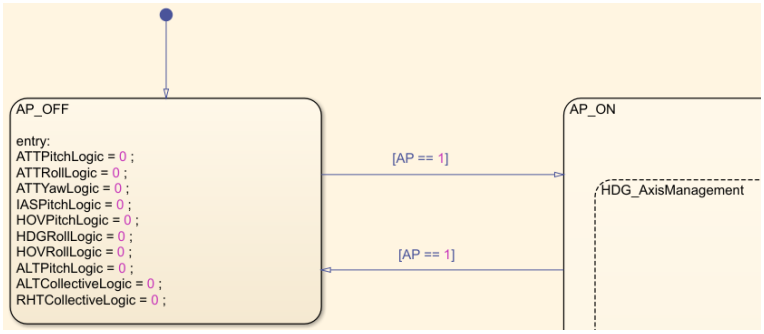

Figure 3.5: Modes Logics Stateflow® : AP off

2. in Figure 3.6, the management states of complex modes, which occur, of course, if the AP is engaged. As explained in Chapter 1, certain AP modes can be engaged on

different axes (e.g. ALT or HDG), and some require the engagement of other modes (e.g. HOV). Due to the complexity of these modes, they are managed separately.

Specifically, managing multi-axis modes separately helps to determine the operating axis and also facilitate the switching among axes during simulation. This approach also provides better visualization of the mode transition on different axes. Instead, managing separately the logic of modes that require the engagement of multiple modes results in a more concise decision logic rather than defining each condition on a specific axis.

It is worth noting that the states depicted in the figure below are sub-states of the AP_ON superstate and are decomposed in a *Parallel* fashion. During simulation, all of these states are executed within a single time step and must follow a specific *Execution Order*. Specifically, these states must be executed before the transition diagram reaches the states managing mode engagement on each helicopter axis. This is essential to determine the status of these complex modes beforehand and to trigger their activation or deactivation conditions.

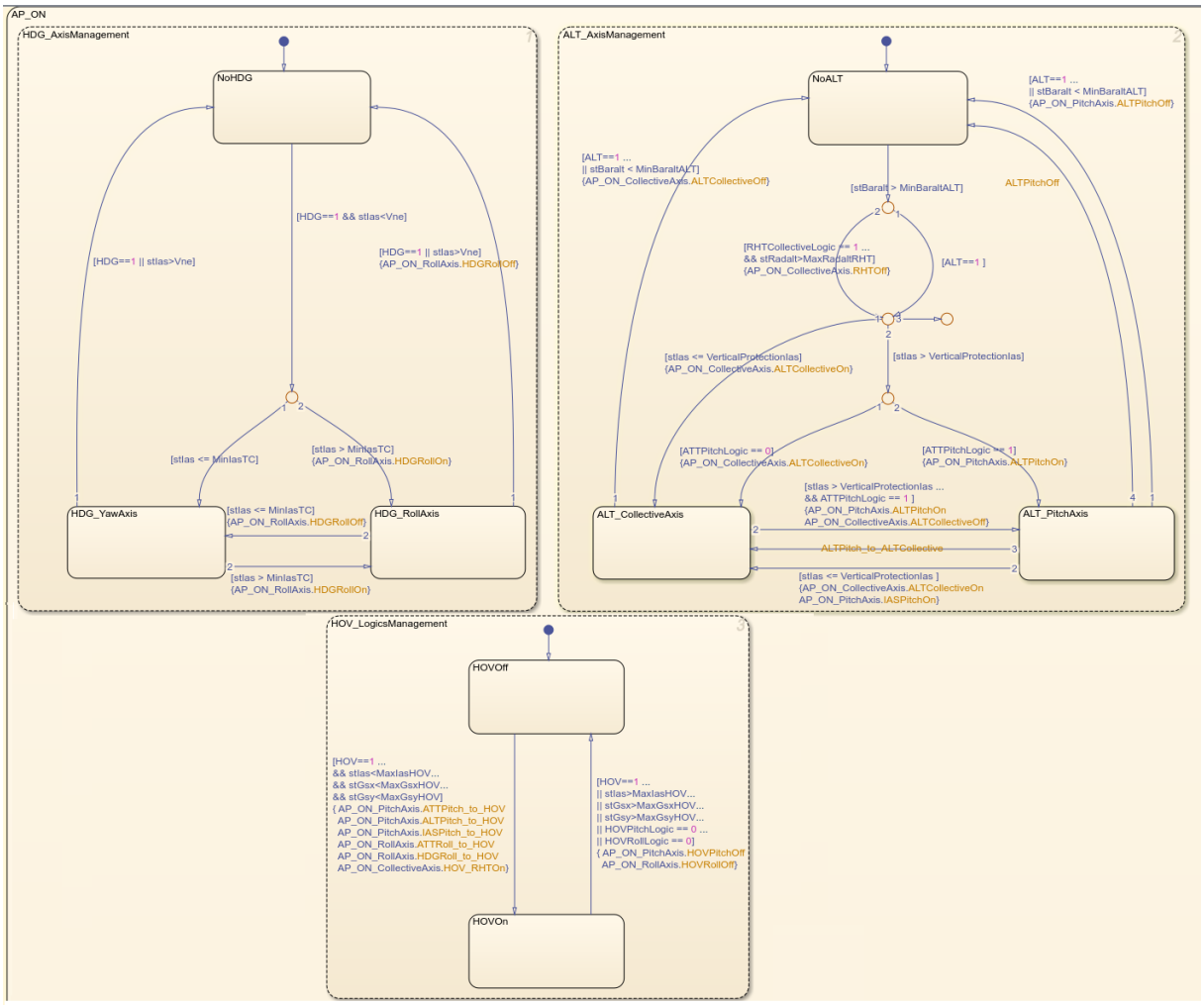

Figure 3.6: Modes Logics Stateflow® : AP on, Management States

3. in Figure 3.7, the states managing mode engagement on each helicopter axis, which occur, of course, if the AP is engaged. The helicopter axes include the pitch, roll, yaw, and collective axes, and a mode can be engaged for each axis based on the pilot's needs.

   During each time step, the transition logic is evaluated, and child states of the same axis superstate (e.g. *ALT,ATT, HOV,IAS* within *AP_ON_PitchAxis*) may switch among themselves or remain in the same state as the previous time step. If a switch occurs from one state to another, the Boolean logic specified in the new state is evaluated as an *entry* action.

   It is also worth noting that these states are decomposed in a *Parallel* manner, and the *Execution Order* has been specified to occur after the management of complex modes, as mentioned earlier.

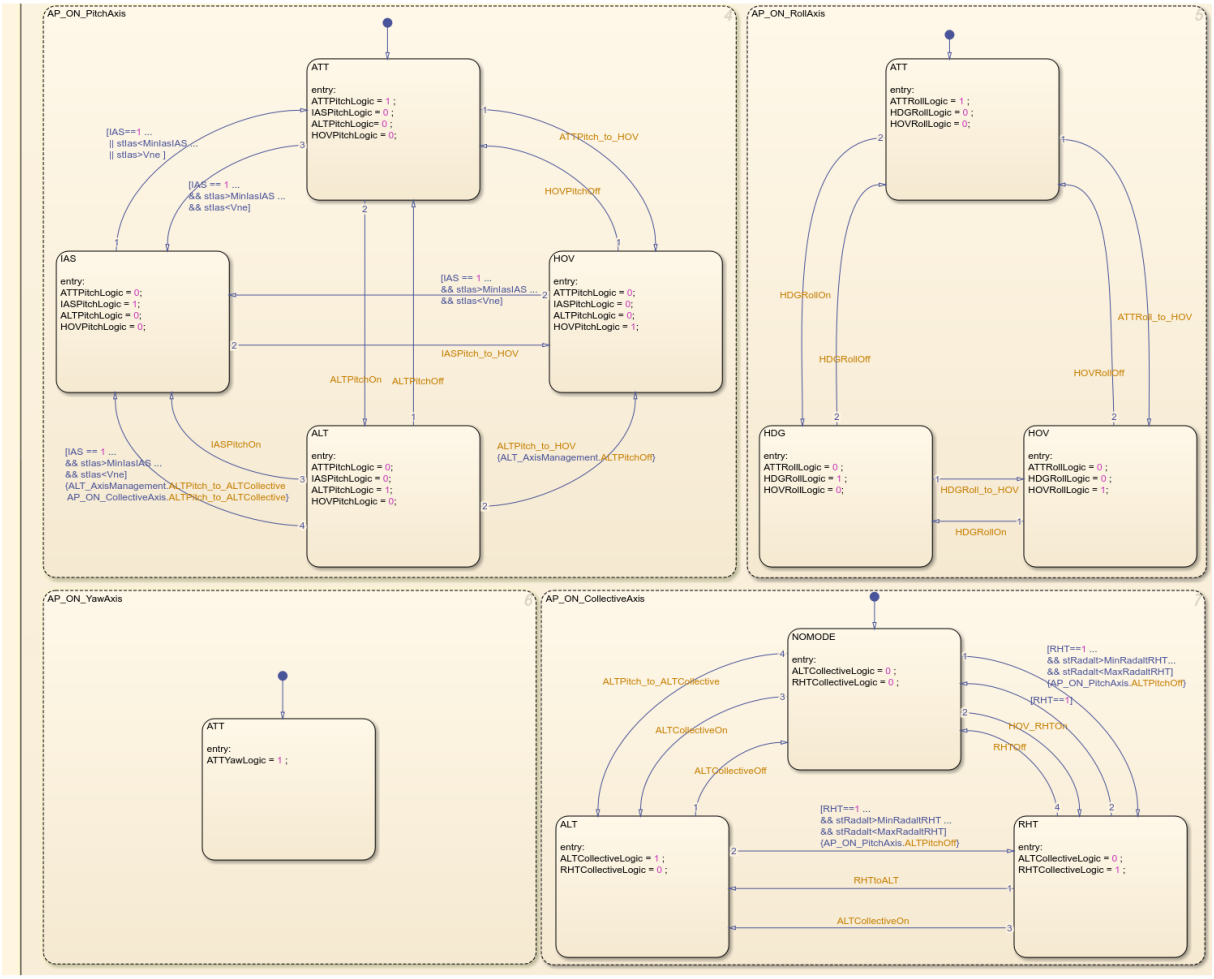

Figure 3.7: Modes Logics Stateflow® : AP on, Axis States

3.3 PID Block & Anti-windup Block

Following the analysis of PID controllers in Section 1.2.2, a two degree of freedom PID Controller has been chosen as the appropriate controller for the generic AFCS due to its superior customization options compared to the standard PID Controller structure. Indeed, by simply setting *b=1, c=1*, the standard PID configuration can be restored.

The *PID Controller (2DOF)* block is readily available in the Simulink® library, but it uses the *time step* information declared in the *Solver* options to compute the integration. Thus, in order to treat this information as a *Tunable* parameter and define the time step from the block's mask, it is necessary to recreate the block from scratch. Additionally, the *PID Controller (2DOF)* library block offers the option for the user to specify an anti-windup strategy, either *clamping* or *back-calculation*. Hence, the custom block is designed to have a fixed structure with a pre-implemented *clamping* anti-windup method for cases

of actuator saturation. However, the anti-windup block has been created separately and added to the personal library as a stand-alone block, which can be reused whenever required. This approach offers more flexibility and allows for easier integration of other anti-windup techniques into the custom PID block in the future. The Figure below displays the two blocks created.

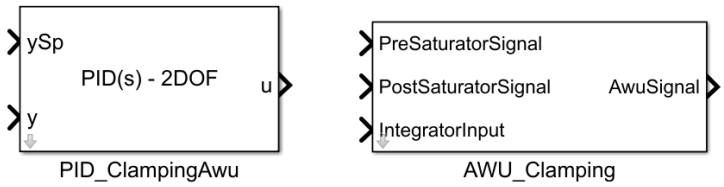

Figure 3.8: PID-2DOF Custom Block and Clamping Anti-Windup Custom Block

Regarding the content of those masks, the Figure 3.9 shows the structure of the custom PID, conversely, the figure 3.10 shows the structure of the conditional integrator anti-windup strategy.

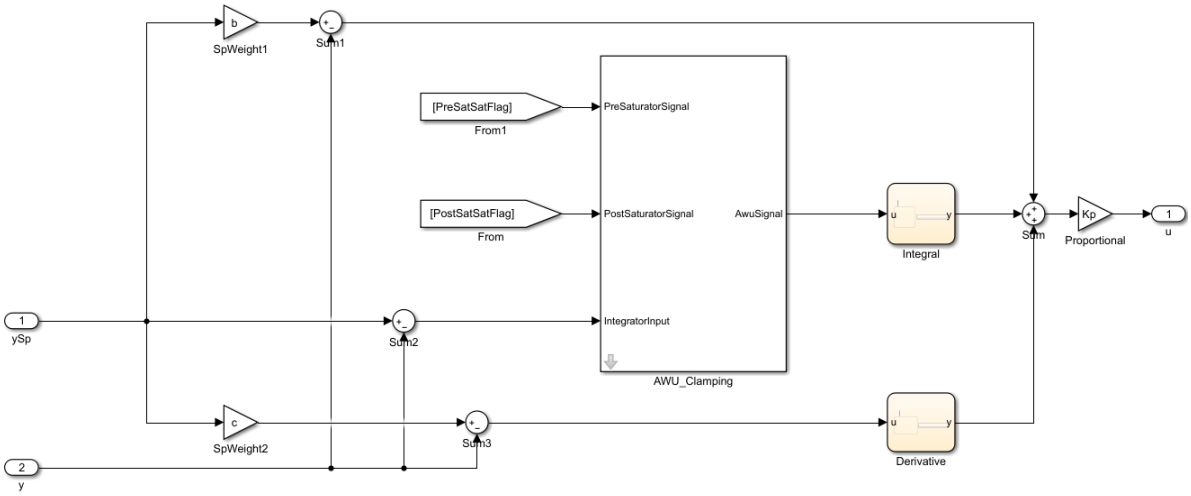

Figure 3.9: Structure of the PID-2DOF Custom Block

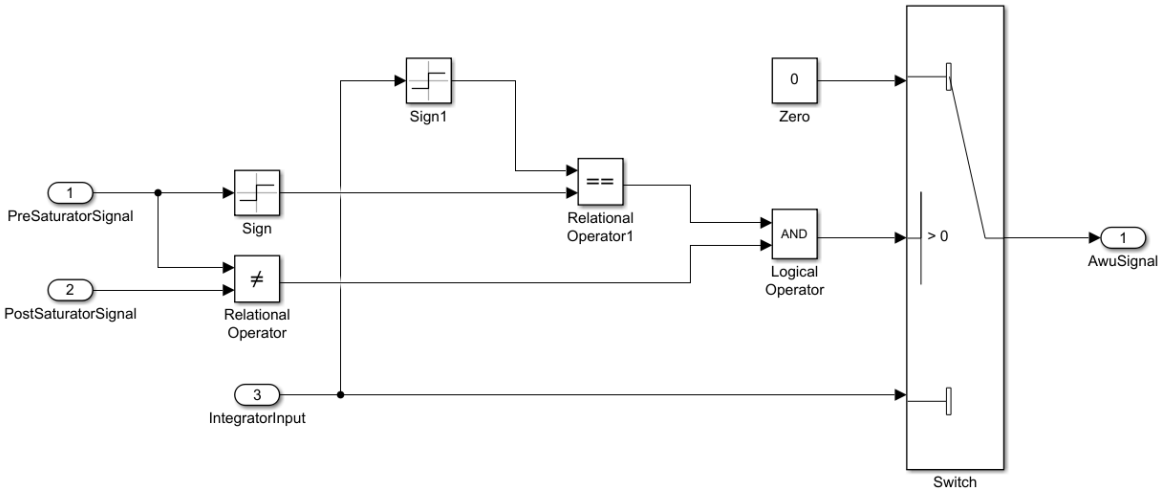

Figure 3.10: Structure of the Clamping Anti-Windup Custom Block

3.3.1 Custom PID Mask Editor

As shown in the Figure below, a mask interface is provided with the custom PID block to facilitate its customization.

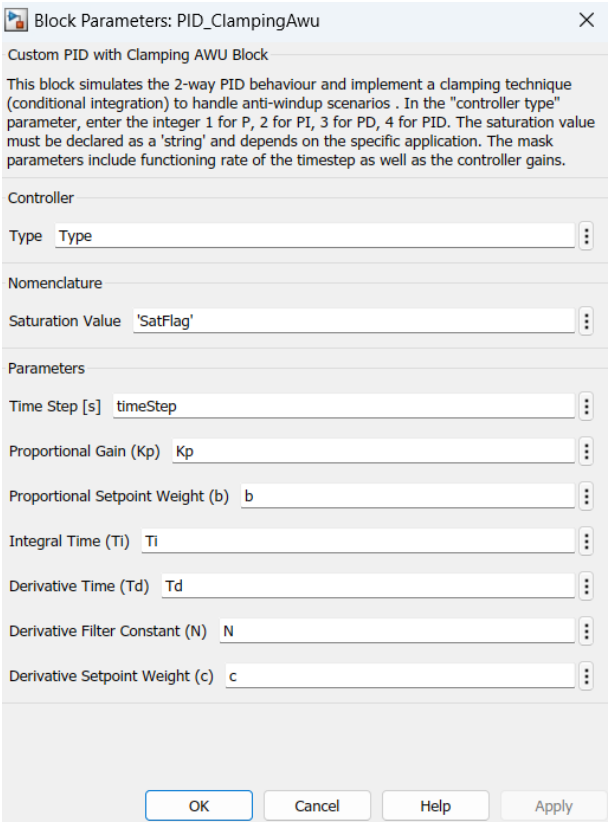

Figure 3.11: Mask of the PID-2DOF Custom Block

The mask interface can modify the number of displayed parameters based on the logic defined in the *Callback* of the *Mask Editor*. The visibility of the PID gains depends on the Controller *Type* specified in the mask, where an integer value of 1 corresponds to P, 2 to PI, 3 to PD, and 4 to PID.

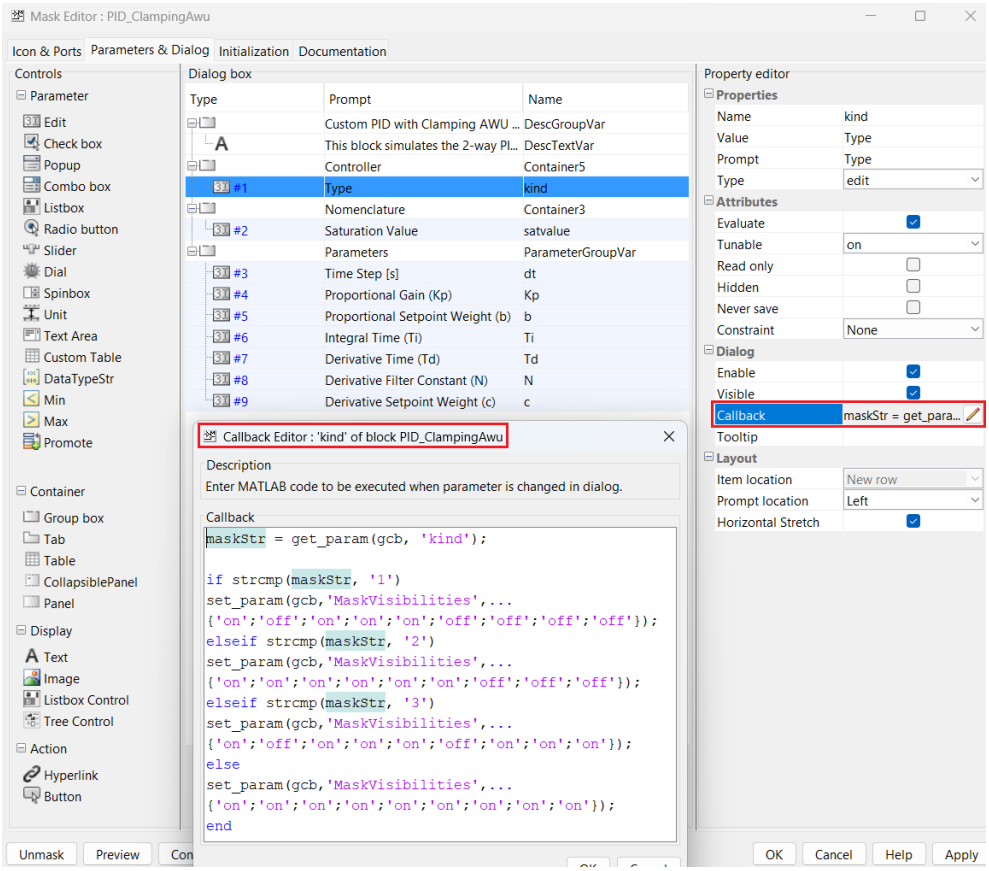

Figure 3.12: Callback property in the Mask Editor of the PID-2DOF Custom Block

Additionally, the *Initialization* panel shown in Figure 3.13 has been thoughtfully crafted to enable accurate naming customization for the *Saturation Value*. This is an essential flag that must be consistently specified based on the particular application. For instance, if the PID controller supplies control input to the AFCS Pitch actuator, then the *Saturation Value* should be assigned as *'Pitch'*.

Furthermore, the *Initialization* interface plays a vital role when using the Simulink® *Control System Tuner*. If the custom PID block is present on the linearization path, it would not be linearized correctly; more information on this topic can be found in the previous thesis [32]. However, by properly specifying the linearization of this block in the *Initialization* interface, it is possible to circumvent this issue and still leverage the capabilities of the Simulink® *Control System Tuner* tool.

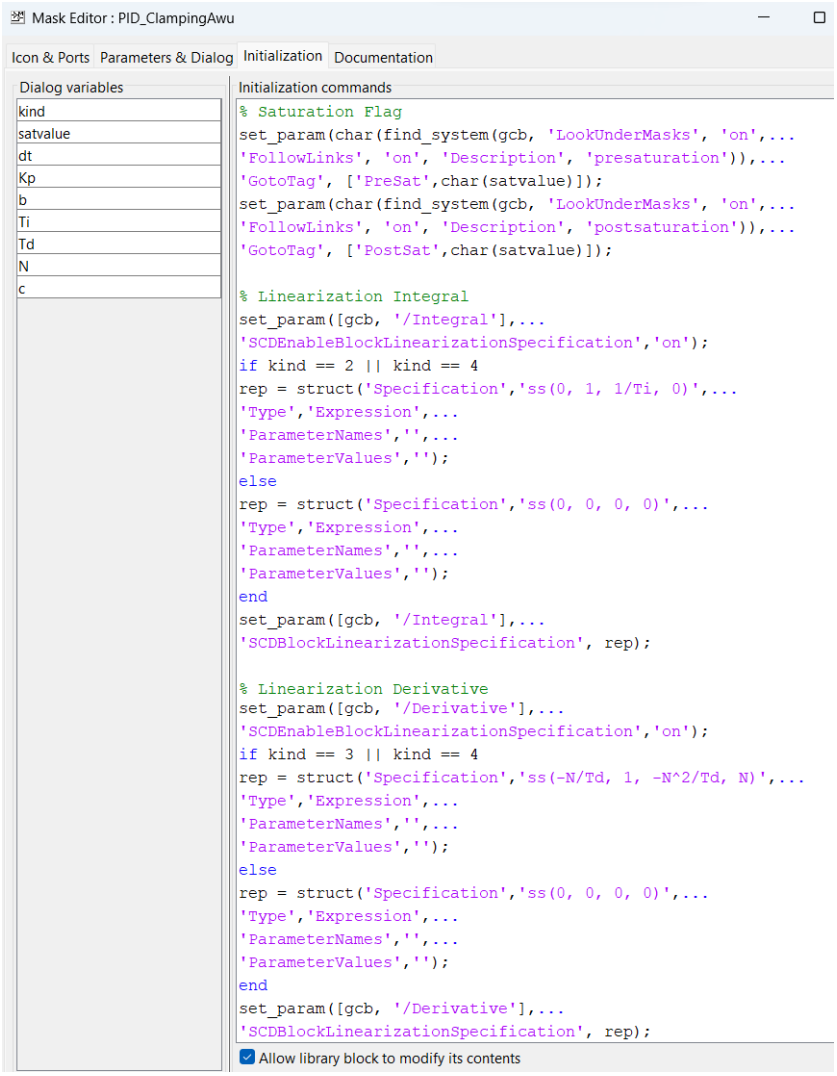

Figure 3.13: Initialization of the Mask Editor of the PID-2DOF Custom Block

3.3.2 Custom PID Integrator and Derivative Transfer Functions

It is important to note that the RK4 algorithm presented in Section 2.4.2 must be implemented within the custom PID block because an integration scheme is required in order to perform the derivative and integration actions of the PID controller. The Stateflow® blocks in Figure 3.9 are responsible for carrying out these tasks. Indeed, while Stateflow® blocks are commonly used for modeling state machines [32], they can also be utilized to create an environment where custom numerical algorithms can be applied according to the specific requirements of the problem.

In order to recreate both the integral and derivative actions of the PID controller, it is necessary to consider the Laplace domain *Standard* form of the 2-DOF PID controller, shown in Equation 3.1.

$$U = K_p \left((bY_{sp} - Y) + \underbrace{\frac{1}{T_i s}}_{\text{Integrator TF}} (Y_{sp} - Y) + \underbrace{\frac{T_d s}{\frac{T_d}{N} s + 1}}_{\text{Derivative TF}} (cY_{sp} - Y) \right) \tag{3.1}$$

Finally, before using the algorithm in Figure 2.15, a transformation from the integral and derivative transfer functions into their respective state-space representations must be performed.

From Transfer Function to State-Space

Although the transformation from transfer function to state-space model is not unique, it is possible to obtain the state variables in the form of phase variables [30]. The state variables are phase variables where each subsequent state is defined to be the derivative of the previous state variable.

Consider a system having input $u(t)$ and output $y(t)$ described by the n-th order linear differential equation:

$$\frac{d^n y(t)}{dt^n} + a_{n-1} \frac{d^{n-1} y(t)}{dt^{n-1}} + \cdots + a_1 \frac{dy(t)}{dt} + a_0 y(t) = b_0 u(t) \tag{3.2}$$

A convenient way to choose the state variables is to use the output $y(t)$ and its $n-1$ derivatives as the state variables. These are called phase variables:

$$\begin{cases} x_1 &= y \\ x_2 &= \frac{dy}{dt} \\ \vdots & \\ x_n &= \frac{d^{n-1} y}{dt^{n-1}} \end{cases} \tag{3.3}$$

Differentiating both sides of the system (3.3) yields:

$$\dot{x}_n = \frac{d^n y}{dt^n} \tag{3.4}$$

Denoting as $\dot{x}_i = \frac{d^i x}{dt^i}$, the system (3.3) can be written also as:

$$\begin{cases} x_1 &= y \\ x_2 &= \frac{dy}{dt} = \frac{dx_1}{dt} = \dot{x}_1 \\ x_3 &= \frac{d^2y}{dt^2} = \frac{dx_2}{dt} = \dot{x}_2 \\ \vdots & \\ x_n &= \frac{d^{n-1}y}{dt^{n-1}} = \frac{dx_{n-1}}{dt} = \dot{x}_{n-1} \end{cases} \tag{3.5}$$

Substituting the definitions 3.3 and 3.4 into 3.2 it results:

$$\dot{x}_n + a_{n-1}x_n + \cdots + a_1x_2 + a_0x_1 = b_0u \tag{3.6}$$

The n-th order differential equation 3.2 can be converted to a system of n first-order differential equations by using the definitions of the derivatives from 3.5 and incorporating the $\dot{x}_n$ obtained from 3.6:

$$\begin{cases} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= x_3 \\ \vdots & \\ \dot{x}_{n-1} &= x_n \\ \dot{x}_n &= -a_0x_1 - a_1x_2 - \cdots - a_{n-1}x_n + b_0u \end{cases} \tag{3.7}$$

In a matrix-vector form, equations 3.7 become:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \\ \vdots \\ \dot{x}_{n-1} \\ \dot{x}_n \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 1 & \cdots & 0 \\ \vdots & & & & & \\ 0 & 0 & 0 & 0 & \cdots & 1 \\ -a_0 & -a_1 & -a_2 & -a_3 & \cdots & -a_{n-1} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ b_0 \end{bmatrix} u \tag{3.8}$$

The phase-variable form of the state equation is represented by equation 3.8. This form is characterized by the pattern of 1's above the main diagonal and 0's in the rest of the state matrix, except for the last row that holds the coefficients of the differential equation written in reverse order.

The output equation in a vector form is:

$$y = \begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} + 0 \cdot u \tag{3.9}$$

From the Integrator TF to its State-Space

Given the transfer function:

$$H(s) = \frac{Y(s)}{U(s)} = \frac{1}{T_i s} = \frac{\frac{1}{T_i}}{s} \tag{3.10}$$

Rearranging equation 3.10, one can obtain the differential equation in the form presented in equation 3.2 by taking the inverse Laplace transform.

$$sY(s) = \frac{1}{T_i} U(s) \Rightarrow \frac{dy}{dt} = \frac{1}{T_i} u \tag{3.11}$$

Given the simplicity of this case it is straightforward to obtain the state-space representation by simply imposing:

$$x = y \Rightarrow \dot{x} = \frac{dy}{dt} = \dot{y} \tag{3.12}$$

The definition above can be substituted into eq. 3.11 to obtain the state equation and output equation:

$$\begin{cases} \dot{x} = \frac{1}{T_i} u \\ y = x \end{cases} \tag{3.13}$$

Hence, the state-space representation is obtained by imposing:

$$A = [0] \qquad B = \begin{bmatrix} \frac{1}{T_i} \end{bmatrix} \qquad C = [1] \qquad D = [0] \tag{3.14}$$

Therefore, based on the aforementioned results, by properly setting the Stateflow®

block and recalling the RK4 algorithm implemented in the *Matlab Function*, one can easily obtain the Simulink® implementation, as shown below.

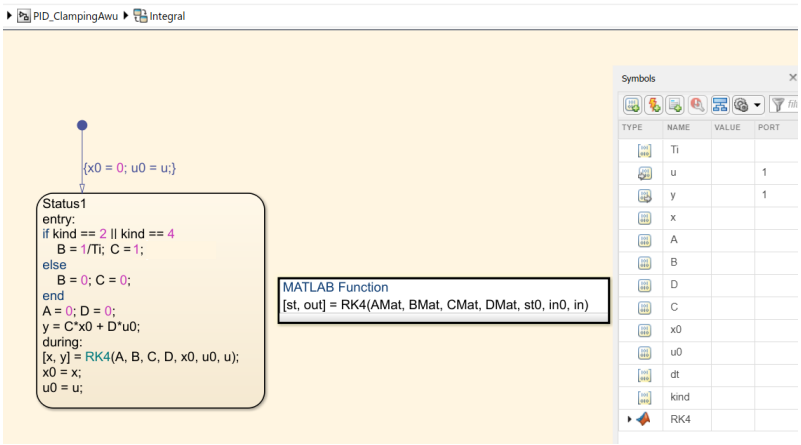

Figure 3.14: PID Custom Integrator obtained through Stateflow®

It is important to notice that during the implementation of this custom integrator, as well as the others custom blocks that require the RK4 algorithm, it is usually necessary to allow visibility and accessibility of certain variables declared within the mask of the subsystem considered. In this case, for instance, the variables needed for the custom integrator that are externally defined in the *PID_ClampingAwu* custom block mask, are the parameters named *kind*, *dt* and *Ti*. Therefore, recalling the Figure 2.8, those variables must be specified in the *Model Explorer* pane, with the *Scope* property set to *Parameter*, as shown below.

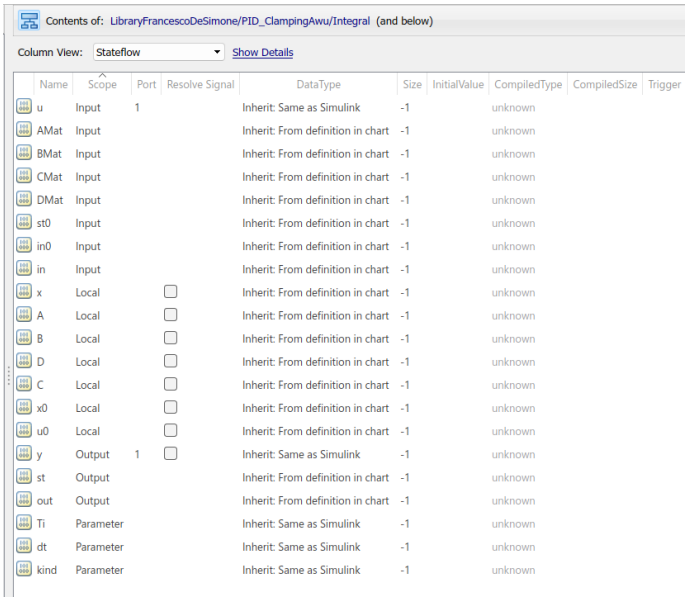

Figure 3.15: PID Custom Integrator data properties settings

From the Derivative TF to its State-Space

Given the transfer function:

$$H(s) = \frac{Y(s)}{U(s)} = \frac{T_d s}{\frac{T_d}{N}s + 1} = \frac{Ns}{s + \frac{N}{T_d}} \tag{3.15}$$

Since in this case the equation 3.15 is still a simple first order TF having a polynomial term at numerator instead of a constant term at numerator, as in the previous case, the transformation to the state-space representation must be handled differently [30]. In particular, from the eq. 3.15, it is necessary to separate the transfer function into two cascaded transfer functions, in which the first is the denominator and the second one is just the numerator. Also by performing the inverse Laplace transform one may obtain:

$$\frac{X(s)}{U(s)} = \frac{1}{s + \frac{N}{T_d}} \quad \Rightarrow \quad \dot{x} + \frac{N}{T_d}x = u \quad \Rightarrow \quad \dot{x} = -\frac{N}{T_d}x + u \tag{3.16}$$

$$\frac{Y(s)}{X(s)} = Ns \quad \Rightarrow \quad y = N\dot{x} \quad \Rightarrow \quad y = -\frac{N^2}{T_d}x + Nu \tag{3.17}$$

Hence, the state-space representation is obtained by imposing:

$$A = \left[-\frac{N}{T_d}\right] \qquad B = [1] \qquad C = \left[-\frac{N^2}{T_d}\right] \qquad D = [N] \tag{3.18}$$

The Simulink® implementation is shown in the Figure below.

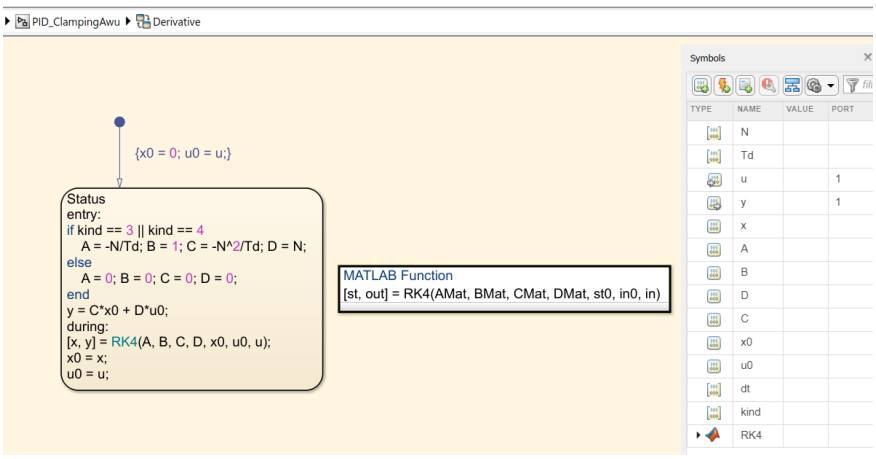

Figure 3.16: PID Custom Derivative obtained through Stateflow®

3.4 SAS Block

The objective of this custom block is to mimic the behavior of a generic Stability Augmentation System. The block's structure is depicted below.

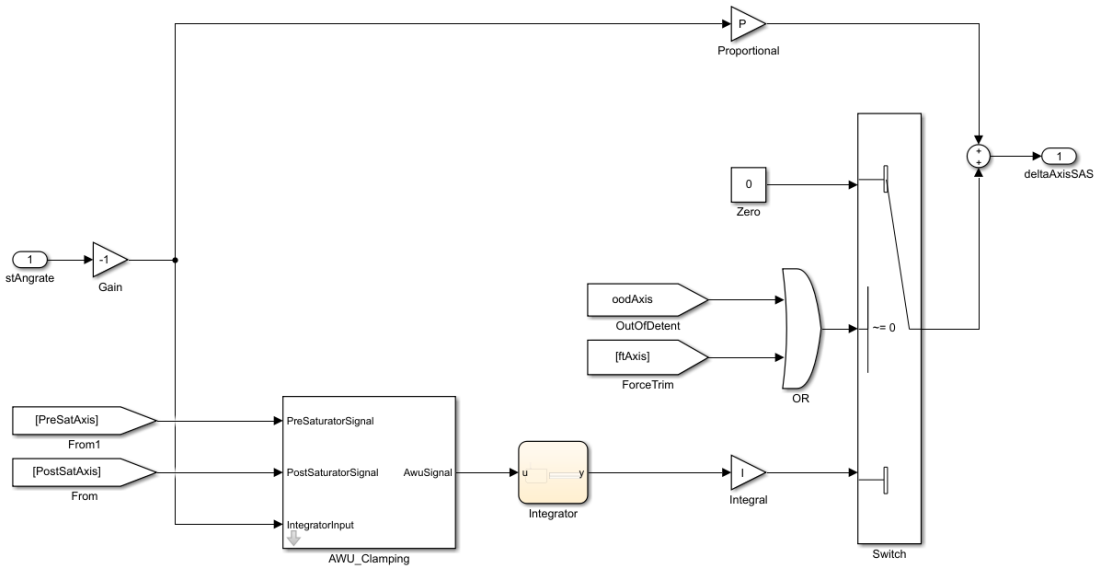

Figure 3.17: Structure of the SAS Custom Block

In this structure, since the P-PI controller is in its *Parallel* form:

$$U = K_p(Y_{sp} - Y) + \frac{K_i}{s}(Y_{sp} - Y) \tag{3.19}$$

the *Integrator* Stateflow® of this block must simulate the transfer function $H(s) = \frac{1}{s}$, hence its state space reduces to:

$$A = [0] \qquad B = [1] \qquad C = [1] \qquad D = [0] \tag{3.20}$$

It is also important to note that a portion of this structure is utilized in the majority of the developed custom blocks: the *Switch* block that uses the pilot's maneuver as a discriminant. In this case, depending on the boolean values of the *force trim* and of the *out of detent* of the specific axis where the SAS is functioning, the controller may switch from a P-controller to a PI-controller or vice versa. Specifically, if the pilot is maneuvering the helicopter, the controller is only proportional to prevent a decrease in the *control augmentation* provided by the pilot.

As for the mask of this block, its GUI is visible in Figure 3.18.

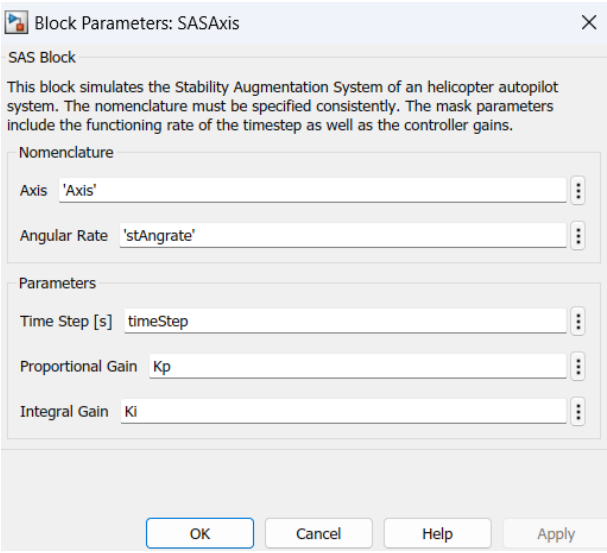

Figure 3.18: Mask of the SAS Custom Block

To provide a quick customization of the block and thus having a single SAS custom block enabling the creation of SAS for pitch, roll and yaw axis, it is necessary to specify the nomenclature of the block through the fields *Axis* and *Angular Rate* (e.g. Axis = 'Roll', Angular Rate = 'stP').

Indeed, as shown in Figure 3.19, the nomenclature specification allows for the name assignment customization within the structure of the SAS custom block.

Initialization commands

```
% Block Name
set_param(gcb, 'Name', ['SAS', char(Axis)]);
% Input Signal
set_param(char(find_system(gcb, 'LookUnderMasks', 'on',...
'FollowLinks', 'on', 'Description', 'Input')),...
'Name', char(Angrate));
% Out of Detent Signal
set_param(char(find_system(gcb, 'LookUnderMasks', 'on',...
'FollowLinks', 'on', 'Description', 'OutOfDetent')),...
'GotoTag', ['ood', char(Axis)]);
% Force Trim Signal
set_param(char(find_system(gcb, 'LookUnderMasks', 'on',...
'FollowLinks', 'on', 'Description', 'ForceTrim')),...
'GotoTag', ['ft', char(Axis)]);
% Output Signal
set_param(char(find_system(gcb, 'LookUnderMasks', 'on',...
'FollowLinks', 'on', 'Description', 'Output')),...
'Name', ['delta', char(Axis), 'SAS']);
ph = get_param(gcb, 'PortHandles');
set_param(ph.Outport(1), 'Name', ['delta', char(Axis), 'SAS']);
% Saturation Flag
set_param(char(find_system(gcb, 'LookUnderMasks', 'on',...
'FollowLinks', 'on', 'Description', 'presaturation')),...
'GotoTag', ['PreSat',char(Axis)]);
set_param(char(find_system(gcb, 'LookUnderMasks', 'on',...
'FollowLinks', 'on', 'Description', 'postsaturation')),...
'GotoTag', ['PostSat',char(Axis)]);
% Linearization
set_param([gcb, '/Integrator'],...
'SCDEnableBlockLinearizationSpecification','on');
rep = struct('Specification','ss(0, 1, 1, 0)',...
'Type','Expression','ParameterNames','','ParameterValues','');
set_param([gcb, '/Integrator'],...
'SCDBlockLinearizationSpecification',rep);
```

Allow library block to modify its contents

Figure 3.19: Initialization of the Mask Editor of the SAS Custom Block

3.5 Mode Setpoint Block

This custom block, together with the *Mode δCommand/δSetpoint* block, are the most fundamental and generic blocks created for the AFCS. By properly customizing their mask and combining these two blocks, all upper/lower modes within the AFCS of the current work can be created. To achieve such modularity and generality of these blocks, careful consideration of the nomenclature at each hierarchy level of the AFCS has been necessary, as well as taking into account every mode characteristic and making the distinguishing feature of each mode as generic as possible.

Since the pilot can maintain a specific datum characteristic of the particular mode in most of the lower/upper modes of the helicopter autopilot, the *Mode Setpoint* block is designed to manage the setpoint value for each of those modes, based on the pilot's actions in the cockpit. The structure of this block is illustrated in Figure 3.20 and its function is explained as follows:

1. The *Mode Setpoint* block takes the characteristic variable of the specific mode as input.
2. The *Switch* block determines the setpoint assignment based on its discriminant Boolean value in input:
   - When the discriminant is equal to 1, the setpoint coincides with the current measured variable input.
   - When the discriminant is equal to 0, which only happens when the pilot is not using the force trim (hence, its Boolean is 0) and the specified mode is correctly engaged (hence, its Boolean is 1), the setpoint may differ from the current value of the variable, depending on the pilot's actions on the beeptrim controlling that specific mode.
3. The *Mode Setpoint* block outputs the correct setpoint of the specific mode on a specific axis.

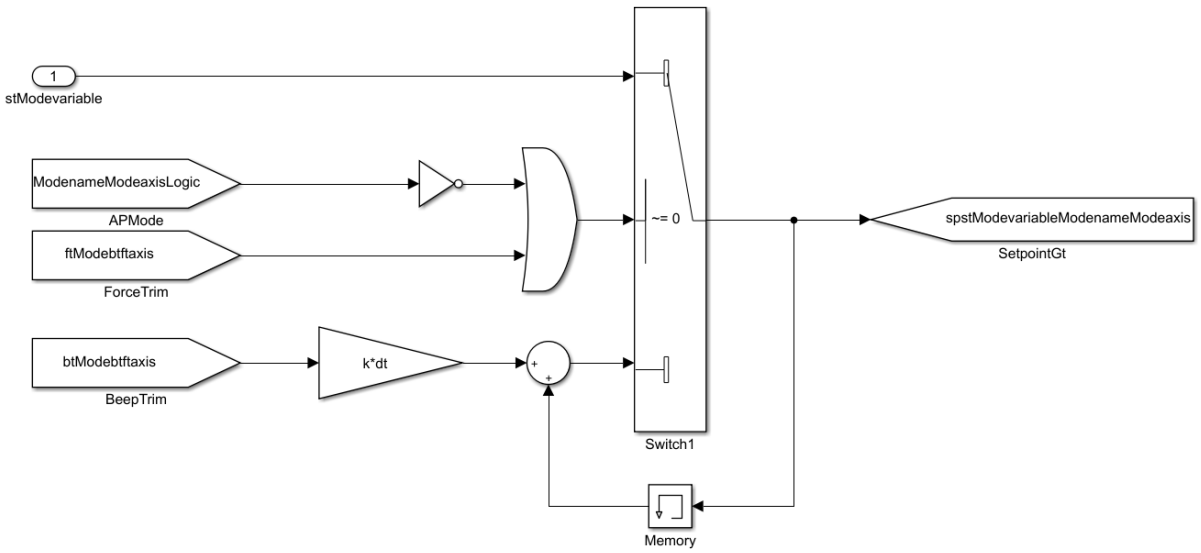

Figure 3.20: Structure of the Mode Setpoint Custom Block

Consistent specification of nomenclature is essential, depending on the properties of the mode. The mask parameters also include the functioning rate of the time step and the specific mode beep trim's functioning rate, as illustrated below.

