Fall 12-2020

Onboard Autonomous Controllability Assessment for Fixed Wing sUAVs

Brian Edward Duvall
Old Dominion University, bduva002@odu.edu

ONBOARD AUTONOMOUS CONTROLLABILITY ASSESSMENT FOR FIXED WING sUAVs

by

Brian Edward Duvall
B.S. May 2014, Old Dominion University
M.S. August 2016, Old Dominion University

A Dissertation Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

AEROSPACE ENGINEERING

OLD DOMINION UNIVERSITY
December 2020

Approved by:

Drew Landman (Director)

Thomas Alberts (Member)

Loc Tran (Member)

Gene Hou (Member)

ABSTRACT

ONBOARD AUTONOMOUS CONTROLLABILITY ASSESSMENT FOR FIXED WING sUAVs

Brian Edward Duvall
Old Dominion University, 2020
Director: Dr. Drew Landman

Traditionally fixed-wing small Unmanned Arial Vehicles (sUAV) are flown while in direct line of sight with commands from a remote operator. However, this is changing with the increased popularity and ready availability of low-cost flight controllers. Flight controllers provide fixed-wing sUAVs with functions that either minimize or eliminate the need for a remote operator. Since the remote operator is no longer controlling the sUAV, it is impossible to determine if the fixed-wing sUAV has proper control authority. In this work, a controllability detection system was designed, built, and flight-tested using COTS hardware. The method features in-situ measurement and analysis of the angular velocity response for the roll, pitch, and yaw axis using a Multi-Input Multi-Output (MIMO) Autoregressive with Exogenous input (ARX) modeling technique. The method is structured so that no prior knowledge of the airplane dimensions, control surface deflection angles, mass, or moment of inertia are required. The diagnostic is performed in flight with no post-processing so that controllability may be assessed during normal operations. This diagnostic works by comparison of baseline healthy control responses to current responses using statistical analysis. The outcome of this work shows that this is a viable way to check for degraded control authority.

This dissertation is dedicated to my parents Michael and Sharon Duvall and to my sister Kimberly Duvall.

ACKNOWLEDGMENTS

There are many people I would like to thank, which made this work possible. First, I would like to thank Dr. Drew Landman, my advisor, for reviewing this document and his dedication to helping me with all my research endeavors since 2014. Dr. Thomas Alberts has also been a great help in providing knowledge on control theory and debugging problems during flight testing. Dr. Gene Hou has provided support in understanding uncertainty as well as general overall project support. I would also like to thank Mr. Engin Baris, a good friend and fellow Ph.D. candidate, for the ability to talk about problems that arose during this work. Additionally, this work has benefitted from ties made at NASA Langley. I would like to thank Dr. Danette Allen for supporting me in my Ph.D. endeavor by being flexible in my work hours at the Autonomy Incubator. Also, Dr. Allen's influence benefitted this project greatly in that it allowed me to learn from fellow coworkers about Python programming and small companion computers in general. This said, I would like to thank the entire NASA Langley Autonomy Incubator team. Specifically, Dr. Loc Tran who was an advisor to this project and was always willing to help when problems arose. Additionally, I would like to thank the entire NASA Langley Aeronautics Systems Engineering Branch. Specifically, I would like to thank Mr. Robert M^{c}Swain my mentor, Mr. Ray Rhew, Mr. Lou Glaab, and Mr. Dave North. They were always there to answer questions and provide help when needed. I would also like to thank Mr. Justin Lisee for drawing the wiring schematic of the SIG EdgeTRA and Mr. Greg Howland for helping me with the Bifilar pendulum rig. I would also like to thank Mr. Chris Meek, Mr. Neil Coffey, Mr. Dave Bradley, Mr. Dave Hare, and Mr. Chester Dolph for brainstorming ideas with me. Lastly, I would like to thank my parents immensely for helping me with the hundreds of hours of flight testing required for this work and supporting me over the years.

NOMENCLATURE

N_f	Number of factors
N_{max}	Number of dimensions
T_L	Load torque
V	DC motor input voltage
θ	Rotor position
I	Armature current
n	Gear ratio
β	Viscous friction coefficient
J	Rotor inertia
R	Armature resistance
L_c	Armature inductance
K_v	DC motor speed constant
θ_{REF}	Desired servo position
X	Force in the x-direction
Y	Force in the y-direction
Z	Force in the z-direction

p	Angular velocity about the x-axis
q	Angular velocity about the y-axis
r	Angular velocity about the z-axis
$\dot{p}$	Angular acceleration about the x-axis
$\dot{q}$	Angular acceleration about the y-axis
$\dot{r}$	Angular acceleration about the z-axis
u	Velocity in the x-direction
v	Velocity in the y-direction
w	Velocity in the z-direction
$\dot{u}$	Acceleration in the x-direction
$\dot{v}$	Acceleration in the y-direction
$\dot{w}$	Acceleration in the z-direction
T	Thrust
g	gravity
m	Mass
L	Moment about the x-axis
M	Moment about the y-axis
N	Moment about the z-axis

ARX	Autoregressive with exogenous input
u_{input}	System input data
y_{output}	System output data
a_{n_a}	System output coefficients to be identified
b_{n_b}	System input coefficients to be identified
n_a	Order of system output coefficients to be identified
n_b	Order of system input coefficients to be identified
SISO	Single-Input Single-Output
MIMO	Multi-Input Multi-Output
$G(z)$	System transfer function in the z domain
$LHS(z)$	Left-hand side in the z domain
$RHS(z)$	Right-hand side in the z domain
y_{roll_rate}	ARX model estimate of the roll rate
y_{pitch_rate}	ARX model estimate of the pitch rate
y_{yaw_rate}	ARX model estimate of the yaw rate
TIC	Theil Inequality Coefficient
PWM	Pulse width modulation
$\hat{y}$	Predicted value at x0

t_s	Student's critical value
n_s	Number of runs in the design
S	Estimated standard deviation
W	Weight of the aircraft
A	Bifilar string separation
t	Period of oscillation
$L_{Bifilar}$	Length of bifilar strings
USB	Universal serial bus
GPIO	General purpose input/output
RC	Remote control
INS	Inertial Navigation System
GCS	Ground Control Station
AIS	Artificial Immune System
PIC	Pilot In Command
MIE	Manual Input Event
ATE	Automatic Trigger Event
PI	Prediction Interval
SID	System Identification

TABLE OF CONTENTS

Page

LIST OF TABLES ... xiv
LIST OF FIGURES ... xvi

Chapter

1. INTRODUCTION ... 1
  1.1 PROBLEM STATEMENT ... 4

2. LITERATURE SEARCH ... 6
  2.1 CURRENT HEALTH DIAGNOSTIC METHODS ... 6
    2.1.1 BIOMIMETIC METHOD AIS NEGATIVE SELECTION ... 6
    2.1.1.1 SUMMARY OF AIS NEGATIVE SELECTION ... 12
    2.1.2 SEMI-AUTONOMOUS sUAV AUTOPILOT LOGIC DESIGN METHOD ... 13
    2.1.3 SERVO FAULT DETECTION MODELING CURRENT FLOW METHOD ... 21
  2.2 DISCUSSION OF CURRENT DIAGNOSTIC METHODS AND RELATED WORK ... 24

3. METHOD ... 28
  3.1 OVERVIEW OF THE DEVELOPED METHOD ... 28
  3.2 SYSTEM IDENTIFICATION AND APPLICATION TO HEALTH DIAGNOSTICS ... 31
  3.2 AUTOREGRESSIVE MODELING TECHNIQUE ... 32
  3.3 LACK OF CONTROLABITY DETECTION ... 35
  3.4 MODES OF FAILURE FOR LACK OF CONTROLABITY CHECK ... 39

4. HARDWARE AND SOFTWARE ... 41
  4.1 HARDWARE ... 41
  4.2 SOFTWARE AND FIRMWARE ... 45

5. EXPERIMENTAL CONFIGURATION AND OPERATION ... 52
  5.1 MANUAL CONTROL ... 53
  5.2 AUTOMATIC CONTROL ... 55
  5.3 FAILURE MODES ... 58

Chapter Page

6. DATA COLLECTION 60
6.1 SENSORS AND DATA COLLECTED 60
6.2 MAVLINK MESSAGES 63
6.3 RASPBERRY PI FLIGHT COMPUTER 65
6.4 START AND STOP OF DATA COLLECTION 67

7. DATA PROCESSING 69

8. RESULTS 73
8.1 MANUAL RC CONTROL MODEL BUILDING 73
8.2 AUTOMATIC CONTROL MODEL BUILDING 83
8.3 LACK OF CONTROLABILTY DETECTION 93
8.3.1 AILERON TWO STUCK NEUTRAL 95
8.3.2 AILERON TWO WITH LIMITED TRAVEL 98
8.3.3 ELEVATOR TWO STUCK NEUTRAL 99
8.3.4 ELEVATOR TWO WITH LIMITED TRAVEL 100
8.3.5 AILERON TWO STUCK NEUTRAL AND ELEVATOR TWO LIMITED TRAVEL 101
8.3.6 AILERON TWO LIMITED TRAVEL AND ELEVATOR TWO STUCK NEUTRAL 103
8.3.7 AILERON TWO AND ELEVATOR TWO STUCK NEUTRAL 105
8.3.8 AILERON TWO AND ELEVATOR TWO WITH LIMITED TRAVEL 107
8.3.9 RUDDER LIMITED TRAVEL 109
8.3.10 RUDDER LIMITED TRAVEL AND ELEVATOR TWO LIMITED TRAVEL ... 110

9. DISCUSSION 113
9.1 MOMENT OF INERTIA STUDY 113
9.2 XFLR5 DYNAMIC STABILITY 117

10. CONCLUSIONS AND FUTURE WORK 124

REFERENCES 127

Page

APPENDIX A 133
APPENDIX B 134
APPENDIX C 151
APPENDIX D 168
APPENDIX E 176

VITA 180

LIST OF TABLES

Table Page

1-Post-processed selection of projections 9

2-Pre-flight parameter checks 15

3-EFSM conditions 18

4-Event definitions 19

5-Category of fault conditions for a servo actuator 24

6-Controllability diagnostic requirements 28

7-MIMO Transfer function design 34

8-Failure modes tested for lack of controllability detection 40

9- Physical properties of EdgeTRA 42

10-Baseline hardware use to fly EdgeTRA 42

11-Additional hardware used for modeling and safety during failure modes 43

12-RAW_IMU message contents 48

13-Available sensors for controllability diagnostic 60

14-Data collected from Cube Orange 61

15-List of MAVLink messages used to receive and transmit information 63

16-Attributes of MAV_DATA_STREAM 65

17-Identified ARX transfer function model for roll, pitch, and yaw rates 76

18- Identified ARX transfer function model for roll, pitch, and yaw rates for auto control........ 86

19- Comparison of manual vs. automatic standard deviation of TIC values 91

20-Mass and experimentally determined MOI properties of the EdgeTRA 115

Table Page

21- Eigenvalues from XFLR for EdgeTRA 119

LIST OF FIGURES

Figure Page

1-List of features to be recorded 7

2-Two-dimensional projection of z acceleration vs. roll attitude features [14] 8

3-Two-dimension projections for case 23,22 under motor failure case one and two, respectively [14] 10

4-History of activated detectors for motor one failure [14] 11

5-History of activated detectors for motor two failure [14] 11

6-Validation data set [14] 12

7-Autopilot logic design in EFSM layout [19] 20

8-Airplane coordinate system 29

9-System Identification block diagram 31

10-TIC with PI showing normal vs. abnormal condition 36

11-Effect on prediction interval based on the selection of alpha 38

12-Control surfaces on fixed-wing aircraft 39

13-SIG EdgeTRA aircraft selected for experimentation 41

14-RNode radio installed in-plane and second RNode connected to GCS 44

15-MUX board implementation 45

16-QGroundControl telemetry display and map while the EdgeTRA is in flight 46

17-SITL diagram 49

18-SITL map view during flight simulation 49

19-Model of Great Planes high wing trainer in X-plane 10 50

Figure Page

20-Flight operations types and Python code used with each 52

21-EdgeTRA in wings-level condition 54

22-Auto control waypoints and flight path 56

23-Aircraft target heading determination diagram 57

24-Flight path with failure modes 59

25-Data used as the input and output to the ARX MIMO model 61

26-Cube Orange input vs. output 62

27-Developed Python scripts that run on the RPI 66

28-Raspberry Pi USB ports used to connect to Cube Orange 67

29-Change of time between MAVLink messages 69

30-Interpolation verification from MIMO_Model_Input_03_14_2020__15_28_51 71

31-SSH terminal screen from RPI on the GCS reporting TIC values 72

32-Inputs used to identify ARX model for run MIMO_4_05_2020__18_03_08........................ 74

33-Outputs used to build ARX model for run MIMO_4_05_2020__18_03_08 75

34-Fitted output using input from the same data used to build the ARX model for run

MIMO_4_05_2020__18_03_08 77

35-Inputs used to validate ARX model for run MIMO_4_05_2020__18_03_08 78

36-Outputs used to validate ARX model for run MIMO_4_05_2020__18_03_08...................... 79

37-Fitted output using validation input for run MIMO_4_05_2020__18_03_08 80

38-Manual control roll TIC vs. run number.. 81

39-Manual control pitch TIC vs. run number 82

40-Manual control yaw TIC vs. run number 82

Figure Page

41-Inputs used to identify ARX model with auto control for run MIMO_6_13_2020__16_18_37 84

42-Outputs used to identify ARX model with auto control for run MIMO_6_13_2020__16_18_37 85

43-Fitted output using input from the same data used to build the ARX model for run MIMO_6_13_2020__16_18_37 88

44-Inputs used to validate ARX model for run MIMO_6_13_2020__16_18_37 89

45-Fitted output using validation for run MIMO_6_13_2020__16_18_37 90

46-Automatic control roll TIC vs. run number 91

47-Automatic control pitch TIC vs. run number 92

48-Automatic control yaw TIC vs. run number 92

49-Fitted output after aileron two and elevator two are stuck neutral for run MIMO_4_05_2020__16_55_18 93

50-Fitted output under nominal conditions for run MIMO_4_05_2020__16_55_18 94

51-Roll TIC vs. run number showing runs when aileron two is failed neutral 95

52-Pitch TIC vs. run number showing runs when aileron two is failed neutral 96

53-Yaw TIC vs. run number showing runs when aileron two is failed neutral 97

54-Roll TIC vs. run number with aileron two having limited travel 98

55-Pitch TIC vs. run number with elevator two stuck neutral 99

56-Pitch TIC vs. run number with elevator two travel limited 100

57-Roll TIC vs. run number with aileron two neutral and elevator two travel limited 101

58-Pitch TIC vs. run number with aileron two neutral and elevator two travel limited 102

Figure Page

59-Roll TIC vs. run number with aileron two limited and elevator two neutral 103

60-Pitch TIC vs. run number with aileron two limited and elevator two neutral 104

61-Roll TIC vs. run number with aileron two and elevator two stuck neutral 105

62-Pitch TIC vs. run number with aileron two and elevator two stuck neutral 106

63-Roll TIC vs. run number with aileron and elevator travel limited 107

64-Pitch TIC vs. run number with aileron and elevator travel limited 108

65-Yaw TIC vs. run number with rudder travel limited 109

66-Pitch TIC vs. run number with elevator two and rudder two travel limited 110

67-Pitch TIC vs. run number with elevator two and rudder two travel limited 112

68-Bifilar MOI suspension configuration for Ixx and Izz 114

69-Fitted output under nominal conditions for run MIMO_04_05_2020__18_21_23 116

70-Fitted output under nominal conditions for run MIMO_04_05_2020__18_19_57 117

71- EdgeTRA XFLR5 dynamic model 118

72-Root locus plot of longitudinal modes for EdgeTRA 120

73-Root locus plot for lateral modes for EdgeTRA 120

74-Rudder input and yaw rate response without active control 122

75-Aileron input and roll rate response with active control 123

CHAPTER 1

INTRODUCTION

In recent years, small fixed-wing Unmanned Aerial Vehicles (sUAV) have become readily available. Their small size makes them enticing test platforms to be used by commercial industry, in academic settings, and by the model airplane hobbyist. Open-source flight controllers, a key enabler for low-cost research and commercial products, can also be added to sUAVs to allow for more advanced control. A Cube Orange is an open-source standard flight controller in the sUAV industry. Adding it to a small aircraft model offers functions to stabilize an sUAV in windy conditions, fly a mission autonomously, and return to the home location, to name a few functions available. These autonomous functions have helped drive the increase in demand for fixed-wing sUAVs because, traditionally, the attrition rate of small fixed-wing aircraft is high. Fixed-wing sUAV flight dynamics are typically slow enough that a human can act as the flight controller. For instance, if the plane is not wings-level, the roll response and aircraft stability allow the human pilot to level it. However, this takes hundreds of hours of training to become proficient. With an autonomous flight controller, controlling an sUAV is simplified. Therefore, the amount of training required to fly a fixed-wing sUAV can be significantly reduced.

Autonomous flight controllers do not only aid the Pilot In Command (PIC), but some vehicle health diagnostics are provided synchronously for the safety of the vehicle and people on the ground. Though these health diagnostics provided by the flight controller do not encompass all possible failure modes of an sUAV, a few examples of the features are that the flight battery voltage and current are monitored [1]. Suppose the flight battery voltage drops below a pre-determined threshold. In that case, the flight controller takes action to automatically return the vehicle to the home location to prevent complete loss of the vehicle, preventing a potential crash, damage to property, or injury to people on the ground. Other health diagnostics include monitoring the remote control radio link connection, the telemetry link with a ground control station, GPS position estimation, and excessive vibrations. All these health diagnostics are important. However, open-source flight controllers, such as the Cube Orange do not have advanced diagnostics to determine if the aircraft is still controllable or suffering from degraded controllability.

Loss of control can be due to many factors but is typically attributed to malfunction of control surface servo actuators, as they are the input to the aircraft. Fixed-wing sUAVs utilize control surfaces that deflect to create positive or negative lift increments on the wing and empennage for in-flight control [2]. These surfaces are driven by servos, which convert signals commanded by the PIC on the ground to a control surface's mechanical movement. Servos are either digital or analog, with the difference being that digital servo position control operates at 300Hz compared to 50Hz of the analog servo. Also, digital servos are not as susceptible to temperature and supply voltage changes that affect analog-servo zero-position [3, 4]. Servo anatomy consists of an electric motor, gear train, motor position feedback sensor such as a potentiometer, and a closed-loop controller [5].

The failures of servos can be divided into electrical and mechanical failure modes. As for electrical failures, the DC motors within the servo can vary in type, such as brushed or brushless. However, all motors are susceptible to electrical short circuits and overheating due to excessive current draw. For the position of the DC motor, feedback of the motor position is provided by a potentiometer. Potentiometers are susceptible to blockage from dirt and debris, which causes false readings [6]. A false reading prevents the desired pilot input from being achieved by the servo, which can be catastrophic.

Mechanical failure modes are attributed to the gear train, communication lines, and power conductors. Low-cost servos used for RC aircraft, such as the Hitec HS-311, have gears made of plastic that are susceptible to deformation of the gear teeth [7]. Deformation can occur from sudden acceleration, such as a control surface being struck by a stationary object when transporting an sUAV or a bird strike in flight [8]. This sudden acceleration causes intermediate gear teeth to be deformed as they cannot rotate with enough angular velocity. The deformation of plastic gears also includes overloading and general wear from use. Also included in mechanical failure modes are communication and power lines. The command signal is transmitted via a wire to the servo from a receiver or flight controller, relaying the pilot's command on the ground. Therefore, the command signal transmission and power wires are susceptible to loose connections, damage due to chafing of insulation, connector corrosion, and melting from an excessive current draw, leading to servo actuator failure.

Currently, vehicle health diagnostics for open-source flight controllers that utilize ArduPilot firmware lack the ability to detect loss of control of an sUAV. Knowing the controllability of an sUAV is even more critical in Beyond Visual Line of Sight (BVLOS) operations, where most of the flight of the sUAV is out of view of the PIC or any other spotter to ensure the vehicle is flight worthy. This is unlike typical Visual Line of Sight (VLOS) operations, where the PIC can check for controllability by RC stick commands and visually see the sUAV response. BVLOS operation, when authorized, is typical for package and medical supplies delivery where the flight path may be over urban environments. Having the ability to determine controllability provides the flight controller with valuable information. Without this knowledge, the flight mission continues despite any damage sustained, creating a dangerous situation for the sUAV. The longer the damaged sUAV stays in the air, the higher the probability of catastrophic loss of control resulting in complete loss of the vehicle, injury to people, and damage to property on the ground.

1.1 PROBLEM STATEMENT

This research aims to develop in-flight diagnostics to detect the loss of controllability in an sUAV. Controllability is defined by an aircraft's ability to maneuver based on available controls under normal circumstances. For a fixed-wing sUAV, controllability is assessed by evaluating the primary control response, measured angular velocities about the roll, pitch, and yaw axis, based on control surface inputs. The method leverages the use of historical knowledge of the response to primary flight control inputs to build empirical models for all axes. Next, a method for rapidly building a new response model in flight is used to compare responses to the baseline model and establish thresholds for minimum controllability through statistics. The work features popular ArduPilot firmware and runs on a commonly available Cube Orange flight controller hardware. This hardware and firmware combination is widely used by industry, academics, and hobbyists, which gives the best opportunity for implementation in a wide variety of sUAVs. Other important considerations are that most sUAVs cannot measure actual deflection angles (closed-loop), inertial mass measurements are unknown, and onboard sensors are limited. These sensor outputs are essential to using system identification techniques that utilize aircraft equations of motion. Although additional hardware could be added, this adds cost and requires expertise in each additional sensor’s setup and calibration. This work focuses on sensors commonly used by typical flight controllers, such as the Cube Orange, Pixhawk, and mRo Control Zero used to fly an sUAV autonomously. The goal was to develop a simple methodology that could be applied across platforms, requiring only an Inertial Measurement Unit (IMU), an airspeed sensor, and a remote-control signal input. Using sensors already available from the flight controller makes the detection system readily transferable from one sUAV to another with few if any hardware changes. Also, this allows the detection system to work on many different sUAV configurations, such as a stable high wing design, maneuverable mid-wing, and Vertical Takeoff and Landing (VTOL) sUAVs because the flight controller can be used in many different vehicle types. The loss-of-primary-control detection system utilizes a *black box system identification* approach instead of aircraft equations of motion, which rely on knowing aircraft inertias and deflection angles of control surfaces. Therefore, to detect primary loss of control, an empirical model can be built to describe how the sUAV is performing at an instant in time, based only on input and output data. This model is then compared to measured historical baseline response data to check for controllability.

CHAPTER 2

LITERATURE SEARCH

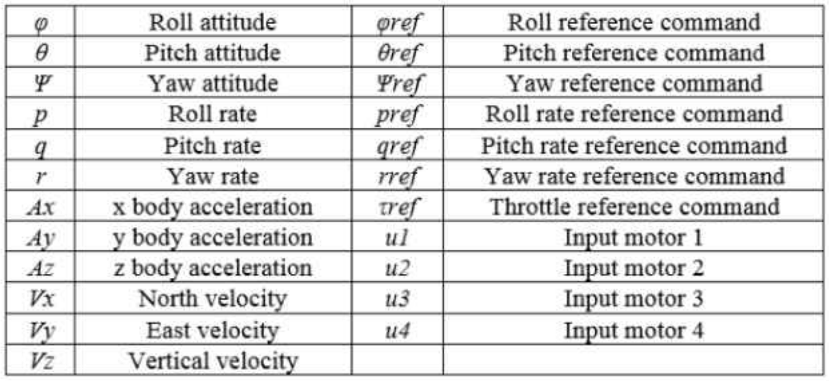
2.1 CURRENT HEALTH DIAGNOSTIC METHODS

2.1.1 BIOMIMETIC METHOD AIS NEGATIVE SELECTION

A biomimetic method called Artificial Immune System (AIS), which is modeled after the human immune system, is a relatively new area of study in health diagnostics for sUAVs. Traditionally, previous AIS applications have been utilized in computer security to protect from viruses, pattern recognition, and fault detection for sensors used in industrial plants [9-11]. AIS is within the context of machine learning. However, AIS is a stand-alone category compared to neural networks and evolutionary algorithm techniques [12, 13].

Garcia et al. applied an AIS for a multi-copter health diagnostics for detecting a motor failure in an sUAV [14, 15]. This paper used an AIS negative selection method to build a health monitoring system in which the AIS algorithm was developed to model how the human body detects bad and good cells. In living organisms, the thymus gland contains T-cells and self-proteins. If a T-cell reacts to a self-protein, then this T-cell is destroyed. A T-cell that does not react to the self-protein can stay and destroy bacteria or viruses. This principle method of *self* and *non-self* discrimination is known as negative selection. The concept is that anything that does not belong to *self* should be deleted. In the case of a living organism, the T-cells that do not belong are eliminated.

For this idea of negative selection, an AIS is to be applied to aircraft. Therefore, understanding what *self* encompasses needs to be defined, which is done by collecting data on many different features. Features are measurements from sensors such as attitude, rates, and accelerations, to name a few. In this paper, the author used 23 features to develop *self*, as shown in Figure 1.



φ	Roll attitude	φref	Roll reference command
θ	Pitch attitude	θref	Pitch reference command
Ψ	Yaw attitude	Ψref	Yaw reference command
p	Roll rate	$pref$	Roll rate reference command
q	Pitch rate	$qref$	Pitch rate reference command
r	Yaw rate	$rref$	Yaw rate reference command
Ax	x body acceleration	τref	Throttle reference command
Ay	y body acceleration	$u1$	Input motor 1
Az	z body acceleration	$u2$	Input motor 2
Vx	North velocity	$u3$	Input motor 3
Vy	East velocity	$u4$	Input motor 4
Vz	Vertical velocity		

Figure 1-List of features to be recorded

Data were collected for the listed features by flying the quadcopter in an altitude hold mode while rolling and pitching the vehicle ±10 degrees for 30 seconds. The responses are then normalized from 0 to 1 and undergo a clustering process. This normalized clustered data forms the *self* clusters for all two-dimensional projection combinations of the features, as shown in Figure 2 by the blue circles.

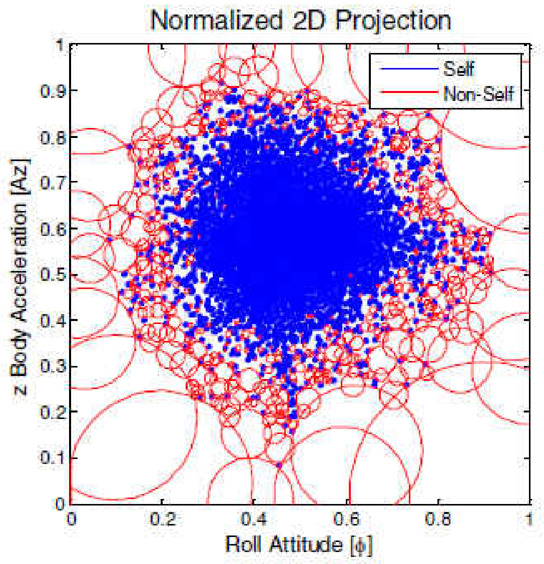


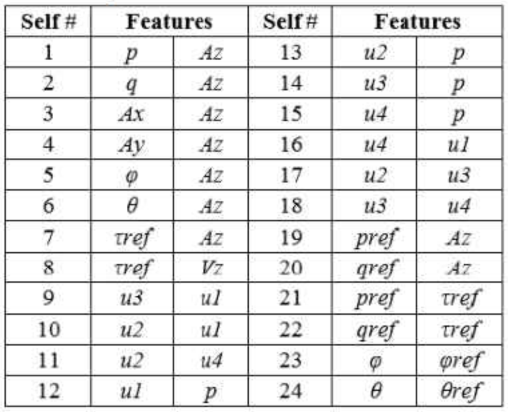
Figure 2-Two-dimensional projection of z acceleration vs. roll attitude features [14]

Before forming a projection, such as the z acceleration vs. roll attitude shown in Figure 2, the entire projection is first considered *non-self*-clusters, which are the red circles. Self-clusters are then overlaid onto the projection from normalized nominal flight data. Anywhere a self-cluster overlaps a *non-self*-cluster, this overlapped *non-self*-cluster is removed. The removal of *non-self*, where *self* overlaps, gives this method the name, negative selection. Once the negative selection process is performed, the algorithm optimizes the amount of non-self-empty space to characterize the entire projection space. The process is repeated for all possible combinations of features. Equation (1) is used to calculate all possible combinations to ensure a complete data set. Since there are 23 features, it is found that 253 projections are needed to describe the entire *self* and *non-self*.

$$N_{self} = C_N^{N_{max}} = \frac{N_f!}{N_{max}!\,(N_f - N_{max})!} = \frac{23!}{2!\,21!} = 253 \qquad (1)$$

With a database of *self* vs. *non-self*-understood, future data points are used as detectors to determine if the data point is a *self* or *non-self*. Calculating the Euclidian distance from each future data point to the centers of all surrounding clusters determines a detector's status, as the closest cluster defines whether the future data point is *self* or *non-self*. Detectors are said to be activated if they are found to be *non-self*. The number of summed activated detectors is then used to determine if there is a failure or not. However, some detectors are always activated due to sensor noise and modeling errors that should be considered. A MATLAB Simulink model is used to test this algorithm. A simulated quadcopter model is used to simulate two different motor failure scenarios, where a 2.5% reduction in efficiency for each motor is the mode of failure.

After post-processing, the author found that out of 253 projections only 24 needed to be considered based on the number of activations. The significant projections considered are shown in Table 1.



Self #	Features		Self #	Features	
1	*p*	*Az*	13	*u2*	*p*
2	*q*	*Az*	14	*u3*	*p*
3	*Ax*	*Az*	15	*u4*	*p*
4	*Ay*	*Az*	16	*u4*	*u1*
5	*φ*	*Az*	17	*u2*	*u3*
6	*θ*	*Az*	18	*u3*	*u4*
7	*rref*	*Az*	19	*pref*	*Az*
8	*rref*	*Vz*	20	*qref*	*Az*
9	*u3*	*u1*	21	*pref*	*rref*
10	*u2*	*u1*	22	*qref*	*rref*
11	*u2*	*u4*	23	*φ*	*φref*
12	*u1*	*p*	24	*θ*	*θref*

Table 1-Post-processed selection of projections

Figure 3 shows an example of projections 22 and 23. The black dots in Figure 3a represent data collected when the motor one had a 2.5% efficiency reduction. Figure 3b shows test data for motor two with a 2.5% efficiency reduction. The average number of test data points or detectors was 600 for each projection during the algorithm's initial testing.

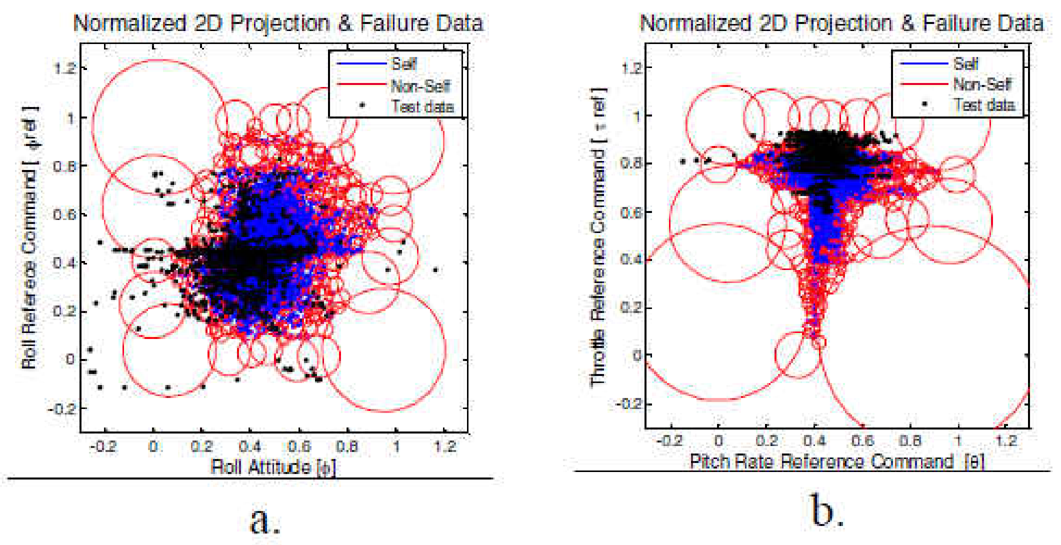


Figure 3-Two-dimension projections for case 23,22 under motor failure case one and two, respectively [14]

Counting the number of activated black dot detectors over time for all 24 projections allows for real-time implementation of the algorithm by creating a time history of activated detectors, as shown in Figure 4 for motor one failure. The author states that no failures have been implemented for the first eight seconds, providing nominal conditions. The number of activated detectors then increases above 50 at 10 seconds into the test showing that a failure has been detected. The activated detectors are not constant because the non-linear dynamic inversion controller used to fly the multi-copter recognizes the failure and attempts to compensate momentarily. This oscillating pattern of increasing and decreasing detectors activated is also seen for motor two failure conditions, as shown in Figure 5.

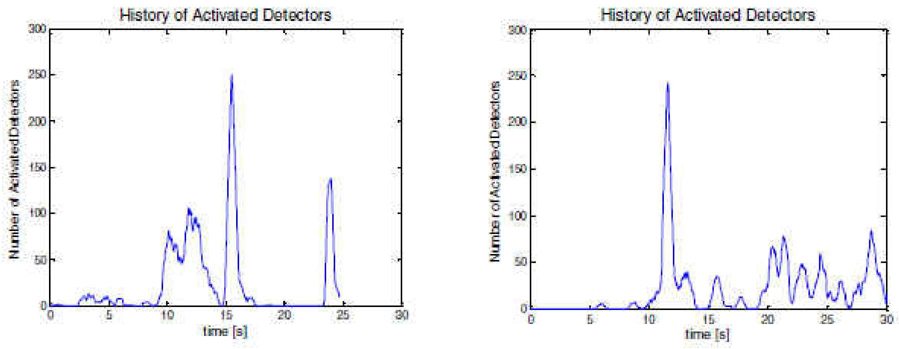


Figure 4-History of activated detectors for motor one failure [14]

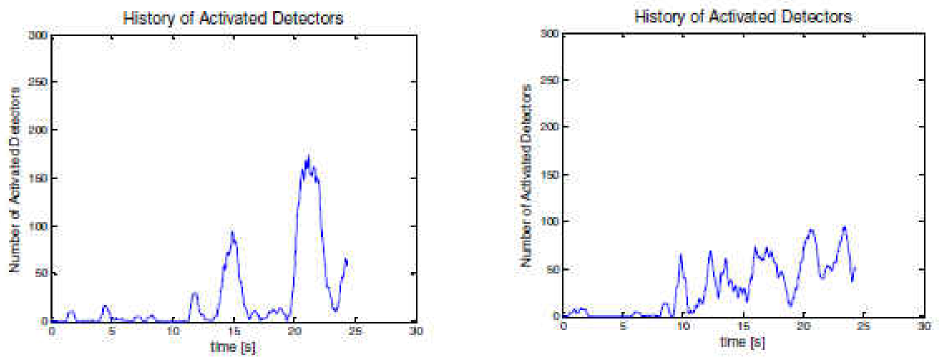


Figure 5-History of activated detectors for motor two failure [14]

These results were validated by performing more flight tests in the same manner to build the *self* and *non-self*-projections. The quadcopter was rolled and pitched ±10 degrees to create a nominal validation data set. Figure 6a shows the roll and pitch values that were collected over 60 seconds.

For this data set, it is shown in Figure 6b there are few activated detectors, which shows the algorithm is effective.

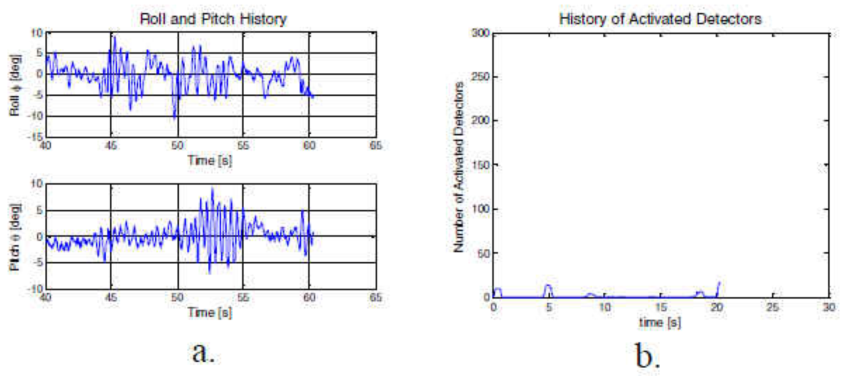


Figure 6-Validation data set [14]

2.1.1.1 SUMMARY OF AIS NEGATIVE SELECTION

The use of AIS negative selection was shown to be an effective approach for fault detection within an sUAV. In this method, a data set of desired features is selected, collected, and normalized, which allows the creation of two-dimension projections of every possible combination from the list of the desired features. The two-dimension projections display the self and non-self-areas, indicating nominal or abnormal regions of the two-dimension projection. These two-dimension projections have future test data called detectors overlaid. Based on where the detector falls in the projection, it is either found to be activated or not activated. An activated detector means failure is present, while a not activated detector indicates no failure present. Continuously counting the activated detectors provides the AIS method with real-time implementation.

Other authors, such as Lopez et al., have implemented the AIS negative selection method similarly for multi-copters [16]. However, in Lopez et al.'s work, the failure modes implemented were completely inoperable motors instead of just reduced efficiency as the mode of failure. Even with the different failure modes, the results were found to be similar. Additionally, applications of AIS negative selection have also been applied to fixed-wing aircraft. In Sanchez et al.'s work, an RC jet aircraft is utilized where an AIS negative selection method is applied to develop a fault detection scheme for control surfaces [17, 18]. Only two failure modes were tested, and they are one of two elevators and ailerons stuck in a neutral position while performing a doublet maneuver. From flight testing the RC jet, the AIS algorithm detected control surface faults for both manually controlled via a pilot and a mode where a stabilization controller assists the pilot. Overall results from these works show AIS negative selection is an effective way to determine fault detection because of its ability to include aerodynamic coupling effects, diversity of possible airframe types, and the ability for real-time implementation.

2.1.2 SEMI-AUTONOMOUS sUAV AUTOPILOT LOGIC DESIGN METHOD

Quan discusses a multi-copter design and control health evaluation method for flight controllers [19]. This health diagnostic focus is on the flight controller itself. For example, is sensor data from the IMU valid? The report covers three different failure types: communication, sensors, and propulsion. Also, the use of an Extended Finite State Machine is developed to semi-autonomously counteract any of the three failure modes and ensure the safety of the sUAV.

Communication failures occur when the RC transmitter loses the link with the receiver on the vehicle. The causes of communication failure can be from hardware failures or even operator errors, such as the transmitter being turned off accidentally when the vehicle is powered. Also, flight controllers that are not calibrated for paired transmitter endpoints fall within this failure type. By not calibrating the flight controller to the transmitter endpoints, the flight controller does not understand what the operator is commanding, leading to flight accidents. An example of this is that the operator wants the vehicle to roll left, but, instead, it rolls right. Communication with the vehicle is not limited to only an RC transmitter. A ground control station (GCS) is also utilized to provide essential telemetry data such as altitude and speed. However, this can be another source of communication failure. Like the RC transmitter, the GCS can lose the link with the vehicle because of hardware failures, range limitations, or loss of power to the GCS.

Sensor failures are defined as when a sensor cannot measure a quantity accurately or it malfunctions altogether. Examples of sensors that can fail are a barometer, compass, GPS, and Inertial Navigation System (INS). Barometer failure is considered when altitude measurements are inconsistent. Similarly, inconsistency in the compass heading and GPS position indicates a failure in these sensors. As for the INS, failure occurs when either the accelerometer or gyroscope is not calibrated, which produces inaccurate vehicle position estimates. Failure of the INS also includes possible hardware failure of the accelerometers or gyroscopes.

Propulsion failure encompasses the entire propulsion system. The system includes the battery, Electronic Speed Controllers (ESC), motors, and propellers. Each one of these components can lead to a failure in the propulsion system. Flight batteries can fail from low capacity, high internal resistance, overcharging, or over-discharging. An ESC can fail due to hardware limitations, such as overheating, limiting power, or eliminating power flow to the motors completely. Flight controllers send commands to the ESC in which some cases, the ESC does not recognize these commands. ESC failures directly relate to motor failures as the motor does not work if the ESC is not working correctly. Lastly, propeller failures occur when blades are worn, loose, cracked, or poorly balanced.

2.1.2.1 HEALTH MONITORING AND DESIGN

With the three failure modes defined (communication, sensors, propulsion), a health evaluation process is developed to determine if the discussed parameters associated with each failure mode are working correctly. The health evaluation is performed before and while in flight. A pre-flight check ensures all essential communication, sensors, and propulsion are functioning before a flight, as shown in Table 2. If there are any failures in the pre-flight check, they are reported to the GCS.

	Check Item	Failure Type
1	Whether the RC has been calibrated	Communication breakdown
2	Whether the RC connection is normal	Communication breakdown
3	Whether the barometer hardware fails	Sensor failure
4	Whether the compass hardware fails	Sensor failure
5	Whether the compass has been calibrated	Sensor failure
6	Whether the GPS signal is normal	Sensor failure
7	Whether the INS has been calibrated	Sensor failure
8	Whether the accelerometer hardware fails	Sensor failure
9	Whether the gyroscope hardware fails	Sensor failure
10	Battery voltage check	Propulsion system anomaly
11	Whether the critical parameter settings are correct	Parameter configuration mistake

Table 2-Pre-flight parameter checks

In-flight, communication is continuously checked to ensure that updated signals are received from the RC transmitter and the GCS. If one of the communication methods does not respond within five seconds, it is assumed there is a loss of contact. The sensors' health diagnostic during flight is best if the vehicle can be at a steady-state to avoid false alarms. Being at a steady-state is particularly important when checking the health status of the barometer. Large fluctuations in altitude measurements produce fault detection. In comparison, large fluctuations in yaw produce fault detections in the compass sensor. However, evaluating the compass sensor in greater depth indicates that the compass sensor is most susceptible to magnetic interference from the propulsion system. Magnetic interference can be measured as it fluctuates in strength, in which interference fluctuates due to varying current flow to increase or decrease motor RPM. These fluctuations must not exceed 60% of the original magnetic field, or the compass reading may suffer from severe interference [20]. The GPS position is checked by comparing it to an estimated position. This estimated position comes from the Extended Kalman Filter, which takes sensor data from the IMU. The GPS sensor is okay if the error between the measured and estimated positions is less than a pre-defined parameter value.

The propulsion system in-flight health monitoring has multiple checks as well, starting with the propellers. These are checked by ensuring excessive vibrations are not present, which is measured by the accelerometers within the flight controller. The battery is monitored by using a combination of methods. One way is to fly the vehicle until the voltage drops below a set value for several seconds. However, a real-time method is to calculate the Reserved Maximum Ampere-Hour (RMAH). There are some difficulties in doing this, as the flight battery voltage cannot be directly measured because of nonlinearity when under load. Also, calculating the remaining capacity of a battery must be continuously recalculated due to changing pilot inputs. Therefore, the State of Charge (SOC) calculates the battery state shown in equation (2) to combat the changing pilot inputs. S is the SOC of the battery, I is the discharge current, R is the battery impedance, Q is the nominal battery capacity, T is the sampling time, and w is the system noise. The SOC equation is then implemented in equation (4) to calculate the battery terminal voltage. C represents constant error offset, v is measurement noise, and OCV(S) is the curve of the Open

Circuit Voltage and SOC (OCV-SOC). The OCV(S) curves are found from battery charge and discharge tests. These equations still require instantaneous input to solve for the SOC and V, which is subject to error. To mitigate error, an Extended Kalman Filter is used to nonlinearly estimate the SOC using equations (2)(3)(4).

$$S_{k+1} = S_k - \frac{I_k T_s}{Q_{max}} + w_{1,k} \tag{2}$$

$$R_{k+1} = R_k + w_{2,k} \tag{3}$$

$$V_k = OCV(S_k) - I_k R_k + C + v_k \tag{4}$$

2.1.2.2 Safe Semi-Autonomous Autopilot Logic Design

A logic design process is used to implement the discussed health monitoring system by developing an Extended Finite State Machine (EFSM), which describes a discrete-event system. It is assumed that all the conditions in Table 3 are true. To use EFSM, all states, flight modes, and events need to be defined. A state refers to whether the vehicle is powered on or off. Flight modes describe what the vehicle is attempting to do. Loiter, stabilize, and landing are examples of flight modes in which the vehicle is holding position, self-leveling, and descending in altitude, respectively.

The system has a finite number of states
System behavior in a specific state should remain the same
The system always stays in a particular mode for a certain period
The number of conditions for the state’s switch is finite
A switch of the system state is the response to a set of events
The time of state switch is negligible

Table 3-EFSM conditions

Events are separated into Manual Input Events (MIE) and Automatic Trigger Events (ATE), which control the states and flight modes. MIE is directly from pilot input, such as arming or disarming the vehicle. MIE also includes switching flight modes like a return to launch, land, and stabilize. ATE is used when the flight controller recognizes there is a problem. For example, the vehicle is in loiter flight mode, but the flight controller finds the GPS unhealthy. To avoid an uncontrollable flight experience, the flight controller automatically switches the flight mode from loitering to altitude hold, which does not require GPS. ATE is similarly used when the battery is found to be unhealthy. No matter the flight mode, the flight controller sets the flight mode to land, preventing a crash. Table 4 defines all events used to build the autopilot logic design.

MIE1	1:denote to arm, 0:denote to disarm
MIE2	Manual operation instruction(1:Switch to MANUAL FLIGHT MODE; 2:Switch to RTL MODE; 3:Switch to AUTO-LANDING MODE)
MIE3	Turn on or turn off the multi-copter(1:turn on;0:turn off)
MIE4	Power cutoff for maintenance (1:repaired;0:repairing)
ATE1	Health status of INS and status of multi-copter (1:healthy;0:unhealthy)
ATE2	Health status of GPS(1:healthy;0:unhealthy)
ATE3	Health status of the barometer(1:healthy;0:unhealthy)
ATE4	Health status of the compass(1:healthy;0:unhealthy)
ATE5	Health status of the propulsion system(1:healthy;0:unhealthy)
ATE6	Status of connections of RC(1:normal;0:abnormal)
ATE7	The status of the battery's capacity(1:adequate, able to perform RTL; 0:inadequate, unable to perform RTL)
ATE8	Comparison of the multi-copter altitudes and a specified threshold (1:the multi-copters altitude is lower than the specified threshold;0:otherwise)
ATE9	Comparison of the multi-copters throttle command and a specified threshold over a time horizon(1:the multi-copters throttle command is less the specified threshold;0:otherwise)
ATE10	Comparison of the multi-copter distance from the home point and a specified threshold (1:the multi-copters distance from the home point is greater than the specified threshold; 0:the multi-copters distance from the home point is not greater than the specified threshold)

Table 4-Event definitions

The EFSM is defined by transition conditions developed using defined states, flight modes, and events. Transition conditions are strings of events, such as from power off to standby and vice versa, as seen in Figure 7, denoted by C1 and C2, respectively. C1 transition condition includes event MIE3=1 while C2 also includes event MIE3 but with a value of 0. By combining more events in the proper order, all states can be achieved.

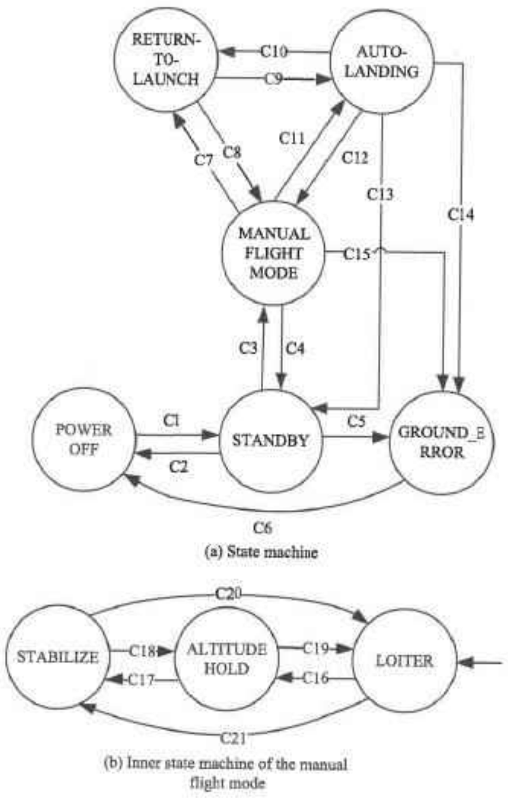


Figure 7-Autopilot logic design in EFSM layout [19]

Equation (5) is an example of transition conditions C1 and C3 needed to enter the manual flight mode state. In this example, events within the transition conditions show the vehicle is powered, arms, changes flight mode to manual, checks INS for health, checks propulsion health status, checks the RC communication, and checks the battery health status. These transition definitions are defined for all states and flight modes. By doing this, a road map is created for the flight controller to follow under normal and abnormal conditions.

$$C1{:}MIE3=1,\ C3{:}(MIE1=1)\&(MIE2=1)\&(ATE1=1)\&(ATE5=1)\&(ATE6=1)\&(ATE7=1) \quad (5)$$

2.1.2.3 SUMMARY OF AUTOPILOT LOGIC DESIGN METHOD

Of the three possible modes of failure discussed, communication, sensors, and propulsion, the ability to detect and react to the failure modes helps ensure the safety of an sUAV at the flight controller firmware level. The health evaluation was implemented before take-off and while in flight to provide the opportunity to monitor for abnormalities continuously. If an abnormality was detected, a developed semi-autonomous logic design would allow the autopilot to switch flight modes automatically. An example would be a scenario in which the current flight mode utilized the GPS for the vehicle location but the GPS signal was lost. The logic is designed so that the flight mode requires GPS changes to a different flight mode, which is not dependent on vehicle location obtained from the GPS. Automatically changing flight modes in this example helps prevent the sUAV from flying out of control, which can lead to flight into restricted airspace, damage to property, and possible injury to people. Tridgell et al. and Meier et al. have implemented this health diagnostic method within the flight controller firmware called ArduPilot and PX4, respectively [21, 22]. Based on these implementations, health diagnostics effectively detect and remedy communication, sensors, and propulsion modes of failure at the firmware level.

2.1.3 SERVO FAULT DETECTION MODELING CURRENT FLOW METHOD

Fuggetti et al. argued that if an aircraft is suffering from a lack of controllability, it is likely due to faulty servo actuators. They provide the input to the aircraft dynamics [23]. In this method, the current absorbed to servo actuators is modeled. This model is then compared to the measured absorbed current, and if both current values do not match, there is a problem with a servo actuator. Using Newton’s First Law and Kirchhoff’s Voltage Law, the DC servo is modeled using an ODE system of equations (6) and (7).

$$Jn\ddot{\theta}(t) = \frac{1}{K_v}I(t) - T_L(t) - \beta n\dot{\theta}(\mathrm{t}) \tag{6}$$

$$V(t) = RI(t) + L_c\dot{I}(t) + \frac{n}{K_v}\dot{\theta}(\mathrm{t}) \tag{7}$$

These equations are put into a transfer function form by understanding the inputs and outputs of a servo. The input to a servo is the desired position θ_{REF}. Knowing the desired position, the servo control loop within the servo applies a voltage to the DC motor to rotate the servo arm. This voltage is then related to the current used to drive the servo to θ_{REF}. Equation (8) describes this in the transfer function form and is populated by applying the Laplace transform to equations (6) and (7), leading to equations (9) and (10), respectively.

$$\frac{\theta(s)}{V(s)} = \frac{\theta(s)}{I(s)}\frac{I(s)}{V(s)} \tag{8}$$

$$\frac{\theta(s)}{I(s)} = \frac{\frac{1}{nK_vJ}}{\frac{\beta}{J}s + s^2} \tag{9}$$

$$\frac{I(s)}{V(s)} = \frac{\frac{\beta}{JL_c} + \frac{1}{J}s}{\frac{\beta RK_V^2 + 1}{JK_v^2L_c} + \frac{JR + \beta L_c}{JL_c}s + s^2} \tag{10}$$

Equation (10) provides the transfer function model of the current absorbed based on an input voltage applied, which can be simplified to identify parameters within A, B, C, and D. These parameters are identified by applying a step voltage and measuring the response current.

$$\frac{I(s)}{V(s)} = \frac{A + Bs}{C + Ds + s^2} \tag{11}$$

$$\frac{\beta}{JL_c} = A \tag{12}$$

$$\frac{1}{J} = B \tag{13}$$

$$\frac{\beta R K_V^2 + 1}{J K_v^2 L_c} = C \tag{14}$$

$$\frac{JR + \beta L_c}{JL_c} = D \tag{15}$$

Fault detection is based on the difference between the measured and estimated current, as shown in equation (16). Based on the difference's magnitude, there are four different fault conditions, as shown in Table 5. Based on initial testing, the nominal range of current flow was from 0 to 0.5A. If any current differences are above 0.5A, there is either a mechanical fault or a short circuit. If no current, then there is an electrical problem with the servo actuator, such as a broken wire or damaged DC motor.

$$r(t) = I(t) - \widehat{I(t)} \tag{16}$$

Fault condition	Residual
Fault-free	0A < r(t) < 0.5A
Mechanical fault	r(t) ≥0.5 A
Short Circuit	r(t) ≥0.5 A
Electrical fault	r(t) ≤ 0 A

Table 5-Category of fault conditions for a servo actuator

2.2 DISCUSSION OF CURRENT DIAGNOSTIC METHODS AND RELATED WORK

Of the three different methods reviewed in-depth, all perform a health diagnostic, but all have some drawbacks. The AIS method required the nominal model to be trained with previously recorded data. Therefore, the AIS diagnostic system cannot entirely be encompassed in one package on the sUAV as post-processing is required, which uses additional hardware to perform the computation to train the nominal model. Post-processing is problematic due to a need for additional hardware and the likely event of a configuration change of the sUAV. For example, suppose a multi-copter sUAV crashed, and as a result, a motor is damaged. Therefore, the motor is replaced. Since all motors differ slightly in terms of efficiency, mass properties, and dimensions due to manufacturing variances, if the AIS is not retrained, these differing motor factors may affect the AIS when the motor is replaced. False alarms may be a common occurrence even though the sUAV is nominal due to the AIS method's sensitivity. In addition to this damaged motor example, a more typical configuration change is changing the flight battery from run to run. Again, as with differences in motors, batteries vary in weight, dimensions, and current discharge rates. Using a different battery affects vehicle factors used in the AIS, such as vehicle acceleration, which can cause false alarms since the original AIS only knows nominal conditions with the battery used in nominal model building runs. Therefore, with any configuration change, it cannot be trusted until the AIS model has been retrained. This retraining process reduces this method's practicality for sUAVs, as an aircraft's payload may change from mission to mission.