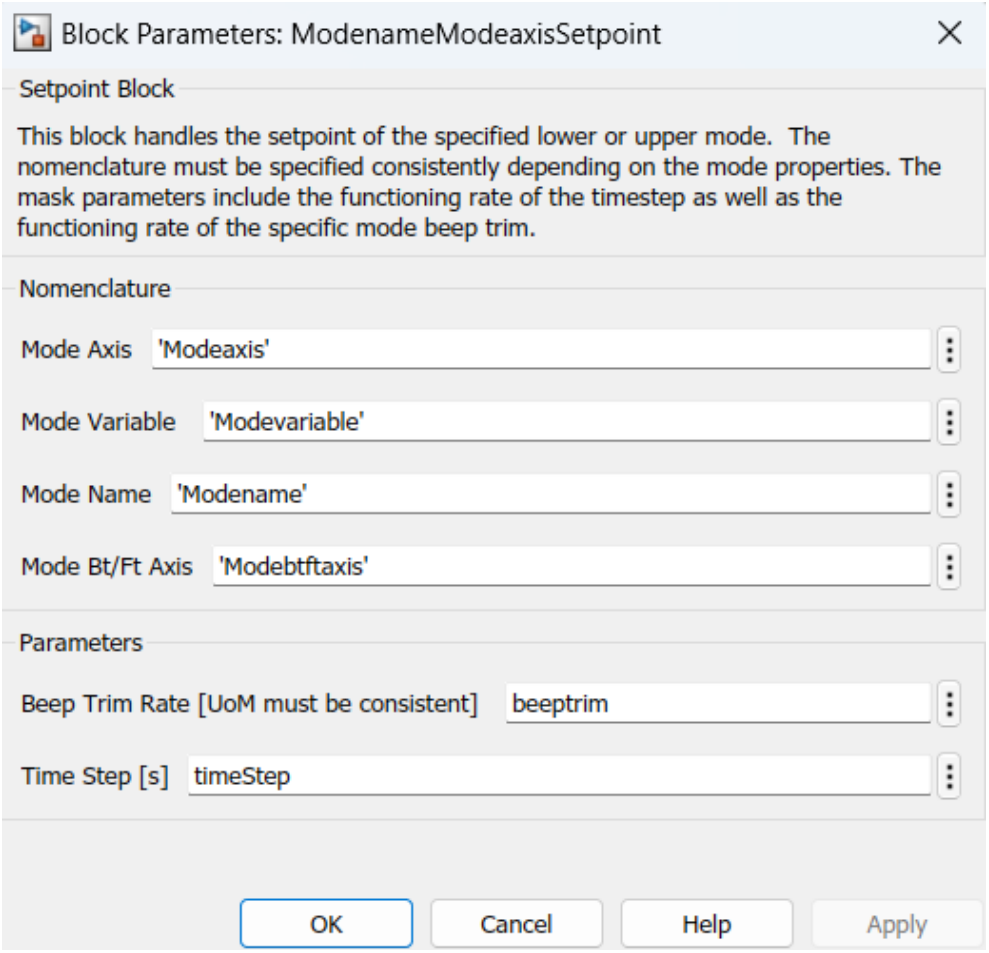

Figure 3.21: Mask of the Mode Setpoint Custom Block

The mask's *Initialization* interface provides the usual method for ensuring the correctness of nomenclature assignment.

However, it should be noted that some modes may be engaged on different axes based on the control logics outlined in Section 3.2. As a result, it is necessary to maintain the same setpoint value regardless of the axis on which the mode is engaged. For instance, suppose the ALT Mode is engaged on the Pitch Axis and then the control logics switch it to the Collective Axis after the IAS Mode engagement: in this case any discrepancy between the setpoint values in the *ALTPitchSetpoint* block and the *ALTCollectiveSetpoint* block could result in improper setpoint management and potentially incorrect autopilot behavior. Consequently, for these particular modes, the *Initialization* interface shown in Figure 3.22 includes a specific sequence of commands that enables the structure in Figure 3.20 to modify its assembly and adapt to the behavior of those modes. For instance, in the case of the ALT Mode, the *NOT* block would receive input from two *From* blocks, namely the *ALTPitchLogic* and the *ALTCollectiveLogic*, instead of just one.

Although there may be other methods to handle this problem, this particular approach has been chosen to ensure that the block's modularity and genericity are maintained at the highest level.

Initialization commands

```
% Block Name
get_param(gcb, 'Name')
set_param(gcb, 'Name', [char(ModeName),char(ModeAxis),'Setpoint']);
% Input Signal
set_param(char(find_system(gcb, 'LookUnderMasks', 'on',...
'FollowLinks', 'on', 'Description', 'Input')),...
'Name', ['st', char(ModeVariable)]);
% Beep Trim Signal
set_param(char(find_system(gcb, 'LookUnderMasks', 'on',...
'FollowLinks', 'on', 'Description', 'BT')),...
'GotoTag', ['bt', char(ModeBtFtAxis)]);
% Force Trim Signal
set_param(char(find_system(gcb, 'LookUnderMasks', 'on',...
'FollowLinks', 'on', 'Description', 'FT')),...
'GotoTag', ['ft', char(ModeBtFtAxis)]);
% Mode Engagement
set_param(char(find_system(gcb, 'LookUnderMasks', 'on',...
'FollowLinks', 'on', 'Description', 'ME')),...
'GotoTag', [char(ModeName),char(ModeAxis),'Logic']);
% Setpoint Signal
set_param(char(find_system(gcb, 'LookUnderMasks', 'on',...
'FollowLinks', 'on', 'Description', 'SP')),...
'GotoTag', ['spst', char(ModeVariable), char(ModeName),char(ModeAxis)]);

% 2-Axis Modes adjustments: They have same setpoint but they manage different axis
try
if (strcmp('ATT',char(ModeName))==1 && strcmp('Yaw',char(ModeAxis))==1)        || ...
   (strcmp('HDG',char(ModeName))==1 && strcmp('Roll',char(ModeAxis))==1)       || ...
   (strcmp('ALT',char(ModeName))==1 && strcmp('Pitch',char(ModeAxis))==1)      || ...
   (strcmp('ALT',char(ModeName))==1 && strcmp('Collective',char(ModeAxis))==1)
          add_block('simulink/Signal Routing/From',[char(gcb),'/APMode2']);
          if (strcmp('ATT',char(ModeName))==1 && strcmp('Yaw',char(ModeAxis))==1)
          set_param([char(gcb),'/APMode2'],'GotoTag','HDGRollLogic');
          elseif (strcmp('HDG',char(ModeName))==1 && strcmp('Roll',char(ModeAxis))==1)
          set_param([char(gcb),'/APMode2'],'GotoTag','ATTYawLogic');
          elseif (strcmp('ALT',char(ModeName))==1 && strcmp('Pitch',char(ModeAxis))==1)
          set_param([char(gcb),'/APMode2'],'GotoTag','ALTCollectiveLogic');
          elseif (strcmp('ALT',char(ModeName))==1 && strcmp('Collective',char(ModeAxis))==1)
          set_param([char(gcb),'/APMode2'],'GotoTag','ALTPitchLogic');
          end
          add_block('simulink/Logic and Bit Operations/Logical Operator',[char(gcb),'/OR1']);
          set_param([char(gcb),'/OR1'],'Operator','OR');
          delete_line(gcb,'APMode/1','NOT1/1');
          add_line(gcb,'APMode/1','OR1/1','autorouting','smart');
          add_line(gcb,'APMode2/1','OR1/2','autorouting','smart');
          add_line(gcb,'OR1/1','NOT1/1','autorouting','smart');
else
          delete_line(gcb,'APMode/1','OR1/1');
          delete_line(gcb,'APMode2/1','OR1/2');
          delete_line(gcb,'OR1/1','NOT1/1');
          delete_block([char(gcb),'/OR1']);
          delete_block([char(gcb),'/APMode2']);
          add_line(gcb,'APMode/1','NOT1/1','autorouting','smart');
end
end
```

Allow library block to modify its contents

Figure 3.22: Initialization of the Mask Editor of the Mode Setpoint Custom Block

3.6 Mode δCommand/δSetpoint Block

As mentioned in the previous paragraph, this custom block is a fundamental component in developing each mode of the AFCS. To create this block, the same building strategy employed fot the *Mode Setpoint* block is also applied here. Additionally, a proper nomenclature assignment is utilized to ensure a high level of modularity and generality. This block functionalities vary based on the user's customization in the mask GUI, which may be depicted in Figure 3.23.

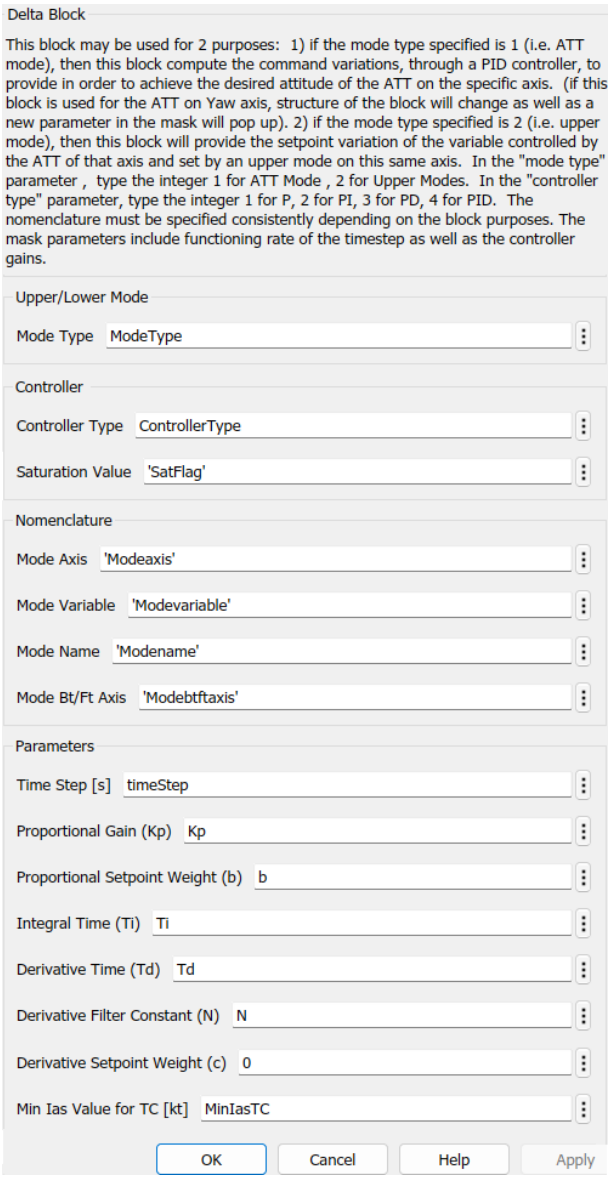

Figure 3.23: Mask of the Mode δCommand/δSetpoint Custom Block

This block has two distinct purposes, depending on the specification entered in the

Mode Type field of the mask. Indeed, during the development of the AFCS it was evident that the two different subsystems needs to rely on the same structural block configuration illustrated in Figure 3.24, but they require different nomenclature settings. To resolve this issue, a decision is made to consolidate them into a single, more versatile block.

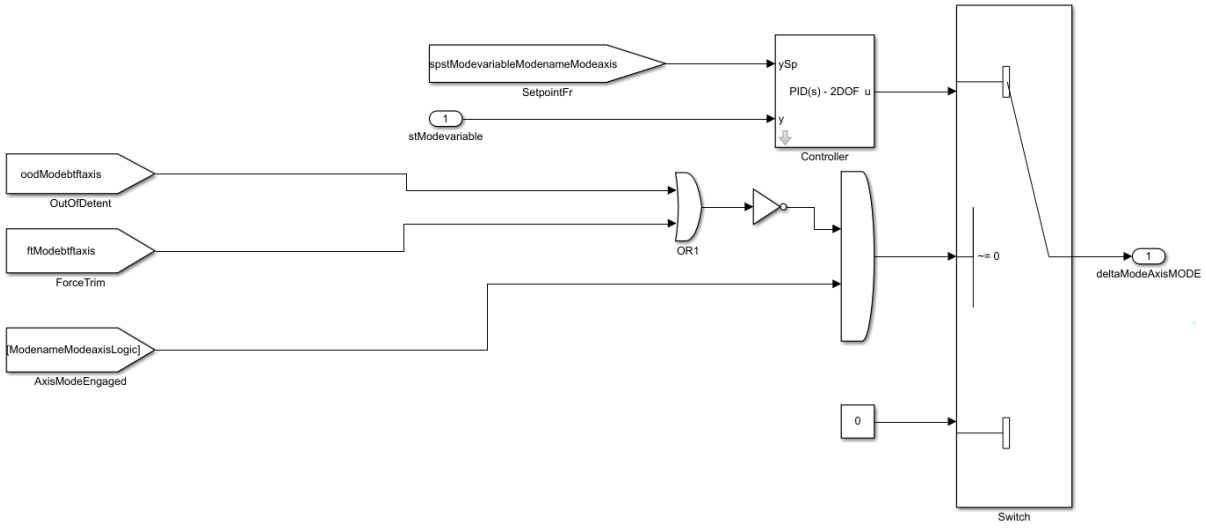

Figure 3.24: Structure of the Mode δCommand/δSetpoint Custom Block

Going into detail, this block can perform the following two tasks:

1. if the *Mode Type* specified is *1*, this block functions as a *Mode δCommand* block, which is essential for the ATT mode to operate. Its task is to compute the command variations through a PID controller having as a setpoint the desired attitude imposed on the ATT of a specific axis. Furthermore, if this block is used for the ATT mode on the yaw axis, its structure changes, and a new parameter in the mask appears. Furthermore, among the upper modes, only the collective modes employ the *Mode δCommand* block to function, as they don't rely on ATT during their functioning.

2. if the *Mode Type* specified is *2*, this block is instead used as a *Mode δSetpoint* block. This second kind of function is fundamental for every upper modes. Indeed, since an upper mode engaged on a specific axis requires the ATT mode on the same axis to function, the purpose of this block is to determine the setpoint change of the ATT given a specific activated upper mode.

   Suppose for instance the IAS mode engaged in the AFCS; in this case, depending on the error between the reference and the current *Indicated Airspeed*, a PID controller computes the variation of *Theta* required to achieve null error. The Mode δSetpoint block is responsible for this task when customized for the IAS mode. However, this

is just the setpoint variation, since in order to obtain the correct setpoint value for the ATT of the pitch axis, the value obtained must be added to the current value of *Theta*.

Furthermore, as already seen for the *Mode Setpoint* custom block, also in this case a *Switch* block determine the value of the δCommand/δSetpoint based on its discriminant Boolean input. Specifically:

- when the discriminant is equal to 0, the *δCommand/δSetpoint* is 0 as well.
- when the discriminant is equal to 1, which only occurs when the pilot is not using the force trim (hence, its Boolean is 0), the command is not out of detent (hence, its Boolean is 0), and the specified mode is correctly engaged (hence, its Boolean is 1), then δCommand/δSetpoint corresponds to the output of the PID controller.

It is worth mentioning that the Figure 3.24 shows that the *PID 2DOF* custom block is nested inside the *Mode δCommand/δSetpoint* custom block. However, it is possible to define the parameters of the *PID 2DOF* custom block without accessing the mask of this custom block. By simply specifying the *Controller Type* field of the mask in Figure 3.23, along with the time step and controller gains, one can directly pass those parameters to the *PID 2DOF* custom block. It is important to note that this result is always achievable when dealing with nested blocks, provided that the parameters are defined through the *Promote* control present in the *Parameters & Dialog* interface in the *Mask Editor* of the custom block.

Moreover, the mask GUI of the *Mode δCommand/δSetpoint* custom block must be filled with the correct nomenclature, depending on the block's purpose. The mask's *Initialization* interface allows as usual for this feature, as shown in Figure 3.25.

Initialization commands

```
% Block Name
if     ModeType == 1 % Lower Modes
          get_param(gcb, 'Name')
          set_param(gcb, 'Name', ['DeltaCommand',char(ModeAxis)]);
elseif ModeType == 2 % Upper Modes
          if      strcmp('Pitch',char(ModeAxis))==1
                     get_param(gcb, 'Name')
                     set_param(gcb, 'Name', ['Deltaspst','Theta',char(ModeName)]);
          elseif  strcmp('Roll',char(ModeAxis))==1
                     get_param(gcb, 'Name')
                     set_param(gcb, 'Name', ['Deltaspst','Phi',char(ModeName)]);
          elseif  strcmp('Yaw',char(ModeAxis))==1
                     get_param(gcb, 'Name')
                     set_param(gcb, 'Name', ['Deltaspst','Psi',char(ModeName)]);
          elseif  strcmp('Collective',char(ModeAxis))==1
                     get_param(gcb, 'Name')
                     set_param(gcb, 'Name', ['DeltaCommand',char(ModeAxis)]);
          end
end
% Force Sense Switch + Force Trim Signal
if     ModeType == 1 % Lower Modes
           %Force Sense Switch
           set_param(char(find_system(gcb, 'LookUnderMasks', 'on','FollowLinks', 'on', 'Description', 'FSS')),'GotoTag', ['ood', char(ModeAxis)]);
           %Force Trim Signal
           set_param(char(find_system(gcb, 'LookUnderMasks', 'on','FollowLinks', 'on', 'Description', 'FT')),'GotoTag', ['ft', char(ModeAxis)]);
elseif ModeType == 2 % Upper Modes
           %Force Sense Switch
           set_param(char(find_system(gcb, 'LookUnderMasks', 'on','FollowLinks', 'on', 'Description', 'FSS')),'GotoTag', ['ood', char(ModeBtFtAxis)]);
           %Force Trim Signal
           set_param(char(find_system(gcb, 'LookUnderMasks', 'on','FollowLinks', 'on', 'Description', 'FT')),'GotoTag', ['ft', char(ModeBtFtAxis)]);
end
% Axis Mode Engagement
if     ModeType == 1 || (ModeType == 2 &&  strcmp('Collective',char(ModeAxis))==1) % Lower Modes + Upper Modes on Collective Axis
          set_param(char(find_system(gcb, 'LookUnderMasks', 'on','FollowLinks', 'on', 'Description', 'AME')),'GotoTag', [char(ModeAxis),'Mode']);
elseif ModeType == 2 % Upper Modes
          set_param(char(find_system(gcb, 'LookUnderMasks', 'on','FollowLinks', 'on', 'Description', 'AME')),'GotoTag', [char(ModeName),char(ModeAxis),'Logic']);
end
% Setpoint Signal
if     ModeType == 1 || (ModeType == 2 &&  strcmp('Collective',char(ModeAxis))==1) % Lower Modes + Upper Modes on Collective Axis
           set_param(char(find_system(gcb, 'LookUnderMasks', 'on','FollowLinks', 'on', 'Description', 'SetpointFrom')),'GotoTag', ['spst', char(ModeVariable)]);
           ph = get_param(char(find_system(gcb, 'LookUnderMasks', 'on','FollowLinks', 'on', 'Description', 'SetpointFrom')), 'PortHandles');
           set_param(ph.Outport(1), 'Name', ['spst', char(ModeVariable)]);
elseif ModeType == 2 % Upper Modes
           set_param(char(find_system(gcb, 'LookUnderMasks', 'on','FollowLinks', 'on', 'Description', 'SetpointFrom')),...
           'GotoTag', ['spst', char(ModeVariable),char(ModeName),char(ModeAxis)]);
           ph = get_param(char(find_system(gcb, 'LookUnderMasks', 'on','FollowLinks', 'on', 'Description', 'SetpointFrom')), 'PortHandles');
           set_param(ph.Outport(1), 'Name', ['spst', char(ModeVariable),char(ModeName),char(ModeAxis)]);
end
% Input Signal
set_param(char(find_system(gcb, 'LookUnderMasks', 'on','FollowLinks', 'on', 'Description', 'Input')),'Name',['st', char(ModeVariable)]);
% Output Signal
if     ModeType == 1 % Lower Modes
           set_param(char(find_system(gcb, 'LookUnderMasks', 'on','FollowLinks', 'on', 'Description', 'Output')),.'Name', ['delta', char(ModeAxis),'MODE']);
           ph = get_param(gcb, 'PortHandles'); set_param(ph.Outport(1), 'Name', ['delta', char(ModeAxis)]);
elseif ModeType == 2 % Upper Modes
           if strcmp('Pitch',char(ModeAxis))==1
               set_param(char(find_system(gcb, 'LookUnderMasks', 'on','FollowLinks', 'on', 'Description', 'Output')),'Name', ['deltaspst', 'Theta',char(ModeName)]);
               ph = get_param(gcb, 'PortHandles'); set_param(ph.Outport(1), 'Name', ['deltaspst', 'Theta',char(ModeName)]);
           elseif strcmp('Roll',char(ModeAxis))==1
               set_param(char(find_system(gcb, 'LookUnderMasks', 'on','FollowLinks', 'on', 'Description', 'Output')),'Name', ['deltaspst', 'Phi',char(ModeName)]);
               ph = get_param(gcb, 'PortHandles'); set_param(ph.Outport(1), 'Name', ['deltaspst', 'Phi',char(ModeName)]);
           elseif strcmp('Yaw',char(ModeAxis))==1
               set_param(char(find_system(gcb, 'LookUnderMasks', 'on','FollowLinks', 'on', 'Description', 'Output')),'Name', ['deltaspst', 'Psi',char(ModeName)]);
               ph = get_param(gcb, 'PortHandles'); set_param(ph.Outport(1), 'Name', ['deltaspst', 'Psi',char(ModeName)]);
           elseif strcmp('Collective',char(ModeAxis))==1
               set_param(char(find_system(gcb, 'LookUnderMasks', 'on','FollowLinks', 'on', 'Description', 'Output')),'Name', ['delta', char(ModeAxis),'MODE']);
               ph = get_param(gcb, 'PortHandles'); set_param(ph.Outport(1), 'Name', ['delta', char(ModeAxis)]);
           end
end
```

Figure 3.25: Initialization of the Nomenclature of the Mode δCommand/δSetpoint Custom Block

On the other hand, as depicted in the Figure 3.26, the mask's *Initialization* interface is necessary to manage also the special case in which this custom block is employed for the ATT on the yaw axis. Indeed, this mode requires an additional check on the helicopter's current *Indicated Airspeed*, which must not exceed a fixed threshold. This check is necessary because the TC Mode, which works for the yaw axis, may be active above this threshold, and in such cases, the ATT on the yaw axis must be turned off since the two AP systems cannot be engaged simultaneously in the AFCS.

```
% (ATT)Yaw adjustments
try % serve per getsire gli errori e andare avanti qualora ci siano

if strcmp('Yaw',char(ModeAxis))==1 && ModeType == 1
        add_block('simulink/Sources/In1',[char(gcb),'/stIas']);
        add_block('simulink/Logic and Bit Operations/Compare To Constant',[char(gcb),'/Iascheck']);
        set_param([char(gcb),'/Iascheck'],'relop','<=');
        set_param([char(gcb),'/Iascheck'],'const','MinIasTC*kt_to_ms');
        add_line(gcb,'stIas/1','Iascheck/1','autorouting','smart');
        delete_line(gcb,'AxisModeEngaged/1','AND/2');
        add_block('simulink/Logic and Bit Operations/Logical Operator',[char(gcb),'/AND2']);
        set_param([char(gcb),'/AND2'],'Operator','AND');
        add_line(gcb,'AxisModeEngaged/1','AND2/1','autorouting','smart');
        add_line(gcb,'Iascheck/1','AND2/2','autorouting','smart');
        add_line(gcb,'AND2/1','AND/2','autorouting','smart');

else
        delete_line(gcb,'stIas/1','Iascheck/1');
        delete_line(gcb,'AxisModeEngaged/1','AND2/1');
        delete_line(gcb,'Iascheck/1','AND2/2');
        delete_line(gcb,'AND2/1','AND/2');
        delete_block([char(gcb),'/stIas']);
        delete_block([char(gcb),'/Iascheck']);
        delete_block([char(gcb),'/AND2']);
        add_line(gcb,'AxisModeEngaged/1','AND/2','autorouting','smart');
end
end
```

Allow library block to modify its contents

Figure 3.26: Initialization of the adjustments occurring when the Mode δCommand/δSetpoint Custom Block is used for the ATT on the yaw axis

Finally, since some fields of the mask GUI of this custom block may be useless to be shown in some configurations, namely the various controller parameters as well as the threshold that characterise the condition imposed only for the ATT on the yaw axis, the visibility of the parameters of the mask GUI of this custom block must be properly set as shown below.

Callback Editor : 'ModeAxis' of block DeltaCommandCollective

Description

Enter MATLAB code to be executed when parameter is changed in dialog.

Callback

```
maskStr0 = get_param(gcb, 'ModeType'); % (1 = Lower Mode) (2 = Upper Mode)
maskStr1 = get_param(gcb, 'ContrType');
maskStr2 = get_param(gcb, 'ModeAxis');

if strcmp(maskStr0, '1') % in case of Lower Mode
    if strcmp(maskStr1, '1')
        if strcmp(maskStr2, '''Yaw''')
           set_param(gcb,'MaskVisibilities',...
      {'on';'on';'on';'on';'on';'off';'off';'on';'on';'on';'off';'off';'off';'off';'on'});
        else
           set_param(gcb,'MaskVisibilities',...
      {'on';'on';'on';'on';'on';'off';'off';'on';'on';'on';'off';'off';'off';'off';'off'})
        end
    elseif strcmp(maskStr1, '2')
        if strcmp(maskStr2, '''Yaw''')
           set_param(gcb,'MaskVisibilities',...
      {'on';'on';'on';'on';'on';'off';'off';'on';'on';'on';'on';'off';'off';'off';'on'});
        else
           set_param(gcb,'MaskVisibilities',...
      {'on';'on';'on';'on';'on';'off';'off';'on';'on';'on';'on';'off';'off';'off';'off'});
        end
    elseif strcmp(maskStr1, '3')
        if strcmp(maskStr2, '''Yaw''')
           set_param(gcb,'MaskVisibilities',...
      {'on';'on';'on';'on';'on';'off';'off';'on';'on';'on';'off';'on';'on';'on';'on'});
        else
           set_param(gcb,'MaskVisibilities',...
      {'on';'on';'on';'on';'on';'off';'off';'on';'on';'on';'off';'on';'on';'on';'off'});
        end
    else
        if strcmp(maskStr2, '''Yaw''')
           set_param(gcb,'MaskVisibilities',...
      {'on';'on';'on';'on';'on';'off';'off';'on';'on';'on';'on';'on';'on';'on';'on'});
        else
           set_param(gcb,'MaskVisibilities',...
      {'on';'on';'on';'on';'on';'off';'off';'on';'on';'on';'on';'on';'on';'on';'off'});
        end
    end
else % in case of Upper Mode
    if     strcmp(maskStr1, '1')
           set_param(gcb,'MaskVisibilities',...
      {'on';'on';'on';'on';'on';'on';'on';'on';'on';'on';'off';'off';'off';'off';'off'});
    elseif strcmp(maskStr1, '2')
           set_param(gcb,'MaskVisibilities',...
      {'on';'on';'on';'on';'on';'on';'on';'on';'on';'on';'on';'off';'off';'off';'off'});
    elseif strcmp(maskStr1, '3')
           set_param(gcb,'MaskVisibilities',...
      {'on';'on';'on';'on';'on';'on';'on';'on';'on';'on';'off';'on';'on';'on';'off'});
    else
          set_param(gcb,'MaskVisibilities',...
     {'on';'on';'on';'on';'on';'on';'on';'on';'on';'on';'on';'on';'on';'on';'off'});
    end
end
```

Figure 3.27: Visibility of the Callback property in the Mask Editor of the Mode δCommand/δSetpoint Custom Block

3.7 TC Block

All of the AFCS lower and upper modes can be developed by utilizing the *SAS* custom block, the *Mode δCommand/δSetpoint* custom block, and the *Mode Setpoint* custom block. Therefore, the only missing system in the current work's AFCS architecture is a custom block able to model the Turn Coordinator behaviour.

The *TC* custom block shares some structural similarity with the *Mode δCommand/δSetpoint* custom block. However, due to the Turn Coordinator's intrinsic functionalities, additional logical conditions must be added to the *Switch* block, compared to the *Mode δCommand/δSetpoint* custom block. As a result, the *TC* custom block is only used once in the creation of the AFCS model and is the least generic block created. Nevertheless, it remains a highly customisable block, like all the other blocks developed. The structure and mask of this custom block are shown in Figures 3.28 and 3.29, respectively.

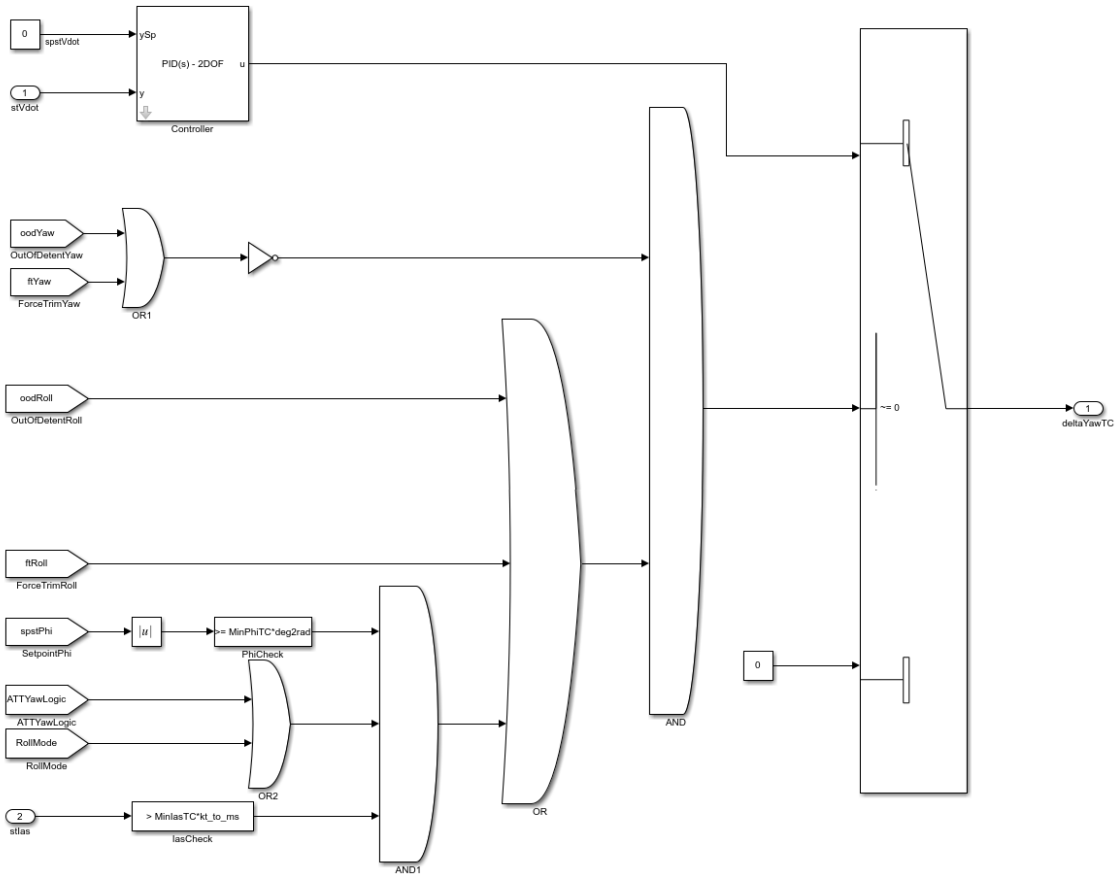

Figure 3.28: Structure of the Turn Coordinator Custom Block

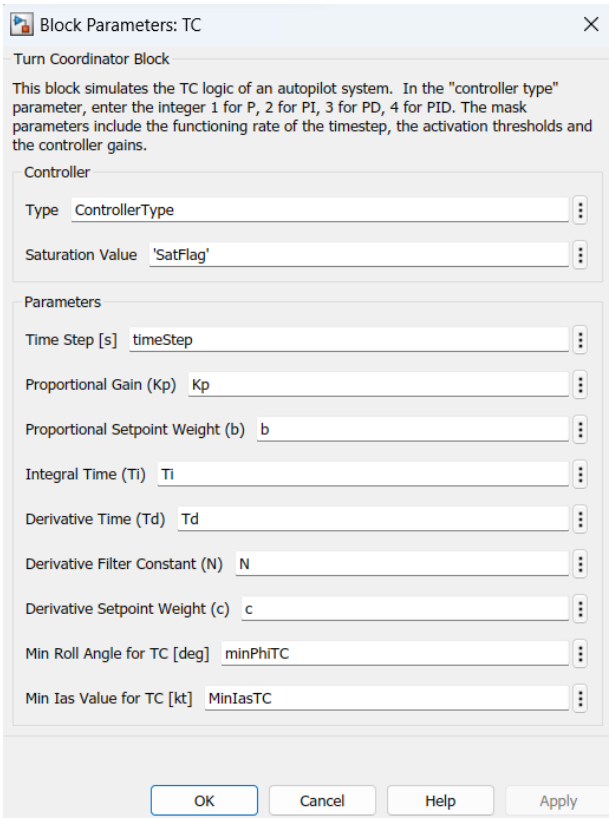

Figure 3.29: Mask of the Turn Coordinator Custom Block

3.8 Saturation Block

The *Saturation Block* is a custom block created to limit the input signal to the upper and lower saturation values set by the user. Additionally, it provides the necessary inputs for the *Anti-Windup* custom block by means of *GoTo* blocks. The images below display the structure of this block, along with the mask GUI and the *Initialization* interface required for naming conventions.

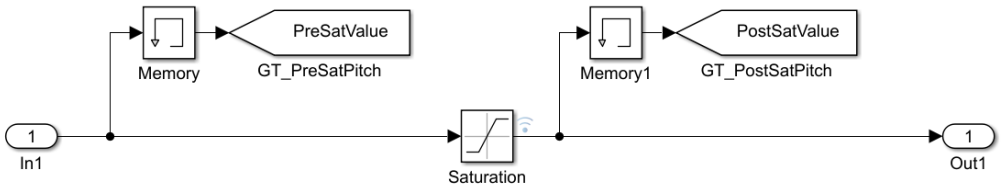

Figure 3.30: Structure of the Saturation Custom Block

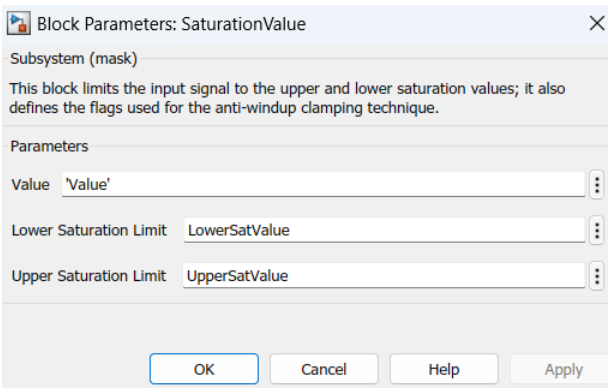

Figure 3.31: Mask of the Saturation Custom Block

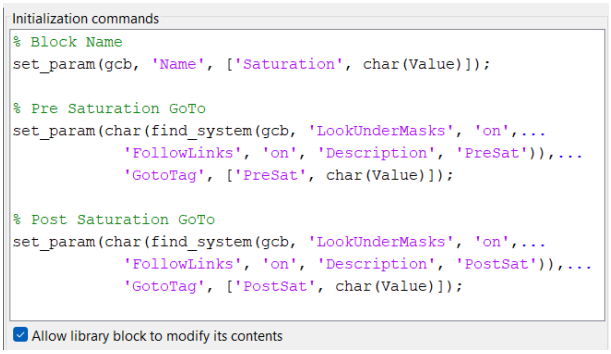

Figure 3.32: Initialization of the Saturation Custom Block

4 | AFCS Controllers Tuning

This chapter aims to present a methodology for tuning the controllers of an automatic flight control system in order to address the control requirements of a highly nonlinear helicopter dynamics model. The AFCS being referred to is the one that can be assembled using the custom blocks outlined in the preceding chapter.

In this regard, it is worth mentioning that different levels of certification exist, depending on the type of helicopter and the level of accuracy required by the flight simulator. These levels are determined by regulatory authorities such as the FAA and the EASA. The level of accuracy is determined by several factors, including the replication of the cockpit environment and flight performance. For this reason, being the aim of the present work the development of a generic AFCS, then also the tuning process of the controllers of the AFCS must allow custom specification of requirements and performance goals which can vary depending on the specific rotorcraft.

Another aspect to consider is that the tuning phase for those kind of applications is typically not carried out in real-time during the simulation, as flight simulators have a fast refresh rate of typically 60 Hz to 120 Hz to ensure smooth movement of the helicopter during the flight simulation. Real-time tuning would be computationally expensive. Therefore, in the current work, the tuning of the controllers is accomplished in an external script.

4.1 Helicopter Model

To perform the tuning process of the AFCS controllers, a suitable helicopter model must first be selected. The helicopter model used in this thesis is a twin-engine light utility helicopter model provided by *TXT e-solutions*. For the validation phase of this work, *TXT e-solutions* also provided the possibility to extrapolate data directly from the same helicopter FFS Level D.

Obviously, a generic certified helicopter full flight simulator employs a complex and highly nonlinear flight model to recreate the behavior of the simulated helicopter. The

detailed description of how this model works is completely beyond the scope of this thesis. However, a brief descriptive explanation regarding the functioning of a generic flight simulator is given in the following in order to understand where the model employed for the tuning process comes from.

4.1.1 Nonlinear Model

The mathematical modeling of helicopter flight behavior presents several challenges as the vehicle is a complex arrangement of interacting subsystems, as illustrated in component form in the Figure below [5].

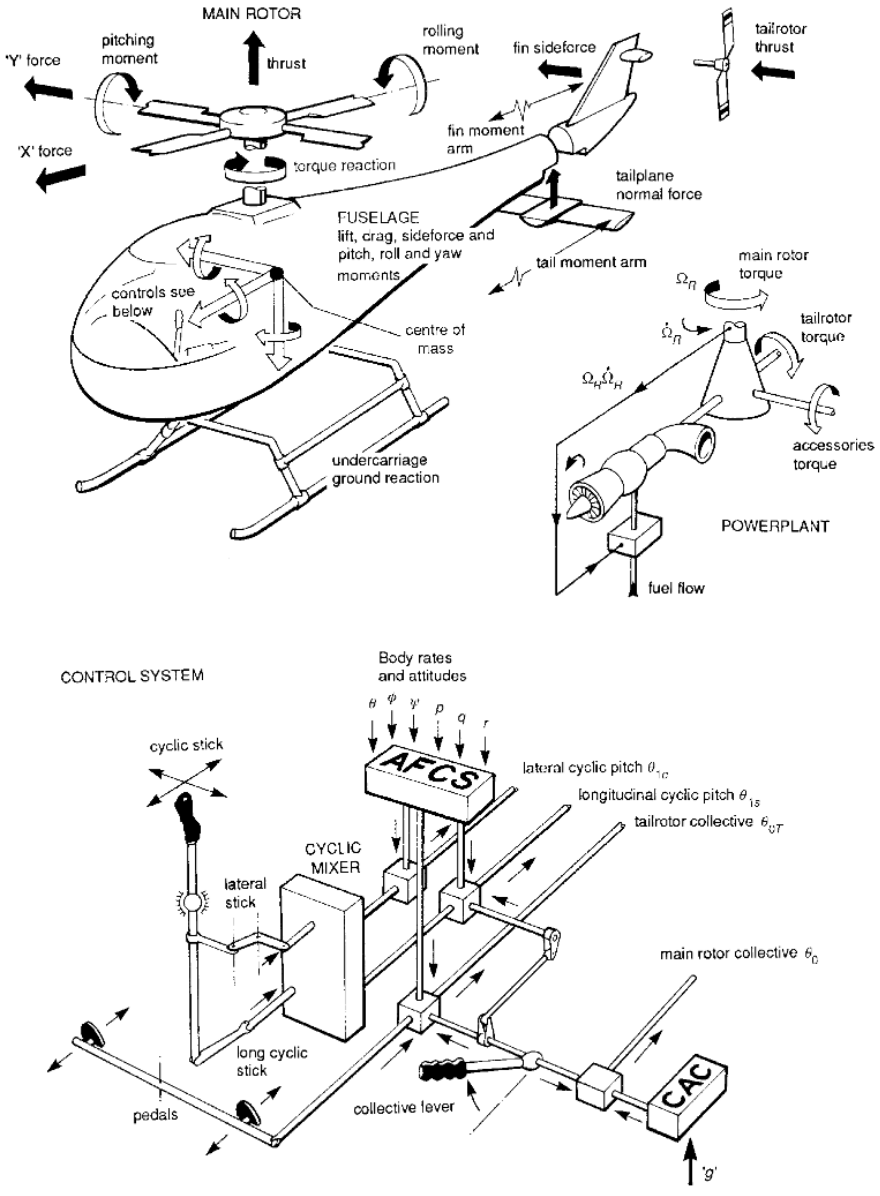

Figure 4.1: Helicopter generic subsystems decomposition

Therefore, a convenient and efficient strategy in the mathematical modeling of such a complex system may be to split the complete model in two parts:

1. the generic modules that are characteristic of each helicopter model.
2. the specific subsystems (e.g. rotor module, engine module, aerodynamics module) characteristic of the particular helicopter model.

This distinction is particularly useful in practice as different subsystems may be modeled or not depending on the aircraft design and configuration. However, a highly recommended approach to developing a mathematical model is to employ the principles of modularity and standardization, as it provides significant efficiency and maintainability benefits. When using a modular approach, it is necessary to define information interfaces, such as specifying which input is required for each module or defining the information that each module should share with external modules. Figure 4.2 below depicts a rough organizational model.

Regarding the division of modules outlined at the beginning, the *Force & Moment Summation* module, the *Equations of Motions* module, and the *Environment* module (which provides air properties throughout the simulation) represent the core modules for any simulation, as they are generic to all types of helicopters. On the other hand, properly modeling each determinant helicopter subsystem is crucial to assess the total forces and moments acting on the aircraft center of gravity. Indeed from their evaluation, it is possible to retrieve from the equations of motion the state variables of the system (as well as numerous ancillary computations) [33].

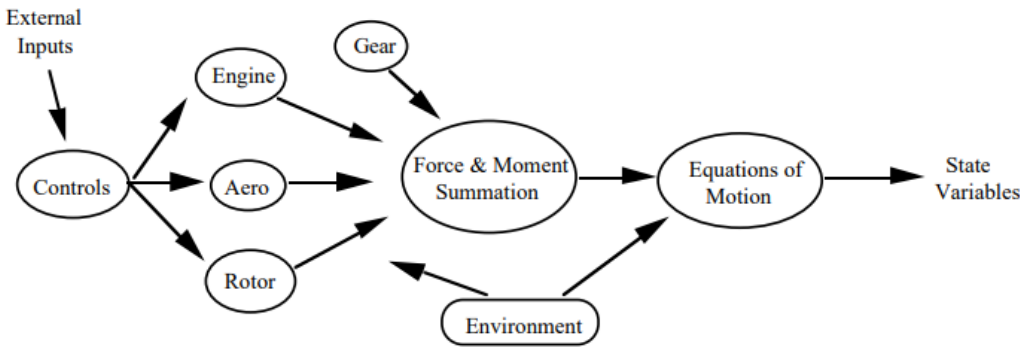

Figure 4.2: Rough Organization of an Helicopter Math Model

Therefore, the core module of each helicopter model is usually the module containing the equations of motion. In this regard, must be noted that usually forces and moments are evaluated in a body-fixed axes system that centers at the aircraft's center of mass and

is oriented at an angle relative to the principal axes of inertia. In this axes system, the *x-direction* points forward along a reference line of the helicopter's fuselage. Formulating the equations of motion involves the application of Newton's laws of motion, which relate the applied forces and moments to the resulting translational and rotational accelerations. Hence, the resulting equations of motion in the *6 DOF Rigid-Body* model of helicopters, which includes three translational and three rotational degrees of freedom, consist of the *Forces Equation* 4.1 and the *Moment Equations* 4.2. The steps required to formulate this system of equations are not included for brevity but can be found in [5].

$$\begin{cases} \dot{u} = -(wq - vr) - g\sin\theta + \frac{X}{Ma} \\ \dot{v} = -(ur - wp) + g\cos\theta\sin\phi + \frac{Y}{Ma} \\ \dot{w} = -(vp - uq) + g\cos\theta\cos\phi + \frac{Z}{Ma} \end{cases} \tag{4.1}$$

$$\begin{cases} I_{xx}\dot{p} = (I_{yy} - I_{zz})\, qr + I_{xz}(\dot{r} + pq) + L \\ I_{yy}\dot{q} = (I_{zz} - I_{xx})\, rp + I_{xz}\,(r^2 - p^2) + M \\ I_{zz}\dot{r} = (I_{xx} - I_{yy})\, pq + I_{xz}(\dot{p} - qr) + N \end{cases} \tag{4.2}$$

In the body axes, the aerodynamic forces applied to the aircraft are denoted by X, Y, and Z, while the aerodynamic moments, corresponding to roll, pitch, and yaw moments, are respectively represented by L, M, and N; the inertial translational velocities in the same moving axes system (body frame) are denoted by u, v, and w; the bank angle, pitch angle, and heading angle are represented by the *321* Euler angles ϕ, θ, and ψ, respectively. The angular velocities measured in the body frame are the roll rate p, the pitch rate q, and the yaw rate r. The fuselage moments of inertia about the reference axes are given by I_{xx}, I_{yy}, I_{zz}, and I_{xz}, while the aircraft mass is denoted by Ma.

Besides the equations above, the relationship between the *321* Euler angles and the airframe angular velocities is also needed and therefore shown in Eqn. 4.3.

$$\begin{cases} p = \dot{\phi} - \dot{\psi}\sin\theta \\ q = \dot{\theta}\cos\phi + \dot{\psi}\sin\phi\cos\theta \\ r = -\dot{\theta}\sin\phi + \dot{\psi}\cos\phi\cos\theta \end{cases} \tag{4.3}$$

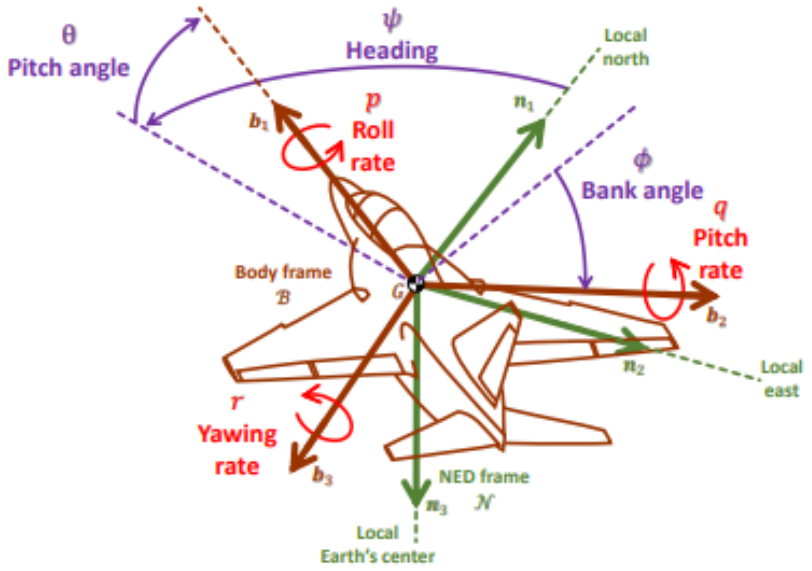

Figure 4.3: Euler angles and vehicle angular velocities

The external aerodynamic forces and moments can be expressed as a sum of contributions from various components and subsystems that are specific to the helicopter considered. In the graph in Figure 4.2, this phase coincides with the *Force & Moment Summation* module. Relevant helicopter modules are modeled and kept as separate as possible from one another, in order to determine the combined forces and moments acting on the rotorcraft. These are then integrated with Equations 4.1, 4.2, 4.3 to obtain the overall equations of motion for the helicopter.

An example of an arrangement architecture that may be used in a helicopter simulation model is shown in Figure 4.4. The simulation model must integrate each subsystem of the helicopter sequentially or concurrently, depending on the processing architecture.

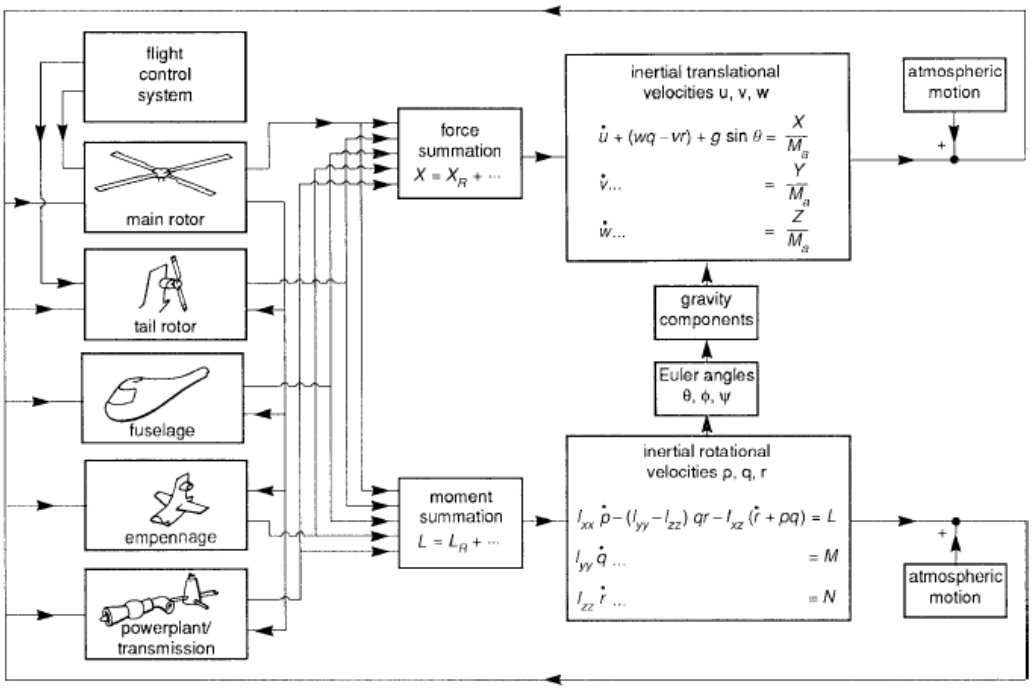

Figure 4.4: Example arrangement of helicopter simulation model [5]

Therefore, following this procedure, the overall helicopter equations of motion are composed by a set of nonlinear differential equations in the form:

$$\dot{\boldsymbol{x}} = \boldsymbol{F}(\boldsymbol{x}, \boldsymbol{u}) \tag{4.4}$$

where the state vector $\boldsymbol{x}$ comprises a number of components that depends on the modeled subsystems. Therefore, the state vector may vary among different helicopter models due to differences in the rotorcraft components and the required level of model accuracy, as determined by the certification body. However, a basic helicopter model should include a state vector that is similar to the one shown below:

$$\begin{cases} \boldsymbol{x} = \{\boldsymbol{x}_f, \boldsymbol{x}_r, \boldsymbol{x}_p, \boldsymbol{x}_c\} \\ \boldsymbol{x}_f = \{u, w, q, \theta, v, p, \phi, r\} \\ \boldsymbol{x}_r = \{\beta_0, \beta_{1c}, \beta_{1s}, \lambda_0, \lambda_{1c}, \lambda_{1s}\} \\ \boldsymbol{x}_p = \{\Omega, Q_e, \dot{Q}_e\} \\ \boldsymbol{x}_c = \{\theta_0, \theta_{1s}, \theta_{1c}, \theta_{0T}\} \end{cases} \tag{4.5}$$

The state subscripts f, r, p, and c used in the equation above represent the fuselage, rotor, engine, and control actuation, respectively. The variables β_0, β_{1c}, and β_{1s} correspond

to the rotor blade coning, longitudinal and lateral flapping angles, while λ_0, λ_{1c}, and λ_{1s} represent the rotor uniform and first harmonic inflow velocities in hub/shaft axes. The main rotor speed is denoted by Ω, and Q_e denotes the engine torque. The variables θ_0, θ_{1s}, and θ_{1c} refer to the main rotor collective pitch angle, longitudinal and lateral cyclic pitch angle, respectively, while θ_{0T} corresponds to the tail rotor collective pitch angle.

The structure of the control vector $\boldsymbol{u}$ remains fixed across different helicopters. While a model designer may define different inputs, the control vector must include at least the main and tail rotor cockpit controls, as illustrated below.

$$\boldsymbol{u} = \{\eta_0, \eta_{1s}, \eta_{1c}, \eta_{0T}\} \tag{4.6}$$

In the control vector $\boldsymbol{u}$, η_0 is the pilot's collective stick input, η_{1s} and η_{1c} are the pilot's cyclic stick inputs while η_{0T} is the pilot's pedal input.

The solution to equation 4.4 relies on the initial conditions, typically the trim state of the helicopter and the time histories of controls and atmospheric disturbances. To calculate the trim conditions, the rates of change of the state vector are set to zero, and the resulting algebraic equations are solved. However, in most practical applications, it is not feasible to define each state of equation 4.4 analytically, and numerical integration methods are employed instead. Therefore, at each time step, following the framework shown in Fig.4.4, the forces and moments on the various components are computed and consolidated to produce the total force and moment at the aircraft center of mass. Finally, the integration scheme allows to evaluate the motion of the aircraft at each time step.

4.1.2 Linearized Model

Despite the nonlinear model of a helicopter represents the highest level of accuracy regarding the behavior of the simulated rotorcraft, it only allows for tackling the rotorcraft's *Response Analysis*. Typically, in a real flight simulator, the nonlinear model of the helicopter is used exclusively for real-time simulations. However, due to the complexity surrounding a helicopter's nonlinear model, analyzing and understanding it from multiple viewpoints can be quite challenging.

Linearization is a process that simplifies nonlinear equations by approximating them with linear equations. This process involves selecting a small region around a specific operating point and assuming that the nonlinear equations can be approximated by a linear equation that describes the system's behavior in that region. The definition of the equilibrium operating point in flight dynamics takes the name of *Trim Problem*. The

resulting linearized model is much simpler and easier to analyze, making it a valuable tool in understanding the helicopter's behavior as well as to perform the so-called *Stability Analysis*.

Furthermore, despite *Nonlinear Control Theory* is a well-developed area of the *Control Theory*, the mathematical techniques developed to handle problems are more rigorous and much less general than the methods offered in the *Linear Control Theory*. Typically, in controller design for aeronautical applications, a solution that is applied is the *Gain Scheduling* approach. This approach is a linear technique for controlling nonlinear or time-varying plants. The idea behind this approach is to compute linear approximations of the plant at various operating configurations, tune the controller gains at each of them, and swap gains during the simulation depending on the operating conditions. This procedure involves the following three major steps:

1. trim and linearize the plant at each operating condition.
2. tune the controller gains for the linearized dynamics at each operating condition.
3. reconcile the gain values to provide a smooth transition between operating conditions.

For the reasons that have just been explained, it is necessary to use the linear models of the helicopter. As a first step, a suitable trimmed flight condition must be selected and evaluated. This involves determining the configuration in which the rate of change of the aircraft's state vector is zero, and the resultant of the applied forces and moments is also zero. To achieve this, the LHS of Equation 4.4 is set equal to zero.

$$0 = \boldsymbol{F}(\boldsymbol{x}, \boldsymbol{u}) \tag{4.7}$$

The trim solution is represented by the zero point of this nonlinear algebraic function, which computes the necessary controls $\boldsymbol{u}_e$ to maintain a defined state $\boldsymbol{x}_e$. Typically, a numerical root-finding algorithm such as the *Newton–Raphson Method* is employed to solve the *Trim Problem*, following the flowchart pattern illustrated in Figure 4.5.

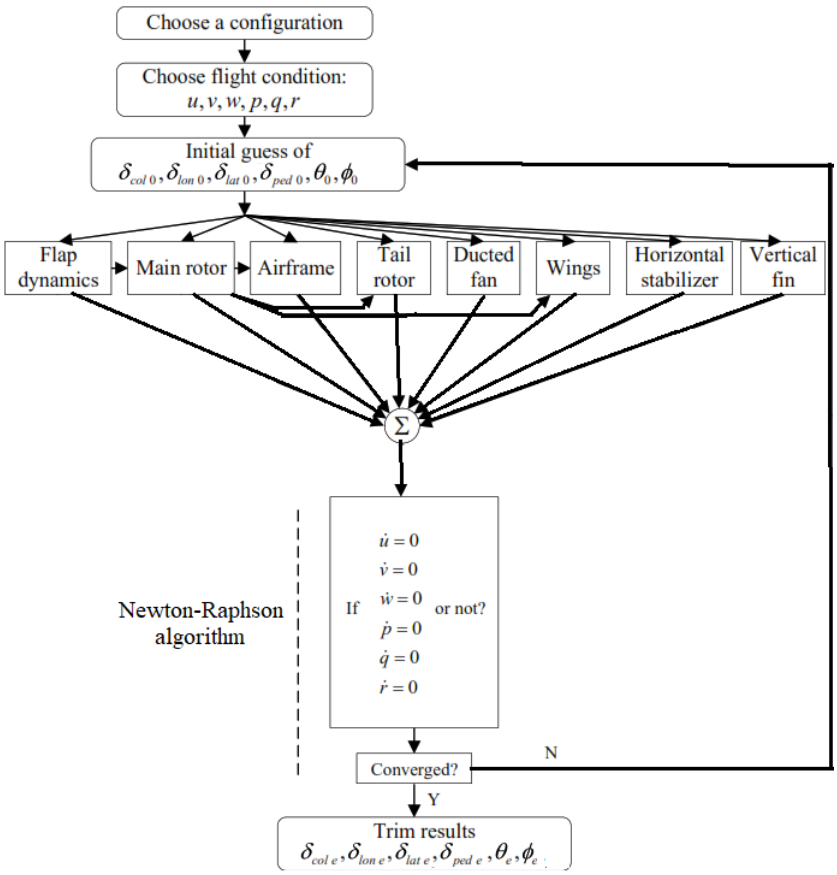

Figure 4.5: Typical Trim Solution Flowchart

From a specific trim point, the nonlinear model model can be linearized. Achieving this requires to recall the nonlinear model, which is expressed in vector form as equation 4.4, and compute its first-order Taylor expansion. As a result, the nonlinear system can be rephrased in the following perturbation form:

$$\dot{\boldsymbol{x}} \approx \boldsymbol{F}\left(\boldsymbol{x}_e, \boldsymbol{u}_e\right) + J_F^x(\boldsymbol{x}_e, \boldsymbol{u}_e)(\boldsymbol{x} - \boldsymbol{x_e}) + J_F^u(\boldsymbol{x}_e, \boldsymbol{u}_e)(\boldsymbol{u} - \boldsymbol{u_e}) \tag{4.8}$$

where $J_F^x(\boldsymbol{x_e}, \boldsymbol{u_e})$ and $J_F^u(\boldsymbol{x_e}, \boldsymbol{u_e})$ form the Jacobian matrix associated with the function $\boldsymbol{F} : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n$ evaluated at the equilibrium point $(\boldsymbol{x}_e, \boldsymbol{u}_e)$, explicitly:

$$J_F^x(\boldsymbol{x}_e, \boldsymbol{u}_e) = \begin{bmatrix} \frac{\partial F_1}{\partial x_1} & \cdots & \frac{\partial F_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial F_n}{\partial x_1} & \cdots & \frac{\partial F_n}{\partial x_n} \end{bmatrix}_{\boldsymbol{x}=\boldsymbol{x}_e, \boldsymbol{u}=\boldsymbol{u}_e} \qquad J_F^u(\boldsymbol{x}_e, \boldsymbol{u}_e) = \begin{bmatrix} \frac{\partial F_1}{\partial u_1} & \cdots & \frac{\partial F_1}{\partial u_m} \\ \vdots & \ddots & \vdots \\ \frac{\partial F_n}{\partial u_1} & \cdots & \frac{\partial F_n}{\partial u_m} \end{bmatrix}_{\boldsymbol{x}=\boldsymbol{x}_e, \boldsymbol{u}=\boldsymbol{u}_e}$$

Then, choosing a trim condition, i.e., $\boldsymbol{F}\left(\boldsymbol{x}_e, \boldsymbol{u}_e\right) = 0$ and neglecting the approximation

error, one can get the linearized model as:

$$\dot{\boldsymbol{x}} = A\boldsymbol{\delta x} + B\boldsymbol{\delta u}$$

where $\boldsymbol{\delta x}$ and $\boldsymbol{\delta u}$ are the perturbations of the state and control vectors from the trim condition. The state matrix A and control matrix B correspond to $J_F^x(\boldsymbol{x}_e, \boldsymbol{u}_e)$ and $J_F^u(\boldsymbol{x}_e, \boldsymbol{u}_e)$, respectively. Hence, the elements in matrices A and B are given by:

$$A_{ij} = \left(\frac{\partial F_i}{\partial x_j}\right)_{\boldsymbol{x}=\boldsymbol{x}_e, \boldsymbol{u}=\boldsymbol{u}_e}$$
$$B_{ij} = \left(\frac{\partial F_i}{\partial u_j}\right)_{\boldsymbol{x}=\boldsymbol{x}_e, \boldsymbol{u}=\boldsymbol{u}_e}$$

Therefore, the linearization process's fundamental assumption is that the aerodynamic forces and moments must be analytic functions of the state and control variables.

Going back to the *Gain Scheduling* control strategy, if the purpose is to implement such a controller design technique, then the aforementioned linearization procedure needs to be carried out as many times as the number of trim points considered. As a result, numerous linear models should be generated, and a suitable linear controller needs to be designed for each one.

However, in this present work, since the *Gain Scheduling* approach implies to repeat the same tuning procedure over different linear models, the controller tuning has been carried out for brevity only for a single operating point being the scope of this work to present a generic AFCS model as well as a tuning methodology for its controllers design.

TXT e-solutions linear model

As already stated before, the case study of this thesis is a twin-engine light utility whose linearized model is provided by *TXT e-solutions*. The matrices that characterise this model are private property of *TXT e-solutions*, hence cannot be shown in the present thesis. However, a description of their content may be provided in the following Tables.

The linear model trim condition is a steady level flight configuration, at *Barometric Altitude* of 5926 $[ft]$ and *Indicated Airspeed* of 68 $[kt]$. The matrices dimension are:

$$\begin{cases} A = 39 \times 39 \\ B = 39 \times 4 \\ C = 12 \times 39 \\ D = 12 \times 4 \end{cases}$$

Matrix States				
Parameter Name	**Symbol**	**UoM**	**Trim Value**	**Reference Frame**
Pitch Rate	q	[rad/s]	0	body
Pitch Angle	θ	[rad]	0.0497	inertial
CG X-velocity	V_x	[m/s]	31.8635	body
CG Z-velocity	V_z	[m/s]	-5.1658	body
CG X-position	X	[m]	0	inertial
CG Z-position	Z	[m]	-1806.4022	inertial
Roll Rate	p	[rad/s]	0	body
Yaw Rate	r	[rad/s]	0	body
Bank Angle	ϕ	[rad]	-0.0044	inertial
Heading Angle	ψ	[rad]	4.4712	inertial
CG Y-velocity	V_y	[m/s]	7.9163	body
CG Y-position	Y	[m]	0	inertial
Rotor Induced Inflow Parameter	λ_0	[-]	0.0416	hub
Rotor Longituinal Induced Inflow Parameter	λ_{1c}	[-]	0.0390	hub
Rotor Lateral Induced Inflow Parameter	λ_{1s}	[-]	-0.0029	hub
Wake Skew Angle	$wake_X$	[-]	0.6387	hub
Wake Spacing	$wake_S$	[-]	0.5270	hub
Wake Longitudinal Curvature	$wake_{Kc}$	[-]	0	hub
Wake Lateral Curvature	$wake_{Ks}$	[-]	0	hub
1st MBC Flap Mode	β_1	[rad]	0.0269	hub
1st MBC Flap Velocity Mode	$d\beta_1$	[rad/s]	0	hub
2nd MBC Flap Mode	β_2	[rad]	-0.0119	hub
2nd MBC Flap Velocity Mode	$d\beta_2$	[rad/s]	0	hub
3rd MBC Flap Mode	β_3	[rad]	-0.0029	hub
3rd MBC Flap Velocity Mode	$d\beta_3$	[rad/s]	0	hub
4th MBC Flap Mode	β_4	[rad]	-0.0013	hub
4th MBC Flap Velocity Mode	$d\beta_4$	[rad/s]	0	hub
5th MBC Flap Mode	β_5	[rad]	-0.0001	hub
5th MBC Flap Velocity Mode	$d\beta_5$	[rad/s]	0	hub
1st MBC Lag Mode	ζ_1	[rad]	-0.0119	hub
1st MBC Lag Velocity Mode	$d\zeta_1$	[rad/s]	0	hub
2nd MBC Lag Mode	ζ_2	[rad]	0.0018	hub
2nd MBC Lag Velocity Mode	$d\zeta_2$	[rad/s]	0	hub
3rd MBC Lag Mode	ζ_3	[rad]	-0.0013	hub
3rd MBC Lag Velocity Mode	$d\zeta_3$	[rad/s]	0	hub
4th MBC Lag Mode	ζ_4	[rad]	0.0001	hub
4th MBC Lag Velocity Mode	$d\zeta_4$	[rad/s]	0	hub
5th MBC Lag Mode	ζ_5	[rad]	0	hub
5th MBC Lag Velocity Mode	$d\zeta_5$	[rad/s]	0	hub

Table 4.1: TXT e-solutions Linearized System States

Matrix Inputs				
Input Name	**UoM**	**Range**	**Description**	**Trim Value**
Longitudinal Cyclic	[-]	[-1.0, 1.0]	-1= Fully Forward; +1=Fully Backward	-0.2014
Lateral Cyclic	[-]	[-1.0, 1.0]	-1= Fully Left; +1=Fully Right	-0.0037
Pedals	[-]	[-1.0, 1.0]	-1= Fully Left; +1=Fully Right	-0.0425
Collective	[-]	[-1.0, 1.0]	-1= Fully Down; +1=Fully Up	0.6258

Table 4.2: TXT e-solutions Linearized System Inputs

Matrix Outputs				
Parameter Name	**Symbol**	**UoM**	**Trim Value**	**Reference Frame**
Pitch Rate	q	[rad/s]	0	body
Pitch Angle	θ	[rad]	0.0497	inertial
Roll Rate	p	[rad/s]	0	body
Yaw Rate	r	[rad/s]	0	body
Bank Angle	ϕ	[rad]	-0.0044	inertial
Heading Angle	ψ	[rad]	4.4712	inertial
CG Y-Acceleration	A_y	[G]	0	body
Radar Height	$Radalt$	[m]	1804.6127	
Longitudinal Groundspeed	GS_x	[m/s]	31.5661	inertial
Lateral Groundspeed	GS_y	[m/s]	7.8934	inertial
Indicated Airspeed	Ias	[m/s]	34.5044	
Barometric Altitude	$Baralt$	[m]	1806.4022	

Table 4.3: TXT e-solutions Linearized System Outputs

4.2 AFCS Model Assembly

Before commencing with the assembly of the Simulink® model, it is worthwhile to discuss a fundamental assumption that has been employed throughout the model's development. In real helicopters, the signal computed by the AFCS is transmitted to an actuator chain. Therefore, in the current AFCS, this component should also be modeled. However, depending on the characteristics of the specific helicopter under consideration, the actuator chain may have varying configurations and purposes. By simply distinguishing between mechanical flight controls, hydro-mechanical flight controls, and fly-by-wire flight controls, one can already comprehend that in order to maintain generality in the development of the AFCS, it is preferable to avoid modeling those actuation chains. Indeed, the aim of this thesis is to model a generic autopilot that can function with any type of helicopter. Furthermore, even in the certified FFS Level D utilized to validate the findings of this study, the AFCS only provides a control input to the series actuator, but the actuators that effectively alter the orientation and position of the swashplate are not even modeled in the simulator. As a result, this custom AFCS is designed to allow for quick integration regardless of the helicopter under consideration. In fact, in this specific instance of a real existing flight simulator, the custom AFCS *C++* code can be exported

and subsequently integrated directly into the actual simulator by simply substituting the existing flight simulator's AFCS. This task would not be feasible, at least with this level of ease, if specific control actuation chains were taken into account during development. Additionally, it has been demonstrated by *TXT e-solutions* know-how that the actuation chain dynamics do not significantly alter the optimal controller gains discovered during the tuning process.

Given this introduction, the primary objective of this section is to elucidate how the Simulink® model depicted in Figure 4.6 was created utilizing the custom blocks stored in the custom library, as well as some other fundamental Simulink® library blocks.

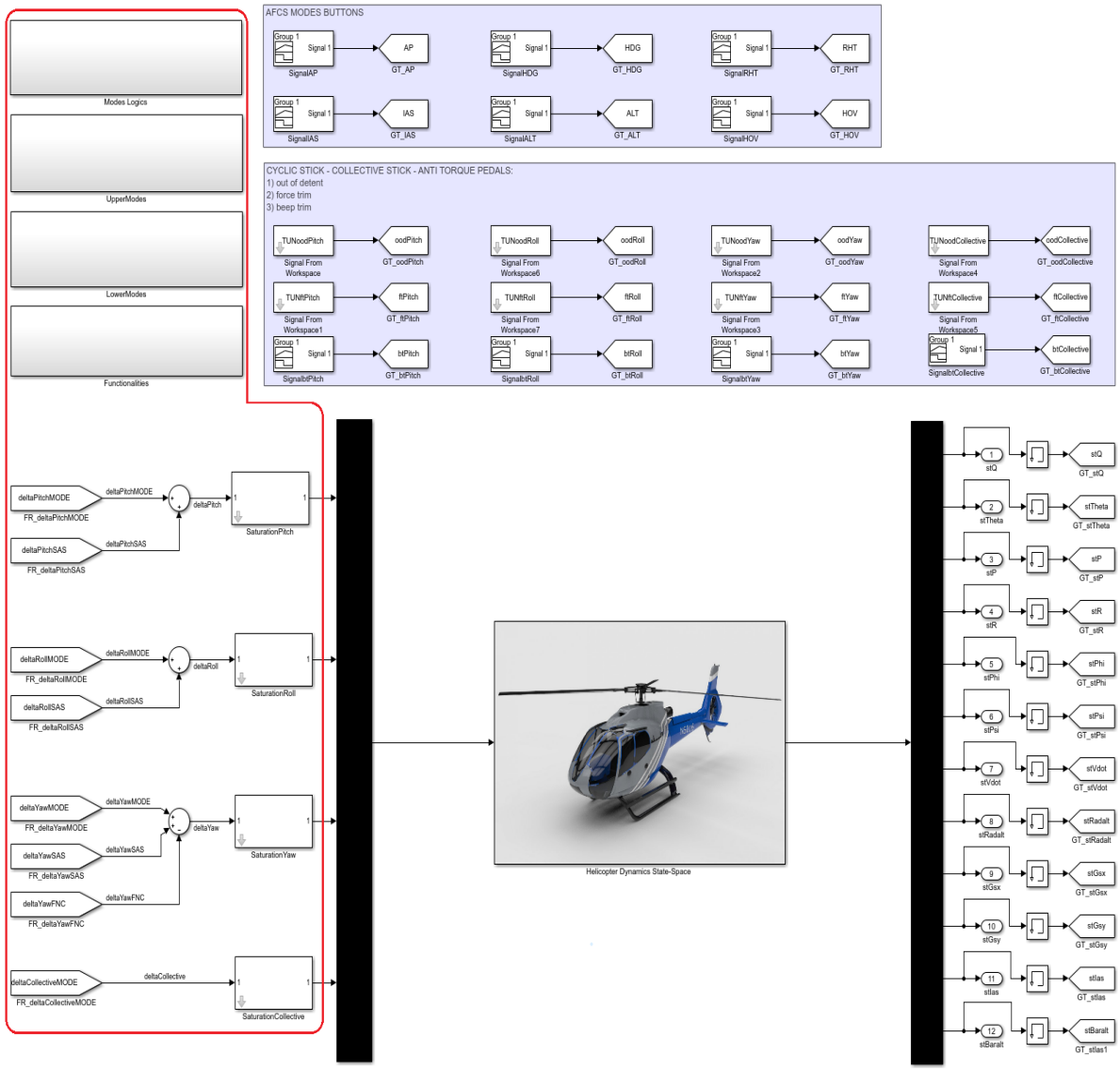

Figure 4.6: AFCS Tuning Model

The Figure shown above represent the highest hierarchical layer of this model. If the AFCS overall structure is the one encircled in red in the picture, the other components, included the state-space representation of the helicopter dynamics, are mainly used for tuning purposes. Those other components are:

1. the APCP buttons, which are modeled for simplicity as *Signal Builder* blocks. The idea is that the pilot presses a generic button for a determined amount of time, and the pression is recorded in Boolean values.
2. the out of detent, force trim and beep trim of each flight control of the generic helicopter. Also in this case, those components may assume only Boolean values.
3. the output measurements of the simulated helicopter dynamics. A *Memory* block is required in each feedback loop since it is necessary to break the algebraic loop generated by the closed-loop configuration [32].

It is important to note that this Simulink® model is intended solely for tuning purposes and is not meant for code generation. However, by making small adjustments, it is possible to retrieve the model of the AFCS intended for export in a *C++* environment and integration into a flight simulator. These adjustments involve mainly the substitution of the components in the bulleted list above with *Input* blocks. Additionally, the helicopter state-space must be removed, and the outputs of each AFCS channel (i.e., Pitch, Roll, Yaw, Collective channels) must be connected with *Output* blocks.

Moving on to the subsystems that compose the AFCS, the first one to be executed during the simulation is the *Modes Logics* subsystem, whose structure is depicted in Figure 4.7.

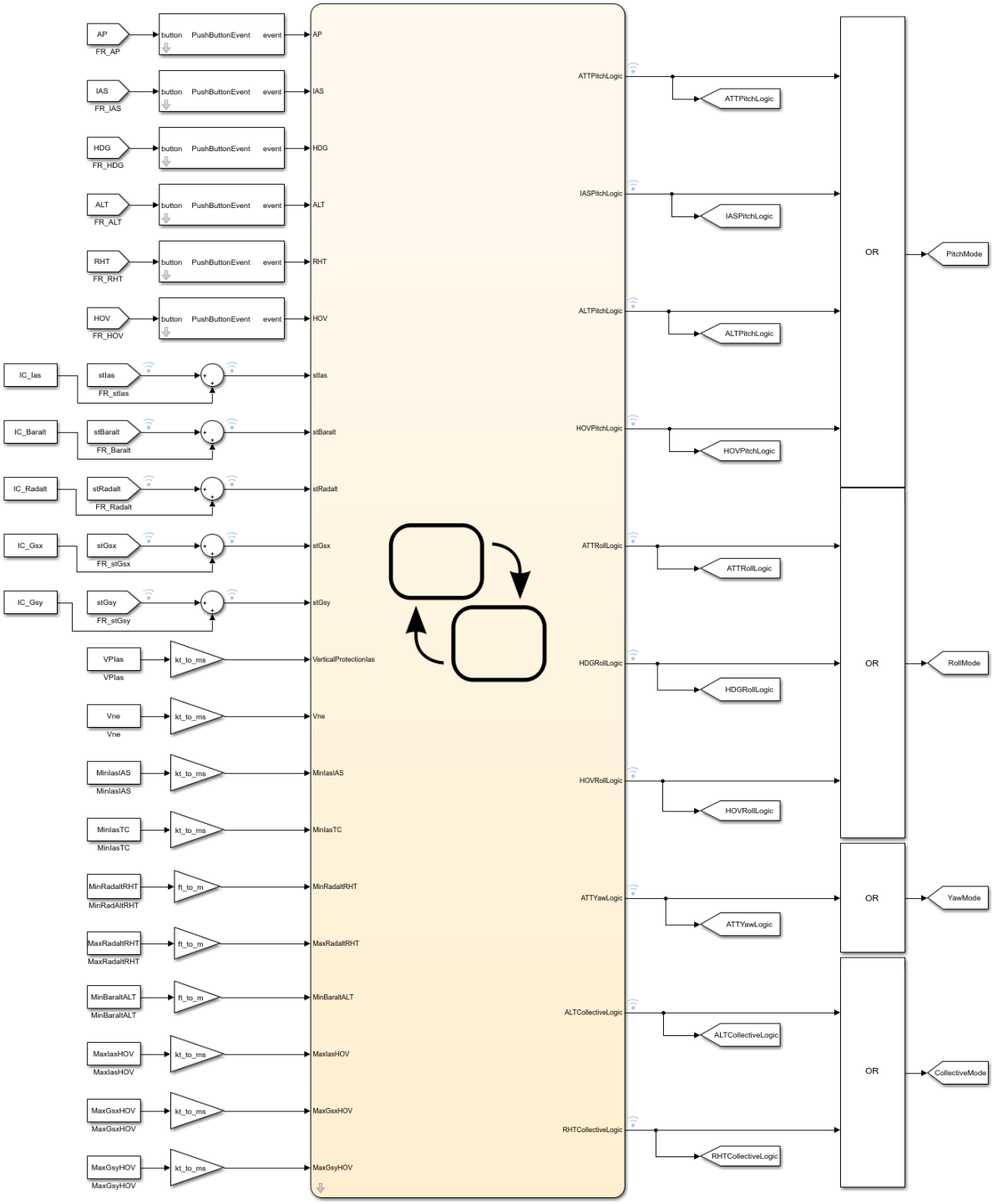

Figure 4.7: AFCS *Modes Logics* Subsystem

This subsystem basically takes as input:

- the pressure recordings of the APCP buttons, which are subsequently analyzed in order to capture only the potential event.

- The values of some of the output measurements of the helicopter dynamics which are required to assess the control logics. A clarification is necessary here: the constant block named *IC_ Output* (e.g. *IC_ Ias*) was added, since in the tuning model the helicopter dynamics simulates at each time step the variation, $\delta Output$, with respect to the trim condition in which the model was linearised. Since the control logics require absolute output measurements instead of their variations, it is necessary to add the trim value to the $\delta Output$ evaluated at each time step. This process is solely for tuning purposes. If the AFCS were to be exported as *C++* code, the constant block and the *Sum* block would be removed.
- the absolute values of some thresholds characteristics of the specific helicopter AP. Those values are necessary to determine some of the control logic conditions within the Stateflow® block.

This subsystem provides in output the logical values of each mode of the AFCS, which in turn are fundamental for the functioning of the *Upper Modes* subsystem, whose structure is depicted in Figure 4.8

The latter aims to replicate the function of each upper mode of the autopilot, as indicated by its name. For the sake of clear visualization and schematic implementation, the upper modes are divided for each helicopter axis command inputs. However, there are a few points to be noted regarding this sybsystem:

1. the yaw axis is not present only because in the present AFCS the only upper mode that may employ this axis is the HDG mode. However, for an implementation choice of the current AFCS, the HDG mode when is engaged on the yaw axis, i.e. when the *Indicated Airspeed* is lower than a certain characteristic threshold of the specific helicopter, this mode coincides with the ATT mode on the yaw axis. In other words, since the HDG upper mode and the lower mode which is beneath its functioning, would coincide in this situation, for practical reason when the HDG is engaged on the yaw axis, in practice it coincides with the ATT mode engaged on this very same axis.
2. upper modes which may be engaged on multiple axis are implemented in separated configurations, one for each potential axis of operation. Hence, there is an ALT mode on the pitch axis and an ALT mode on the collective axis, as well as there is an HDG mode on the roll axis and an HDG mode on the yaw axis (coincident, for what explained just above, with the ATT on the yaw axis).
3. collective modes are by definition the only AFCS modes that are not dependent

from the ATT. Hence they are considered upper modes but basically they behave as they were lower modes as they directly provide in output the command variation to fournish at the actuator of the collective axis. Despite being classified as upper modes, they behave like lower modes by directly issuing command variations to the actuator of the collective axis. This holds true for the collective modes implemented in this work, including ALT and RHT, but may also be accurate for other collective modes.

4. in general, apart from the already explained collective upper modes, every other upper mode provides in output the setpoint change of their respective ATT. However, this variation must be summed with the current value of the relative ATT attitude in order to become the setpoint of the relative ATT.

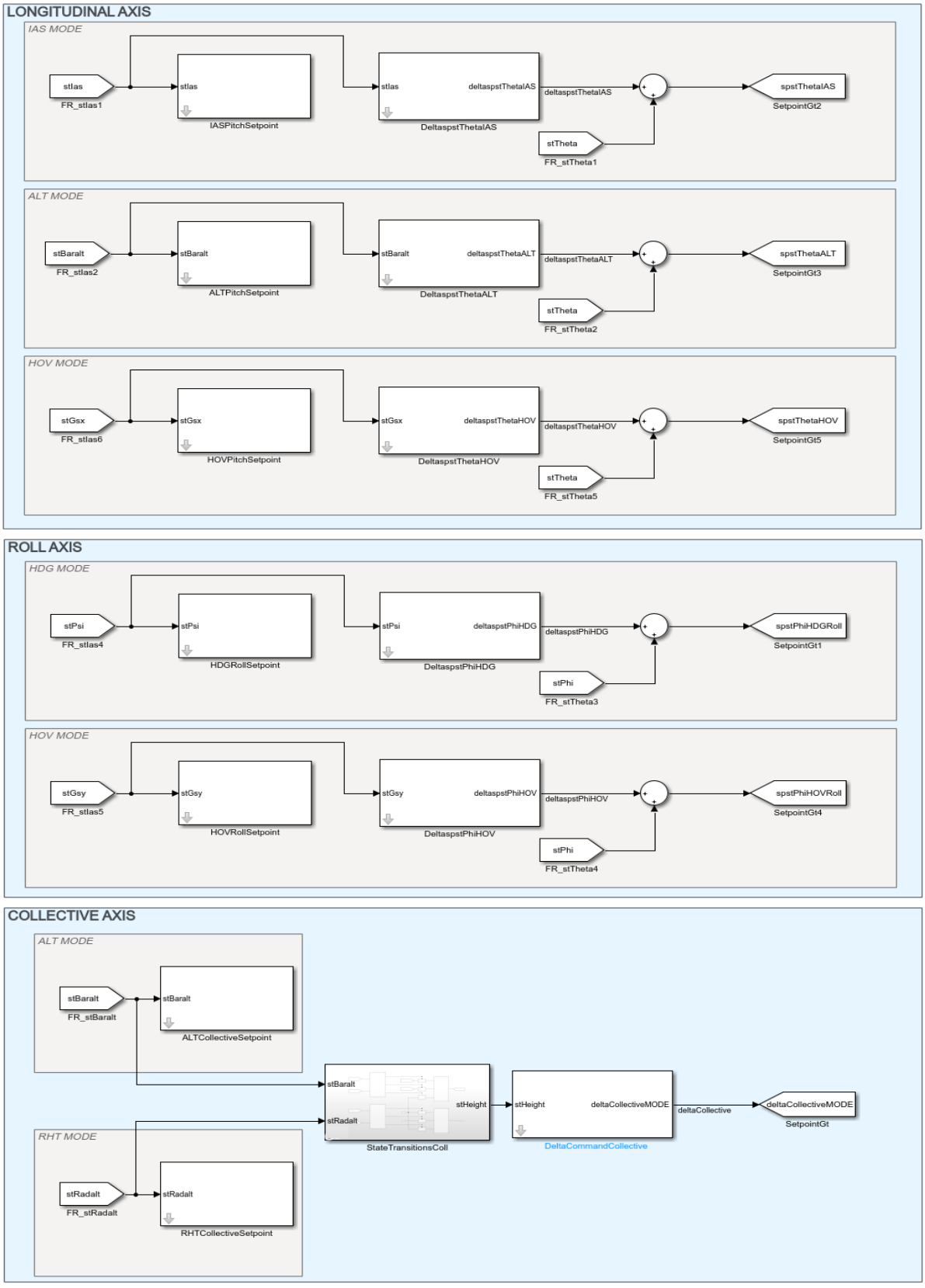

Figure 4.8: AFCS *Upper Modes* Subsystem

The subsystem structure has been obtained through the employment of the *Mode Setpoint* custom block and the *Mode δCommand/δSetpoint* custom block, already encountered respectively in Sec.3.5 and Sec.3.6. The configuration of those masks must be carefully and properly set depending on the mode considered. In Appendix A.1 are shown the masks configuration adopted for building the current AFCS upper modes.

Apart from the case of collective modes, the *GoTo* blocks of this subsystem are redirected to the *Lower Modes* subsystem, whose structure is shown in Figure 4.9. The goal of this subsystem, is to replicate the behaviour of the lower modes of the autopilot. In more detail the task performed by this subsystem are:

1. to provide in output the SAS contribution for each axis.
2. to determine the setpoint of the ATT on each axis.
3. to determine through the *State Transitions* subsystem of each axis, which mode, among the lower and the upper modes, is engaged on the AFCS. This is necessary to provide the correct setpoint to the block in charge of evaluating the command variation.
4. to produce command variations for the pitch, roll, and yaw axes, which are then transmitted to the appropriate actuator.

Also this subsystem structure was created by utilizing the custom blocks, *Mode Setpoint* custom block and the *Mode δCommand/δSetpoint* custom block. The configuration of these blocks, along with the contents of each *State Transitions* subsystem, can be found in Appendix A.2.

Additionally, going back to the Figure 4.6, some *Saturation* blocks have been introduced for each helicopter axis in order to account for the physical limitations of the specific actuators on the rotorcraft.

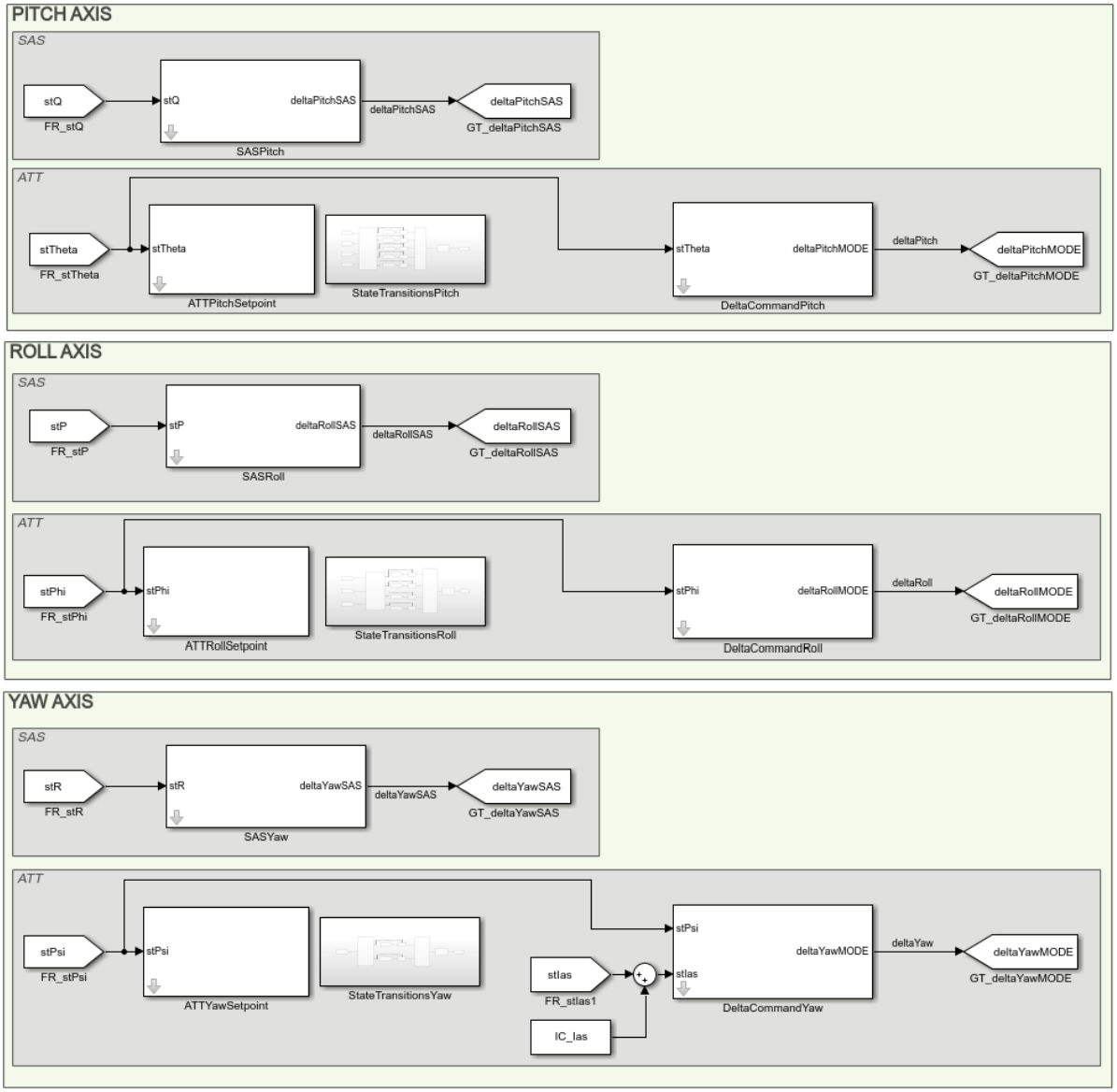

Figure 4.9: AFCS *Lower Modes* Subsystem

The last subsystem that is required to be analysed is the *Functionalities* subsystem which basically just contain the *TC* custom block, as it is the only functionality developed in the present work.

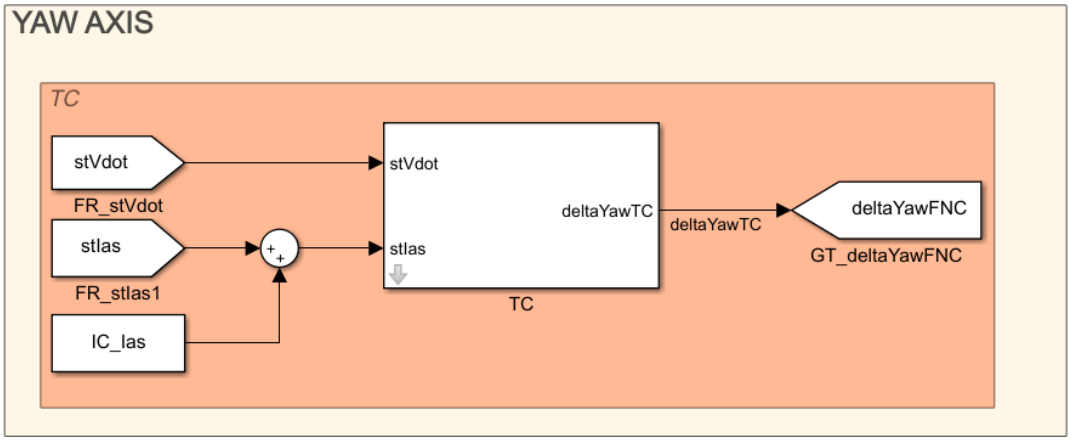

Figure 4.10: AFCS *Functionalities* Subsystem

4.3 AFCS Tuning

In traditional programming, tuning a control system can be a challenging and time-consuming task. The designer relies on their experience to manually adjust control system parameters through trial-and-error. This is especially difficult for control systems with multiple feedback loops or tunable components, such as cascaded PID loops or MIMO control loops with significant cross-coupling. Indeed, in principle, one element or one control loop at a time should be tuned, making the process iterative, time-consuming, and not robust in terms of optimality of the final design.

In contrast, with a model-based design approach, the control system designer has a mathematical representation of the plant, derived from first principles or by data-driven techniques. Model-based tuning can then be applied, allowing for more efficient and automatic techniques to tune a control system.

In particular, given the structure of the current AFCS, it is necessary to tune its PID controllers, based on the linear dynamics of the case study helicopter. In general, in order to tune the PID gains, mainly two broad approaches [29] must be analyzed:

1. Manual tuning, which is basically the use of the designer knowledge of *Control Theory* to choose the proper gains. Several methods may be exploited, however tree of the most popular techniques are:

    - *Pole Placement.* It is a control design technique used to place the closed-loop poles of a system in desired locations in order to achieve the desired

performance characteristics, such as stability, settling time, overshoot, and damping. The PID contoller introduces in the system tunable zeros, poles and gain that allow to modify the closed-loop behaviour.

- *Loop Shaping*. This method looks at the frequency response of the loop transfer function and exploit the knowledge of the open-loop gain margin, phase margin and crossover frequency in order to predict how the closed-loop system would behave. Indeed, by adding the PID controller and adjusting the location of its zeros, poles and gain it is possible to shape the loop transfer function *Bode Diagram* to get the desired frequency characteristics.
- *Heuristics Methods* as *Cohen-Coon* method or *Ziegler-Nichols* method. Those techniques rely on rules of thumb or empirical formulas, rather than rigorous *Control Theory* tools. The goal of these methods is to quickly obtain initial PID gains that can then be fine-tuned to achieve optimal performance.

2. Automatic tuning, which instead, employing the aid of software tools, enable to generate optimal PID gains based on the requirements specified by the control system designer. Usually, most of the auto-tune programs utilize the same techniques as manual tuning, but they are presented in a more streamlined and accessible format.

However, at the end of the parameters computation in both approaches, in order to get the perfect response, it is always a good practice to check if it is needed a manual fine-tune of the PID controller gains.

4.3.1 Simulink Control System Tuner

Since the AFCS Simulink® model relies on PID controllers, it requires the tuning of their gains to attain the desired performance. Therefore, being already employed an auto-tuning methodology in the previous work [32], the same technique may be applied also in this case since the current AFCS shares the same modeling philosophy as the previous thesis.

The auto-tuning suite used is the *Control System Tuner* of Simulink® [9]. This tool can tune any control system architecture by defining design goals. It is capable of tuning multiple fixed-order, fixed-structure SISO or MIMO control elements distributed over any number of feedback loops. In the *Control System Tuner*, various tuning goals are available, including reference tracking, disturbance rejection, loop shapes, pole constraints, closed-loop damping, and stability margins.

The *Control System Tuner* employs different algorithms for each tuning goal since

each goal requires satisfying a distinct task. However, the software solves always the imposed constraint as a minimization problem, where each tuning goal is converted into a normalized scalar value $f(\mathbf{x})$, with $\mathbf{x}$ the vector of tunable parameters defined in the control system. The software then evaluates the optimal parameter values that minimize $f(\mathbf{x})$ (*soft goal* constraint) or reduce $f(\mathbf{x})$ below the value of 1 (*hard goal* constraint).

Regarding the tuning of the current AFCS, the *Control System Tuner* goals employed are:

1. *Tracking of Step Commands* [27]. This requirement is imposed on almost every controller of the AFCS. It enables finding the optimal PID gains that can produce the step response from specific inputs to specific outputs as closely as possible to the desired response. This goal may be applied to constrain SISO or MIMO responses, however in this application only the SISO case is needed. Step response target may be defined in terms of first-order system characteristics (time constant definition is needed) or second-order system characteristics (natural frequency and percent overshoot specifications are needed) or even defining a custom reference system.

2. *Rejection of Step Disturbances* [12]. This goal is used to constrain how a step disturbance injected at a specified location in the control system affects the signal at a specified output location. The desired response can be specified in time-domain terms of peak value, settling time, and damping ratio. This requirement is only used once in the development of AFCS tuning, in the controller tuning of the TC.

3. *Constraint on Closed-Loop Dynamics* [18]. This goal allows to constrain the dynamics of the entire control system or specified feedback loops of the control system, which in practise is a constraint on the dynamics of the *Sensitivity Function* measured at a specified location in the control system. Specifications for minimum decay rate or minimum damping for the poles need to be enforced to properly set the goal. This constraint is applied only for the tuning of the PI controllers of the SAS of the AFCS.