This dissertation also discusses a semi-autonomous health diagnostic autopilot logic design built into the flight controller firmware. This method applies health diagnostic monitoring to sensors within a flight controller, omitting other necessary equipment, such as servo actuators and electronic speed controllers. For instance, in the event of a failed rudder control linkage in a fixed-wing sUAV, as semi-autonomous health diagnostic is only diagnosing the sensors within the flight controller, it might find everything normal even though the aircraft has no primary yaw control. Not having the ability to detect these kinds of controllability problems leaves this method with an incomplete health diagnosis.

Additionally, the method focused on the servo actuators, which are the direct input to the aircraft aerodynamics. The modeling technique was specific to one servo actuator, as transfer function models were built using data from a bench test rig with HXT-900 servos. With the technique applied to only one type of servo, this is problematic if the servo utilized is changed, which is likely the case from one fixed-wing sUAV to another. Using a different servo would require new data to be obtained from the servo of interest through bench testing and post-processing, which cannot be performed in-flight. This approach is also invasive as the method requires the knowledge of the voltage applied to the DC motor that drives the servo. Typically, the input voltage to the servo's DC motor is not available with Commercial Off the Shelf (COTS) servo actuators, where a constant voltage is applied, and an internal control loop regulates the voltage to the servo's DC motor. Therefore, the servo case must be removed to obtain this measurement, which, if not carefully performed, can introduce unnecessary problems that can create failures. In continuation, this requires additional hardware to measure the current absorbed by the servo, which also adds to the complexity and the number of parts that can fail.

As a way to mitigate the problems and limitations of the previously discussed work, additional literature was reviewed. In Gertler and Ding's work, the general approach to detecting faults is separated into two different methods [24, 25]. These methods are model-free and model-based. The model-free approach utilizes redundancy and established limits to perform fault detection. An example of the redundancy model-free approach is the use of multiple IMU sensors. With multiple sensors, the readings can be compared with one another to check for proper operation. If there are several IMU sensors, then a voting scheme can be implemented to determine which IMU is genuinely malfunctioning. In the case of the established limit, an example is a fixed-wing sUAV air velocity that is below stall velocity. Being below the stall velocity limit indicates a fault that the aircraft is flying too slow.

For the model-based approach, an explicit mathematical model of the system of interest is used, such as governing equations of motion, state-space models, and transfer functions. The calculation of residuals determines the detection of a fault. Residuals are the difference between the mathematical model estimate and the measured quantity from a sensor, and since there is always noise in a system, the residuals are never zero. Therefore, for the model-based approach, a residual evaluation process is conducted to compare the residuals to an established threshold, determined by experimentation or theoretical knowledge.

This model-based method has been demonstrated using an E-flite Ultra Stick 25 in the work of Freeman et al. The aircraft governing equations of motion are required, and the focus is on fault detection for control actuators [26, 27]. While monitoring the Ultra Stick 25 attitude, a command is applied at the same time. The detection of a faulty control actuator is performed by analyzing residuals. These residuals are the difference between the aerodynamics model's estimated attitudes and the measured attitudes from the IMU. After analysis, results show this method is feasible for controllability diagnostics of aircraft.

Following the literature review of current methods available for health diagnostics, some methods showing promising results have been found, although one gap in the previous research is the ability of a health diagnostic to detect whether an sUAV is suffering from a lack of controllability. Specifically, a controllability diagnostic capable of functioning with an sUAV that changes mass configurations often, such as a package delivery sUAV where the payload mass varies from run to run, can affect previously built nominal models. Therefore, the ability for a diagnostic to be developed in-flight without any post-processing or the use of large data sets to identify a nominal model represents a significant improvement to the state of the art. Another shortcoming identified in the literature search is that the vast majority of low-cost sUAVs entering the market are not suitable for typical model-based health diagnostics due to a lack of available sensors. For example, using the model-based method with aircraft equations of motion, sensors such as alpha and beta potentiometers, generally found on research sUAVs, are two variables needed when using the aircraft equations of motion as a nominal model. However, in standard low-cost sUAVs, these sensors are typically omitted to reduce cost and complexity, as they are not required for flight. Additionally, low-cost sUAVs often lack the necessary parameters, such as mass properties, required for an aircraft's complete mathematical model. This lack of prior knowledge about an sUAV is also to be considered if a health diagnostic is to apply to many different sUAVs.

CHAPTER 3

METHOD

3.1 OVERVIEW OF THE DEVELOPED METHOD

Fixed-wing sUAVs using ArduPilot firmware has been found to lack the ability to check for degraded controllability. Specifically, controllability checks performed while an aircraft is in flight include the immediate use of any previously created nominal models. Therefore, from the time a fixed-wing aircraft takes off and lands, a controllability check should be performed. Also, there is a lack of sensors for performing a controllability check for consumer-grade sUAVs. The reason is that sensors are costly and add complexity to a fixed-wing sUAV. Sensors can be added to sUAVs, but many require unique installation and calibration knowledge-making established controllability checks impractical for the average fixed-wing sUAV. In addition to this, aircraft constants, such as moments of inertia data, are not readily available, limiting the ability to use aerodynamic equations of motion as they require these constants.

Controllability check performed in-flight (no post-processing)
No knowledge of aircraft moment of inertias
No measurement of the control surface deflection angles
No measurement of the aircraft angle of attack or sideslip angle
Diagnostic of controllability is not to be configuration specific (eg. high wing vs. mid-wing)

Table 6-Controllability diagnostic requirements

The work performed in this research represents a way to accommodate the aforementioned limitations with requirements, as shown in Table 6. This work focused on the fact that all fixed-wing sUAVs have a principal axis, as shown in Figure 8. It is shown that the x-axis is out of the nose, the y-axis to the right wingtip and the z-axis points out of the bottom of the fuselage. For each principal axis, there is an associated force, velocity, angular velocity, and moment.

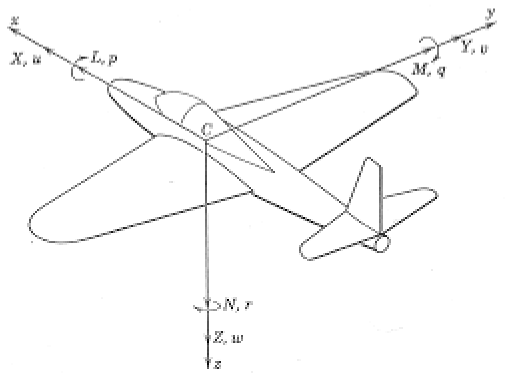


Figure 8-Airplane coordinate system

The force equations (17) to (19) require an unknown angle of attack and sideslip angle as well as accelerations and velocities to solve for forces X, Y, and Z [28]. Similarly, to solve for moments L, M, and, N, in equations (20) to (21) requires the aircraft inertia and angular rates. This work assumes inertias are unknown.

Force Equations:

$$\dot{u} = (rv - qw) + \frac{X}{m} - g\sin\theta + \frac{T}{m} \tag{17}$$

$$\dot{v} = (pw - ru) + \frac{Y}{m} + g\cos\theta\sin\phi \tag{18}$$

$$\dot{w} = (qu - pv) + \frac{Z}{m} + g\cos\theta\cos\phi \tag{19}$$

Moment Equations:

$$\dot{p} - \left(\frac{I_{xx}}{I_x}\right)\dot{r} = -\frac{qr(I_z - I_y)}{I_x} + \frac{qpI_{xz}}{I_x} + \frac{L}{I_x} \tag{20}$$

$$\dot{q} = -\frac{pr(I_x - I_z)}{Iy} - \frac{(p^2 - r^2)I_{xz}}{I_y} + \frac{M}{I_y} \tag{21}$$

$$\dot{r} - \left(\frac{I_{xz}}{I_z}\right)\dot{p} = -\frac{pq(I_y - I_x)}{I_z} + \frac{qrI_{xz}}{I_z} + \frac{N}{I_z} \tag{22}$$

Of the four values, angular velocity is intriguing because it is the only value that can be readily measured from a sensor for each axis, which meets the requirement that inertias and other sensors to measure the angle of attack and sideslip are not needed. With the ability to measure the angular velocity, the controllability check is defined by creating a mathematical model under nominal conditions of the angular velocities for roll, pitch, and yaw. Measured angular velocities are then compared to estimates from the model built under nominal conditions. The fit of the model vs. the measured angular velocities is based on a fit coefficient (metric). This coefficient's value is a type of go/no-go conditional that determines if the aircraft is suffering from a lack of controllability. For this work, a lack of controllability is defined as any roll, pitch, or yaw axis whose fit coefficient falls above a nominal threshold established from a Prediction Interval (PI). In other words, the controllability diagnostic is suitable to detect the partial or complete loss of control of an sUAV, such as in the case of a servo actuator malfunctioning.

3.2 SYSTEM IDENTIFICATION AND APPLICATION TO HEALTH DIAGNOSTICS

System Identification (SID) is the process of developing mathematical models of physical systems based on imperfect observations or measurements, and models are not unique [29]. Observations are the output of the system, which is caused by some input to the system. Using the input and output relationship allows the identification of the model for the system, as shown in Figure 9.

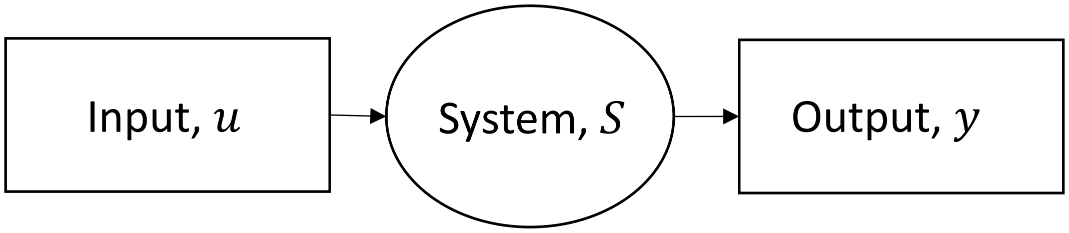


Figure 9-System Identification block diagram

For this paper, the aircraft is the defined system. Models of the system can estimate the physical system's values, such as angular velocity and acceleration. In addition, models can be used to estimate specific parameters within a set of governing equations. For example, in the work of Noah Favaregh, the pitching moment equation is used to solve the damping stability and control derivatives using a linear least-squares regression SID technique [30].

SID can be performed in the time or frequency domain. Frequency domain SID has the advantage that it offers a better understanding of the aircraft dynamics with the ability to create a bode plot [31-33]. However, frequency-domain SID requires excitation over a wide range of frequencies, increasing the needed run time. sUAVs are usually limited in-flight duration capability and physical air space within the ground-based pilot's view. In comparison, time-domain SID only requires excitation at a few frequencies of interest. However, the frequencies of interest may be unknown, making the frequency domain a more straightforward choice. In the case of a controllability diagnostic, the frequency of excitation can be chosen based on the frequency that safely excites the aircraft while avoiding resonant frequencies and is low enough to meet Nyquist theorem rules to prevent aliasing in data recording [34].

In the development of a controllability diagnostic, understanding the aircraft in a nominal state is crucial. SID gives the ability for the nominal model to be identified without the need for large data sets such as machine learning methods described in the literature review. Also, *black-box* approaches to modeling the system where there are no governing equations of the system make SID practical for controllability checks. For this paper, this is important as the assumption of no known physical aircraft properties prevents the use of aircraft governing equations. It should be noted that transfer functions can substitute for aircraft governing equations of motion.

3.2 AUTOREGRESSIVE MODELING TECHNIQUE

Autoregressive with Exogenous input (ARX) modeling is used to identify the roll pitch and yaw angular velocity models. ARX models are based on a discrete-time series transfer function approach where data from the past is used to predict the future based on an input [35, 36]. Equation (23) is the governing equation for the ARX model structure for a Single Input Single Output (SISO) where y_{output} is the output and u_{input} is the input. The left-hand side represents output terms while the right-hand side represents the input terms.

$$\begin{aligned} y_{output}(t) + a_1 y_{output}(t-1) + \cdots + a_{n_a} y_{output}(t-n_a) \\ = b_1 u_{input}(t-1) + \cdots + b_{n_b} u_{input}(t-n_b) \end{aligned} \tag{23}$$

The Laplace transform and z transform theorems are applied to each side of the equation to convert to the z domain shown in equations (24) and (25) [37].

$$LHS(z) = 1 + a_1 z^{-} + \cdots + a_{n_a} z^{-n_a} \tag{24}$$

$$RHS(z) = b_1 z^{-1} + \cdots + b_{n_b} z^{-n_b} \tag{25}$$

It is understood that the $LHS(z)$ represents the output, and the $RHS(z)$ represents the input, which allows substitution into the transfer function form that relates the input with the output, as shown in equation (26). $G(z)$ represents the mathematical model used to estimate the angular velocities for roll, pitch, and yaw. Based on an input u the output is modeled, as shown in equation (27).

$$G(z) = \frac{Inputs}{Outputs} = \frac{RHS(z)}{LHS(z)} = \frac{\left(b_1 z^{-1} + \cdots + b_{n_b} z^{-n_b}\right)}{\left(1 + a_1 z^{-1} + \cdots + a_{n_a} z^{-n_a}\right)} \tag{26}$$

$$y_{output} = G(z) u_{input} \tag{27}$$

Coefficients a_{n_a} and b_{n_b} are the terms to be identified and relate to the output and input, respectively. The coefficients are identified using linear regression after providing a discrete-time series of input and output data [38]. The order of the system dictates the number of coefficients. n_a and n_b set the order in $G(z)$ and are user-selectable parameters. Through experimentation, it was found $n_b = 2$ and $n_a = 3$ provided sufficient fit of the model to measured data for this lack of controllability diagnostic.

The aircraft equations of motion show coupling prominent within the roll and yaw axis. For example, rudder affects the yaw and roll, and ailerons affect roll and yaw as well. $G(z)$ in equation (26) only assumes SISO. For aircraft, the system must be Multi-Input Multi-Output (MIMO) to account for coupling. Therefore, a transfer function is required to relate each input to each output shown in Table 7. Four inputs were selected as aileron, elevator, rudder, and airspeed. The three outputs are roll, pitch, and yaw angular velocity.

Inputs	Roll Rate Output	Pitch Rate Output	Yaw Rate Output
Aileron (PWM)	$g(z)_{1,1}$	$g(z)_{2,1}$	$g(z)_{3,1}$
Elevator (PWM)	$g(z)_{1,2}$	$g(z)_{2,2}$	$g(z)_{3,2}$
Rudder (PWM)	$g(z)_{1,3}$	$g(z)_{2,3}$	$g(z)_{3,3}$
Airspeed (m/s)	$g(z)_{1,4}$	$g(z)_{2,4}$	$g(z)_{3,4}$

Table 7-MIMO Transfer function design

$$y_{roll\_rate} = g(z)_{1,1}u_{\mathrm{Aileron(PWM)}} + g(z)_{1,2}u_{Elevator\ (PWM)} + g(z)_{1,3}u_{Rudder(PWM)} + g(z)_{1,4}u_{Airspeed\ (\frac{m}{s})} \tag{28}$$

$$y_{pitch\_rate} = g(z)_{2,1}u_{\mathrm{Aileron(PWM)}} + g(z)_{2,2}u_{Elevator\ (PWM)} + g(z)_{2,3}u_{Rudder(PWM)} + g(z)_{2,4}u_{Airspeed\ (\frac{m}{s})} \tag{29}$$

$$y_{yaw\_rate} = g(z)_{3,1}u_{\mathrm{Aileron(PWM)}} + g(z)_{3,2}u_{Elevator\ (PWM)} + g(z)_{3,3}u_{Rudder(PWM)} + g(z)_{3,4}u_{Airspeed\ (\frac{m}{s})} \tag{30}$$

Equations (28), (29), and (30) show the addition of each column in the table. It is shown that each output is dependent on the four inputs, and these capture coupling effects. These three equations are developed in the same manner as the SISO. Discrete time-domain input and output data are used to solve for coefficients using linear regression. With the model defined, the output or response of the aircraft can be estimated with provided inputs.

3.3 LACK OF CONTROLLABILITY DETECTION

Understanding if there is a lack of controllability is based on how well the model estimate compares to measured angular rates from sensors. Evaluating the Theil Inequality Coefficient (TIC) compares the current model to historically measured results [39]. TIC is a metric of fit on a scale from zero to one, with zero as the perfect fit. Equation (31) defines the formula for the calculation of TIC. Measured sensor data is represented by x_i and model estimated data is $\widehat{x_i}$.

$$TIC = \frac{\sqrt{\frac{1}{n}\sum_i^n (x_i - \widehat{x_i})^2}}{\sqrt{\frac{1}{n}\sum_i^n x_i^2} + \sqrt{\frac{1}{n}\sum_i^n \widehat{x_i^2}}} \tag{31}$$

TIC only provides one observation of the fit per run and is susceptible to variance from run to run due to sensor noise and imperfect modeling. Therefore, the use of a Prediction Interval (PI) on a mean value is used. The PI is a form of confidence interval used for comparing individual future values to understand whether they belong to the original population [40, 41]. In this work, a PI is developed from a sample of multiple model evaluations. Equation (32) shows a two-sided PI [42]. Since minimum controllability is established using a threshold TIC value, a single-sided interval is most appropriate. Therefore, the positive side of the PI is selected, as shown in equation (33).

$$\hat{y} \pm t_{s_{\left(\frac{\alpha}{2}, n_s - 1\right)}} \cdot S \cdot \sqrt{1 + \frac{1}{n_s}} \tag{32}$$

$$\hat{y} + t_{s_{(\alpha, n_s - 1)}} \cdot S \cdot \sqrt{1 + \frac{1}{n_s}} \tag{33}$$

The PI threshold is demonstrated in Figure 10, where the bound is calculated using data available to build the ARX models. Acceptable TIC values fall below the calculated PI limit. However, if a TIC value is greater than the PI, then there is a lack of controllability detected.

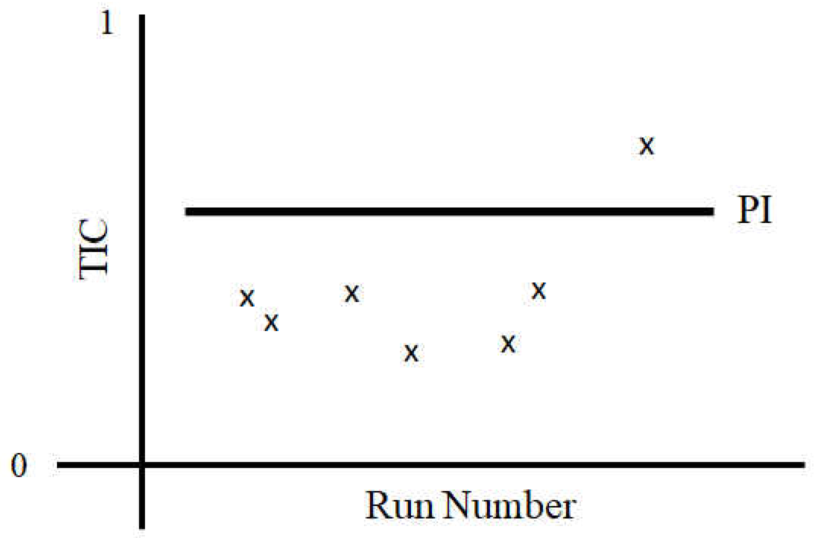


Figure 10-TIC with PI showing normal vs. abnormal condition

It is also worth noting that the sensitivity of detection can be adjusted by selecting different alpha values in the calculation of the PI. Alpha, the level of significance, is traditionally set at 5% for many engineering problems [43]. The PI evaluation is typically used for confirmation runs in an experimental setting to understand if the model is adequate for prediction. The alpha value sets the probability of determining a new observation as confirmed when it is not. Confirmation infers that the model controllability is unchanged from the nominal model. In order to determine alpha, a set of data collection runs were performed under nominal conditions. Then additional runs were performed with known problems introduced, such as limited throw of a control surface. Figure 11 shows the nominal results used in the selection of alpha. Nominal runs are indicated by red dots, while the black squares show runs with a stuck control surface failure introduced. The five black dashed lines represent different possible PI based on the selection of alpha. If alpha is set to a small percentage such as 5%, or a 95% PI, this leads to a greater chance that the algorithm determines the aircraft has full control authority. An alpha value set to 20%, resulting in an 80% PI, reduces the chance that the algorithm finds the aircraft to have full control authority, increasing the probability that the algorithm detects a lack of controllability.

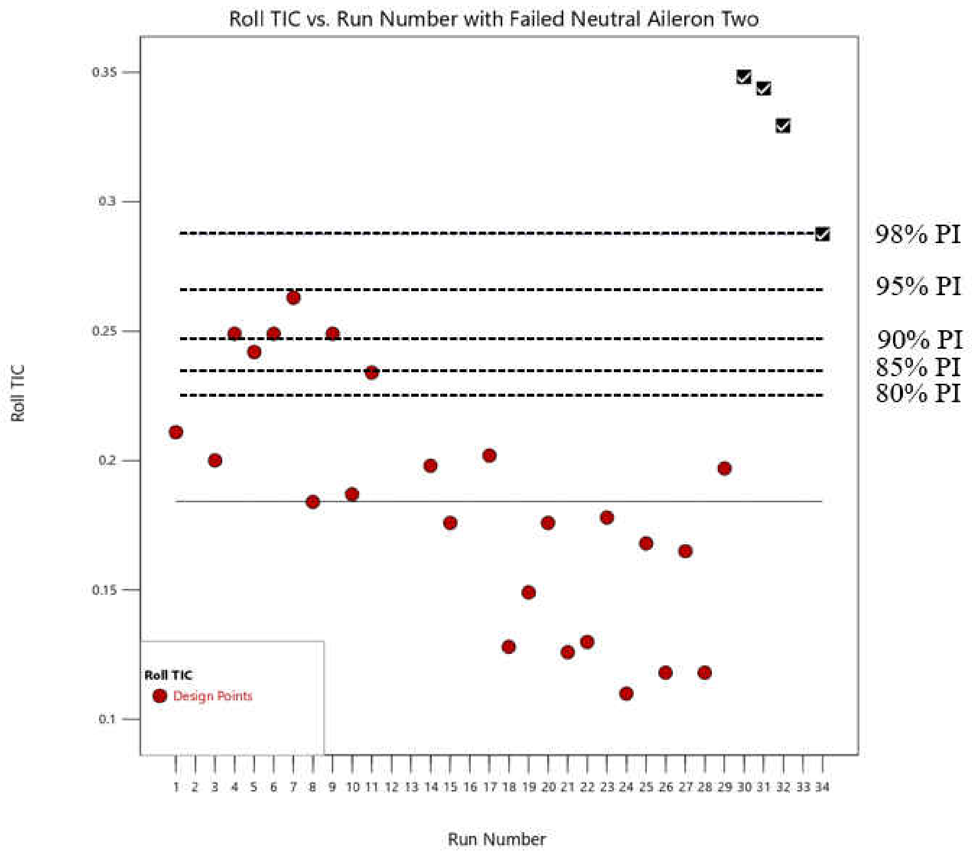


Figure 11-Effect on prediction interval based on the selection of alpha

Since this work focuses on detecting a lack of controllability, increasing the probability of detection is desired, which increases the chance of detection for small off-nominal failures, such as control surfaces with a limited throw. However, there is a fine line about how much alpha can be increased because overly increasing alpha can lead to significant false positives. Analyzing additional failure mode TIC results with a control surface with limited throw allowed the selection of alpha to be 20%, allowing the system to be sensitive when there is a failure while simultaneously not triggering many false alarms.

3.4 MODES OF FAILURE FOR LACK OF CONTROLLABILITY CHECK

This work focuses on the servo actuators being the root cause of the lack of controllability based on the literature search. Figure 12 displays in red commonly available control surfaces, such as the aileron, elevator, and rudder found on a fixed-wing sUAV.

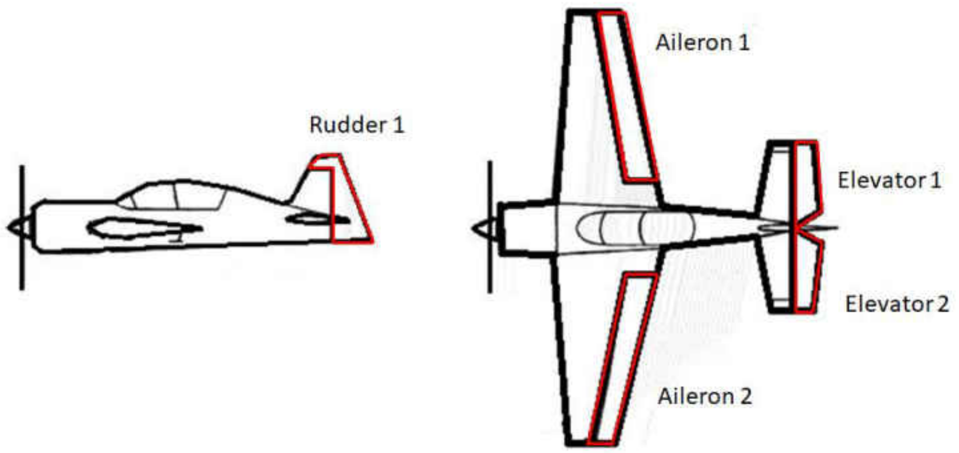


Figure 12-Control surfaces on fixed-wing aircraft

A list of several different failure modes is considered using these available control surfaces. The failure modes are: complete actuator failure, limited movement failure, and combinations of complete and limited failure modes. For example, aileron one and elevator both fail either entirely or partially, which is to include not only mechanical or electrical issues with the actuators but also external sources such as bird strikes. The thought process is damage is likely to occur to more than one surface at the same time. Table 8 displays all the failure modes tested for lack of controllability detection.

Failure Mode:	**Action to achieve failure mode:**
Stuck neutral aileron	Aileron two fixed neutral
Limited aileron	Aileron two throw limited to $\pm 25\%$
Stuck neutral elevator	Elevator two fixed neutral
Limited elevator	Elevator two throw limited to $\pm 25\%$
Stuck neutral aileron and limited elevator	Aileron two fixed neutral and elevator two throw limited to $\pm 25\%$
Limited aileron and stuck neutral elevator	Aileron two throw limited $\pm 25\%$ and elevator two fixed neutral
Stuck neutral aileron and stuck neutral elevator	Aileron two fixed neutral and Elevator two fixed neutral
Limited aileron and limited elevator	Aileron two throw limited $\pm 25\%$, and elevator two throw limited to $\pm 25\%$
Limited rudder	Rudder 1B throw limited $\pm 25\%$
Limited rudder and limited elevator	Rudder 1B throw limited $\pm 25\%$, and elevator one throw limited to $\pm 25\%$

Table 8-Failure modes tested for lack of controllability detection

Since the ARX model utilized the input PWM signal to aileron one, elevator one, and rudder one, the failures needed not to be introduced on these channels. Otherwise, the model estimates the response based on what it should be with the signal used to simulate a failure mode. Therefore, the failure modes need to be external to the ARX model to prevent the estimation of the angular rates with the failed signal, which is done by using aileron two, elevator two, and rudder 1B. These surfaces are external to the ARX model inputs used to estimate angular velocity for roll, pitch, and yaw.

CHAPTER 4

HARDWARE AND SOFTWARE

4.1 HARDWARE

The SIG EdgeTRA is the selected test platform for this work and is shown in Figure 13, with physical properties shown in Table 9. A few reasons for selecting this aircraft are the 60-inch wingspan and fuselage length with removable wing that ease transportation requirements. Also, there is a spacious interior for data acquisition equipment, and its dynamic characteristics are suitable for large input excitations. The EdgeTRA is an Almost Ready to Fly (ARF) model aircraft, meaning that the final configuration of the electronics is left to the end-user.

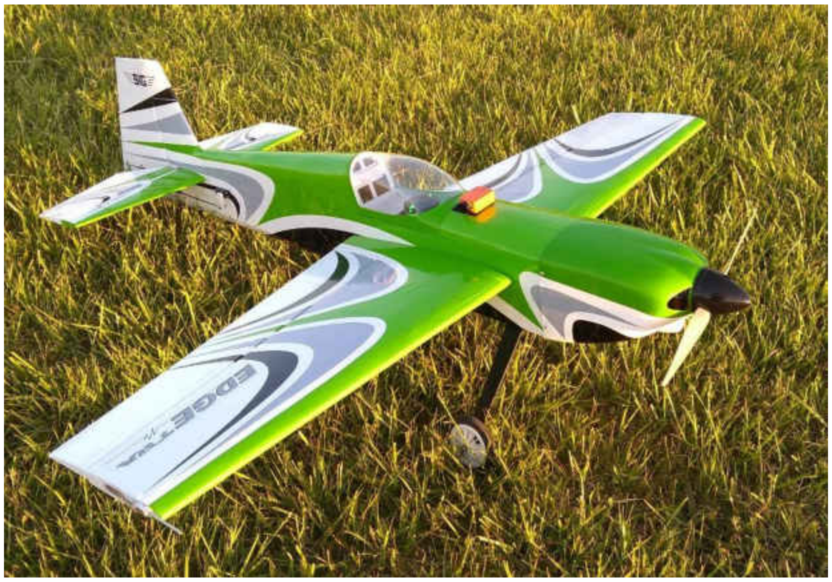


Figure 13-SIG EdgeTRA aircraft selected for experimentation

Wingspan	60 in.
Wing Area	675 sq. in.
Length	60 in.
Height	18 in.
Flying weight	8.57 lbs.
Landing gear main wheel diameter	4 in.

Table 9- Physical properties of EdgeTRA

Table 10 summarizes all additional components selected to complete the ARF EdgeTRA for flight. These components are selected based on recommendations from SIG, the manufacturer of the EdgeTRA. However, additional consideration was taken when selecting the receiver. Traditionally, the EdgeTRA aircraft only requires a four-channel receiver that accepts aileron, elevator, throttle, and rudder. Any pairs of control actuators such as the aileron servos traditionally would be joined together (y configuration) before plugging them into the four-channel receiver. To simulate servo failures to test the controllability diagnostic required all servos to be independent of one another. Therefore, each servo is assigned to a channel on the receiver requiring it to have at least six channels for aileron one, aileron two, elevator one, elevator two, throttle, and rudder one. Also, there is additional hardware that requires input from the PIC for data collection and safety equipment. With this, the receiver was required to have 9 channels. Therefore, the Spectrum AR9320T was selected.

Motor	E-flite Power 32
ESC	Castle Creations 100-amp Phoenix Edge Lite
Servos	HiTEC servos HS-5245MG
Receiver	Spectrum AR9320T
Battery	3 cell 5200 Lipo
Propeller	APC 14x8

Table 10-Baseline hardware use to fly EdgeTRA

Table 11 shows the additional hardware used for data acquisition. The Cube Orange flight controller is the basis of this work, which gives the EdgeTRA autonomous flight modes such as RETURN TO LAUNCH, LOITER, and AUTO. The same sensors used to perform the flight modes are also used for the ARX modeling of the angular velocities. The Cube Orange also controls failure modes, as it can limit travel or fix any servo position.

Cube Orange Flight Controller with Arduplane 4.0.5 Firmware
Here 2 GPS Antenna
4525 Digital Airspeed Sensor
3DR 900 MHz Telemetry Radio
Raspberry Pi 3B with Raspbian Stretch OS
433 MHz Rnode Radio
Cytron 8-Channel RC Multiplexer

Table 11-Additional hardware used for modeling and safety during failure modes

The failure modes are controlled, and data acquisition is performed using an onboard Raspberry Pi 3B (RPI) using Python scripts. The RPI is hard wired to the Cube Orange using two different serial links. One serial link was dedicated to data acquisition connected to the telemetry 2 port. In contrast, a second serial link dedicated to setting failure modes and general MAVlink commands was connected to the GPS 2 port. Respectively, the baud rate for each serial link is 921600 and 57600. In addition, to execute the Python scripts, 433 MHz Rnode radios are used. These radios use a LoRa network to provide a long-range, low power remote connection from one computer to another [44]. In this case, the RPI flight computer and the Ground Control Station (GCS) are the two computers connected via the Rnode radios, as illustrated in Figure 14.

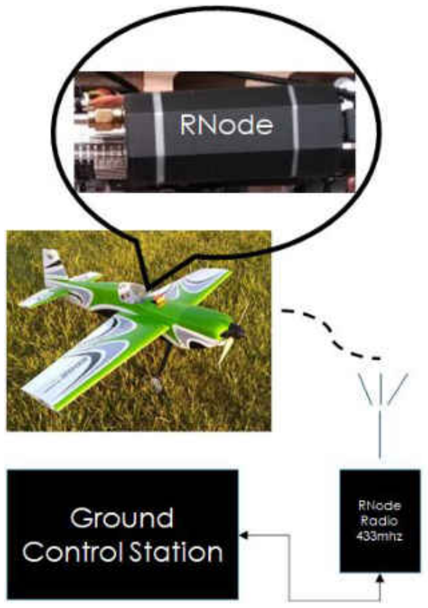


Figure 14-RNode radio installed in-plane and second RNode connected to GCS

In the event of a failure mode or if the Cube Orange malfunctions, an 8-channel Multiplexer (MUX) board is used to bypass all Cube Orange and RPI related commands, as shown in Figure 15. The MUX board has two inputs and one output. Input A is the master, and B is secondary. The output is where control actuators and the electronic speed controller (ESC) are connected. The AUX 2 channel on the Spectrum AR 9320T controls whether input A or B passes through the MUX board based on a Pulse Width Modulated (PWM) value. A PWM value ranges from 1.0 ms to 2.0 ms. When the AUX 2 signal is above 1.5 ms, commands from input A or commands from the pilot can pass through. If AUX 2 is less than 1.5 ms, commands from the Cube Orange can pass. Notice the intersection between the output of the receiver and the input A of the MUX board. This intersection eliminates the need to use two separate receivers, as the Cube Orange also requires pilot input to operate for general flight commands and flight mode changes. However, even though the Cube Orange is always receiving signals from the receiver, the Cube Orange commands are ignored if the MUX input selection is A. A complete wiring diagram in detail for the EdgeTRA is shown in APPENDIX A.

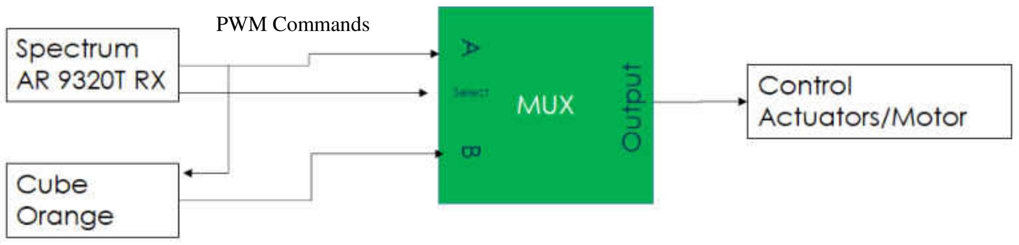


Figure 15-MUX board implementation

4.2 SOFTWARE AND FIRMWARE

ArduPlane 4.0.5 is the selected firmware to be run on the Cube Orange, which is a popular open-source firmware used by many commercial entities and hobbyists from around the world. ArduPlane provides the Cube Orange with the flexible setup configuration required for this work. For example, the ability to have each control surface actuator on independent channels such as aileron one and aileron two. Independent control surfaces allow for failures of individual servos to be tested. Also, autonomous capabilities to fly waypoint missions, return to launch, and loiter, to name a few, are used in this work. Lastly, the firmware provides access for the RPI companion computer, so pertinent sensor data can be collected for controllability diagnostics. Firmware setup and telemetry feedback of ArduPlane firmware are done using a GCS.

Mission Planner and QGroundControl are two GCS programs used by the ArduPlane firmware. The majority of this work utilized QGroundControl to set up the ArduPlane firmware, while Mission Planner helped the gain tuning process. The setup performed involved calibrations of the accelerometer, compass, and airspeed sensor of the Cube Orange. Failsafe parameters are also configured with QGroundControl, such as in the event of a loss of radio link, low battery conditions, and geofences. During flight operations, QGroundControl is used to provide telemetry information of the aircraft location via a satellite imagery map, airspeed, battery voltage, and altitude, as shown in Figure 16.

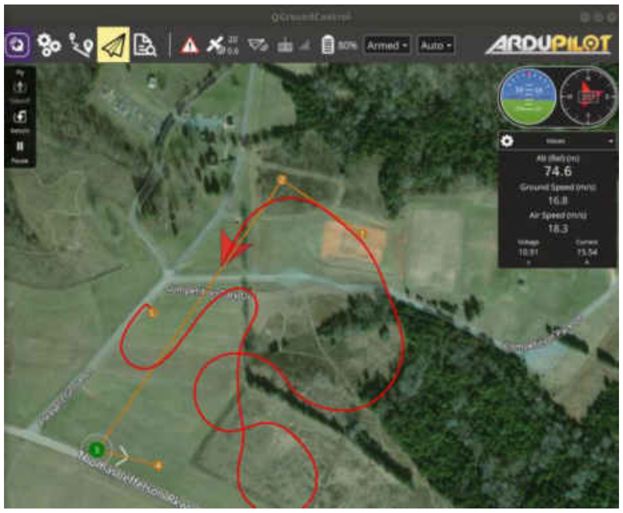


Figure 16-QGroundControl telemetry display and map while the EdgeTRA is in flight

In addition to ArduPlane, Python, a high-level scripting language, is used to develop the controllability diagnostic. Python offers plotting tools, dynamic systems, and control toolboxes similar to commercial MATLAB variants with serial connection interfaces. The main benefit is that Python runs on most operating systems and can be used on small single-board computers such as the RPI. A Python-based communication framework had already been developed to communicate from ArduPlane to a companion computer called DroneKit. DroneKit is a Python package that allows a user to send commands and receive data between a companion computer and a Cube Orange flight controller. DroneKit uses Pymavlink, which is the framework that processes Micro Aerial Vehicle messages (MAVLink) to send and receive from the Cube Orange flight controller [45]. There are two general categories of MAVLink messages. The first category contains messages sent from the companion computer to the Cube Orange, such as setting a value to change the vehicle's airspeed, position, and altitude. These messages utilize either COMMAND_INT or COMMAND_LONG encoding structure. COMMAND_INT is essential when the coordinate reference frame is important, such as sending a waypoint location to fly to. COMMAND_LONG is more suitable for sending desired changes in airspeed, dropping a payload, or retracting the landing gear, to name a few examples. The second category is the companion computer receives MAVLink messages from the Cube Orange. As these messages are being sent from the Cube Orange, Pymavlink provides a function called rev_match() to gather the desired message, as many different messages are streaming at the same time. IMU data is an example of the Cube Orange's desired message, which is published under the RAW_IMU message name. Attributes within the RAW_IMU message define the acceleration, angular velocity, and magnetic field for each axis shown in Table 12. Many other messages can also be viewed, such as the RC transmitter commands to the Cube Orange. The full listing of available messages is found in the MAVLink documentation [46].

RAW_IMU

Field Name	Units	Description
Time_usec	us	Timestamp since boot
xacc	m/s^2	X acceleration
yacc	m/s^2	Y acceleration
zacc	m/s^2	Z acceleration
xgyro	rad/s	Angular speed around the X-axis
ygyro	rad/s	Angular speed around the Y-axis
zgyro	rad/s	Angular speed around the Z-axis
xmag	$gauss$	X Magnetic field
ymag	$gauss$	Y Magnetic field
zmag	$gauss$	Z Magnetic field

Table 12-RAW_IMU message contents

Additionally, Sim_vehicle.py, a simulation written in Python, was used [47]. This simulation runs the ArduPlane firmware on a computer as if the Cube Orange was running the firmware. Sim_vehicle.py utilizes Software in The Loop (SITL), where no hardware is used. Local network connections through the computer running the simulation are created, as shown in Figure 17. These local network connections allow developed Python scripts that utilize DroneKit to be connected to the simulation and tested similarly to real hardware. These connections also allow GCS applications, such as Mission Planner or QGroundControl, to connect to the simulated Cube Orange and perform vehicle setup, change a parameter, and view telemetry while the simulation is running.

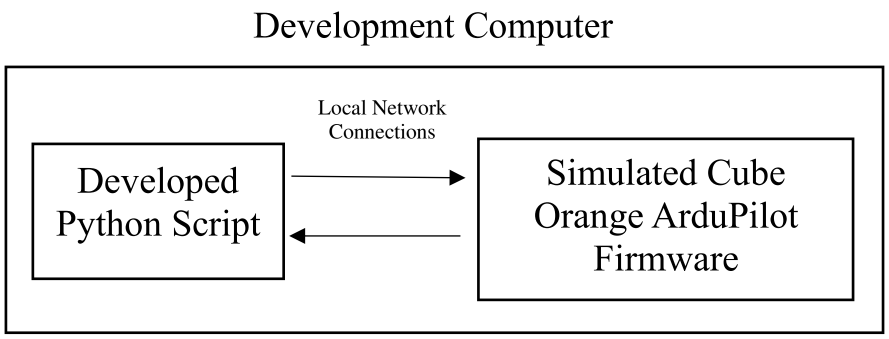

Development Computer
Local Network Connections
Developed Python Script
Simulated Cube Orange ArduPilot Firmware

Figure 17-SITL diagram

The benefit is the ability to test and debug developed Python scripts that control the aircraft. For example, a Python script is developed using the DroneKit package to send MAVLink messages to the Cube Orange to fly to four waypoints in an oval racetrack pattern. Using the SITL reduces the risk in that the waypoints to fly to, altitude, and flight duration can be verified before using any actual hardware. However, Sim_vehicle.py SITL alone can only provide a two-dimensional view of the aircraft flight path, as shown in Figure 18. This two-dimensional view limits the ability to see how an aircraft behaves in the roll, pitch, and yaw axis.

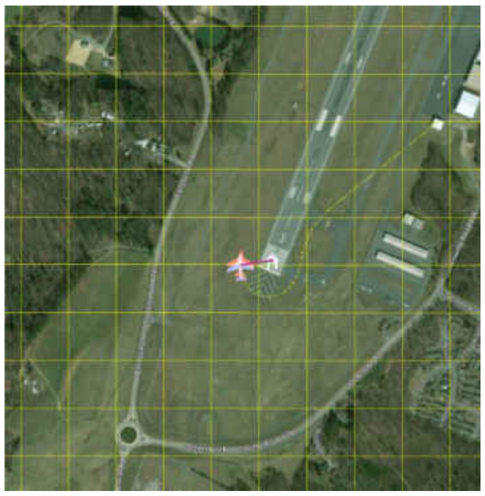


Figure 18-SITL map view during flight simulation

Visual aids from 3rd party 3D flight simulators can be connected to Sim_vehicle.py. This work used X-plane 10, an aircraft simulator typically used for full-scale aircraft and supports model aircraft, such as the Great Planes 40 high wing trainer shown in Figure 19. This three-dimensional view provides an inflight experience that allows all the control surfaces to be observed and is particularly useful in testing the described failure modes. Each failure mode implementation could be visually verified.

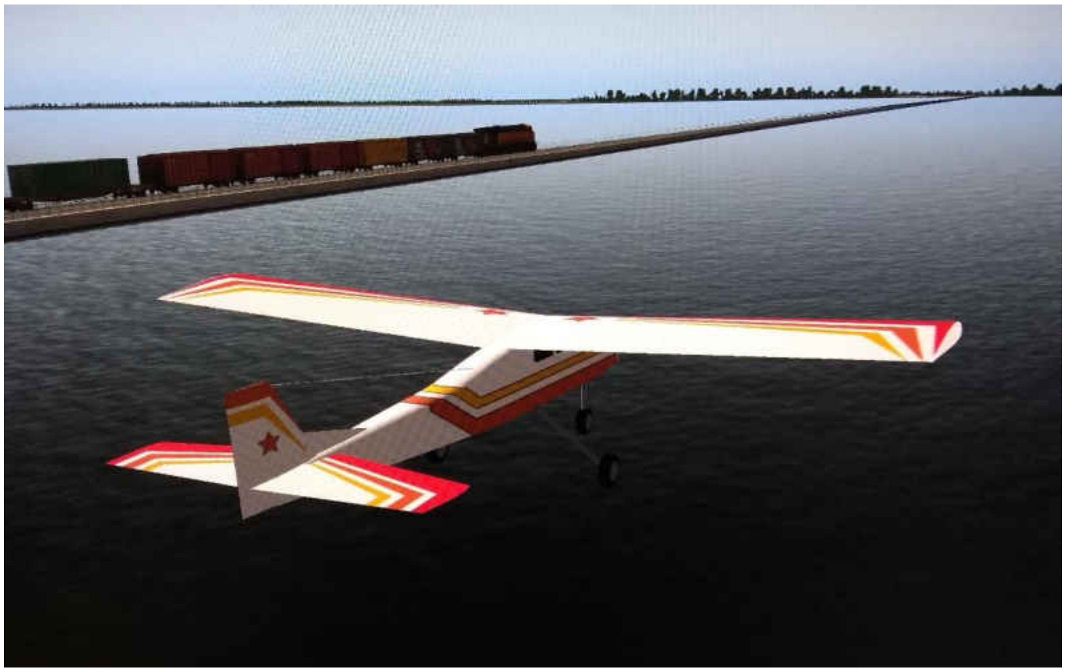


Figure 19-Model of Great Planes high wing trainer in X-plane 10

Lastly, the simulation of the Great Planes 40 was a way to determine the feasibility of the System Identification Package for Python (SIPPY) for building the ARX transfer function angular velocity models. SIPPY is currently one of the few Python packages covering the MIMO transfer function and state-space identification methods of SID [48]. SIPPY is focused on linear modeling methods in the discrete-time domain that utilize only input and output data sets for the *black-box* modeling technique.

CHAPTER 5

EXPERIMENTAL CONFIGURATION AND OPERATION

Flight experiments used three different Python codes developed for this work called the Data Recorder, Servo Failure, and Plane flyer, which could be used either under manual or auto control, as shown in Figure 20. The Data Recorder was used to collect, record, and process any collected data and was used in conjunction with the Servo Failure or Plane Flyer scripts. The Servo Failure code was used to communicate with the Cube Orange to command specific control actuators to stop functioning and how. Plane Flyer communicates with the Cube Orange to upload a four-point mission, change the flight mode, and provide an excitation input. Before each flight, a Secure Shell (SSH) connection is established between the GCS laptop and the RPI companion computer in the EdgeTRA, which allowed for any of these Python scripts to be started in flight if necessary. However, the Data Recorder was always started before the EdgeTRA took off as this code would idle, waiting for pilot input to start or stop taking data with the RC transmitter. More detail on these codes' specific use, manual and auto control methods are included in this chapter's following sections.

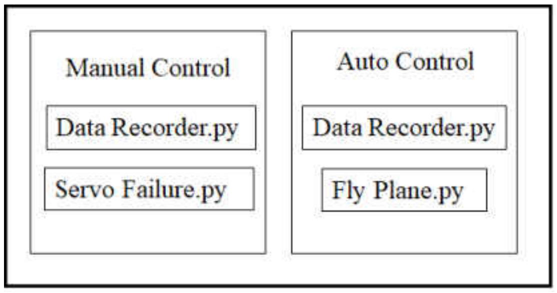


Figure 20-Flight operations types and Python code used with each

5.1 MANUAL CONTROL

Initial flight testing showed that the controllability diagnostic running autonomously would be complex and would require more than one Python script functioning simultaneously. Therefore, initial work focused on the aircraft being manually piloted while the aircraft was underway with the Cube Orange in STABILIZE flight mode. This controllability check's final intent is to use it while the aircraft is under a fully autonomous mode, such as the AUTO mode, where the plane is flying to waypoints. However, while in AUTO mode, the Cube Orange flight controller has its own Proportional, Integral, and Derivative (PID) gains, affecting how the ARX model is built. Therefore, STABILIZE mode is used during manual control testing, and the Cube Orange flight controller is still in the loop, and its effect is captured just as if AUTO mode is used.

In manual control, the basic operation is that the pilot provides some RC input to the aircraft to excite it in a way that is as non-invasive as possible to its trajectory. For example, a roll input that follows a sine wave trajectory allows the aircraft to start neutral roll left or right, depending on the sign convention, and return to neutral. This sine wave input is non-invasive in that the aircraft is left on its original heading when the maneuver is completed. The sine wave input can also be applied to the pitch and yaw axis similarly.

Before starting the input excitations, the pilot flies the plane downwind to the desired altitude of 300ft approximately using an RC transmitter from a 3rd person view and visually checks the aircraft for wings level trim condition. An example of what the pilot would consider wings level was captured using a GoPro Max 360 camera, as shown in Figure 21.

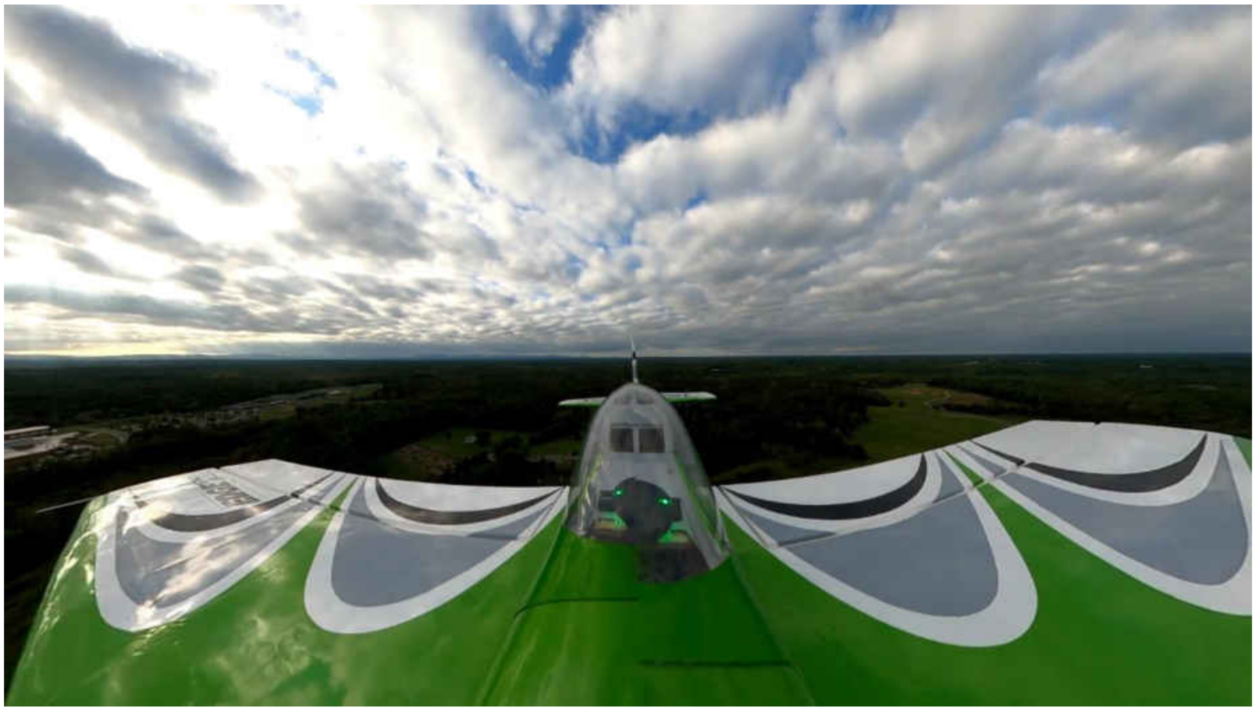


Figure 21-EdgeTRA in wings-level condition

Once these conditions were met, the data recorder was started using an auxiliary switch on the RC transmitter. Approximately two seconds of no excitations were provided to allow the data recorder to capture some trim condition data. After this period, the pilot then executed sine wave inputs to the aircraft via the RC transmitter. First, the roll, then pitch, then yaw was excited in this order one at a time manually. Once the yaw excitation was complete, the aircraft was set back to trim condition for approximately two seconds before data collection was stopped with the RC transmitter. The run's entire duration is about 15 seconds but dependent on how long the pilot spends with each excitation and air space available. This routine is performed two times but with the pilot changing the input excitation slightly each time. This is once to collect data to build the ARX transfer function model and a second time to validate how well the ARX model predicts.

Immediately after the switch on the RC transmitter is set to the stop taking data position, the Data Recorder Python code (shown in APPENDIX B) processes the collected data. If the data collected from the run is the first data set, this data is used to identify the ARX model. Any data sets thereafter use the ARX model to estimate the angular velocity responses. The data recorder also calculates the TIC values for each data set and creates pertinent plots of the data collected. Therefore, just after two laps around the field, roll, pitch, and yaw angular velocity models have been built with data collected on the first lap and validated with collected data on the second lap.

5.2 AUTOMATIC CONTROL

Automatic control was used to fly the aircraft in an oval racetrack pattern similar to the manual control mode. Auto control is done using the Fly Plane Python code shown in APPENDIX D. The pilot manually takes off and flies to an altitude of approximately 300 ft. From this point, the ground control station operator starts the Fly Plane Python script and the Data Recorder script. The Fly Plane script performs multiple tasks. First, using the GPS coordinates from where the EdgeTRA is initially powered, a home point is established. Four waypoints relative to the home location form a rectangle approximately 1,000 ft x 400 ft, as shown in Figure 22. This mission is then sent from the RPI to the Cube Orange, and the Cube Orange flight mode is set to AUTO, all via the Fly Plane script.

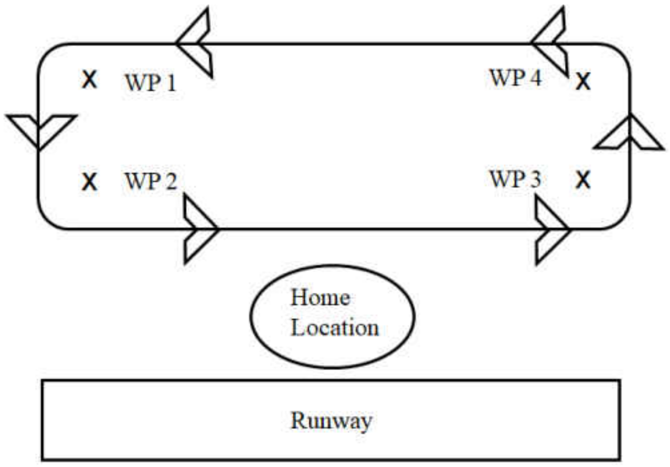


Figure 22-Auto control waypoints and flight path

While en route, the Fly Plane script is responsible for providing the excitations to the EdgeTRA in a similar manner to the manual control method. However, these excitations were only to be performed on the straightaway section between WP two and three, as shown in Figure 22. Waypoint three is established to be the target waypoint. Therefore, when the Plane Flyer script reads from the Cube Orange that the next waypoint is three, excitations are introduced. However, as the plane flies from waypoint one to two, the Cube Orange accepts that waypoint two had been reached prematurely due to acceptance criteria that waypoint two has been reached. Prematurely accepting waypoint two being reached is problematic as the next waypoint is the target heading, and the aircraft is still turning to achieve the target heading when the sine wave excitations are performed. Therefore, to know when the EdgeTRA is to start sine wave maneuvers, a method is developed, as shown in Figure 23. Since the coordinates of the target waypoint (waypoint 3) and the airplane are known from GPS, the desired heading relative to these coordinates is calculated. Then the desired heading can be compared to the actual heading of the EdgeTRA. The difference between the two vectors is called theta. If theta is ± 10 degrees and the next waypoint is three, then it is known that excitations can be started. Also, as the EdgeTRA is in flight, the desired heading is calculated every tenth of a second.

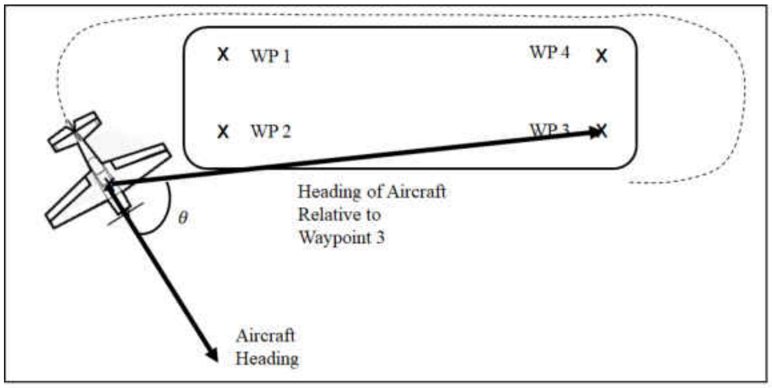


Figure 23-Aircraft target heading determination diagram

Once the EdgeTRA is between waypoints two and three, the Fly Plane script starts the Data Recorder by sending a low PWM signal on the same channel the pilot uses in the manual control method. A few seconds of delay is allowed to collect neutral conditions, then sine wave inputs are sent to the Cube Orange from the Fly Plane script using the RC_OVERRIDE MAVLink message. Sine wave inputs for roll, pitch, and yaw are excited independently in this order. After excitations are completed, the data recorder is stopped, and the collected data is processed, which all happens before reaching waypoint three. Similar to the manual control method, the first data set collected is used to build the angular velocity models. A second data set is used for the validation of the models. After the second data set is collected, the aircraft is manually landed by the pilot.

5.3 FAILURE MODES

Failure modes are tested by having the pilot take off and climb to approximately 300 ft. Just as before, two laps around an oval track pattern are performed. However, in this case, the first lap is used to build the angular velocities model. The applied sine wave excitations are the same as before where roll, pitch, and then yaw are independently excited in that order. Before the second circuit, the GCS operator executes the Servo Failure Python script (found in APPENDIX C). This script requires the GCS operator input for failure mode to enable and duration. A message reports on the GCS operator's screen once the desired surface is failed. The timing of this is critical. If the failure mode starts too early, the selected failure mode time duration may expire before maneuvers are complete. Therefore, the aircraft is under normal conditions when the test for abnormal conditions is in progress. The duration of the failed control surface or surfaces can be increased, but this runs the risk the aircraft still has a failed control surface after data collection is complete, making it hard to control when resetting to collect more data. To mitigate these issues, the EdgeTRA is loitered near waypoints 1 and 2, as shown in Figure 24. Then the GCS operator executes the Servo Failure Python script with the duration set to 15 seconds. Once the GCS operator reports the failure has occurred, the PIC immediately stops loitering, starts the data recorder, and flies towards waypoint three, performing excitations en route. After excitations are completed, the data recorder is then stopped, normal and abnormal data are compared, and the EdgeTRA is landed.

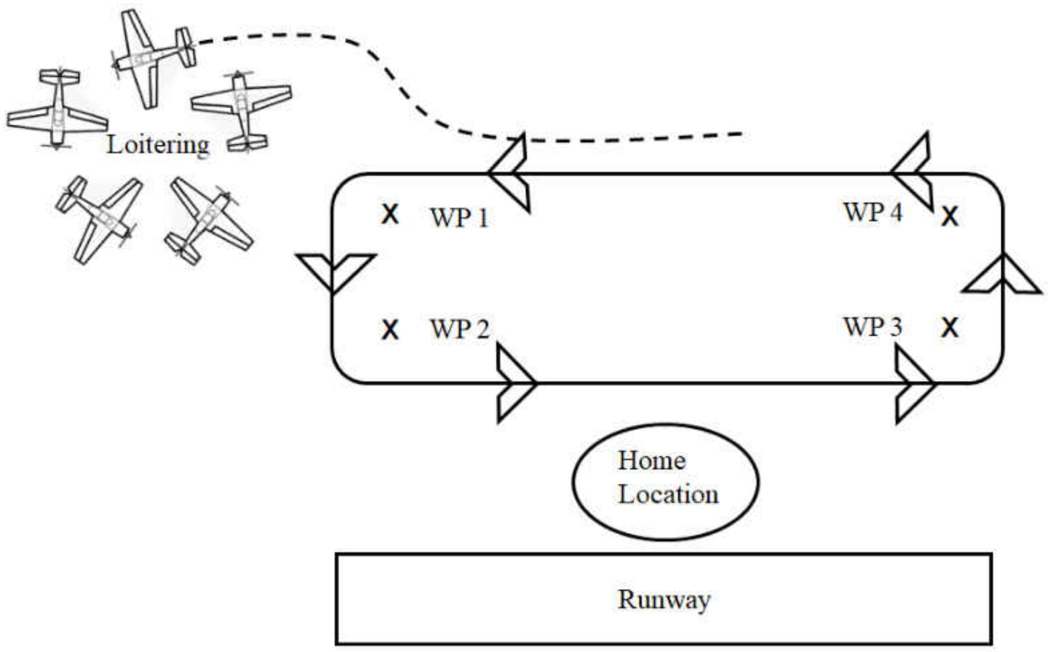


Figure 24-Flight path with failure modes

CHAPTER 6

DATA COLLECTION

6.1 SENSORS AND DATA COLLECTED

For this work, the use of specialty sensors such as strain gauges is to be excluded so that the typical user can implement the controllability diagnostic. Therefore, all collected data must be provided by the Cube Orange flight controller and its auxiliary sensors. Table 13 shows the available sensors that can be used for the controllability diagnostic. Many of these sensors are redundant between the Cube Orange and the auxiliary Here 2 GPS module. This redundancy is needed due to a lack of space requiring the Cube Orange to be installed near other wires, equipment, and metallic aircraft structure. This proximity to metallic objects causes errors in the compass readings. The Here 2 module requires a clear view of the sky. Therefore, it is mounted in the open, reducing compass interference. Additionally, these sensors' redundancy allows the ArduPlane health diagnostic to perform checks on the listed sensors for correct operation.

Cube Orange	
Accelerometer	ICM20948 / ICM20649 / ICM20602
Gyroscope	ICM20948 / ICM20649 / ICM20602
Compass	ICM20948
Barometric Pressure Sensor	MS5611 ×2
Here 2 GPS	
GPS	72-channel u-blox M8N /QZSS L1C/A
Accelerometer	ICM20948
Gyroscope	ICM20948
Compass	ICM20948
Barometric Pressure Sensor	MS5611
Auxiliary Sensors on I2C Bus	
Airspeed	4525 Digital Pressure Transducer

Table 13-Available sensors for controllability diagnostic

In Chapter 3, the relevant data to collect was presented and shown to be gyroscope data, commands to the servos, and aircraft airspeed to model angular velocities, as shown in Figure 25.

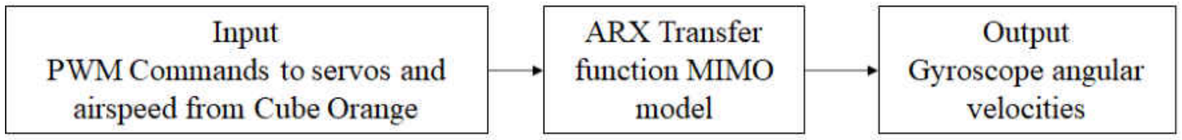

Input
PWM Commands to servos and airspeed from Cube Orange
ARX Transfer function MIMO model
Output
Gyroscope angular velocities

Figure 25-Data used as the input and output to the ARX MIMO model

Table 14 shows all the data collected, such as all PWM commands into the Cube Orange marked by RC_Channel_X while the Cube Orange's output commands are denoted as Servo_X.

RC_Channel_1 (PWM)	Ailerons
RC_Channel_2 (PWM)	Elevators
RC_Channel_3 (PWM)	Throttle
RC_Channel_4 (PWM)	Rudder
RC_Channel_5 (PWM)	Cube Orange flight mode select
RC_Channel_6 (PWM)	Data record start and stop
Servo_1 (PWM)	Aileron one
Servo_2 (PWM)	Elevator one
Servo_3 (PWM)	Throttle
Servo_4 (PWM)	Rudder
Servo_5 (PWM)	Aileron Two
Servo_6 (PWM)	Elevator Two
Servo_7 (PWM)	Rudder Two
Angular velocity x (rad/sec)	Roll rate
Angular velocity y (rad/sec)	Pitch rate
Angular velocity z (rad/sec)	Yaw rate
Airspeed (m/s)	Aircraft airspeed

Table 14-Data collected from Cube Orange

The inputs are RC commands from the PIC, while the output is an altered signal depending on the flight mode. For example, there is no flight control algorithm in MANUAL mode, and the controls are directly passed without alterations. In this work, data collection is either occurring in a STABILIZED or AUTO mode. Both modes alter the RC input to the Cube Orange as the control algorithm tries to maintain level flight due to windy conditions or is navigating to a waypoint. Therefore, to account for this alteration in the input due to the Cube Orange control algorithm, the output to the servos is utilized as the input to the ARX model, as shown in Figure 26.

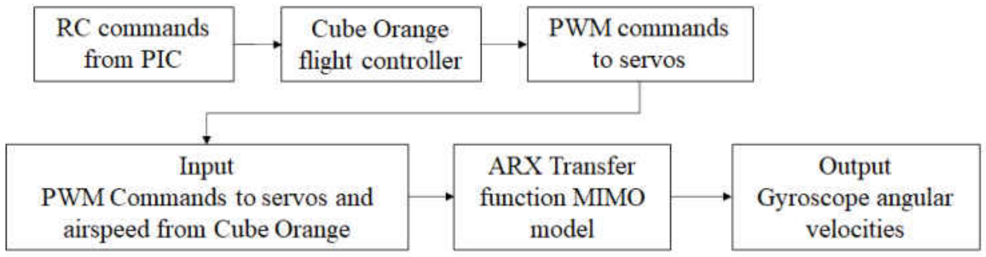

RC commands from PIC
Cube Orange flight controller
PWM commands to servos
Input PWM Commands to servos and airspeed from Cube Orange
ARX Transfer function MIMO model
Output Gyroscope angular velocities

Figure 26-Cube Orange input vs. output

Additionally, as shown in Table 14, RC_Channels 5 and 6 are collected for debugging to ensure the desired flight mode and proper state of the data recorder were achieved during a run. Angular velocity data were collected from only one of the three gyroscopes, as the MAVLink protocol used to collect the data is limited by the number of messages and transmission rate. Therefore, additional data such as battery voltage, location, altitude, and a plethora of other telemetry data were recorded on the SD card of the Cube Orange. This data is important, but only data required to build the ARX transfer function models and perform the controllability diagnostic is collected on the RPI companion computer for further processing.

6.2 MAVLINK MESSAGES

MAVLink messages are used to communicate with the Cube Orange to either send or receive data. This type of serial communication is used with the Cube Orange and is widely used in other flight controller platforms as well, making a data collection system built around MAVLink messages versatile [46]. Traditionally, MAVLink messages are used in conjunction with a telemetry radio pair, allowing GCS to send commands and receive telemetry data from a flight controller. In this work, MAVLink messages are transmitted over a wire directly between the RPI and Cube Orange. Table 15 shows a list of the messages used. In the received column, the previously discussed RC_CHANNELS_RAW, SERVO_OUTPUT_RAW, VFR_HUD, and RAW_IMU were used in the ARX angular velocity model building. Also, the PARAM_VALUE messages are used during failure modes of operation to determine if the failure mode sent to the Cube Orange is received. In the transmitted column is all messages sent via a Python script running on the RPI. The RC_OVERRIDE message provides RC input to the Cube Orange as if an RC Transmitter is used, which is essential when building models autonomously as there is no human interaction, and excitation is required.

Received	**Transmitted**
RC_CHANNELS_RAW	RC_OVERRIDE
SERVO_OUTPUT_RAW	MAV_CMD_DO_SET_SERVO
VFR_HUD	PARAM_SET
RAW_IMU	MAV_DATA_STREAM
PARAM_VALUE	

Table 15-List of MAVLink messages used to receive and transmit information

MAV_CMD_DO_SET_SERVO message is used during the implementation of a stuck control surface failure mode. This message is sent with a desired servo output number and PWM value to drive the servo. As a safety feature, ArduPlane does not allow MAV_CMD_DO_SET_SERVO to be used on any servo output of the Cube Orange designated for flight control. Therefore, this message is inoperable on any output of the Cube Orange listed as aileron, elevator, throttle, and rudder. The PARAM_SET message is used to work around this by temporarily changing the servo output assignments. Then the MAV_CMD_DO_SET_SERVO message can be implemented to set a servo to the desired PWM value. For example, to fail aileron two, which is physically connected to servo output five on the Cube Orange, the PARAM_SET message is set to temporally change the function of servo output five from aileron to *none*. Setting the function of output five to *none* allows the MAV_CMD_DO_SET_SERVO to be implemented, simulating a stuck control surface failure. Once the failure is complete, PARAM_SET is used to return the function of SERVO five to its original state nullifying the failure. For the limited travel failure mode, only the PARAM_SET message is used to reduce the allowable throw limits of the desired servo, and it is also used to revert the limited failure mode to nominal conditions.

Lastly, MAV_DATA_STREAM is used to set the Cube Orange rate to transmit MAVLink messages from its ports. ArduPlane separates the data into eight categories, with a data rate assigned to each category, as shown in Table 16. For this work, only RAW_SENSORS, RC_CHANNELS, and EXTRA2 are needed. Therefore, the remaining categories' data rates were set to zero. As each category's rate was increased, or as more categories were added, the maximum attainable rate for all categories was affected. For example, if all categories are set to a requested rate of 50Hz, the RAW_SENSORS category can only be received at 15Hz. The other categories such as the RC_CHANNELS are affected as well. Setting unnecessary categories to requested data rates of 0 Hz, the RAW_SENSORS category is found to be the requested rate of 50Hz. Therefore, limiting to only the necessary categories, RAW_SENSORS, RC_CHANNELS, and EXTRA2, allowed the data to be collected at 50 Hz, 25Hz, 25Hz, respectively.

MAV_DATA_STREAM	
RAW_SENSORS	IMU, Compass, Location
EXTENDED_STATUS	
RC_CHANNELS	RC_Input, Servo_Output
RAW_COTROLLER	
POSITION	
EXTRA1	
EXTRA2	Airspeed Sensor
EXTRA3	

Table 16-Attributes of MAV_DATA_STREAM

6.3 RASPBERRY PI FLIGHT COMPUTER

The RPI 3B is a lightweight, compact single-board computer that runs the Raspbian Stretch operating system using 1GB of RAM and a Quad-Core 1.2Ghz BCM2837 64 CPU. As a companion computer to the Cube Orange, the RPI runs the developed Python scripts explained in Chapter 5. Figure 27 shows an overview of the three Python scripts that run on the RPI, which are used to control the Cube Orange and collect all data via MAVLink messages.

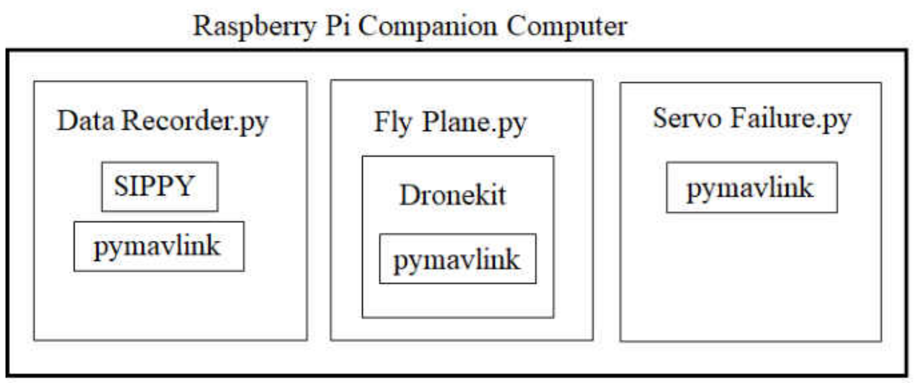


Figure 27-Developed Python scripts that run on the RPI

The RPI offers four USB serial ports, as shown in Figure 28. General Purpose Input Output (GPIO) and I2C pins are just a few. For data collection, one of the four USB serial ports is devoted to the Data Recorder.py script. A second USB port is used for either the Fly Plane.py or the Servo Failure.py scripts, while the remaining ports are used for communicating with the RPI over the RNode radio SSH connection to start and stop the developed Python scripts. Also, to aid in data processing, the time and date of each run were collected. A real-time clock (RTC) was added to the RPI, as usually the RPI syncs the date and time when connected to the internet, but that is not the case, of course, in flight. A PCF8523 real-time clock is used to keep the date and time, even after shutdown. Therefore, all collected data sets are saved with the time and date, allowing the data to be compared with the flight log notes if there is a discrepancy.

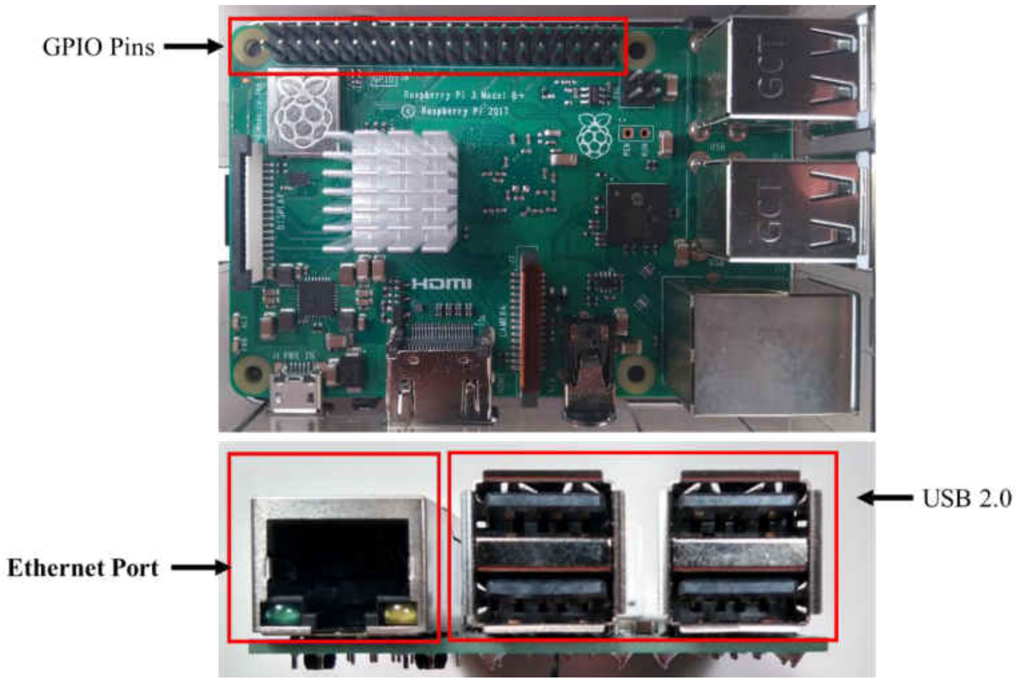


Figure 28-Raspberry Pi USB ports used to connect to Cube Orange

6.4 START AND STOP OF DATA COLLECTION

The data collection process needed to be dynamic, in that the time duration between the start and stop was not always the same due to imperfect human excitation inputs and delays in Python scripts. For example, when the PIC would provide the sine wave input in the manual control mode, the duration of the time spent rolling the aircraft can vary from time spent exciting pitch and yaw. Therefore, if the data recorder only collects data for a predetermined period and the PIC has not finished the input, then only a portion of the run is collected. The same is also true for when the Fly_plane.py code is providing the inputs. In the event the Fly_Plane.py code is delayed, not all of the input commands would be captured if the data recorder only collects data for a fixed period. Therefore, the Data Recorder.py script was made to run continuously in the background waiting for a command from an RC transmitter switch. The data recorder continuously monitors channel 6 of the Cube Orange RC input, controlled by a three-position switch on the RC transmitter. If the switch sends a low-PWM value, this tells the data recorder to start collecting data. When a high-PWM value is received, the data recorder stops taking data, and the data is further processed. However, in the case that an excitation maneuver did not go as intended, the middle position of channel 6 is used, sending a mid-PWM value of 1500, instead of a high-PWM. A mid-PWM value stops the data recorder but does not process or save any of the data. Doing this allows another run to be made in that the data recorder idles until the low PWM values are received again. Every time a low value is seen, any previous data that has not been processed is cleared. This process worked for both manual and auto control methods. However, in auto control, channel 6 is controlled using the RC_OVERRIDE MAVLink messages rather than the three-position switch on the RC transmitter.

CHAPTER 7

DATA PROCESSING

One of the requirements for this work is that all data processing is to be done while in flight, and all data is saved and processed using the onboard RPI. After the data is collected, it is first discretized. MAVLink messages are secondary to any flight control computations within the ArduPlane firmware architecture, meaning the rate at which data is collected may not be constant. Figure 29 shows this inconsistent data rate for the IMU, RC Channels input and output, and airspeed categories, respectively, versus the number of MAVLink messages collected. However, the average message rate is the requested rate of 50 Hz for IMU data, 25 Hz for RC channels data, and 25 Hz for airspeed data. This nonconstant data rate is problematic, as the change in time for discrete transfer functions must be fixed intervals when building ARX models of the roll, pitch, and yaw angular velocities.

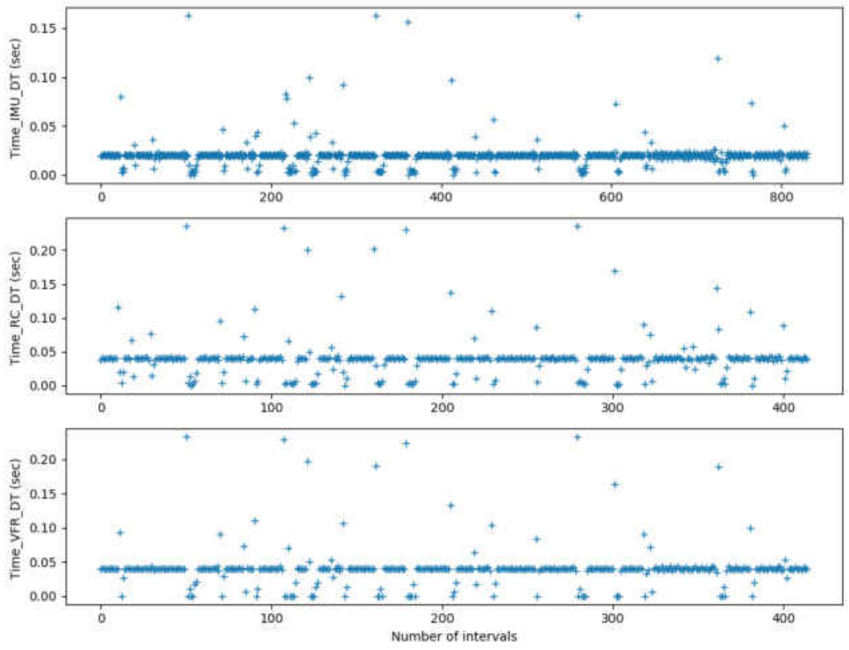


Figure 29-Change of time between MAVLink messages

Therefore, linear interpolation is used to fix the data into discrete intervals. Before interpolation, a verification process is performed. This checks that the average data rate obtained meets the requested data rate, and if so, interpolation is performed. For example, if the IMU average data rate is ±5 Hz of the requested 50 Hz, then the data set is interpolated. This verification is to ensure the gaps to be interpolated are small. Verification is also done with the RC channels and airspeed messages. However, the verification is for ±5 Hz of the requested 25 Hz data instead of 50 Hz for IMU messages. Another need for interpolation is to make the input and output data set arrays the same length. Since the input data, RC channels, and airspeed are collected at 25 Hz while the output data, IMU, is at 50 Hz for a given period, there are only half the input data points compared to the output data points. Therefore, the inputs are interpolated to provide 50 Hz data, making the input and output data sets arrays the same length. The data is then passed to SIPPY where the inputs and outputs are used to build the ARX transfer function model.

This interpolation process is visually verified, as shown in Figure 30. On the y-axis, the input to the aileron, elevator, throttle, rudder, and the measured airspeed is shown. The x-axis is the time in seconds since the Cube Orange has been powered. Interpolation verification is provided by the blue plus and orange triangle symbols. The blue plus symbols represent data in the raw form where the change in time is not discrete, while the orange triangles are the interpolated data points in discrete time intervals of 0.02 sec. Since the symbols overlap, an informal verification of interpolation is provided.

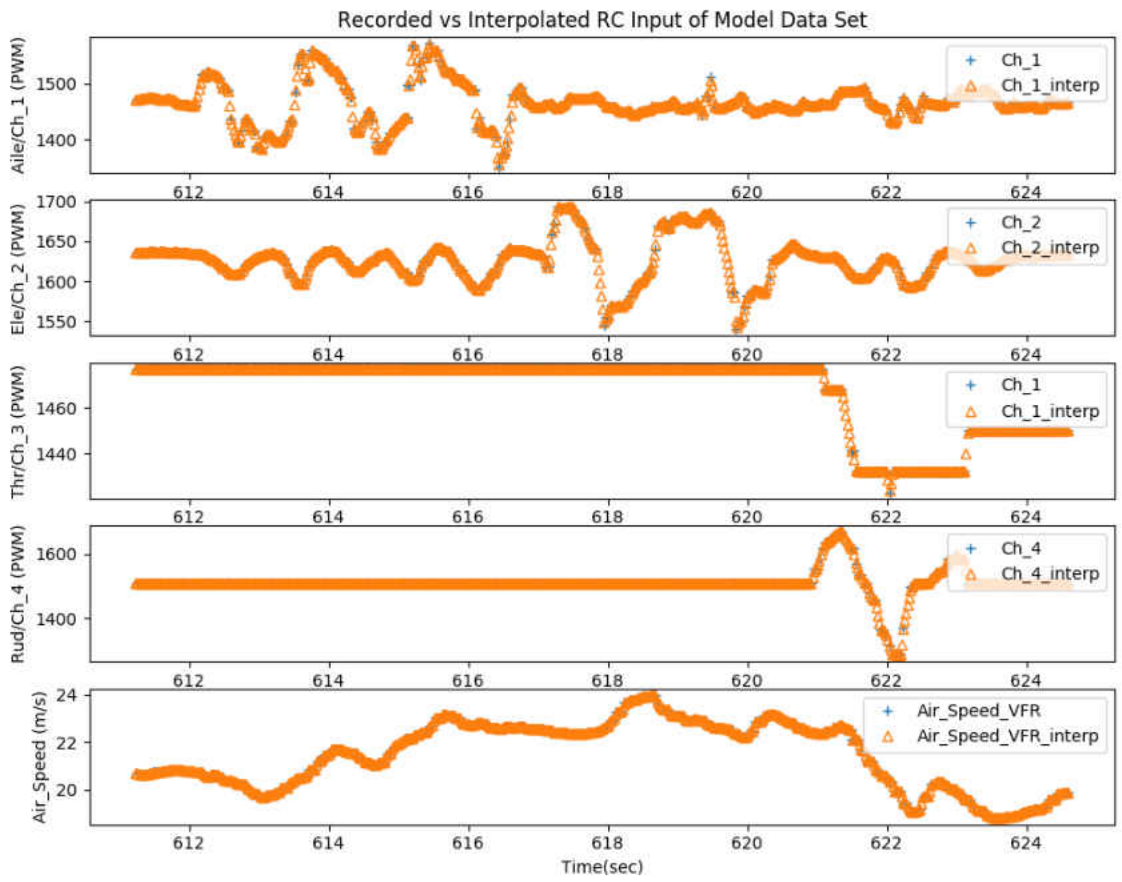
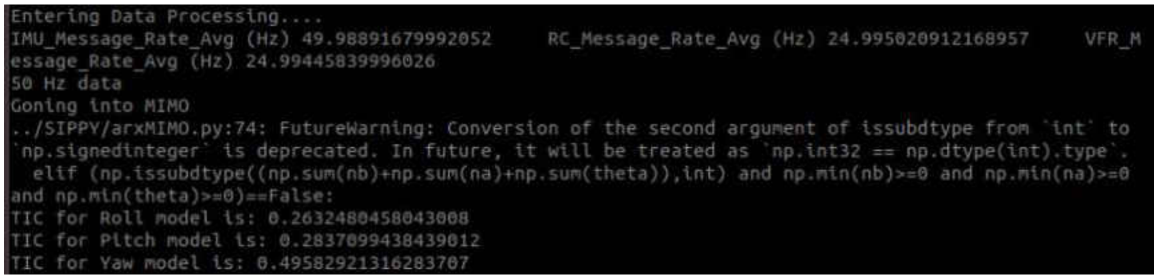


Figure 30-Interpolation verification from MIMO_Model_Input_03_14_2020__15_28_51

Data were temporarily stored in memory on the RPI during the data collection process, and for data to be saved for future post-processing, it is saved in a CSV file format in three different files using Pandas, a Python library. The first saved file contains data in its raw form, while the data used to build the ARX model and the identified MIMO transfer function is saved in a second CSV file. The third CSV file contains data used to validate the model and TIC results. Each CSV file is saved with the name as the time and date in the 24-hour clock format and dependent on the run; they are sorted into a folder named “Model” or “Validation”. Saving the data this way allows for a model building run to be paired with its respected validation run. Data processing also included calculating the TIC values and saving them in their respective

CSV files, whether for model building or validation. However, TIC is also printed on the GCS operators screen for inflight fit performance evaluation of the runs, as shown in Figure 31.



```
Entering Data Processing....
IMU_Message_Rate_Avg (Hz) 49.98891679992052     RC_Message_Rate_Avg (Hz) 24.995020912168957     VFR_M
essage_Rate_Avg (Hz) 24.99445839996026
50 Hz data
Goning into MIMO
../SIPPY/arxMIMO.py:74: FutureWarning: Conversion of the second argument of issubdtype from `int` to
`np.signedinteger` is deprecated. In future, it will be treated as `np.int32 == np.dtype(int).type`.
  elif (np.issubdtype((np.sum(nb)+np.sum(na)+np.sum(theta)),int) and np.min(nb)>=0 and np.min(na)>=0
and np.min(theta)>=0)==False:
TIC for Roll model is: 0.2632480458043008
TIC for Pitch model is: 0.2837099438439012
TIC for Yaw model is: 0.49582921316283707
```

Figure 31-SSH terminal screen from RPI on the GCS reporting TIC values

Furthermore, plots of all collected models and validation data are created and saved for further inspection if need be. The plots include data in the raw format vs. interpolated data to inspect for proper interpolation. Also, model predictions and measured angular velocity are overlaid on one another for a visual inspection of the fit, which gives the ability to quickly check the fit of the model versus the measured angular velocity following the landing of the EdgeTRA. These plots are saved as PNG files similarly to the CSV files in that the time and date is used as the file name and sorted into folders of “Model” or “Validation” as well.

CHAPTER 8

RESULTS

8.1 MANUAL RC CONTROL MODEL BUILDING

As discussed in Chapter 5, manual control uses input excitation commands from the PIC while the EdgeTRA is in the STABILIZE flight mode, including the flight controller algorithm in a similar way AUTO mode would. The first step in this work is to determine a nominal model of the EdgeTRA roll, pitch, and yaw angular velocities. Figure 32 shows all inputs recorded via the Data Recorder.py script. Aileron, elevator, rudder, and airspeed are used as input data to build an ARX model. It is shown that the input excitation occurs for roll, pitch, and yaw in that order with respect to time. The inputs applied are attempted sine waves from the PIC between 0.5 and 1Hz frequencies. However, the inputs applied to the servos are not a smooth sine wave, as the Cube Orange flight controller is in the loop. Therefore, when excitation is not performed on an axis, the input signal is not constant. For example, in the Ele/Ch2 plot between 262 to 266 and 272 to 277 seconds, the elevator servo input is sporadic about a small magnitude. This small change in command is due to the Cube Orange attempting to maintain a constant altitude. Additionally, a maximum bank angle of ± 45 degrees and a pitch limit of ± 30 degrees are configured. In this run, the PIC did not achieve the roll limit, although the pitch limit was achieved, shown in the Ele/Ch_2 plot at 269 sec. An increase in PWM on the elevator channel correlates to the EdgeTRA pitching upward. Therefore, the PIC is commanding the EdgeTRA to pitch up. However, the pitch angle of 30 degrees is achieved, and the Cube Orange flight controller reduces the PWM value to the elevator servo. The reached pitch limit of 30 degrees forms a valley at the peak of the sine wave input, and the purpose of this is to show the importance of using the actual input to the servos after the Cube Orange flight controller. Using the inputs directly via the PIC to the Cube Orange results in improper modeling because this would not account for these described limits.

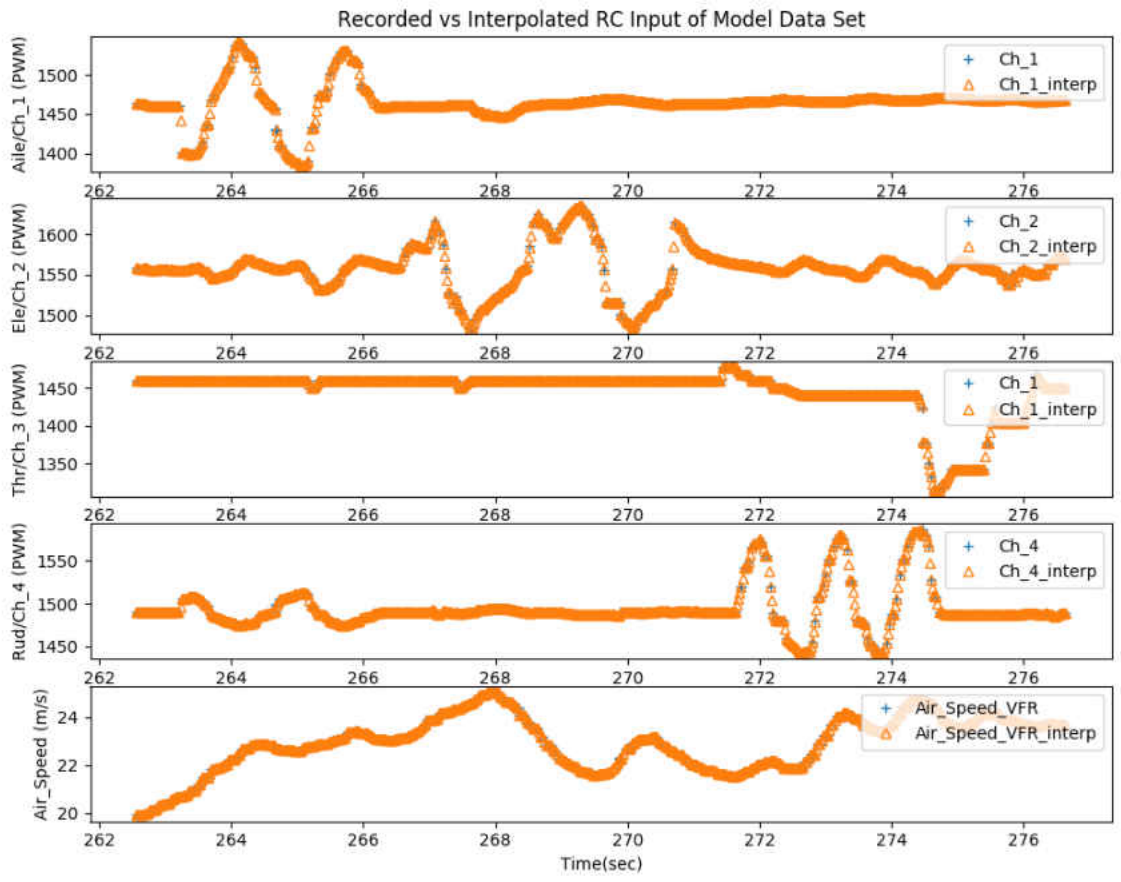


Figure 32-Inputs used to identify ARX model for run MIMO_4_05_2020__18_03_08

Figure 33 shows the output or response from the applied input in Figure 32. Roll, Pitch, and Yaw rate are shown respectively, while the x-axis shows the time since the Cube Orange has been powered. Similar to the applied inputs, the response is not a perfectly smooth sine wave. The discussed limit is achieved when looking at the pitch rate at 269 seconds when there is a change in the pitch rate magnitude. It is also important to note that the aileron is mixed with the rudder movement by 10%. This mixing is used to aid in the navigation of the EdgeTRA while in AUTO mode since there is no active control on the yaw axis. Mixing effects can be seen in the yaw rate response between 262 and 267 seconds while the ailerons are moved. Mixing of the aileron to rudder is only one way, in that if the rudder is moved, the ailerons are unaffected. However, it can be seen in the roll rate plots at 272 and 276 seconds there is some rolling movement when the rudder is excited. This rolling movement is not due to mixing but rather the coupling of the aircraft dynamics.

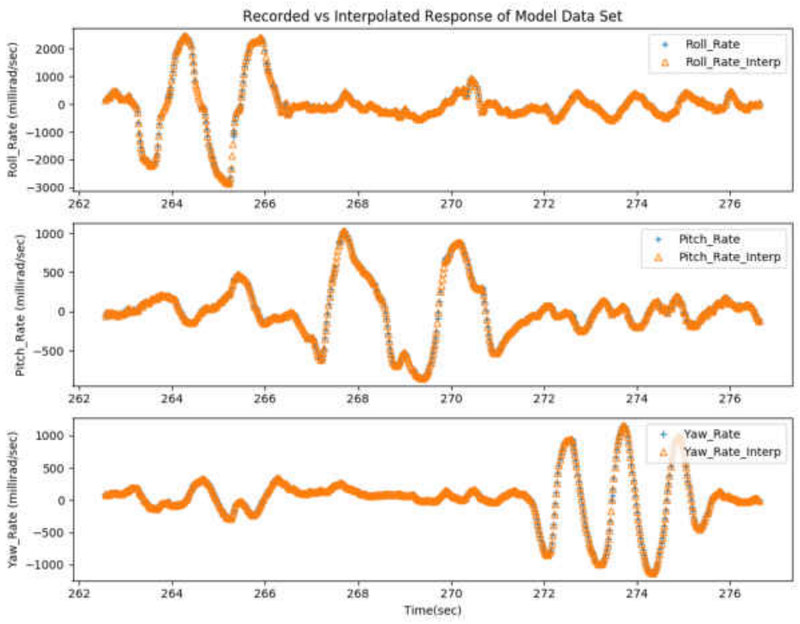


Figure 33-Outputs used to build ARX model for run MIMO_4_05_2020__18_03_08

Using the input and output data from Figure 32 and Figure 33, Table 17 shows the MIMO identified transfer function used to model the response of the roll, pitch, and yaw angular velocities. Developing a MIMO model includes any coupling within the EdgeTRA as each input's effect can be related to each output. In the case of the EdgeTRA, the coupling effect of the aileron and rudder should be negligible due to the zero degree dihedral angle of the wing [49]. However, the rudder may still cause some rolling since the rudder area is not evenly distributed about the longitudinal centerline. Additionally, for each column, it is shown that the denominator has the same coefficients, while the numerator differs. As previously discussed, the transfer function relates the inputs to the outputs. Therefore, the denominator remains the same as the output data remains the same throughout a column, and the numerator changes base on the applied input. For example, focusing on the roll rate output column, the roll rate output data's polynomial is identified and placed into the transfer function's denominator. Then, the aileron input data polynomial is identified and placed in the numerator of the transfer function. This process repeats for the elevator, rudder, and airspeed inputs. However, only the numerator needs to be identified thereafter because the column's roll rate output curve is the same.

Inputs	Roll Rate Output	Pitch Rate Output	Yaw Rate Output
Aileron (PWM)	$\frac{4.148z - 2.435}{z^4 - 1.764z^3 + 0.9605z^2 - 0.1428z}$	$\frac{0.1036z - 0.2063}{z^4 - 1.284z^3 + 0.1198z^2 - 0.2287z}$	$\frac{-0.3238z + 0.3188}{z^4 - 1.674z^3 + 0.453z^2 - 0.2403z}$
Elevator (PWM)	$\frac{0.1924z - 0.5182}{z^4 - 1.764z^3 + 0.9605z^2 - 0.1428z}$	$\frac{-2.386z + 1.592}{z^4 - 1.284z^3 + 0.1198z^2 - 0.2287z}$	$\frac{-0.07169z + 0.06467}{z^4 - 1.674z^3 + 0.453z^2 + 0.2403z}$
Rudder (PWM)	$\frac{0.5386z - 0.851}{z^4 - 1.764z^3 + 0.9605z^2 - 0.1428z}$	$\frac{0.2618z - 0.2458}{z^4 - 1.284z^3 + 0.1198z^2 - 0.2287z}$	$\frac{-1.05z + 1.019}{z^4 - 1.674z^3 + 0.453z^2 - 0.2403z}$
Airspeed (m/s)	$\frac{7.506z - 7.34}{z^4 - 1.764z^3 + 0.9605z^2 - 0.1428z}$	$\frac{21.75z - 21.44}{z^4 - 1.284z^3 + 0.1198z^2 - 0.2287z}$	$\frac{-7.825z + 7.936}{z^4 - 1.674z^3 + 0.453z^2 + 0.2403z}$

Table 17-Identified ARX transfer function model for roll, pitch, and yaw rates

Figure 34 shows the identified model plotted over the measured response. The blue dots represent angular velocity measured from the Cube Orange gyroscope for roll, pitch, and yaw, while the orange plus symbol is the model predicted values. The measured and estimated angular velocities overlap one another well. However, in this figure, the same input data used to identify the ARX model is used to estimate the shown response. Therefore, the fit is expected to be good. A second run is performed to validate this model to show that the modeling works even when a different input is applied.

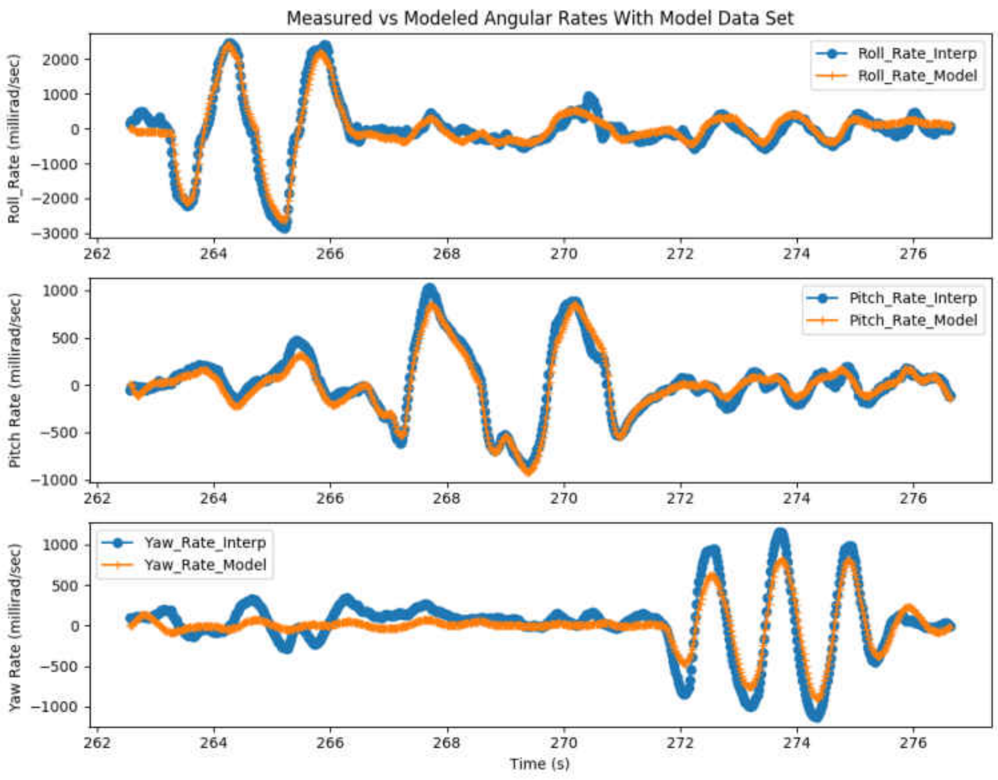


Figure 34-Fitted output using input from the same data used to build the ARX model for run MIMO_4_05_2020__18_03_08

Figure 35 shows the inputs used to validate the previously built ARX model. The inputs are applied similarly as before in that roll, pitch, and yaw are excited in this order. Inputs are still sine waves. However, the frequency has been reduced by about half, and the amplitude varies approximately 25 PWM more than the input used to build the model. Also, in this validation run, no limits were achieved. Therefore, the inputs mimicked the sine wave more in the validation than in the previous model building run.

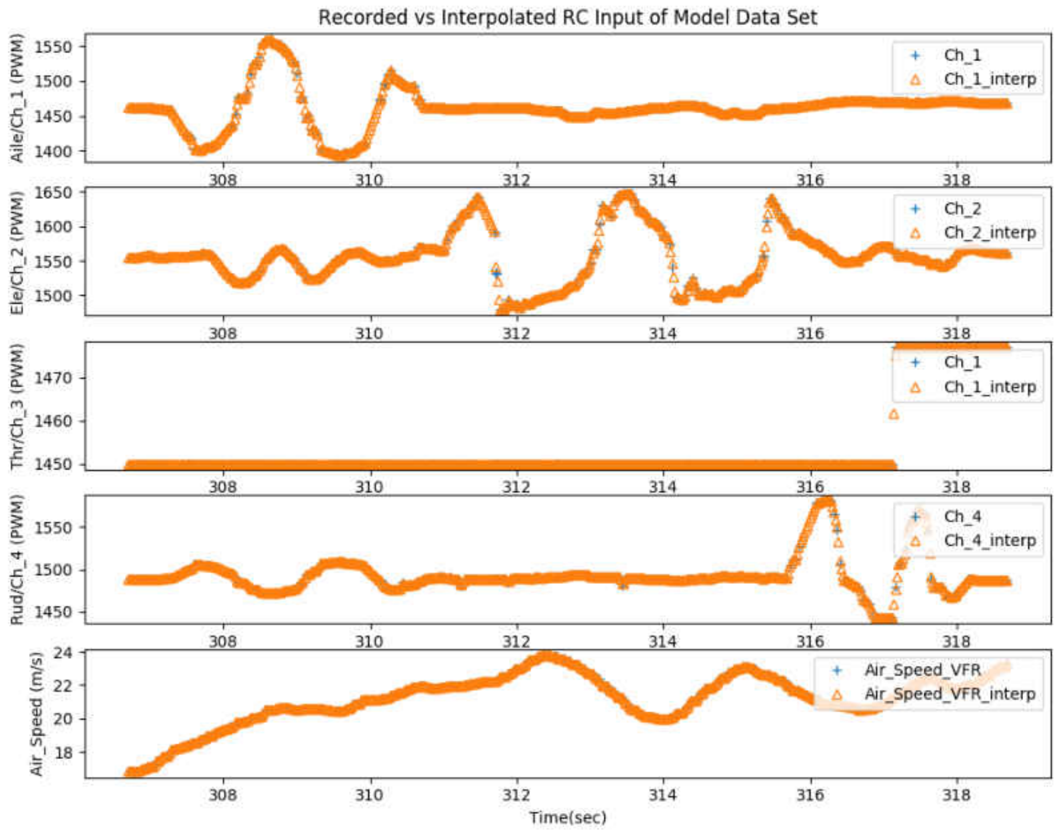


Figure 35-Inputs used to validate ARX model for run MIMO_4_05_2020__18_03_08

Figure 36 shows the outputs, or the response, from Figure 35 validation inputs. An increase in amplitude is seen in the roll rate plot at 309 seconds. The maximum magnitude achieved is 3000 milliradians/sec compared to 2200 milliradians/sec in the model building run. There also is more activity from the Cube Orange to maintain level flight when an input is not applied. Specifically, looking at the roll rate after 311 seconds, the plotted response is jagged. The jagged response is also seen in the pitch rate plot before 311 and after 316 seconds.

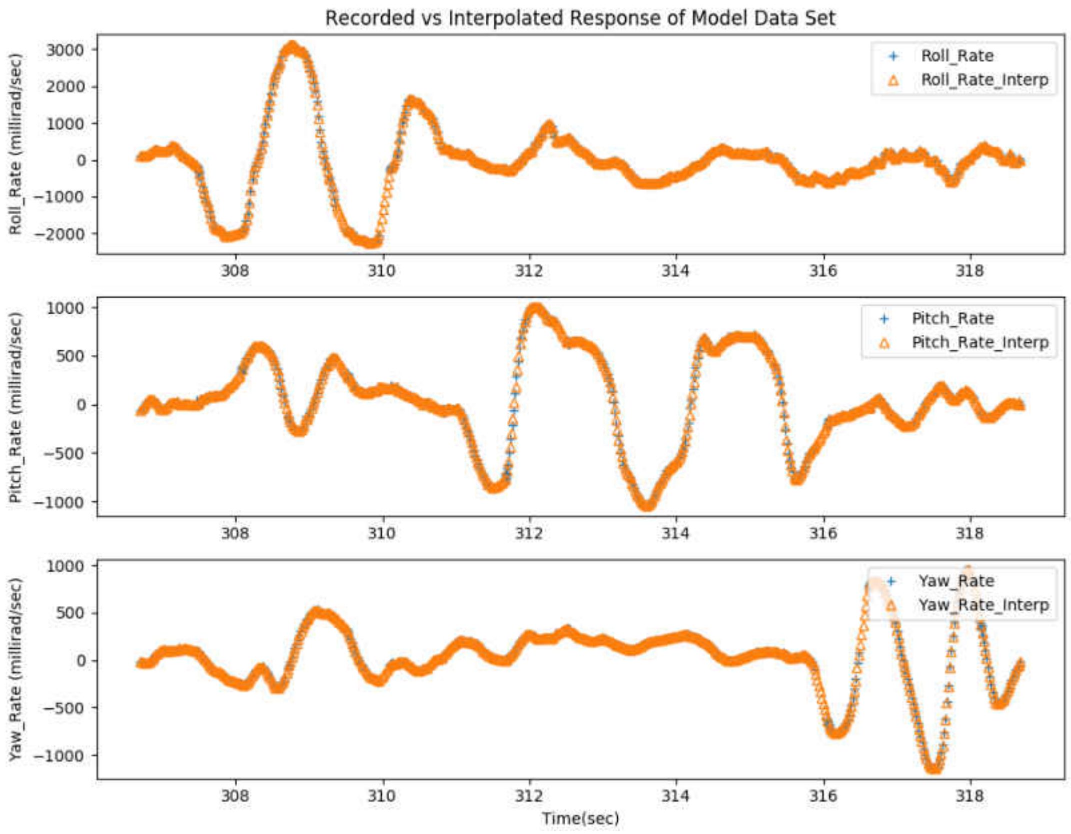


Figure 36-Outputs used to validate ARX model for run MIMO_4_05_2020__18_03_08

Figure 37 shows an overlay of the estimated and measured angular velocities for validation data. The blue dots show measured angular velocities, while the orange plus symbols are estimated angular velocities based on validation inputs. The fit of the two lines visually appears to be suitable for roll and pitch. Due to non-linearity, the yaw rate does not fit well, which is discussed further in the next chapter. For this run, the TIC metric of fit values is 0.126, 0.096, and 0.372 for roll, pitch, and yaw, respectively, which supports the assumption that as a TIC value tends to zero, the fit is considered to be better. Results from Dorobantu et al. are found to be similar with TIC values of 0.12, 0.07, and 0.26 for roll, pitch, and yaw, respectively, using a high-wing ultra stick [31].