4. *Minimum Stability Margins* [20]. This requirement allows to constrain the gain and the phase margins above some threshold decided by the user. The evaluation of those margins rely on the disk margin theory. Like classical gain and phase margins, disk margins quantify the stability of a closed-loop system against gain or phase variations in the open-loop response. Disk margins also take into account all frequencies and loop interactions. Therefore, disk-based margin analysis provides a stronger guarantee of stability than the classical gain and phase margins.

The complete tuning workflow of the AFCS controllers occurs at the command-line, as shown in Appendix B.1. This is fundamental for this specific tuning process as it allows for the utilization of some Matlab® commands that are not directly accessible from the fixed GUI of the *Control System Tuner*. Those fundamental commands are taken mainly from the Control System Toolbox™(e.g. *TunableGain*, *TunablePID2*). This issue is mainly caused due to the presence in the Simulink® model of custom blocks, such as custom PID blocks, which require different handling [32] with respect to standard PID blocks, and therefore they cannot be directly tuned in the *Control System Tuner*. However, the *Control System Tuner* can still be utilized for a portion of each tuning process. One of its features is the ability to auto-generate the code for the tuning specified in its interface. As a result, a strategy is to establish a tuning process that closely resembles the desired one and then export the generated Matlab® code, which can subsequently be adjusted in the Matlab® *Editor*. This procedure saves time and reduces errors since most of the tuning workflow is auto-generated.

The AFCS is configured in a multiloop feedback setup where each mode has its own tunable controller gains. However, the tuning process cannot be done in a single optimization phase due to the following reasons:

- Firstly, the SAS on the pitch, roll, and yaw axis are always considered to be functioning and must be tuned together. Thus, they need to be the first systems tuned in the AFCS. Once the optimal gains for each SAS are determined, other modes can be tuned. This is because, in a real helicopter, if back-up SAS is activated, those systems may operate and provide stability augmentation on pitch, roll, and yaw axes even if both autopilots are lost. The same working principle does not apply to any of the other AFCS systems implemented in this thesis.
- Secondly, most of the upper modes rely on the ATT, so the ATT must be the second system tuned.
- Thirdly, upper modes that work on the same axis cannot be tuned simultaneously.
- Lastly, the tuning process must align with the control logics requirement of the specific modes. Therefore, it may be necessary to tune a mode while another mode is also engaged in the AP. However, if it is not strictly required by the control logics, this should be avoided since most of the upper modes can be independently engaged in the AP.

Despite the need to repeat the tuning process multiple times, the steps required to carry out a single system tuning are always the same [6] and can be summarized as follows:

1. Ensure that the mode or functionality under tuning is engaged in the AFCS and able to provide the desired behavior. This can be done varying the state of the APCP buttons, *out of detent*, *force trim*, and *beep trim*.

   Just for verification purposes, the Simulink® *Model Linearizer* and its *Linearization Manager* can be used to check if those configurations are correctly imposed and translated into a well-defined linearization path.

2. Select and define the blocks under tuning in the Simulink® model.

3. Select the *Analysis Points* in the Simulink® model. Those locations are mainly used to specify goal definitions in the following.

4. Specify an *Operating Point* for the time instant in which the Simulink® model is linearized. The time instant specified must be subsequent to the time instant in which the mode under tuning is engaged in the AFCS.

5. Build an *slTuner* interface containing the specific Simulink® model, the blocks under tuning, the eventual *Operating Point*, and *Options*. This interface is mandatory for every tuning Matlab® command such as *systune*, which operates on a linear model. *slTuner* automatically computes and stores a linearization of the selected Simulink® model.

6. Construct a tunable model of the control system by setting the specific parameterization of the tuned block. This can be achieved using pre-defined tunable elements such as *TunableGain* or *TunablePID2*, which can be assigned to Simulink® model elements using the *setBlockParam* command. Tunable element's initial, maximum, and minimum values may also be specified.

7. Specify single or multiple tuning goals depending on the specific tuning process (e.g. *Tracking of Step Commands*, *Rejection of Step Disturbances*, etc.).

8. Run the tuning process using the *systune* command.

   Generally, *systune*, *looptune*, and *hinfstruct* tune the controller parameters optimizing the H_∞ norm across a closed-loop system [28]. These commands apply structured H_∞ synthesis, which allows defining the structure and configuration of feedback loops, specifying the parameterization of each tunable component, and combining multiple requirements on separate closed-loop transfer functions.

   Most of those features are not allowed in traditional H_∞ synthesis, and therefore commands as *hinfsyn* or *loopsyn* are not appropriate for this specific AFCS tuning.

4.3.2 Tuning Results

As mentioned previously, the objective of this work is not to replicate the AFCS model of a particular helicopter, nor to fine-tune its controllers to match the behavior of that specific helicopter. Rather, the goal of this thesis is to create a generic AFCS model that can be exported in *C++*, and whose controllers can be tuned using an auto-tuning script capable of specifying multiple generic requirements.

Therefore, the goals imposed throughout the tuning process in Appendix B.1, are not to be intended as the real AFCS requirements used to tune the AFCS gains of this specific helicopter or its certified flight simulator.

Instead, the purpose of the tuning process outlined in Appendix B.1 is to demonstrate how different types of specifications can be imposed and achieved using the auto-tuning methodology described earlier.

However, to validate the AFCS modeling and the employed tuning methodology, the requirements for the tuning of most of the AFCS modes are set based on the responses used to validate this work, which are presented in the next Chapter. Indeed, roughly measuring the response characteristics, such as *settling time* and *overshoot*, it was possible to define a criteria to determine most of the tuning goals.

In this regard, it is worth noting that since the *TXT e-solutions* case study helicopter does not have an upper mode HOV, tuning for this mode is not obviously carried out. However, if it was present, the tuning methodology would be the same as that for the other upper modes.

Additionally, since the tuning process is performed at a specific trim condition (due to the plant controlled being the *TXT e-solutions* linearized model shown in Section 4.1.2), certain tunable blocks defined in the AFCS model cannot be tuned, as their control logics prevent their engagement under these conditions. This is the case of the ATT on the yaw axis, since the *Indicated Airspeed* exceeds the threshold imposed for the functioning of this system. While the same reasoning should apply to the RHT, because the *Radar Height* is above the maximum value for the engagement of this mode, the RHT controller gains are still tuned as this controller is shared between RHT and ALT on the collective axis. In fact, the latter can be engaged even in this trim condition under specific circumstances determined by its control logic.

Furthermore, in each tuning process, the pilot is supposed to fly hands-off and feet-off, relying solely on the AFCS. This procedure enables the tuning of each AFCS controller as it provides a linearized path that differs from zero.

SAS Pitch, SAS Roll, SAS Yaw

As already said, the first loop to be tuned is the most internal one as well as the only one that can operate when both helicopter autopilots fail. Therefore, the PI controller in the SAS requires tuning for its *Proportional* and *Integral* gains. The SAS aims to stabilize the aircraft's angular rate in the short term, specifically on the pitch, roll, and yaw axis, in order to minimize the effects of external disturbances like turbulence.

Therefore, considering the linear dynamics of the case study helicopter, the command *pzmap* allows to retrieve its pole-zero plot, as shown in the Figure 4.11.

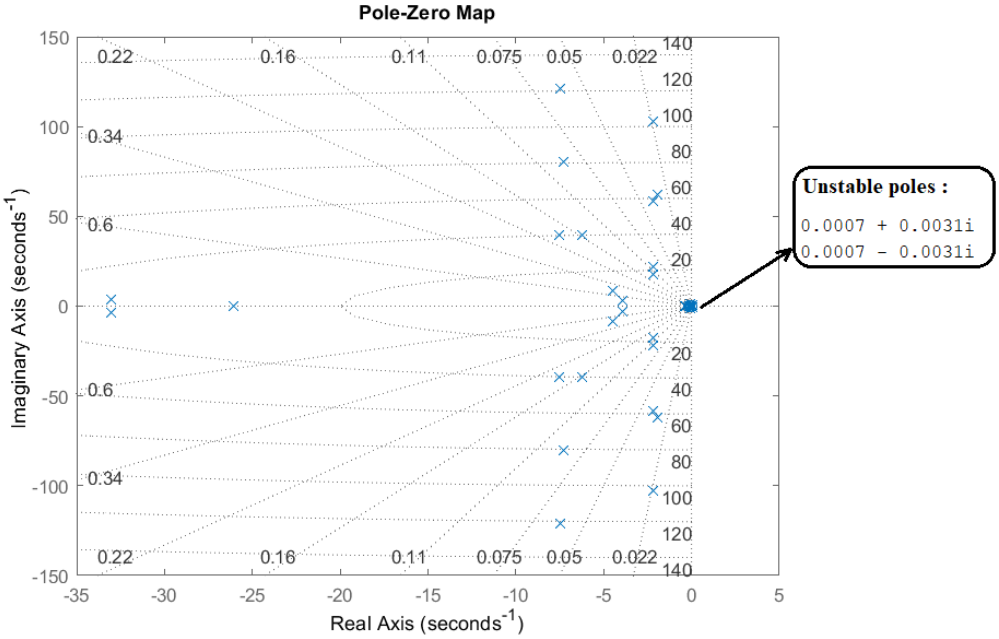

Figure 4.11: Pole-Zero map of the *TXT e-solutions* linear model

Being the dynamics of the helicopter unstable due to a pair of complex conjugate poles in the RHP, the *Stability Augementation System* must shift the poles of the closep-loop system such that every poles is located in the LHP. Therefore, the goal employed in the *systune* tuning process is the *Constraint on Closed-Loop Dynamics* which in turn refers to the command *TuningGoal.Poles*. It constrains the closed-loop poles in terms of minimum decay rate (*mindecay*), minimum damping ratio (*mindamping*) and maximum natural frequency (*maxfreq*).

However, for the SAS only, *TXT e-solutions* was able to provide some flight test responses with the SAS activated. These flight tests were conducted for certification purposes, following the procedures of EASA document CS-FSTD(H). Therefore, based on

these responses and the requirements specified in the CS-FSTD(H) for each SAS, some of the optimal gains found with the *systune* tuning process were manually fine-tuned to meet the certification requirements for the SAS.

Figure 4.12 shows the closed-loop poles obtained with the auto-tuning process and the subsequent manual gains fine-tune.

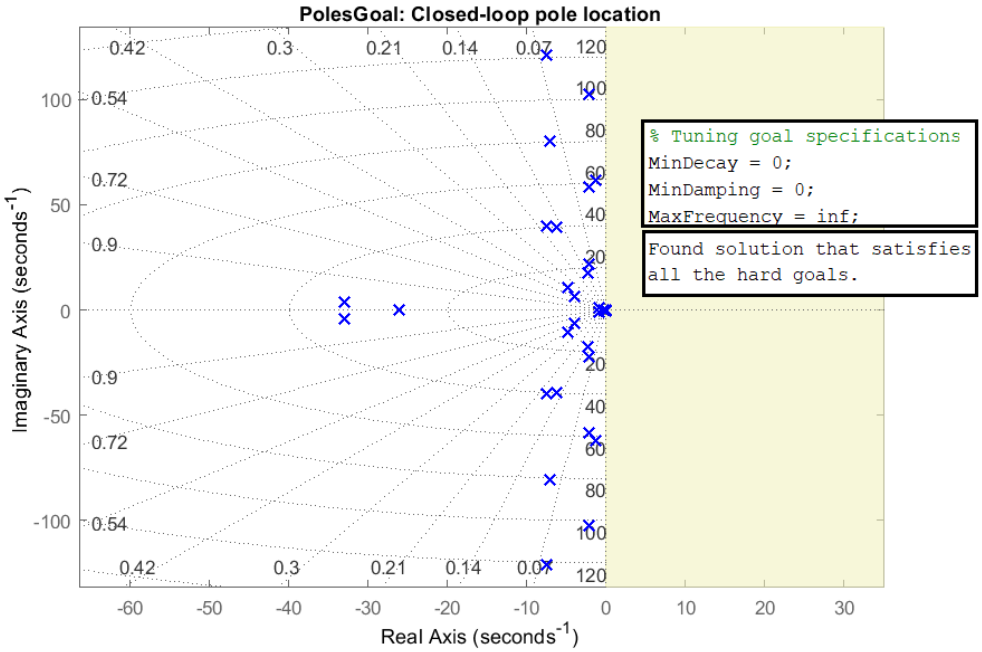

Figure 4.12: Tuning SAS: Closed-Loop Poles

ATT Pitch, ATT Roll, TC

Once the gains for the SAS PI controllers are found, the next step involves the tuning of the *Attitude Hold System*, which is another basic mode of the current AFCS. To obtain the correct linearization path during the tuning process, the AP button must be pressed to turn on the AFCS. Moreover, since the case study trim *Indicated Airspeed* is greater that the maximum threshold for the engagement of the ATT on the yaw axis, the *Turn Coordinator* system provides non-zero control inputs in the yaw axis. Additionally, the SAS stabilizes the aircraft in pitch, roll, and yaw axis.

Therefore, in this trim condition, three PID controllers must be jointly tuned. Those controllers refer to the ATT on the pitch axis, the ATT on the roll axis and the TC.

For the specification of requirements for those systems, the controller designer may decide to impose constraints in the frequency domain. However, since the validation

of the tuning process is carried out based on time-domain responses, the requirements imposed for the tuning are also chosen in the same domain. Therefore:

- The time-domain goal imposed for the ATT on the pitch and for the ATT on the roll axis, is the *Tracking of Step Commands* which refers to the in-line command *TuningGoal.StepTracking*; indeed, the purpose of the *Attitude Hold System* is to set and hold a reference attitude, respectively for the first ATT, the pitch attitude θ, and for the second, the roll attitude ϕ. In this case, reference step response specifications are given in the form of second-order responses [27], specifying desired time constant (*tau*) and overshoot (*overshoot*) for each ATT.
- the time-domain goal imposed for the TC is the *Rejection of Step Disturbances* which refers to the in-line command *TuningGoal.StepRejection*; in fact, the purpose of the *Turn Coordinator* is to annihilate the lateral acceleration sensed by the system. Therefore, in order to generate this kind of acceleration, a roll maneuver must be introduced. Thus, the setpoint of the *Roll Angle* ϕ is given in *input*, and the measurement value of the acceleration A_y (*Vdot* in the Simulink® model) in *output*. The goal is to annihilate the lateral acceleration response that is generated by a step applied in the *input*, given a reference response [12] specified in terms of peak value (*peak*), settling time (*tSettle*), and damping ratio (*zeta*).

However, since those requirements provide only nominal performance for the controlled plant, assuming that, as it is in the real world, the helicopter model is subject to uncertainties, then a robust stability requirement must be imposed. Several techniques may be employed, such as the *Minimum Stability Margins*, which refers to the in-line command *TuningGoal.Margins* [20]. It allows to provide multivariable gain margin (*gainmargin*) as well as phase margin (*phasemargin*) at plant selected locations. In this tuning process, it is supposed necessary to provide gain and phase margins at the plant outputs that are feedback into the AFCS tuned systems. Obviously, those latter requirements are set without any criteria with respect to the real helicopter or its flight simulator.

Therefore, this tuning process involves the joint tuning of the free parameters of three PID controllers subject to four tuning goals. In this regard it is important to note that the specification given in terms of soft goals (*SoftReqs*) and hard goals (*HardReqs*) highly constrain the *systune* algorithm during the optimization. Hence, since in this case, the *TuningGoal.StepTracking* imposed for the ATT on the roll axis and the *TuningGoal.StepRejection* imposed for the TC are conflicting requisites, for better tuning results it is convenient to define one of them as soft goal and the other one as hard goal. For this reason, in this tuning process every requisites is set as an hard goal apart from the TC's

one which is set as soft goal. The results achieved after the tuning process are shown in the Figures 4.13, 4.14, 4.15, 4.16.

The last note concerns the fact that, since the ATT is a prerequisite for the functioning of the upper modes, the gains just found in this tuning process are also used to serve the upper modes operations.

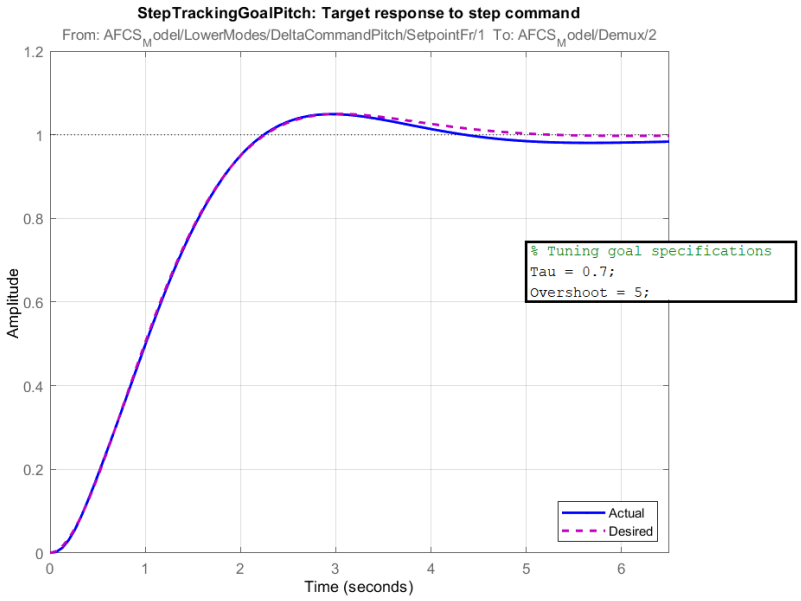

Figure 4.13: Tuning ATT Pitch, ATT Roll and TC: θ_{ref} vs θ

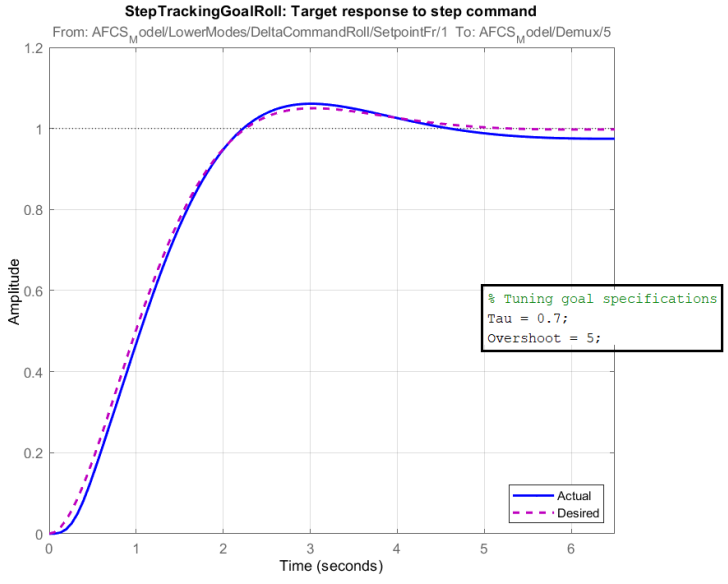

Figure 4.14: Tuning ATT Pitch, ATT Roll and TC: ϕ_{ref} vs ϕ

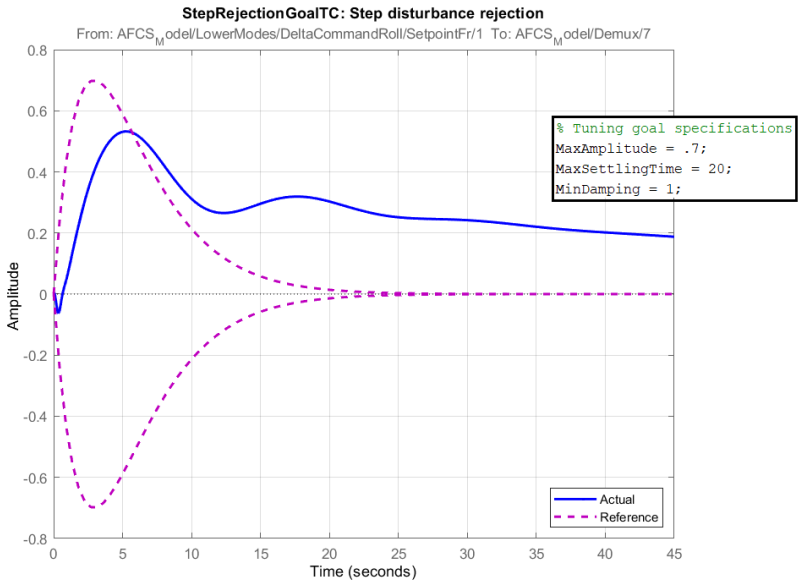

Figure 4.15: Tuning ATT Pitch, ATT Roll and TC: A_{yref} vs A_y

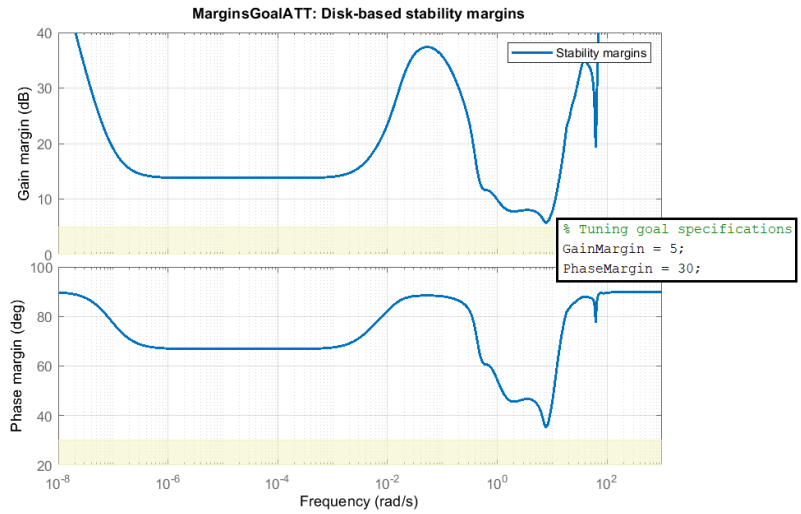

Figure 4.16: Tuning ATT Pitch, ATT Roll and TC: Stability Margins

IAS

The first upper mode that may be tuned is the IAS. This mode provides the capability to set and hold a reference *Indicated Airspeed* during the flight.

In the trim flight condition of the linearized dynamics case study, the IAS mode can be engaged by first pressing the AP button and then the IAS button on the APCP. In

this specific condition, the IAS controls the pitch axis, ATT Roll controls the roll axis, TC controls the yaw axis, while the collective axis remains free. In addition, the SAS operates in all three axes.

The tuning process of the IAS involves the tuning of the PID controller which based on the reference and current values of *Indicated Airspeed*, outputs a variation in $\delta\theta$, which is summed to the current value of θ and provided as setpoint for the ATT Pitch.

Therefore, for the reasons already analyzed, also in this case the requirements may be supplied in the form of *TuningGoal.StepTracking* and *TuningGoal.Margins*, specifying both as hard goals. The results obtained are shown in the Figures 4.17 and 4.18.

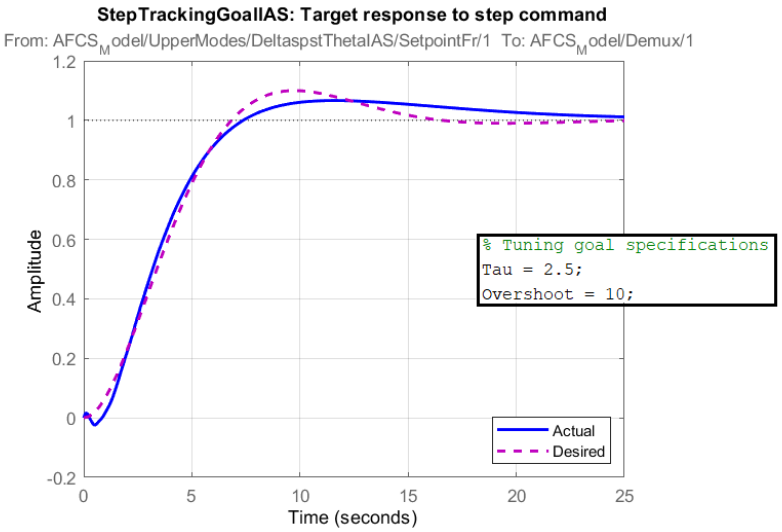

Figure 4.17: Tuning IAS: Ias_{ref} vs Ias

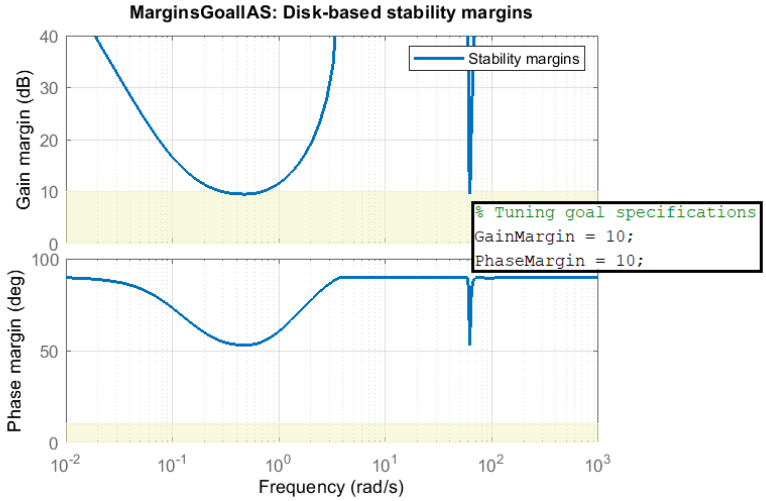

Figure 4.18: Tuning IAS: Stability Margins

ALT Collective

The next upper mode to tune is the ALT. This system provides the capability to set and hold a reference *Barometric Altitude* during the flight. Since this mode can be engaged on either the collective or pitch axes, depending on the flight conditions and AFCS channel available, two separate systems have been defined in the AFCS modeling. Therefore, two separate PID controllers must be tuned.

Starting from the ALT Collective, given the trim flight condition of the linearized dynamics case study, the check on the minimum *Barometric Altitude* is positive, while the check on the *Indicated Airspeed* threshold would provide the engagement of the ALT on the pitch axis. However, if the pitch channel is already in use by another upper mode, the ALT can be engaged on the collective axis.

Therefore, the engagement of the ALT Collective occurs by triggering the AP button, then engaging a pitch axis upper mode (e.g. IAS) and finally triggering the ALT button on the APCP. Therefore, in this specific condition, the pitch axis is controlled by the IAS, the roll axis by the ATT Roll, the yaw axis by the TC, the collective axis by the ALT Collective. Additionally, the SAS operates in all three axes.

The tuning process of this mode involves the tuning of the PID controller which based on the reference and current values of *Barometric Altitude*, outputs a command variation $\delta Collective$.

Requirements are set in the form of *TuningGoal.StepTracking* and *TuningGoal.Margins*, specifying both as hard goals. The results obtained are shown in the Figures 4.19 and 4.20.

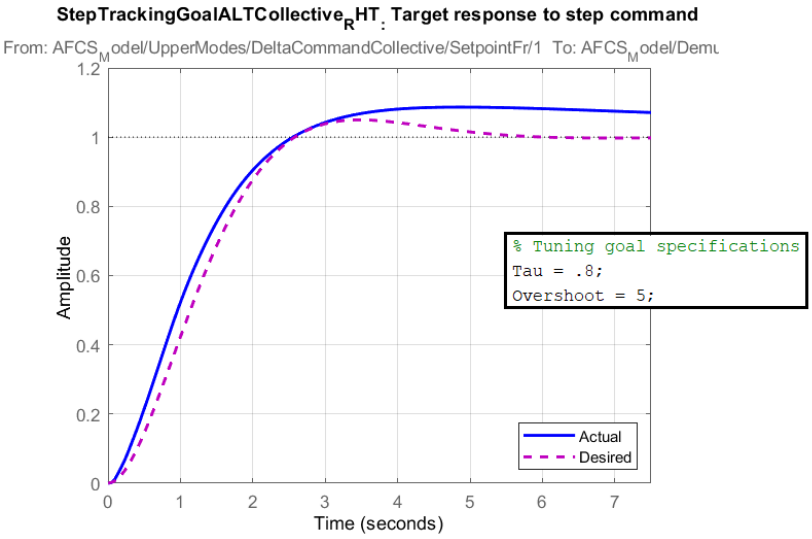

Figure 4.19: Tuning ALT Collective: $Baralt_{ref}$ vs $Baralt$

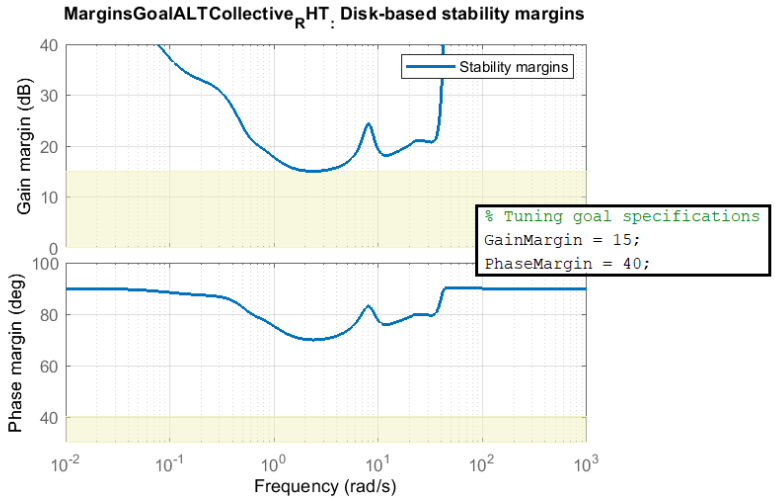

Figure 4.20: Tuning ALT Collective: Stability Margins

ALT Pitch

Regarding instead the tuning of the upper mode ALT Pitch, the engagement procedure is more straightforward. Firstly, the AP button must be triggered and then the ALT button must be pressed. Once engaged, the ALT Pitch system controls the pitch axis, while the roll axis is controlled by the ATT Roll, the yaw axis by the TC, and the collective axis is left free. The SAS operates in all three axes.

When the ALT Pitch is engaged, its PID controller can be tuned. This controller takes the reference and the current values of *Barometric Altitude*, and outputs a a variation

in $\delta\theta$, which is summed to the current value of θ and provided as setpoint for the ATT Pitch.

As with the other upper modes, the tuning process for ALT Pitch involves setting goals using *TuningGoal.StepTracking* and *TuningGoal.Margins*, both specified as hard goals. The achieved results are shown in Figures 4.21 and 4.22.

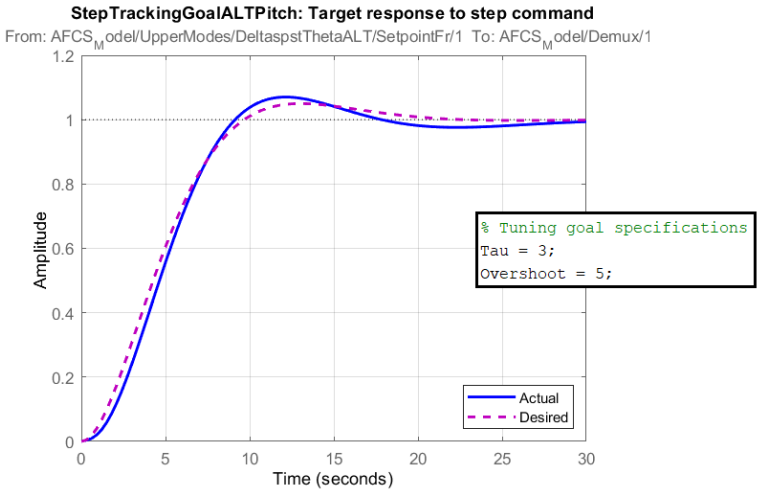

Figure 4.21: Tuning ALT Pitch: $Baralt_{ref}$ vs $Baralt$

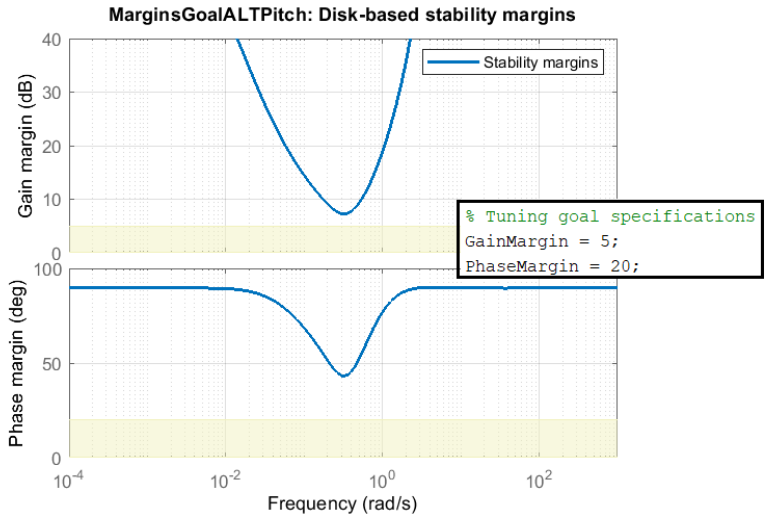

Figure 4.22: Tuning ALT Pitch: Stability Margins

HDG Roll

Finally, the last upper mode which can be tuned is the HDG Roll. This system provides the capability to set and hold a reference *Heading Angle* during the flight. This mode can

be engaged either on the roll or yaw axes, depending on the operating *Indicated Airspeed*. In the yaw case this mode coincides with the functionalities provided by the ATT on the yaw axis.

As already said, the trim flight condition of the case study linearized dynamics doesn't allow for the engagement of the HDG on the yaw axis. Instead, to engage the HDG on the roll axis, it is necessary to trigger the AP button and then select the HDG button on the APCP.

In this situation, the pitch axis is controlled by the ATT Pitch, the roll axis by the HDG Roll, the yaw axis by the TC. SAS are operating in all the three axes.

The tuning process of this mode involves the tuning of its PID controller which based on the reference and current values of ψ, outputs a a variation in $\delta\phi$, which is summed to the current value of ϕ and provided as setpoint for the ATT Roll.

Requirements are provided in the form of *TuningGoal.StepTracking* and *TuningGoal.Margins*, specifying both as hard goals. The results obtained are shown in the Figures 4.23 and 4.24.

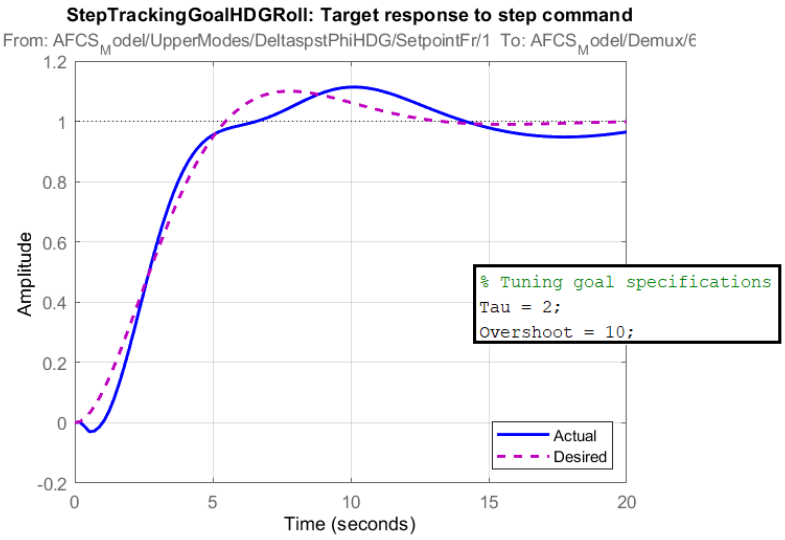

Figure 4.23: Tuning HDG Roll: ϕ_{ref} vs ϕ

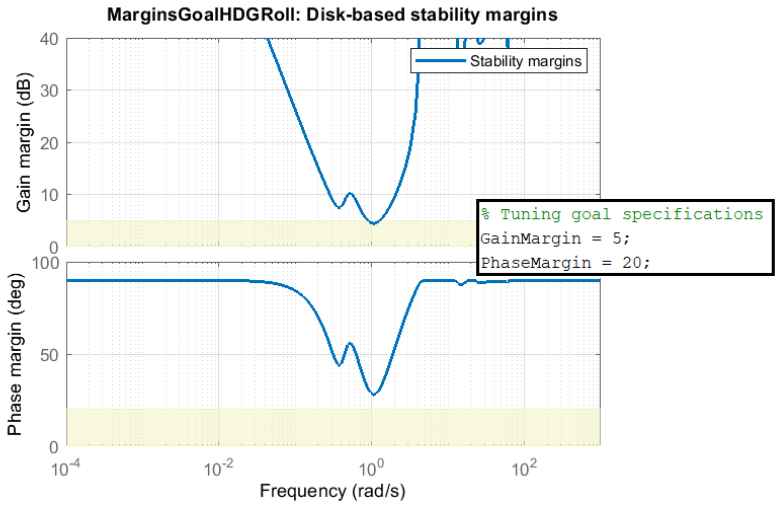

Figure 4.24: Tuning HDG Roll: Stability Margins

5 | AFCS Model Validation

In the so-called *V-Model*, the missing passages to complete the Simulink® model-based design of the current AFCS are related to the *Code-Generation*, *Integration* and *Validation* phases.

However, in the previous work [32], the proofs of concept of the *Code-Generation* and *Integration* phases have already met satisfying results. Indeed, the previous generic AFCS model was shown to be suitable for code export and integration within a real helicopter flight simulator. In the present work, despite the complete reorganization of the AFCS Simulink® model structure and the addition of new features, the design methodology and modeling philosophy have remained unchanged. Additionally, as already explained in 4.2, by making small adjustments, it is possible to retrieve the model of the AFCS intended for code generation from Simulink® environment. This strategy allowed to keep the previous results as granted also for this work, while focusing more on the modeling and simulation of new components. Therefore, the model-based design phase that cannot be skipped in the current work, is the last one, the *Validation* phase.

For this reason, this Chapter outlines the test methodologies employed to verify the correctness of the present AFCS model.

5.1 Upper Modes Control Logics

Stateflow® environment provides powerful tools not only for modelling state machines but also for analyze their behaviour and debug complex transitions.

Those debugging techniques rely on:

- the Stateflow® chart animation [8]. During simulation, animation provides visual verification of expected behaviour of modeled charts. Animation highlights active objects in a chart as execution progresses. Speed animation may be changed and set as *Lightning Fast*, *Fast*, *Medium*, *Slow*, *None* (which turn off animation).
- the Stateflow® *breakpoints* [7] and the data inspection tools [10]. A *breakpoint* is a

circular red badge set on a Stateflow® element (e.g. on charts, states, transitions, graphical or truth table functions, events, etc.) which pauses the simulation in order to examine the status of the chart and check the values of the current data through, for example, the *Symbols Pane* interface.

The Figure 5.1 shows an example animation taken from the debugging of the *Modes Logics Stateflow®* custom block.

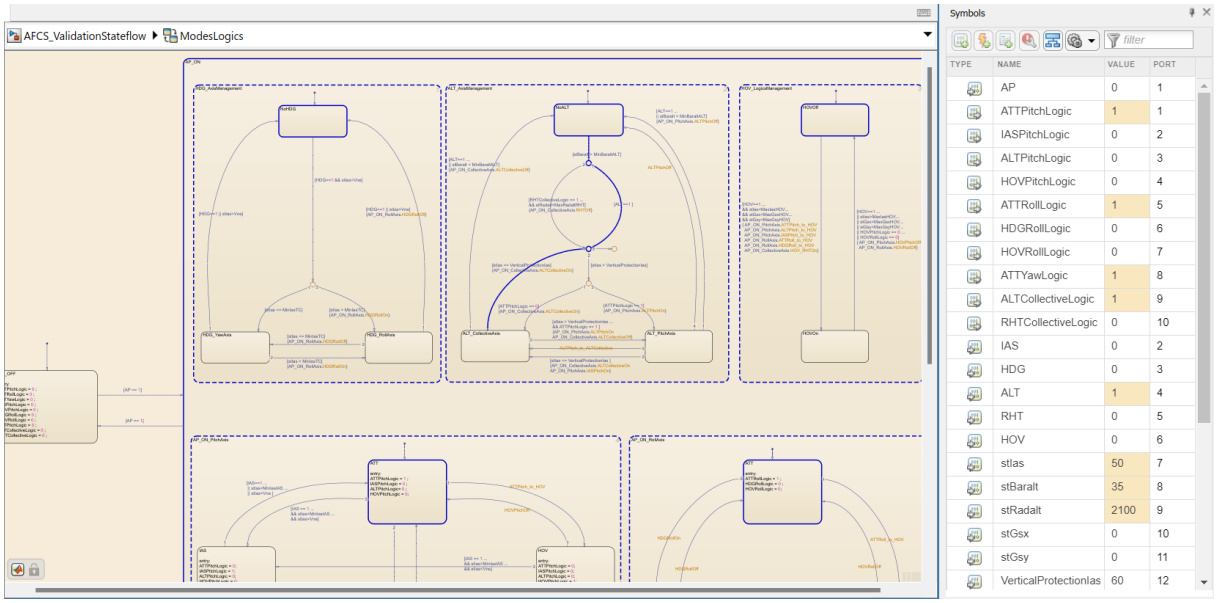

Figure 5.1: Example Debugging of the *Modes Logics Stateflow®* custom block

Therefore, to validate the correct functionality of the upper modes control logics modeled as state machines, it is necessary to define test cases that cover all possible states and transitions of the state machine, and verify that the output signals are correct. Additionally, during the simulation of each test, the debugging tools explained above can provide another source of charts behavior assessment, which can be useful in cases where the potential error is not directly evident from the output of the state machine.

For the specific validation process of the *Modes Logics Stateflow®* custom block, once the functional tests have been identified, the standard *Signal Builder* block can be employed to translate those test cases in Simulink® environment [19]. Finally, running each test, it is possible to verify if the simulated behaviour matches the expected test output. The validation model of the upper modes control logics in shown in Figure 5.2.

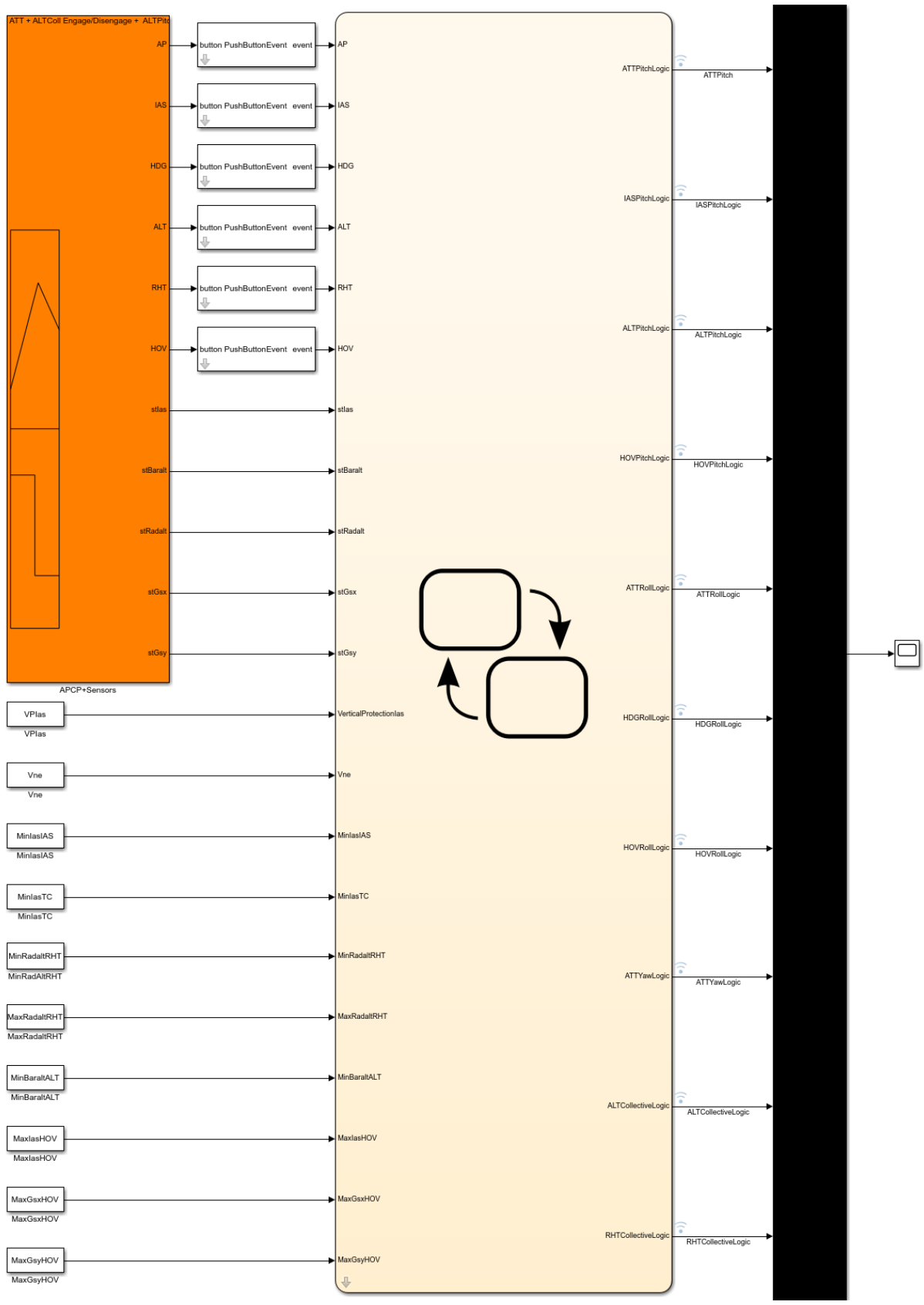

Figure 5.2: Validation Model of the *Modes Logics Stateflow*® custom block

For instance, one of the most complex test case developed is illustrated in Figure 5.3. During this test's simulation, all AFCS modes are engaged and disengaged. This is achieved by triggering most of the APCP buttons. However, this is not the only source of engagement or disengagement of modes of this test, as also numerous automatic transitions occur due to the variation of flight parameters such as Ias or $Radalt$.