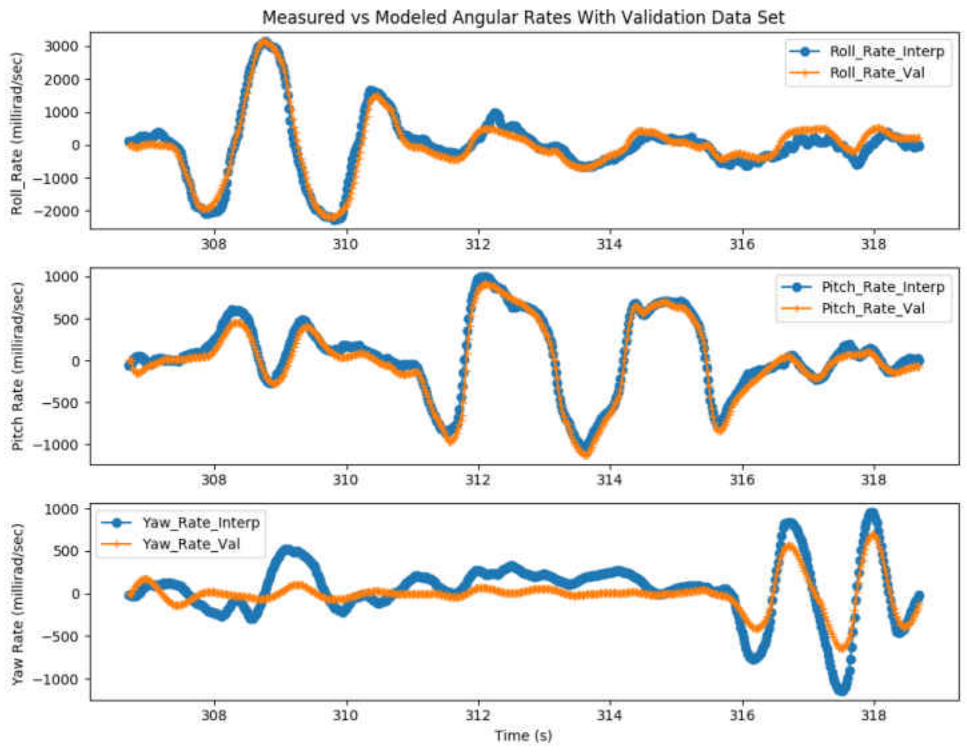


Figure 37-Fitted output using validation input for run MIMO_4_05_2020__18_03_08

In total, 29 runs were performed while under manual control to understand the variance in the TIC value from run to run. Figures 38 through 40 show the TIC values for roll, pitch, and yaw, respectively, vs. the run number from validation runs. Of the 29 runs, four runs were omitted as outliers because the EdgeTRA had reached the end of the field, and the PIC had to abort excitations before completion. There is a general trend that as the run number increases, the TIC values decrease, indicating a better fit of the angular velocity models. The trend is believed to be caused by human errors, such as the PIC is learning to perform the excitations in a more repeatable fashion as the run number increases. Weather also affected this decrease in the TIC coefficient. Runs 1-10 were performed on days where the flight logbook stated wind conditions gusting 11 to 13 mph on the ground. The remaining runs were performed in calm conditions or winds of 3 to 5 mph.

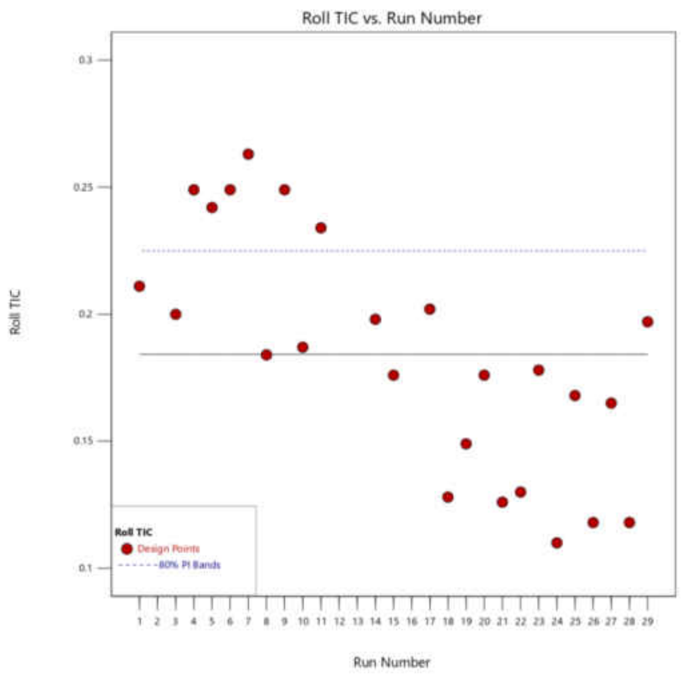


Figure 38-Manual control roll TIC vs. run number

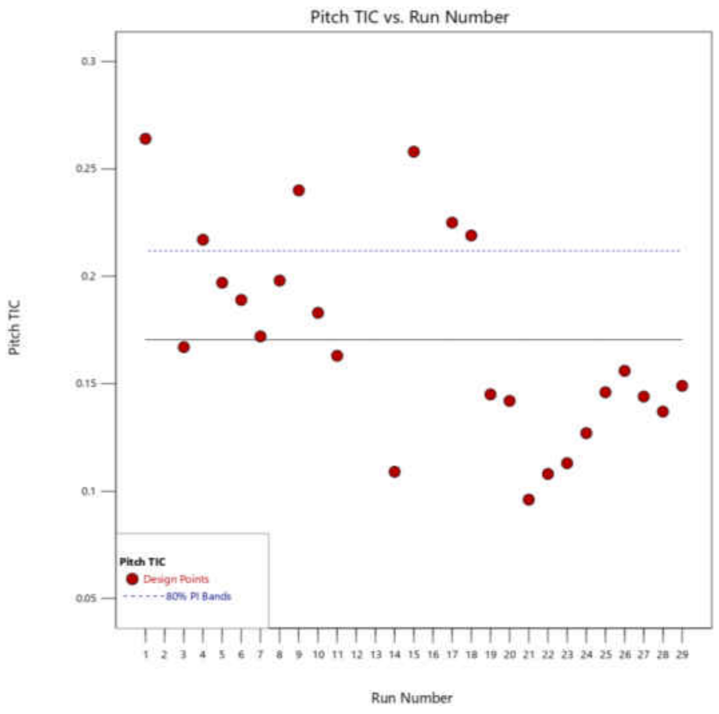


Figure 39-Manual control pitch TIC vs. run number

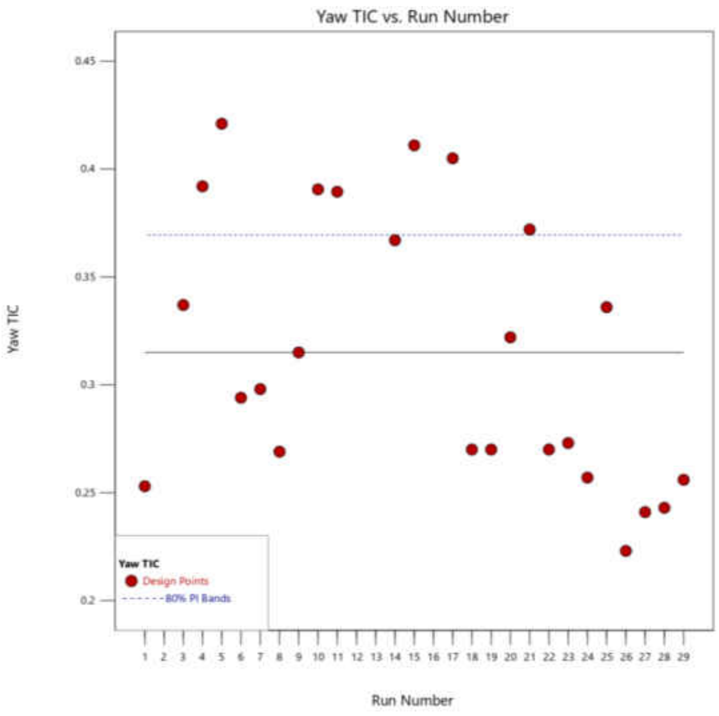


Figure 40-Manual control yaw TIC vs. run number

8.2 AUTOMATIC CONTROL MODEL BUILDING

With the ability to build roll, pitch, and yaw angular velocity models with manual control proven, the focus was shifted to automatic control. Automatic control occurs when the EdgeTRA is flying with no human input, invoking the Fly_Plane.py script, as discussed in Chapter 5. Figure 41 shows the inputs applied via MAVLink messages from the Fly_Plane.py script. Sine wave inputs of 1 Hz, 0.5 Hz, and 1 Hz for roll, pitch, and yaw are applied, respectively. The sine waves' amplitude is 200 PWM about the trim PWM signal used to neutralize the control surface, and excitations were once again performed in the order of roll, pitch, and yaw. Aileron, elevator, rudder, and airspeed are used to build the ARX transfer function model of the inputs shown. Just as in manual control model building, the inputs shown are not smooth sine waves as the Cube Orange flight controller alters the input for stability, navigation, or if a limit is achieved. A reached limit example is shown in the Rudd/Ch4 subplot at 400 seconds; the tops of the sine wave's inputs are truncated. The plateau is caused by the commanded PWM signal being greater or less than the allowable PWM limit set for the rudder channel. Until the commanded PWM signal is back in range, the Cube Orange keeps sending the maximum or minimum PWM signal, which gives the plateau in the input.

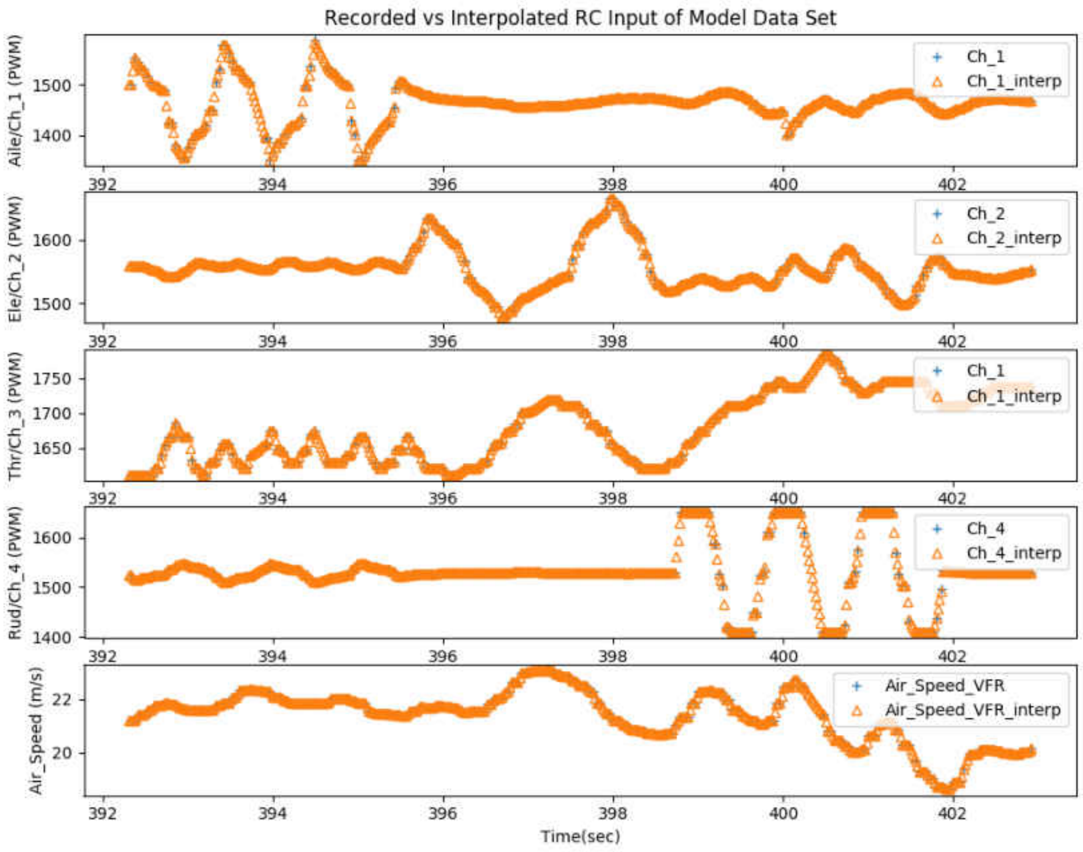


Figure 41-Inputs used to identify ARX model with auto control for run MIMO_6_13_2020__16_18_37

Figure 42 shows the output, or response, to the applied inputs in Figure 41. After 399 seconds, there is more movement in the pitch rate than in manual control runs as the flight controller is attempting to maintain a desired altitude in the AUTO flight mode. This additional movement was deemed insignificant, as it minimally affected the TIC coefficient for the pitch axis. This output is used in addition to the input to identify the ARX transfer function model.

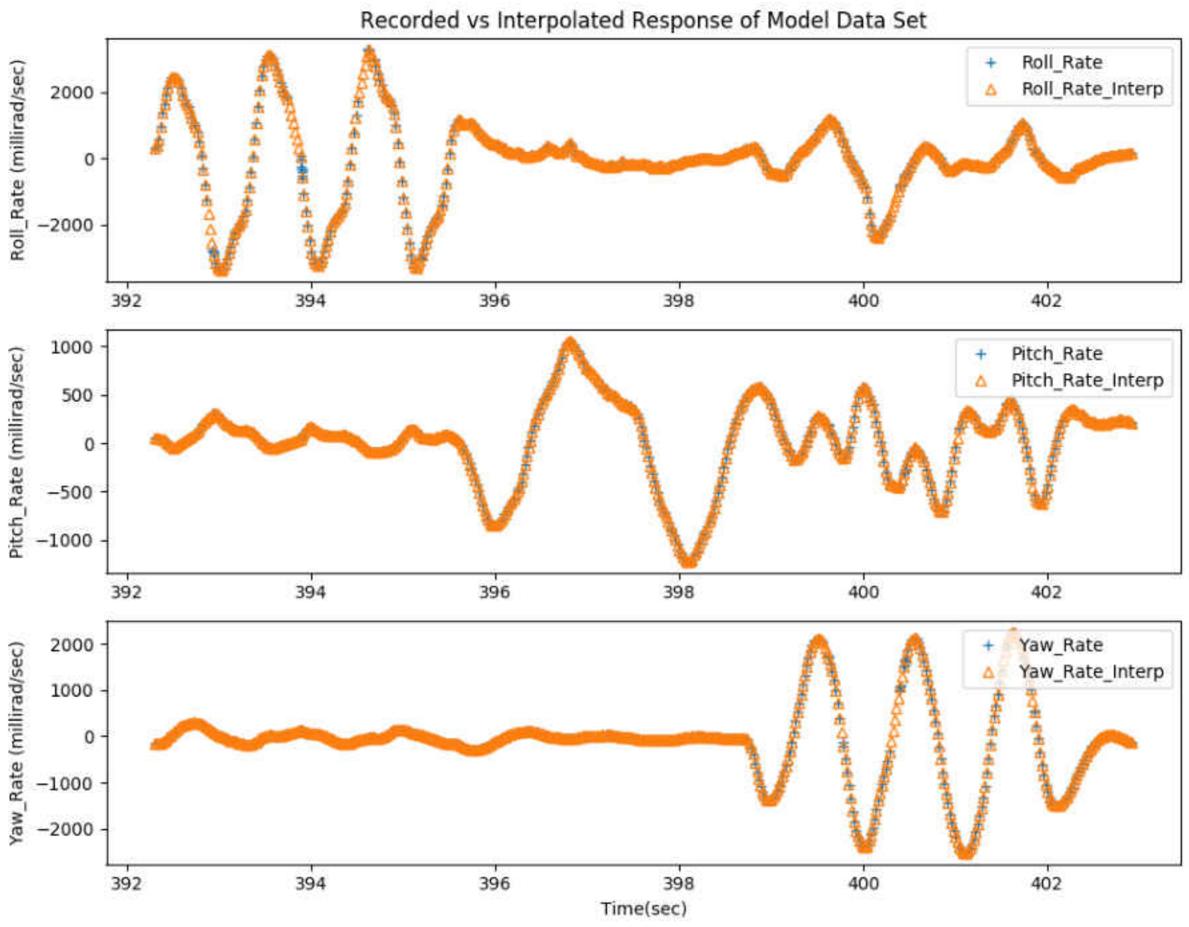


Figure 42-Outputs used to identify ARX model with auto control for run MIMO_6_13_2020__16_18_37

Table 18 shows the identified ARX transfer function model while under automatic control based on the input and output data shown in Figure 41 and Figure 42. Just as in the manual control mode, the model is based on MIMO. Therefore, any coupling between the axis is captured in the model. Additionally, the denominator is the same for each column as it relates to the output data curve. Simultaneously, the numerators are all different because they relate each input's effect on the desired output. The summation of each column provides the complete model for each axis.

Inputs	Roll Rate Output	Pitch Rate Output	Yaw Rate Output
Aileron (PWM)	$\frac{2.412z - 1.431}{z^4 - 2.131z^3 + 1.505z^2 - 0.3361z}$	$\frac{-0.01323z - 0.0146}{z^4 - 1.771z^3 + 0.7341z^2 + 0.0709z}$	$\frac{-0.06946z + 0.0861}{z^4 - 1.926z^3 + 0.9247z^2 + 0.01303z}$
Elevator (PWM)	$\frac{0.179z - 0.2505}{z^4 - 2.131z^3 + 1.505z^2 - 0.3361z}$	$\frac{-0.9721z + 0.6156}{z^4 - 1.771z^3 + 0.7341z^2 + 0.0709z}$	$\frac{-0.1153z + 0.1184}{z^4 - 1.926z^3 + 0.9247z^2 + 0.01303z}$
Rudder (PWM)	$\frac{0.3203z - 0.4617}{z^4 - 2.131z^3 + 1.505z^2 - 0.3361z}$	$\frac{0.1452z - 0.135}{z^4 - 1.771z^3 + 0.7341z^2 + 0.0709z}$	$\frac{-0.4781z + 0.5526}{z^4 - 1.926z^3 + 0.9247z^2 + 0.01303z}$
Airspeed (m/s)	$\frac{-34.81z + 34.15}{z^4 - 2.131z^3 + 1.505z^2 - 0.3361z}$	$\frac{9.365z - 7.845}{z^4 - 1.771z^3 + 0.7341z^2 + 0.0709z}$	$\frac{-12.14z + 12.77}{z^4 - 1.926z^3 + 0.9247z^2 + 0.01303z}$

Table 18- Identified ARX transfer function model for roll, pitch, and yaw rates for auto control

Figure 43 shows the identified ARX model plotted over the measured angular velocity data used to identify the model. The blue dots represent the measured angular velocity from the Cube Orange flight controller. In contrast, the orange plus symbol represents the estimated angular velocity based on the input data used to build the model. The estimated and measured angular velocity appear to correlate well based on an informal visual inspection. However, in the pitch rate plot after 399 seconds, there is a mismatch in the model. Once the pitch excitation is completed, the EdgeTRA is no longer at the desired altitude of 75 meters set via the Fly_Plane.py script. Therefore, the Cube Orange attempts to reacquire the desired altitude by driving the elevator with small inputs. However, based on the measured response, these small inputs do not correlate linearly to the output. As the ARX modeling structure is for linear modeling, the fit is not expected to be good in this period.

Additionally, in the roll rate plot, after 399 seconds, there is a rolling motion. This rolling motion is partially due to coupling in the lateral axis between the rudder and aileron. However, while the rudder excitation is performed, the elevator maintains the desired altitude of 75 meters. When the elevator and rudder are moved simultaneously, this creates a force that rolls the EdgeTRA [50]. However, the Cube Orange is in the loop and counteracts the rolling motion created by the rudder and elevator. Therefore, the motion found in the roll rate plot after 399 seconds is attributed to the Cube Orange reacting to the rolling motion produced by the elevator and rudder movement at the same time.

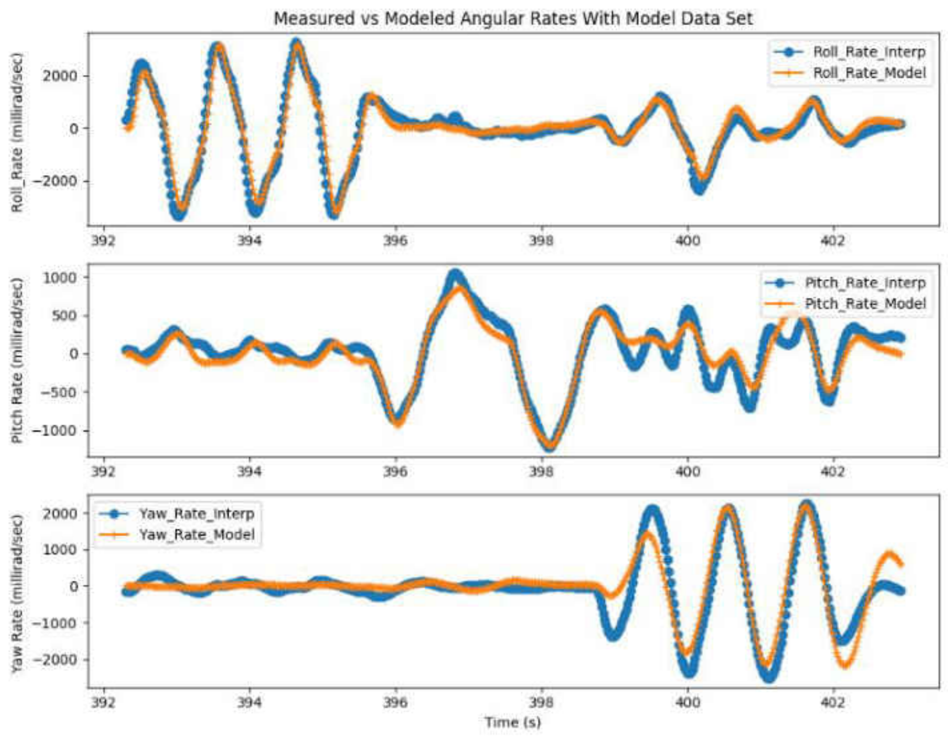


Figure 43-Fitted output using input from the same data used to build the ARX model for run MIMO_6_13_2020__16_18_37

Figure 44 shows the input used to validate the previously identified ARX model for automatic control. Figure 45 shows the blue dots' measured responses and the modeled responses shown by the orange plus signs based on this validation input. As discrepancies were seen in the fit of the measured and estimated angular velocities for the model building run, this validation run shows similar discrepancies. In the pitch rate subplot, after 456 seconds, the pitch excitations have been completed. However, there is still a nonlinear change in pitch rate relative to the applied input. Also, there is still a rolling motion seen in the roll rate subplot after 454 seconds due to the rudder and elevator's simultaneous actuation. However, with these discrepancies, the TIC fit coefficients were not drastically affected as they are still similar to TIC values from the manual control method. For this validation run, the TIC values are 0.184, 0.198, 0.214 for roll, pitch, and yaw, respectively.

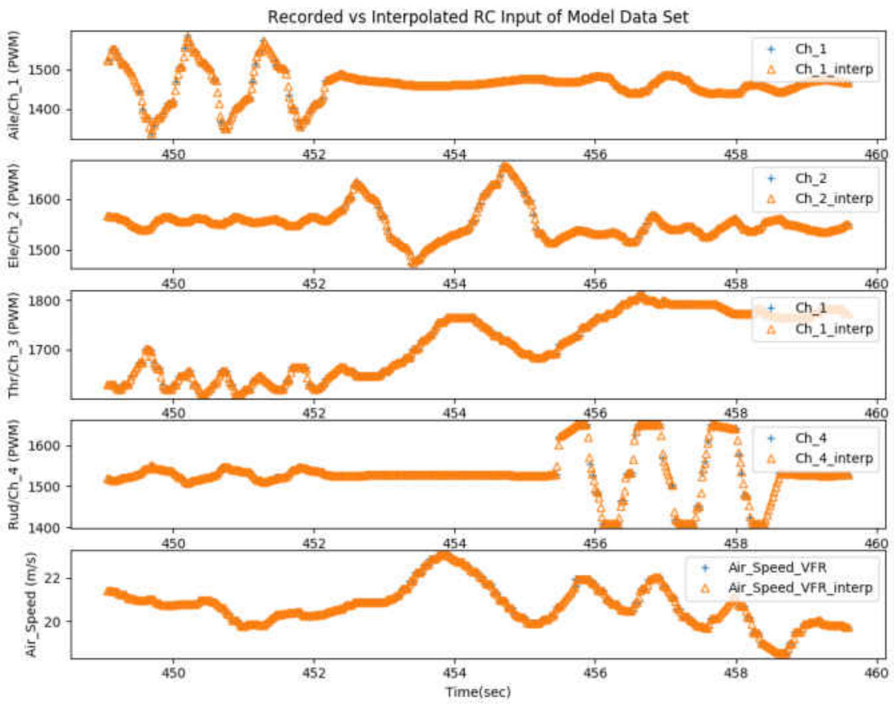


Figure 44-Inputs used to validate ARX model for run MIMO_6_13_2020__16_18_37

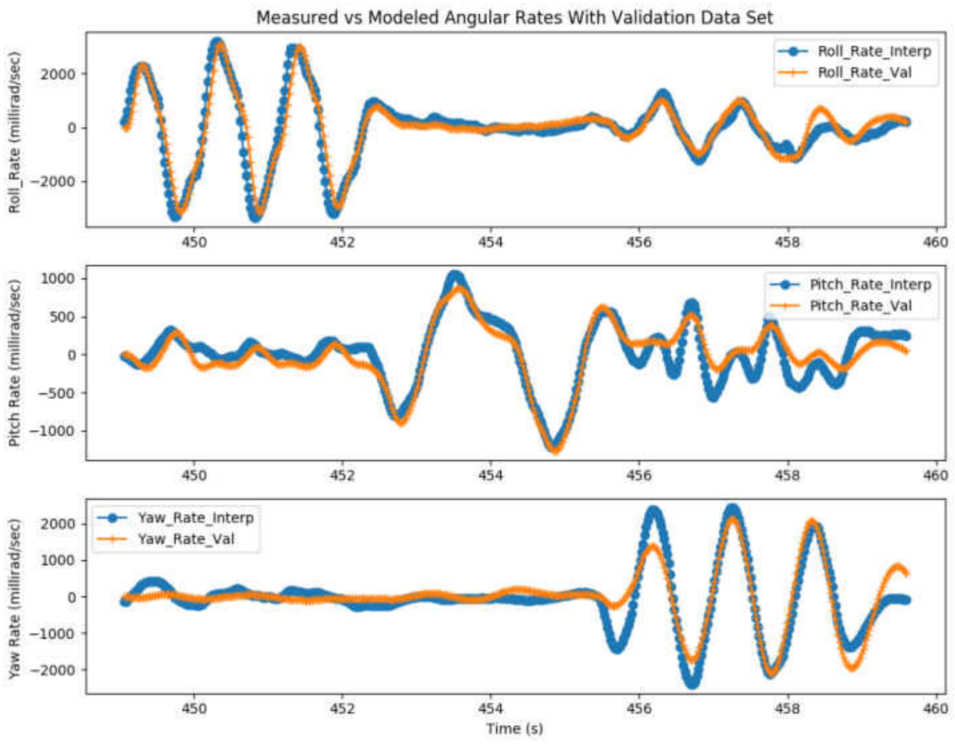


Figure 45-Fitted output using validation for run MIMO_6_13_2020__16_18_37

In total, 13 nominal runs were collected using the automatic control method. Figures 46 to 48 show the TIC values for the 13 runs, except for run 4, which is omitted because the yaw excitation was incomplete before the end of the run. For roll, pitch, and yaw, TIC values appear to have a neutral slope. Compared to the manual control method, using the auto control method with excitations commanded by the Fly_Plane.py script appears to provide more repeatable results, indicated by the TIC standard deviation values for automatic control being less than manual control as shown in Table 19.

	Manual Control TIC Standard Deviation	Automatic Control TIC Standard Deviation
Roll	0.0466	0.0275
Pitch	0.0473	0.0163
Yaw	0.0623	0.0345

Table 19- Comparison of manual vs. automatic standard deviation of TIC values

Also, automatic control runs were performed over varying weather conditions, similar to weather conditions when manual control runs were performed. Automatic control runs 1-8 were performed with ground speed wind conditions of 8-12 mph, while the remaining runs were performed in weather conditions with wind 5 mph or less.

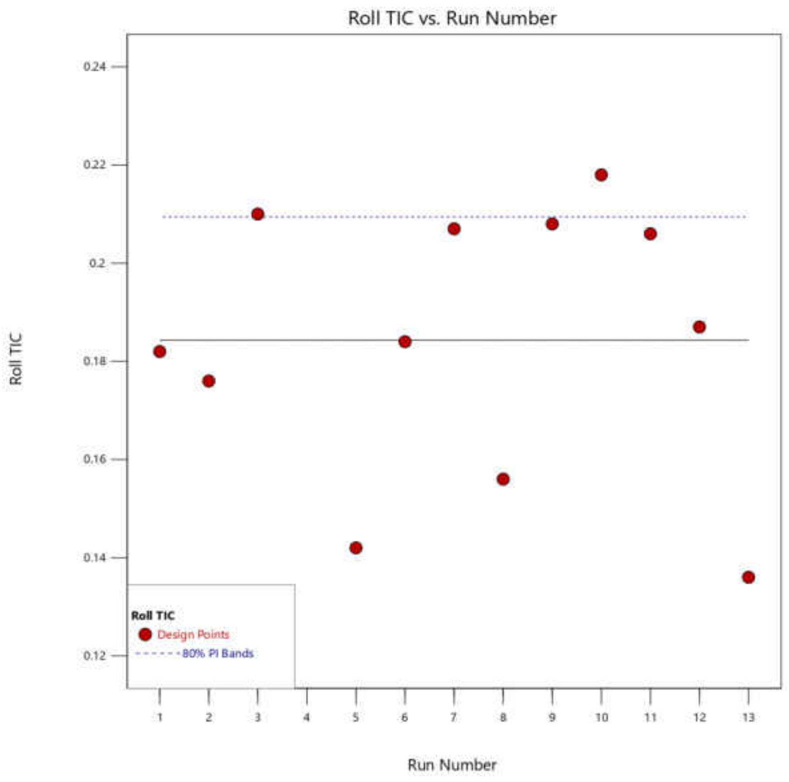


Figure 46-Automatic control roll TIC vs. run number

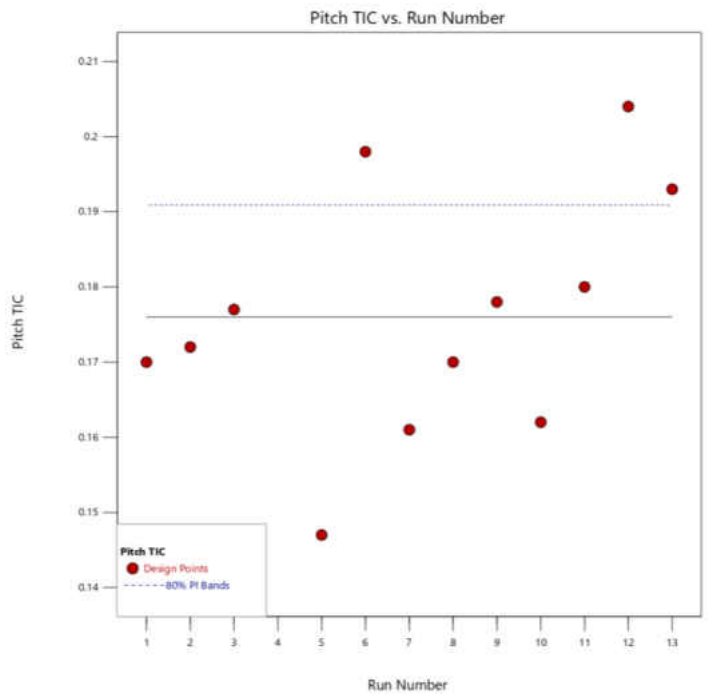


Figure 47-Automatic control pitch TIC vs. run number

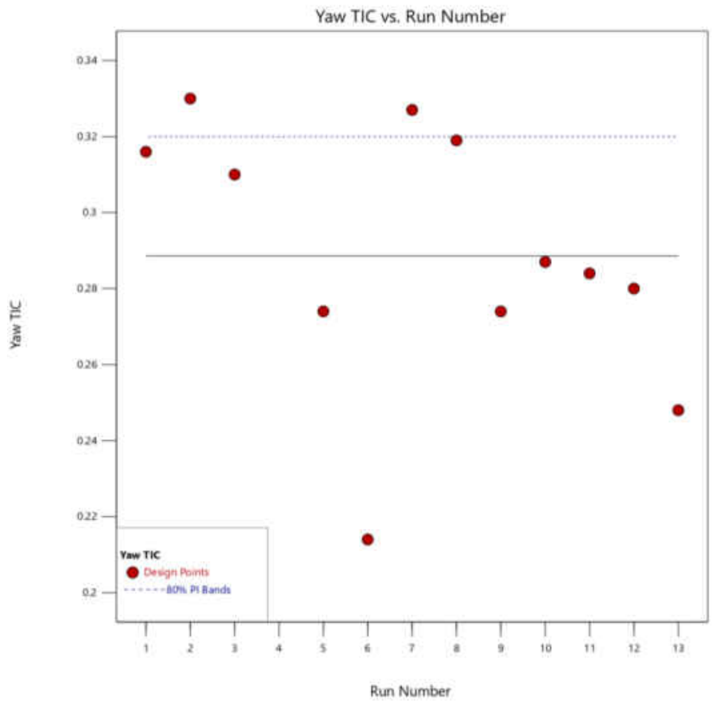


Figure 48-Automatic control yaw TIC vs. run number

8.3 LACK OF CONTROLLABILITY DETECTION

With the baseline model of the angular velocities established, this work is now focused on detecting a lack of controllability. In Table 8, the failure modes were described and demonstrated using the manual control method. Each failure mode was implemented one at a time, and the results were collected. A lack of controllability can be seen visually in Figure 49. In this run, both aileron two and elevator two are stuck in a neutral position. In the roll rate versus time subplot between 502 and 506 seconds, the ARX model indicates that the roll rate should have a greater magnitude than the measured roll rate. A greater pitch rate is also indicated in the pitch rate subplot between 505 and 511seconds.

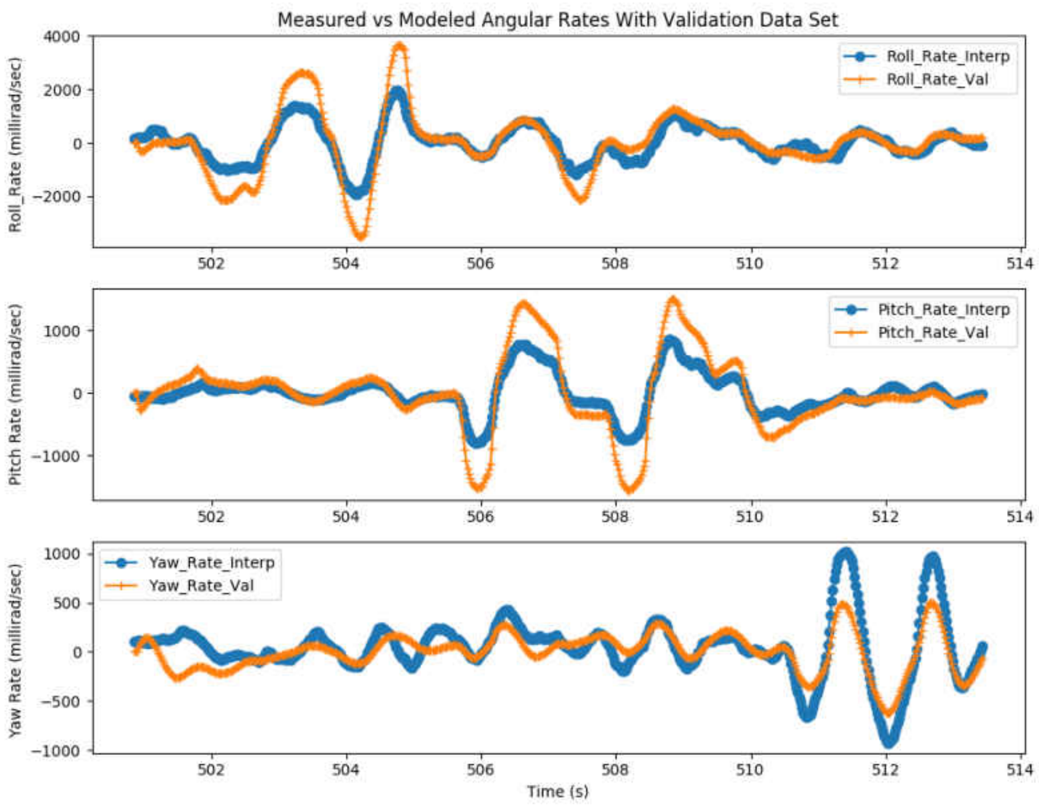


Figure 49-Fitted output after aileron two and elevator two are stuck neutral for run MIMO_4_05_2020__16_55_18

Before the introduced failure, as shown in Figure 49, a nominal run was made with results shown in Figure 50. The roll, pitch, and yaw angular velocity subplots show the ARX model estimate, and the measured angular velocities agreed before the failure was introduced.

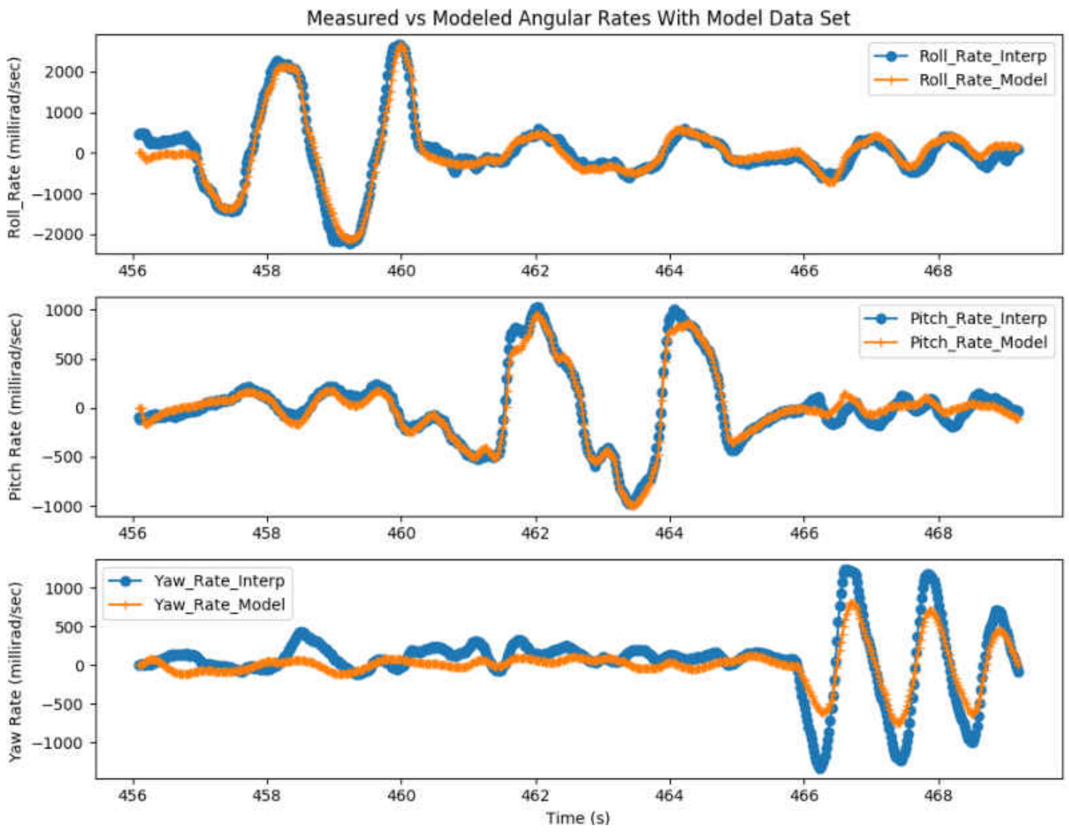


Figure 50-Fitted output under nominal conditions for run MIMO_4_05_2020__16_55_18

Therefore, the discrepancy between the ARX model and the measured angular velocity increases the TIC value and is flagged as a lack of controllability detection on the respective axis. Results are shown for all failure mode's TIC values in detail in the following sections.

8.3.1 AILERON TWO STUCK NEUTRAL

Figure 51 shows the TIC values for all collected data under nominal conditions, as shown by the red circles. In addition, runs shown by black squares with checkmarks indicate when aileron two was stuck in the neutral position. A lack of controllability is detected if the TIC value of a run is greater than the prediction interval, as shown by the light blue dashed line. The prediction interval is based on the mean and standard deviation of the TIC values for the 29 nominal runs and the alpha choice. Alpha equal to 20% is selected, as this allows for greater sensitivity for lack of controllability detection. When aileron two is stuck in the neutral position, the TIC value for these runs is clearly above the nominal range established by the prediction interval, indicating a lack of controllability in the roll axis.

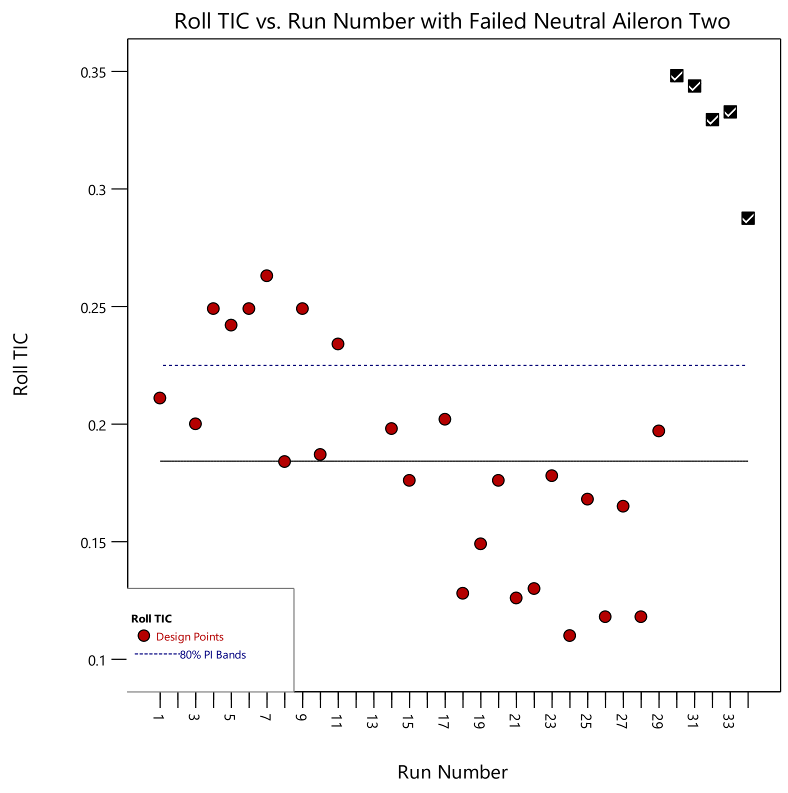


Figure 51-Roll TIC vs. run number showing runs when aileron two is failed neutral

While aileron two was stuck in the neutral position, no failure is implemented on the pitch axis. Therefore, pitch TIC values should be at or below the prediction interval threshold, which is the case for most runs made with this failure, as shown in Figure 52, denoted by the black squares with checkmarks. However, run 33 is found to be above the PI, indicating a false alarm for the pitch axis.

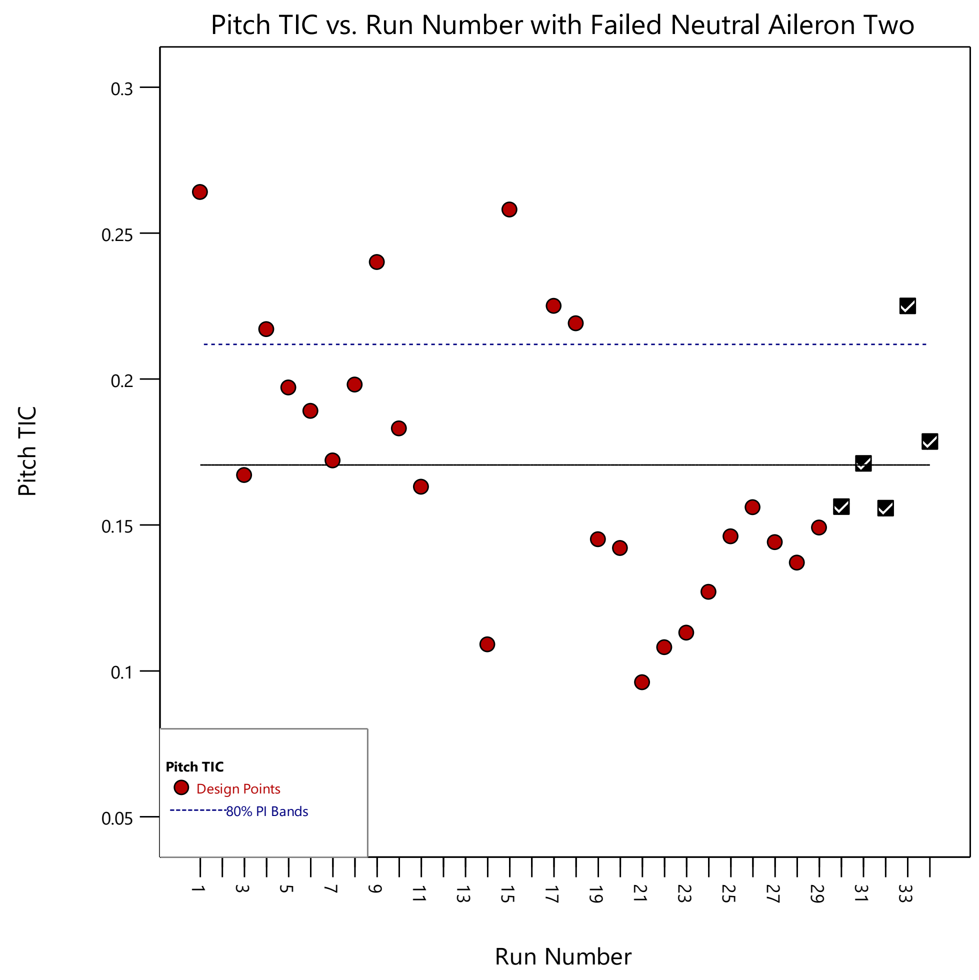


Figure 52-Pitch TIC vs. run number showing runs when aileron two is failed neutral

Similarly, for the yaw axis, while aileron two is stuck in the neutral position, the yaw axis had no failure introduced. Therefore, the yaw TIC values are at or below the prediction interval, as shown in Figure 53 for most failure mode runs, denoted by the black checked squares. However, run 32 is above the threshold in the yaw axis, indicating a false alarm on the yaw axis.

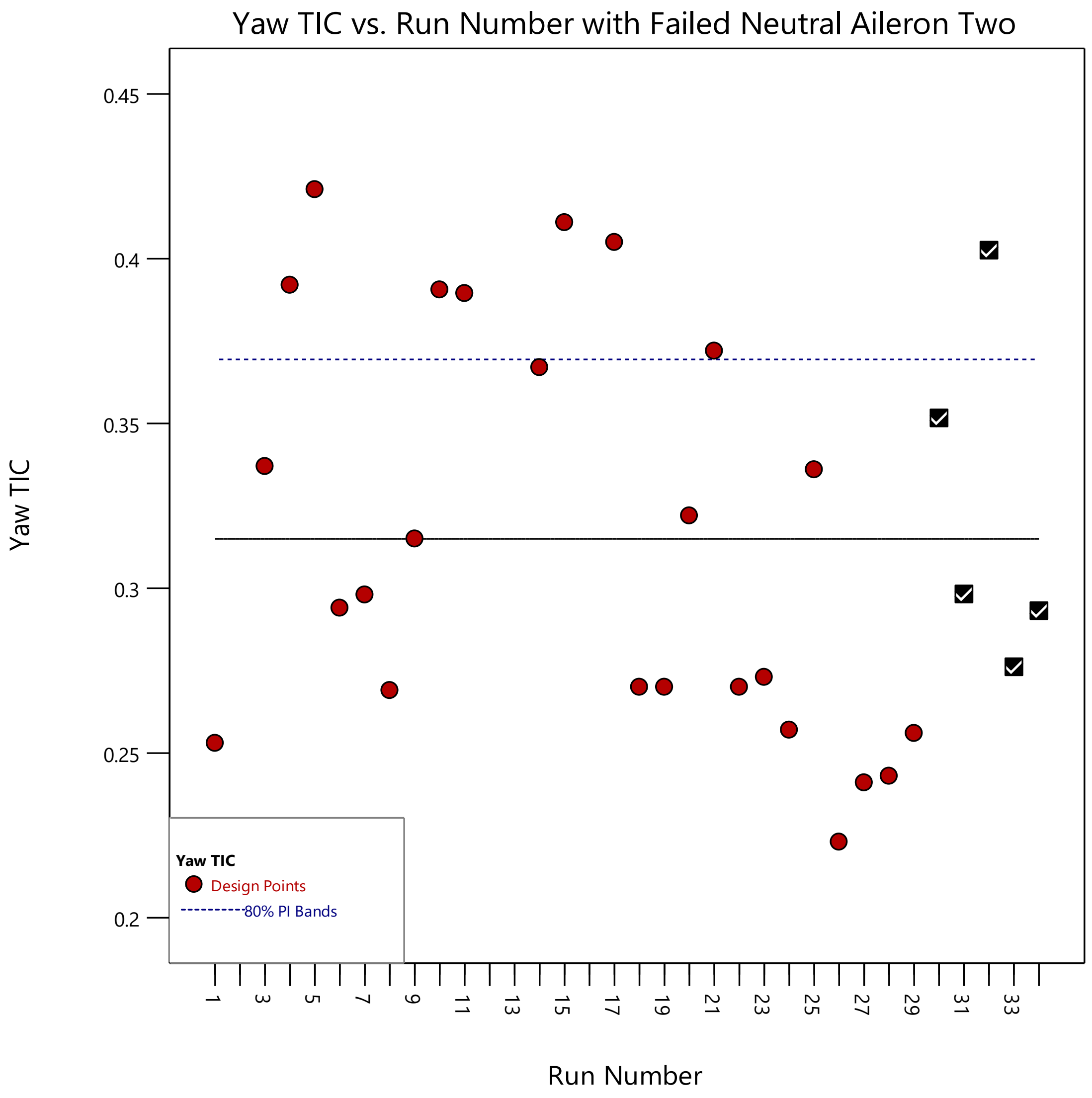


Figure 53-Yaw TIC vs. run number showing runs when aileron two is failed neutral

8.3.2 AILERON TWO WITH LIMITED TRAVEL

Figure 54 shows the lack of controllability detection for the limited throw by ±25% of the aileron two case, shown by the black squares with checkmarks above the prediction interval. Compared to the stuck in neutral position aileron case, the limited aileron case has lower TIC values, which are expected, as the failure is not as drastic. Also, for pitch and yaw, the TIC values do not detect a failure since no failure is introduced on those axes.