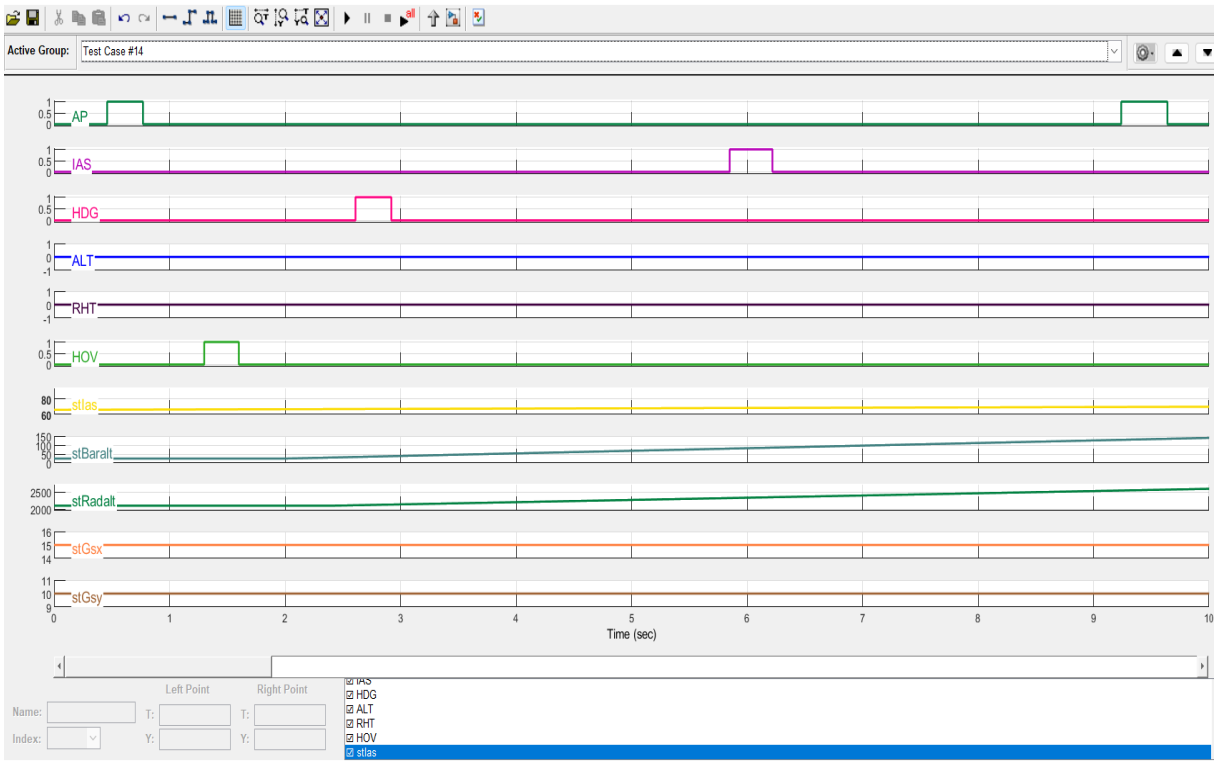

Figure 5.3: Example Test Case of the Upper Modes Control Logics validation

The outputs generated by this test and displayed in Figure 5.4, must be compared with the input signal depicted in Figure 5.3. Here, it is crucial to carefully check whether the behavior of the simulated model matches the expected behavior.

In this example case, the simulation shows the following events:

- at $t \approx 0.5s$, the AP button is pressed, which results in the automatic engagement of ATT mode in the AFCS. The pitch, roll, and yaw axes are controlled by ATT Pitch, ATT Roll, and ATT Yaw, respectively, while the collective axis remains free.
- At $t \approx 1.3s$ the HOV button is pressed and this mode is engaged in the AFCS. This leads to the disengagement of the ATT Pitch and the engagement of the HOV Pitch on the pitch axis, the disengagement of the ATT Roll and the engagement of the HOV Roll on the roll axis, the engagement of the RHT on the previously free collective axis. The mode engaged on the yaw axis remains the same.

- At $t \approx 2.6s$ the HDG button is pressed and this mode is engaged in the AFCS. It causes the disengagement of the HOV Pitch and the automatic backup engagement of the ATT Pitch on the pitch axis, the disengagement of the HOV Roll and the engagement of the HDG Roll on the roll axis, while the other axes remain unchanged.
- At $t \approx 3.9s$ the *Radalt* exceeds the maximum threshold for RHT functioning, and as a result, this mode is automatically disengaged from the AFCS. Being the ALT the backup mode of the RHT, the ALT Pitch is engaged on the pitch axis since the Ias is above the $VPIas$ and no upper modes are engaged on the pitch axis.
- At $t \approx 5.8s$ the IAS button is pressed and this mode is engaged in the AFCS. This results in the disengagement of the ALT Pitch and the engagement of the IAS on the pitch axis. At this point the ALT mode is automatically transitioned from the pitch axis to the collective axis and therefore the ALT Collective is engaged.
- At $t \approx 9.2s$ the AP button is pressed, and this turns off the autopilot. As a result, all the modes engaged in the AFCS are automatically disengaged.

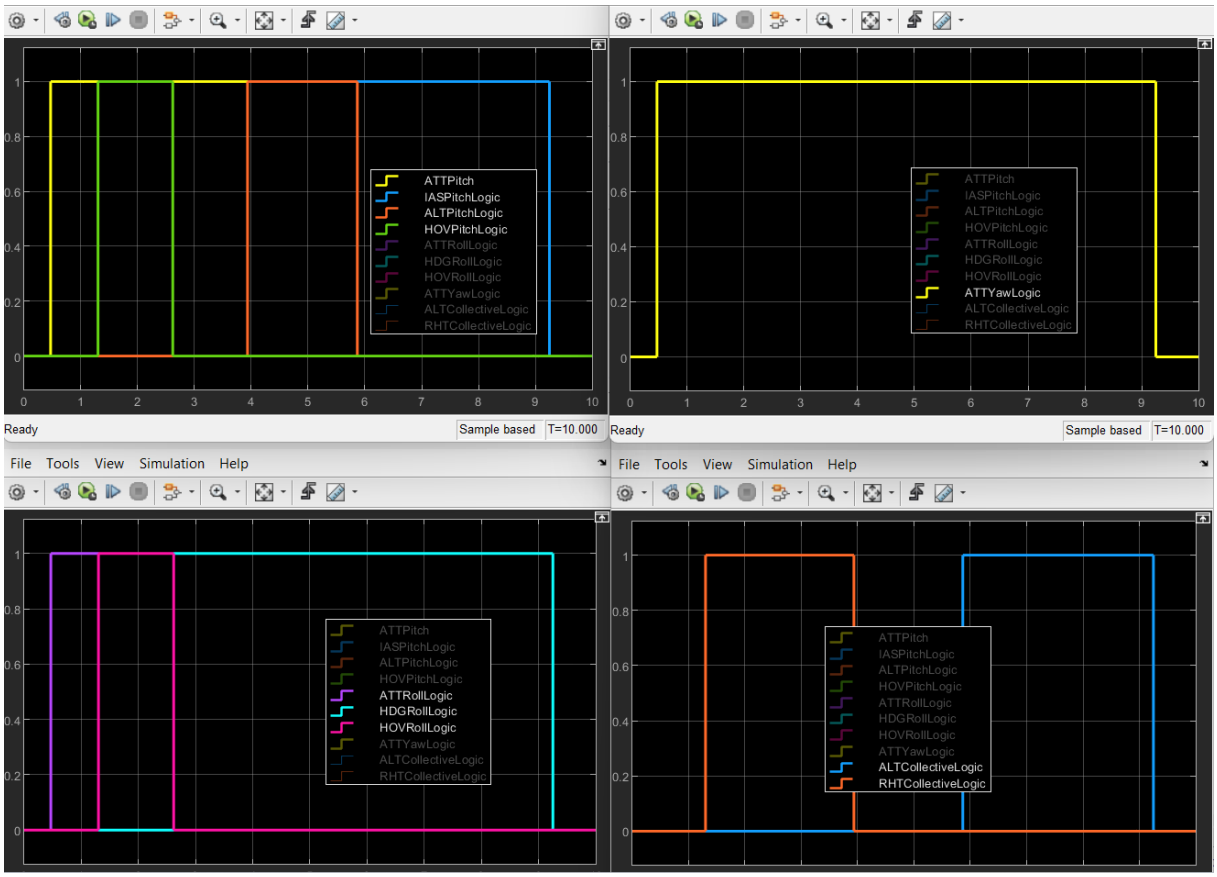

Figure 5.4: Engagement of Modes in Pitch, Roll, Yaw and Collective axis during the Simulation of the Control Logics Example Test Case

Numerous tests have been carried out varying numerous flight conditions and triggering the activation of different combination of upper modes. Since at the end of the process each test was successful, the upper modes control logics are resulted robust enough to be validated.

5.2 Stability Augmentation System

As discussed in the previous chapter, real flight test data provided by *TXT e-solutions* were used to fine-tune the SAS on the pitch, roll, and yaw axis.

In principle, these flight test data were originally used to certify the existing FFS Level D, demonstrating that the simulated and the real helicopter behaviour were closely matching. Indeed, the flight test campaigns required for the certification of a helicopter flight simulator must adhere to EASA guidelines outlined in the document CS-FSTD(H).

However, if some of these flight tests were conducted under conditions close to the trim point for which the case study's linearized dynamics matrices were produced, then the same flight tests could be utilized to tune and validate the present work.

Therefore, this current work utilizes two validation FSTD tests listed in the *MC1 FSTD(H).300 Qualification Basis* [3], namely the *2.c.(1)* test and the *2.d.(1)(i)* test. Figures 5.5 and 5.6 respectively provide explanations for these tests.

Since the purpose of this validation is to assess the correct behaviour of the SAS while their are functioning and not to prove the helicopter model truthfulness, then only the flight condition *Cruise Stability Augmentation On* must be taken into account. In both of those tests, helicopter is excited with cyclic step input, respectively for *2.c.(1)* in the longitudinal plane and for *2.d.(1)(i)* in the lateral one. The test is passed if the measured values of the simulated results fall within the certification body's imposed tolerance.

TESTS		TOLERANCE	FLIGHT CONDITIONS	FSTD LEVEL											COMMENTS
				FFS				FTD			FNPT				
				A	B	C	D	1	2	3	I	II	III	MCC	
2.	**HANDLING QUALITIES**														
c.	Longitudinal Handling Qualities														
	(1) Control response	Pitch rate ± 10 % or ± 2°/s Pitch angle change ± 10 % or ± 1.5°	Cruise Stability augmentation on and off		✓	✓	✓		C T & M	✓					Two cruise airspeeds to include minimum power required speed. Step control input. Off axis response must show correct trend for unaugmented cases.

Figure 5.5: CS-FSTD(H) Validation Test 2.c.(1)

EASA

Easy Access Rules for Helicopter Flight Simulation Training Devices (CS-FSTD(H)) (Initial issue)

SUBPART C – HELICOPTER FLIGHT SIMULATION TRAINING DEVICES

TESTS	TOLERANCE	FLIGHT CONDITIONS	FSTD LEVEL											COMMENTS
			FFS				FTD			FNPT				
			A	B	C	D	1	2	3	I	II	III	MCC	
2. HANDLING QUALITIES														
d. Lateral & Directional Handling Qualities														
(1) Control response (i) Lateral	Roll rate ± 10 % or ± 3°/s Bank angle change ± 10 % or ± 3°	Cruise stability augmentation on and off		✓	✓	✓		C T & M	✓	✓	✓	✓	✓	Two airspeeds to include one at or near the minimum power required speed. Step control input. Off axis response must show correct trend for unaugmented cases.

Figure 5.6: CS-FSTD(H) Validation Test 2.d.(1)(i)

Therefore, to carry out this validation, a validation model of the current AFCS has been developed specifically for this purpose. The model allows to simulate pilot inputs in the form of longitudinal and lateral cyclic inputs, pedals inputs, and collective inputs, as depicted in Figure 5.7. These control actions are combined with the ones provided from the AFCS, and the total control input is then supplied to the linearized helicopter dynamics.

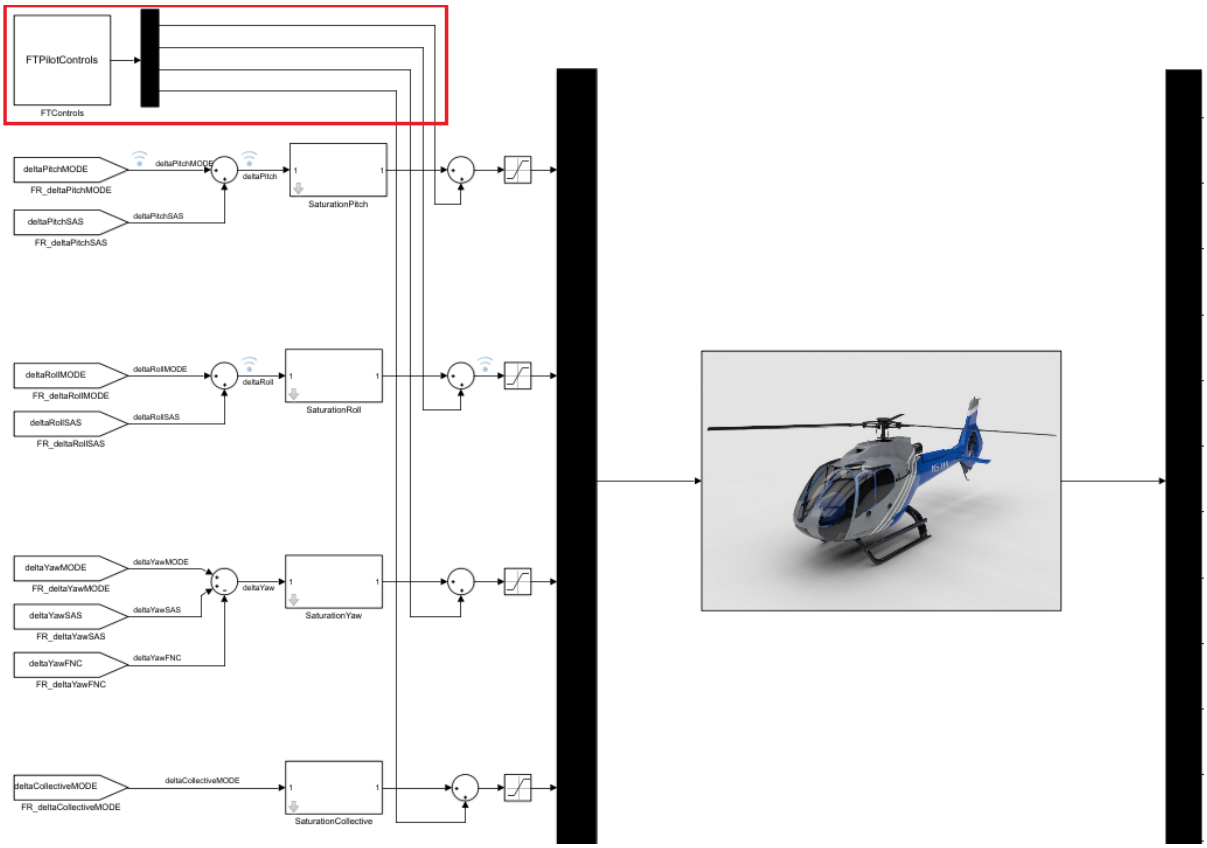

Figure 5.7: AFCS Simulink® Model provided of Pilot Inputs

In the flight test *2.c.(1)*, the measured pilot commands are shown in Figure 5.8. The responses generated from the real helicopter and from the tuned AFCS with just the SAS activated, are shown in Figures 5.9 and 5.10. Both the tolerances imposed from the certification body are respected.

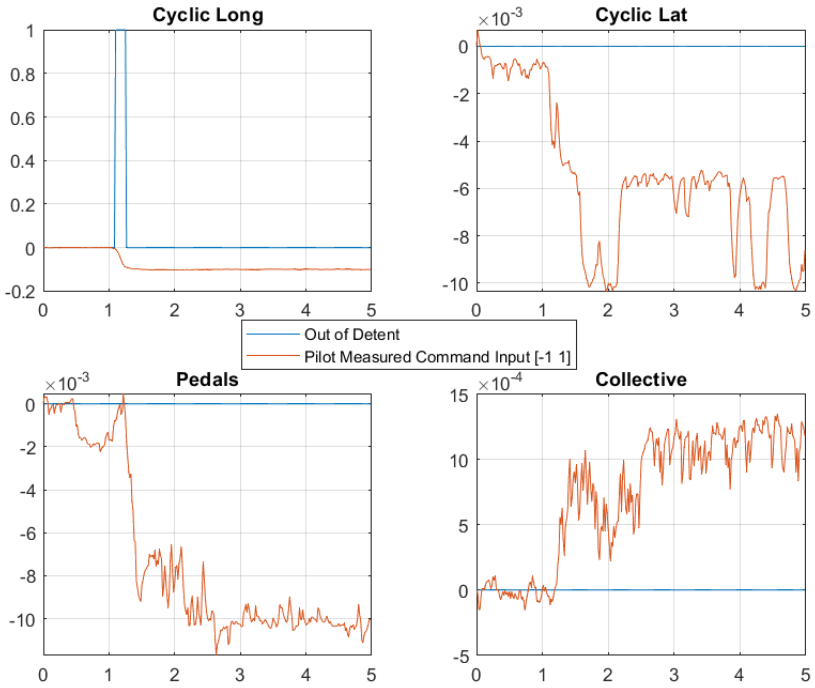

Figure 5.8: *2.c.(1)*: Flight Test Pilot Inputs

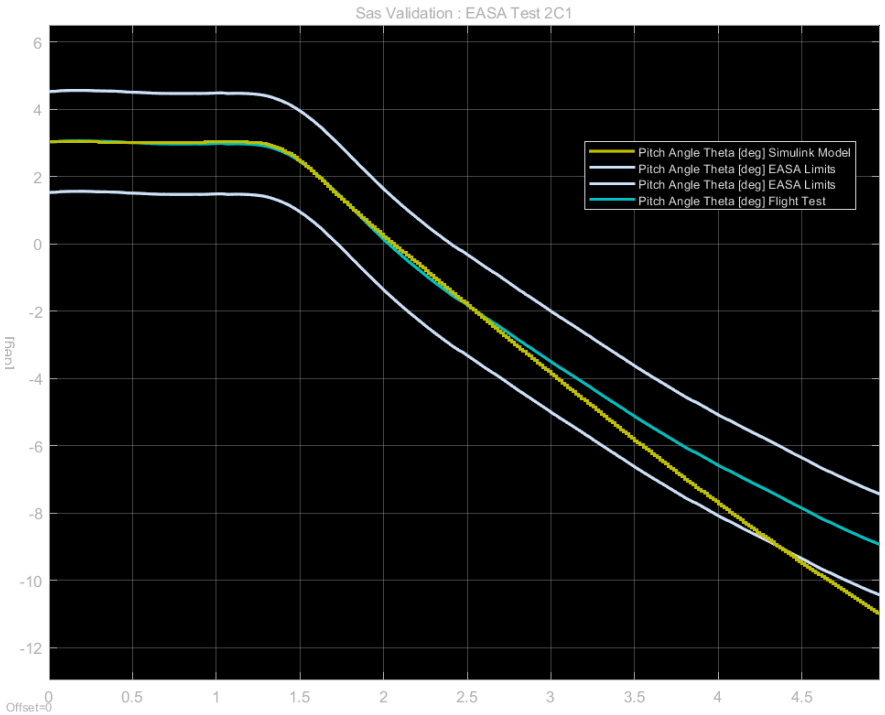

Figure 5.9: *2.c.(1)*: Flight Test and Simulated Pitch Angle Responses with SAS activated

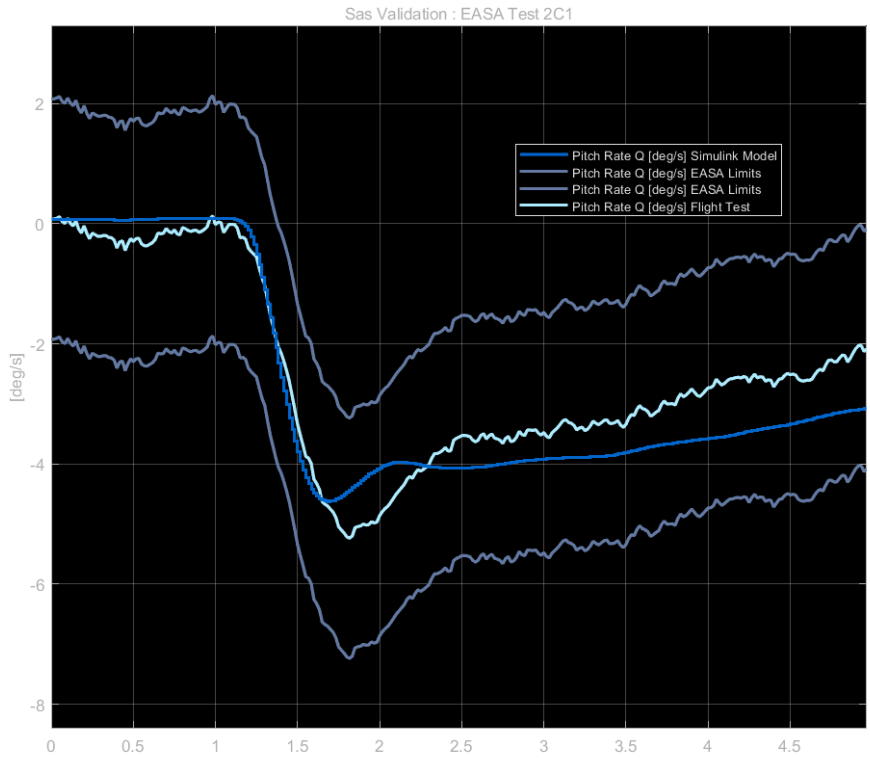

Figure 5.10: *2.c.(1)*: Flight Test and Simulated Pitch Rate Responses with SAS activated

Same reasoning for the the flight test *2.d.(1)(i)*, where the inputs are shown in Figure 5.11, while the responses in Figures 5.12 and 5.13. Also in this case, simulated values are within the imposed limits.

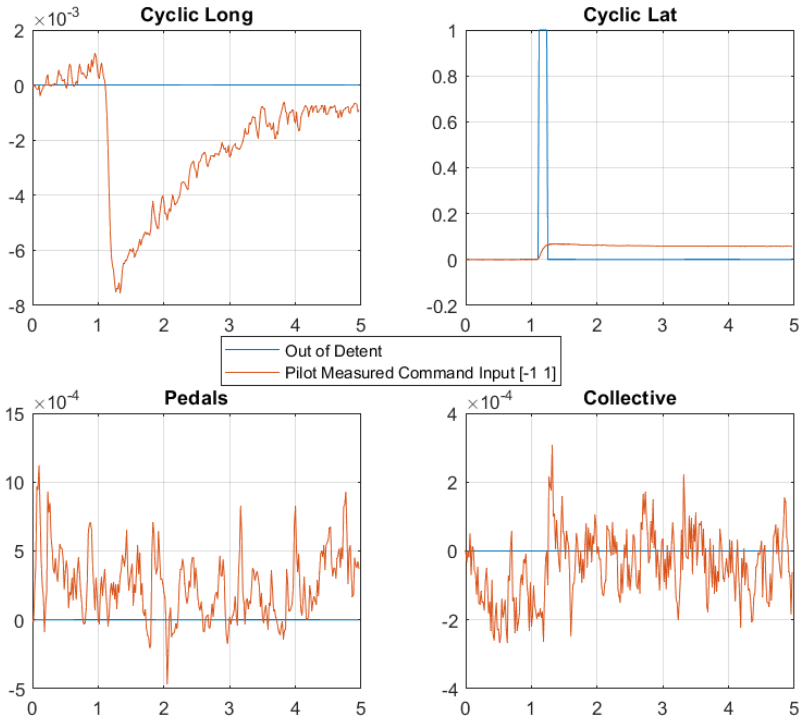

Figure 5.11: *2.d.(1)(i)*: Flight Test Pilot Inputs

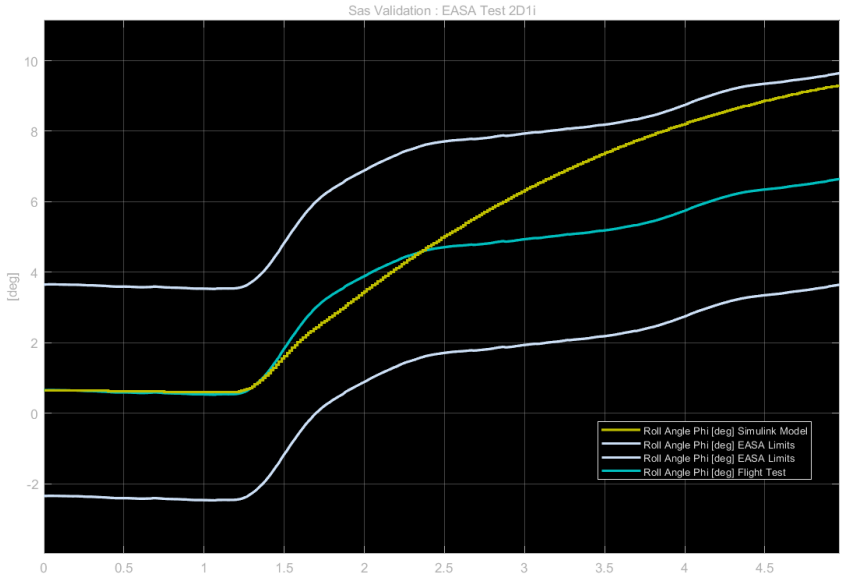

Figure 5.12: *2.d.(1)(i)*: Flight Test and Simulated Bank Angle Responses with SAS activated

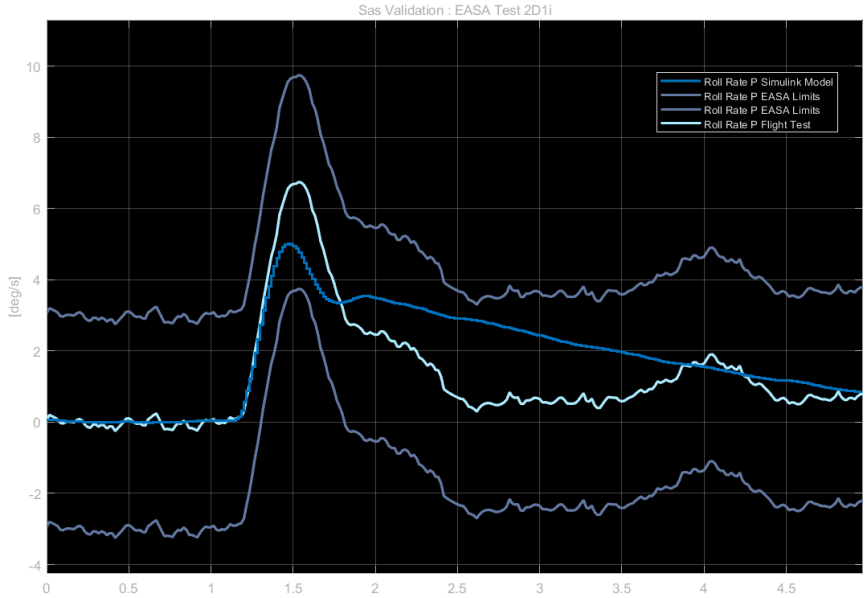

Figure 5.13: *2.d.(1)(i)*: Flight Test and Simulated Roll Rate Responses with SAS activated

5.3 Attitude Hold System and Upper Modes

Finally, the last type of validation analysis conducted proves that the ATT and upper modes tuned in the previous Chapter are capable of producing similar responses to those obtainable with the certified FFS Level D.

For this phase, *TXT e-solutions* provided the possibility to log data directly from the existing FFS Level D of the specific helicopter used to tune the AFCS Simulink® model. The workflow of the validation process can be summarized as follows:

- the first step involved the creation of validation tests having the property of being reproducible in both the FFS and the Simulink® AFCS model. According to the modeling strategy employed to build the AFCS in Simulink®, external inputs may be provided from four sources:

  1. Pressing buttons on the APCP.
  2. Pressing the force trim buttons present either on the cyclic and on the collective sticks.
  3. Moving upward/backward/left/right the beep trim present either on the cyclic and on the collective sticks.
  4. Moving the cyclic, collective and pedals to trigger their respective out of detent sensor.

  Therefore, since the pursuit of those tests is to engage a mode and verify if their functioning is correct, the APCP buttons as well as the beep trim on both the cyclic and collective sticks are used to perform validation tests in the FFS. Although the force trim could have been used to carry out some tests with some mode engaged, this was not feasible due to hardware problems in the data logging from the FFS. Indeed, for each validation test, a *.csv* file is produced; this file containis relevant information such as measured attitudes, rates, AFCS status, mode engaged for each axis, setpoint of each mode, beep trim switching, etc. Therefore, several real-time simulations in operating flight conditions similar to the one used to obtain the linearized model, are run. In each of those simulations, a different mode is engaged and is subject to beep trim excitements.

- in the second step a Matlab® script is created to store data in the *Workspace* from each *.csv* test file. This file is essential to reproduce the same test performed in the FFS in the Simulink® environment. In this regard, the model used for the AFCS tuning and shown in Figure 4.6, is also employed for this validation phase. Hence,

each block of this model is updated with the optimal gains found during the tuning process.

- The third step involves the preparation of a Matlab® script in which each test is separately conducted, updating each time the Simulink® model with the specific test APCP buttons time histories as well as with the beep trim time histories. In addition, the *StopTime* of each simulation is set equal to the duration of each specific FFS test. Obviously, those data are taken from the Matlab® script created in the previous step. Finally, each simulation can be run and results can be plotted and compared to the ones obtained with the FFS.

Before showing the results it is worth mention the limitations of this analysis:

1. The FFS employs a nonlinear model of the helicopter while in the Simulink® model a linearized version is utilized. Hence, executing tests in these models provide almost reliable results only in the neighborhood of the trim condition of the linear model.
2. The tuning executed in the previous Chapter has been set with goals that were not the same used to build the FFS AFCS. Moreover, requirements imposed in terms of *Stability Margins* were set without any criteria with respect to the responses of the FFS, since the pursuit was just to show a possible tuning methodology and not to explain how to replicate a specific tuning process.

Despite the limitations, for most of the modes the obtained results are consistent with or above expectations. In the following some considerations about the outcomes of this validation.

- Modes that can be engaged on different axes effectively transition the control action from one axis to another and continue to generate effective responses, as illustrated by the test of the ALT in Figure 5.18. During this experiment, the *Ias* was intentionally reduced providing two pitch up commands in the helicopter at $t \approx 40s$ and $t \approx 90s$. Consequently, the *Ias* dropped below the threshold for disengagement of the ALT on the pitch axis and the engagement of the ALT on the collective axis.
- The setpoint imposed by beep trim in the FFS perfectly matches the one obtained in the Simulink® model. The dotted lines in each figure shown below are always overlapping, which means that this function is correctly modeled.
- The Simulink® model's responses well capture the specific mode's task and often exhibits better performance in terms of settling time and overshoot compared to the FFS responses. For instance, this is the case for the IAS, as depicted in Figure 5.17. Additionally, another notable result is achieved concerning the TC, as shown

in Figure 5.16. This system can eliminate lateral accelerations more efficiently while maintaining the proper operation of the other mode excited in the same test, i.e. the ATT Roll, shown in Figure 5.15. Therefore, these results demonstrate the accuracy of both the blocks modeling of the modes and the auto-tuning methodology employed to select the gains for their controllers.

ATT Pitch

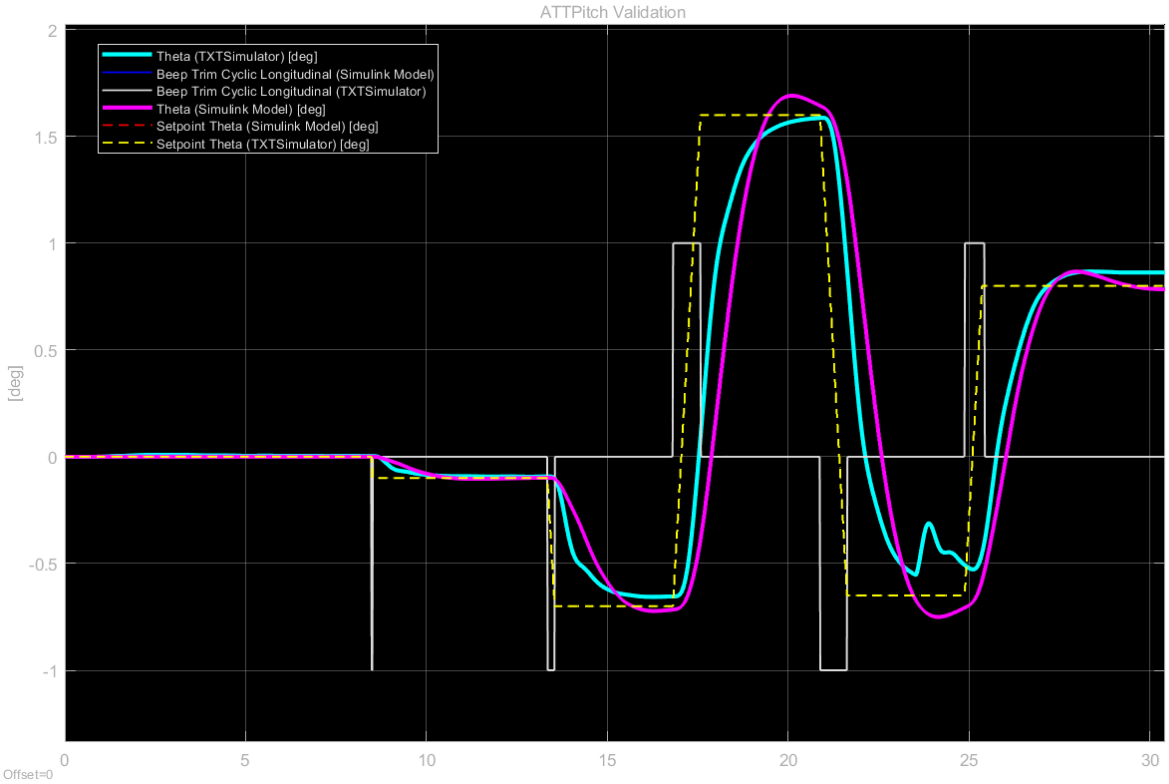

Figure 5.14: Validation of the ATT Pitch

ATT Roll and TC (Same Test)

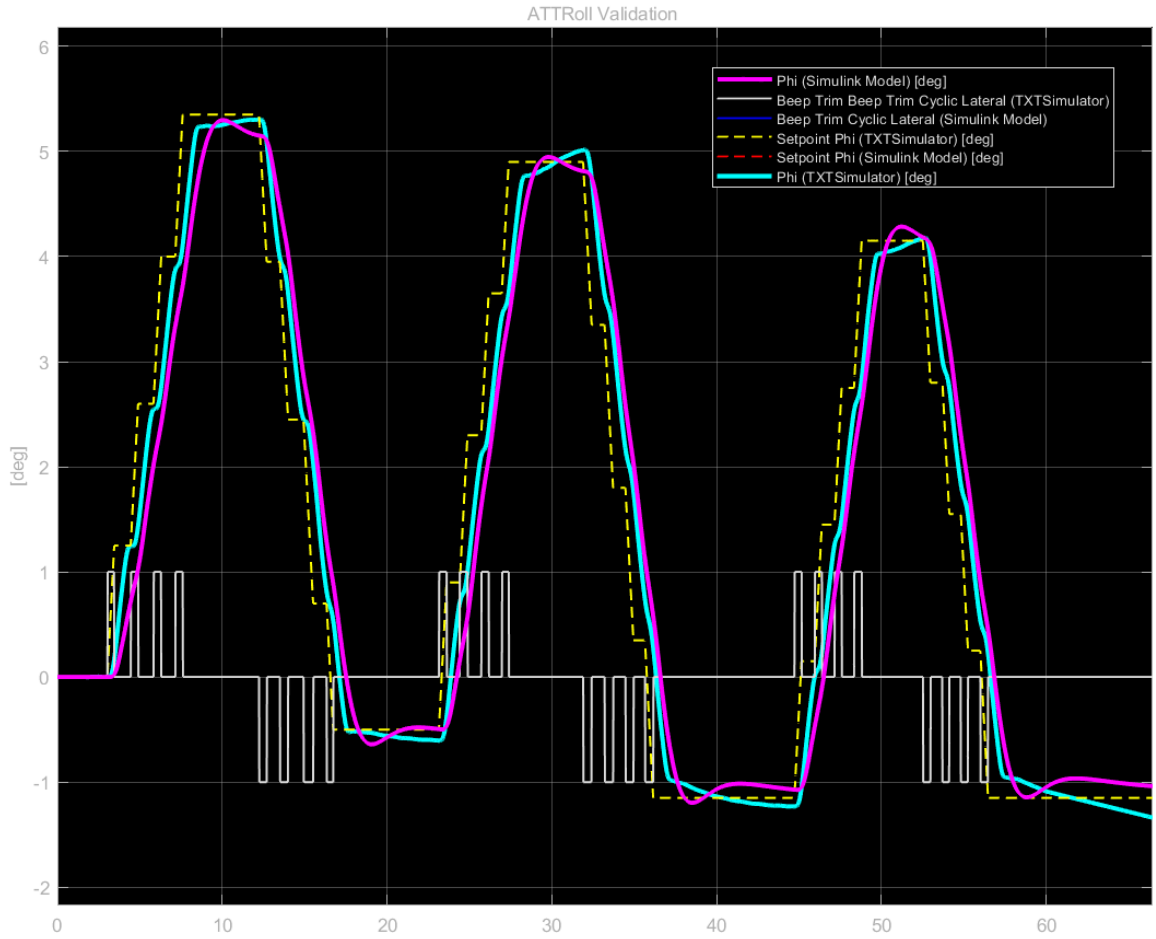

Figure 5.15: Validation of the ATT Roll

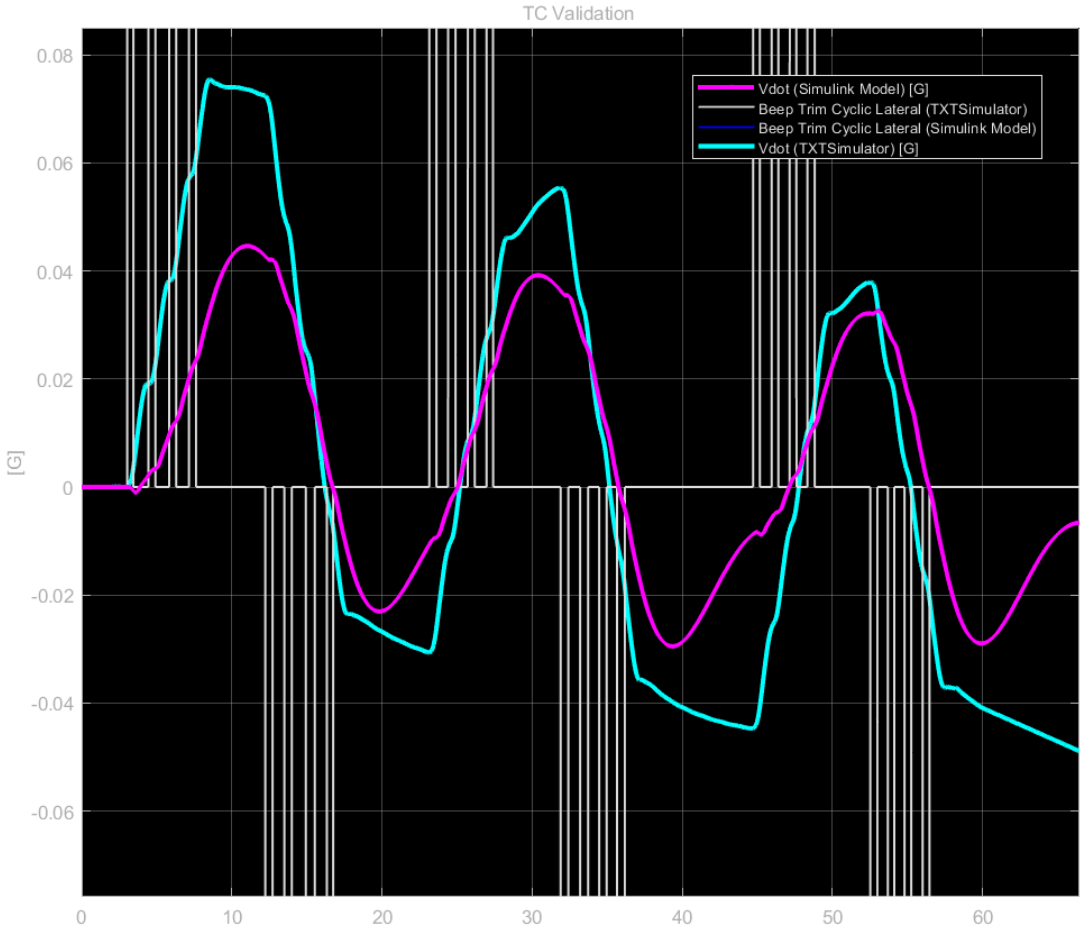

Figure 5.16: Validation of the TC

IAS

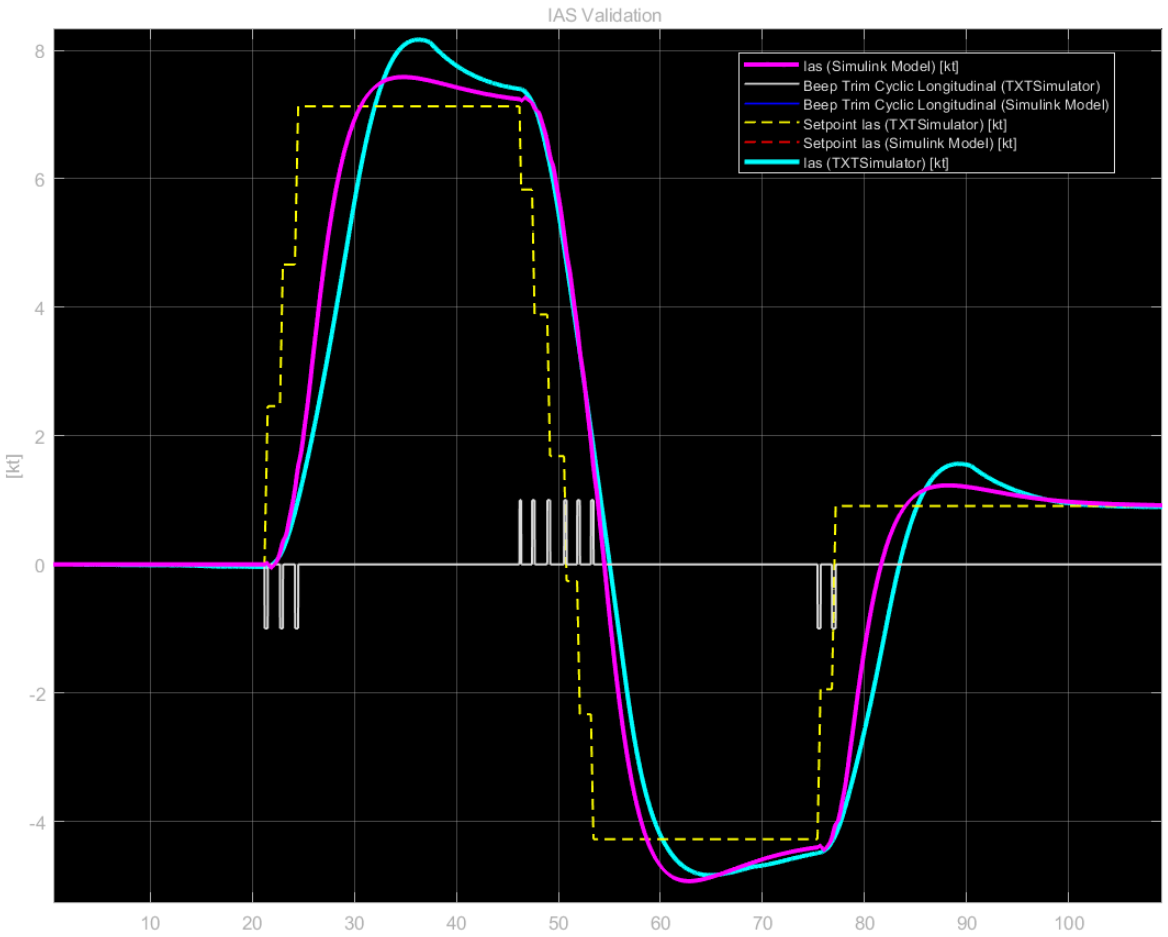

Figure 5.17: Validation of the IAS

ALT

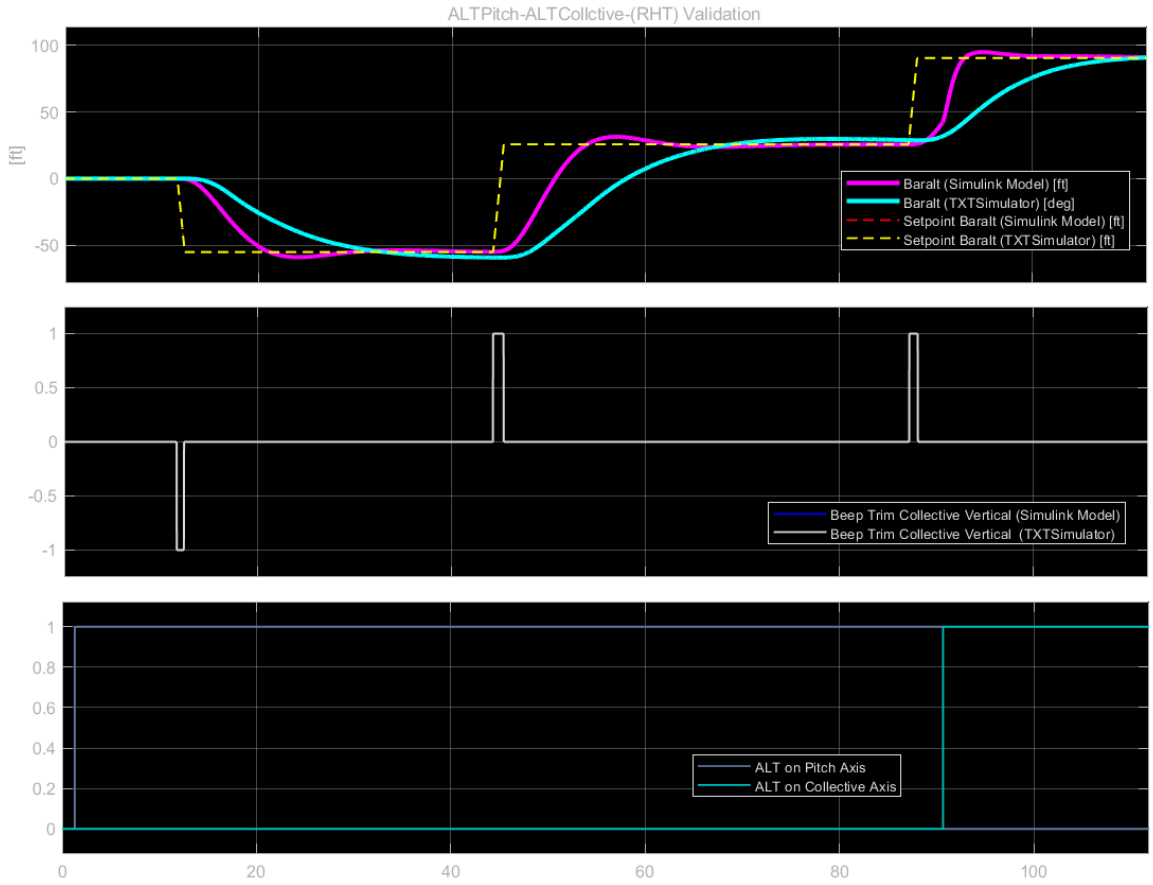

Figure 5.18: Validation of the ALT Collective and ALT Pitch

HDG Roll

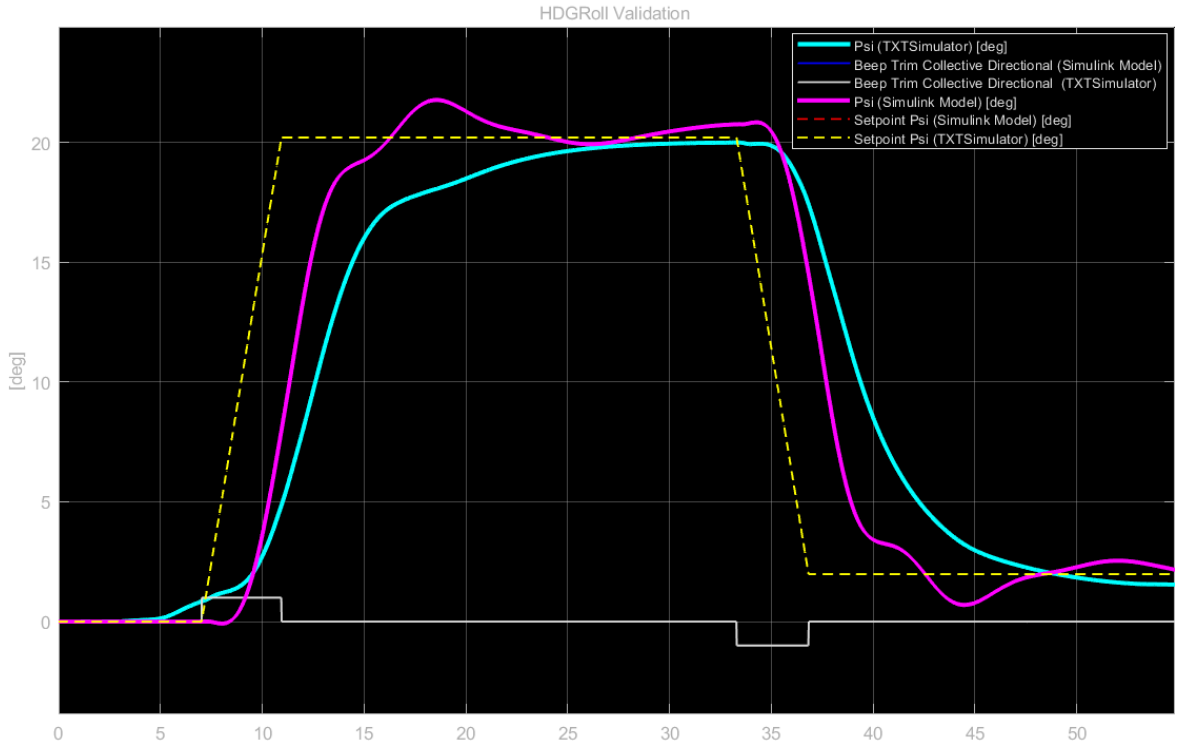

Figure 5.19: Validation of the HDG Roll

Conclusions

In conclusion, the aim of this work was to develop, through a model-based approach, a generic helicopter automatic flight control system that guarantee certain requisites with respect to the results obtained in the previous work. Those requirements, which have been all satisfied, were:

1. To fulfill all the goals already achieved in the previous research, i.e. the development of the lower modes of the generic autopilot in Simulink® enviroment, the development of a tuning methodology for those controllers and the suitability of the model for code generation purposes.
2. To enhance the level of complexity of the autopilot, introducing the most important upper modes, their control logics and a tuning methodology for the controllers of those upper modes.
3. To increase the modularity of each Simulink® block, identifying and segregating repetitive system components within the autopilot.
4. To increase the level of generality of each Simulink® block, seeking the highest levels of customisation.
5. To reorganise the autopilot structure by developing a framework that facilitates the inclusion of new components and functions in future AFCS developments.