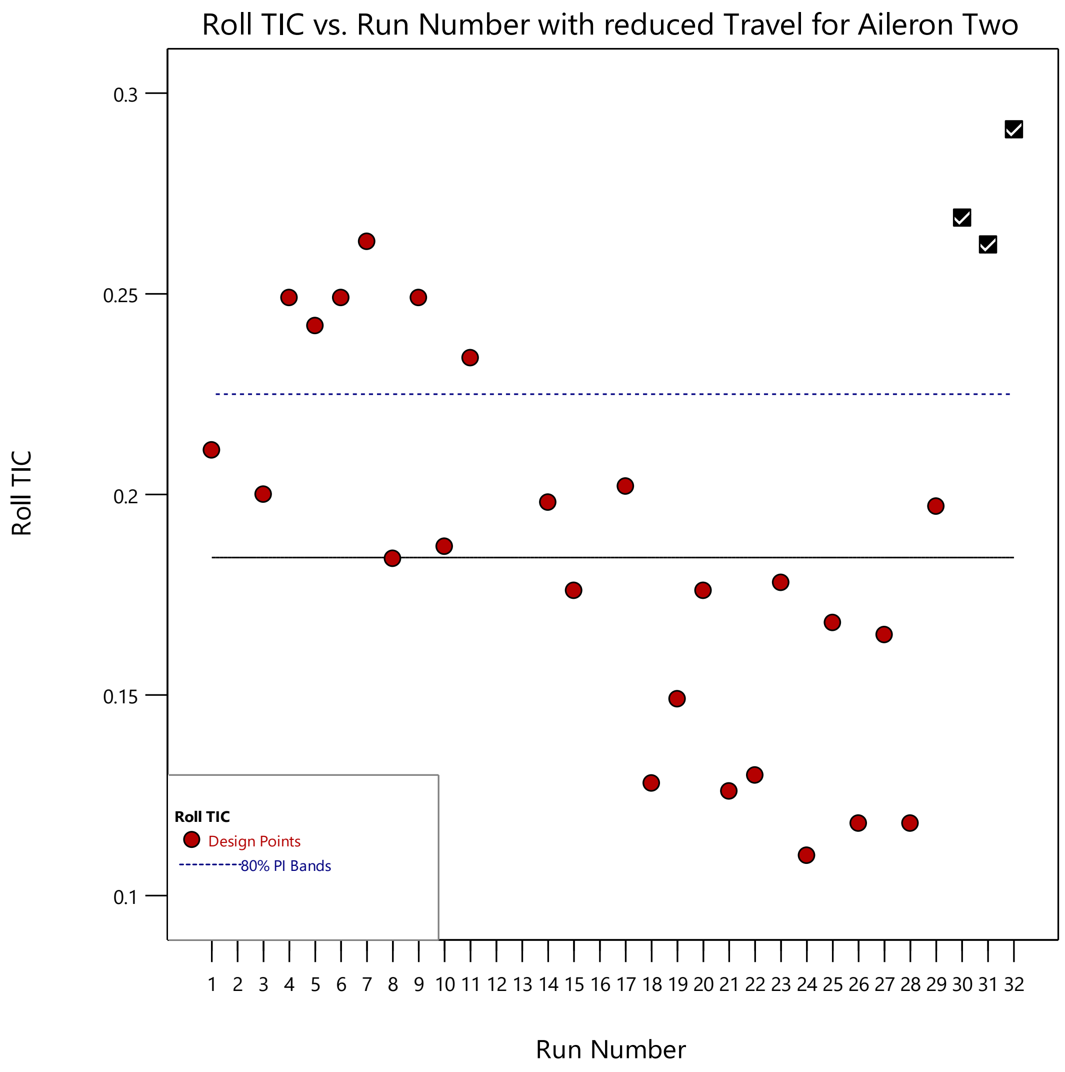


Figure 54-Roll TIC vs. run number with aileron two having limited travel

8.3.3 ELEVATOR TWO STUCK NEUTRAL

When elevator two is stuck in the neutral position, the TIC values are indicated by the black squares with checkmarks, as shown in Figure 55, and are all found to be above the prediction interval. This indicates that the failure mode has been detected. However, some of the initial nominal runs collected have TIC values near the same magnitude as failure mode runs 31 and 33. The TIC values above the PI are believed to be due to gusty weather conditions and insufficient PIC input excitation when initial nominal runs were collected.

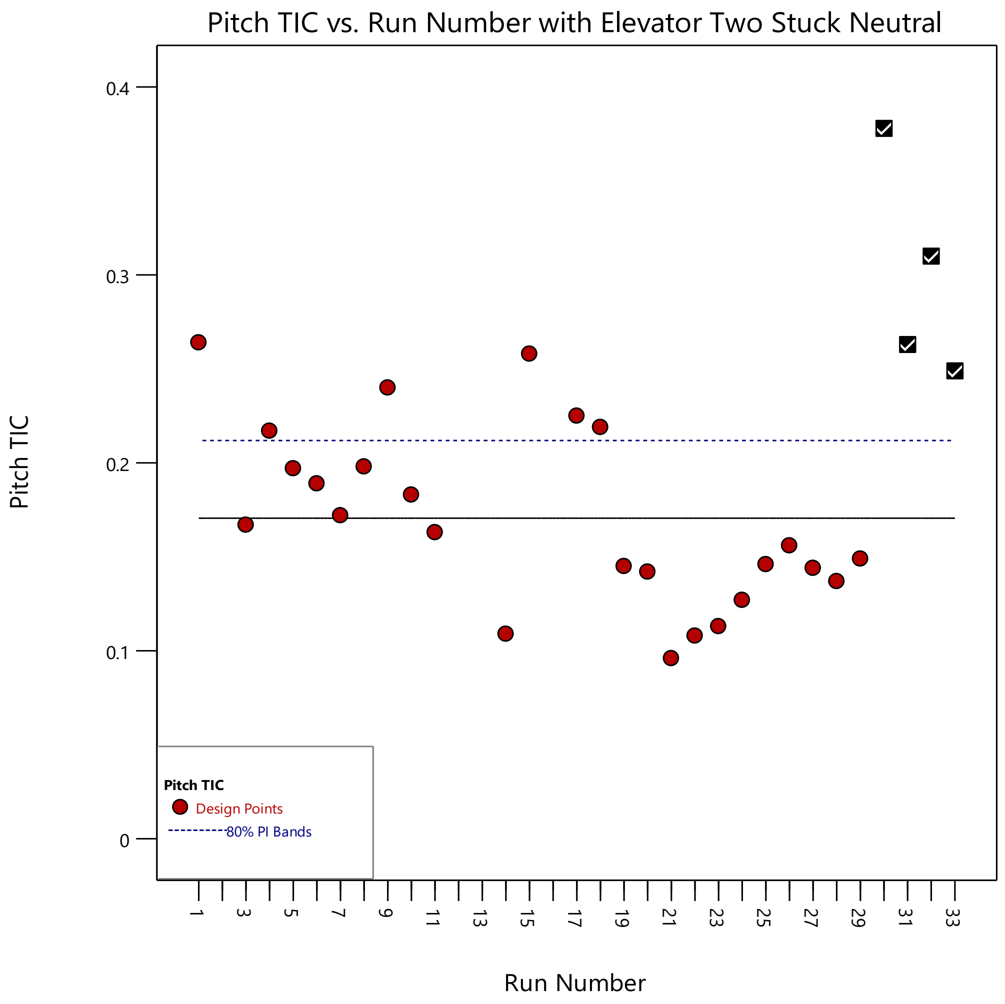


Figure 55-Pitch TIC vs. run number with elevator two stuck neutral

8.3.4 ELEVATOR TWO WITH LIMITED TRAVEL

For the limited travel of the elevator failure mode, the travel was limited to ± 25%. The TIC values for this failure mode are indicated by black squares with checkmarks, as shown in Figure 56. All failure mode tests are above the prediction interval, indicating a detection of a lack of controllability.

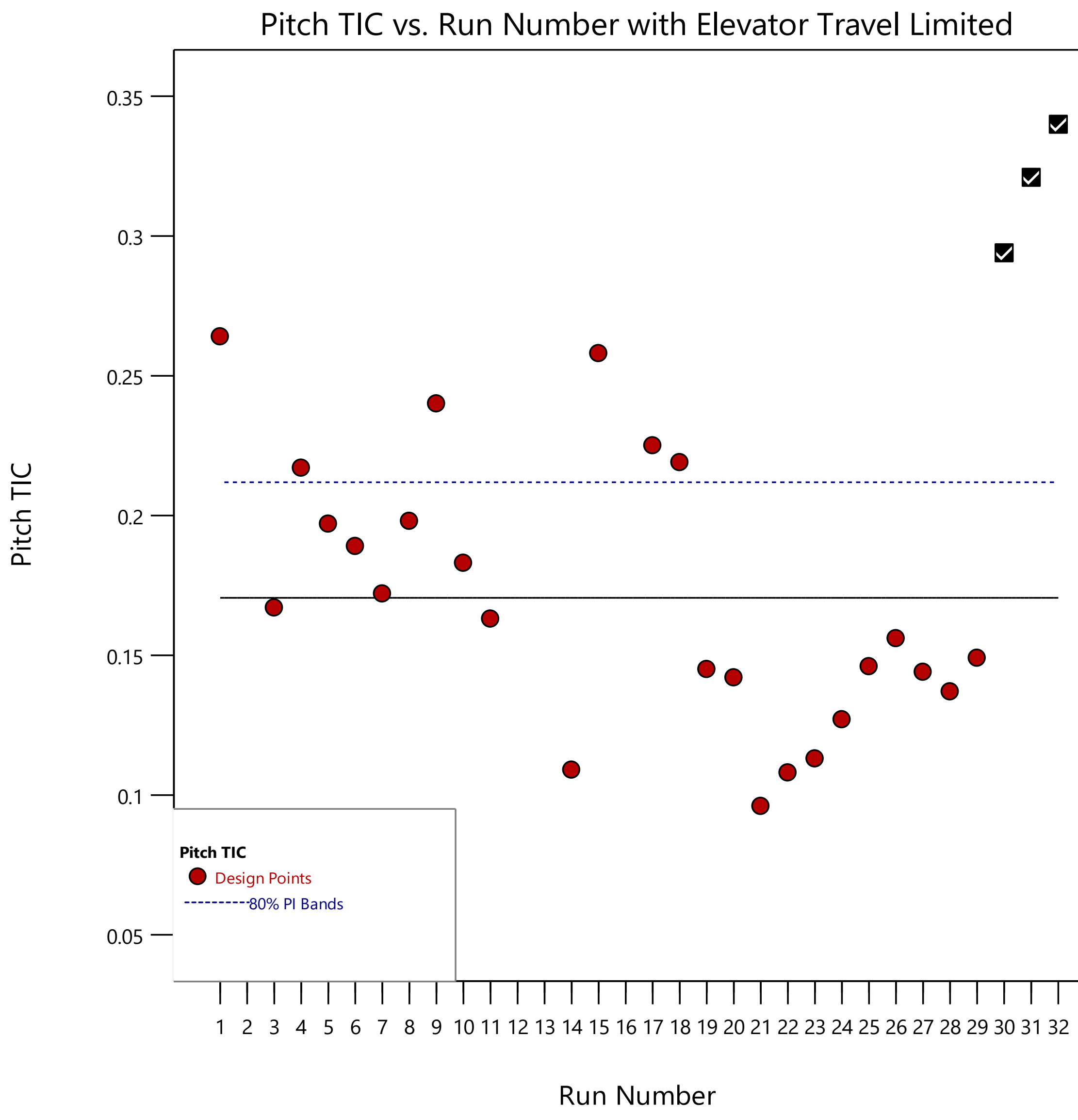


Figure 56-Pitch TIC vs. run number with elevator two travel limited

8.3.5 AILERON TWO STUCK NEUTRAL AND ELEVATOR TWO LIMITED TRAVEL

Combinations of failure modes are also tested. For this failure mode, aileron two is fixed at its neutral point, while elevator two is limited to only $\pm$ 25% of its full travel simultaneously. The TIC value results during the roll axis's failure mode are marked by the black squares with checkmarks, as shown in Figure 57. As the TIC values for the runs to test the failure mode in the roll axis are above the prediction interval, there is a detection of a lack of controllability in the roll axis.

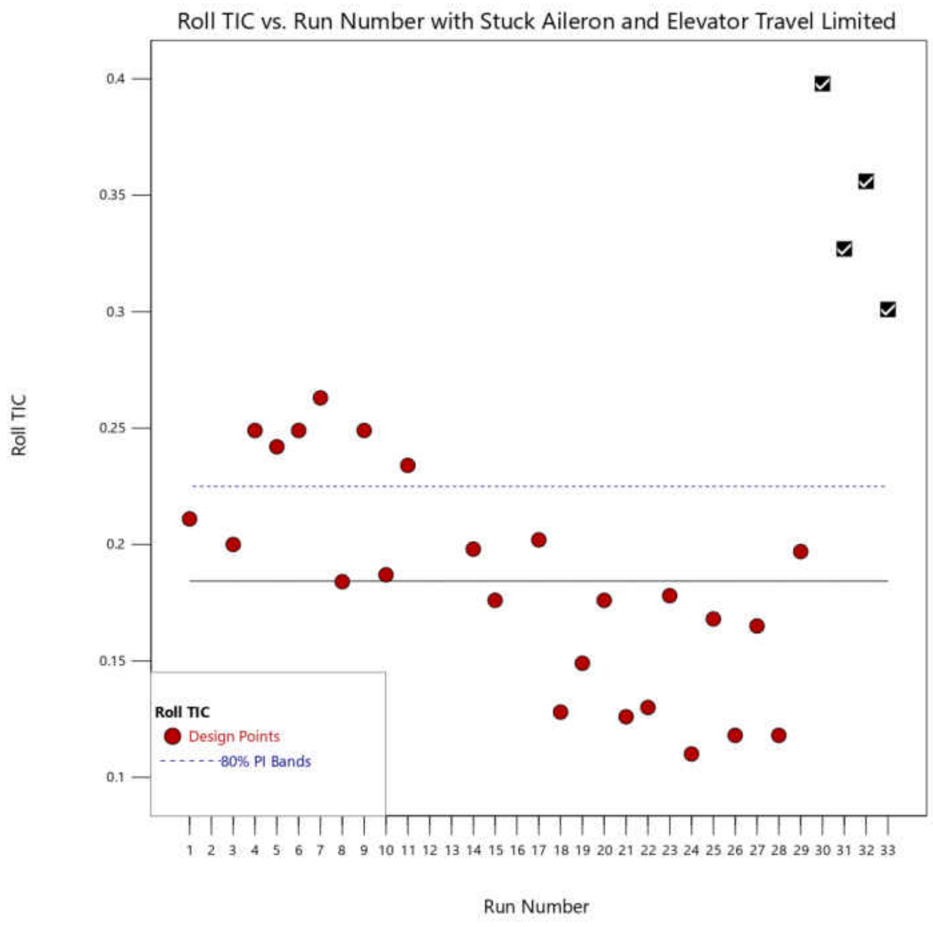


Figure 57-Roll TIC vs. run number with aileron two neutral and elevator two travel limited

Additionally, since this failure mode contains two compromised control surfaces, the pitch axis is also reviewed. As expected, the pitch axis detection of a lack of controllability is found by the black squares with checkmarks above the prediction interval, as shown in Figure 58.

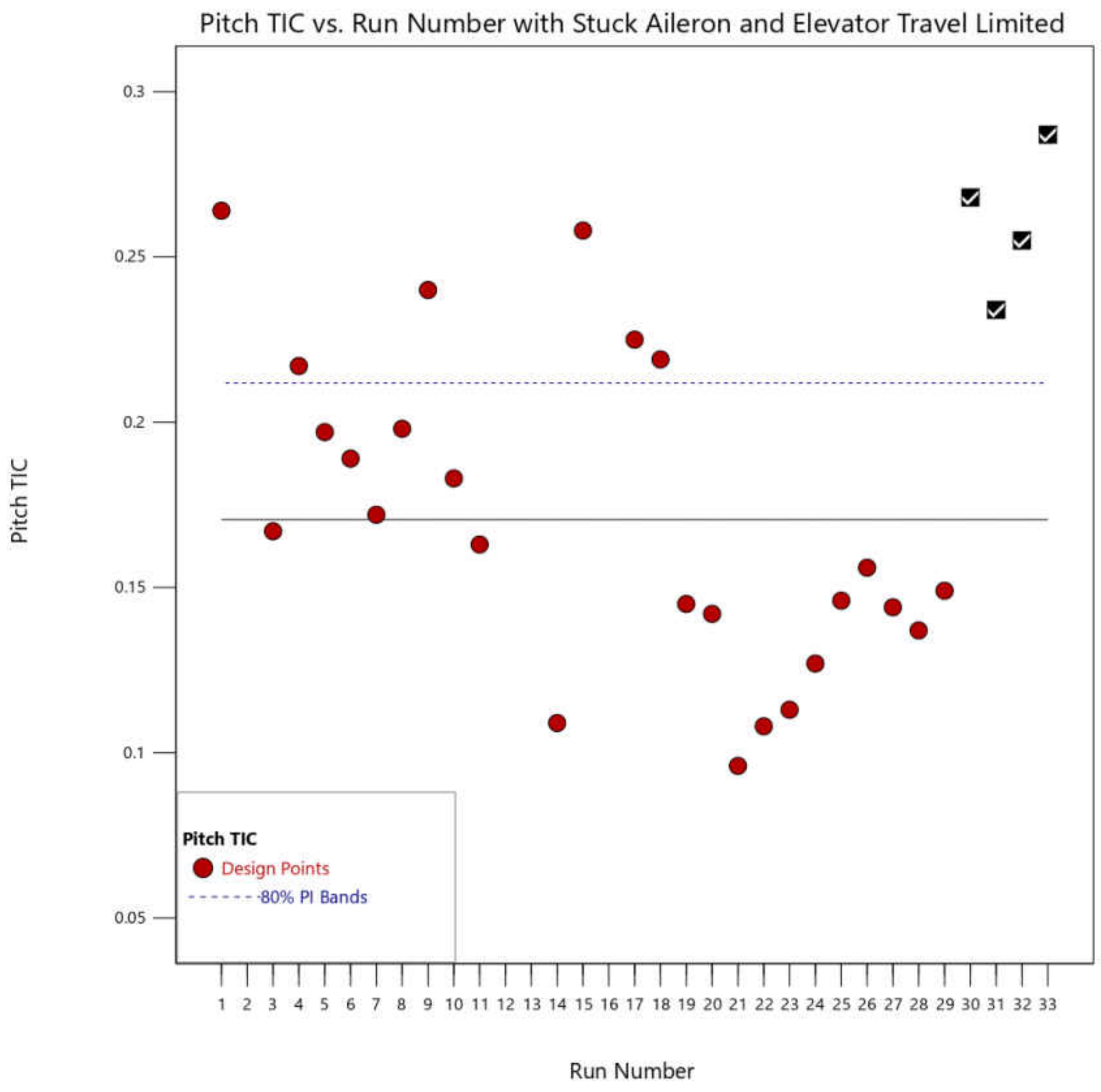


Figure 58-Pitch TIC vs. run number with aileron two neutral and elevator two travel limited

8.3.6 AILERON TWO LIMITED TRAVEL AND ELEVATOR TWO STUCK NEUTRAL

The failure mode combination of aileron two with its travel limited to ± 25% of its original travel, and elevator two fixed to its neutral position, results are shown in Figure 59 for the roll axis. As failure mode runs with the black squares with checkmarks are above the prediction interval, there is a detection of a lack of controllability.

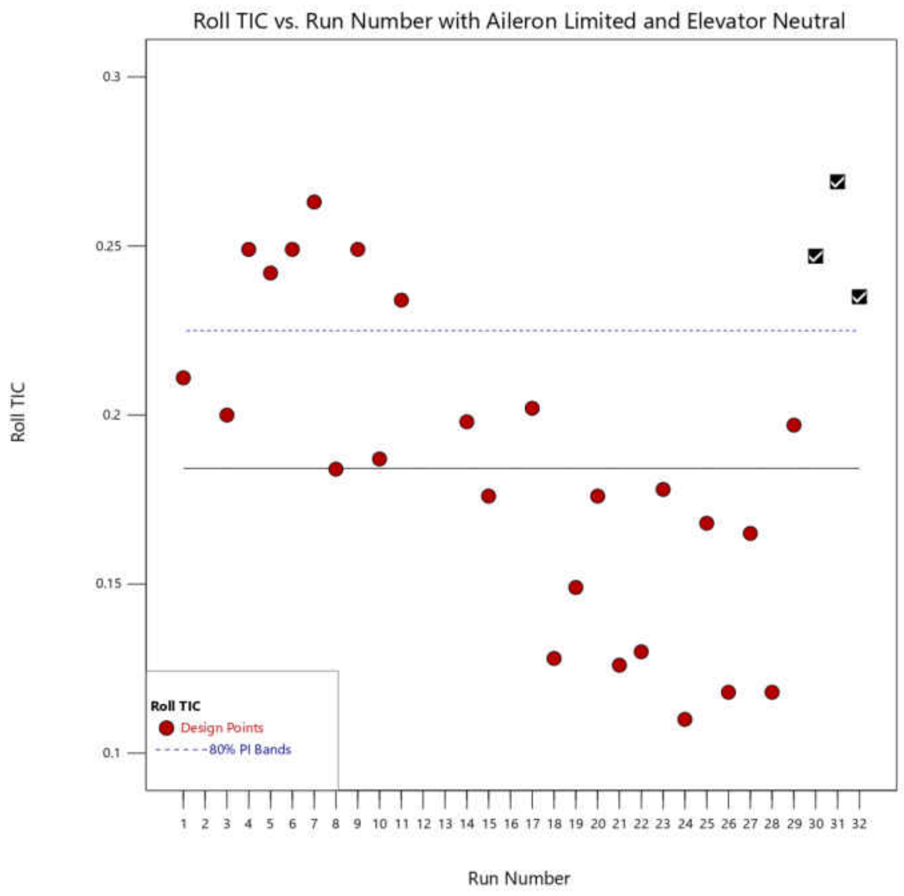


Figure 59-Roll TIC vs. run number with aileron two limited and elevator two neutral

The same is also found for the pitch axis, indicated by the black squares' TIC values with checkmarks above the prediction interval, as shown in Figure 60. In comparison to failure modes where a surface is fixed to its neutral position or is limited in travel, these results show how more drastic failure modes affect the chance of detection. Such as with the failed neutral elevator, the TIC values have a larger magnitude than the limited elevator case.

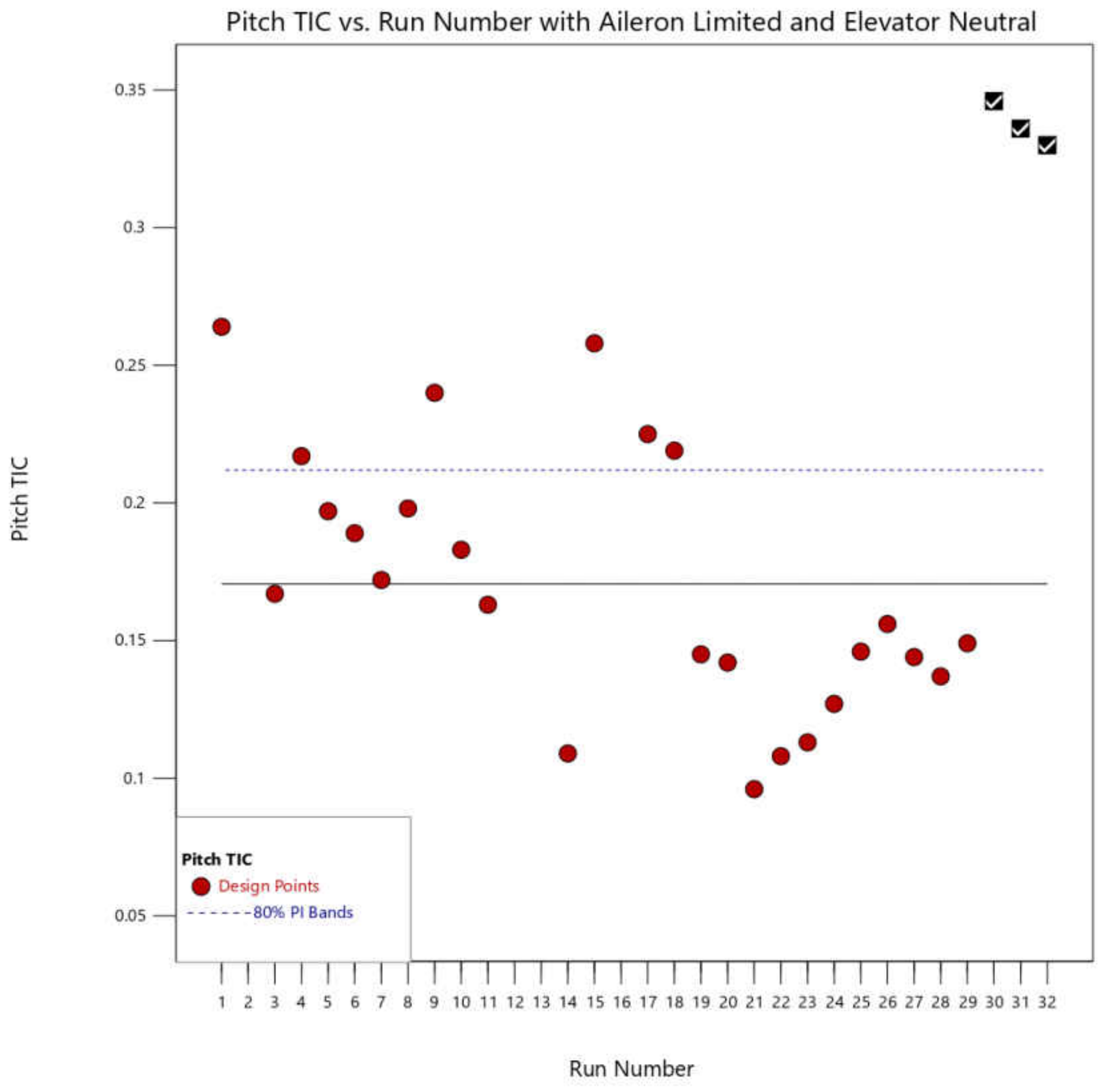


Figure 60-Pitch TIC vs. run number with aileron two limited and elevator two neutral

8.3.7 AILERON TWO AND ELEVATOR TWO STUCK NEUTRAL

When aileron two and elevator two are fixed in their neutral position, a detection of a lack of controllability is definitively found for the roll axis. As shown in Figure 61, the black squares with checkmarks are for runs 30 to 32.

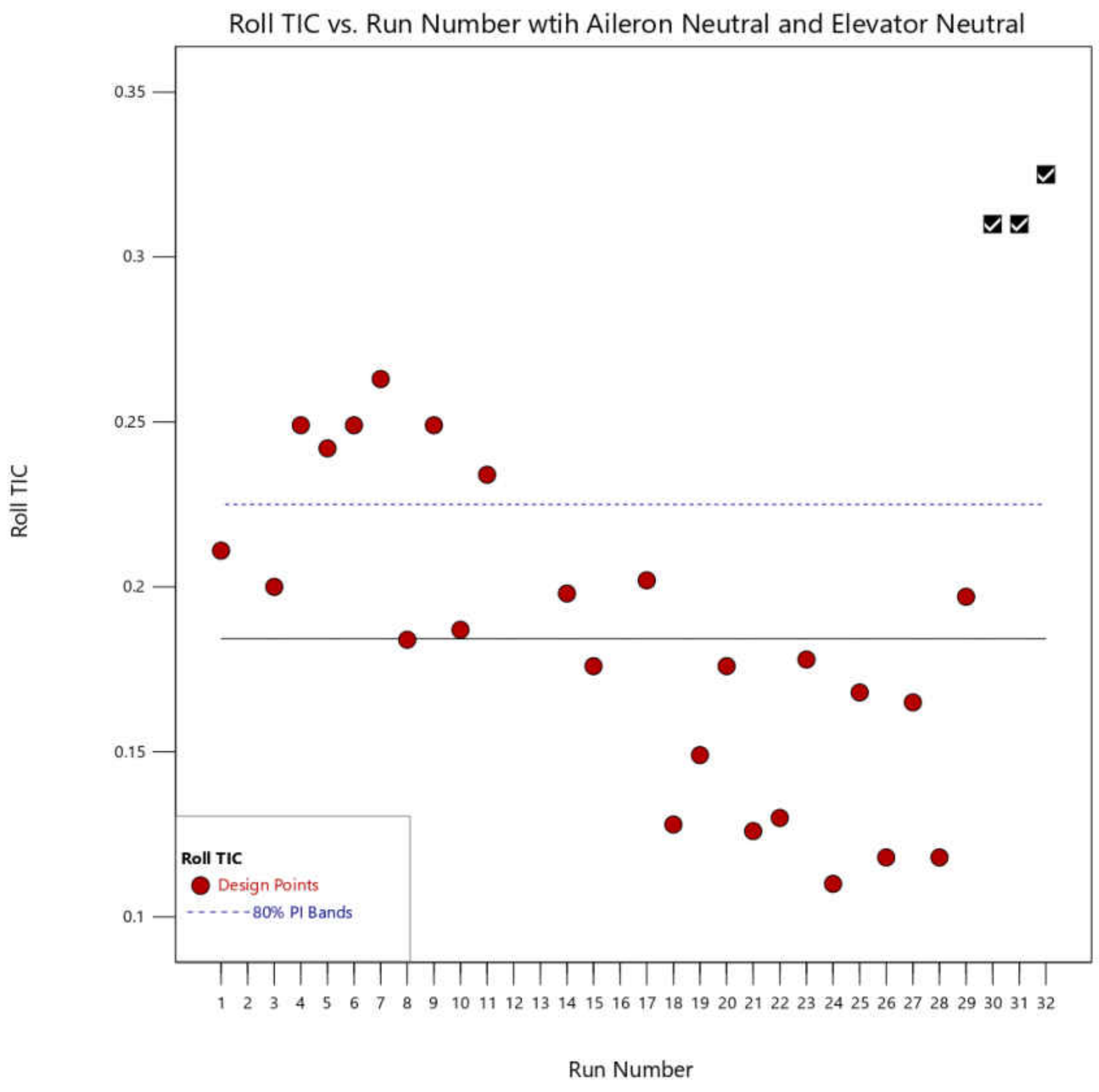


Figure 61-Roll TIC vs. run number with aileron two and elevator two stuck neutral

For the pitch axis, detection of a lack of controllability is also found due to the fixed aileron and elevator failure mode combination, indicated by the black squares with checkmarks above the prediction interval, as shown in Figure 62.

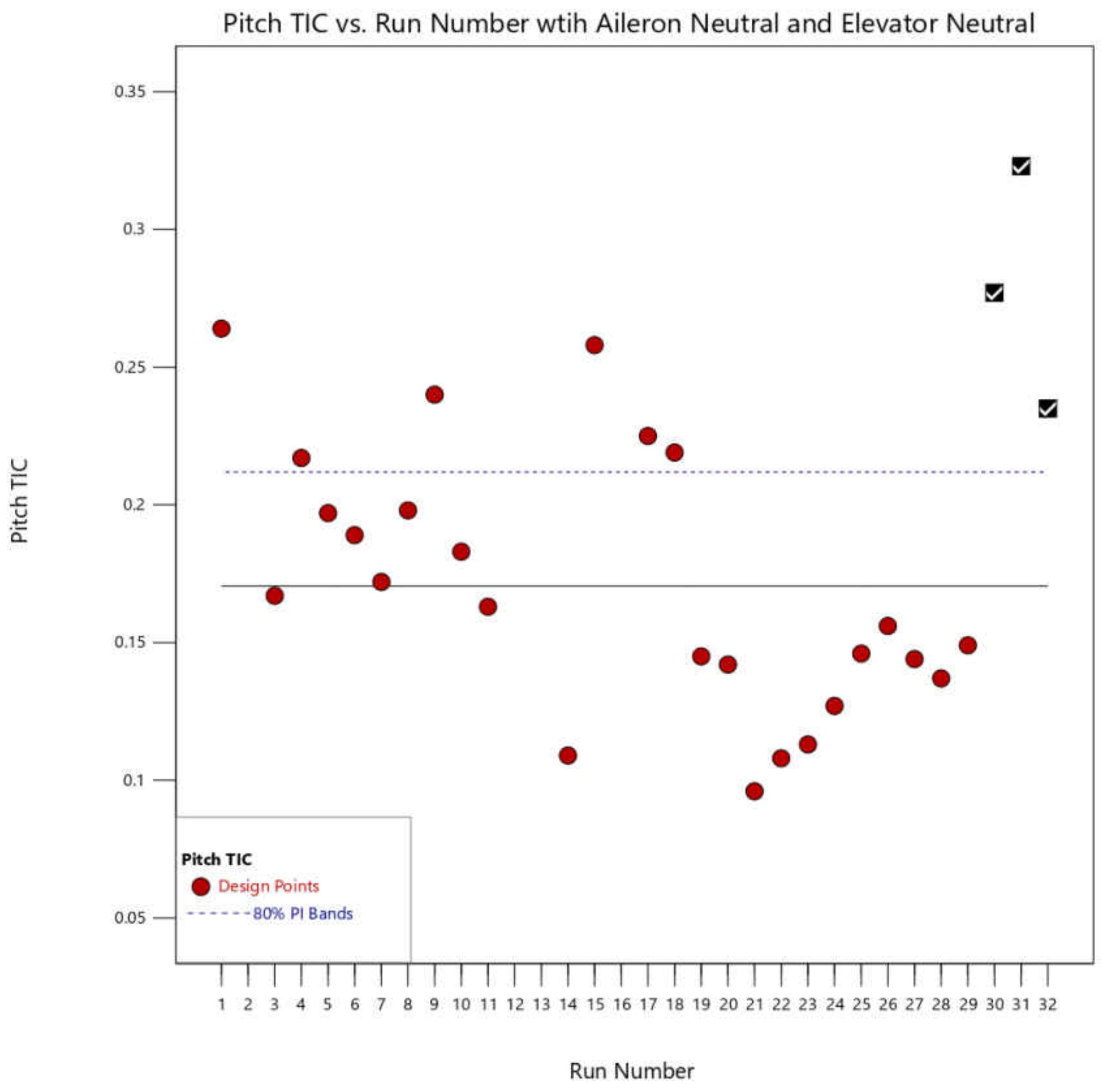


Figure 62-Pitch TIC vs. run number with aileron two and elevator two stuck neutral

8.3.8 AILERON TWO AND ELEVATOR TWO WITH LIMITED TRAVEL

Results from both aileron two and elevator two having limited travel of ± 25% of their original travel show a lack of controllability for the roll axis, which is denoted by the black squares with checkmarks, as shown in Figure 63. However, for the roll axis, approximately only 50% of the runs made with this combination failure mode fall above the prediction interval. This partial detection of a lack of controllability is believed to be due to elevator two having limited travel while elevator one has full travel. This mismatch in the travel between elevators one and two aids in rolling the EdgeTRA. Therefore, even with aileron two being compromised, the roll axis angular velocity is closer to its nominal rate due to the travel mismatch between elevator one and two, which drives down the roll axis TIC value, indicating a better fit.

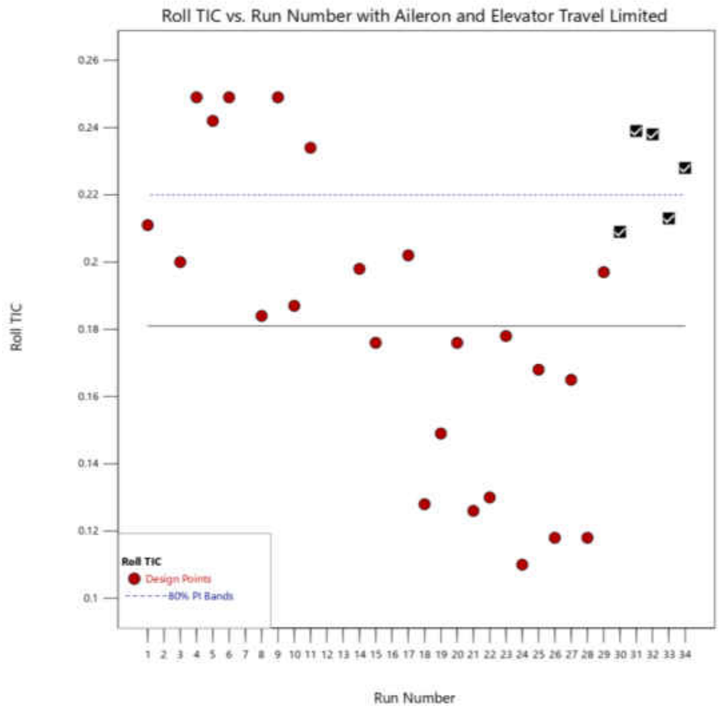


Figure 63-Roll TIC vs. run number with aileron and elevator travel limited

Additionally, for the pitch axis, this combination failure mode is showing detection of a lack of controllability indicated by the black squares with checkmarks, as shown in Figure 64. Compared to the single failure mode of just elevator two having its travel limited to $\pm$ 25% of its original travel, the TIC values of the runs made with this combination failure mode appear not to be affected for the pitch axis, which is unlike the roll axis.

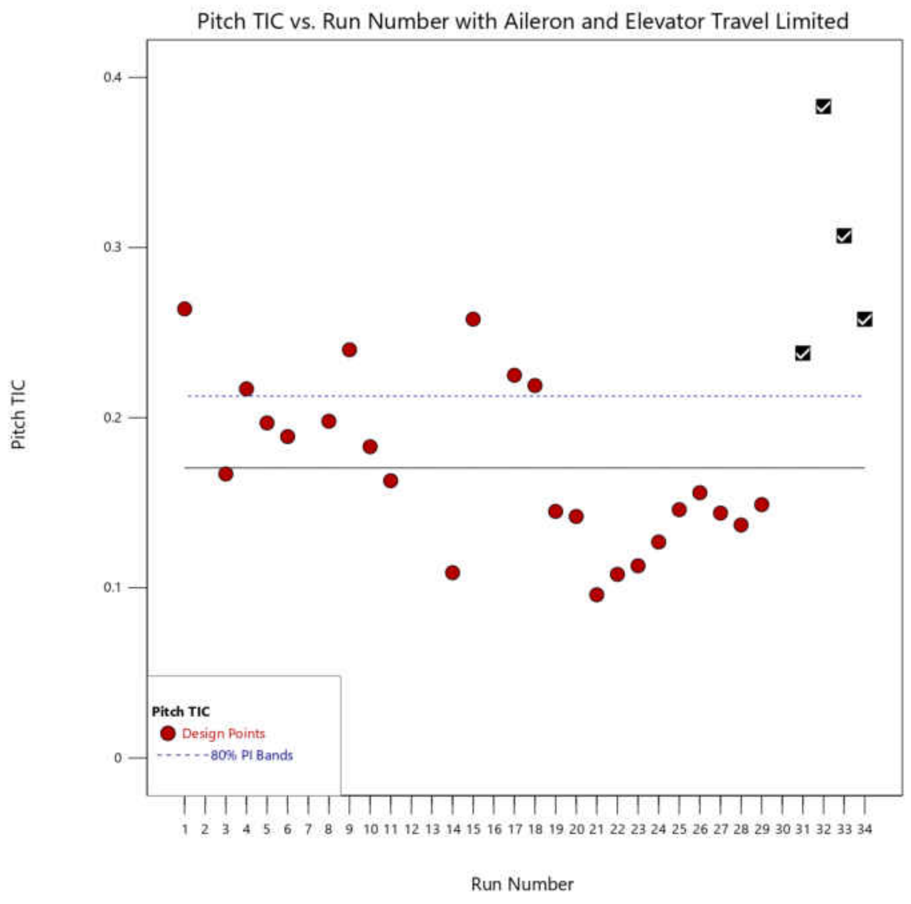


Figure 64-Pitch TIC vs. run number with aileron and elevator travel limited

8.3.9 RUDDER LIMITED TRAVEL

For the rudder travel limited to $\pm$ 25% case, a lack of controllability is found, which is indicated by the black squares with checkmarks, as shown in Figure 65. The yaw axis angular velocity was the most challenging axis to model due to its moment of inertia, explained in detail in Chapter 9. The yaw axis had a large variance in the TIC values for nominal runs, which means the chance for false positives for the yaw axis is high. However, the rudder-limited failure mode runs still show TIC values above the prediction interval, indicating a problem in the yaw axis.

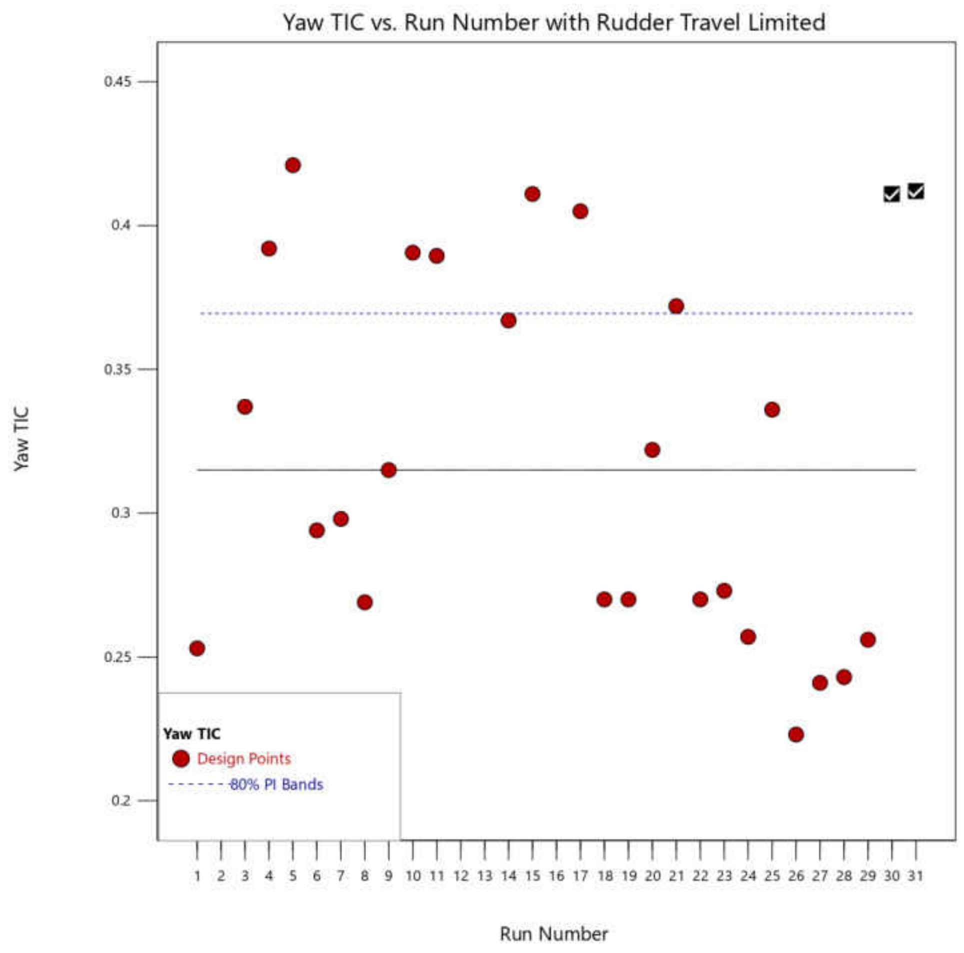


Figure 65-Yaw TIC vs. run number with rudder travel limited

8.3.10 RUDDER LIMITED TRAVEL AND ELEVATOR TWO LIMITED TRAVEL

A combination of both the rudder and elevator two limited to ± 25% of their original throw are tested simultaneously. For the pitch axis, a lack of controllability is found by the black squares with checkmarks above the prediction interval, as shown in Figure 66.

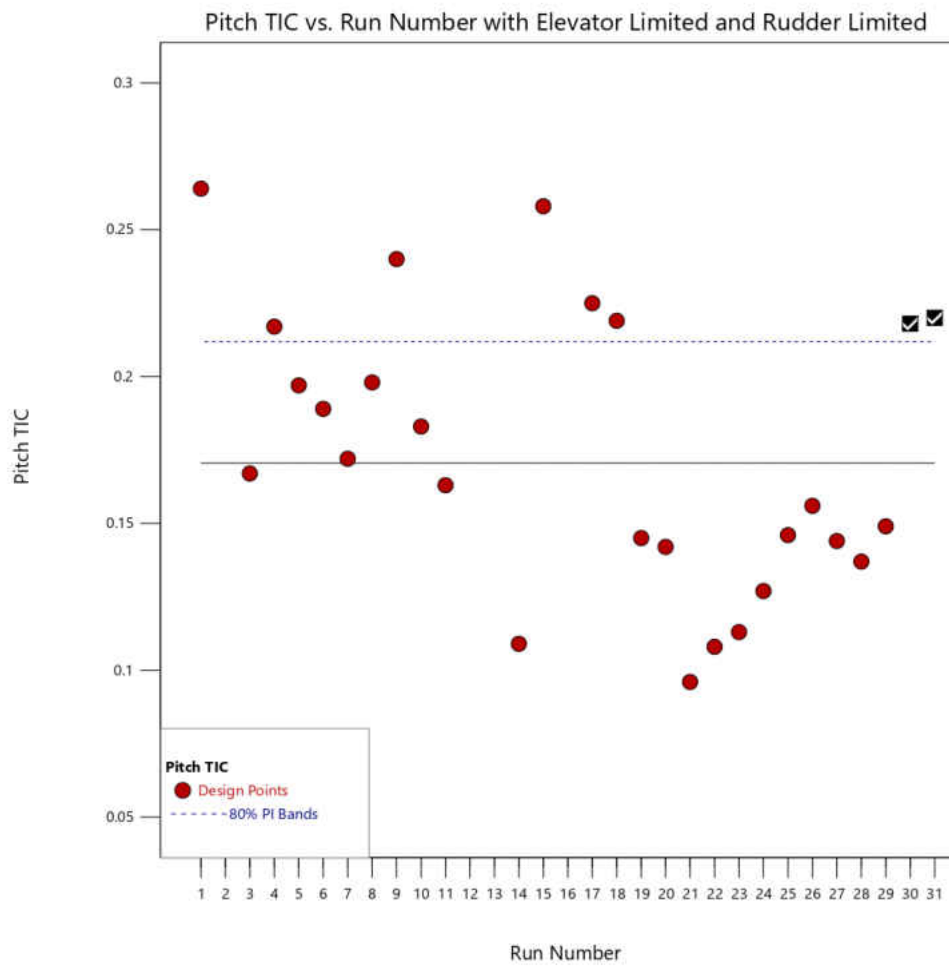


Figure 66-Pitch TIC vs. run number with elevator two and rudder two travel limited

For the yaw axis, the detection of a lack of controllability is found, indicated by the black squares with checkmarks above the prediction interval, as shown in Figure 67. Compared to the single failure mode of just the rudder limited to ± 25%, results from this combination failure mode runs show TIC values of greater magnitude. This is believed to be due to angular velocity models being built for each run, and the yaw axis has a significant variance. Since models are built for each run, this changes the prediction effectiveness from run to run. Also, as the variance is large for the yaw axis's nominal runs, sometimes a model from one run predicts better than other runs. Therefore, when the runs are made with the combination failure mode of limited travel for both the rudder and elevator two, it is found that these models had higher TIC values than when a nominal input was applied. When the failure mode is implemented, the yaw axis TIC values only increase, which explains the difference in the yaw axis TIC values for just the rudder limited and the combination failure mode of the rudder and elevator two limited.

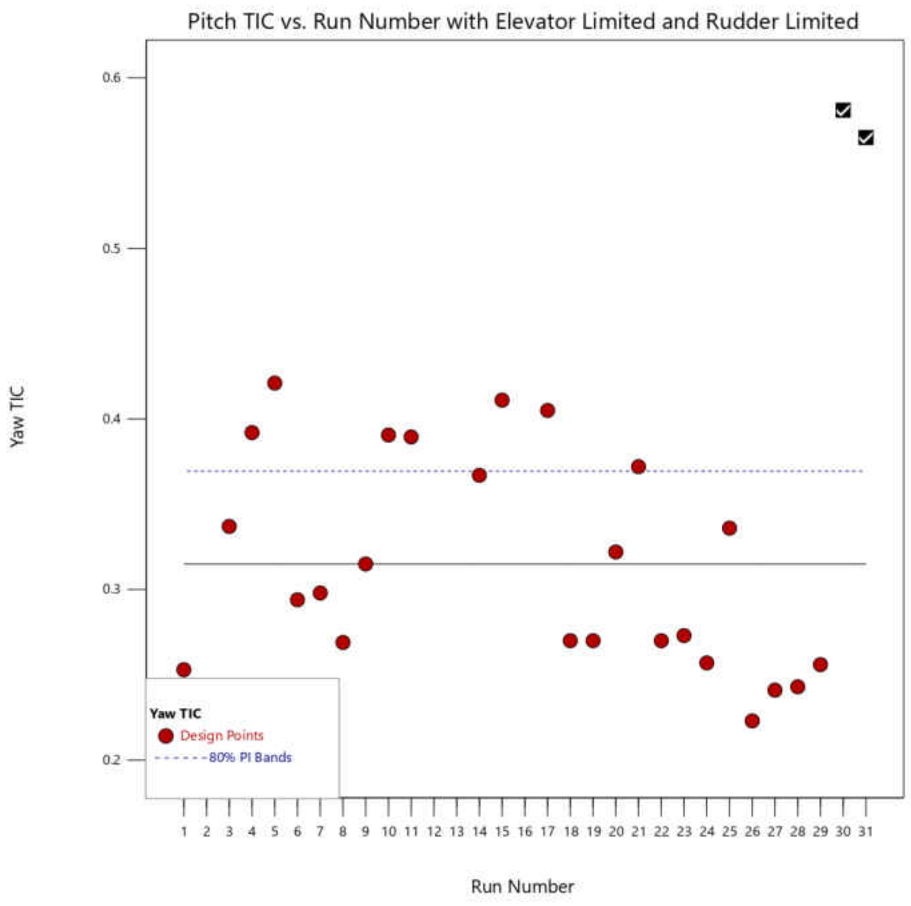


Figure 67-Pitch TIC vs. run number with elevator two and rudder two travel limited

CHAPTER 9

DISCUSSION

9.1 MOMENT OF INERTIA STUDY

Results show that ARX angular velocity models fit the roll and pitch axis better than the yaw axis, which is based on the fact that TIC values for roll and pitch are closer to zero while at the same time have less variance than the yaw axis TIC values. To further understand the reasoning behind this, the mass properties of the EdgeTRA are studied. Specifically, the moments of inertia (MOI) were measured for the fully configured EdgeTRA in a flight-ready state, including the flight battery. Since MOIs are not known for the EdgeTRA, the MOIs are experimentally determined using a Bifilar pendulum method, allowing variables in equation (34) to be determined.

$$I = \frac{WA^2t^2g}{16pi^2L_{Bifilar}} \tag{34}$$

In this method, the EdgeTRA is suspended from two wires oscillating about each principal axis. Simultaneously, the time duration for a desired number of cycles is recorded to calculate the period [51, 52]. Using two support lines with a known length, $L_{Bifilar}$, which are separated by some known distance, A, the Bifilar pendulum method suspends the EdgeTRA about its center of gravity. While these two variables are constants, the lengths vary due to the changing orientation of the EdgeTRA in determining the MOI for each axis. For example, for estimation of the I_{xx} MOI, the EdgeTRA was required to be oriented nose up, and the two Bifilar lines were attached to a mounting point 8 feet above the ground. The value of $L_{Bifilar}$ for I_{xx} is relatively short, which prevents the 6 foot long fuselage of the EdgeTRA from touching the ground in the nose up configuration, as shown in Figure 68. In comparison, the same 8-foot mounting point was used in a setup to oscillate about the z-axis, as shown in Figure 68 for I_{zz}. This configuration allows the $L_{Bifilar}$ lines to be longer, as the EdgeTRA is close to resting on its landing gear rather than the rudder.

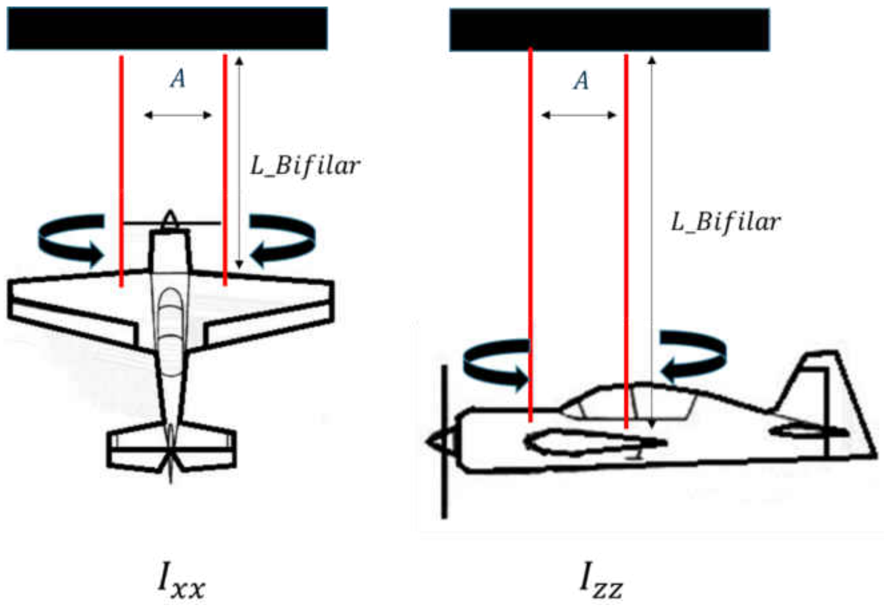


Figure 68-Bifilar MOI suspension configuration for I_{xx} and I_{zz}

A total of 10 oscillations were timed with a stopwatch to determine the oscillation period for each axis. Since the stopwatch's exact start and stop is subject to human error, five sets of 10 oscillations each were timed so the period could be averaged. All periods for each experiment and Bifilar wire lengths are shown in APPENDIX E.

The determined MOIs from the Bifilar experiment are shown in Table 20, in addition to the mass and center of gravity locations for the flight-ready EdgeTRA.