The validation phase carried out in the final Chapter has shown that the modelling of the whole autopilot and the tuning of its control system were able to achieve remarkable results even in comparison with a certified full flight simulator.

Therefore, the generic autopilot architecture developed in this thesis can be used as a starting point for future research in this topic, and the knowledge gained from this study can be applied to develop new components and subsystems in future developments of the current autopilot.

To this purpose, the structural architecture developed for the model assembly (Section 4.2) could pave the way for an easy implementation of new upper modes, such as *Vertical*

Speed Mode (VS) or *Go Around Mode (GA)* and most importantly to the introduction of a new category of modes, namely the *Flight Director Modes* such as *En-Route Navigation Mode (NAV)*, *Localizer Mode (LOC)*, *Glideslope Mode (GS)*, and others.

Bibliography

[1] K. J. Åström and R. M. Murray. *Feedback Systems. An Introduction for Scientists and Engineers*. Princeton University Press, 2020.

[2] J. C. Butcher. *Numerical methods for ordinary differential equations*. John Wiley & Sons, 2016.

[3] EASA. *CS-FSTD(H) (Initial issue)*. 2020.

[4] A. B. et al. *Bramwell's Helicopter Dynamics*. Butterworth-Heinemann, 2001.

[5] G.D.Padfield. *Helicopter Flight Dynamics*. John Wiley & Sons, 2018.

[6] MathWorks®. *Automated Tuning Workflow*. `https://it.mathworks.com/help/control/ug/automated-tuning-workflows.html`, .

[7] MathWorks®. *Set Breakpoints to Debug Charts*. `https://it.mathworks.com/help/stateflow/ug/set-breakpoints-to-debug-charts.html`, .

[8] MathWorks®. *Animate Stateflow Charts*. `https://it.mathworks.com/help/stateflow/ug/animate-stateflow-charts.html`, .

[9] MathWorks®. *Step Tracking Goal*. `https://it.mathworks.com/help/slcontrol/ug/step-tracking-goal.html`, .

[10] MathWorks®. *Inspect and Modify Data and Messages While Debugging*. `https://it.mathworks.com/help/stateflow/ug/watching-data-values-during-simulation.html`, .

[11] MathWorks®. *Default Parameter Behavior*. `https://it.mathworks.com/help/rtw/ref/defaultparameterbehavior.html`, .

[12] MathWorks®. *Tune a Control System Using Control System Tuner*. `https://it.mathworks.com/help/control/ug/tuning-control-systems-with-control-system-tuner.html`, .

[13] MathWorks®. *Embedded Coder*. https://it.mathworks.com/help/ecoder/ref/embeddedcoder-app.html, .

[14] MathWorks®. *Add Libraries to Library Browser*. https://it.mathworks.com/help/simulink/ug/adding-libraries-to-the-library-browser.html, .

[15] MathWorks®. *Mask Editor Overview*. https://it.mathworks.com/help/simulink/gui/mask-editor-overview.html, .

[16] MathWorks®. *Model Parameter Configuration Dialog Box*. https://it.mathworks.com/help/simulink/gui/model-parameter-configuration-dialog-box.html, .

[17] MathWorks®. *Creating Flow Graphs With The Pattern Wizard*. https://it.mathworks.com/help/stateflow/ug/creating-flow-graphs-with-the-pattern-wizard.html, .

[18] MathWorks®. *Poles Goal*. https://it.mathworks.com/help/slcontrol/ug/poles-goal.html, .

[19] MathWorks®. *Creating Test Cases to Verify Your Simulink Design Using the Signal Builder Block*. https://it.mathworks.com/company/newsletters/articles/creating-test-cases-to-verify-your-simulink-design-using-the-signal-builder-block.html, .

[20] MathWorks®. *Stability Margins in Control System Tuning*. https://it.mathworks.com/help/control/ug/interpreting-stability-margin-plots.html, .

[21] MathWorks®. *Definizione del comportamento del grafico tramite l'utilizzo delle azioni*. https://it.mathworks.com/help/stateflow/gs/actions.html, .

[22] MathWorks®. *Synchronize Model Components by Broadcasting Events*. https://it.mathworks.com/help/stateflow/ug/how-events-work-in-stateflow-charts.html, .

[23] MathWorks®. *Set Data Properties*. https://it.mathworks.com/help/stateflow/ug/set-data-properties-1.html, .

[24] MathWorks®. *Define Exclusive and Parallel Modes by Using State Decomposition*. https://it.mathworks.com/help/stateflow/ug/state-decomposition.html, .

[25] MathWorks®. *Transition Between Operating Modes*. https://it.mathworks.com/help/stateflow/ug/transitions.html, .

[26] MathWorks®. *Choose Storage Class for Controlling Data Representation in Generated Code*. https://it.mathworks.com/help/rtw/ug/choose-a-built-in-storage-class-for-controlling-data-representation-in-the-generated-code.html, .

[27] MathWorks®. *Step Rejection Goal*. https://it.mathworks.com/help/slcontrol/ug/step-rejection-goal.html, .

[28] MathWorks®. *Difference Between Fixed-Structure Tuning and Traditional H-Infinity Synthesis*. https://it.mathworks.com/help/robust/gs/difference-between-fixed-structure-tuning-and-traditional-h-infinity-synthesis.html, .

[29] MathWorks®. *Manual and Automatic PID Tuning Methods*. https://www.youtube.com/watch?v=qj8vTO1eIHo&list=PLn8PRpmsu08pQBgjxYFXSsODEF3Jqmm-y&index=6, 2019.

[30] N. S. Nisey. *Control System Engineering, Eighth Edition*. John Wiley & Sons, 2019.

[31] E. Pallett. *Automatic Flight Control*. Blackwwell Publishing, 1993.

[32] G. Romagnoli. *Model-Based Design of a Generic Autopilot System for Helicopter Flight Simulators in Simulink*. 2021.

[33] T. S.Alderete. *Simulator Aero Model Implementation*. 1997.

[34] R. W.Prouty and H. Jr. *Helicopter Control Systems: A History*. Journal of guidance, control and dynamics, 2003.

A | Appendix: AFCS Masks Configuration

A.1 Upper Modes Subsystem

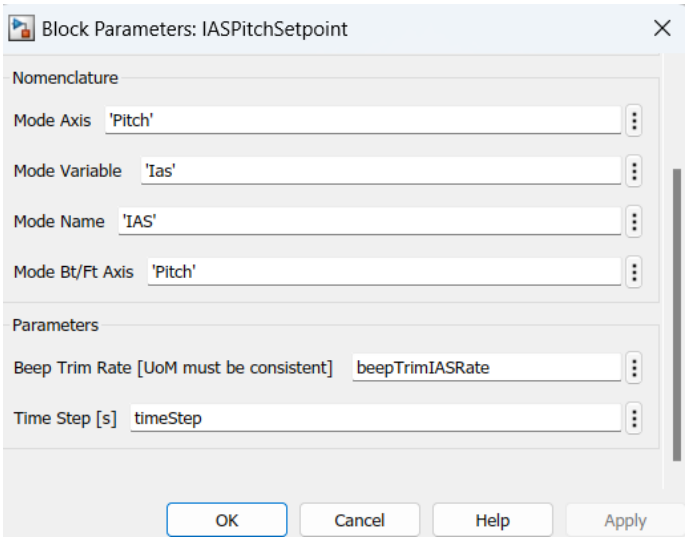

Figure A.1: *Mode Setpoint* custom block configured for IAS Mode on Pitch Axis

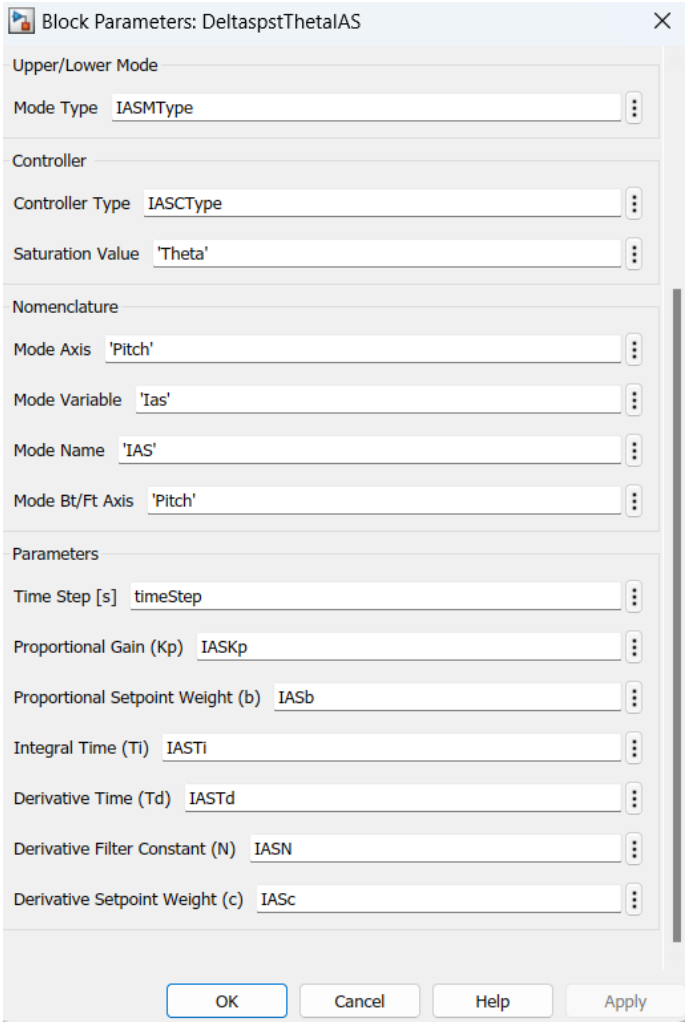

Figure A.2: *Mode δCommand/δSetpoint* custom block configured as *Mode δSetpoint* block for IAS Mode on Pitch Axis

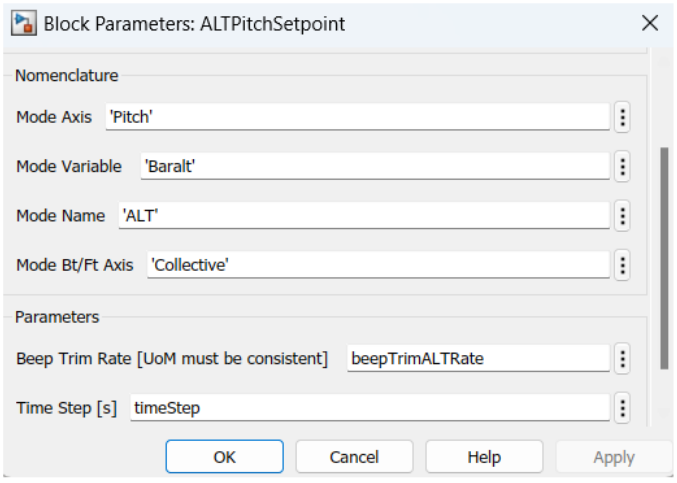

Figure A.3: *Mode Setpoint* custom block configured for ALT Mode on Pitch Axis

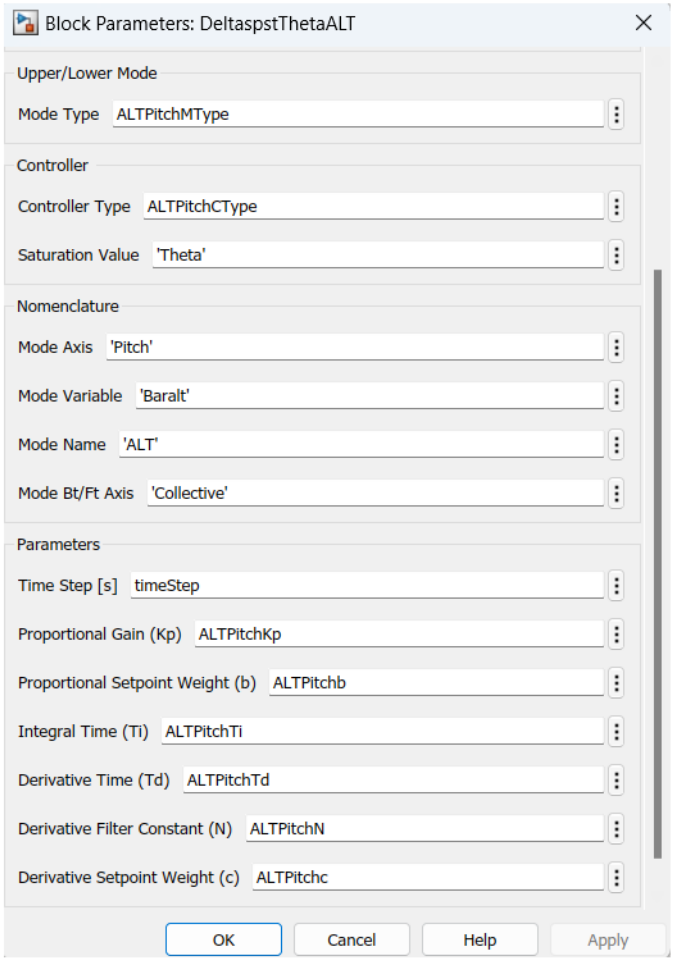

Figure A.4: *Mode δCommand/δSetpoint* custom block configured as *Mode δSetpoint* block for ALT Mode on Pitch Axis

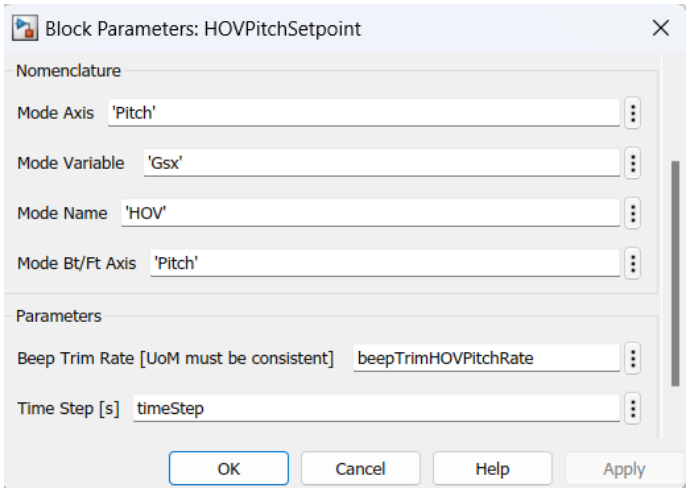

Figure A.5: *Mode Setpoint* custom block configured for HOV Mode on Pitch Axis

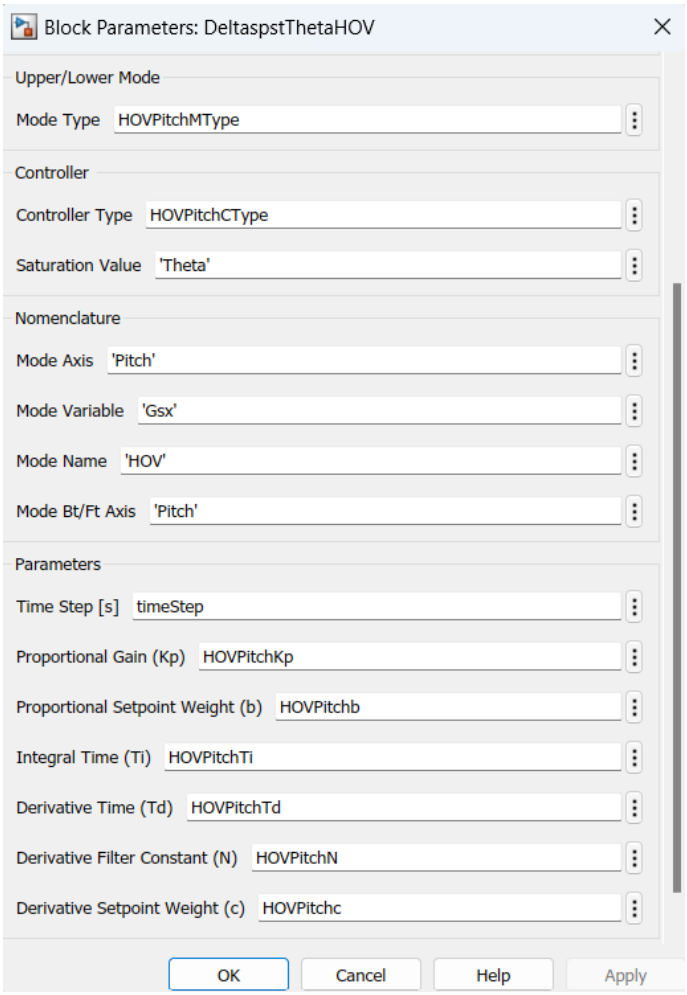

Figure A.6: *Mode δCommand/δSetpoint* custom block configured as *Mode δSetpoint* block for HOV Mode on Pitch Axis

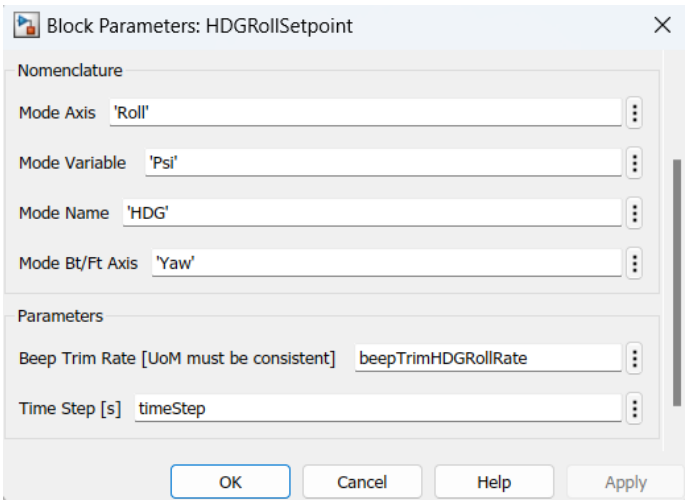

Figure A.7: *Mode Setpoint* custom block configured for HDG Mode on Roll Axis

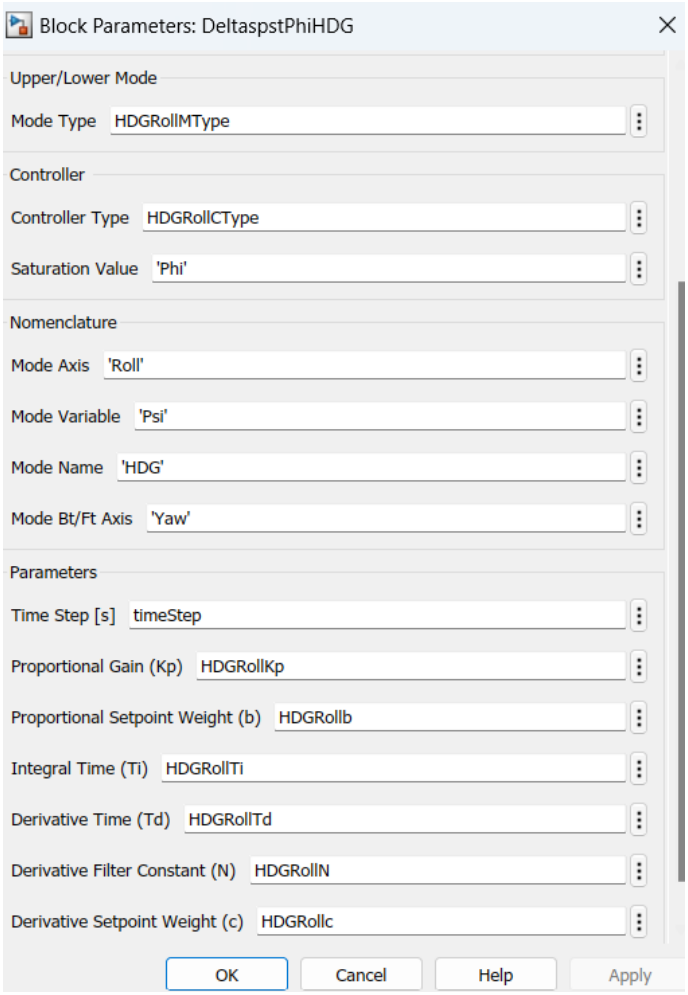

Figure A.8: *Mode δCommand/δSetpoint* custom block configured as *Mode δSetpoint* block for HDG Mode on Roll Axis

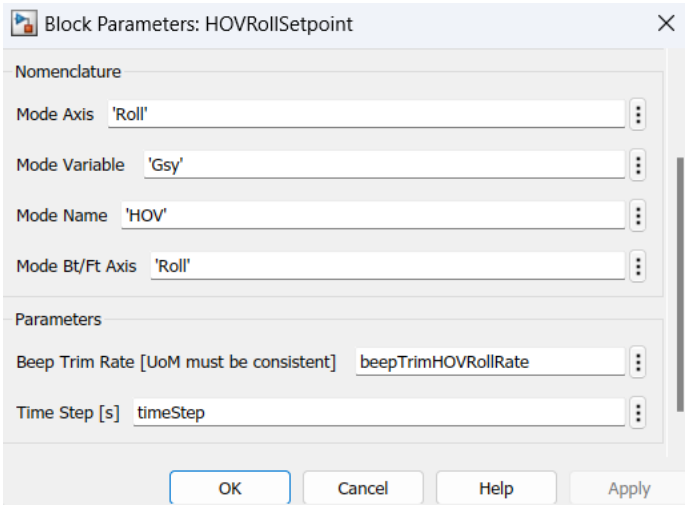

Figure A.9: *Mode Setpoint* custom block configured for HOV Mode on Roll Axis

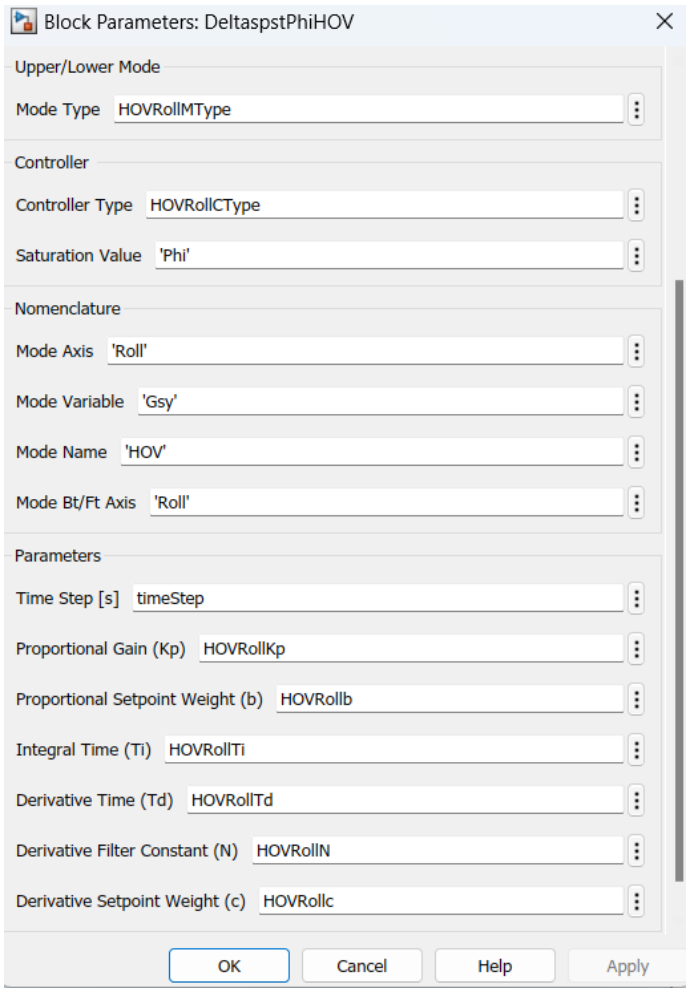

Figure A.10: *Mode δCommand/δSetpoint* custom block configured as *Mode δSetpoint* block for HOV Mode on Roll Axis

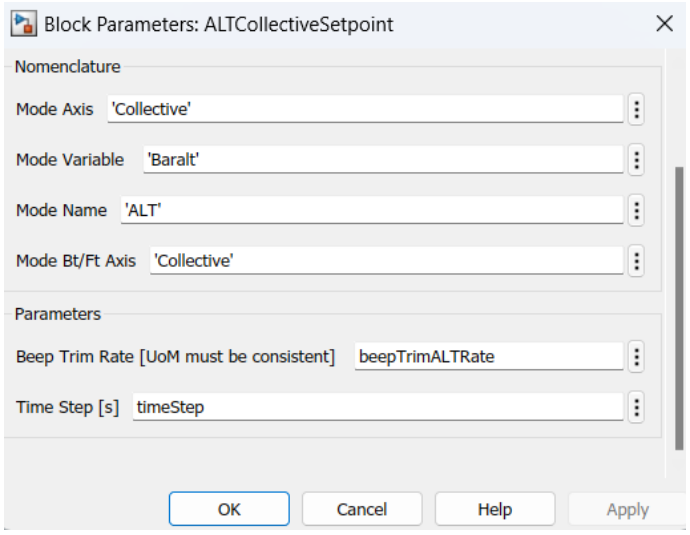

Figure A.11: *Mode Setpoint* custom block configured for ALT Mode on Collective Axis

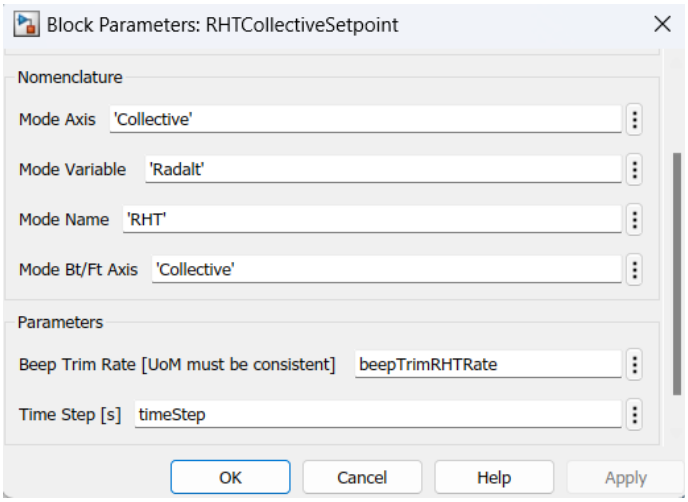

Figure A.12: *Mode Setpoint* custom block configured for RHT Mode on Collective Axis

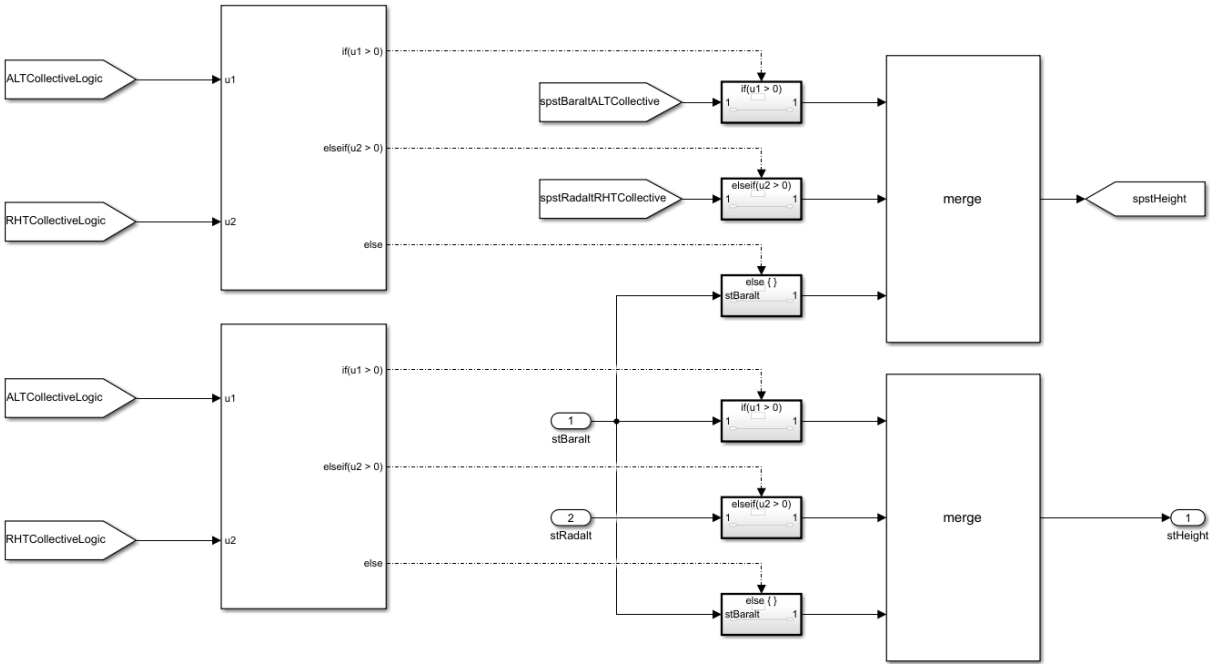

Figure A.13: Structure of the the subsystem that enables the State Transitions (both Setpoint and Current State Transitions) among Collective Axis Modes

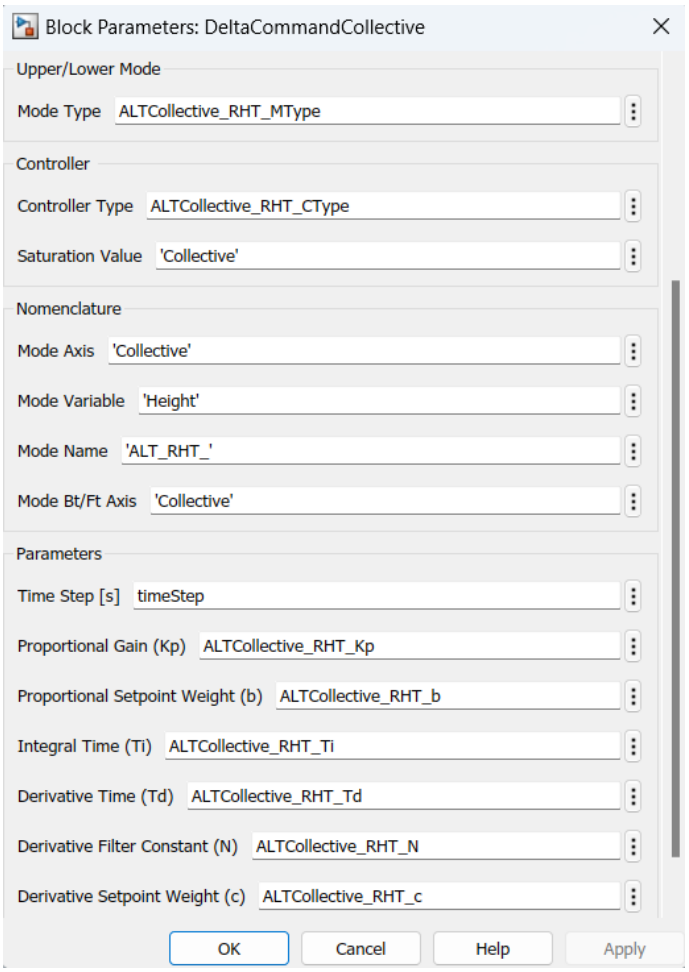

Figure A.14: *Mode δCommand/δSetpoint* custom block configured as *Mode δCommand* block for Collective Axis Modes

A.2 Lower Modes Subsystem

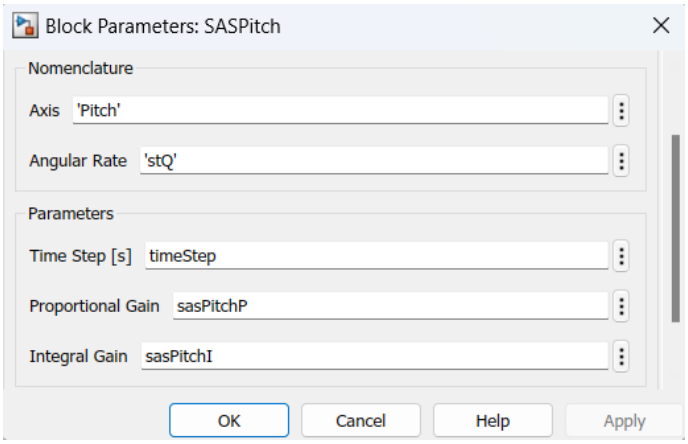

Figure A.15: *SAS* custom block configured for the Pitch Axis

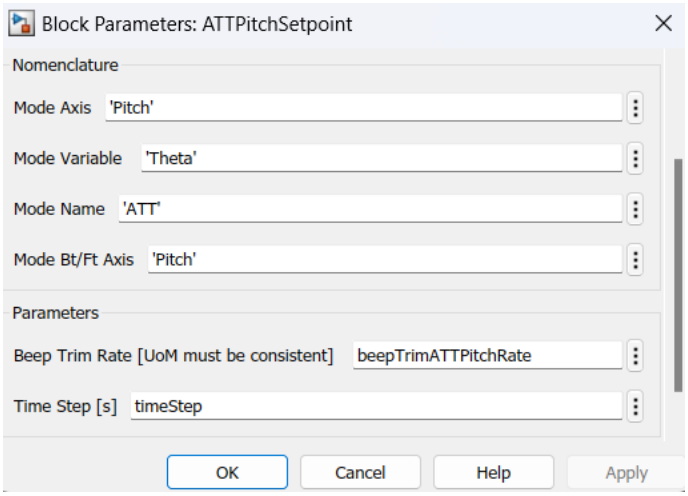

Figure A.16: *Mode Setpoint* custom block configured for ATT Mode on Pitch Axis

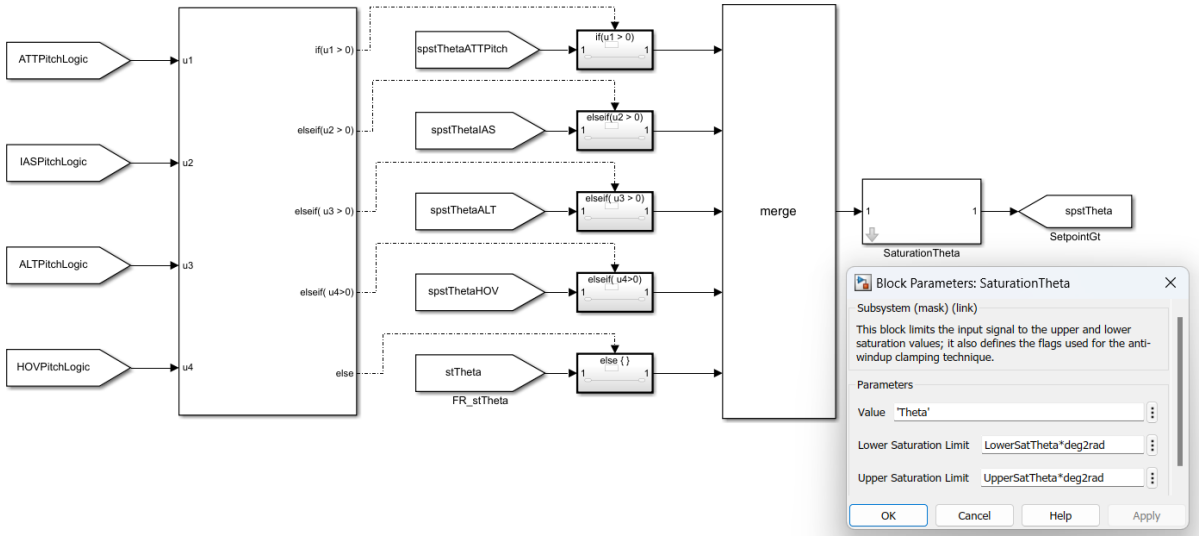

Figure A.17: Structure of the the subsystem that enables the Setpoint Transition among Pitch Axis Modes

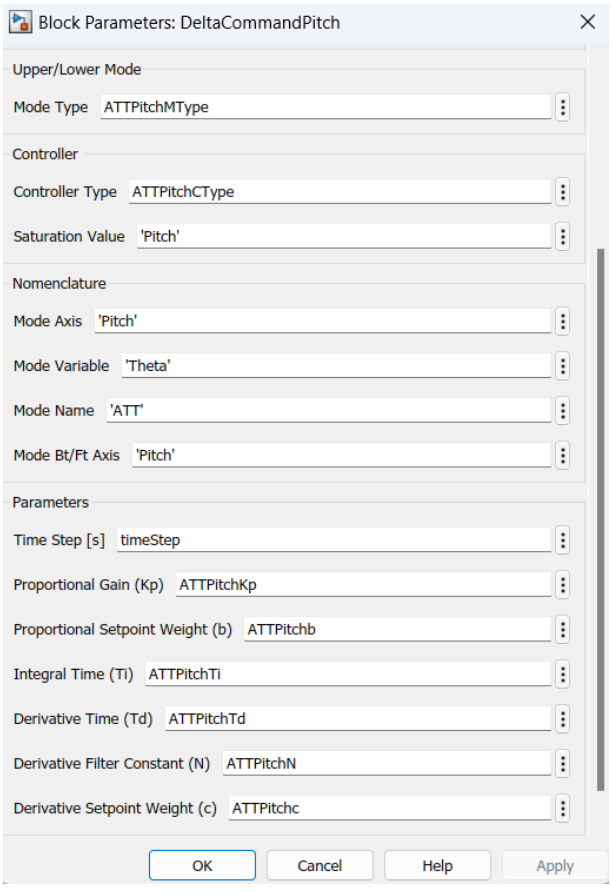

Figure A.18: *Mode δCommand/δSetpoint* custom block configured as *Mode δCommand* block for Pitch Axis Modes

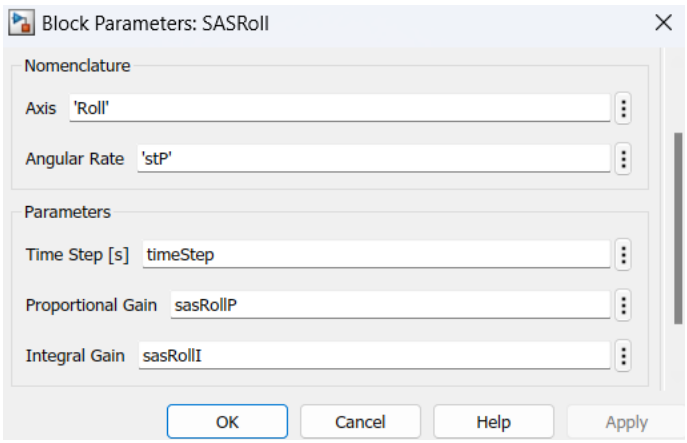

Figure A.19: *SAS* custom block configured for the Roll Axis

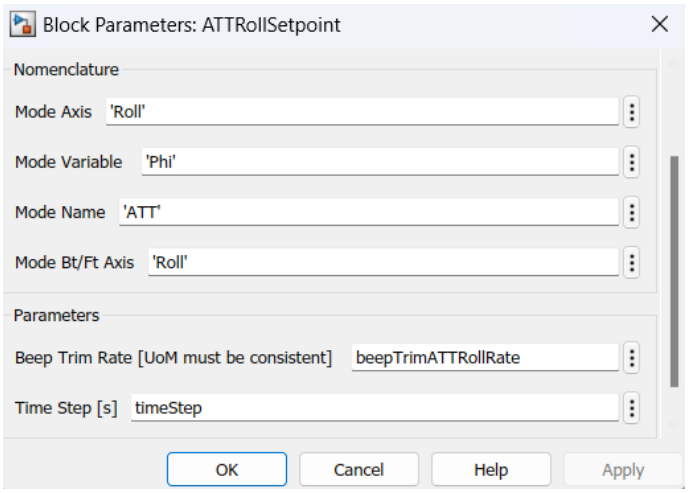

Figure A.20: *Mode Setpoint* custom block configured for ATT Mode on Roll Axis

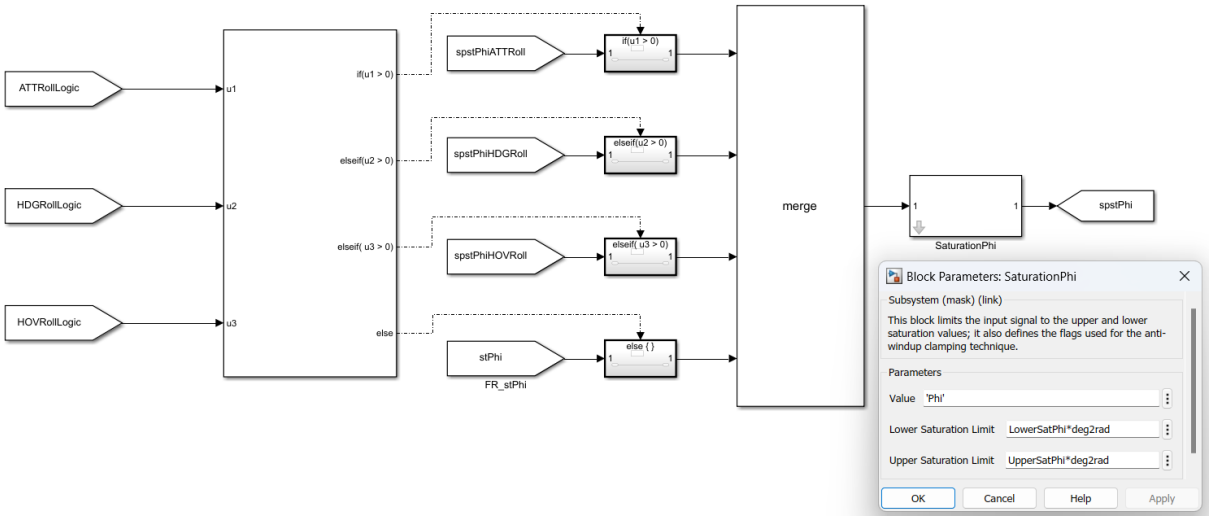

Figure A.21: Structure of the the subsystem that enables the Setpoint Transition among Roll Axis Modes

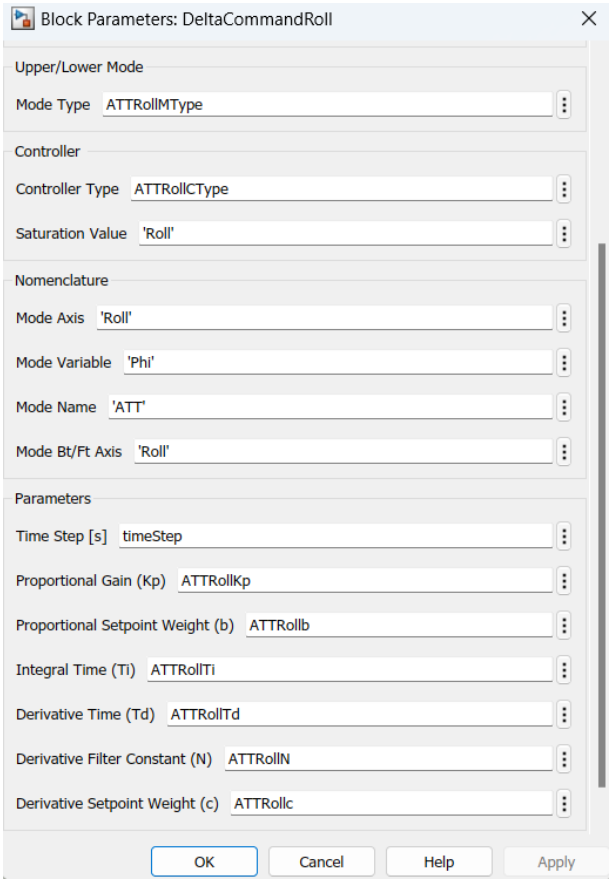

Figure A.22: *Mode δCommand/δSetpoint* custom block configured as *Mode δCommand* block for Roll Axis Modes