Mass	3.89 kg
CoG_x	0.107 m
CoG_z	0.012 m
Ixx	0.157 $kg\ m^2$
Iyy	0.527 $kg\ m^2$
Izz	0.589 $kg\ m^2$
Ixz	0.331 $kg\ m^2$

Table 20-Mass and experimentally determined MOI properties of the EdgeTRA

The MOIs describe how a rotational movement about an axis resists a change in direction [53]. Motion about the roll axis (x) is relatively unimpeded due to the low magnitude of I_{xx}, allowing roll changes to happen quickly. In comparison, I_{zz} was found to have a large order of magnitude, which means more resistance to change in the yaw direction. The larger magnitude in I_{zz} is understandable as the combination of the mass of the fuselage and wings affects this axis, which allows the EdgeTRA to yaw more than commanded. For example, the EdgeTRA is flying in a straight line at trim conditions. The rudder is commanded to yaw the EdgeTRA to the right for a 0.5 second period and then immediately following, commanded to yaw left for 0.5 seconds. Since I_{zz} is large, the change in yaw direction is not instantaneous, allowing the EdgeTRA yaw motion to overshoot the commanded yaw input. Drifting past the commanded yaw input is problematic, as it introduces nonlinearity into the yaw axis in that the response does not directly correlate to the provided input, therefore making the linear ARX model incapable of predicting as well as shown in Figure 69. In the yaw rate versus time subplot, between 409 and 415 seconds, the estimated yaw rate indicated by the orange plus signs does not fully capture the measured yaw rate indicated by the blue dots.

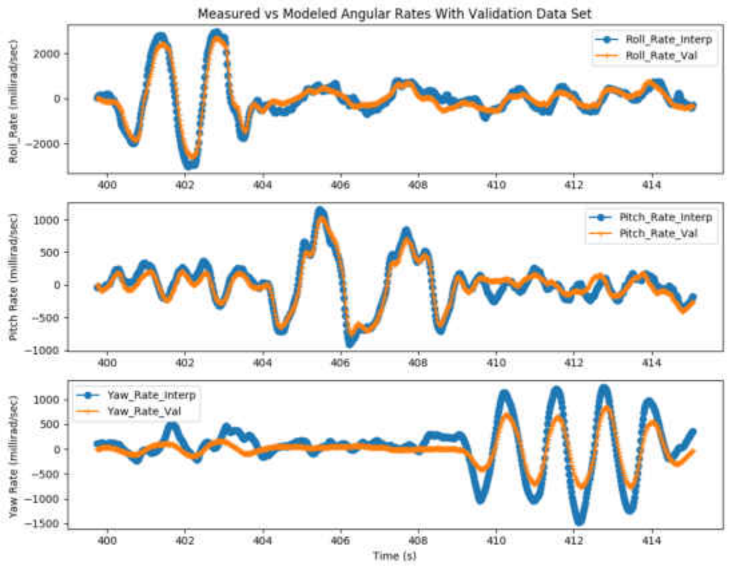


Figure 69-Fitted output under nominal conditions for run MIMO_04_05_2020__18_21_23

However, it was found that if greater amplitude input deflections of the rudder are applied, this produces a large enough yawing moment capable of overcoming I_{zz} more quickly, which allowed for a more linear input to output relation, as shown in Figure 70. The yaw rate versus time plot between 327 to 332 seconds, as the model Yaw_Rate_Val, fits the measured data, Yaw_Rate_Interp, better visually as indicated by the yaw TIC value of 0.257. In comparison to Figure 69, a smaller rudder deflection input was applied, which resulted in a larger yaw TIC value of 0.336, indicating a less desirable fit.

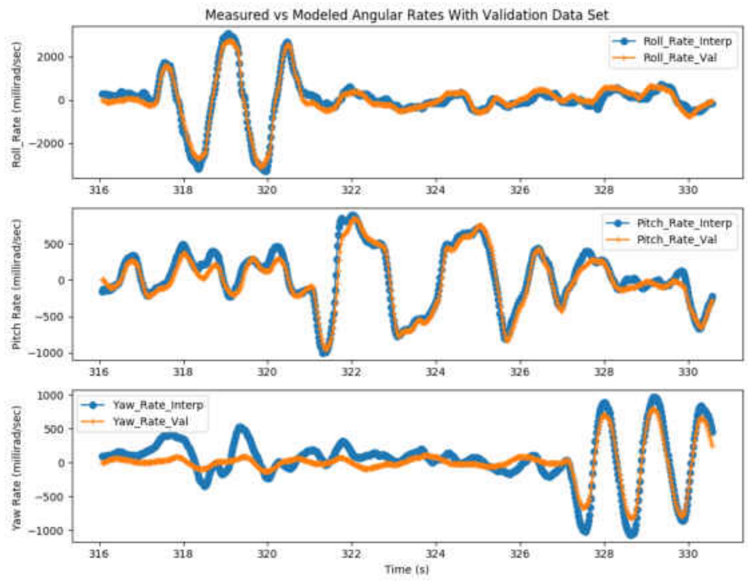
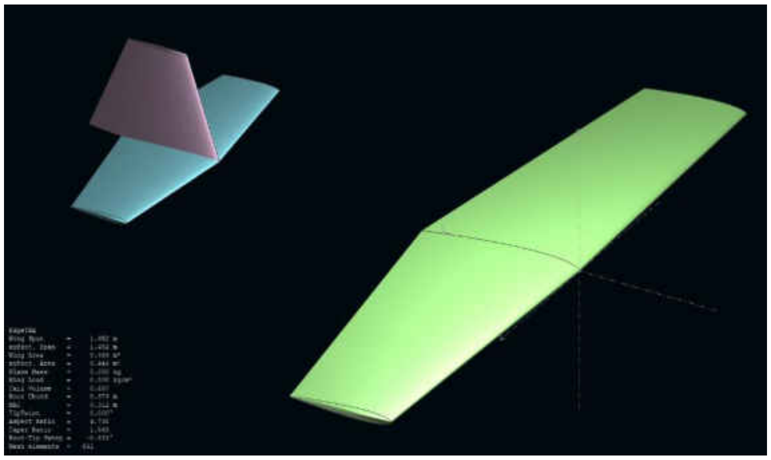


Figure 70-Fitted output under nominal conditions for run MIMO_04_05_2020__18_19_57

This phenomenon did not occur for the roll or pitch axis, although the pitch axis MOI, I_{yy}, was lower but close to I_{zz}. Not seeing nonlinearity in the pitch axis is believed to be due to the elevators having double the surface area compared to the rudder. Having double the surface area increases the force capable of overcoming the pitch MOI and aids in a more linear input to output relation, similar to when greater amplitude inputs to the rudder were applied. The fit was found to be better. In summary, when using the linear ARX modeling technique, the proper amplitude of excitation is critical to acquire a model that predicts well.

9.2 XFLR5 DYNAMIC STABILITY

An XFLR5 model of the EdgeTRA, as shown in Figure 71, was developed to give a better understanding of the dynamic stability. The model is built by providing mass properties, physical dimensions, and the airfoil for the wing and tail. The exact airfoil for the wing and empennage of the EdgeTRA is unknown. However, the NACA 0011 airfoil is a close match and used throughout in the XFLR5 model.



```
EdgeTRA
Wing Span        =   [illegible] m
xyProj. Span     =   [illegible] m
Wing Area        =   [illegible] m²
xyProj. Area     =   [illegible] m²
Plane Mass       =   [illegible] kg
Wing Load        =   [illegible] kg/m²
Tail Volume      =   [illegible]
Root Chord       =   [illegible] m
MAC              =   [illegible] m
TipTwist         =   [illegible]°
Aspect Ratio     =   [illegible]
Taper Ratio      =   [illegible]
Root-Tip Sweep   =   [illegible]°
Mesh elements    =   [illegible]
```

Figure 71- EdgeTRA XFLR5 dynamic model

Aircraft dynamics are divided into longitudinal and lateral groups. Within the longitudinal group, two modes are contained, phugoid and short-period mode. Phugoid mode is slow, lightly damped oscillations, and short-period mode is a high frequency or fast oscillations that are moderately damped in the pitch axis. Within the lateral group, there are three different modes: roll, spiral, and dutch roll. Roll mode pertains to moderately damped low-frequency oscillations, the spiral mode has low dampening with low frequency, and the dutch roll mode is moderately damped with high frequency. Using eigenvalues, each of these modes can be identified for the EdgeTRA. Table 21 shows the eigenvalues of the dynamic modes of the EdgeTRA obtained from XFLR5.

Longitudinal	**Eigenvalue**
Phugoid	$-0.0091 \pm 0.5859i$
Short Period	$-5.7136 \pm 6.1030i$
Lateral	**Eigenvalue**
Roll Mode	$0.1411 \pm 0.00i$
Spiral Mode	$85.22 \pm 0.00i$
Dutch Roll	$-2.0422 \pm 4.7302i$

Table 21- Eigenvalues from XFLR for EdgeTRA

Plotting the eigenvalues in a root locus plot allows dynamic modes of the EdgeTRA to be visually shown as in Figure 72 for longitudinal modes and Figure 73 for lateral modes. For the imaginary axis, as a closed-loop pole moves further away from the origin, the frequency increases. If a closed-loop pole moves more negative in the real axis, then this relates to increased dampening. Therefore, each mode can be identified based on the expectation of how the mode behaves. For example, the longitudinal phugoid mode is known to have low frequency with a small dampening amount, which can be found on the longitudinal root locus near the origin.

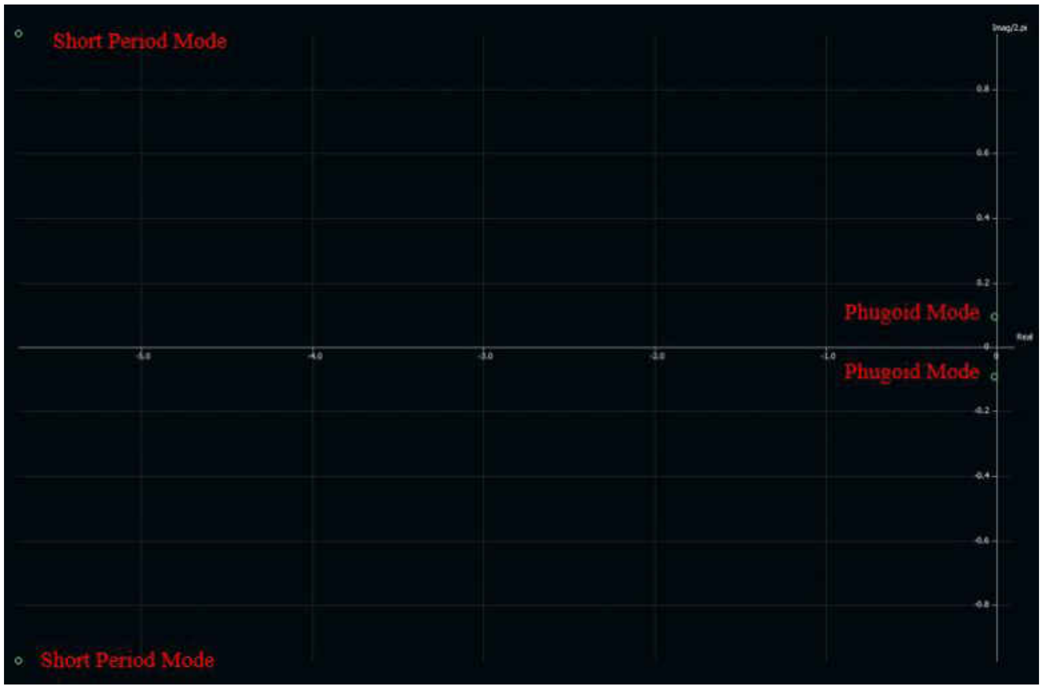


Figure 72-Root locus plot of longitudinal modes for EdgeTRA

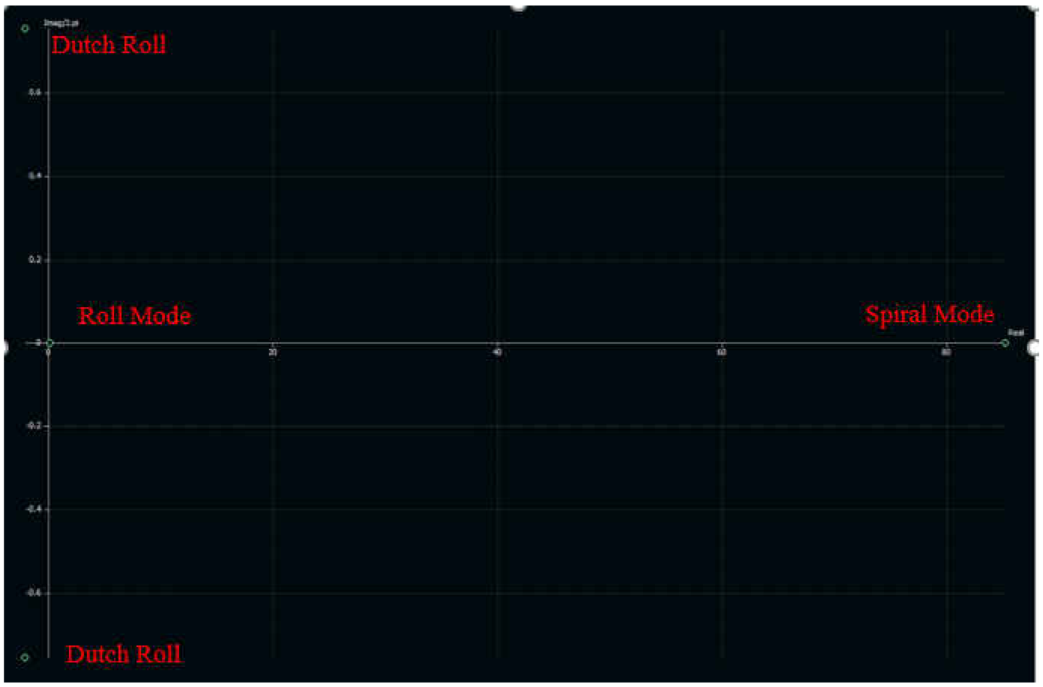


Figure 73-Root locus plot for lateral modes for EdgeTRA

Knowing the eigenvalues of the EdgeTRA, can provide additional reasoning for why the yaw angular velocity model does not predict as well as the roll and pitch models. The lateral eigenvalues for the roll and spiral modes show the EdgeTRA to be unstable laterally. The Cube Orange provides active control to the roll and pitch axis in both manual PIC input and automatic input control methods. However, no active control is provided to the yaw axis. Therefore, the roll axis's lateral instability is compensated for by the Cube Orange, but that is not the case for the yaw axis.

Not compensating for this instability affects the modeling by allowing a response to be present when there is no correlated input, as shown in the red box in Figure 74. Specifically, it is shown that even when the rudder input is constant, between 335 seconds and 349 seconds, there are still oscillations in the yaw axis output believed to be due to lateral instability. This instability reduces the direct correlation of rudder input to the yaw rate response, making it non-linear, which reduces the linear yaw rates model ability to predict well. During the time frame encompassed by the red box, roll and pitch input maneuvers are implemented. The rudder input applied between 340 seconds and 345 seconds is due to the previously discussed mixing of the aileron and the rudder. Otherwise, the input signal should be constant until it's time for the yaw axis to be excited by the rudder.

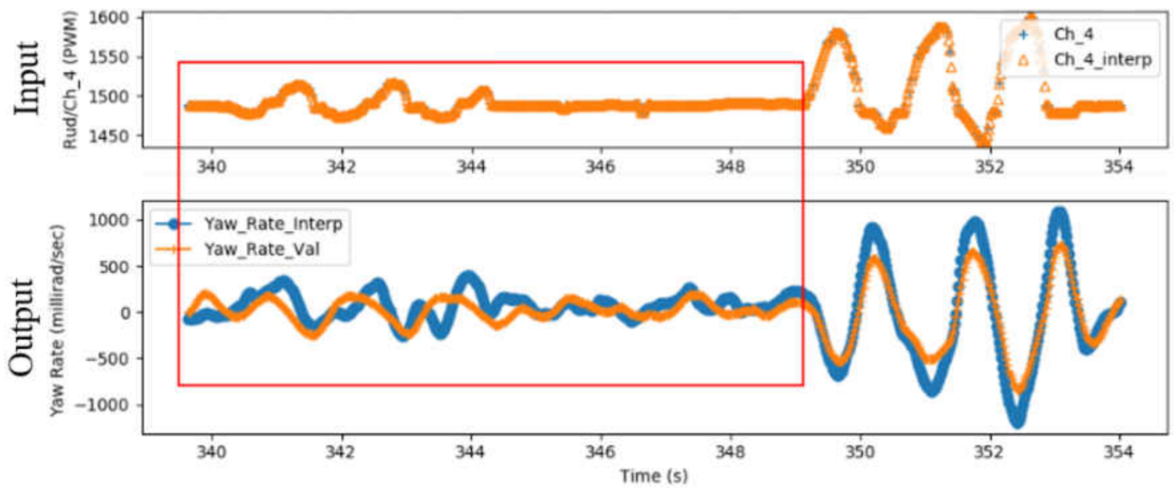


Figure 74-Rudder input and yaw rate response without active control

Compared to the roll axis, which is also affected by the same lateral instability, the aileron input is never truly constant as the flight controller compensates for instability and external disturbances, such as wind. By compensating, this provides the model with a more correlated roll input to roll rate output for the linear roll rate model since this compensated input is used to build the roll rate model, which provides a better fit. An example of this is shown in Figure 75. After the roll excitation has been completed, 345 sec and greater, the measured Roll_Rate_Interp data fits the roll rate model data, Roll_Rate_Val better throughout the run, in the output subplot.

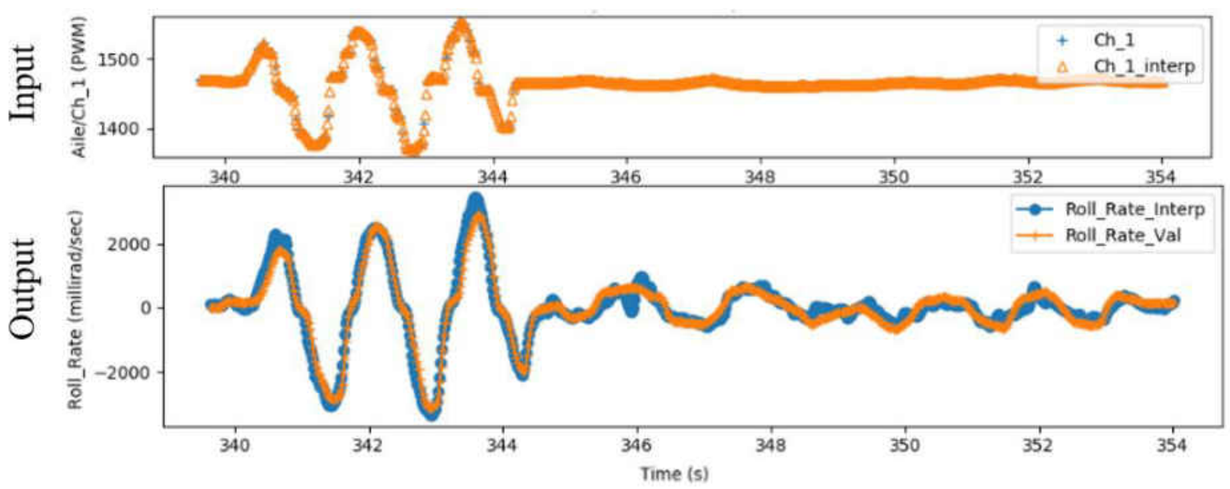


Figure 75-Aileron input and roll rate response with active control

CHAPTER 10

CONCLUSIONS AND FUTURE WORK

This work has shown that it's possible to use response models for roll, pitch, and yaw angular velocities as a function of primary control inputs to detect a lack of controllability in a sUAV. An entirely onboard controllability detection system was demonstrated using a COTS flight controller and aircraft model with no knowledge of mass properties or servo deflection angles and a minimum additional sensor suite consisting of airspeed, GPS antenna, and RPI.

Data collection was performed using MAVLink messages, a common serial communication protocol used by the Cube Orange flight controller running ArduPlane firmware. MAVLink messages gathered sensor data from the Cube Orange and transmitted them to the RPI companion computer. These messages transmitted commands from the RPI to the Cube Orange to perform maneuvers and change flight modes. It was found that these messages have secondary priority to any main flight control functions. Therefore, the messages' rate was not constant, which was problematic as the ARX model was in discrete time. In order to correct this inconsistent message rate, MAVLink messages were linearly interpolated based upon the average data rate of a particular message group.

The MIMO ARX *black-box* modeling technique was used to identify transfer function models of the roll, pitch, and yaw angular velocities using only the input and the system's output. The models were validated using newly collected input and output data, where the input is passed through the model to estimate the output, which is then compared to the measured output. Using TIC as a metric for goodness of fit allowed the comparison of the modeled and the measured angular velocity on a scale from zero to one. A simple threshold for TIC based on a prediction interval proved useful in this first effort but may be improved upon with a TIC rating scale in the future rather than a go/no-go value.

Since the collected data is experimental, it was found to be susceptible to sensor noise, pilot learning, and weather disturbances such as wind. The TIC value varies from run to run because of this. A series of nominal runs are made to determine the TIC value threshold of the EdgeTRA under nominal conditions. A prediction interval is used as the threshold, created from nominal runs, determining if a run is nominal or abnormal based on its TIC value. If the prediction interval is increased by reducing alpha, this gives greater acceptance that the aircraft is nominal while reducing the acceptance that the EdgeTRA is abnormal. A decreased prediction interval or an increased choice of alpha does the opposite, by accepting more of the TIC range as abnormal and less nominal. Therefore, based on the collected results of the nominal runs, a compromise was made to set the prediction interval to 80%. Setting alpha to 80% increases the chance for a false positive but simultaneously increases the chance to detect a lack of controllability. Again, future work could look at a graded scale.

For this work, a Lack of Controllability is defined as any roll, pitch, or yaw axis with a TIC value that falls above the established prediction interval of 80%. A total of 10 different failure modes were developed to simulate possible modes of failure to test for the controllability of the EdgeTRA. The failure modes tested the control authority of a single axis as well as multiple axes simultaneously. Failures were simulated by either completely failing a servo or limiting the travel. Results show that a lack of controllability is detected when appropriate with minimal false alarms. Even in the case of the limited throw authority, detection of a lack of controllability still occurred, showing the method is sensitive to small changes from nominal conditions. This finding is felt to be significant as detecting small changes in controllability is essential before it catastrophically affects the aircraft.

In future work, this controllability diagnostic could benefit from real-time implementation. Currently, the developed lack of controllability detection system only runs and reports the status of the aircraft when commanded. However, the developed method does not disrupt the mission of the sUAV, and testing can be done en route to the next waypoint and can be performed many times during a flight.

Additionally, the ability to detect the direct cause of the controllability problem would be a subject for future work. For example, there is a lack of controllability detected on the roll axis. The reason could be due to a failing servo, damaged linkage, loss of covering to the wing, or wing structural failure. Knowing the cause of failure would help resolve the problem and decide the next action for resolution.

As the foundation of this work's data collection is based on MAVLink messages, future work would include investigating other airframe types such as multi-copters and VTOL sUAVs. The same concept of roll, pitch, and yaw angular velocity models can be used, except that many more inputs can be added. For example, in an octocopter, the signal to each motor can be used as the input, which replaces the aileron, elevator, and rudder used for traditional fixed-wing sUAVs. The output stays the same as roll, pitch, and yaw angular velocity. A VTOL vehicle, such as the *Langley Aerodrome No. 8*, which is a mix of a multi-copter and fixed-wing sUAV again, could follow the same approach. This aircraft has 21 different inputs that affect roll, pitch, and yaw angular velocities, which differ in hover and forward flight conditions. Therefore, future work could increase understanding of how well this controllability detection diagnostic functions as more inputs are added to the system, and complexity increases.

REFERENCES

1. Cai, G., B.M. Chen, and T.H. Lee, *Unmanned Rotorcraft Systems*. 2011: Springer-Verlag London.
2. Bertin, J.J. and R.M. Cummings, *Aerodynamics for Engineers*. 6th ed. 2014: Pearson.
3. Behnke, S. and M. Schreiber, *Digital Position Control for Analog Servos*, in *RAS International Conference on Humanoid Robots*. 2006, IEEE: Genoa, Italy. p. 56-61.
4. Ryan. *The Difference between Analog and Digital RC Servos*. 2020; Available from: https://www.radiocontrolinfo.com/the-difference-between-analog-and-digital-rc-servos/.
5. Basu, A., S.A. Moosavian, and R. Morandini, *Mechanical Optimization of Servo Motor.* Journal of Mechanical Design, 2005.
6. McManis, C. *R/C Servos 101*. 1995; Available from: http://pages.cs.wisc.edu/~bolo/shipyard/servos101.html.
7. *HS-311 Standard Economy Servo*. 2020; Available from: https://hitecrcd.com/products/servos/analog/sport-2/hs-311/product.
8. Lyons, M., K. Brandis, C. Callaghan, J. McCann, C. Mills, S. Ryall, and R. Kingsford, *Bird interactions with drones, from individuals to large colonies.* 2017.
9. Forrest, S., A.S. Perelson, L. Allen, and R. Cherukuri. *Self-nonself discrimination in a computer*. in *Proceedings of 1994 IEEE Computer Society Symposium on Research in Security and Privacy*. 1994.
10. Ishida, Y., *An immune network model and its applications to process diagnosis.* Systems and Computers in Japan, 1993. **24**(6): p. 38-46.
11. Jerne, N.K., *Towards a network theory of the immune system.* Ann Immunol (Paris), 1974. **125c**(1-2): p. 373-89.

12. Dasgupta, D., *Artificial Immune Sytems and Their Applications*, ed. D. Dasgupta. 1999: Springer.

13. Timmis, J. and P. Andrews, *A Beginners Guide to Artificial Immune Systems*. 2007, Boston, MA: Springer.

14. Garcia, D., H. Moncayo, A. Perez, and C. Jain, *Low cost implementation of a biomimetic approach for UAV health management.* IEEE, 2016: p. 2265-2270.

15. Herrera, D.F.G., *Design, Development and Implementation of Intelligent Algorithms to Increase Autonomy of Quadrotor Unmanned Missions*, in *Graduate Studies*. 2017, Embry-Riddle.

16. Lopez, K.P.R., H. Moncayo, J.A. Verberne, and D. F., *Design and Implementation of Intelligent Decision-Making Algorithms for Unmanned Aerial Vehicles Mission Protection*, in *AIAA Scitech 2019 Forum*.

17. Sanchez, S., M. Perhinschi, H. Moncayo, M. Napolitano, J. Davis, and M. Fravolini, *In-Flight Actuator Failure Detection and Identification for a Reduced Size UAV Using the Artificial Immune System Approach*, in *AIAA Guidance, Navigation, and Control Conference*.

18. Perhinschi, M.G., H. Moncayo, and D.A. Azzawi, *Integrated Immunity-Based Framework for Aircraft Abnormal Conditions Management.* Journal of Aircraft, 2014. **51**(6): p. 1726-1739.

19. Quan, Q., *Introduction to Multicopter Design and Control*. 2017, Singapore Springer Nature.

20. Han, L., Y. Li, S. Mi, and H. Liu, *Research on compass error compensation of certain UAS*, in *Information and Automation* 2013, IEEE: Yinchuan, China.

21. Tridgell, A., F. Ferreira, G. Morphett, J. Walser, L.D. Marchi, M.d. Breuil, P. Barker, R. Mackay, T. Pittenger, B. Geyer, C. Olson, E. Castelnuovo, E. Shamaev, G. Staroselskii, G.J.d. Sousa, J. Beraud, L. Hall, M. Lawrence, M. Badaire, M. Denecke, P. Riseborough, P. Kancir, V.M. Vilches, A. Lucas, and S. Tabor. *ArduPilot*. 2016; Available from: https://ardupilot.org/.

22. Meier, L., D. Agar, J. Oes, B. Küng, M. Grob, H. Willee, D. Sidrane, R. Bapst, M. Bresciani, J. Vautherin, J. Kent, M. Rivizzigno, D. Gagne, G. Grubba, N. Marques, P. Kirienko, and JaeyoungLim. *PX4 Autopilot User Guide*. 2020; Available from: https://px4.io/.

23. Fuggetti, G., A. Ghetti, and M. Zanzi, *Safety Improvement of Fixed Wing Mini-UAV Based on Handy FDI Current Sensor and a FailSafe Configuration of Control Surface Actuators* IEEE, 2015.

24. Gertler, J.J., *Fault Detection and Diagnosis in Engineering Systems*. 1st ed. 2017: CRC Press. 504.

25. Ding, S., *Model-based fault diagnosis techniques: Design schemes, algorithms, and tools*. 2008.

26. Freeman, P., R. Pandita, N. Srivastava, and G.J. Bakas, *Model-Based and Data-Driven Fault Detection Performance for a Small UAV.* IEEE/ASME Transactions on Mechatronics 2013. **18**(4): p. 1300-1309.

27. Freeman, P. and G.J. Balas, *Analytical Fault Detection for a Small UAV*, in *AIAA Infotech@Aerospace (I@A) Conference*.

28. Yechout, T.R., S.L. Morris, D.E. Bossert, W.F. Hallgren, and J.K. Hall, *Introduction to Aircraft Flight Mechanics*. Second ed. 2014: American Institute of Aeronautics and Astronautics.

29. Morelli, E.A. and V. Klein, *Aircraft System Identification Theory and Practice*. Vol. 2. 2016: Sunflyte Enterprises.

30. Favaregh, N.M., *Global Modeling of Pitch Damping From Flight Data*, in *Aerospace Engineering*. 2006, Old Dominion University.

31. Dorobantu, A., A. Murch, B. Mettler, and G. Balas, *System Identification for Small, Low-Cost, Fixed-Wing Unmanned Aircraft.* Journal of Aircraft, 2013. **50**(4): p. 1117-1130.

32. Perry, A.T., T. Bretl, and P.J. Ansell, *System Identification and Dynamics Modeling of a Distributed Electric Propulsion Aircraft.* AIAA, 2019.

33. Woodrow, P., M. Tischler, G. Mendoza, S.G. Hagerott, and J. Hunter, *Low Cost Flight-Test Platform to Demonstrate Flight Dynamics Concepts using Frequency-Domain System Identification Methods*, in *AIAA Atmospheric Flight Mechanics (AFM) Conference*.

34. Vaidyanathan, P.P., *Generalizations of the sampling theorem: Seven decades after Nyquist.* IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications, 2001. **48**(9): p. 1094 1109.

35. *ARX Time Series Model*. 2019 [cited 2020; Available from: https://apmonitor.com/wiki/index.php/Apps/ARXTimeSeries.

36. Chetouani, Y., *Using ARX Approach For Modeling and Prediction of the Dynamics of a Reactor-Exchanger*. 2008, Université de Rouen, : IChemE.

37. Nise, N.S., *Control System Engineering*. 7th ed. 2014: Wiley. 944.

38. Haldar, A. and S. Mahadevan, *Probability, Reliability, and Statistical Methods in Engineering Design*. 1999: Wiley. 320.

39. Bliemel, F., *Theil's Forecast Accuracy Coefficient: A Clarification.* Journal of Marketing Research, 1973. **10**(4): p. 444-446.

40. Montgomery, D.C., *Design and Analysis of Experiments* 2013: John Wiley and Sons,Inc.

41. Whitmore, G.A., *Prediction Limits for a Univariate Normal Observation.* The American Statistician, 1986. **40**(2): p. 141-143.

42. *Point Prediction Intervals*. Design-Expert [cited 2020; Available from: https://www.statease.com/docs/v11/contents/analysis/point-prediction-intervals/.

43. Anderson, D.R., K.P. Burnham, and W.L. Thompson, *Null Hypothesis Testing: Problems, Prevalence, and an Alternative.* The Journal of Wildlife Management, 2000. **64**(4): p. 912-923.

44. Edward, P., S. Elzeiny, M. Ashour, and T. Elshabrawy, *On the Coexistence of LoRa- and Interleaved Chirp Spreading LoRa-Based Modulations*, in *International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*. 2019: Barcelona, Spain.

45. *pymavlink*. 2018; Available from: https://github.com/ArduPilot/pymavlink.

46. *MAVLINK*. 2020; Available from: https://mavlink.io/en/messages/common.html.

47. *SITL Simulator*. 2020; Available from: https://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html.

48. Armenise, G., M. Vaccari, R.B.D. Capaci, and G. Pannocchia, *An Open-Source System Identification Package for Multivariable Processes.* IEEE, 2018.

49. Phillips, W.F. and B.W. Santana, *Aircraft Small-Disturbance Theory with Longitudinal-Lateral Coupling.* Journal of Aircraft, 2002. **39**(6): p. 973-980.

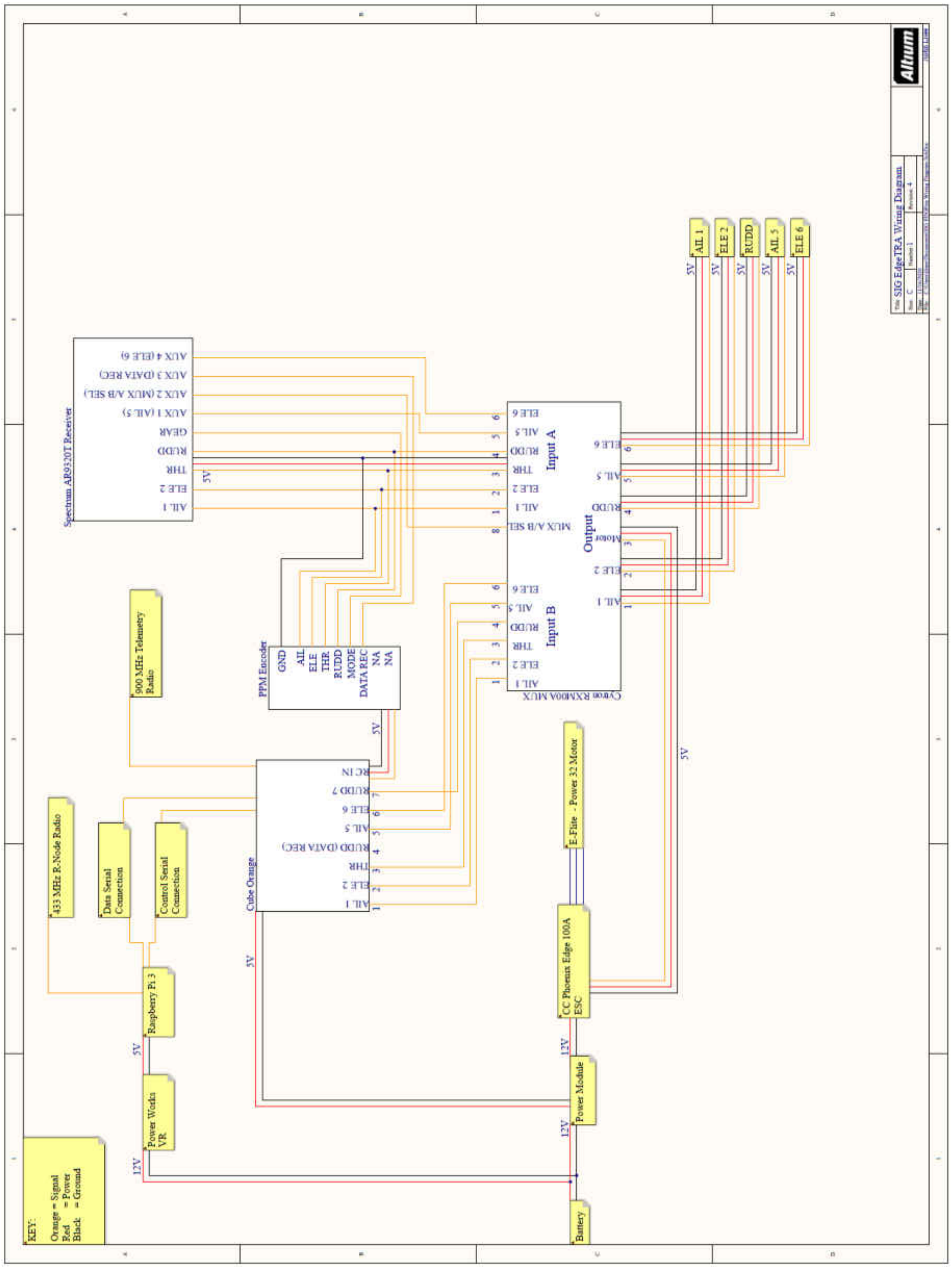
50. Gudmundsson, S., *Chapter 11 - The Anatomy of the Tail*, in *General Aviation Aircraft Design*, S. Gudmundsson, Editor. 2014, Butterworth-Heinemann: Boston. p. 459-519.

51. Jardin, M. and E. Mueller, *Optimized Measurements of UAV Mass Moment of Inertia with a Bifilar Pendulum*, in *AIAA Guidance, Navigation and Control Conference and Exhibit*.

52. Dantsker, O.D., M. Vahora, S. Imtiaz, and M. Caccamo, *High Fidelity Moment of Inertia Testing of Unmanned Aircraft*, in *2018 Applied Aerodynamics Conference*.

53. COLLINS, J.D. and W.T. THOMSON, *Statistics of rigid body moment of inertia computation.* Journal of Spacecraft and Rockets, 1967. **4**(12): p. 1673-1675.

APPENDIX A


SIG EdgeTRA Wiring Diagram
Revision 4
Altium
Spectrum AR9320T Receiver
AIL 1
ELE 2
THR
RUDD
GEAR
AUX 1 (AIL 5)
AUX 2 (MUX A/B SEL)
AUX 3 (DATA REC)
AUX 4 (ELE 6)
5V
Cytron RXM00A MUX
Input A
Input B
Output
MUX A/B SEL
ELE 6
AIL 5
Motor
PPM Encoder
GND
AIL
ELE
THR
RUDD
MODE
DATA REC
NA
NA
Cube Orange
RC IN
RUDD 7
ELE 6
AIL 5
RUDD (DATA REC)
THR
ELE 2
AIL 1
900 MHz Telemetry Radio
433 MHz R-Node Radio
Data Serial Connection
Control Serial Connection
Raspberry Pi 3
Power Works VR
E-Flite - Power 32 Motor
CC Phoenix Edge 100A ESC
Power Module
Battery
12V
KEY:
Orange = Signal
Red = Power
Black = Ground