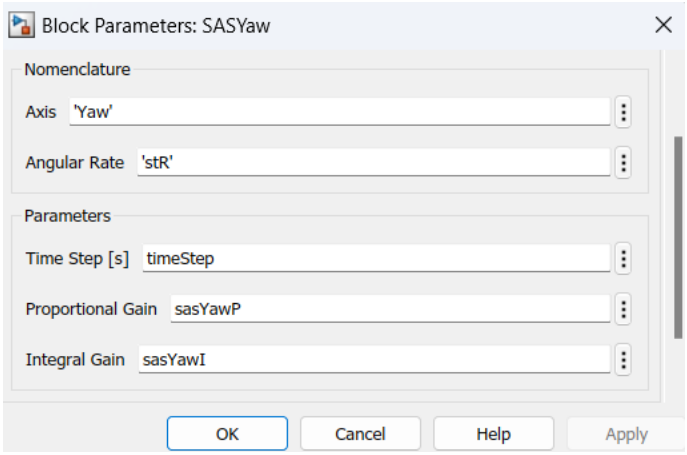

Figure A.23: *SAS* custom block configured for the Yaw Axis

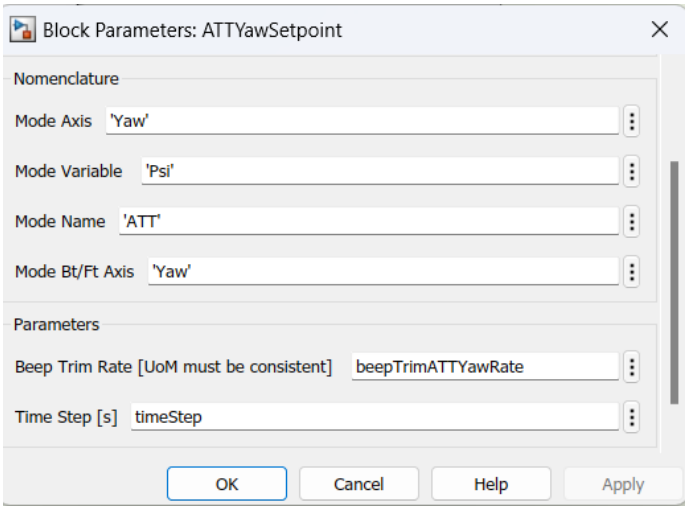

Figure A.24: *Mode Setpoint* custom block configured for ATT Mode on Yaw Axis

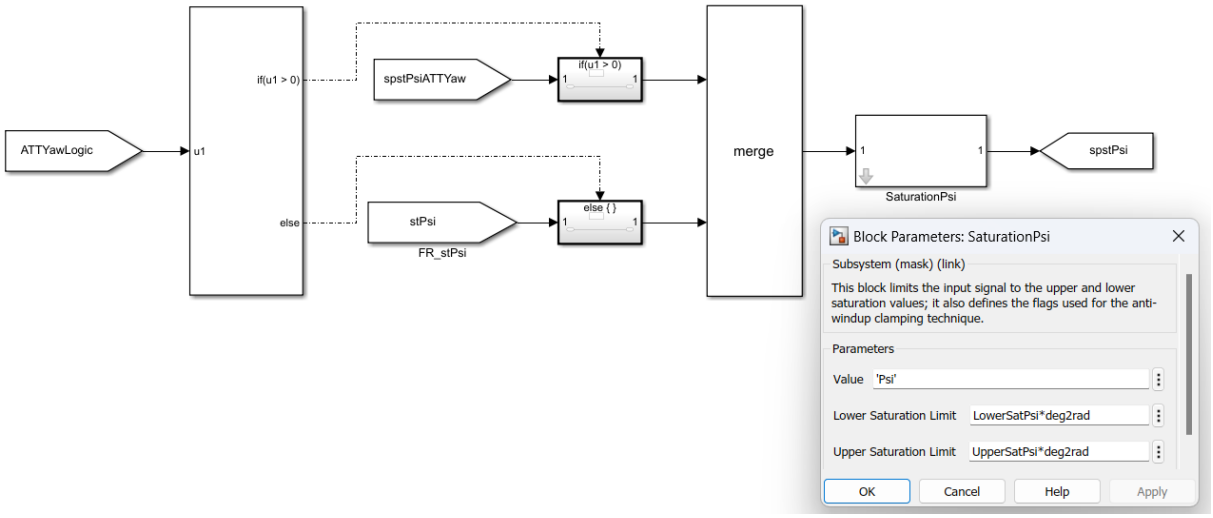

Figure A.25: Structure of the the subsystem that enables the Setpoint Transition among Yaw Axis Modes

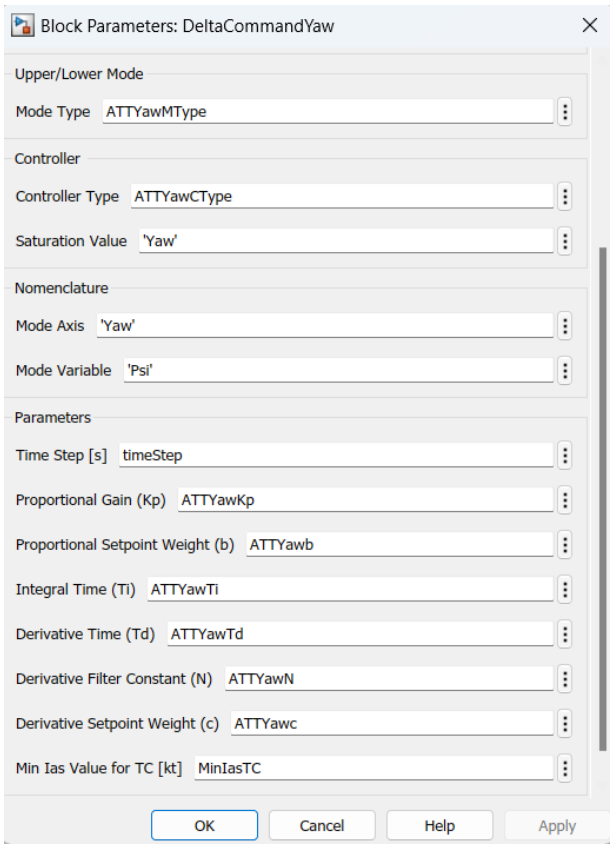

Figure A.26: *Mode δCommand/δSetpoint* custom block configured as *Mode δCommand* block for Yaw Axis Modes

A.3 Functionalities Subsystem

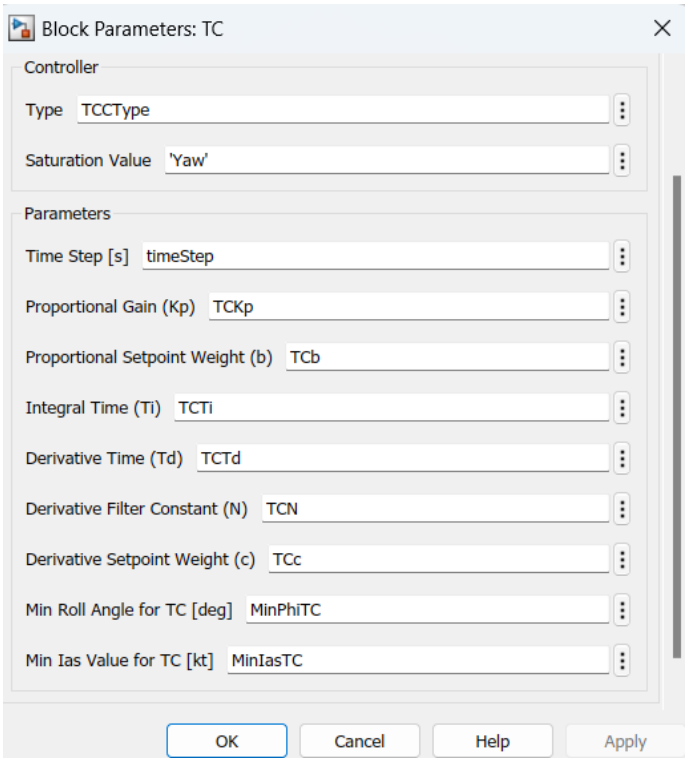

Figure A.27: *TC* custom block configuration

A.4 Saturation of the Actuators

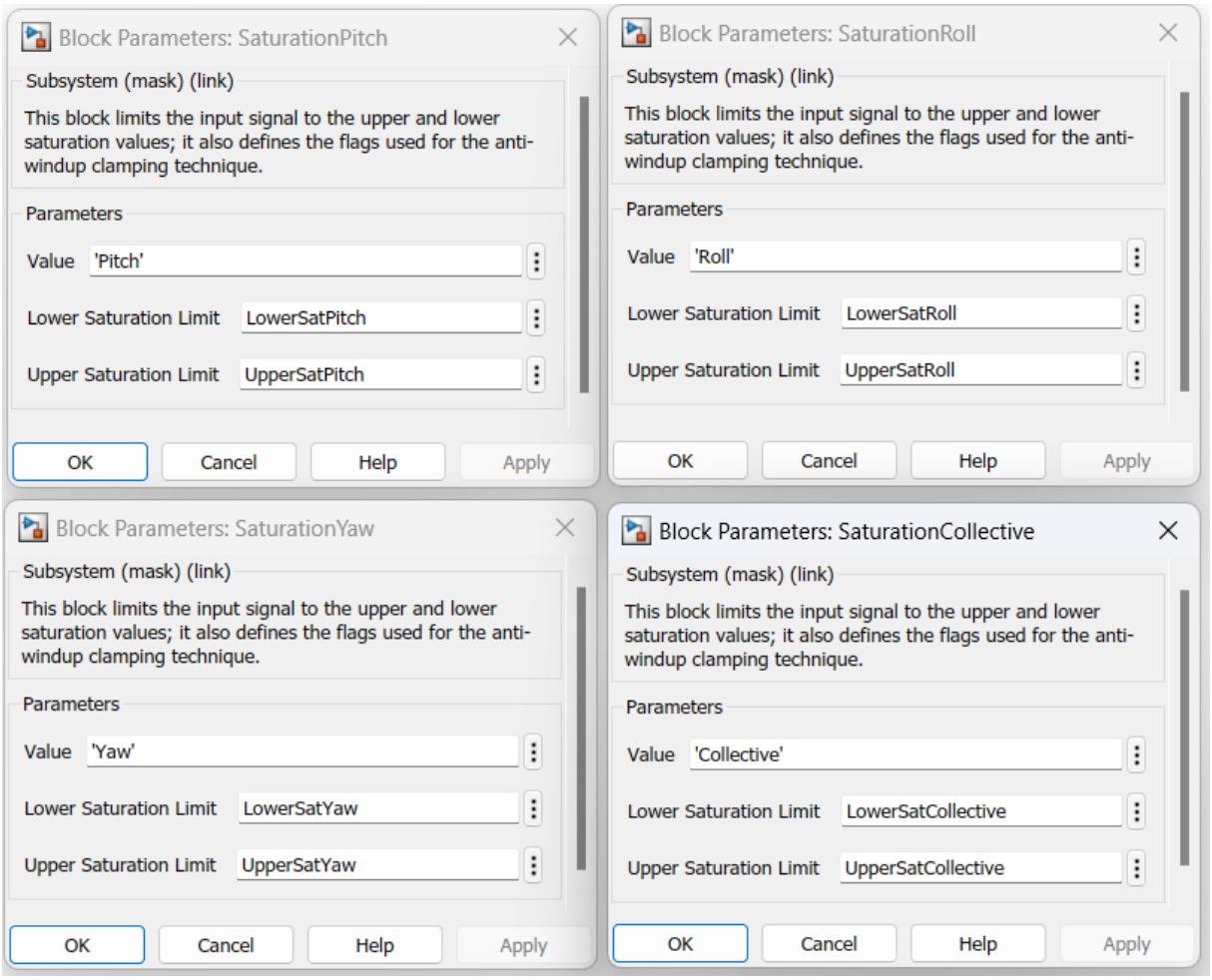

Figure A.28: *Saturation* custom block configured for the actuator of each helicopter axis

B | Appendix: Matlab Scripts

B.1 Tuning Script

```
clear all ;
close all ;
clc ;

%% HELICOPTER LINEAR DYNAMICS

% Import stateSpace matrices and trim conditions
[stateSpace, userData] = txtImportScript('
   Linearized_68kt_5926ft.tab') ;
MatA = stateSpace.A ; % [39x39]
MatB = stateSpace.B ; % [39x4]
MatC = stateSpace.C ; % [12x39]
MatD = stateSpace.D ; % [12x4]

% Assign output initial condition for tuning simulation (
   needed only in the tuning script for mode logics
   management and results verification)
IC_Q      = userData.Output.InitValue{1}(1); %[rad/s]
IC_Theta  = userData.Output.InitValue{2}(1) ; %[rad]
IC_P      = userData.Output.InitValue{7}(1) ; %[rad/s]
IC_R      = userData.Output.InitValue{8}(1) ; %[rad/s]
IC_Phi    = userData.Output.InitValue{9}(1) ; %[rad]
IC_Psi    = userData.Output.InitValue{10}(1) ; %[rad]
IC_Ny     = userData.Output.InitValue{16}(1) ; %[G]
IC_Radalt = userData.Output.InitValue{22}(1) ; %[m]
IC_Gsx    = userData.Output.InitValue{29}(1) ; %[m/s]
IC_Gsy    = userData.Output.InitValue{30}(1) ; %[m/s]
```

```
IC_Ias    = userData.Output.InitValue{31}(1) ; %[m/s]
IC_Baralt = userData.Output.InitValue{32}(1) ; %[m]

% Dynamical system poles-zeros
[poles,zeros] = pzmap(ss(MatA, MatB, MatC, MatD)) ;
figure
pzmap(ss(MatA, MatB, MatC, MatD)) ;
grid minor ;

%% FIXED PARAMETERS

% Measurement Unit Conversion
kt_to_ms = 0.514444 ; % [m/(s*kt)]
ft_to_m  = 0.3048 ; % [m/ft]
deg2rad  = 0.0174533 ; % [rad/deg]

% Simulation Timestep
timeStep = 1/60 ; % [s]

% Treshold Parameters of Modes Logics
VPIas         = 60 ; % [kt]
Vne           = 150 ; % [kt]
MinIasIAS     = 30 ; % [kt]
MinIasTC      = 40 ; % [kt]
MinRadaltRHT = 3 ; % [ft]
MaxRadaltRHT = 2200 ; % [ft]
MinBaraltALT = 30 ; % [ft]
MaxIasHOV     = 80 ; % [kt]
MaxGsxHOV     = 40 ; % [kt]
MaxGsyHOV     = 20 ; % [kt]

% Actuators Saturation Limits (expressed as actuator stroke
   percentages normalized to 1)
UpperSatPitch       = 0.11 ;
LowerSatPitch       = -0.11 ;
UpperSatRoll        = 0.19 ;
LowerSatRoll        = -0.19 ;
```

```
61 UpperSatYaw         = 0.215 ;
62 LowerSatYaw         = -0.215 ;
63 UpperSatCollective = 0.07 ;
64 LowerSatCollective = -0.07 ;
65
66 % Helicopter Attitude Limits
67 LowerSatTheta = -12 ; % [deg]
68 UpperSatTheta = +14 ; %[deg]
69 LowerSatPhi   = -30 ; %[deg]
70 UpperSatPhi   = +30 ; %[deg]
71 LowerSatPsi   = -360 ; %[deg]
72 UpperSatPsi   = +360 ; %[deg]
73
74 % ATTPitch
75 ATTPitchMType           = 1 ; % 1=Lower Mode, 2=Upper Mode
76 ATTPitchCType           = 4 ; % 1=P, 2=PI, 3=PD, 4=PID
77 beepTrimATTPitchRate    = 0.01745*3 ; % [rad/s]
78
79 % ATTRoll
80 ATTRollMType            = 1 ; % 1=Lower Mode, 2=Upper Mode
81 ATTRollCType            = 4 ; % 1=P, 2=PI, 3=PD, 4=PID
82 beepTrimATTRollRate     = 0.01745*3 ; % [rad/s]
83
84 % ATTYaw (Equal to HDGYaw)
85 ATTYawMType             = 1 ; % 1=Lower Mode, 2=Upper Mode
86 ATTYawCType             = 4 ; % 1=P, 2=PI, 3=PD, 4=PID
87 beepTrimATTYawRate      = 0.09 ; % [rad/s]
88
89 % TC
90 TCCType                 = 4 ; % 1=P, 2=PI, 3=PD, 4=PID
91 MinPhiTC                = 0 ; % [deg]
92
93 % IAS
94 IASMType                = 2 ; 1=Lower Mode, 2=Upper Mode
95 IASCType                = 4 ; % 1=P, 2=PI, 3=PD, 4=PID
96 beepTrimIASRate         = -4 ; % [m/s]
97
```

```
98  % HDGRoll
99  HDGRollMType              = 2 ; % 1=Lower Mode, 2=Upper Mode
100 HDGRollCType              = 4 ; % 1=P, 2=PI, 3=PD, 4=PID
101 beepTrimHDGRollRate       = 0.09 ; %[rad/s]
102
103 % ALTPitch/ALTCollective/RHT
104 ALTCollective_RHT_MType   = 2 ; % 1=Lower Mode, 2=Upper Mode
105 ALTPitchMType             = 2 ; % 1=Lower Mode, 2=Upper Mode
106 ALTCollective_RHT_CType   = 4 ; % 1=P, 2=PI, 3=PD, 4=PID
107 ALTPitchCType             = 4 ; % 1=P, 2=PI, 3=PD, 4=PID
108 beepTrimRHTRate          = 22.4 ; % [m/s]
109 beepTrimALTRate          = 22.4 ; % [m/s]
110
111 % HOVPitch/HOVRoll
112 HOVPitchMType             = 2 ; % 1=Lower Mode, 2=Upper Mode
113 HOVRollMType              = 2 ; % 1=Lower Mode, 2=Upper Mode
114 HOVPitchCType             = 4 ; % 1=P, 2=PI, 3=PD, 4=PID
115 HOVRollCType              = 4 ; % 1=P, 2=PI, 3=PD, 4=PID
116 beepTrimHOVPitchRate      = 1 ; % [m/s]
117 beepTrimHOVRollRate       = 1 ; % [m/s]
118
119 %% TUNABLE PARAMETERS (TUNparameters are only used for tuning
        purposes)
120
121 % SAS
122 sasPitchP        = .85 ;
123 TUNsasPitchPm    = .84 ;
124 TUNsasPitchPM    = .86 ;
125 sasPitchI        = .1 ;
126 TUNsasPitchIm    = .09 ;
127 TUNsasPitchIM    = .11 ;
128
129 sasRollP         = .3 ;
130 TUNsasRollPm     = 0.29 ;
131 TUNsasRollPM     = .31 ;
132 sasRollI         = .1 ;
133 TUNsasRollIm     = .09 ;
```

```
TUNsasRollIM    = .11 ;

sasYawP         = 1.2 ;
TUNsasYawPm     = 1.19 ;
TUNsasYawPM     = 1.21 ;
sasYawI         = .7 ;
TUNsasYawIm     = .69 ;
TUNsasYawIM     = .71 ;

% ATT
ATTPitchKp     = .1 ;
TUNATTPitchKpm = -10 ;
TUNATTPitchKpM = 10 ;
ATTPitchb      = 1 ;
TUNATTPitchbm  = 0 ;
TUNATTPitchbM  = 1 ;
ATTPitchTi     = 10 ;
TUNATTPitchKi  = ATTPitchKp/ATTPitchTi ;
TUNATTPitchKim = -10 ;
TUNATTPitchKiM = 10 ;
ATTPitchTd     = 1 ;
TUNATTPitchKd  = ATTPitchKp * ATTPitchTd ;
TUNATTPitchKdm = -10 ;
TUNATTPitchKdM = 10 ;
ATTPitchN      = 10 ;
TUNATTPitchTf  = TUNATTPitchKd/(ATTPitchKp * ATTPitchN) ;
TUNATTPitchTfm = .01 ;
TUNATTPitchTfM = 10 ;
ATTPitchc      = 0 ;
TUNATTPitchcm  = 0 ;
TUNATTPitchcM  = 1 ;

ATTRollKp     = .1 ;
TUNATTRollKpm = -10 ;
TUNATTRollKpM = 10 ;
ATTRollb      = 1 ;
TUNATTRollbm  = 0 ;
```

```
171 TUNATTRollbM  = 1 ;
172 ATTRollTi     = 10 ;
173 TUNATTRollKi  = ATTRollKp/ATTRollTi ;
174 TUNATTRollKim = -10 ;
175 TUNATTRollKiM = 10 ;
176 ATTRollTd     = 1 ;
177 TUNATTRollKd  = ATTRollKp * ATTRollTd ;
178 TUNATTRollKdm = -10 ;
179 TUNATTRollKdM = 10 ;
180 ATTRollN      = 10 ;
181 TUNATTRollTf  = TUNATTRollKd/(ATTRollKp * ATTRollN) ;
182 TUNATTRollTfm = .01 ;
183 TUNATTRollTfM = 10 ;
184 ATTRollc      = 0 ;
185 TUNATTRollcm  = 0 ;
186 TUNATTRollcM  = 1 ;
187
188 ATTYawKp     = .1 ;
189 TUNATTYawKpm = -10 ;
190 TUNATTYawKpM = 10 ;
191 ATTYawb      = 1 ;
192 TUNATTYawbm  = 0 ;
193 TUNATTYawbM  = 1 ;
194 ATTYawTi     = 10 ;
195 TUNATTYawKi  = ATTYawKp/ATTYawTi ;
196 TUNATTYawKim = -10 ;
197 TUNATTYawKiM = 10 ;
198 ATTYawTd     = 1 ;
199 TUNATTYawKd  = ATTYawKp * ATTYawTd ;
200 TUNATTYawKdm = -10 ;
201 TUNATTYawKdM = 10 ;
202 ATTYawN      = 10 ;
203 TUNATTYawTf  = TUNATTYawKd/(ATTYawKp * ATTYawN) ;
204 TUNATTYawTfm = .01 ;
205 TUNATTYawTfM = 10 ;
206 ATTYawc      = 0 ;
207 TUNATTYawcm  = 0 ;
```

```
208 TUNATTYawcM  = 1 ;
209
210 % TC
211 TCKp         = .1 ;
212 TUNTCKpm     = .1 ;
213 TUNTCKpM     = .3 ;
214 TCb          = 1 ;
215 TUNTCbm      = 0 ;
216 TUNTCbM      = 1 ;
217 TCTi         = .1 ;
218 TUNTCKi      = TCKp/TCTi ;
219 TUNTCKim     = 0 ;
220 TUNTCKiM     = 2 ;
221 TCTd         = .1 ;
222 TUNTCKd      = TCKp * TCTd ;
223 TUNTCKdm     = 0 ;
224 TUNTCKdM     = 2 ;
225 TCN          = 1 ;
226 TUNTCTf      = TUNTCKd/(TCKp * TCN) ;
227 TUNTCTfm     = .4 ;
228 TUNTCTfM     = 2 ;
229 TCc          = 0 ;
230 TUNTCcm      = 0 ;
231 TUNTCcM      = 1 ;
232
233 % ALTCollective / RHT
234 ALTCollective_RHT_Kp     = .1 ;
235 TUNALTCollective_RHT_Kpm = -10 ;
236 TUNALTCollective_RHT_KpM = 10 ;
237 ALTCollective_RHT_b      = 1 ;
238 TUNALTCollective_RHT_bm  = 0 ;
239 TUNALTCollective_RHT_bM  = 1 ;
240 ALTCollective_RHT_Ti     = 10 ;
241 TUNALTCollective_RHT_Ki  = ALTCollective_RHT_Kp/
      ALTCollective_RHT_Ti ;
242 TUNALTCollective_RHT_Kim = -10 ;
243 TUNALTCollective_RHT_KiM = 10 ;
```

```
244 ALTCollective_RHT_Td     = 1 ;
245 TUNALTCollective_RHT_Kd  = ALTCollective_RHT_Kp *
       ALTCollective_RHT_Td ;
246 TUNALTCollective_RHT_Kdm = -10 ;
247 TUNALTCollective_RHT_KdM = 10 ;
248 ALTCollective_RHT_N      = 10 ;
249 TUNALTCollective_RHT_Tf  = TUNALTCollective_RHT_Kd/(
       ALTCollective_RHT_Kp * ALTCollective_RHT_N) ;
250 TUNALTCollective_RHT_Tfm = .01 ;
251 TUNALTCollective_RHT_TfM = 10 ;
252 ALTCollective_RHT_c      = 0 ;
253 TUNALTCollective_RHT_cm  = 0 ;
254 TUNALTCollective_RHT_cM  = 1 ;
255
256 % IAS
257 IASKp     = .1 ;
258 TUNIASKpm = -10 ;
259 TUNIASKpM = 10 ;
260 IASb      = 1 ;
261 TUNIASbm  = 0 ;
262 TUNIASbM  = 1 ;
263 IASTi     = 10 ;
264 TUNIASKi  = IASKp/IASTi ;
265 TUNIASKim = -10 ;
266 TUNIASKiM = 10 ;
267 IASTd     = 1 ;
268 TUNIASKd  = IASKp * IASTd ;
269 TUNIASKdm = -10 ;
270 TUNIASKdM = 10 ;
271 IASN      = 10 ;
272 TUNIASTf  = TUNIASKd/(IASKp * IASN) ;
273 TUNIASTfm = .01 ;
274 TUNIASTfM = 10 ;
275 IASc      = 0 ;
276 TUNIAScm  = 0 ;
277 TUNIAScM  = 1 ;
278
```

```
% ALTPitch
ALTPitchKp     = .1 ;
TUNALTPitchKpm = -10 ;
TUNALTPitchKpM = 10 ;
ALTPitchb      = 1 ;
TUNALTPitchbm  = 0 ;
TUNALTPitchbM  = 1 ;
ALTPitchTi     = 10 ;
TUNALTPitchKi  = ALTPitchKp/ALTPitchTi ;
TUNALTPitchKim = -10 ;
TUNALTPitchKiM = 10 ;
ALTPitchTd     = 1 ;
TUNALTPitchKd  = ALTPitchKp * ALTPitchTd ;
TUNALTPitchKdm = -10 ;
TUNALTPitchKdM = 10 ;
ALTPitchN      = 10 ;
TUNALTPitchTf  = TUNALTPitchKd/(ALTPitchKp * ALTPitchN) ;
TUNALTPitchTfm = .01 ;
TUNALTPitchTfM = 10 ;
ALTPitchc      = 0 ;
TUNALTPitchcm  = 0 ;
TUNALTPitchcM  = 1 ;

% HDGRoll
HDGRollKp     = .1 ;
TUNHDGRollKpm = -10 ;
TUNHDGRollKpM = 10 ;
HDGRollb      = 1 ;
TUNHDGRollbm  = 0 ;
TUNHDGRollbM  = 1 ;
HDGRollTi     = 10 ;
TUNHDGRollKi  = HDGRollKp/HDGRollTi ;
TUNHDGRollKim = -10 ;
TUNHDGRollKiM = 10 ;
HDGRollTd     = 1 ;
TUNHDGRollKd  = HDGRollKp * HDGRollTd ;
TUNHDGRollKdm = -10 ;
```

```
316 TUNHDGRollKdM = 10 ;
317 HDGRollN      = 10 ;
318 TUNHDGRollTf  = TUNHDGRollKd/(HDGRollKp * HDGRollN) ;
319 TUNHDGRollTfm = .01 ;
320 TUNHDGRollTfM = 10 ;
321 HDGRollc      = 0 ;
322 TUNHDGRollcm  = 0 ;
323 TUNHDGRollcM  = 1 ;
324
325 % HOVPitch
326 HOVPitchKp     = .1 ;
327 TUNHOVPitchKpm = -10 ;
328 TUNHOVPitchKpM = 10 ;
329 HOVPitchb      = 1 ;
330 TUNHOVPitchbm  = 0 ;
331 TUNHOVPitchbM  = 1 ;
332 HOVPitchTi     = 10 ;
333 TUNHOVPitchKi  = HOVPitchKp/HOVPitchTi ;
334 TUNHOVPitchKim = -10 ;
335 TUNHOVPitchKiM = 10 ;
336 HOVPitchTd     = 1 ;
337 TUNHOVPitchKd  = HOVPitchKp * HOVPitchTd ;
338 TUNHOVPitchKdm = -10 ;
339 TUNHOVPitchKdM = 10 ;
340 HOVPitchN      = 10 ;
341 TUNHOVPitchTf  = TUNHOVPitchKd/(HOVPitchKp * HOVPitchN) ;
342 TUNHOVPitchTfm = .01 ;
343 TUNHOVPitchTfM = 10 ;
344 HOVPitchc      = 0 ;
345 TUNHOVPitchcm  = 0 ;
346 TUNHOVPitchcM  = 1 ;
347
348 % HOVRoll
349 HOVRollKp     = .1 ;
350 TUNHOVRollKpm = -10 ;
351 TUNHOVRollKpM = 10 ;
352 HOVRollb      = 1 ;
```

```
353 TUNHOVRollbm  = 0 ;
354 TUNHOVRollbM  = 1 ;
355 HOVRollTi     = 10 ;
356 TUNHOVRollKi  = HOVRollKp/HOVRollTi ;
357 TUNHOVRollKim = -10 ;
358 TUNHOVRollKiM = 10 ;
359 HOVRollTd     = 1 ;
360 TUNHOVRollKd  = HOVRollKp * HOVRollTd ;
361 TUNHOVRollKdm = -10 ;
362 TUNHOVRollKdM = 10 ;
363 HOVRollN      = 10 ;
364 TUNHOVRollTf  = TUNHOVRollKd/(HOVRollKp * HOVRollN) ;
365 TUNHOVRollTfm = .01 ;
366 TUNHOVRollTfM = 10 ;
367 HOVRollc      = 0 ;
368 TUNHOVRollcm  = 0 ;
369 TUNHOVRollcM  = 1 ;
370
371 %% INITIALIZATION OF SIGNALS
372
373 open('AFCS_Model')
374
375 %Modes buttons AFCS
376 signalbuilder('AFCS_Model/SignalAP' ,'set','Signal 1','Group
       1',[0 timeStep timeStep 1 1 10],[0 0 0 0 0 0])
377 signalbuilder('AFCS_Model/SignalIAS','set','Signal 1','Group
       1',[0 timeStep timeStep 1 1 10],[0 0 0 0 0 0])
378 signalbuilder('AFCS_Model/SignalHOV','set','Signal 1','Group
       1',[0 timeStep timeStep 1 1 10],[0 0 0 0 0 0])
379 signalbuilder('AFCS_Model/SignalHDG','set','Signal 1','Group
       1',[0 timeStep timeStep 1 1 10],[0 0 0 0 0 0])
380 signalbuilder('AFCS_Model/SignalALT','set','Signal 1','Group
       1',[0 timeStep timeStep 1 1 10],[0 0 0 0 0 0])
381 signalbuilder('AFCS_Model/SignalRHT','set','Signal 1','Group
       1',[0 timeStep timeStep 1 1 10],[0 0 0 0 0 0])
382
```

```
383 %Out of detent of the pilot commands (i.e. cyclic/collective/
       pedals ood)
384 TUNoodPitch = 0 ; TUNoodRoll = 0 ;  TUNoodYaw = 0 ;
       TUNoodCollective = 0 ;
385
386 %Force Trim button (i.e. cyclic/collective ft button)
387 TUNftPitch = 0 ; TUNftRoll = 0 ;  TUNftYaw = 0 ;
       TUNftCollective = 0 ;
388
389 %Beeptrim button (i.e. cyclic/collective bt button)
390 signalbuilder('AFCS_Model/SignalbtPitch'     ,'set','Signal 1
       ','Group 1',[0 timeStep timeStep 1 1 10],[0 0 0 0 0 0])
391 signalbuilder('AFCS_Model/SignalbtRoll'      ,'set','Signal 1
       ','Group 1',[0 timeStep timeStep 1 1 10],[0 0 0 0 0 0])
392 signalbuilder('AFCS_Model/SignalbtYaw'       ,'set','Signal 1
       ','Group 1',[0 timeStep timeStep 1 1 10],[0 0 0 0 0 0])
393 signalbuilder('AFCS_Model/SignalbtCollective','set','Signal 1
       ','Group 1',[0 timeStep timeStep 1 1 10],[0 0 0 0 0 0])
394
395 %% ----- TUNING SAS PI -----
396
397 %% Create system data with slTuner interface
398
399 TunedBlocks = {'AFCS_Model/LowerModes/SASPitch/Proportional';
        ...
400                'AFCS_Model/LowerModes/SASRoll/Proportional';
                     ...
401                'AFCS_Model/LowerModes/SASYaw/Proportional';
                     ...
402                'AFCS_Model/LowerModes/SASPitch/Integral'; ...
403                'AFCS_Model/LowerModes/SASRoll/Integral'; ...
404                'AFCS_Model/LowerModes/SASYaw/Integral'};
405
406 AnalysisPoints = {'AFCS_Model/FR_deltaPitchMODE/1'; ...
407                   'AFCS_Model/FR_deltaRollMODE/1'; ...
408                   'AFCS_Model/FR_deltaYawMODE/1'; ...
409                   'AFCS_Model/FR_deltaCollectiveMODE/1'; ...
```

```
410                     'AFCS_Model/Demux/1'; ...
411                     'AFCS_Model/Demux/3'; ...
412                     'AFCS_Model/Demux/4'};
413
414
415 % Specify the custom options
416 Options = slTunerOptions('AreParamsTunable',true);
417 % Create the slTuner object
418 CL0 = slTuner('AFCS_Model',TunedBlocks,AnalysisPoints,Options
    );
419
420 % Set the parameterization of the tuned block
421 AFCS_Model_SASPitch_Proportional = tunableGain('
    AFCS_Model_SASPitch_Proportional',1,1);
422 AFCS_Model_SASPitch_Proportional.Gain.Value = sasPitchP;
423 AFCS_Model_SASPitch_Proportional.Gain.Minimum = TUNsasPitchPm
    ;
424 AFCS_Model_SASPitch_Proportional.Gain.Maximum = TUNsasPitchPM
    ;
425 setBlockParam(CL0,'AFCS_Model/LowerModes/SASPitch/
    Proportional',AFCS_Model_SASPitch_Proportional);
426
427 % Set the parameterization of the tuned block
428 AFCS_Model_SASRoll_Proportional = tunableGain('
    AFCS_Model_SASRoll_Proportional',1,1);
429 AFCS_Model_SASRoll_Proportional.Gain.Value = sasRollP;
430 AFCS_Model_SASRoll_Proportional.Gain.Minimum = TUNsasRollPm;
431 AFCS_Model_SASRoll_Proportional.Gain.Maximum = TUNsasRollPM;
432 setBlockParam(CL0,'AFCS_Model/LowerModes/SASRoll/Proportional
    ',AFCS_Model_SASRoll_Proportional);
433
434 % Set the parameterization of the tuned block
435 AFCS_Model_SASYaw_Proportional = tunableGain('
    AFCS_Model_SASYaw_Proportional',1,1);
436 AFCS_Model_SASYaw_Proportional.Gain.Value = sasYawP;
437 AFCS_Model_SASYaw_Proportional.Gain.Minimum = TUNsasYawPm;
438 AFCS_Model_SASYaw_Proportional.Gain.Maximum = TUNsasYawPM;
```

```
439 setBlockParam(CL0,'AFCS_Model/LowerModes/SASYaw/Proportional'
       ,AFCS_Model_SASYaw_Proportional);
440
441 % Set the parameterization of the tuned block
442 AFCS_Model_SASPitch_Integral = tunableGain('
       AFCS_Model_SASPitch_Integral',1,1);
443 AFCS_Model_SASPitch_Integral.Gain.Value = sasPitchI;
444 AFCS_Model_SASPitch_Integral.Gain.Minimum = TUNsasPitchIm;
445 AFCS_Model_SASPitch_Integral.Gain.Maximum = TUNsasPitchIM;
446 setBlockParam(CL0,'AFCS_Model/LowerModes/SASPitch/Integral',
       AFCS_Model_SASPitch_Integral);
447
448 % Set the parameterization of the tuned block
449 AFCS_Model_SASRoll_Integral = tunableGain('
       AFCS_Model_SASRoll_Integral',1,1);
450 AFCS_Model_SASRoll_Integral.Gain.Value = sasRollI;
451 AFCS_Model_SASRoll_Integral.Gain.Minimum = TUNsasRollIm;
452 AFCS_Model_SASRoll_Integral.Gain.Maximum = TUNsasRollIM;
453 setBlockParam(CL0,'AFCS_Model/LowerModes/SASRoll/Integral',
       AFCS_Model_SASRoll_Integral);
454
455 % Set the parameterization of the tuned block
456 AFCS_Model_SASYaw_Integral = tunableGain('
       AFCS_Model_SASYaw_Integral',1,1);
457 AFCS_Model_SASYaw_Integral.Gain.Value = sasYawI;
458 AFCS_Model_SASYaw_Integral.Gain.Minimum = TUNsasYawIm;
459 AFCS_Model_SASYaw_Integral.Gain.Maximum = TUNsasYawIM;
460 setBlockParam(CL0,'AFCS_Model/LowerModes/SASYaw/Integral',
       AFCS_Model_SASYaw_Integral);
461
462 %% Create tuning goal to constrain the dynamics of the closed
       -loop system
463
464 % Tuning goal specifications
465 MinDecay = 0; % Minimum decay rate of closed-loop poles
466 MinDamping = 0; % Minimum damping of closed-loop poles
```

```
467 MaxFrequency = inf; % Maximum natural frequency of closed-
       loop poles
468 % Create tuning goal for closed-loop poles
469 PolesGoal = TuningGoal.Poles(MinDecay,MinDamping,MaxFrequency
       );
470 PolesGoal.Name = 'PolesGoal'; % Tuning goal name
471
472 %% Create option set for systune command
473 Options = systuneOptions();
474 Options.Display = 'final'; % Tuning display level ('final', '
       sub', 'iter', 'off')
475 Options.RandomStart = 100; % Number of randomized starts
476
477 %% Set soft and hard goals
478 SoftGoals = [];
479 HardGoals = [PolesGoal];
480
481 %% Tune the parameters with soft and hard goals
482 [CL1,~,~,~] = systune(CL0,SoftGoals,HardGoals,Options);
483
484 Sas = getTunedValue(CL1) ;
485
486 sasPitchP     = Sas.AFCS_Model_SASPitch_Proportional.D ;
487 sasRollP      = Sas.AFCS_Model_SASRoll_Proportional.D ;
488 sasYawP       = Sas.AFCS_Model_SASYaw_Proportional.D ;
489 sasPitchI     = Sas.AFCS_Model_SASPitch_Integral.D ;
490 sasRollI      = Sas.AFCS_Model_SASRoll_Integral.D ;
491 sasYawI       = Sas.AFCS_Model_SASYaw_Integral.D ;
492
493 %% View tuning results
494 figure ;
495 viewGoal([SoftGoals;HardGoals],CL1) ;
496
497 %% ----- TUNING ATTPITCH + ATTROLL + TC -----
498
499 %% Create system data with slTuner interface
500 % Engage ATT
```

```
501 signalbuilder('AFCS_Model/SignalAP','set','Signal 1','Group 1
      ',[0 timeStep timeStep 1 1 10],[0 0 1 1 0 0])
502
503 TunedBlocks = {'AFCS_Model/LowerModes/DeltaCommandPitch/
      Controller'; ...
504                 'AFCS_Model/LowerModes/DeltaCommandRoll/
                      Controller'; ...
505                 'AFCS_Model/Functionalities/TC/Controller'};
506 AnalysisPoints = {'AFCS_Model/Demux/2'; ...
507                   'AFCS_Model/Demux/5'; ...
508                   'AFCS_Model/Demux/7'; ...
509                   'AFCS_Model/LowerModes/DeltaCommandPitch/
                        SetpointFr/1'; ...
510                   'AFCS_Model/LowerModes/DeltaCommandRoll/
                        SetpointFr/1'};
511
512 OperatingPoints = timeStep*5;  % linearization point
513 % Specify the custom options
514 Options = slTunerOptions('AreParamsTunable',false);
515 % Create the slTuner object
516 CL0 = slTuner('AFCS_Model',TunedBlocks,AnalysisPoints,
      OperatingPoints,Options);
517
518 % Set the parameterization of the tuned block
519 AFCS_Model_ATTPitch_Controller = tunablePID2('
      AFCS_Model_ATTPitch_Controller','PID');
520 AFCS_Model_ATTPitch_Controller.Kp.Value = ATTPitchKp;
521 AFCS_Model_ATTPitch_Controller.Kp.Minimum = TUNATTPitchKpm;
522 AFCS_Model_ATTPitch_Controller.Kp.Maximum = TUNATTPitchKpM;
523 AFCS_Model_ATTPitch_Controller.b.Value = ATTPitchb;
524 AFCS_Model_ATTPitch_Controller.b.Free = 1;
525 AFCS_Model_ATTPitch_Controller.b.Minimum = TUNATTPitchbm;
526 AFCS_Model_ATTPitch_Controller.b.Maximum = TUNATTPitchbM;
527 AFCS_Model_ATTPitch_Controller.Ki.Value = TUNATTPitchKi;
528 AFCS_Model_ATTPitch_Controller.Ki.Minimum = TUNATTPitchKim;
529 AFCS_Model_ATTPitch_Controller.Ki.Maximum = TUNATTPitchKiM;
530 AFCS_Model_ATTPitch_Controller.Kd.Value = TUNATTPitchKd;
```

```
AFCS_Model_ATTPitch_Controller.Kd.Minimum = TUNATTPitchKdm;
AFCS_Model_ATTPitch_Controller.Kd.Maximum = TUNATTPitchKdM;
AFCS_Model_ATTPitch_Controller.Tf.Value = TUNATTPitchTf;
AFCS_Model_ATTPitch_Controller.Tf.Minimum = TUNATTPitchTfm;
AFCS_Model_ATTPitch_Controller.Tf.Maximum = TUNATTPitchTfM;
AFCS_Model_ATTPitch_Controller.c.Value = ATTPitchc;
AFCS_Model_ATTPitch_Controller.c.Free = 1;
AFCS_Model_ATTPitch_Controller.c.Minimum = TUNATTPitchcm;
AFCS_Model_ATTPitch_Controller.c.Maximum = TUNATTPitchcM;
setBlockParam(CL0,'AFCS_Model/LowerModes/DeltaCommandPitch/
   Controller',AFCS_Model_ATTPitch_Controller);

% Set the parameterization of the tuned block
AFCS_Model_ATTRoll_Controller = tunablePID2('
   AFCS_Model_ATTRoll_Controller','PID');
AFCS_Model_ATTRoll_Controller.Kp.Value = ATTRollKp;
AFCS_Model_ATTRoll_Controller.Kp.Minimum = TUNATTRollKpm;
AFCS_Model_ATTRoll_Controller.Kp.Maximum = TUNATTRollKpM;
AFCS_Model_ATTRoll_Controller.b.Value = ATTRollb;
AFCS_Model_ATTRoll_Controller.b.Free = 1;
AFCS_Model_ATTRoll_Controller.b.Minimum = TUNATTRollbm;
AFCS_Model_ATTRoll_Controller.b.Maximum = TUNATTRollbM;
AFCS_Model_ATTRoll_Controller.Ki.Value = TUNATTRollKi;
AFCS_Model_ATTRoll_Controller.Ki.Minimum = TUNATTRollKim;
AFCS_Model_ATTRoll_Controller.Ki.Maximum = TUNATTRollKiM;
AFCS_Model_ATTRoll_Controller.Kd.Value = TUNATTRollKd;
AFCS_Model_ATTRoll_Controller.Kd.Minimum = TUNATTRollKdm;
AFCS_Model_ATTRoll_Controller.Kd.Maximum = TUNATTRollKdM;
AFCS_Model_ATTRoll_Controller.Tf.Value = TUNATTRollTf;
AFCS_Model_ATTRoll_Controller.Tf.Minimum = TUNATTRollTfm;
AFCS_Model_ATTRoll_Controller.Tf.Maximum = TUNATTRollTfM;
AFCS_Model_ATTRoll_Controller.c.Value = ATTRollc;
AFCS_Model_ATTRoll_Controller.c.Free = 1;
AFCS_Model_ATTRoll_Controller.c.Minimum = TUNATTRollcm;
AFCS_Model_ATTRoll_Controller.c.Maximum = TUNATTRollcM;
setBlockParam(CL0,'AFCS_Model/LowerModes/DeltaCommandRoll/
   Controller',AFCS_Model_ATTRoll_Controller);
```

```
565
566 % Set the parameterization of the tuned block
567 AFCS_Model_TC_Controller = tunablePID2('
       AFCS_Model_TC_Controller','PID');
568 AFCS_Model_TC_Controller.Kp.Value = TCKp;
569 AFCS_Model_TC_Controller.Kp.Minimum = TUNTCKpm;
570 AFCS_Model_TC_Controller.Kp.Maximum = TUNTCKpM;
571 AFCS_Model_TC_Controller.b.Value = TCb;
572 AFCS_Model_TC_Controller.b.Free = 1;
573 AFCS_Model_TC_Controller.b.Minimum = TUNTCbm;
574 AFCS_Model_TC_Controller.b.Maximum = TUNTCbM;
575 AFCS_Model_TC_Controller.Ki.Value = TUNTCKi;
576 AFCS_Model_TC_Controller.Ki.Free = 1;
577 AFCS_Model_TC_Controller.Ki.Minimum = TUNTCKim;
578 AFCS_Model_TC_Controller.Ki.Maximum = TUNTCKiM;
579 AFCS_Model_TC_Controller.Kd.Value = TUNTCKd;
580 AFCS_Model_TC_Controller.Kd.Free = 1;
581 AFCS_Model_TC_Controller.Kd.Minimum = TUNTCKdm;
582 AFCS_Model_TC_Controller.Kd.Maximum = TUNTCKdM;
583 AFCS_Model_TC_Controller.Tf.Value = TUNTCTf;
584 AFCS_Model_TC_Controller.Tf.Free = 1;
585 AFCS_Model_TC_Controller.Tf.Minimum = TUNTCTfm;
586 AFCS_Model_TC_Controller.Tf.Maximum = TUNTCTfM;
587 AFCS_Model_TC_Controller.c.Value = TCc;
588 AFCS_Model_TC_Controller.c.Free = 1;
589 AFCS_Model_TC_Controller.c.Minimum = TUNTCcm;
590 AFCS_Model_TC_Controller.c.Maximum = TUNTCcM;
591 setBlockParam(CL0,'AFCS_Model/Functionalities/TC/Controller',
       AFCS_Model_TC_Controller);
592
593 %% Create tuning goal to shape how the closed-loop system
       responds to a specific input signal
594 % Inputs and outputs
595 Inputs = {'AFCS_Model/LowerModes/DeltaCommandPitch/SetpointFr
       /1'};
596 Outputs = {'AFCS_Model/Demux/2'};
597 % Tuning goal specifications
```

```
598 Tau = 0.7; % Time constant
599 Overshoot = 5; % Overshoot (%)
600 % Create tuning goal for step tracking
601 StepTrackingGoalPitch = TuningGoal.StepTracking(Inputs,
       Outputs,Tau,Overshoot);
602 StepTrackingGoalPitch.Name = 'StepTrackingGoalPitch'; %
       Tuning goal name
603
604 %% Create tuning goal to shape how the closed-loop system
       responds to a specific input signal
605 % Inputs and outputs
606 Inputs = {'AFCS_Model/LowerModes/DeltaCommandRoll/SetpointFr
       /1'};
607 Outputs = {'AFCS_Model/Demux/5'};
608 % Tuning goal specifications
609 Tau = 0.7; % Time constant
610 Overshoot = 5; % Overshoot (%)
611 % Create tuning goal for step tracking
612 StepTrackingGoalRoll = TuningGoal.StepTracking(Inputs,Outputs
       ,Tau,Overshoot);
613 StepTrackingGoalRoll.Name = 'StepTrackingGoalRoll'; % Tuning
       goal name
614
615 %% Create tuning goal to reject step disturbances with the
       minimum performance as in the desired response
616 % Inputs and outputs
617 Inputs = {'AFCS_Model/LowerModes/DeltaCommandRoll/SetpointFr
       /1'};
618 Outputs = {'AFCS_Model/Demux/7'};
619 % Tuning goal specifications
620 MaxAmplitude = .7;
621 MaxSettlingTime = 20;
622 MinDamping = 1;
623 % Create tuning goal for step tracking
624 StepRejectionGoalTC = TuningGoal.StepRejection(Inputs,Outputs
       ,MaxAmplitude,MaxSettlingTime,MinDamping);
```

```
625 StepRejectionGoalTC.Name = 'StepRejectionGoalTC'; % Tuning
       goal name
626
627 %% Create tuning goal to enforce specific gain and phase
       margins
628 % Feedback loop locations
629 Locations = {'AFCS_Model/Demux/2';...
630              'AFCS_Model/Demux/5';...
631              'AFCS_Model/Demux/7';};
632 % Tuning goal specifications
633 GainMargin = 5; % Required minimum gain margin
634 PhaseMargin = 30; % Required minimum phase margin
635 % Create tuning goal for margins
636 MarginsGoalATT = TuningGoal.Margins(Locations,GainMargin,
       PhaseMargin);
637 MarginsGoalATT.Name = 'MarginsGoalATT'; % Tuning goal name
638
639 %% Create option set for systune command
640 Options = systuneOptions();
641 Options.Display = 'final'; % Tuning display level ('final', '
       sub', 'iter', 'off')
642 Options.RandomStart = 10; % Number of randomized starts
643
644 %% Set soft and hard goals
645 SoftGoals = [ StepRejectionGoalTC ];
646 HardGoals = [ StepTrackingGoalPitch, StepTrackingGoalRoll,
       MarginsGoalATT];
647
648 %% Tune the parameters with soft and hard goals
649 [CL1,~,~,~] = systune(CL0,SoftGoals,HardGoals,Options);
650
651 AP = getTunedValue(CL1) ;
652
653 ATTPitchKp    = AP.AFCS_Model_ATTPitch_Controller.Kp ;
654 ATTPitchb     = AP.AFCS_Model_ATTPitch_Controller.b ;
655 ATTPitchTi    = AP.AFCS_Model_ATTPitch_Controller.Kp/AP.
       AFCS_Model_ATTPitch_Controller.Ki ;
```

```
656 ATTPitchTd    = AP.AFCS_Model_ATTPitch_Controller.Kd/AP.
       AFCS_Model_ATTPitch_Controller.Kp ;
657 ATTPitchN     = AP.AFCS_Model_ATTPitch_Controller.Kd/(AP.
       AFCS_Model_ATTPitch_Controller.Kp * AP.
       AFCS_Model_ATTPitch_Controller.Tf) ;
658 ATTPitchc     = AP.AFCS_Model_ATTPitch_Controller.c ;
659
660 ATTRollKp     = AP.AFCS_Model_ATTRoll_Controller.Kp ;
661 ATTRollb      = AP.AFCS_Model_ATTRoll_Controller.b ;
662 ATTRollTi     = AP.AFCS_Model_ATTRoll_Controller.Kp/AP.
       AFCS_Model_ATTRoll_Controller.Ki ;
663 ATTRollTd     = AP.AFCS_Model_ATTRoll_Controller.Kd/AP.
       AFCS_Model_ATTRoll_Controller.Kp ;
664 ATTRollN      = AP.AFCS_Model_ATTRoll_Controller.Kd/(AP.
       AFCS_Model_ATTRoll_Controller.Kp * AP.
       AFCS_Model_ATTRoll_Controller.Tf) ;
665 ATTRollc      = AP.AFCS_Model_ATTRoll_Controller.c ;
666
667 TCKp    = AP.AFCS_Model_TC_Controller.Kp ;
668 TCb     = AP.AFCS_Model_TC_Controller.b ;
669 TCTi    = AP.AFCS_Model_TC_Controller.Kp/AP.
       AFCS_Model_TC_Controller.Ki ;
670 TCTd    = AP.AFCS_Model_TC_Controller.Kd/AP.
       AFCS_Model_TC_Controller.Kp ;
671 TCN     = AP.AFCS_Model_TC_Controller.Kd/(AP.
       AFCS_Model_TC_Controller.Kp * AP.AFCS_Model_TC_Controller.
       Tf) ;
672 TCc     = AP.AFCS_Model_TC_Controller.c ;
673
674 %% View tuning results
675 figure ;
676 viewGoal(SoftGoals(1),CL1) ;
677 figure ;
678 viewGoal(HardGoals(1),CL1) ;
679 figure ;
680 viewGoal(HardGoals(2),CL1) ;
681 figure ;
```

```
682 viewGoal(HardGoals(3),CL1) ;
683
684 %% Restore initial configuration
685 signalbuilder('AFCS_Model/SignalAP','set','Signal 1','Group 1
      ',[0 1 1 2 2 10],[0 0 0 0 0 0])
686
687
688 %% ----- TUNING IAS -----
689
690 %% Create system data with slTuner interface
691
692 % Engage ATT
693 signalbuilder('AFCS_Model/SignalAP','set','Signal 1','Group 1
      ',[0 timeStep timeStep 0.5 0.5 10],[0 0 1 1 0 0])
694
695 % Engage IAS after ATT engagement
696 signalbuilder('AFCS_Model/SignalIAS','set','Signal 1','Group
      1',[0 1+timeStep 1+timeStep 1.5 1.5 10],[0 0 1 1 0 0])
697
698 TunedBlocks = {'AFCS_Model/UpperModes/DeltaspstThetaIAS/
      Controller'};
699 AnalysisPoints = {'AFCS_Model/Demux/11'; ...
700                   'AFCS_Model/UpperModes/DeltaspstThetaIAS/
                        SetpointFr/1'};
701
702 OperatingPoints = 1+5*timeStep;  % linearization point
703 % Specify the custom options
704 Options = slTunerOptions('AreParamsTunable',false);
705 % Create the slTuner object
706 CL0 = slTuner('AFCS_Model',TunedBlocks,AnalysisPoints,
      OperatingPoints,Options);
707
708 % Set the parameterization of the tuned block
709 AFCS_Model_IAS_Controller = tunablePID2('
      AFCS_Model_IAS_Controller','PID');
710 AFCS_Model_IAS_Controller.Kp.Value = IASKp;
711 AFCS_Model_IAS_Controller.Kp.Minimum = TUNIASKpm;
```

```
712 AFCS_Model_IAS_Controller.Kp.Maximum = TUNIASKpM;
713 AFCS_Model_IAS_Controller.b.Value = IASb;
714 AFCS_Model_IAS_Controller.b.Free = 1;
715 AFCS_Model_IAS_Controller.b.Minimum = TUNIASbm;
716 AFCS_Model_IAS_Controller.b.Maximum = TUNIASbM;
717 AFCS_Model_IAS_Controller.Ki.Value = TUNIASKi;
718 AFCS_Model_IAS_Controller.Ki.Minimum = TUNIASKim;
719 AFCS_Model_IAS_Controller.Ki.Maximum = TUNIASKiM;
720 AFCS_Model_IAS_Controller.Kd.Value = TUNIASKd;
721 AFCS_Model_IAS_Controller.Kd.Minimum = TUNIASKdm;
722 AFCS_Model_IAS_Controller.Kd.Maximum = TUNIASKdM;
723 AFCS_Model_IAS_Controller.Tf.Value = TUNIASTf;
724 AFCS_Model_IAS_Controller.Tf.Minimum = TUNIASTfm;
725 AFCS_Model_IAS_Controller.Tf.Maximum = TUNIASTfM;
726 AFCS_Model_IAS_Controller.c.Value = IASc;
727 AFCS_Model_IAS_Controller.c.Free = 1;
728 AFCS_Model_IAS_Controller.c.Minimum = TUNIAScm;
729 AFCS_Model_IAS_Controller.c.Maximum = TUNIAScM;
730 setBlockParam(CL0,'AFCS_Model/UpperModes/DeltaspstThetaIAS/
    Controller',AFCS_Model_IAS_Controller);
731
732 %% Create tuning goal to shape how the closed-loop system
    responds to a specific input signal
733 % Inputs and outputs
734 Inputs = {'AFCS_Model/UpperModes/DeltaspstThetaIAS/SetpointFr
    /1'};
735 Outputs = {'AFCS_Model/Demux/11'};
736 % Tuning goal specifications
737 Tau = 2.5; % Time constant
738 Overshoot = 10; % Overshoot (%)
739 % Create tuning goal for step tracking
740 StepTrackingGoalIAS = TuningGoal.StepTracking(Inputs,Outputs,
    Tau,Overshoot);
741 StepTrackingGoalIAS.Name = 'StepTrackingGoalIAS'; % Tuning
    goal name
742
```

```
743 %% Create tuning goal to enforce specific gain and phase
       margins
744 % Feedback loop locations
745 Locations = {'AFCS_Model/Demux/11'};
746 % Tuning goal specifications
747 GainMargin = 10; % Required minimum gain margin
748 PhaseMargin = 10; % Required minimum phase margin
749 % Create tuning goal for margins
750 MarginsGoalIAS = TuningGoal.Margins(Locations,GainMargin,
       PhaseMargin);
751 MarginsGoalIAS.Name = 'MarginsGoalIAS'; % Tuning goal name
752
753 %% Create option set for systune command
754 Options = systuneOptions();
755 Options.Display = 'final'; % Tuning display level ('final', '
       sub', 'iter', 'off')
756 Options.RandomStart = 10; % Number of randomized starts
757
758 %% Set soft and hard goals
759 SoftGoals = [ ];
760 HardGoals = [ StepTrackingGoalIAS, MarginsGoalIAS];
761
762 %% Tune the parameters with soft and hard goals
763 [CL1,~,~,~] = systune(CL0,SoftGoals,HardGoals,Options);
764
765 AP = getTunedValue(CL1) ;
766
767 IASKp    = AP.AFCS_Model_IAS_Controller.Kp ;
768 IASb     = AP.AFCS_Model_IAS_Controller.b ;
769 IASTi    = AP.AFCS_Model_IAS_Controller.Kp/AP.
       AFCS_Model_IAS_Controller.Ki ;
770 IASTd    = AP.AFCS_Model_IAS_Controller.Kd/AP.
       AFCS_Model_IAS_Controller.Kp ;
771 IASN     = AP.AFCS_Model_IAS_Controller.Kd/(AP.
       AFCS_Model_IAS_Controller.Kp * AP.
       AFCS_Model_IAS_Controller.Tf) ;
772 IASc     = AP.AFCS_Model_IAS_Controller.c ;
```

```
773
774 %% View tuning results
775 figure ;
776 viewGoal(HardGoals(1),CL1) ;
777 figure ;
778 viewGoal(HardGoals(2),CL1) ;
779
780 %% Restore initial configuration
781 signalbuilder('AFCS_Model/SignalAP','set','Signal 1','Group 1
       ',[0 1 1 2 2 10],[0 0 0 0 0 0])
782 signalbuilder('AFCS_Model/SignalIAS','set','Signal 1','Group
       1',[0 1 1 2 2 10],[0 0 0 0 0 0])
783
784 %% ----- TUNING ALTCollective (note that Ias >
       VerticalProtectionIas; then ALT Mode would be engaged on
       the Pitch Axis. However, if IAS Mode is already engaged on
        Pitch Axis, ALT Mode works on Collective Axis) -----
785
786 %% Create system data with slTuner interface
787
788 % Engage ATT
789 signalbuilder('AFCS_Model/SignalAP','set','Signal 1','Group 1
       ',[0 timeStep timeStep 0.5 0.5 10],[0 0 1 1 0 0])
790
791 % Engage IAS after ATT engagement
792 signalbuilder('AFCS_Model/SignalIAS','set','Signal 1','Group
       1',[0 1+timeStep 1+timeStep 1.5 1.5 10],[0 0 1 1 0 0])
793
794 % Engage ALT after IAS engagement
795 signalbuilder('AFCS_Model/SignalALT','set','Signal 1','Group
       1',[0 1.1+timeStep 1.1+timeStep 1.5 1.5 10],[0 0 1 1 0 0])
796
797 TunedBlocks = {'AFCS_Model/UpperModes/DeltaCommandCollective/
       Controller'};
798 AnalysisPoints = {'AFCS_Model/Demux/12'; ...
799                   'AFCS_Model/UpperModes/
                        DeltaCommandCollective/SetpointFr/1'};
```

```
800
801 OperatingPoints = 1.1+5*timeStep;  % linearization point
802 % Specify the custom options
803 Options = slTunerOptions('AreParamsTunable',false);
804 % Create the slTuner object
805 CL0 = slTuner('AFCS_Model',TunedBlocks,AnalysisPoints,
      OperatingPoints,Options);
806
807 % Set the parameterization of the tuned block
808 AFCS_Model_ALTCollective_RHT__Controller = tunablePID2('
      AFCS_Model_ALTCollective_RHT__Controller','PID');
809 AFCS_Model_ALTCollective_RHT__Controller.Kp.Value =
      ALTCollective_RHT_Kp;
810 AFCS_Model_ALTCollective_RHT__Controller.Kp.Minimum =
      TUNALTCollective_RHT_Kpm;
811 AFCS_Model_ALTCollective_RHT__Controller.Kp.Maximum =
      TUNALTCollective_RHT_KpM;
812 AFCS_Model_ALTCollective_RHT__Controller.b.Value =
      ALTCollective_RHT_b;
813 AFCS_Model_ALTCollective_RHT__Controller.b.Free = 1;
814 AFCS_Model_ALTCollective_RHT__Controller.b.Minimum =
      TUNALTCollective_RHT_bm;
815 AFCS_Model_ALTCollective_RHT__Controller.b.Maximum =
      TUNALTCollective_RHT_bM;
816 AFCS_Model_ALTCollective_RHT__Controller.Ki.Value =
      TUNALTCollective_RHT_Ki;
817 AFCS_Model_ALTCollective_RHT__Controller.Ki.Minimum =
      TUNALTCollective_RHT_Kim;
818 AFCS_Model_ALTCollective_RHT__Controller.Ki.Maximum =
      TUNALTCollective_RHT_KiM;
819 AFCS_Model_ALTCollective_RHT__Controller.Kd.Value =
      TUNALTCollective_RHT_Kd;
820 AFCS_Model_ALTCollective_RHT__Controller.Kd.Minimum =
      TUNALTCollective_RHT_Kdm;
821 AFCS_Model_ALTCollective_RHT__Controller.Kd.Maximum =
      TUNALTCollective_RHT_KdM;
```

```
822 AFCS_Model_ALTCollective_RHT__Controller.Tf.Value =
       TUNALTCollective_RHT_Tf;
823 AFCS_Model_ALTCollective_RHT__Controller.Tf.Minimum =
       TUNALTCollective_RHT_Tfm;
824 AFCS_Model_ALTCollective_RHT__Controller.Tf.Maximum =
       TUNALTCollective_RHT_TfM;
825 AFCS_Model_ALTCollective_RHT__Controller.c.Value =
       ALTCollective_RHT_c;
826 AFCS_Model_ALTCollective_RHT__Controller.c.Free = 1;
827 AFCS_Model_ALTCollective_RHT__Controller.c.Minimum =
       TUNALTCollective_RHT_cm;
828 AFCS_Model_ALTCollective_RHT__Controller.c.Maximum =
       TUNALTCollective_RHT_cM;
829 setBlockParam(CL0,'AFCS_Model/UpperModes/
       DeltaCommandCollective/Controller',
       AFCS_Model_ALTCollective_RHT__Controller);
830
831 %% Create tuning goal to shape how the closed-loop system
       responds to a specific input signal
832 % Inputs and outputs
833 Inputs = {'AFCS_Model/UpperModes/DeltaCommandCollective/
       SetpointFr/1'};
834 Outputs = {'AFCS_Model/Demux/12'};
835 % Tuning goal specifications
836 Tau = .8; % Time constant
837 Overshoot = 5; % Overshoot (%)
838 % Create tuning goal for step tracking
839 StepTrackingGoalALTCollective_RHT_ = TuningGoal.StepTracking(
       Inputs,Outputs,Tau,Overshoot);
840 StepTrackingGoalALTCollective_RHT_.Name = '
       StepTrackingGoalALTCollective_RHT_'; % Tuning goal name
841
842 %% Create tuning goal to enforce specific gain and phase
       margins
843 % Feedback loop locations
844 Locations = {'AFCS_Model/Demux/12'};
845 % Tuning goal specifications
```

```
846 GainMargin = 15; % Required minimum gain margin
847 PhaseMargin = 40; % Required minimum phase margin
848 % Create tuning goal for margins
849 MarginsGoalALTCollective_RHT_ = TuningGoal.Margins(Locations,
       GainMargin,PhaseMargin);
850 MarginsGoalALTCollective_RHT_.Name = '
       MarginsGoalALTCollective_RHT_'; % Tuning goal name
851
852 %% Create option set for systune command
853 Options = systuneOptions();
854 Options.Display = 'final'; % Tuning display level ('final', '
       sub', 'iter', 'off')
855 Options.RandomStart = 10; % Number of randomized starts
856
857 %% Set soft and hard goals
858 SoftGoals = [ ];
859 HardGoals = [ StepTrackingGoalALTCollective_RHT_,
       MarginsGoalALTCollective_RHT_];
860
861 %% Tune the parameters with soft and hard goals
862 [CL1,~,~,~] = systune(CL0,SoftGoals,HardGoals,Options);
863
864 AP = getTunedValue(CL1) ;
865
866 ALTCollective_RHT_Kp    = AP.
       AFCS_Model_ALTCollective_RHT__Controller.Kp ;
867 ALTCollective_RHT_b     = AP.
       AFCS_Model_ALTCollective_RHT__Controller.b ;
868 ALTCollective_RHT_Ti    = AP.
       AFCS_Model_ALTCollective_RHT__Controller.Kp/AP.
       AFCS_Model_ALTCollective_RHT__Controller.Ki ;
869 ALTCollective_RHT_Td    = AP.
       AFCS_Model_ALTCollective_RHT__Controller.Kd/AP.
       AFCS_Model_ALTCollective_RHT__Controller.Kp ;
870 ALTCollective_RHT_N     = AP.
       AFCS_Model_ALTCollective_RHT__Controller.Kd/(AP.
       AFCS_Model_ALTCollective_RHT__Controller.Kp * AP.
```

```
   AFCS_Model_ALTCollective_RHT__Controller.Tf) ;
ALTCollective_RHT_c    = AP.
   AFCS_Model_ALTCollective_RHT__Controller.c ;

%% View tuning results
figure ;
viewGoal(HardGoals(1),CL1) ;
figure ;
viewGoal(HardGoals(2),CL1) ;

%% Restore initial configuration
signalbuilder('AFCS_Model/SignalAP','set','Signal 1','Group 1
   ',[0 1 1 2 2 10],[0 0 0 0 0 0])
signalbuilder('AFCS_Model/SignalIAS','set','Signal 1','Group
   1',[0 1 1 2 2 10],[0 0 0 0 0 0])
signalbuilder('AFCS_Model/SignalALT','set','Signal 1','Group
   1',[0 1 1 2 2 10],[0 0 0 0 0 0])

%% ----- TUNING ALTPitch -----

%% Create system data with slTuner interface

% Engage ATT
signalbuilder('AFCS_Model/SignalAP','set','Signal 1','Group 1
   ',[0 timeStep timeStep 0.5 0.5 10],[0 0 1 1 0 0])

% Engage ALT after ATT engagement
signalbuilder('AFCS_Model/SignalALT','set','Signal 1','Group
   1',[0 1.1+timeStep 1.1+timeStep 1.5 1.5 10],[0 0 1 1 0 0])

TunedBlocks = {'AFCS_Model/UpperModes/DeltaspstThetaALT/
   Controller'};
AnalysisPoints = {'AFCS_Model/Demux/12'; ...
                  'AFCS_Model/UpperModes/DeltaspstThetaALT/
                     SetpointFr/1'};

OperatingPoints = 1.1+5*timeStep ; % linearization point
```

```
899 % Specify the custom options
900 Options = slTunerOptions('AreParamsTunable',false);
901 % Create the slTuner object
902 CL0 = slTuner('AFCS_Model',TunedBlocks,AnalysisPoints,
       OperatingPoints,Options);
903
904 % Set the parameterization of the tuned block
905 AFCS_Model_ALTPitch_Controller = tunablePID2('
       AFCS_Model_ALTPitch_Controller','PID');
906 AFCS_Model_ALTPitch_Controller.Kp.Value = ALTPitchKp;
907 AFCS_Model_ALTPitch_Controller.Kp.Minimum = TUNALTPitchKpm;
908 AFCS_Model_ALTPitch_Controller.Kp.Maximum = TUNALTPitchKpM;
909 AFCS_Model_ALTPitch_Controller.b.Value = ALTPitchb;
910 AFCS_Model_ALTPitch_Controller.b.Free = 1;
911 AFCS_Model_ALTPitch_Controller.b.Minimum = TUNALTPitchbm;
912 AFCS_Model_ALTPitch_Controller.b.Maximum = TUNALTPitchbM;
913 AFCS_Model_ALTPitch_Controller.Ki.Value = TUNALTPitchKi;
914 AFCS_Model_ALTPitch_Controller.Ki.Minimum = TUNALTPitchKim;
915 AFCS_Model_ALTPitch_Controller.Ki.Maximum = TUNALTPitchKiM;
916 AFCS_Model_ALTPitch_Controller.Kd.Value = TUNALTPitchKd;
917 AFCS_Model_ALTPitch_Controller.Kd.Minimum = TUNALTPitchKdm;
918 AFCS_Model_ALTPitch_Controller.Kd.Maximum = TUNALTPitchKdM;
919 AFCS_Model_ALTPitch_Controller.Tf.Value = TUNALTPitchTf;
920 AFCS_Model_ALTPitch_Controller.Tf.Minimum = TUNALTPitchTfm;
921 AFCS_Model_ALTPitch_Controller.Tf.Maximum = TUNALTPitchTfM;
922 AFCS_Model_ALTPitch_Controller.c.Value = ALTPitchc;
923 AFCS_Model_ALTPitch_Controller.c.Free = 1;
924 AFCS_Model_ALTPitch_Controller.c.Minimum = TUNALTPitchcm;
925 AFCS_Model_ALTPitch_Controller.c.Maximum = TUNALTPitchcM;
926 setBlockParam(CL0,'AFCS_Model/UpperModes/DeltaspstThetaALT/
       Controller',AFCS_Model_ALTPitch_Controller);
927
928 %% Create tuning goal to shape how the closed-loop system
       responds to a specific input signal
929 % Inputs and outputs
930 Inputs = {'AFCS_Model/UpperModes/DeltaspstThetaALT/SetpointFr
       /1'};
```

```
931 Outputs = {'AFCS_Model/Demux/12'};
932 % Tuning goal specifications
933 Tau = 3; % Time constant
934 Overshoot = 5; % Overshoot (%)
935 % Create tuning goal for step tracking
936 StepTrackingGoalALTPitch = TuningGoal.StepTracking(Inputs,
       Outputs,Tau,Overshoot);
937 StepTrackingGoalALTPitch.Name = 'StepTrackingGoalALTPitch'; %
        Tuning goal name
938
939 %% Create tuning goal to enforce specific gain and phase
       margins
940 % Feedback loop locations
941 Locations = {'AFCS_Model/Demux/12'};
942 % Tuning goal specifications
943 GainMargin = 5; % Required minimum gain margin
944 PhaseMargin = 20; % Required minimum phase margin
945 % Create tuning goal for margins
946 MarginsGoalALTPitch = TuningGoal.Margins(Locations,GainMargin
       ,PhaseMargin);
947 MarginsGoalALTPitch.Name = 'MarginsGoalALTPitch'; % Tuning
       goal name
948
949 %% Create option set for systune command
950 Options = systuneOptions();
951 Options.Display = 'final'; % Tuning display level ('final', '
       sub', 'iter', 'off')
952 Options.RandomStart = 10; % Number of randomized starts
953
954 %% Set soft and hard goals
955 SoftGoals = [ ];
956 HardGoals = [ StepTrackingGoalALTPitch, MarginsGoalALTPitch];
957
958 %% Tune the parameters with soft and hard goals
959 [CL1,~,~,~] = systune(CL0,SoftGoals,HardGoals,Options);
960
961 AP = getTunedValue(CL1) ;
```

```
962
963 ALTPitchKp    = AP.AFCS_Model_ALTPitch_Controller.Kp ;
964 ALTPitchb     = AP.AFCS_Model_ALTPitch_Controller.b ;
965 ALTPitchTi    = AP.AFCS_Model_ALTPitch_Controller.Kp/AP.
       AFCS_Model_ALTPitch_Controller.Ki ;
966 ALTPitchTd    = AP.AFCS_Model_ALTPitch_Controller.Kd/AP.
       AFCS_Model_ALTPitch_Controller.Kp ;
967 ALTPitchN     = AP.AFCS_Model_ALTPitch_Controller.Kd/(AP.
       AFCS_Model_ALTPitch_Controller.Kp * AP.
       AFCS_Model_ALTPitch_Controller.Tf) ;
968 ALTPitchc     = AP.AFCS_Model_ALTPitch_Controller.c ;
969
970 %% View tuning results
971 figure ;
972 viewGoal(HardGoals(1),CL1) ;
973 figure ;
974 viewGoal(HardGoals(2),CL1) ;
975
976 %% Restore initial configuration
977 signalbuilder('AFCS_Model/SignalAP','set','Signal 1','Group 1
       ',[0 1 1 2 2 10],[0 0 0 0 0 0])
978 signalbuilder('AFCS_Model/SignalALT','set','Signal 1','Group
       1',[0 1 1 2 2 10],[0 0 0 0 0 0])
979
980
981 %% ----- TUNING HDGRoll -----
982
983 %% Create system data with slTuner interface
984
985 % Engage ATT
986 signalbuilder('AFCS_Model/SignalAP','set','Signal 1','Group 1
       ',[0 timeStep timeStep 0.5 0.5 10],[0 0 1 1 0 0])
987
988 % Engage HDG after ATT engagement
989 signalbuilder('AFCS_Model/SignalHDG','set','Signal 1','Group
       1',[0 1.1+timeStep 1.1+timeStep 1.5 1.5 10],[0 0 1 1 0 0])
990
```

```
991  TunedBlocks = {'AFCS_Model/UpperModes/DeltaspstPhiHDG/
        Controller'};
992  AnalysisPoints = {'AFCS_Model/Demux/6'; ...
993                    'AFCS_Model/UpperModes/DeltaspstPhiHDG/
                          SetpointFr/1'};
994
995  OperatingPoints = 1.1+5*timeStep;  % linearization point
996  % Specify the custom options
997  Options = slTunerOptions('AreParamsTunable',false);
998  % Create the slTuner object
999  CL0 = slTuner('AFCS_Model',TunedBlocks,AnalysisPoints,
        OperatingPoints,Options);
1000
1001 % Set the parameterization of the tuned block
1002 AFCS_Model_HDGRoll_Controller = tunablePID2('
        AFCS_Model_HDGRoll_Controller','PID');
1003 AFCS_Model_HDGRoll_Controller.Kp.Value = HDGRollKp;
1004 AFCS_Model_HDGRoll_Controller.Kp.Minimum = TUNHDGRollKpm;
1005 AFCS_Model_HDGRoll_Controller.Kp.Maximum = TUNHDGRollKpM;
1006 AFCS_Model_HDGRoll_Controller.b.Value = HDGRollb;
1007 AFCS_Model_HDGRoll_Controller.b.Free = 1;
1008 AFCS_Model_HDGRoll_Controller.b.Minimum = TUNHDGRollbm;
1009 AFCS_Model_HDGRoll_Controller.b.Maximum = TUNHDGRollbM;
1010 AFCS_Model_HDGRoll_Controller.Ki.Value = TUNHDGRollKi;
1011 AFCS_Model_HDGRoll_Controller.Ki.Minimum = TUNHDGRollKim;
1012 AFCS_Model_HDGRoll_Controller.Ki.Maximum = TUNHDGRollKiM;
1013 AFCS_Model_HDGRoll_Controller.Kd.Value = TUNHDGRollKd;
1014 AFCS_Model_HDGRoll_Controller.Kd.Minimum = TUNHDGRollKdm;
1015 AFCS_Model_HDGRoll_Controller.Kd.Maximum = TUNHDGRollKdM;
1016 AFCS_Model_HDGRoll_Controller.Tf.Value = TUNHDGRollTf;
1017 AFCS_Model_HDGRoll_Controller.Tf.Minimum = TUNHDGRollTfm;
1018 AFCS_Model_HDGRoll_Controller.Tf.Maximum = TUNHDGRollTfM;
1019 AFCS_Model_HDGRoll_Controller.c.Value = HDGRollc;
1020 AFCS_Model_HDGRoll_Controller.c.Free = 1;
1021 AFCS_Model_HDGRoll_Controller.c.Minimum = TUNHDGRollcm;
1022 AFCS_Model_HDGRoll_Controller.c.Maximum = TUNHDGRollcM;
```

```
1023 setBlockParam(CL0,'AFCS_Model/UpperModes/DeltaspstPhiHDG/
        Controller',AFCS_Model_HDGRoll_Controller);
1024
1025 %% Create tuning goal to shape how the closed-loop system
        responds to a specific input signal
1026 % Inputs and outputs
1027 Inputs = {'AFCS_Model/UpperModes/DeltaspstPhiHDG/SetpointFr/1
        '};
1028 Outputs = {'AFCS_Model/Demux/6'};
1029 % Tuning goal specifications
1030 Tau = 2; % Time constant
1031 Overshoot = 10; % Overshoot (%)
1032 % Create tuning goal for step tracking
1033 StepTrackingGoalHDGRoll = TuningGoal.StepTracking(Inputs,
        Outputs,Tau,Overshoot);
1034 StepTrackingGoalHDGRoll.Name = 'StepTrackingGoalHDGRoll'; %
        Tuning goal name
1035
1036 %% Create tuning goal to enforce specific gain and phase
        margins
1037 % Feedback loop locations
1038 Locations = {'AFCS_Model/Demux/6'};
1039 % Tuning goal specifications
1040 GainMargin = 5; % Required minimum gain margin
1041 PhaseMargin = 20; % Required minimum phase margin
1042 % Create tuning goal for margins
1043 MarginsGoalHDGRoll = TuningGoal.Margins(Locations,GainMargin,
        PhaseMargin);
1044 MarginsGoalHDGRoll.Name = 'MarginsGoalHDGRoll'; % Tuning goal
         name
1045
1046 %% Create option set for systune command
1047 Options = systuneOptions();
1048 Options.Display = 'final'; % Tuning display level ('final', '
        sub', 'iter', 'off')
1049 Options.RandomStart = 10; % Number of randomized starts
1050
```

```
%% Set soft and hard goals
SoftGoals = [ ];
HardGoals = [ StepTrackingGoalHDGRoll, MarginsGoalHDGRoll];

%% Tune the parameters with soft and hard goals
[CL1,~,~,~] = systune(CL0,SoftGoals,HardGoals,Options);

AP = getTunedValue(CL1) ;

HDGRollKp    = AP.AFCS_Model_HDGRoll_Controller.Kp ;
HDGRollb     = AP.AFCS_Model_HDGRoll_Controller.b ;
HDGRollTi    = AP.AFCS_Model_HDGRoll_Controller.Kp/AP.
   AFCS_Model_HDGRoll_Controller.Ki ;
HDGRollTd    = AP.AFCS_Model_HDGRoll_Controller.Kd/AP.
   AFCS_Model_HDGRoll_Controller.Kp ;
HDGRollN     = AP.AFCS_Model_HDGRoll_Controller.Kd/(AP.
   AFCS_Model_HDGRoll_Controller.Kp * AP.
   AFCS_Model_HDGRoll_Controller.Tf) ;
HDGRollc     = AP.AFCS_Model_HDGRoll_Controller.c ;

%% View tuning results
figure ;
viewGoal(HardGoals(1),CL1) ;
figure ;
viewGoal(HardGoals(2),CL1) ;

%% Restore initial configuration
signalbuilder('AFCS_Model/SignalAP','set','Signal 1','Group 1
   ',[0 1 1 2 2 10],[0 0 0 0 0 0])
signalbuilder('AFCS_Model/SignalHDG','set','Signal 1','Group
   1',[0 1 1 2 2 10],[0 0 0 0 0 0])


%% ----- SAVE PARAMETERS FOR VALIDATION MODEL -----
% Save the fixed and tuned parameters to a file named "
   ParametersList.mat"