APPENDIX B

```
#!/usr/bin/env python
################################################################
#Writen by Brian Duvall January 2020
#Collects Data from ArduPlane Firmware and builds MIMO models of p,q,r
#DT is fixed by interpolating all data after recoding to  the Time
#Debug help:
#import code
#code.interact(local=locals())
################################################################

import sys, os
from optparse import OptionParser
import time
import numpy as np
import math
import matplotlib
matplotlib.use('Agg')  #This lets plts run over ssh but prevents output
try:
    from SIPPY import *
except ImportError:
    import sys, os
    sys.path.append(os.pardir)
    from SIPPY import *
from SIPPY import functionset as fset
from SIPPY import functionsetSIM as fsetSIM
import control as cnt
from control.matlab import *
import pandas
import matplotlib.pyplot as plt
from datetime import datetime
from pymavlink import mavutil
from distutils.version import StrictVersion
if StrictVersion(cnt.__version__) >= StrictVersion('0.8.2'):
    lsim = cnt.matlab.lsim
else:
    def lsim(sys, U = 0.0, T = None, X0 = 0.0):
        U_ = U
        if isinstance(U_, (np.ndarray, list)):
            U_ = U_.T
        return cnt.matlab.lsim(sys, U_, T, X0)

ROLL_DATA_SET = 0
PITCH_DATA_SET = 0
YAW_DATA_SET = 0
All_AXIS_DATA_SET = 0
Roll_ID_SYS = None
Pitch_ID_SYS = None
Yaw_ID_SYS = None
MIMO_ID_SYS = None
Data_Model = None
File_Name = None
Data_RAW_Model = None
```

```
def handle_heartbeat(msg):
    mode = mavutil.mode_string_v10(msg)
    is_armed = msg.base_mode & mavutil.mavlink.MAV_MODE_FLAG_SAFETY_ARMED
    is_enabled = msg.base_mode & mavutil.mavlink.MAV_MODE_FLAG_GUIDED_ENABLED

def handle_rc_raw(msg):
#This is the input from RX to Pixhawk
    channel_1 = msg.chan1_raw #Aileron Right
    channel_2 = msg.chan2_raw #Elevator
    channel_3 = msg.chan3_raw #Throttle
    channel_4 = msg.chan4_raw #Rudder
    channel_5 = msg.chan5_raw #Mode switch
    channel_6 = msg.chan6_raw #Data record start stop
    rc_in_time = (msg.time_boot_ms)*0.001  #Time since boot of each message
    return channel_1, channel_2, channel_3, channel_4, channel_5, channel_6,
rc_in_time

def handle_rc_raw_out(msg):
#This is the output side from Pixhawk to the servo
    channel_1_out = msg.servo1_raw #Aileron Right
    channel_2_out = msg.servo2_raw #Elevator Left
    channel_3_out = msg.servo3_raw #Throttle
    channel_4_out = msg.servo4_raw #Rudder
    channel_5_out = msg.servo5_raw #Aileron Left
    channel_6_out = msg.servo6_raw #Elevator Right
    rc_out_time = (msg.time_usec)*0.000001 # Time when the mavlink message is
created
    return channel_1_out, channel_2_out, channel_3_out, channel_4_out,
channel_5_out, channel_6_out, rc_out_time

def handle_attitude(msg):
    attitude_data = (msg.roll, msg.pitch, msg.yaw, msg.rollspeed,
    msg.pitchspeed, msg.yawspeed)

def handle_raw_imu(msg):
    raw_imu_time = (msg.time_usec)*0.000001 #Time at which the IMU message is
created
    raw_imu_roll = msg.xgyro #Roll rate
    raw_imu_pitch = msg.ygyro #Pitch rate
    raw_imu_yaw = msg.zgyro # Yaw rate
    return raw_imu_time,raw_imu_roll, raw_imu_pitch, raw_imu_yaw

def handle_VFR_HUD(msg):
    air_speed = msg.airspeed
    return air_speed

def Store_Model_Data_CSV(MIMO_ID_SYS, Roll_Rate_Model, Pitch_Rate_Model,
Yaw_Rate_Model, TIC_Roll_Model, TIC_Pitch_Model, TIC_Yaw_Model):
    global File_Name
    Data_Model = pandas.DataFrame({'Id_SYS':MIMO_ID_SYS.G,
'Roll_Rate_Model':Roll_Rate_Model, 'Pitch_Rate_Model':Pitch_Rate_Model,
'Yaw_Rate_Model':Yaw_Rate_Model, 'TIC_Roll_Model':TIC_Roll_Model,
'TIC_Pitch_Model':TIC_Pitch_Model, 'TIC_Yaw_Model':TIC_Yaw_Model)

    Data_Model.to_csv('MODEL_DATA/MIMO/MODEL/MIMO_Model_'+ str(File_Name) )
    return
```

```
def Store_Validation_SIM_Data(Roll_Rate_Val, Pitch_Rate_Val, Yaw_Rate_Val,
TIC_Roll_Val, TIC_Pitch_Val, TIC_Yaw_Val):
    global File_Name
    Data_Val = pandas.DataFrame({'Roll_Rate_Val':Roll_Rate_Val,
'Pitch_Rate_Val':Pitch_Rate_Val, 'Yaw_Rate_Val':Yaw_Rate_Val,
'TIC_Roll_Val':TIC_Roll_Val, 'TIC_Pitch_Val':TIC_Pitch_Val,
'TIC_Yaw_Val':TIC_Yaw_Val})
    Data_Val.to_csv('MODEL_DATA/MIMO/VALIDATION/MIMO_Val'+ str(File_Name) )
    return

def Store_Data_RAW_CSV(Channel_1_interpolated, Channel_2_interpolated,
Channel_3_interpolated, Channel_4_interpolated, Channel_5_interpolated,
Channel_6_interpolated,
                       Channel_1_OUT_interpolated,
Channel_2_OUT_interpolated, Channel_3_OUT_interpolated,
Channel_4_OUT_interpolated, Channel_5_OUT_interpolated,
Channel_6_OUT_interpolated,
                       Roll_Rate_Interpolated, Pitch_Rate_Interpolated,
Yaw_Rate_Interpolated, Air_speed_VFR_interpolated, MIMO_DATA_SET, Time, DT,
DT_avg_IMU, DT_avg_RC, DT_avg_VFR):
    global Data_RAW_Model #Saves first data set until the second one is
collected
    global File_Name

    if MIMO_DATA_SET == 0:
        File_Save_Time = datetime.now()
        File_Name = File_Save_Time.strftime("%m_%d_%Y__%H:%M:%S")
        Data_RAW_Model = pandas.DataFrame({'Channel_1 (PWM)':
Channel_1_interpolated,'Channel_2 (PWM)':Channel_2_interpolated,'Channel_3
(PWM)':Channel_3_interpolated, 'Channel_4 (PWM)':Channel_4_interpolated,
'Channel_5 (PWM)':Channel_5_interpolated, 'Channel_6
(PWM)':Channel_6_interpolated,
                                   'Channel_1_OUT
(PWM)':Channel_1_OUT_interpolated,'Channel_2_OUT
(PWM)':Channel_2_OUT_interpolated, 'Channel_3_OUT
(PWM)':Channel_3_OUT_interpolated, 'Channel_4_OUT
(PWM)':Channel_4_OUT_interpolated,'Channel_5_OUT
(PWM)':Channel_5_OUT_interpolated, 'Channel_6_OUT
(PWM)':Channel_6_OUT_interpolated,
                                   'Roll_Rate
(millirad/sec)':Roll_Rate_Interpolated, 'Pitch_Rate (millirad/sec)':
Pitch_Rate_Interpolated,'Yaw_Rate (millirad/sec)':Yaw_Rate_Interpolated,
'Air_Speed_VFR':Air_speed_VFR_interpolated, 'Time (sec)':Time, 'DT':DT,
'DT_avg_IMU':DT_avg_IMU, 'DT_avg_RC':DT_avg_RC, 'DT_avg_VFR':DT_avg_VFR})

    if MIMO_DATA_SET != 0:
        Data_RAW_Val = pandas.DataFrame({'Channel_1 (PWM)':
Channel_1_interpolated,'Channel_2 (PWM)':Channel_2_interpolated,'Channel_3
(PWM)':Channel_3_interpolated, 'Channel_4 (PWM)':Channel_4_interpolated,
'Channel_5 (PWM)':Channel_5_interpolated, 'Channel_6
(PWM)':Channel_6_interpolated,
                           'Channel_1_OUT
(PWM)':Channel_1_OUT_interpolated,'Channel_2_OUT
(PWM)':Channel_2_OUT_interpolated, 'Channel_3_OUT
(PWM)':Channel_3_OUT_interpolated, 'Channel_4_OUT
(PWM)':Channel_4_OUT_interpolated,'Channel_5_OUT
(PWM)':Channel_5_OUT_interpolated, 'Channel_6_OUT
(PWM)':Channel_6_OUT_interpolated,
                           'Roll_Rate (millirad/sec)':Roll_Rate_Interpolated,
'Pitch_Rate (millirad/sec)': Pitch_Rate_Interpolated,'Yaw_Rate
(millirad/sec)':Yaw_Rate_Interpolated,
'Air_Speed_VFR':Air_speed_VFR_interpolated, 'Time (sec)':Time, 'DT':DT,
'DT_avg_IMU':DT_avg_IMU, 'DT_avg_RC':DT_avg_RC, 'DT_avg_VFR':DT_avg_VFR})
        All_Data = pandas.concat([Data_RAW_Model,Data_RAW_Val],
keys=['Model_Data', 'Validation_Data'])

        All_Data.to_csv('RAW_DATA/MIMO/MIMO_Raw_Data_'+ str(File_Name) )
    return

def Delta_Time(time):
    i=0
    sdeltatime=[]
    while i < len(time)-1:
        delta_time= time[i+1]-time[i]
        i=i+1
        sdeltatime= np.append(sdeltatime,delta_time)
    dt_avg = np.average(sdeltatime)
    return dt_avg, sdeltatime

def Master_Time(time, dt):
    Number_of_Samples = (time[len(time)-1] - time[0])/dt -1
#Not accounting for first and last sample
    Time = np.linspace(time[0], time[len(time)-1], Number_of_Samples + 2)
#The plus 2 accounts for the start and stop parts of linspace
    return(Time)

def TIC(Measured, Predictions):
    NUM = np.sqrt(((Predictions - Measured) ** 2).mean())
    DOM1 = np.sqrt(((Predictions)**2).mean())
    DOM2 = np.sqrt(((Measured)**2).mean())
    DOM_TOT = DOM1 + DOM2
    return NUM/DOM_TOT

def Airspeed(q):
#Not currently used
    rho = 1.225 #kg/m^3
    velocity = np.sqrt((q*0.1*2)/rho)
    print "velocity"
    return velocity

def Centering(array):
    mean = np.mean(array)
    centered_value = array-mean
    return centered_value

def Make_Model(y,u,dt,axis):
    ordersna = [2]
#Order for Output
    ordersnb = [[1,1]]
#Order for Input
    theta_list = [[1,0]]
#Time delay list
    id_sys=system_identification(y,u, 'ARX',centering='MeanVal',
ARX_orders=[ordersna, ordersnb, theta_list], tsample=dt)#Built SIMO model
    #print "Transfer function built for:", axis, "axis",id_sys.G
#Prints built TF model
    return(id_sys)

def Make_Model_MIMO(y,u,dt,axis):
    ordersna = [3,3,3]
#Order for Output
    ordersnb = [[2,2,2,2],[2,2,2,2],[2,2,2,2]]
#Order for Input
    ordersnc = [[1,1,1,1],[1,1,1,1],[1,1,1,1]]
    theta_list = [[2,2,2,2],[2,2,2,2],[2,2,2,2]]
#Time delay list
    #id_sys=system_identification(y,u, 'ARMAX',centering='MeanVal',
ARX_orders=[ordersna, ordersnb,ordersnc, theta_list], tsample=dt,
ARMAX_max_iterations = 500)#Built MIMO model
    id_sys=system_identification(y,u, 'ARX',centering='MeanVal',
ARX_orders=[ordersna, ordersnb, theta_list], tsample=dt)
    #print "Transfer function built for:", axis, "axis",id_sys.G
#Prints built TF model
    return(id_sys)


def SIM_OUTPUT(sys,u,master_time):
#master_time= Time for the run with fiexed width dt intervals see def
master_time
    time = master_time - master_time[0]
# Time must start at zero
    sim_output, T_lsim, Xsim = lsim(sys.G,u,time)
    return(sim_output)

def Process_All_Axis_MIMO(Channel_1, Channel_2, Channel_3, Channel_4,
Channel_5, Channel_6, Channel_IN_Time,
                          Channel_1_OUT, Channel_2_OUT, Channel_3_OUT,
Channel_4_OUT, Channel_5_OUT, Channel_6_OUT, Channel_Out_Time,
                          Time_IMU, Roll_Rate, Pitch_Rate, Yaw_Rate, Time_VFR
,Air_speed_VFR, MIMO_DATA_SET, Time, DT, DT_avg_IMU, SDelta_Time_IMU,
DT_avg_RC, SDelta_Time_RC, DT_avg_VFR, SDelta_Time_VFR):
    global MIMO_ID_SYS
    global File_Name
    Axis_type = "MIMO"

#############################################################################
##############################################
    #Reciver input to Pixhawk
    Channel_1_interpolated = np.interp(Time, Channel_IN_Time, Channel_1)#Roll
    Channel_2_interpolated = np.interp(Time, Channel_IN_Time,
Channel_2)#Pitch
    Channel_3_interpolated = np.interp(Time, Channel_IN_Time,
Channel_3)#Throtle
    Channel_4_interpolated = np.interp(Time, Channel_IN_Time, Channel_4)#Yaw
    Channel_5_interpolated = np.interp(Time, Channel_IN_Time, Channel_5)#Mode
switch
    Channel_6_interpolated = np.interp(Time, Channel_IN_Time, Channel_6)#Data
recorded start stop

#############################################################################
```

```
#############################################
    #Pixhawk input to servos
    Channel_1_OUT_interpolated = np.interp(Time, Channel_Out_Time,
Channel_1_OUT)#Aileron Right
    Channel_2_OUT_interpolated = np.interp(Time, Channel_Out_Time,
Channel_2_OUT)#Elevator
    Channel_3_OUT_interpolated = np.interp(Time, Channel_Out_Time,
Channel_3_OUT)#Throttle
    Channel_4_OUT_interpolated = np.interp(Time, Channel_Out_Time,
Channel_4_OUT)#Rudder
    Channel_5_OUT_interpolated = np.interp(Time, Channel_Out_Time,
Channel_5_OUT)#Aileron Left
    Channel_6_OUT_interpolated = np.interp(Time, Channel_Out_Time,
Channel_6_OUT)#Extra channel that can be used in the future

############################################################################
#############################################
    #Sensor input
    Air_speed_VFR_interpolated = np.interp(Time, Time_VFR, Air_speed_VFR)

############################################################################
#############################################
    #Interpolated measured responses
    Roll_Rate_Interpolated = np.interp(Time, Time_IMU, Roll_Rate) #Roll Rate
    Pitch_Rate_Interpolated = np.interp(Time, Time_IMU, Pitch_Rate) #Pitch
Rate
    Yaw_Rate_Interpolated = np.interp(Time, Time_IMU, Yaw_Rate) #Yaw Rate

############################################################################
#############################################
    #Input data used for lsim centered
    Roll_In_Center = Centering(Channel_1_OUT_interpolated)
    Pitch_In_Center = Centering(Channel_2_OUT_interpolated)
    Yaw_In_Center = Centering(Channel_4_OUT_interpolated)
    Velocity_Center = Centering(Air_speed_VFR_interpolated)

############################################################################
#############################################
    #Input and Output arrays built for modeling or lsim
    U = np.array([Channel_1_OUT_interpolated, Channel_2_OUT_interpolated,
Channel_4_OUT_interpolated, Air_speed_VFR_interpolated])
    U_Center = np.array([Roll_In_Center, Pitch_In_Center, Yaw_In_Center,
Velocity_Center])
    Y = np.array([Roll_Rate_Interpolated, Pitch_Rate_Interpolated,
Yaw_Rate_Interpolated])

############################################################################
#############################################

    #Saveing data that is all the same length
    Store_Data_RAW_CSV(Channel_1_interpolated, Channel_2_interpolated,
Channel_3_interpolated, Channel_4_interpolated, Channel_5_interpolated,
Channel_6_interpolated,
                       Channel_1_OUT_interpolated,
Channel_2_OUT_interpolated, Channel_3_OUT_interpolated,
Channel_4_OUT_interpolated, Channel_5_OUT_interpolated,
Channel_6_OUT_interpolated,
                       Roll_Rate_Interpolated, Pitch_Rate_Interpolated,
Yaw_Rate_Interpolated, Air_speed_VFR_interpolated, MIMO_DATA_SET, Time, DT,
DT_avg_IMU, DT_avg_RC, DT_avg_VFR)

    if MIMO_DATA_SET == 0:
        MIMO_ID_SYS = Make_Model_MIMO(Y, U, DT, Axis_type) #Makes MIMO
transfer function model


        MIMO_ID_Model = SIM_OUTPUT(MIMO_ID_SYS,U_Center,Time) #Based on
MIMO_ID_SYS the input data is used to simulate the responses

        Roll_Rate_Model = MIMO_ID_Model[:,0]  #Simulated Roll Rate
(millirad/sec)
        Pitch_Rate_Model = MIMO_ID_Model[:,1] #Simulated Pitch Rate
(millirad/sec)
        Yaw_Rate_Model = MIMO_ID_Model[:,2]   #Simulated Yaw Rate
(millirad/sec)

        TIC_Roll_Model = TIC(Y[0,:], Roll_Rate_Model)    #Estimates how well
the model fits the measured for roll rate with data used to build the model
        TIC_Pitch_Model = TIC(Y[1,:], Pitch_Rate_Model) #Estimates how well
the model fits the measured for pitch rate with data used to build the model
        TIC_Yaw_Model = TIC(Y[2,:], Yaw_Rate_Model)      #Estimates how well
the model fits the measured for yaw rate with data used to build the model
        print"TIC for Roll model is: " + str(TIC_Roll_Model)
        print"TIC for Pitch model is: " + str(TIC_Pitch_Model)
        print"TIC for Yaw model is: " + str(TIC_Yaw_Model)



        #Saveing the ID_TF and simulated responses from model data as well as
the TIC values
        Store_Model_Data_CSV(MIMO_ID_SYS, Roll_Rate_Model, Pitch_Rate_Model,
Yaw_Rate_Model,TIC_Roll_Model, TIC_Pitch_Model, TIC_Yaw_Model)



        #Check of interpolation for input model-building data set
        plt.figure(1)
        plt.subplot(511)
        Ch_1, = plt.plot(Channel_Out_Time, Channel_1_OUT,'+', label='Ch_1')
        Ch_1_interp, = plt.plot(Time,
Channel_1_OUT_interpolated,'^',markerfacecolor='None', label='Ch_1_interp')
        plt.ylabel('Aile/Ch_1 (PWM)')
        plt.legend(handles=[Ch_1, Ch_1_interp],loc='upper right')
        plt.title('Recorded vs Interpolated RC Input of Model Data Set')

        plt.subplot(512)
        Ch_2, = plt.plot(Channel_Out_Time, Channel_2_OUT,'+', label='Ch_2')
        Ch_2_interp, = plt.plot(Time,
Channel_2_OUT_interpolated,'^',markerfacecolor='None', label='Ch_2_interp')
        plt.ylabel('Ele/Ch_2 (PWM)')
        plt.legend(handles=[Ch_2, Ch_2_interp],loc='upper right')

        plt.subplot(513)
        Ch_3, = plt.plot(Channel_Out_Time, Channel_3_OUT,'+', label='Ch_3')
        Ch_3_interp, = plt.plot(Time,
Channel_3_OUT_interpolated,'^',markerfacecolor='None', label='Ch_3_interp')
        plt.ylabel('Thr/Ch_3 (PWM)')
        plt.legend(handles=[Ch_1, Ch_1_interp],loc='upper right')

        plt.subplot(514)
        Ch_4, = plt.plot(Channel_Out_Time, Channel_4_OUT,'+',label='Ch_4')
        Ch_4_interp, = plt.plot(Time,
Channel_4_OUT_interpolated,'^',markerfacecolor='None', label='Ch_4_interp')
        plt.ylabel('Rud/Ch_4 (PWM)')
        plt.legend(handles=[Ch_4, Ch_4_interp],loc='upper right')

        plt.subplot(515)
        Air_Speed, = plt.plot(Time_VFR, Air_speed_VFR,'+',
label='Air_Speed_VFR')
        Air_Speed_interp, = plt.plot(Time,
Air_speed_VFR_interpolated,'^',markerfacecolor='None',label='Air_Speed_VFR_in
terp')
        plt.ylabel('Air_Speed (m/s)')
        plt.xlabel('Time(sec)')
        plt.legend(handles=[Air_Speed, Air_Speed_interp],loc='upper right')

        plt.gcf().set_size_inches(11,8.5)
        plt.savefig('PLOTS/MIMO/MODEL/MIMO_Model_Input_' + str(File_Name))
#Saveing Input data set plots used to build the model


        #Check of interpolation for output model-building data set
        plt.figure(2)
        plt.subplot(311)
        Roll_rate, = plt.plot(Time_IMU, Roll_Rate,'+', label='Roll_Rate')
        Roll_rate_interp, = plt.plot(Time,
Roll_Rate_Interpolated,'^',markerfacecolor='None', label='Roll_Rate_Interp')
        plt.ylabel('Roll_Rate (millirad/sec)')
        plt.title('Recorded vs Interpolated Response of Model Data Set')
        plt.legend(handles=[Roll_rate, Roll_rate_interp],loc='upper right')

        plt.subplot(312)
        Pitch_rate, = plt.plot(Time_IMU, Pitch_Rate,'+', label='Pitch_Rate')
        Pitch_rate_interp, = plt.plot(Time,
Pitch_Rate_Interpolated,'^',markerfacecolor='None',label='Pitch_Rate_Interp')
        plt.ylabel('Pitch_Rate (millirad/sec)')
        plt.legend(handles=[Pitch_rate, Pitch_rate_interp],loc='upper right')

        plt.subplot(313)
        Yaw_rate, = plt.plot(Time_IMU, Yaw_Rate,'+', label='Yaw_Rate')
        Yaw_rate_interp, = plt.plot(Time,
Yaw_Rate_Interpolated,'^',markerfacecolor='None', label='Yaw_Rate_Interp')
        plt.ylabel('Yaw_Rate (millirad/sec)')
        plt.xlabel('Time(sec)')
        plt.legend(handles=[Yaw_rate, Yaw_rate_interp],loc='upper right')

        plt.gcf().set_size_inches(11,8.5)
        plt.savefig('PLOTS/MIMO/MODEL/MIMO_Model_Output_' + str(File_Name))
#Saveing Output data set plots used to build the model
```

```
        plt.figure(3)
        #Mesured Roll and Model roll
        plt.subplot(311)
        Roll_Rate_Interp, = plt.plot(Time, Roll_Rate_Interpolated,'-o',
label='Roll_Rate_Interp')
        Roll_Rate_model, = plt.plot(Time, Roll_Rate_Model,'-+',
label='Roll_Rate_Model')
        plt.ylabel('Roll_Rate (millirad/sec) ')
        plt.title('Measured vs. Modeled Angular Rates With Model Data Set')
        plt.legend(handles=[Roll_Rate_Interp, Roll_Rate_model],loc='upper
right')
        # Measured Pitch and Model Pitch
        plt.subplot(312)
        Pitch_Rate_Interp, = plt.plot(Time, Pitch_Rate_Interpolated, '-o',
label='Pitch_Rate_Interp')
        Pitch_Rate_model, = plt.plot(Time, Pitch_Rate_Model, '-+',
label='Pitch_Rate_Model')
        plt.ylabel('Pitch Rate (millirad/sec)')
        plt.legend(handles=[Pitch_Rate_Interp, Pitch_Rate_model],loc='upper
right')
        # Measured Yaw and Model Yaw
        plt.subplot(313)
        Yaw_Rate_Interp, = plt.plot(Time, Yaw_Rate_Interpolated, '-o',
label='Yaw_Rate_Interp')
        Yaw_Rate_model, = plt.plot(Time, Yaw_Rate_Model, '-+',
label='Yaw_Rate_Model')
        plt.ylabel('Yaw Rate (millirad/sec) ')
        plt.xlabel('Time (s)')
        plt.legend(handles=[Yaw_Rate_Interp, Yaw_Rate_model],loc='upper
left')

        plt.gcf().set_size_inches(11,8.5)
        plt.savefig('PLOTS/MIMO/MODEL/MIMO_Model_Fitted_Input_Output_' +
str(File_Name)) #Saveing mesured and modeled responses for model data set


        #Check time interval between messages
        plt.figure(4)
        #Time_IMU
        plt.subplot(311)
        plt.plot(SDelta_Time_IMU, '+')
        plt.ylabel('Time_IMU_DT (sec) ')
        #plt.xlabel('Number of intervals')
        #Time_RC
        plt.subplot(312)
        plt.plot(SDelta_Time_RC, '+')
        plt.ylabel('Time_RC_DT (sec) ')
        #plt.xlabel('Number of intervals')
        #Time_VFR
        plt.subplot(313)
        plt.plot(SDelta_Time_VFR, '+')
        plt.ylabel('Time_VFR_DT (sec) ')
        plt.xlabel('Number of intervals')

        plt.gcf().set_size_inches(11,8.5)
        plt.savefig('PLOTS/MIMO/MODEL/MIMO_Model_Delta_Time_' +
str(File_Name)) #Saveing change of time for diffrent mavlink messaages for
```

```
model data set

        plt.close(1)
        plt.close(2)
        plt.close(3)
        plt.close(4)
        print "Data processing for model building compleat!"
        #plt.show()



    if MIMO_DATA_SET != 0:
        MIMO_ID_Val = SIM_OUTPUT(MIMO_ID_SYS,U_Center,Time)

        Roll_Rate_Val = MIMO_ID_Val[:,0]  #Simulated Roll Rate (millirad/sec)
with calidation data
        Pitch_Rate_Val = MIMO_ID_Val[:,1] #Simulated Roll Rate (millirad/sec)
with validation data
        Yaw_Rate_Val = MIMO_ID_Val[:,2]   #Simulated Roll Rate (millirad/sec)
with validation data

        TIC_Roll_Val = TIC(Y[0,:], MIMO_ID_Val[:,0])  #Estimates how well the
model fits the mesured for roll rate with validation data
        TIC_Pitch_Val = TIC(Y[1,:], MIMO_ID_Val[:,1]) #Estimates how well the
model fits the mesured for pitch rate with validation data
        TIC_Yaw_Val = TIC(Y[2,:], MIMO_ID_Val[:,2])   #Estimates how well the
model fits the mesured for yaw rate with validation data
        print("TIC for Roll_Validation is: " + str(TIC_Roll_Val))
        print("TIC for Pitch_Validation is: " + str(TIC_Pitch_Val))
        print("TIC for Yaw_Validation is: " + str(TIC_Yaw_Val))

        #Saveing the simulated responses for validation as well as the TIC
values
        Store_Validation_SIM_Data(Roll_Rate_Val, Pitch_Rate_Val,
Yaw_Rate_Val, TIC_Roll_Val, TIC_Pitch_Val, TIC_Yaw_Val)

        if TIC_Roll_Val < 0.16:
            print " Roll Okay!!!!"
        elif TIC_Roll_Val > 0.18:
            print "Problem with Roll Axis"


        ##Check of interpolation for input validation data set
        plt.figure(1)
        plt.subplot(511)
        Ch_1, = plt.plot(Channel_Out_Time, Channel_1_OUT,'+', label='Ch_1')
        Ch_1_interp, = plt.plot(Time,
Channel_1_OUT_interpolated,'^',markerfacecolor='None', label='Ch_1_interp')
        plt.ylabel('Aile/Ch_1 (PWM)')
        plt.legend(handles=[Ch_1, Ch_1_interp],loc='upper right')
        plt.title('Recorded vs Interpolated RC Input of Model Data Set')

        plt.subplot(512)
        Ch_2, = plt.plot(Channel_Out_Time, Channel_2_OUT,'+', label='Ch_2')
        Ch_2_interp, = plt.plot(Time,
Channel_2_OUT_interpolated,'^',markerfacecolor='None', label='Ch_2_interp')
```

```
        plt.ylabel('Ele/Ch_2 (PWM)')
        plt.legend(handles=[Ch_2, Ch_2_interp],loc='upper right')

        plt.subplot(513)
        Ch_3, = plt.plot(Channel_Out_Time, Channel_3_OUT,'+', label='Ch_3')
        Ch_3_interp, = plt.plot(Time,
Channel_3_OUT_interpolated,'^',markerfacecolor='None', label='Ch_3_interp')
        plt.ylabel('Thr/Ch_3 (PWM)')
        plt.legend(handles=[Ch_1, Ch_1_interp],loc='upper right')

        plt.subplot(514)
        Ch_4, = plt.plot(Channel_Out_Time, Channel_4_OUT,'+',label='Ch_4')
        Ch_4_interp, = plt.plot(Time,
Channel_4_OUT_interpolated,'^',markerfacecolor='None', label='Ch_4_interp')
        plt.ylabel('Rud/Ch_4 (PWM)')
        plt.legend(handles=[Ch_4, Ch_4_interp],loc='upper right')

        plt.subplot(515)
        Air_Speed, = plt.plot(Time_VFR, Air_speed_VFR,'+',
label='Air_Speed_VFR')
        Air_Speed_interp, = plt.plot(Time,
Air_speed_VFR_interpolated,'^',markerfacecolor='None',label='Air_Speed_VFR_in
terp')
        plt.ylabel('Air_Speed (m/s)')
        plt.xlabel('Time(sec)')
        plt.legend(handles=[Air_Speed, Air_Speed_interp],loc='upper right')


        plt.gcf().set_size_inches(11,8.5)
        plt.savefig('PLOTS/MIMO/VALIDATION/MIMO_Val_Input_' + str(File_Name))
#Saveing Input data set plots used for model validation


        plt.figure(2)
        plt.subplot(311)
        Roll_rate, = plt.plot(Time_IMU, Roll_Rate,'+', label='Roll_Rate')
        Roll_rate_interp, = plt.plot(Time,
Roll_Rate_Interpolated,'^',markerfacecolor='None', label='Roll_Rate_Interp')
        plt.ylabel('Roll_Rate (millirad/sec)')
        plt.title('Recorded vs Interpolated Response of Model Data Set')
        plt.legend(handles=[Roll_rate, Roll_rate_interp],loc='upper right')

        plt.subplot(312)
        Pitch_rate, = plt.plot(Time_IMU, Pitch_Rate,'+', label='Pitch_Rate')
        Pitch_rate_interp, = plt.plot(Time,
Pitch_Rate_Interpolated,'^',markerfacecolor='None',label='Pitch_Rate_Interp')
        plt.ylabel('Pitch_Rate (millirad/sec)')
        plt.legend(handles=[Pitch_rate, Pitch_rate_interp],loc='upper right')

        plt.subplot(313)
        Yaw_rate, = plt.plot(Time_IMU, Yaw_Rate,'+', label='Yaw_Rate')
        Yaw_rate_interp, = plt.plot(Time,
Yaw_Rate_Interpolated,'^',markerfacecolor='None', label='Yaw_Rate_Interp')
        plt.ylabel('Yaw_Rate (millirad/sec)')
        plt.xlabel('Time(sec)')
        plt.legend(handles=[Yaw_rate, Yaw_rate_interp],loc='upper right')
        plt.gcf().set_size_inches(11,8.5)
        plt.savefig('PLOTS/MIMO/VALIDATION/MIMO_Val_Output_' + 
str(File_Name)) #Saveing Output data set plots used for model validation

        plt.figure(3)
        #Mesured Roll and Model roll
        plt.subplot(311)
        Roll_Rate_Interp, = plt.plot(Time, Roll_Rate_Interpolated,'-o', 
label='Roll_Rate_Interp')
        Roll_Rate_val, = plt.plot(Time, Roll_Rate_Val,'-+', 
label='Roll_Rate_Val')
        plt.ylabel('Roll_Rate (millirad/sec) ')
        plt.title('Measured vs Modeled Angular Rates With Validation Data 
Set')
        plt.legend(handles=[Roll_Rate_Interp, Roll_Rate_val],loc='upper 
right')
        # Measured Pitch and Model Pitch
        plt.subplot(312)
        Pitch_Rate_Interp, = plt.plot(Time, Pitch_Rate_Interpolated, '-o', 
label='Pitch_Rate_Interp')
        Pitch_Rate_val, = plt.plot(Time, Pitch_Rate_Val, '-+', 
label='Pitch_Rate_Val')
        plt.ylabel('Pitch Rate (millirad/sec)')
        plt.legend(handles=[Pitch_Rate_Interp, Pitch_Rate_val],loc='upper 
right')
        # Measured Yaw and Model Yaw
        plt.subplot(313)
        Yaw_Rate_Interp, = plt.plot(Time, Yaw_Rate_Interpolated, '-o', 
label='Yaw_Rate_Interp')
        Yaw_Rate_val, = plt.plot(Time, Yaw_Rate_Val, '-+', 
label='Yaw_Rate_Val')
        plt.ylabel('Yaw Rate (millirad/sec) ')
        plt.xlabel('Time (s)')
        plt.legend(handles=[Yaw_Rate_Interp, Yaw_Rate_val],loc='upper left')


        plt.gcf().set_size_inches(11,8.5)
        plt.savefig('PLOTS/MIMO/VALIDATION/MIMO_Val_Fitted_Input_Output_' + 
str(File_Name)) #Saveing mesured and modeled responses for validation data 
set


        #Check time interval between messages
        plt.figure(4)
        #Time_IMU
        plt.subplot(311)
        plt.plot(SDelta_Time_IMU, '+')
        plt.ylabel('Time_IMU_DT (sec) ')
        #plt.xlabel('Number of intervals')
        #Time_RC
        plt.subplot(312)
        plt.plot(SDelta_Time_RC, '+')
        plt.ylabel('Time_RC_DT (sec) ')
        #plt.xlabel('Number of intervals')
        #Time_VFR
        plt.subplot(313)
        plt.plot(SDelta_Time_VFR, '+')
        plt.ylabel('Time_VFR_DT (sec) ')
        plt.xlabel('Number of intervals')

        plt.gcf().set_size_inches(11,8.5)
        plt.savefig('PLOTS/MIMO/VALIDATION/MIMO_Val_Delta_Time_' +
str(File_Name)) #Saveing change of time for diffrent mavlink messaages for
model data set
        print "Data processing for validation compleat!"
        #plt.show()

    return

def Process_Data(Channel_1, Channel_2, Channel_3, Channel_4, Channel_5,
Channel_6, Channel_IN_Time,
                 Channel_1_OUT, Channel_2_OUT, Channel_3_OUT, Channel_4_OUT,
Channel_5_OUT, Channel_6_OUT, Channel_Out_Time,
                 Time_IMU, Roll_Rate, Pitch_Rate, Yaw_Rate, Time_VFR
,Air_speed_VFR):
    global ROLL_DATA_SET
    global PITCH_DATA_SET
    global YAW_DATA_SET
    global All_AXIS_DATA_SET

    #First check for correct DT value
    DT_avg_IMU, SDelta_Time_IMU = Delta_Time(Time_IMU)       #DT stats on IMU
mavlink messages
    DT_avg_RC, SDelta_Time_RC = Delta_Time(Channel_IN_Time) #DT stats on RC
Channel mavlink messages
    DT_avg_VFR, SDelta_Time_VFR = Delta_Time(Time_VFR)       #DT stats on
Time_VFR
    print "IMU_Message_Rate_Avg (Hz)", 1/DT_avg_IMU, "\tRC_Message_Rate_Avg
(Hz)", 1/DT_avg_RC , "\tVFR_Message_Rate_Avg (Hz)", 1/DT_avg_VFR



    if (DT_avg_IMU > 0.015) and (DT_avg_IMU < 0.07):
        DT = 0.02                              #Sets the time interval for all
samples to collected at
        print "50 Hz data"
    if (DT_avg_IMU >0.002) and (DT_avg_IMU < 0.01):
        DT = 0.005
        print "200 Hz data"

    Time = Master_Time(Time_IMU, DT)      #Creates Time vector based on length
of the test used as the baseline for all interpolation

##    if (any(Channel_1_OUT>1500)) and (any(Channel_1_OUT<1450)):
##        if (any(Channel_2_OUT>1700)) and (any(Channel_2_OUT<1535)):
##            if (any(Channel_4_OUT>1600)) and (any(Channel_4_OUT<1500)):
    print "Going into MIMO"
    Process_All_Axis_MIMO(Channel_1, Channel_2, Channel_3, Channel_4,
Channel_5, Channel_6, Channel_IN_Time,
                          Channel_1_OUT, Channel_2_OUT, Channel_3_OUT,
Channel_4_OUT, Channel_5_OUT, Channel_6_OUT, Channel_Out_Time,
                          Time_IMU, Roll_Rate, Pitch_Rate, Yaw_Rate, Time_VFR
,Air_speed_VFR, All_AXIS_DATA_SET, Time, DT, DT_avg_IMU, SDelta_Time_IMU,
DT_avg_RC, SDelta_Time_RC, DT_avg_VFR, SDelta_Time_VFR)
    All_AXIS_DATA_SET = All_AXIS_DATA_SET + 1
    return

def read_loop(m):
    stime_channel = np.array([])
    schannel_1 = np.array([])
    schannel_2 = np.array([])
    schannel_3 = np.array([])
    schannel_4 = np.array([])
    schannel_5 = np.array([])
    schannel_6 = np.array([])
    #########################
    stime_channel_out = np.array([])
    schannel_1_out = np.array([])
    schannel_2_out = np.array([])
    schannel_3_out = np.array([])
    schannel_4_out = np.array([])
    schannel_5_out = np.array([])
    schannel_6_out = np.array([])
    ##########################
    stime_imu= np.array([])
    sxgyro = np.array([])
    sygyro = np.array([])
    szgyro = np.array([])
    #########################
    svfr_time = np.array([])
    sair_speed = np.array([])


    while True:


        #print"Waiting to get data"
        channel_6 =1500
        msg = None
        while not msg:
            msg = m.recv_match()


        msg_type = msg.get_type()
        if msg_type == "BAD_DATA":
                if mavutil.all_printable(msg.data):
                        sys.stdout.write(msg.data)
                        sys.stdout.flush()
        elif msg_type == "RC_CHANNELS":
                channel_1, channel_2, channel_3, channel_4, channel_5,
channel_6, rc_in_time  = handle_rc_raw(msg)


        last_imu_time = None
        while channel_6 < 1450:

            if last_imu_time == None:
                print("Takeing Data")

            msg = None
            while not msg:
                msg = m.recv_match()


            # handle the message based on its type
            msg_type = msg.get_type()
            if msg_type == "BAD_DATA":
                    if mavutil.all_printable(msg.data):
                            sys.stdout.write(msg.data)
                            sys.stdout.flush()
            elif msg_type == "RC_CHANNELS":
                    channel_1, channel_2, channel_3, channel_4, channel_5,
channel_6, rc_in_time  = handle_rc_raw(msg)
                    stime_channel= np.append(stime_channel, rc_in_time)
                    schannel_1 = np.append(schannel_1, channel_1)
                    schannel_2 = np.append(schannel_2, channel_2)
                    schannel_3 = np.append(schannel_3, channel_3)
                    schannel_4 = np.append(schannel_4, channel_4)
                    schannel_5 = np.append(schannel_5, channel_5)
                    schannel_6 = np.append(schannel_6, channel_6)

            elif msg_type == "SERVO_OUTPUT_RAW":
                    channel_1_out, channel_2_out, channel_3_out,
channel_4_out, channel_5_out, channel_6_out, rc_out_time =
handle_rc_raw_out(msg)
                    stime_channel_out = np.append(stime_channel_out,
rc_out_time)
                    schannel_1_out = np.append(schannel_1_out, channel_1_out)
                    schannel_2_out = np.append(schannel_2_out, channel_2_out)
                    schannel_3_out = np.append(schannel_3_out, channel_3_out)
                    schannel_4_out = np.append(schannel_4_out, channel_4_out)
                    schannel_5_out = np.append(schannel_5_out, channel_5_out)
                    schannel_6_out = np.append(schannel_6_out, channel_6_out)

            #elif msg_type == "HEARTBEAT":
                    #handle_heartbeat(msg)

            elif msg_type == "RAW_IMU":
                    raw_imu_time,raw_imu_roll, raw_imu_pitch, raw_imu_yaw =
handle_raw_imu(msg)
                    stime_imu= np.append(stime_imu,raw_imu_time)
                    sxgyro = np.append(sxgyro, raw_imu_roll)
                    sygyro = np.append(sygyro, raw_imu_pitch)
                    szgyro = np.append(szgyro, raw_imu_yaw)
                    last_imu_time = raw_imu_time


            elif msg_type =="VFR_HUD":
                    air_speed = handle_VFR_HUD(msg)
                    #if last_imu_time is not None:
                    if (last_imu_time != None) and (air_speed > 1):
#This eliminates zeros airspeed values!!!!!!!!!!!!!!!! need to check!!!!!
                        svfr_time = np.append(svfr_time, last_imu_time)
                        sair_speed = np.append(sair_speed, air_speed)

            elif msg_type == "ATTITUDE":
                    handle_attitude(msg)
```

```
        if (schannel_1.size > 1) and (channel_6 > 1600):
                print("Entering Data Processing....")
                Process_Data(schannel_1, schannel_2, schannel_3, schannel_4,
schannel_5, schannel_6, stime_channel, schannel_1_out, schannel_2_out,
schannel_3_out, schannel_4_out, schannel_5_out, schannel_6_out,
stime_channel_out, stime_imu, sxgyro, sygyro, szgyro, svfr_time ,sair_speed)
                stime_channel = np.array([])
                schannel_1 = np.array([])
                schannel_2 = np.array([])
                schannel_3 = np.array([])
                schannel_4 = np.array([])
                schannel_5 = np.array([])
                schannel_6 = np.array([])
                #########################
                stime_channel_out = np.array([])
                schannel_1_out = np.array([])
                schannel_2_out = np.array([])
                schannel_3_out = np.array([])
                schannel_4_out = np.array([])
                schannel_5_out = np.array([])
                schannel_6_out = np.array([])
                ##########################
                stime_imu= np.array([])
                sxgyro = np.array([])
                sygyro = np.array([])
                szgyro = np.array([])
                #########################
                svfr_time = np.array([])
                sair_speed = np.array([])



def main():

    # read command-line options
    parser = OptionParser("readdata.py [options]")
    parser.add_option("--baudrate", dest="baudrate", type='int',
                                      help="master port baud rate",
default=921600)
    parser.add_option("--device", dest="device", default=
"/dev/ttyPIXHAWK_DATA", help="serial device")
    parser.add_option("--rate", dest="rate", default=50, type='int',
help="requested stream rate")
    parser.add_option("--source-system", dest='SOURCE_SYSTEM', type='int',
                                      default=255, help='MAVLink source
system for this GCS')
    parser.add_option("--showmessages", dest="showmessages",
action='store_true',
                                      help="show incoming messages",
default=False)



    (opts, args) = parser.parse_args()
    if opts.device is None:
            print("You must specify a serial device")
            sys.exit(1)

    # create a mavlink serial instance
    master = mavutil.mavlink_connection(opts.device, baud=opts.baudrate)

    # wait for the heartbeat msg to find the system ID
    master.wait_heartbeat()

    # request data to be sent at the given rate for IMU
    master.mav.request_data_stream_send(master.target_system,
master.target_component,
            mavutil.mavlink.MAV_DATA_STREAM_RAW_SENSORS, 50, 1)
    # request data to be sent at the given rate for EXTENDED STATUS
    master.mav.request_data_stream_send(master.target_system,
master.target_component,
            mavutil.mavlink.MAV_DATA_STREAM_EXTENDED_STATUS, 25, 0) #Not used
    # request data to be sent at the given rate for RC
    master.mav.request_data_stream_send(master.target_system,
master.target_component,
            mavutil.mavlink.MAV_DATA_STREAM_RC_CHANNELS, 25, 1)
    # request data to be sent at the given rate for RAW_CONTROLLER
    master.mav.request_data_stream_send(master.target_system,
master.target_component,
            mavutil.mavlink.MAV_DATA_STREAM_RAW_CONTROLLER, 0, 0) #Not used
    # request data to be sent at the given rate for the position
    master.mav.request_data_stream_send(master.target_system,
master.target_component,
            mavutil.mavlink.MAV_DATA_STREAM_POSITION, 0, 0)#Not used
    # request data to be sent at the given rate EXTRA 1
    master.mav.request_data_stream_send(master.target_system,
master.target_component,
            mavutil.mavlink.MAV_DATA_STREAM_EXTRA1, 5, 0) #Not used
    # request data to be sent at the given rate EXTRA 2 (VFR)
    master.mav.request_data_stream_send(master.target_system,
master.target_component,
            mavutil.mavlink.MAV_DATA_STREAM_EXTRA2, 25, 1)
    # request data to be sent at the given rate EXTRA 3
    master.mav.request_data_stream_send(master.target_system,
master.target_component,
            mavutil.mavlink.MAV_DATA_STREAM_EXTRA3, 0, 0) #Not used


    print "Connected going to data collection"


    read_loop(master)


if __name__ == '__main__':
   main()
```