```

```
1081 save('ParametersList.mat', ...
1082           'kt_to_ms', ...
1083      'ft_to_m', ...
1084      'deg2rad', ...
1085      'timeStep', ...
1086      'VPIas', ...
1087      'Vne', ...
1088      'MinIasIAS', ...
1089      'MinIasTC', ...
1090      'MinRadaltRHT', ...
1091      'MaxRadaltRHT', ...
1092      'MinBaraltALT', ...
1093      'MaxIasHOV', ...
1094      'MaxGsxHOV', ...
1095      'MaxGsyHOV', ...
1096      'UpperSatPitch', ...
1097      'LowerSatPitch', ...
1098      'UpperSatRoll', ...
1099      'LowerSatRoll', ...
1100      'UpperSatYaw', ...
1101      'LowerSatYaw', ...
1102      'UpperSatCollective', ...
1103      'LowerSatCollective', ...
1104      'LowerSatTheta', ...
1105      'UpperSatTheta', ...
1106      'LowerSatPhi', ...
1107      'UpperSatPhi', ...
1108      'LowerSatPsi', ...
1109      'UpperSatPsi', ...
1110      'ATTPitchMType', ...
1111      'ATTPitchCType', ...
1112      'beepTrimATTPitchRate', ...
1113      'ATTRollMType', ...
1114      'ATTRollCType', ...
1115      'beepTrimATTRollRate', ...
1116      'ATTYawMType', ...
1117      'ATTYawCType', ...
```

```
1118        'beepTrimATTYawRate', ...
1119        'TCCType', ...
1120        'MinPhiTC', ...
1121        'IASMType', ...
1122        'IASCType', ...
1123        'beepTrimIASRate', ...
1124        'HDGRollMType', ...
1125        'HDGRollCType', ...
1126        'beepTrimHDGRollRate', ...
1127        'ALTCollective_RHT_MType', ...
1128        'ALTPitchMType', ...
1129        'ALTCollective_RHT_CType', ...
1130        'ALTPitchCType', ...
1131        'beepTrimRHTRate', ...
1132        'beepTrimALTRate', ...
1133        'HOVPitchMType', ...
1134        'HOVRollMType', ...
1135        'HOVPitchCType', ...
1136        'HOVRollCType', ...
1137        'beepTrimHOVPitchRate', ...
1138        'beepTrimHOVRollRate', ...
1139        'sasPitchP', ...
1140        'sasPitchI', ...
1141        'sasRollP', ...
1142        'sasRollI', ...
1143        'sasYawP', ...
1144        'sasYawI', ...
1145        'ATTPitchKp', ...
1146        'ATTPitchb', ...
1147        'ATTPitchTi', ...
1148        'ATTPitchTd', ...
1149        'ATTPitchN', ...
1150        'ATTPitchc', ...
1151        'ATTRollKp', ...
1152        'ATTRollb', ...
1153        'ATTRollTi', ...
1154        'ATTRollTd', ...
```

```
1155        'ATTRollN', ...
1156        'ATTRollc', ...
1157        'ATTYawKp', ...
1158        'ATTYawb', ...
1159        'ATTYawTi', ...
1160        'ATTYawTd', ...
1161        'ATTYawN', ...
1162        'ATTYawc', ...
1163        'TCKp', ...
1164        'TCb', ...
1165        'TCTi', ...
1166        'TCTd', ...
1167        'TCN', ...
1168        'TCc', ...
1169        'ALTCollective_RHT_Kp', ...
1170        'ALTCollective_RHT_b', ...
1171        'ALTCollective_RHT_Ti', ...
1172        'ALTCollective_RHT_Td', ...
1173        'ALTCollective_RHT_N', ...
1174        'ALTCollective_RHT_c', ...
1175        'IASKp', ...
1176        'IASb', ...
1177        'IASTi', ...
1178        'IASTd', ...
1179        'IASN', ...
1180        'IASc', ...
1181        'ALTPitchKp', ...
1182        'ALTPitchb', ...
1183        'ALTPitchTi', ...
1184        'ALTPitchTd', ...
1185        'ALTPitchN', ...
1186        'ALTPitchc', ...
1187        'HDGRollKp', ...
1188        'HDGRollb', ...
1189        'HDGRollTi', ...
1190        'HDGRollTd', ...
1191        'HDGRollN', ...
```

```
1192      'HDGRollc', ...
1193      'HOVPitchKp', ...
1194      'HOVPitchb', ...
1195      'HOVPitchTi', ...
1196      'HOVPitchTd', ...
1197      'HOVPitchN', ...
1198      'HOVPitchc', ...
1199      'HOVRollKp', ...
1200      'HOVRollb', ...
1201      'HOVRollTi', ...
1202      'HOVRollTd', ...
1203      'HOVRollN', ...
1204      'HOVRollc') ;
1205
1206 %% ----- SAVE PARAMETERS FOR C++ MODEL -----
1207 % Collect the parameter used for the C++ Model
1208
1209 FILE = fopen('ParametersList.cpp', 'w') ;
1210 fprintf(FILE, '#include ".\\Simulink\\rtwtypes.h"\n\n') ;
1211 fclose(FILE) ;
1212
1213 FILE = fopen('ParametersList.cpp', 'a') ;
1214
1215 formatSpec = 'real_T %s = %.6f ;\n\n' ;
1216
1217 names = ['kt_to_ms                ';
1218          'ft_to_m                 ';
1219          'deg2rad                 ';
1220          'timeStep                ';
1221          'VPIas                   ';
1222          'Vne                     ';
1223          'MinIasIAS               ';
1224          'MinIasTC                ';
1225          'MinRadaltRHT            ';
1226          'MaxRadaltRHT            ';
1227          'MinBaraltALT            ';
1228          'MaxIasHOV               ';
```

```
1229            'MaxGsxHOV                ';
1230            'MaxGsyHOV                ';
1231            'UpperSatPitch            ';
1232            'LowerSatPitch            ';
1233            'UpperSatRoll             ';
1234            'LowerSatRoll             ';
1235            'UpperSatYaw              ';
1236            'LowerSatYaw              ';
1237            'UpperSatCollective       ';
1238            'LowerSatCollective       ';
1239            'LowerSatTheta            ';
1240            'UpperSatTheta            ';
1241            'LowerSatPhi              ';
1242            'UpperSatPhi              ';
1243            'LowerSatPsi              ';
1244            'UpperSatPsi              ';
1245            'ATTPitchMType            ';
1246            'ATTPitchCType            ';
1247            'beepTrimATTPitchRate     ';
1248            'ATTRollMType             ';
1249            'ATTRollCType             ';
1250            'beepTrimATTRollRate      ';
1251            'ATTYawMType              ';
1252            'ATTYawCType              ';
1253            'beepTrimATTYawRate       ';
1254            'TCCType                  ';
1255            'MinPhiTC                 ';
1256            'IASMType                 ';
1257            'IASCType                 ';
1258            'beepTrimIASRate          ';
1259            'HDGRollMType             ';
1260            'HDGRollCType             ';
1261            'beepTrimHDGRollRate      ';
1262            'ALTCollective_RHT_MType';
1263            'ALTPitchMType            ';
1264            'ALTCollective_RHT_CType';
1265            'ALTPitchCType            ';
```

```
1266            'beepTrimRHTRate          ';
1267            'beepTrimALTRate          ';
1268            'HOVPitchMType            ';
1269            'HOVRollMType             ';
1270            'HOVPitchCType            ';
1271            'HOVRollCType             ';
1272            'beepTrimHOVPitchRate     ';
1273            'beepTrimHOVRollRate      ';
1274            'sasPitchP                ';
1275            'sasPitchI                ';
1276            'sasRollP                 ';
1277            'sasRollI                 ';
1278            'sasYawP                  ';
1279            'sasYawI                  ';
1280            'ATTPitchKp               ';
1281            'ATTPitchb                ';
1282            'ATTPitchTi               ';
1283            'ATTPitchTd               ';
1284            'ATTPitchN                ';
1285            'ATTPitchc                ';
1286            'ATTRollKp                ';
1287            'ATTRollb                 ';
1288            'ATTRollTi                ';
1289            'ATTRollTd                ';
1290            'ATTRollN                 ';
1291            'ATTRollc                 ';
1292            'ATTYawKp                 ';
1293            'ATTYawb                  ';
1294            'ATTYawTi                 ';
1295            'ATTYawTd                 ';
1296            'ATTYawN                  ';
1297            'ATTYawc                  ';
1298            'TCKp                     ';
1299            'TCb                      ';
1300            'TCTi                     ';
1301            'TCTd                     ';
1302            'TCN                      ';
```

```
1303            'TCc                        ';
1304            'ALTCollective_RHT_Kp       ';
1305            'ALTCollective_RHT_b        ';
1306            'ALTCollective_RHT_Ti       ';
1307            'ALTCollective_RHT_Td       ';
1308            'ALTCollective_RHT_N        ';
1309            'ALTCollective_RHT_c        ';
1310            'IASKp                      ';
1311            'IASb                       ';
1312            'IASTi                      ';
1313            'IASTd                      ';
1314            'IASN                       ';
1315            'IASc                       ';
1316            'ALTPitchKp                 ';
1317            'ALTPitchb                  ';
1318            'ALTPitchTi                 ';
1319            'ALTPitchTd                 ';
1320            'ALTPitchN                  ';
1321            'ALTPitchc                  ';
1322            'HDGRollKp                  ';
1323            'HDGRollb                   ';
1324            'HDGRollTi                  ';
1325            'HDGRollTd                  ';
1326            'HDGRollN                   ';
1327            'HDGRollc                   ';
1328            'HOVPitchKp                 ';
1329            'HOVPitchb                  ';
1330            'HOVPitchTi                 ';
1331            'HOVPitchTd                 ';
1332            'HOVPitchN                  ';
1333            'HOVPitchc                  ';
1334            'HOVRollKp                  ';
1335            'HOVRollb                   ';
1336            'HOVRollTi                  ';
1337            'HOVRollTd                  ';
1338            'HOVRollN                   ';
1339            'HOVRollc                   ' ] ;
```

```
1340
1341 values = [kt_to_ms                ; ...
1342           ft_to_m                 ; ...
1343           deg2rad                 ; ...
1344           timeStep                ; ...
1345           VPIas                   ; ...
1346           Vne                     ; ...
1347           MinIasIAS               ; ...
1348           MinIasTC                ; ...
1349           MinRadaltRHT            ; ...
1350           MaxRadaltRHT            ; ...
1351           MinBaraltALT            ; ...
1352           MaxIasHOV               ; ...
1353           MaxGsxHOV               ; ...
1354           MaxGsyHOV               ; ...
1355           UpperSatPitch           ; ...
1356           LowerSatPitch           ; ...
1357           UpperSatRoll            ; ...
1358           LowerSatRoll            ; ...
1359           UpperSatYaw             ; ...
1360           LowerSatYaw             ; ...
1361           UpperSatCollective      ; ...
1362           LowerSatCollective      ; ...
1363           LowerSatTheta           ; ...
1364           UpperSatTheta           ; ...
1365           LowerSatPhi             ; ...
1366           UpperSatPhi             ; ...
1367           LowerSatPsi             ; ...
1368           UpperSatPsi             ; ...
1369           ATTPitchMType           ; ...
1370           ATTPitchCType           ; ...
1371           beepTrimATTPitchRate    ; ...
1372           ATTRollMType            ; ...
1373           ATTRollCType            ; ...
1374           beepTrimATTRollRate     ; ...
1375           ATTYawMType             ; ...
1376           ATTYawCType             ; ...
```

```
1377            beepTrimATTYawRate      ; ...
1378            TCCType                 ; ...
1379            MinPhiTC                ; ...
1380            IASMType                ; ...
1381            IASCType                ; ...
1382            beepTrimIASRate         ; ...
1383            HDGRollMType            ; ...
1384            HDGRollCType            ; ...
1385            beepTrimHDGRollRate     ; ...
1386            ALTCollective_RHT_MType; ...
1387            ALTPitchMType           ; ...
1388            ALTCollective_RHT_CType; ...
1389            ALTPitchCType           ; ...
1390            beepTrimRHTRate         ; ...
1391            beepTrimALTRate         ; ...
1392            HOVPitchMType           ; ...
1393            HOVRollMType            ; ...
1394            HOVPitchCType           ; ...
1395            HOVRollCType            ; ...
1396            beepTrimHOVPitchRate    ; ...
1397            beepTrimHOVRollRate     ; ...
1398            sasPitchP               ; ...
1399            sasPitchI               ; ...
1400            sasRollP                ; ...
1401            sasRollI                ; ...
1402            sasYawP                 ; ...
1403            sasYawI                 ; ...
1404            ATTPitchKp              ; ...
1405            ATTPitchb               ; ...
1406            ATTPitchTi              ; ...
1407            ATTPitchTd              ; ...
1408            ATTPitchN               ; ...
1409            ATTPitchc               ; ...
1410            ATTRollKp               ; ...
1411            ATTRollb                ; ...
1412            ATTRollTi               ; ...
1413            ATTRollTd               ; ...
```

```
1414            ATTRollN                ; ...
1415            ATTRollc                ; ...
1416            ATTYawKp                ; ...
1417            ATTYawb                 ; ...
1418            ATTYawTi                ; ...
1419            ATTYawTd                ; ...
1420            ATTYawN                 ; ...
1421            ATTYawc                 ; ...
1422            TCKp                    ; ...
1423            TCb                     ; ...
1424            TCTi                    ; ...
1425            TCTd                    ; ...
1426            TCN                     ; ...
1427            TCc                     ; ...
1428            ALTCollective_RHT_Kp    ; ...
1429            ALTCollective_RHT_b     ; ...
1430            ALTCollective_RHT_Ti    ; ...
1431            ALTCollective_RHT_Td    ; ...
1432            ALTCollective_RHT_N     ; ...
1433            ALTCollective_RHT_c     ; ...
1434            IASKp                   ; ...
1435            IASb                    ; ...
1436            IASTi                   ; ...
1437            IASTd                   ; ...
1438            IASN                    ; ...
1439            IASc                    ; ...
1440            ALTPitchKp              ; ...
1441            ALTPitchb               ; ...
1442            ALTPitchTi              ; ...
1443            ALTPitchTd              ; ...
1444            ALTPitchN               ; ...
1445            ALTPitchc               ; ...
1446            HDGRollKp               ; ...
1447            HDGRollb                ; ...
1448            HDGRollTi               ; ...
1449            HDGRollTd               ; ...
1450            HDGRollN                ; ...
```

```
1451            HDGRollc                ; ...
1452            HOVPitchKp              ; ...
1453            HOVPitchb               ; ...
1454            HOVPitchTi              ; ...
1455            HOVPitchTd              ; ...
1456            HOVPitchN               ; ...
1457            HOVPitchc               ; ...
1458            HOVRollKp               ; ...
1459            HOVRollb                ; ...
1460            HOVRollTi               ; ...
1461            HOVRollTd               ; ...
1462            HOVRollN                ; ...
1463            HOVRollc                ;] ;
1464
1465 for ii = 1:length(values)
1466     fprintf(FILE, formatSpec, names(ii,:), values(ii,:)) ;
1467 end
1468
1469 fclose('all') ;
```