APPENDIX C

```
#!/usr/bin/env python
"""
Servo_Failer
Code was written by Brian Duvall March 2020
Danger this code changes SERVOX_FUNCTION params when run!!!
Danger this code changes SERVOX_MAX and MIN endpoints!!!
DO NOT USE WITHOUT MUX BOARD!!!
Gets servo trim PWM value
Sets desired servo to its trim condition to simulate a fail-safe
import code
code.interact(local=locals())
"""

import sys, os
from optparse import OptionParser
import time
from pymavlink import mavutil
import math



def Set_RC_Channel_PWM(master, id, pwm=1500):
    """ Set RC channel PWM value
    Args:
        id (TYPE): Channel ID
        pwm (int, optional): Channel pwm value 1100-1900
    """
    if id < 1:
        print("Channel does not exist.")
        return

    # We only have 8 channels
    # https://mavlink.io/en/messages/common.html#RC_CHANNELS_OVERRIDE
    if id < 9:
        rc_channel_values = [65535 for _ in range(8)]
        rc_channel_values[id - 1] = pwm
        master.mav.rc_channels_override_send(
            master.target_system,                   # target_system
            master.target_component,                # target_component
            *rc_channel_values)                     # RC channel list, in
microseconds.
    return


def Channel_Overide(master, ch1, ch2, ch3, ch4, ch5, ch6, ch7, ch8):
    msg = master.mav.rc_channels_override_send(
    0,#master.target_system,
    0,#master.target_component,
    ch1,
    ch2,
    ch3,
    ch4,
    ch5,
    ch6,
    ch7,
    ch8)
    master.mav.send(msg)
    print ("Sent message")
    return

def Read_Param_Value(master,param):
    while True:
        master.mav.param_request_read_send(master.target_system,
master.target_component,param,-1)
        message = master.recv_match(type='PARAM_VALUE',
blocking=True).to_dict()
        time.sleep(0.02)
        if param == message['param_id']:
            #print('name: %s\tvalue: %d' % (message['param_id'].decode("utf-
8"), message['param_value']))
            return message['param_value']

def Set_Param(master, param, param_value):
    master.mav.param_set_send(
    master.target_system, master.target_component,
    param,
    param_value,
    mavutil.mavlink.MAV_PARAM_TYPE_REAL32
    )
def Set_Servo(master, servo_number, pwm_value):
    msg = master.mav.command_long_encode(
    master.target_system,
    master.target_component,
    mavutil.mavlink.MAV_CMD_DO_SET_SERVO,
    0,
    servo_number,
    pwm_value,
    0, 0, 0, 0, 0)

    master.mav.send(msg)
    return


def read_loop(m):
    Ch1=1
    Ch2=2
    Ch3=3
    Ch4=4
    Ch5=5
    Ch6=6

    Ch1_Trim = Read_Param_Value(m,'SERVO1_TRIM') #Right_Aileron
    CH1_Min_Orig = Read_Param_Value(m,'SERVO1_MIN')
    CH1_Max_Orig = Read_Param_Value(m,'SERVO1_MAX')
    ######################################################
    Ch5_Trim = Read_Param_Value(m,'SERVO5_TRIM') #Left_Aileron
    CH5_Min_Orig = Read_Param_Value(m,'SERVO5_MIN')
    CH5_Max_Orig = Read_Param_Value(m,'SERVO5_MAX')
    ######################################################
    Ch2_Trim = Read_Param_Value(m,'SERVO2_TRIM') #Right_Elevator
    CH2_Min_Orig = Read_Param_Value(m,'SERVO2_MIN')
    CH2_Max_Orig = Read_Param_Value(m,'SERVO2_MAX')
    ######################################################
    Ch6_Trim = Read_Param_Value(m,'SERVO6_TRIM') #Left_Elevator
    CH6_Min_Orig = Read_Param_Value(m,'SERVO6_MIN')
    CH6_Max_Orig = Read_Param_Value(m,'SERVO6_MAX')
    ######################################################
    Ch7_Trim = Read_Param_Value(m,'SERVO7_TRIM') #Rudder
    CH7_Min_Orig = Read_Param_Value(m, 'SERVO7_MIN')    #Rudder Min
    CH7_Max_Orig = Read_Param_Value(m, 'SERVO7_MAX')    #Rudder Max

    #Failure combo
    #C1 = F_AL5_ELE6
    #C2 = F_AL5_L_ELE6
    #C3 = L_AL5_F_ELE6
    #C4 = L_AL5_ELE6
    while (True):
        #print"C1 = F_AL5_ELE6, C2 = F_AL5_L_ELE6, C3 = L_AL5_F_ELE6, C4 = 
L_AL5_ELE6"
        print"C5 = L_ELE6_L_RUDD7, C6 = F_ELE6_L_RUDD7, C7 = L_AL5_L_RUDD7, 
C8 = F_AL5_L_RUDD7"
        failure_mode = raw_input("Enter a failure mode, LIM_AL5, AL5, 
LIM_ELE6, ELE6, RUDD7, C1, C2, C3, C4, C5, C6, C7, C8:")
        duration = input("Enter the time duration of failure:")

        if failure_mode == 'LIM_AL1':
            CH1_Min_New = int(Ch1_Trim - abs((CH1_Min_Orig - Ch1_Trim)/4)) 
#Get new Min PWM limit
            CH1_Max_New = int(Ch1_Trim + abs((CH1_Max_Orig - Ch1_Trim)/4)) 
#Get new Max PWM limit


            CH1_Min = Read_Param_Value(m, 'SERVO1_MIN')     #Aileron1 Min
            CH1_Max = Read_Param_Value(m, 'SERVO1_MAX')     #Aileron1 Max
            while (CH1_Min != CH1_Min_New) and (CH1_Max != CH1_Max_New):
                Set_Param(m,'SERVO1_MIN',CH1_Min_New)            #Setting the 
new Min PWM Limit
                time.sleep(0.02)
                Set_Param(m,'SERVO1_MAX',CH1_Max_New)            #Setting the 
new Max PWM Limit
                time.sleep(0.02)
                CH1_Min = Read_Param_Value(m, 'SERVO1_MIN')     #Aileron1 Min
                CH1_Max = Read_Param_Value(m, 'SERVO1_MAX')     #Aileron1 Max
            print"AL1 Limited!"

            time.sleep(duration)

            CH1_Min = Read_Param_Value(m, 'SERVO1_MIN')     #Aileron1 Min
            CH1_Max = Read_Param_Value(m, 'SERVO1_MAX')     #Aileron1 Max
            while (CH1_Min != CH1_Min_Orig) and (CH1_Max != CH1_Max_Orig):
                Set_Param(m,'SERVO1_MIN',CH1_Min_Orig)            #Setting 
back to original Min PWM Limit
                time.sleep(0.02)
                Set_Param(m,'SERVO1_MAX',CH1_Max_Orig)            #Setting 
back to the original Max PWM Limit
                time.sleep(0.02)
                CH1_Min = Read_Param_Value(m, 'SERVO1_MIN')    #Aileron1 Min
                CH1_Max = Read_Param_Value(m, 'SERVO1_MAX')    #Aileron Max
            print"AL1 Limit Removed!"

        if failure_mode == 'LIM_AL5':
            CH5_Min_New = int(Ch5_Trim - abs((CH5_Min_Orig - Ch5_Trim)/4))
#Get new Min PWM limit
            CH5_Max_New = int(Ch5_Trim + abs((CH5_Max_Orig - Ch5_Trim)/4))
#Get new Max PWM limit


            CH5_Min = Read_Param_Value(m, 'SERVO5_MIN')     #Aileron5 Min
            CH5_Max = Read_Param_Value(m, 'SERVO5_MAX')     #Aileron5 Max
            while (CH5_Min != CH5_Min_New) and (CH5_Max != CH5_Max_New):
                Set_Param(m,'SERVO5_MIN',CH5_Min_New)           #Setting the
new Min PWM Limit
                time.sleep(0.02)
                Set_Param(m,'SERVO5_MAX',CH5_Max_New)           #Setting the
new Max PWM Limit
                time.sleep(0.02)
                CH5_Min = Read_Param_Value(m, 'SERVO5_MIN')     #Aileron5 Min
                CH5_Max = Read_Param_Value(m, 'SERVO5_MAX')     #Aileron5 Max

            print"AL5 Limited!"

            time.sleep(duration)

            CH5_Min = Read_Param_Value(m, 'SERVO5_MIN')     #Aileron5 Min
            CH5_Max = Read_Param_Value(m, 'SERVO5_MAX')     #Aileron5 Max
            while (CH5_Min != CH5_Min_Orig) and (CH5_Max != CH5_Max_Orig):
                Set_Param(m,'SERVO5_MIN',CH5_Min_Orig)           #Setting
back to original Min PWM Limit
                time.sleep(0.02)
                Set_Param(m,'SERVO5_MAX',CH5_Max_Orig)           #Setting
back to the original Max PWM Limit
                time.sleep(0.1)
                CH5_Min = Read_Param_Value(m, 'SERVO5_MIN')    #Aileron5 Min
                CH5_Max = Read_Param_Value(m, 'SERVO5_MAX')    #Aileron5 Max
            print"AL5 Limit Removed!"

        if failure_mode == 'LIM_AL':
            CH1_Min_New = int(Ch1_Trim - abs((CH1_Min_Orig - Ch1_Trim)/4))
#Get new Min PWM limit
            CH1_Max_New = int(Ch1_Trim + abs((CH1_Max_Orig - Ch1_Trim)/4))
#Get new Max PWM limit
            CH5_Min_New = int(Ch5_Trim - abs((CH5_Min_Orig - Ch5_Trim)/4))
#Get new Min PWM limit
            CH5_Max_New = int(Ch5_Trim + abs((CH5_Max_Orig - Ch5_Trim)/4))
#Get new Max PWM limit


            CH1_Min = Read_Param_Value(m, 'SERVO1_MIN')     #Aileron1 Min
            CH1_Max = Read_Param_Value(m, 'SERVO1_MAX')     #Aileron1 Max
            CH5_Min = Read_Param_Value(m, 'SERVO5_MIN')     #Aileron5 Min
            CH5_Max = Read_Param_Value(m, 'SERVO5_MAX')     #Aileron5 Max
            while (CH1_Min != CH1_Min_New) and (CH1_Max != CH1_Max_New) and
(CH5_Min != CH5_Min_New) and (CH5_Max != CH5_Max_New):
                Set_Param(m,'SERVO1_MIN',CH1_Min_New)           #Setting the
new Min PWM Limit
```

```
                time.sleep(0.02)
                Set_Param(m,'SERVO1_MAX',CH1_Max_New)           #Setting the
new Max PWM Limit
                time.sleep(0.02)
                Set_Param(m,'SERVO5_MIN',CH5_Min_New)           #Setting the
new Min PWM Limit
                time.sleep(0.02)
                Set_Param(m,'SERVO5_MAX',CH5_Max_New)           #Setting the
new Max PWM Limit
                time.sleep(0.02)

                CH1_Min = Read_Param_Value(m, 'SERVO1_MIN')     #Aileron1 Min
                CH1_Max = Read_Param_Value(m, 'SERVO1_MAX')     #Aileron1 Max
                CH5_Min = Read_Param_Value(m, 'SERVO5_MIN')     #Aileron5 Min
                CH5_Max = Read_Param_Value(m, 'SERVO5_MAX')     #Aileron5 Max
            print"AL Limited!"

            time.sleep(duration)

            CH1_Min = Read_Param_Value(m, 'SERVO1_MIN')     #Aileron1 Min
            CH1_Max = Read_Param_Value(m, 'SERVO1_MAX')     #Aileron1 Max
            CH5_Min = Read_Param_Value(m, 'SERVO5_MIN')     #Aileron5 Min
            CH5_Max = Read_Param_Value(m, 'SERVO5_MAX')     #Aileron5 Max
            while (CH1_Min != CH1_Min_Orig) and (CH1_Max != CH1_Max_Orig) and
(CH5_Min != CH5_Min_Orig) and (CH5_Max != CH5_Max_Orig):
                Set_Param(m,'SERVO1_MIN',CH1_Min_Orig)           #Setting
back to original Min PWM Limit
                time.sleep(0.02)
                Set_Param(m,'SERVO1_MAX',CH1_Max_Orig)           #Setting
back to the original Max PWM Limit
                time.sleep(0.02)
                Set_Param(m,'SERVO5_MIN',CH5_Min_Orig)           #Setting
back to original Min PWM Limit
                time.sleep(0.02)
                Set_Param(m,'SERVO5_MAX',CH5_Max_Orig)           #Setting
back to the original Max PWM Limit
                time.sleep(0.02)

                CH1_Min = Read_Param_Value(m, 'SERVO1_MIN')    #Aileron1 Min
                CH1_Max = Read_Param_Value(m, 'SERVO1_MAX')    #Aileron1 Max
                CH5_Min = Read_Param_Value(m, 'SERVO5_MIN')    #Aileron5 Min
                CH5_Max = Read_Param_Value(m, 'SERVO5_MAX')    #Aileron5 Max
            print"AL Limit Removed!"

        if failure_mode == 'AL5':
            ServoFunctionValue= Read_Param_Value(m,'SERVO5_FUNCTION')#Verifiy
servo function is disabled
            while ServoFunctionValue != 0:
                Set_Param(m, 'SERVO5_FUNCTION', 0) #Disables aileron
                ServoFunctionValue=
Read_Param_Value(m,'SERVO5_FUNCTION')#Verifiy servo function is disabled
                time.sleep(0.02)
            print"AL5 FAILED!"
            start_time = time.time()
            end_time = start_time + duration
            while time.time() < end_time:
                Set_Servo(m,5,Ch5_Trim)#Do set servo command
                time.sleep(0.02)

            ServoFunctionValue= Read_Param_Value(m,'SERVO5_FUNCTION')#Verifiy
servo function is disabled
            while ServoFunctionValue != 4:
                Set_Param(m, 'SERVO5_FUNCTION', 4) #Enalbles aileron
                ServoFunctionValue=
Read_Param_Value(m,'SERVO5_FUNCTION')#Verifiy servo function is disabled
            print"AL5 Restored!"


        if failure_mode == 'LIM_ELE2':
            CH2_Min_New = int(Ch2_Trim - abs((CH2_Min_Orig - Ch2_Trim)/4))
#Get new Min PWM limit
            CH2_Max_New = int(Ch2_Trim + abs((CH2_Max_Orig - Ch2_Trim)/4))
#Get new Max PWM limit


            CH2_Min = Read_Param_Value(m, 'SERVO2_MIN')    #Elevator2 Min
            CH2_Max = Read_Param_Value(m, 'SERVO2_MAX')    #Elevator2 Max
            while (CH2_Min != CH2_Min_New) and (CH2_Max != CH2_Max_New):
                Set_Param(m,'SERVO2_MIN',CH2_Min_New)          #Setting the
new Min PWM Limit
                time.sleep(0.02)
                Set_Param(m,'SERVO2_MAX',CH2_Max_New)          #Setting the
new Max PWM Limit
                time.sleep(0.02)
                CH2_Min = Read_Param_Value(m, 'SERVO2_MIN')    #Elevator2 Min
                CH2_Max = Read_Param_Value(m, 'SERVO2_MAX')    #Elevator2 Max
            print"ELE2 Limited!"

            time.sleep(duration)

            CH2_Min = Read_Param_Value(m, 'SERVO2_MIN')    #Elevator2 Min
            CH2_Max = Read_Param_Value(m, 'SERVO2_MAX')    #Elevator2 Max
            while (CH2_Min != CH2_Min_Orig) and (CH2_Max != CH2_Max_Orig):
                Set_Param(m,'SERVO2_MIN',CH2_Min_Orig)          #Setting
back to original Min PWM Limit
                time.sleep(0.02)
                Set_Param(m,'SERVO2_MAX',CH2_Max_Orig)          #Setting
back to the original Max PWM Limit
                time.sleep(0.02)
                CH2_Min = Read_Param_Value(m, 'SERVO2_MIN')   #Elevator2 Min
                CH2_Max = Read_Param_Value(m, 'SERVO2_MAX')   #Elevator2 Max
            print"ELE2 Limit Removed!"

        if failure_mode == 'LIM_ELE6':
            CH6_Min_New = int(Ch6_Trim - abs((CH6_Min_Orig - Ch6_Trim)/4))
#Get new Min PWM limit
            CH6_Max_New = int(Ch6_Trim + abs((CH6_Max_Orig - Ch6_Trim)/4))
#Get new Max PWM limit


            CH6_Min = Read_Param_Value(m, 'SERVO6_MIN')    #Elevator6 Min
            CH6_Max = Read_Param_Value(m, 'SERVO6_MAX')    #Elevator6 Max
            while (CH6_Min != CH6_Min_New) and (CH6_Max != CH6_Max_New):
                Set_Param(m,'SERVO6_MIN',CH6_Min_New)          #Setting the
```

```
new Min PWM Limit
                time.sleep(0.02)
                Set_Param(m,'SERVO6_MAX',CH6_Max_New)           #Setting the
new Max PWM Limit
                time.sleep(0.02)
                CH6_Min = Read_Param_Value(m, 'SERVO6_MIN')     #Elevator6 Min
                CH6_Max = Read_Param_Value(m, 'SERVO6_MAX')     #Elevator6 Max
            print"ELE6 Limited!"

            time.sleep(duration)

            CH6_Min = Read_Param_Value(m, 'SERVO6_MIN')     #Elevator6 Min
            CH6_Max = Read_Param_Value(m, 'SERVO6_MAX')     #Elevator6 Max
            while (CH6_Min != CH6_Min_Orig) and (CH6_Max != CH6_Max_Orig):
                Set_Param(m,'SERVO6_MIN',CH6_Min_Orig)           #Setting
back to original Min PWM Limit
                time.sleep(0.02)
                Set_Param(m,'SERVO6_MAX',CH6_Max_Orig)           #Setting
back to the original Max PWM Limit
                time.sleep(0.02)
                CH6_Min = Read_Param_Value(m, 'SERVO6_MIN')    #Elevator6 Min
                CH6_Max = Read_Param_Value(m, 'SERVO6_MAX')    #Elevator6 Max
            print"ELE6 Limit Removed!"

        if failure_mode == 'LIM_ELE':
            CH2_Min_New = int(Ch2_Trim - abs((CH2_Min_Orig - Ch2_Trim)/4))
#Get new Min PWM limit
            CH2_Max_New = int(Ch2_Trim + abs((CH2_Max_Orig - Ch2_Trim)/4))
#Get new Max PWM limit
            CH6_Min_New = int(Ch6_Trim - abs((CH6_Min_Orig - Ch6_Trim)/4))
#Get new Min PWM limit
            CH6_Max_New = int(Ch6_Trim + abs((CH6_Max_Orig - Ch6_Trim)/4))
#Get new Max PWM limit

            CH2_Min = Read_Param_Value(m, 'SERVO2_MIN')     #Elevator2 Min
            CH2_Max = Read_Param_Value(m, 'SERVO2_MAX')     #Elevator2 Max
            CH6_Min = Read_Param_Value(m, 'SERVO6_MIN')     #Elevator6 Min
            CH6_Max = Read_Param_Value(m, 'SERVO6_MAX')     #Elevator6 Max
            while (CH2_Min != CH2_Min_New) and (CH2_Max != CH2_Max_New) and
(CH6_Min != CH6_Min_New) and (CH6_Max != CH6_Max_New):
                Set_Param(m,'SERVO2_MIN',CH2_Min_New)           #Setting the
new Min PWM Limit
                time.sleep(0.02)
                Set_Param(m,'SERVO2_MAX',CH2_Max_New)           #Setting the
new Max PWM Limit
                time.sleep(0.02)
                Set_Param(m,'SERVO6_MIN',CH6_Min_New)           #Setting the
new Min PWM Limit
                time.sleep(0.02)
                Set_Param(m,'SERVO6_MAX',CH6_Max_New)           #Setting the
new Max PWM Limit
                time.sleep(0.02)

                CH2_Min = Read_Param_Value(m, 'SERVO2_MIN')     #Elevator2 Min
                CH2_Max = Read_Param_Value(m, 'SERVO2_MAX')     #Elevator2 Max
                CH6_Min = Read_Param_Value(m, 'SERVO6_MIN')     #Elevator6 Min
                CH6_Max = Read_Param_Value(m, 'SERVO6_MAX')     #Elevator6 Max
            print"ELE Limited!"

            time.sleep(duration)

            CH2_Min = Read_Param_Value(m, 'SERVO2_MIN')     #Elevator2 Min
            CH2_Max = Read_Param_Value(m, 'SERVO2_MAX')     #Elevator2 Max
            CH6_Min = Read_Param_Value(m, 'SERVO6_MIN')     #Elevator6 Min
            CH6_Max = Read_Param_Value(m, 'SERVO6_MAX')     #Elevator6 Max
            while (CH2_Min != CH2_Min_Orig) and (CH2_Max != CH2_Max_Orig) and
(CH6_Min != CH6_Min_Orig) and (CH6_Max != CH6_Max_Orig):
                Set_Param(m,'SERVO2_MIN',CH2_Min_Orig)           #Setting
back to original Min PWM Limit
                time.sleep(0.02)
                Set_Param(m,'SERVO2_MAX',CH2_Max_Orig)           #Setting
back to the original Max PWM Limit
                time.sleep(0.02)
                Set_Param(m,'SERVO6_MIN',CH6_Min_Orig)           #Setting
back to original Min PWM Limit
                time.sleep(0.02)
                Set_Param(m,'SERVO6_MAX',CH6_Max_Orig)           #Setting
back to the original Max PWM Limit
                time.sleep(0.02)

                CH2_Min = Read_Param_Value(m, 'SERVO2_MIN')    #Elevator2 Min
                CH2_Max = Read_Param_Value(m, 'SERVO2_MAX')    #Elevator2 Max
                CH6_Min = Read_Param_Value(m, 'SERVO6_MIN')    #Elevator6 Min
                CH6_Max = Read_Param_Value(m, 'SERVO6_MAX')    #Elevator6 Max
            print"ELE Limit Removed!"


        if failure_mode == 'ELE6':
            ServoFunctionValue= Read_Param_Value(m,'SERVO6_FUNCTION')#Verifiy
servo function is disabled
            while ServoFunctionValue != 0:
                Set_Param(m, 'SERVO6_FUNCTION', 0) #Disables aileron
                ServoFunctionValue=
Read_Param_Value(m,'SERVO6_FUNCTION')#Verifiy servo function is disabled
                time.sleep(0.02)
            print"ELE6 FAILED!"
            start_time = time.time()
            end_time = start_time + duration
            while time.time() < end_time:
                Set_Servo(m,6,Ch6_Trim)#Do set servo command
                time.sleep(0.02)

            ServoFunctionValue= Read_Param_Value(m,'SERVO6_FUNCTION')#Verifiy
servo function is disabled
            while ServoFunctionValue != 19:
                Set_Param(m, 'SERVO6_FUNCTION', 19) #Enalbles Elevator
                ServoFunctionValue=
Read_Param_Value(m,'SERVO6_FUNCTION')#Verifiy servo function is disabled
            print"ELE6 Restored!"


        if failure_mode == 'RUDD7':
            CH7_Min_New = int(Ch7_Trim - abs((CH7_Min_Orig - Ch7_Trim)/2))
#Get new Min PWM limit
```

```
            CH7_Max_New = int(Ch7_Trim + abs((CH7_Max_Orig - Ch7_Trim)/2))
#Get new Max PWM limit


            CH7_Min = Read_Param_Value(m, 'SERVO7_MIN')     #Rudder Min
            CH7_Max = Read_Param_Value(m, 'SERVO7_MAX')     #Rudder Max
            while (CH7_Min != CH7_Min_New) and (CH7_Max != CH7_Max_New):
                Set_Param(m,'SERVO7_MIN',CH7_Min_New)           #Setting the
new Min PWM Limit
                time.sleep(0.02)
                Set_Param(m,'SERVO7_MAX',CH7_Max_New)           #Setting the
new Max PWM Limit
                time.sleep(0.02)
                CH7_Min = Read_Param_Value(m, 'SERVO7_MIN')     #Rudder Min
                CH7_Max = Read_Param_Value(m, 'SERVO7_MAX')     #Rudder Max
            print"Rudder Limited!"

            time.sleep(duration)

            CH7_Min = Read_Param_Value(m, 'SERVO7_MIN')     #Rudder Min
            CH7_Max = Read_Param_Value(m, 'SERVO7_MAX')     #Rudder Max
            while (CH7_Min != CH7_Min_Orig) and (CH7_Max != CH7_Max_Orig):
                Set_Param(m,'SERVO7_MIN',CH7_Min_Orig)           #Setting
back to original Min PWM Limit
                time.sleep(0.02)
                Set_Param(m,'SERVO7_MAX',CH7_Max_Orig)           #Setting
back to the original Max PWM Limit
                time.sleep(0.02)
                CH7_Min = Read_Param_Value(m, 'SERVO7_MIN')     #Rudder Min
                CH7_Max = Read_Param_Value(m, 'SERVO7_MAX')     #Rudder Max
            print"Rudder Limit Removed!"
#############################################################################
###############################
        #Combination failure modes: F_AL5_ELE6, F_AL5_L_ELE6, L_AL5_F_ELE6,
L_AL5_ELE6, L_ELE6_L_RUDD, F_ELE6_L_RUDD
        if failure_mode == 'C1':
            ServoFunctionValue5=
Read_Param_Value(m,'SERVO5_FUNCTION')#Verifiy servo function is disabled
            ServoFunctionValue6=
Read_Param_Value(m,'SERVO6_FUNCTION')#Verifiy servo function is disabled
            while (ServoFunctionValue5 != 0) and (ServoFunctionValue6 != 0):
                Set_Param(m, 'SERVO5_FUNCTION', 0) #Disables aileron
                Set_Param(m, 'SERVO6_FUNCTION', 0) #Disables elevator
                time.sleep(0.02)
                ServoFunctionValue5 =
Read_Param_Value(m,'SERVO5_FUNCTION')#Verifiy servo function is disabled
                ServoFunctionValue6=
Read_Param_Value(m,'SERVO6_FUNCTION')#Verifiy servo function is disabled

            print"AL5 and ELE6 FAILED!"
            start_time = time.time()
            end_time = start_time + duration
            while time.time() < end_time:
                Set_Servo(m,5,Ch5_Trim)#Do set servo command
                time.sleep(0.02)
                Set_Servo(m,6,Ch6_Trim)#Do set servo command
                time.sleep(0.02)
            ServoFunctionValue5=
Read_Param_Value(m,'SERVO5_FUNCTION')#Verifiy servo function is disabled
            ServoFunctionValue6=
Read_Param_Value(m,'SERVO6_FUNCTION')#Verifiy servo function is disabled
            while (ServoFunctionValue5 != 4) and (ServoFunctionValue6 != 19):
                Set_Param(m, 'SERVO5_FUNCTION', 4) #Enalbles aileron
                Set_Param(m, 'SERVO6_FUNCTION', 19) #Enalbles Elevator
                ServoFunctionValue5=
Read_Param_Value(m,'SERVO5_FUNCTION')#Verifiy servo function
                ServoFunctionValue6=
Read_Param_Value(m,'SERVO6_FUNCTION')#Verifiy servo function
            print"AL5 and ELE6 Restored!"
#############################################################################
################
        if failure_mode == 'C2':
            CH6_Min_New = int(Ch6_Trim - abs((CH6_Min_Orig - Ch6_Trim)/4))
#Get new Min PWM limit
            CH6_Max_New = int(Ch6_Trim + abs((CH6_Max_Orig - Ch6_Trim)/4))
#Get new Max PWM limit


            ServoFunctionValue5=
Read_Param_Value(m,'SERVO5_FUNCTION')#Verifiy servo function
            CH6_Min = Read_Param_Value(m, 'SERVO6_MIN')     #Elevator6 Min
            CH6_Max = Read_Param_Value(m, 'SERVO6_MAX')     #Elevator6 Max

            while (ServoFunctionValue5 != 0) and (CH6_Min != CH6_Min_New) and
(CH6_Max != CH6_Max_New):
                Set_Param(m,'SERVO5_FUNCTION', 0)       #Disables aileron
                Set_Param(m,'SERVO6_MIN',CH6_Min_New)   #Setting the new Min
PWM Limit

                Set_Param(m,'SERVO6_MAX',CH6_Max_New)   #Setting the new Max
PWM Limit

                time.sleep(0.02)
                CH6_Min = Read_Param_Value(m, 'SERVO6_MIN')     #Elevator6 Min
                CH6_Max = Read_Param_Value(m, 'SERVO6_MAX')     #Elevator6 Max
                ServoFunctionValue5=
Read_Param_Value(m,'SERVO5_FUNCTION')#Verifiy servo function

            print"AL5 FAILED ELE6 LIMITED!"

            start_time = time.time()
            end_time = start_time + duration
            while time.time() < end_time:
                Set_Servo(m,5,Ch5_Trim)#Do set servo command
                time.sleep(0.02)


            ServoFunctionValue5=
Read_Param_Value(m,'SERVO5_FUNCTION')#Verifiy servo function
            CH6_Min = Read_Param_Value(m, 'SERVO6_MIN')     #Elevator6 Min
            CH6_Max = Read_Param_Value(m, 'SERVO6_MAX')     #Elevator6 Max
            while (ServoFunctionValue5 != 4) and (CH6_Min != CH6_Min_Orig)
and (CH6_Max != CH6_Max_Orig):
                Set_Param(m,'SERVO5_FUNCTION', 4)       #Enalbles aileron
                Set_Param(m,'SERVO6_MIN',CH6_Min_Orig)  #Setting back to
```

```
original Min PWM Limit
                Set_Param(m,'SERVO6_MAX',CH6_Max_Orig)  #Setting back to the
original Max PWM Limit
                time.sleep(0.02)
                CH6_Min = Read_Param_Value(m, 'SERVO6_MIN')    #Elevator6 Min
                CH6_Max = Read_Param_Value(m, 'SERVO6_MAX')    #Elevator6 Max
                ServoFunctionValue5=
Read_Param_Value(m,'SERVO5_FUNCTION')#Verifiy servo function
            print"AL5 Restored ELE6 LIMIT REMOVED!"


##############################################################################
###################
        if failure_mode == 'C3':
            CH5_Min_New = int(Ch5_Trim - abs((CH5_Min_Orig - Ch5_Trim)/4))
#Get new Min PWM limit
            CH5_Max_New = int(Ch5_Trim + abs((CH5_Max_Orig - Ch5_Trim)/4))
#Get new Max PWM limit

            CH5_Min = Read_Param_Value(m, 'SERVO5_MIN')    #Aileron5 Min
            CH5_Max = Read_Param_Value(m, 'SERVO5_MAX')    #Aileron5 Max
            ServoFunctionValue6=
Read_Param_Value(m,'SERVO6_FUNCTION')#Verifiy servo function
            while (ServoFunctionValue6 != 0) and (CH5_Min != CH5_Min_New) and
(CH5_Max != CH5_Max_New):
                Set_Param(m,'SERVO6_FUNCTION', 0)       #Disables aileron
                Set_Param(m,'SERVO5_MIN',CH5_Min_New)   #Setting the new Min
PWM Limit
                Set_Param(m,'SERVO5_MAX',CH5_Max_New)   #Setting the new Max
PWM Limit
                time.sleep(0.02)
                CH5_Min = Read_Param_Value(m, 'SERVO5_MIN')    #Aileron5 Min
                CH5_Max = Read_Param_Value(m, 'SERVO5_MAX')    #Aileron5 Max
                ServoFunctionValue6=
Read_Param_Value(m,'SERVO6_FUNCTION')#Verifiy servo function

            print"AL5 LIMITED ELE6 FAILED!"
            start_time = time.time()
            end_time = start_time + duration
            while time.time() < end_time:
                Set_Servo(m,6,Ch6_Trim)#Do set servo command
                time.sleep(0.02)

            ServoFunctionValue6=
Read_Param_Value(m,'SERVO6_FUNCTION')#Verifiy servo function
            CH5_Min = Read_Param_Value(m, 'SERVO5_MIN')    #Aileron5 Min
            CH5_Max = Read_Param_Value(m, 'SERVO5_MAX')    #Aileron5 Max
            while (ServoFunctionValue6 != 19) and (CH5_Min != CH5_Min_Orig)
and (CH5_Max != CH5_Max_Orig) :
                Set_Param(m,'SERVO6_FUNCTION', 19)         #Enalbles Elevator
                Set_Param(m,'SERVO5_MIN',CH5_Min_Orig)     #Setting back to
original Min PWM Limit
                Set_Param(m,'SERVO5_MAX',CH5_Max_Orig)     #Setting back to
the original Max PWM Limit
                time.sleep(0.02)
                CH5_Min = Read_Param_Value(m, 'SERVO5_MIN')   #Aileron5 Min
                CH5_Max = Read_Param_Value(m, 'SERVO5_MAX')   #Aileron5 Max
                ServoFunctionValue6=
Read_Param_Value(m,'SERVO6_FUNCTION')#Verifiy servo function
            print"AL5 LIMIT REMOVED ELE6 RESTORED!"

############################################################################
####################
        if failure_mode == 'C4':
            CH5_Min_New = int(Ch5_Trim - abs((CH5_Min_Orig - Ch5_Trim)/4))
#Get new Min PWM limit
            CH5_Max_New = int(Ch5_Trim + abs((CH5_Max_Orig - Ch5_Trim)/4))
#Get new Max PWM limit
            CH6_Min_New = int(Ch6_Trim - abs((CH6_Min_Orig - Ch6_Trim)/4))
#Get new Min PWM limit
            CH6_Max_New = int(Ch6_Trim + abs((CH6_Max_Orig - Ch6_Trim)/4))
#Get new Max PWM limit


            CH5_Min = Read_Param_Value(m, 'SERVO5_MIN')     #Aileron5 Min
            CH5_Max = Read_Param_Value(m, 'SERVO5_MAX')     #Aileron5 Max
            CH6_Min = Read_Param_Value(m, 'SERVO6_MIN')     #Elevator6 Min
            CH6_Max = Read_Param_Value(m, 'SERVO6_MAX')     #Elevator6 Max
            while (CH5_Min != CH5_Min_New) and (CH5_Max != CH5_Max_New) and
(CH6_Min != CH6_Min_New) and (CH6_Max != CH6_Max_New):
                Set_Param(m,'SERVO5_MIN',CH5_Min_New)           #Setting the
new Min PWM Limit
                Set_Param(m,'SERVO5_MAX',CH5_Max_New)           #Setting the
new Max PWM Limit
                Set_Param(m,'SERVO6_MIN',CH6_Min_New)           #Setting the
new Min PWM Limit
                Set_Param(m,'SERVO6_MAX',CH6_Max_New)           #Setting the
new Max PWM Limit
                time.sleep(0.02)
                CH5_Min = Read_Param_Value(m, 'SERVO5_MIN')     #Aileron5 Min
                CH5_Max = Read_Param_Value(m, 'SERVO5_MAX')     #Aileron5 Max
                CH6_Min = Read_Param_Value(m, 'SERVO6_MIN')     #Elevator6 Min
                CH6_Max = Read_Param_Value(m, 'SERVO6_MAX')     #Elevator6 Max

            print"AL5 and ELE6 LIMITED!"

            time.sleep(duration)

            CH5_Min = Read_Param_Value(m, 'SERVO5_MIN')     #Aileron5 Min
            CH5_Max = Read_Param_Value(m, 'SERVO5_MAX')     #Aileron5 Max
            CH6_Min = Read_Param_Value(m, 'SERVO6_MIN')     #Elevator6 Min
            CH6_Max = Read_Param_Value(m, 'SERVO6_MAX')     #Elevator6 Max
            while (CH5_Min != CH5_Min_Orig) and (CH5_Max != CH5_Max_Orig) and
(CH6_Min != CH6_Min_Orig) and (CH6_Max != CH6_Max_Orig) :
                Set_Param(m,'SERVO5_MIN',CH5_Min_Orig)           #Setting
back to original Min PWM Limit
                Set_Param(m,'SERVO5_MAX',CH5_Max_Orig)           #Setting
back to the original Max PWM Limit
                Set_Param(m,'SERVO6_MIN',CH6_Min_Orig)           #Setting
back to original Min PWM Limit
                Set_Param(m,'SERVO6_MAX',CH6_Max_Orig)           #Setting
back to the original Max PWM Limit
                time.sleep(0.1)
                CH5_Min = Read_Param_Value(m, 'SERVO5_MIN')    #Aileron5 Min
                CH5_Max = Read_Param_Value(m, 'SERVO5_MAX')    #Aileron5 Max
                CH6_Min = Read_Param_Value(m, 'SERVO6_MIN')    #Elevator6 Min
                CH6_Max = Read_Param_Value(m, 'SERVO6_MAX')    #Elevator6 Max
            print"AL5 and ELE6 LIMIT REMOVED!"
#############################################################################
####################

        if failure_mode == 'C5':#L_ELE6_L_RUDD7

            CH6_Min_New = int(Ch6_Trim - abs((CH6_Min_Orig - Ch6_Trim)/4))
#Get new Min PWM limit
            CH6_Max_New = int(Ch6_Trim + abs((CH6_Max_Orig - Ch6_Trim)/4))
#Get new Max PWM limit
            CH7_Min_New = int(Ch7_Trim - abs((CH7_Min_Orig - Ch7_Trim)/4))
#Get new Min PWM limit
            CH7_Max_New = int(Ch7_Trim + abs((CH7_Max_Orig - Ch7_Trim)/4))
#Get new Max PWM limit


            CH6_Min = Read_Param_Value(m, 'SERVO6_MIN')     #Elevator6 Min
            CH6_Max = Read_Param_Value(m, 'SERVO6_MAX')     #Elevator6 Max
            CH7_Min = Read_Param_Value(m, 'SERVO7_MIN')     #Rudd7 Min
            CH7_Max = Read_Param_Value(m, 'SERVO7_MAX')     #Rudd7 Max
            while (CH6_Min != CH6_Min_New) and (CH6_Max != CH6_Max_New) and
(CH7_Min != CH7_Min_New) and (CH7_Max != CH7_Max_New) :
                Set_Param(m,'SERVO6_MIN',CH6_Min_New)           #Setting the
new Min PWM Limit
                Set_Param(m,'SERVO6_MAX',CH6_Max_New)           #Setting the
new Max PWM Limit
                Set_Param(m,'SERVO7_MIN',CH7_Min_New)           #Setting the
new Min PWM Limit
                Set_Param(m,'SERVO7_MAX',CH7_Max_New)           #Setting the
new Max PWM Limit
                time.sleep(0.02)
                CH6_Min = Read_Param_Value(m, 'SERVO6_MIN')     #Elevator6 Min
                CH6_Max = Read_Param_Value(m, 'SERVO6_MAX')     #Elevator6 Max
                CH7_Min = Read_Param_Value(m, 'SERVO7_MIN')     #Rudd7 Min
                CH7_Max = Read_Param_Value(m, 'SERVO7_MAX')     #Rudd7 Max

            print"Rudd7 and ELE6 LIMITED!"

            time.sleep(duration)

            CH6_Min = Read_Param_Value(m, 'SERVO6_MIN')     #Elevator6 Min
            CH6_Max = Read_Param_Value(m, 'SERVO6_MAX')     #Elevator6 Max
            CH7_Min = Read_Param_Value(m, 'SERVO7_MIN')     #Rudd7 Min
            CH7_Max = Read_Param_Value(m, 'SERVO7_MAX')     #Rudd7 Max
            while (CH6_Min != CH6_Min_Orig) and (CH6_Max != CH6_Max_Orig) and
(CH7_Min != CH7_Min_Orig) and (CH7_Max != CH7_Max_Orig) :
                Set_Param(m,'SERVO6_MIN',CH6_Min_Orig)           #Setting
back to original Min PWM Limit
                Set_Param(m,'SERVO6_MAX',CH6_Max_Orig)           #Setting
back to the original Max PWM Limit
                Set_Param(m,'SERVO7_MIN',CH7_Min_Orig)           #Setting
back to original Min PWM Limit
                Set_Param(m,'SERVO7_MAX',CH7_Max_Orig)           #Setting
back to the original Max PWM Limit
```

```
                time.sleep(0.1)
                CH6_Min = Read_Param_Value(m, 'SERVO6_MIN')    #Elevator6 Min
                CH6_Max = Read_Param_Value(m, 'SERVO6_MAX')    #Elevator6 Max
                CH7_Min = Read_Param_Value(m, 'SERVO7_MIN')    #Rudd7 Min
                CH7_Max = Read_Param_Value(m, 'SERVO7_MAX')    #Rudd7 Max
            print"Rudd7 and ELE6 LIMIT REMOVED!"

############################################################################
######################
        if failure_mode == 'C6': # F_ELE6_L_Rudd
            CH7_Min_New = int(Ch7_Trim - abs((CH7_Min_Orig - Ch7_Trim)/4))
#Get new Min PWM limit
            CH7_Max_New = int(Ch7_Trim + abs((CH7_Max_Orig - Ch7_Trim)/4))
#Get new Max PWM limit

            CH7_Min = Read_Param_Value(m, 'SERVO7_MIN')    #Rudd7 Min
            CH7_Max = Read_Param_Value(m, 'SERVO7_MAX')    #Rudd7 Max
            ServoFunctionValue6=
Read_Param_Value(m,'SERVO6_FUNCTION')#Verifiy servo function
            while (ServoFunctionValue6 != 0) and (CH7_Min != CH7_Min_New) and
(CH7_Max != CH7_Max_New):
                Set_Param(m,'SERVO6_FUNCTION', 0)      #Disables aileron
                Set_Param(m,'SERVO7_MIN',CH7_Min_New)    #Setting the new Min
PWM Limit
                Set_Param(m,'SERVO7_MAX',CH7_Max_New)    #Setting the new Max
PWM Limit
                time.sleep(0.02)
                CH7_Min = Read_Param_Value(m, 'SERVO7_MIN')     #Rudd7 Min
                CH7_Max = Read_Param_Value(m, 'SERVO7_MAX')     #Rudd7 Max
                ServoFunctionValue6=
Read_Param_Value(m,'SERVO6_FUNCTION')#Verifiy servo function

            print"RUDD7 LIMITED ELE6 FAILED!"
            start_time = time.time()
            end_time = start_time + duration
            while time.time() < end_time:
                Set_Servo(m,6,Ch6_Trim)#Do set servo command
                time.sleep(0.02)

            ServoFunctionValue6=
Read_Param_Value(m,'SERVO6_FUNCTION')#Verifiy servo function
            CH7_Min = Read_Param_Value(m, 'SERVO7_MIN')     #Rudd7 Min
            CH7_Max = Read_Param_Value(m, 'SERVO7_MAX')     #Rudd7 Max
            while (ServoFunctionValue6 != 19) and (CH7_Min != CH7_Min_Orig)
and (CH7_Max != CH7_Max_Orig) :
                Set_Param(m,'SERVO6_FUNCTION', 19)        #Enalbles Elevator
                Set_Param(m,'SERVO7_MIN',CH7_Min_Orig)      #Setting back to
original Min PWM Limit
                Set_Param(m,'SERVO7_MAX',CH7_Max_Orig)      #Setting back to
the original Max PWM Limit
                time.sleep(0.02)
                CH7_Min = Read_Param_Value(m, 'SERVO7_MIN')    #Rudd7 Min
                CH7_Max = Read_Param_Value(m, 'SERVO7_MAX')    #Rudd7 Max
                ServoFunctionValue6=
Read_Param_Value(m,'SERVO6_FUNCTION')#Verifiy servo function
            print"RUDD7 LIMIT REMOVED ELE6 RESTORED!"
############################################################################
```

```
#####################
        if failure_mode == 'C7': #L_AL5_L_Rudd
            CH5_Min_New = int(Ch5_Trim - abs((CH5_Min_Orig - Ch5_Trim)/4))
#Get new Min PWM limit
            CH5_Max_New = int(Ch5_Trim + abs((CH5_Max_Orig - Ch5_Trim)/4))
#Get new Max PWM limit
            CH7_Min_New = int(Ch7_Trim - abs((CH7_Min_Orig - Ch7_Trim)/4))
#Get new Min PWM limit
            CH7_Max_New = int(Ch7_Trim + abs((CH7_Max_Orig - Ch7_Trim)/4))
#Get new Max PWM limit


            CH5_Min = Read_Param_Value(m, 'SERVO5_MIN')    #Aileron5 Min
            CH5_Max = Read_Param_Value(m, 'SERVO5_MAX')    #Aileron5 Max
            CH7_Min = Read_Param_Value(m, 'SERVO7_MIN')    #Rudd7 Min
            CH7_Max = Read_Param_Value(m, 'SERVO7_MAX')    #Rudd7 Max
            while (CH5_Min != CH5_Min_New) and (CH5_Max != CH5_Max_New) and
(CH7_Min != CH7_Min_New) and (CH7_Max != CH7_Max_New):
                Set_Param(m,'SERVO5_MIN',CH5_Min_New)           #Setting the
new Min PWM Limit
                Set_Param(m,'SERVO5_MAX',CH5_Max_New)           #Setting the
new Max PWM Limit
                Set_Param(m,'SERVO7_MIN',CH7_Min_New)           #Setting the
new Min PWM Limit
                Set_Param(m,'SERVO7_MAX',CH7_Max_New)           #Setting the
new Max PWM Limit
                time.sleep(0.02)
                CH5_Min = Read_Param_Value(m, 'SERVO5_MIN')    #Aileron5 Min
                CH5_Max = Read_Param_Value(m, 'SERVO5_MAX')    #Aileron5 Max
                CH7_Min = Read_Param_Value(m, 'SERVO7_MIN')    #Rudd7 Min
                CH7_Max = Read_Param_Value(m, 'SERVO7_MAX')    #Rudd7 Max

            print"AL5 and RUDD7 LIMITED!"

            time.sleep(duration)

            CH5_Min = Read_Param_Value(m, 'SERVO5_MIN')    #Aileron5 Min
            CH5_Max = Read_Param_Value(m, 'SERVO5_MAX')    #Aileron5 Max
            CH7_Min = Read_Param_Value(m, 'SERVO7_MIN')    #Rudd7 Min
            CH7_Max = Read_Param_Value(m, 'SERVO7_MAX')    #Rudd7 Max
            while (CH5_Min != CH5_Min_Orig) and (CH5_Max != CH5_Max_Orig) and
(CH7_Min != CH7_Min_Orig) and (CH7_Max != CH7_Max_Orig) :
                Set_Param(m,'SERVO5_MIN',CH5_Min_Orig)          #Setting
back to original Min PWM Limit
                Set_Param(m,'SERVO5_MAX',CH5_Max_Orig)          #Setting
back to the original Max PWM Limit
                Set_Param(m,'SERVO7_MIN',CH7_Min_Orig)          #Setting
back to original Min PWM Limit
                Set_Param(m,'SERVO7_MAX',CH7_Max_Orig)          #Setting
back to the original Max PWM Limit
                time.sleep(0.1)
                CH5_Min = Read_Param_Value(m, 'SERVO5_MIN')    #Aileron5 Min
                CH5_Max = Read_Param_Value(m, 'SERVO5_MAX')    #Aileron5 Max
                CH7_Min = Read_Param_Value(m, 'SERVO7_MIN')    #Elevator6 Min
                CH7_Max = Read_Param_Value(m, 'SERVO7_MAX')    #Elevator6 Max
            print"AL5 and RUDD7 LIMIT REMOVED!"
#############################################################################
```

```
######################
        if failure_mode == 'C8': # F_AL5_L_Rudd
            CH7_Min_New = int(Ch7_Trim - abs((CH7_Min_Orig - Ch7_Trim)/4))
#Get new Min PWM limit
            CH7_Max_New = int(Ch7_Trim + abs((CH7_Max_Orig - Ch7_Trim)/4))
#Get new Max PWM limit


            ServoFunctionValue5=
Read_Param_Value(m,'SERVO5_FUNCTION')#Verifiy servo function
            CH7_Min = Read_Param_Value(m, 'SERVO7_MIN')     #Rudd7 Min
            CH7_Max = Read_Param_Value(m, 'SERVO7_MAX')     #Rudd7 Max

            while (ServoFunctionValue5 != 0) and (CH7_Min != CH7_Min_New) and
(CH7_Max != CH7_Max_New):
                Set_Param(m,'SERVO5_FUNCTION', 0)       #Disables aileron
                Set_Param(m,'SERVO7_MIN',CH7_Min_New)   #Setting the new Min
PWM Limit
                Set_Param(m,'SERVO7_MAX',CH7_Max_New)   #Setting the new Max
PWM Limit
                time.sleep(0.02)
                CH7_Min = Read_Param_Value(m, 'SERVO7_MIN')     #Rudd7 Min
                CH7_Max = Read_Param_Value(m, 'SERVO7_MAX')     #Rudd7 Max
                ServoFunctionValue5=
Read_Param_Value(m,'SERVO5_FUNCTION')#Verifiy servo function

            print"AL5 FAILED Rudd7 LIMITED!"

            start_time = time.time()
            end_time = start_time + duration
            while time.time() < end_time:
                Set_Servo(m,5,Ch5_Trim)#Do set servo command
                time.sleep(0.02)


            ServoFunctionValue5=
Read_Param_Value(m,'SERVO5_FUNCTION')#Verifiy servo function
            CH7_Min = Read_Param_Value(m, 'SERVO7_MIN')     #Rudd7 Min
            CH7_Max = Read_Param_Value(m, 'SERVO7_MAX')     #Rudd7 Max
            while (ServoFunctionValue5 != 4) and (CH7_Min != CH7_Min_Orig)
and (CH7_Max != CH7_Max_Orig):
                Set_Param(m,'SERVO5_FUNCTION', 4)       #Enalbles aileron
                Set_Param(m,'SERVO7_MIN',CH7_Min_Orig)  #Setting back to
original Min PWM Limit
                Set_Param(m,'SERVO7_MAX',CH7_Max_Orig)  #Setting back to the
original Max PWM Limit
                time.sleep(0.02)
                CH7_Min = Read_Param_Value(m, 'SERVO7_MIN')    #Rudd7 Min
                CH7_Max = Read_Param_Value(m, 'SERVO7_MAX')    #Rudd7 Max
                ServoFunctionValue5=
Read_Param_Value(m,'SERVO5_FUNCTION')#Verifiy servo function
            print"AL5 Restored Rudd7 LIMIT REMOVED!"
```

```
    return


def main():

   # read command-line options
   parser = OptionParser("readdata.py [options]")
   parser.add_option("--baudrate", dest="baudrate", type='int',
                     help="master port baud rate", default=921600)
   parser.add_option("--device", dest="device",
default="/dev/ttyPIXHAWK_CONTROL", help="serial device")
   parser.add_option("--rate", dest="rate", default=4, type='int',
help="requested stream rate")
   parser.add_option("--source-system", dest='SOURCE_SYSTEM', type='int',
                     default=255, help='MAVLink source system for this GCS')
   parser.add_option("--showmessages", dest="showmessages",
action='store_true',
                     help="show incoming messages", default=False)
   (opts, args) = parser.parse_args()

   if opts.device is None:
      print("You must specify a serial device")
      sys.exit(1)

   # create a mavlink serial instance
   master = mavutil.mavlink_connection(opts.device, baud=opts.baudrate)

   # wait for the heartbeat msg to find the system ID
   master.wait_heartbeat()

   # request data to be sent at the given rate
   master.mav.request_data_stream_send(master.target_system,
master.target_component,
      mavutil.mavlink.MAV_DATA_STREAM_ALL, opts.rate, 1)



   # enter the data loop
   read_loop(master)




if __name__ == '__main__':
   main()
```

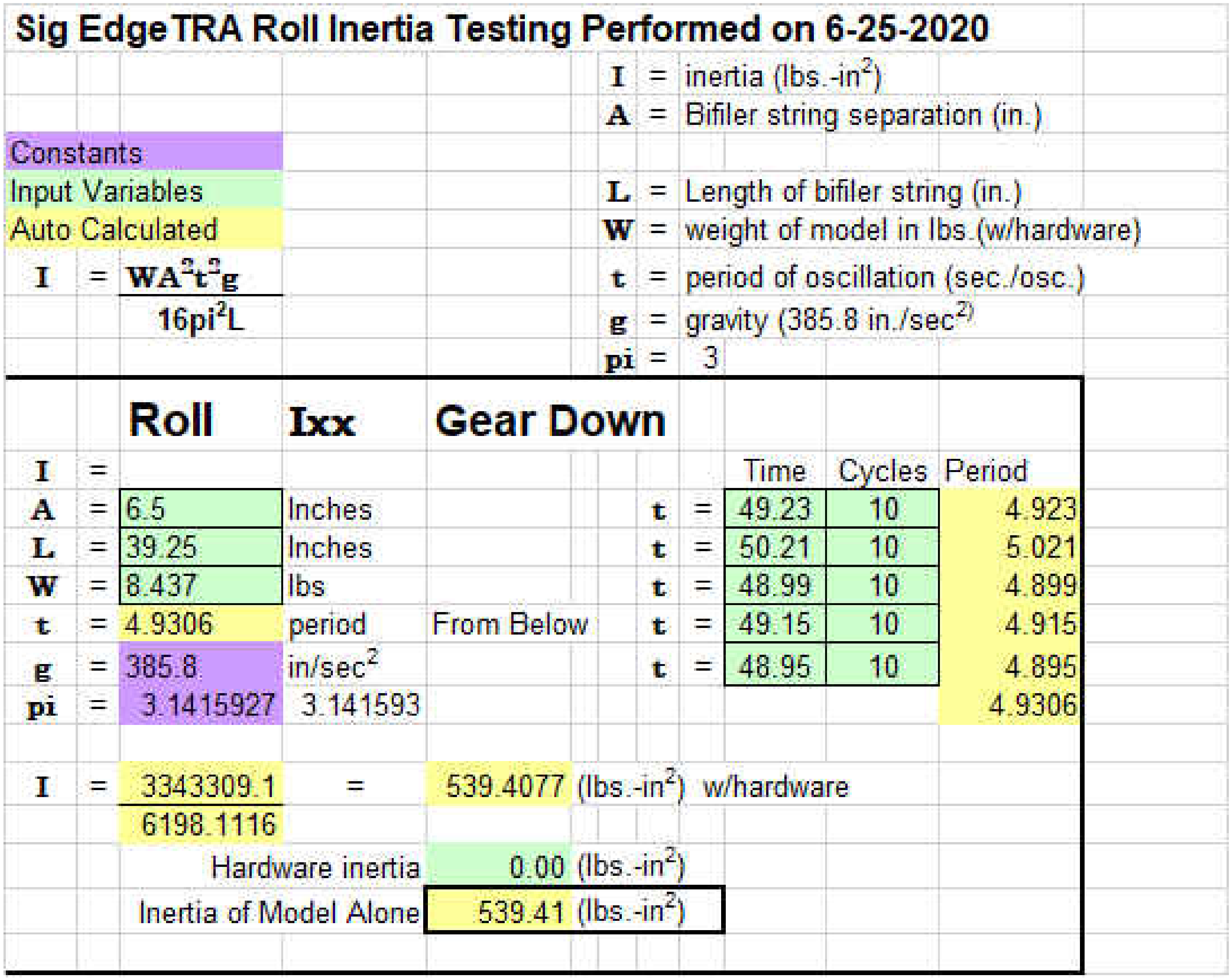
APPENDIX D

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

"""

Fly_Mission_and_Maneuver_Plane_Rev2
Code Written by Brian Duvall April 2020
Flys plane in an oval pattern about four waypoints,in addition, provides
inputs between two of the four points
import code
code.interact(local=locals())
"""
from __future__ import print_function, division
from dronekit import connect, VehicleMode, LocationGlobalRelative,
LocationGlobal, Command
from my_vehicle import MyVehicle #Our custom vehicle class
import time
import math
import numpy as np
from pymavlink import mavutil
#import matplotlib.pyplot as plt
import os

#Set up option parsing to get the connection string
import argparse
parser = argparse.ArgumentParser(description='Demonstrates basic mission
operations.')
parser.add_argument('--connect',default= "/dev/ttyPIXHAWK_CONTROL",
help="vehicle connection target string")
args = parser.parse_args()
connection_string = args.connect
# Connect to the Vehicle
print('Connecting to vehicle on: %s' % connection_string)
vehicle = connect(connection_string, wait_ready=True, baud=57600,
vehicle_class=MyVehicle)

point3 = None #Global value
vehicle.channels.overrides['6'] = 1500 #Force to wait to take data
#os.system('python /Data_Recorder/Ardupilot/Data_Recorder_MIMO.py ') # Trying
to autostart data colection code



def get_location_metres(original_location, dNorth, dEast):

    """
    Returns a LocationGlobal object containing the latitude/longitude
`dNorth` and `dEast` meters from the
    specified `original_location`. The returned Location has the same `alt`
value
    as `original_location`.

    The function is useful when you want to move the vehicle around
specifying locations relative to
```

```
    the current vehicle position.
    The algorithm is relatively accurate over small distances (10m within
1km) except close to the poles.
    For more information, see:
    http://gis.stackexchange.com/questions/2951/algorithm-for-offsetting-a-
latitude-longitude-by-some-amount-of-meters
    """
    earth_radius=6378137.0 #Radius of "spherical" earth
    #Coordinate offsets in radians
    dLat = dNorth/earth_radius
    dLon = dEast/(earth_radius*math.cos(math.pi*original_location.lat/180))

    #New position in decimal degrees
    newlat = original_location.lat + (dLat * 180/math.pi)
    newlon = original_location.lon + (dLon * 180/math.pi)
    return LocationGlobal(newlat, newlon,original_location.alt)


def get_distance_metres(aLocation1, aLocation2):
    """
    Returns the ground distance in meters between two LocationGlobal objects.

    This method is an approximation, and will not be accurate over large
distances and close to the
    earth's poles. It comes from the ArduPilot test code:

https://github.com/diydrones/ardupilot/blob/master/Tools/autotest/common.py
    """
    dlat = aLocation2.lat - aLocation1.lat
    dlong = aLocation2.lon - aLocation1.lon
    return math.sqrt((dlat*dlat) + (dlong*dlong)) * 1.113195e5



def distance_to_current_waypoint():
    """
    Gets distance in meters to the current waypoint.
    It returns None for the first waypoint (Home location).
    """
    nextwaypoint = vehicle.commands.next
    if nextwaypoint==0:
        return None
    missionitem=vehicle.commands[nextwaypoint-1] #commands are zero indexed
    lat = missionitem.x
    lon = missionitem.y
    alt = missionitem.z
    targetWaypointLocation = LocationGlobalRelative(lat,lon,alt)
    distancetopoint = get_distance_metres(vehicle.location.global_frame,
targetWaypointLocation)
    return distancetopoint


def download_mission():
    """
    Download the current mission from the vehicle.
    """
    cmds = vehicle.commands
    cmds.download()
    cmds.wait_ready() # wait until download is complete.


def adds_takeoff_mission(aLocation):
    """
    Only used when connected to SIM
    Adds a takeoff command
    The function assumes vehicle.commands matches the vehicle mission state
    (you must have called download at least once in the session and after
clearing the mission)
    """
    cmds = vehicle.commands
    print(" Clear any existing commands")
    cmds.clear()
    print(" Define/add new commands.")
    # Add new commands. The meaning/order of the parameters is documented in
the Command class.
    cmds.add(Command( 0, 0, 0, mavutil.mavlink.MAV_FRAME_GLOBAL_RELATIVE_ALT,
mavutil.mavlink.MAV_CMD_NAV_TAKEOFF, 0, 0, 0, 0, 0, 0, aLocation.lat,
aLocation.lon, 100))
    cmds.add(Command( 0, 0, 0, mavutil.mavlink.MAV_FRAME_GLOBAL_RELATIVE_ALT,
mavutil.mavlink.MAV_CMD_NAV_TAKEOFF, 0, 0, 0, 0, 0, 0, aLocation.lat,
aLocation.lon, 100))

    print(" Upload new commands to vehicle")
    cmds.upload()

def adds_fly_between_mission(aLocation):
    """
    The function assumes vehicle.commands matches the vehicle mission state
    (you must have called download at least once in the session and after
clearing the mission)
    """
    global point3 #This is the target point to fly to when doing a mauver

    cmds = vehicle.commands

    #download_mission()

    print(" Clear any existing commands")
    cmds.clear()

    print(" Define/add new commands.")
    # Add new commands. The meaning/order of the parameters is documented in
the Command class.
##  (North/South, East/West)
##    point1 = get_location_metres(aLocation, 120, 230) # Old points that
worked well in sim
##    point2 = get_location_metres(aLocation, 170, 150) # Old point that
worked well in sim
    point1 = get_location_metres(aLocation, 100, 300)
    point2 = get_location_metres(aLocation, 170, 170)
    point3 = get_location_metres(aLocation,-180, -70)
    point4 = get_location_metres(aLocation,-200, 75)
    cmds.add(Command( 0, 0, 0, mavutil.mavlink.MAV_FRAME_GLOBAL_RELATIVE_ALT,
mavutil.mavlink.MAV_CMD_NAV_WAYPOINT, 0, 0, 0, 0, 0, 0, point1.lat,
point1.lon, 75))
    cmds.add(Command( 0, 0, 0, mavutil.mavlink.MAV_FRAME_GLOBAL_RELATIVE_ALT,
mavutil.mavlink.MAV_CMD_NAV_WAYPOINT, 0, 0, 0, 0, 0, 0, point2.lat,
point2.lon, 75))
    cmds.add(Command( 0, 0, 0, mavutil.mavlink.MAV_FRAME_GLOBAL_RELATIVE_ALT,
mavutil.mavlink.MAV_CMD_NAV_WAYPOINT, 0, 0, 0, 0, 0, 0, point3.lat,
point3.lon, 75))
    cmds.add(Command( 0, 0, 0, mavutil.mavlink.MAV_FRAME_GLOBAL_RELATIVE_ALT,
mavutil.mavlink.MAV_CMD_NAV_WAYPOINT, 0, 0, 0, 0, 0, 0, point4.lat,
point4.lon, 75))
    cmds.add(Command( 0, 0, 0, mavutil.mavlink.MAV_FRAME_GLOBAL_RELATIVE_ALT,
mavutil.mavlink.MAV_CMD_DO_JUMP, 0, 0, 1, 1, 0, 0, 0, 0, 0))

    print(" Upload new commands to vehicle")
    cmds.upload()

def arm_and_takeoff(aTargetAltitude):
    """
    Arms vehicle and fly to aTargetAltitude.
    """

    print("Basic pre-arm checks")
    # Don't let the user try to arm until autopilot is ready
    while not vehicle.is_armable:
        print("Status",vehicle.is_armable)
        print(" Waiting for vehicle to initialize...")
        time.sleep(1)


    print("Taking Off!")
   #Confirm vehicle armed before attempting to take off
    while not vehicle.armed:
        print(" Waiting for arming...")
        vehicle.armed = True
        time.sleep(1)

    while vehicle.mode != 'AUTO':
        print("setting mode AUTO")
        vehicle.mode = VehicleMode("AUTO")
        time.sleep(1)

    # Wait until the vehicle reaches a safe height before processing the goto
(otherwise the command
    while True:
        print(" Altitude: ", vehicle.location.global_relative_frame.alt)
        if vehicle.location.global_relative_frame.alt>=aTargetAltitude*0.95:
#Trigger just below target alt.
            print("Reached target altitude changing")
            vehicle.mode = VehicleMode("RTL")
            break

def Sin_wave_generator(S_Rate,freq,Duration, Amp, RC_Trim):
    #y = Asin(2*PI*f*t+phi)
    T= 1/S_Rate #Time period of one sample
    N = S_Rate*Duration #Number of samples in the given duration
    omega = 2*np.pi*freq #Angular freqency
    t_seq = np.arange(N)*T #Time Sequence
    y = Amp*np.sin(omega*t_seq) + RC_Trim #Sin wave function
    y = y.astype(int) #Convert to integers for PWM values
    return(y, t_seq)

def Home_Location_Check():
    while not vehicle.home_location:
        cmds= vehicle.commands
        cmds.download()
        cmds.wait_ready()
    print ("Got Home Location")

def calculate_compass_bearing(pointA, pointB):
    """
    Calculates the bearing between two points.
    The formula used is the following:
        θ = atan2(sin(Δlong).cos(lat2),
                  cos(lat1).sin(lat2) − sin(lat1).cos(lat2).cos(Δlong))
    :Parameters:
      - `pointA: The tuple representing the latitude/longitude for the
        first point. Latitude and longitude must be in decimal degrees
      - `pointB: The tuple representing the latitude/longitude for the
        second point. Latitude and longitude must be in decimal degrees
    :Returns:
      The bearing in degrees
    :Returns Type:
      float
    """
    if (type(pointA) != tuple) or (type(pointB) != tuple):
        raise TypeError("Only tuples are supported as arguments")

    lat1 = math.radians(pointA[0])
    lat2 = math.radians(pointB[0])

    diffLong = math.radians(pointB[1] - pointA[1])

    x = math.sin(diffLong) * math.cos(lat2)
    y = math.cos(lat1) * math.sin(lat2) - (math.sin(lat1)
            * math.cos(lat2) * math.cos(diffLong))

    initial_bearing = math.atan2(x, y)

    # Now we have the initial bearing but math.atan2 return values
    # from -180° to + 180° which is not what we want for a compass bearing
    # The solution is to normalize the initial bearing as shown below
    initial_bearing = math.degrees(initial_bearing)
    compass_bearing = (initial_bearing + 360) % 360

    return compass_bearing

def Input_Command_Builder_MIMO():
    """
    Inputs used in SIM

    roll_S_Input, t = Sin_wave_generator(25,1,3,200,1480)
    pitch_S_Input, t = Sin_wave_generator(25,1,3,200,1520)
    #yaw_S_Input, t = Sin_wave_generator(25,1,3,70,1500)
    yaw_S_Input, t = Sin_wave_generator(25,1,3,200,1550)
    """
    #Inputs used in plane
    roll_S_Input, t = Sin_wave_generator(100,1,3,200,1480)# Go above 1480 to
bias right roll from tail
    pitch_S_Input, t = Sin_wave_generator(100,0.5,3,250,1520)#Go below 1520
to bias pitch up
    yaw_S_Input, t = Sin_wave_generator(100,1,3,200,1550)#Go below 1500 to
bias right yaw

    ##plt.plot(t,roll_S_Input,'+-')
    ##plt.show()
    # padding input arrays with Nones to send to the Cube Orange at one time
    N = len(roll_S_Input) + len(pitch_S_Input) + len(yaw_S_Input)
    Number_of_None_Padding_Roll= N-len(roll_S_Input)
    Number_of_None_Padding_Pitch = N- len(pitch_S_Input)
    Number_of_None_Padding_Yaw = N-len(yaw_S_Input)

    i=1
    while i <= Number_of_None_Padding_Roll:
        roll_S_Input = np.append(roll_S_Input,0)
        i=i+1
    roll_padded = np.append(roll_S_Input,0) # added an extra None to array so
rudder gose nutral
    #####################################
    front = np.array([])
    back = np.array([])
    i=1
    while i <= Number_of_None_Padding_Pitch/2: # Building front array of
Nones
        front = np.append(front,0)
        i=i+1
    i=1
    while i <= Number_of_None_Padding_Pitch/2:# Building back array of Nones
        back = np.append(back,0)
        i=i+1

    front_pitch = np.append(front,pitch_S_Input)# append the None's to the
beginning of pitch signal
    pitch_padded = np.append(front_pitch,back) # append beginning None's and
pitch to the back
    pitch_padded = np.append(pitch_padded,0)# added an extra None to array so
rudder gose nutral

    #####################################
    front = np.array([])
    i=1
    while i <= Number_of_None_Padding_Yaw:
        front = np.append(front,0)
        i=i+1
    yaw_padded = np.append(front,yaw_S_Input)
    yaw_padded = np.append(yaw_padded,0) # added an extra None to array so
rudder gose nutral

    ####################################
    #This is here to prevent the channel overrides from stoping data
```

```
recording
    ch6_padded = np.array([])
    i=1
    while i<=N:
        ch6_padded = np.append(ch6_padded,1200)
        i=i+1
    ch6_padded = np.append(ch6_padded,1200) # Keeps arrays the same length
due to adition None for rudder to go nutral


    return(roll_padded, pitch_padded, yaw_padded, ch6_padded)

def Channel_Override(roll,pitch,yaw,ch6):
    vehicle.message_factory.rc_channels_override_send(
    0,#master.target_system
    0,#master.target_component
    roll,  #Aileron  1
    pitch, #Elevator 2
    0,     #Throttle 3
    yaw,   #Rudder   4
    0,     #Channel  5
    ch6,   #Channel  6
    0,     #Channel  7
    0)     #Channel  8


def Manuver_Plane(roll_padded, pitch_padded, yaw_padded, ch6_padded):
    nextwaypoint=vehicle.commands.next
    while True:
        nextwaypoint=vehicle.commands.next
        Heading = vehicle.heading
        Vehicle_Location = (vehicle.location.global_relative_frame.lat,
vehicle.location.global_relative_frame.lon)
        Waypoint_Location = (point3.lat, point3.lon)
        Waypoint_Heading3 = calculate_compass_bearing(Vehicle_Location,
Waypoint_Location)
        roll_attitude = vehicle.attitude.roll
##        print("Roll_Attitude",vehicle.attitude.roll)
##        print("Roll_Target_Attitude",
vehicle.nav_controller_output.nav_roll)
##        print("Waypoint_Heading3",Waypoint_Heading3)
##        print("Next waypoint", nextwaypoint)
##        print('Distance to waypoint (%s): %s' % (nextwaypoint,
distance_to_current_waypoint()))
##        print ("Heading",Heading)

        if (nextwaypoint == 3) and (Heading in range(int(Waypoint_Heading3)-
5, int(Waypoint_Heading3)+5)):
            time.sleep(2)# Give some time for the plane to get trim
conditions
            print ("In-line ready to start maneuver")
            ##########################################
            vehicle.channels.overrides['6'] = 1200 # Start collecting data!
            i=0
            dt = 0.01 # send messages at this interval
            while i < len(roll_padded):
                Channel_Override(roll_padded[i], pitch_padded[i],
yaw_padded[i], ch6_padded[i]) #Simple version of channel override
                time.sleep(dt) #wait to send the next message
                i=i+1

            time.sleep(1) # Give some time for the plane to go back to
neutral
            vehicle.channels.overrides['6'] = 1900 # Stop collecting data and
process!
            ##########################################
            count = 0
            while nextwaypoint != 1:
                if count == 0:
                    print ("Done, waiting to go around")
                    count= count+1
                nextwaypoint=vehicle.commands.next
        time.sleep(0.1)


#############################################################################
#############################################################################
########################################
#Starting to run the script
if connection_string == '127.0.0.1:14551':
    adds_takeoff_mission(vehicle.location.global_frame) #Send to the comand
to have the plane takeoff
    arm_and_takeoff(50)# ARM the vehicle and set it to auto
    time.sleep(10)
#############################################################################
#####
Home_Location_Check()                                                   # Ensure
home location is availibule
adds_fly_between_mission(vehicle.home_location)                         # Writes
waypoints for plane to fly to
print("Mission Loaded")
roll_padded,pitch_padded,yaw_padded,ch6_padded = Input_Command_Builder_MIMO()
# Gets intput command sin_wave for roll pitch yaw
print("Inputs_Built")


print("Starting mission, setting mode to AUTO")
# Reset mission set to first (0) waypoint
vehicle.commands.next=0

# Set mode to AUTO to start the mission
while vehicle.mode != "AUTO":
    vehicle.mode = VehicleMode("AUTO")

Manuver_Plane(roll_padded,pitch_padded,yaw_padded,ch6_padded)
#############################################################################
#############################################################################
######################################



#Close vehicle object before exiting the script
print("Close vehicle object")
vehicle.close()
```

APPENDIX E



Sig Edge TRA Roll Inertia Testing Performed on 6-25-2020

I	=	inertia (lbs.-in^2)
A	=	Bifiler string separation (in.)
L	=	Length of bifiler string (in.)
W	=	weight of model in lbs.(w/hardware)
t	=	period of oscillation (sec./osc.)
g	=	gravity (385.8 in./sec^2)
pi	=	3

Constants
Input Variables
Auto Calculated

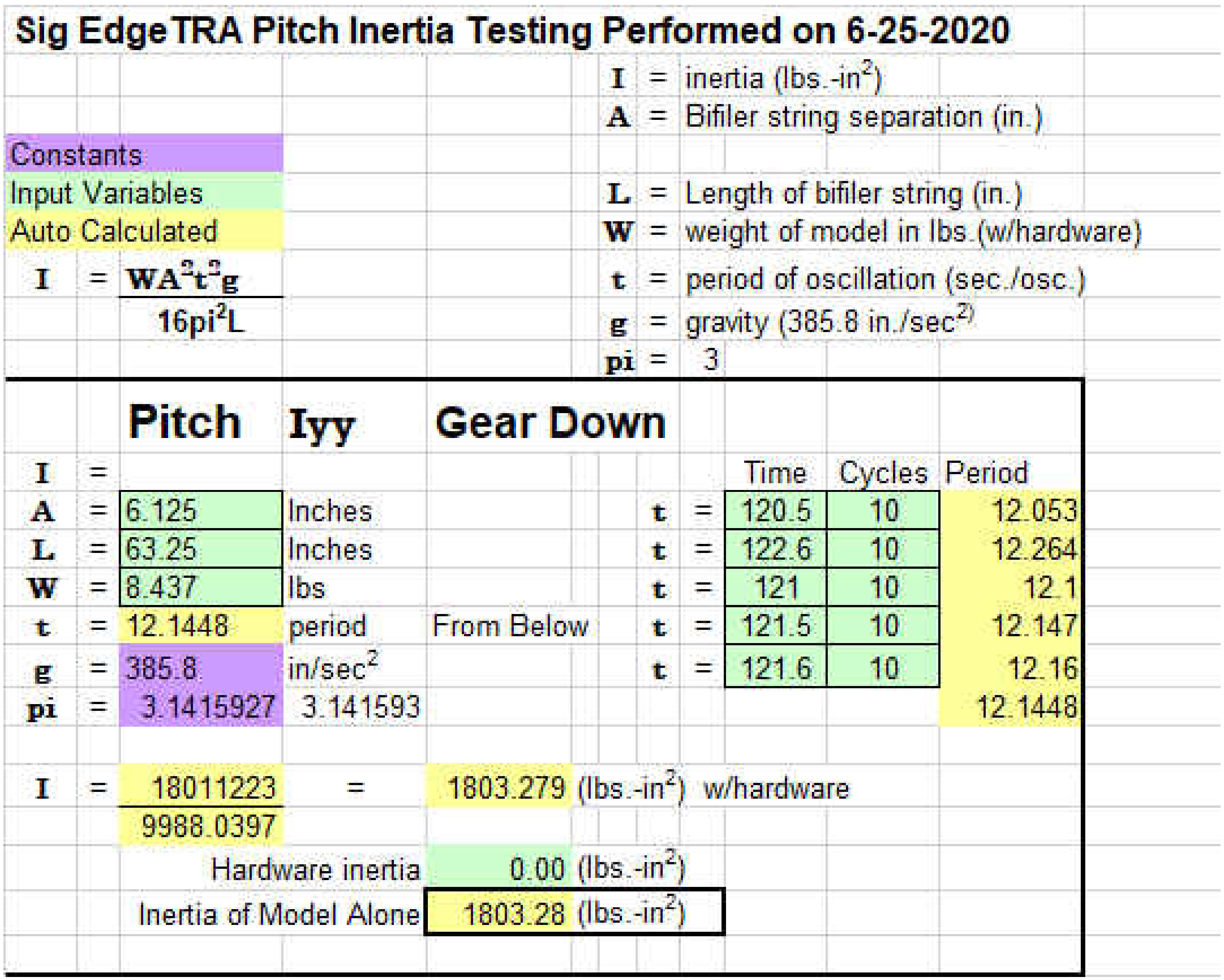
$$I = \frac{WA^2t^2g}{16pi^2L}$$

Roll **Ixx** **Gear Down**

							Time	Cycles	Period
I	=								
A	=	6.5	Inches		**t**	=	49.23	10	4.923
L	=	39.25	Inches		**t**	=	50.21	10	5.021
W	=	8.437	lbs		**t**	=	48.99	10	4.899
t	=	4.9306	period	From Below	**t**	=	49.15	10	4.915
g	=	385.8	in/sec^2		**t**	=	48.95	10	4.895
pi	=	3.1415927	3.141593						4.9306

$$I = \frac{3343309.1}{6198.1116} = 539.4077 \text{ (lbs.-in}^2\text{) w/hardware}$$

Hardware inertia	0.00	(lbs.-in^2)
Inertia of Model Alone	539.41	(lbs.-in^2)



Sig EdgeTRA Pitch Inertia Testing Performed on 6-25-2020

I	=	inertia (lbs.-in^2)
A	=	Bifiler string separation (in.)
L	=	Length of bifiler string (in.)
W	=	weight of model in lbs.(w/hardware)
t	=	period of oscillation (sec./osc.)
g	=	gravity (385.8 in./sec$^{2)}$
pi	=	3

Constants 🟪

Input Variables 🟩

Auto Calculated 🟨

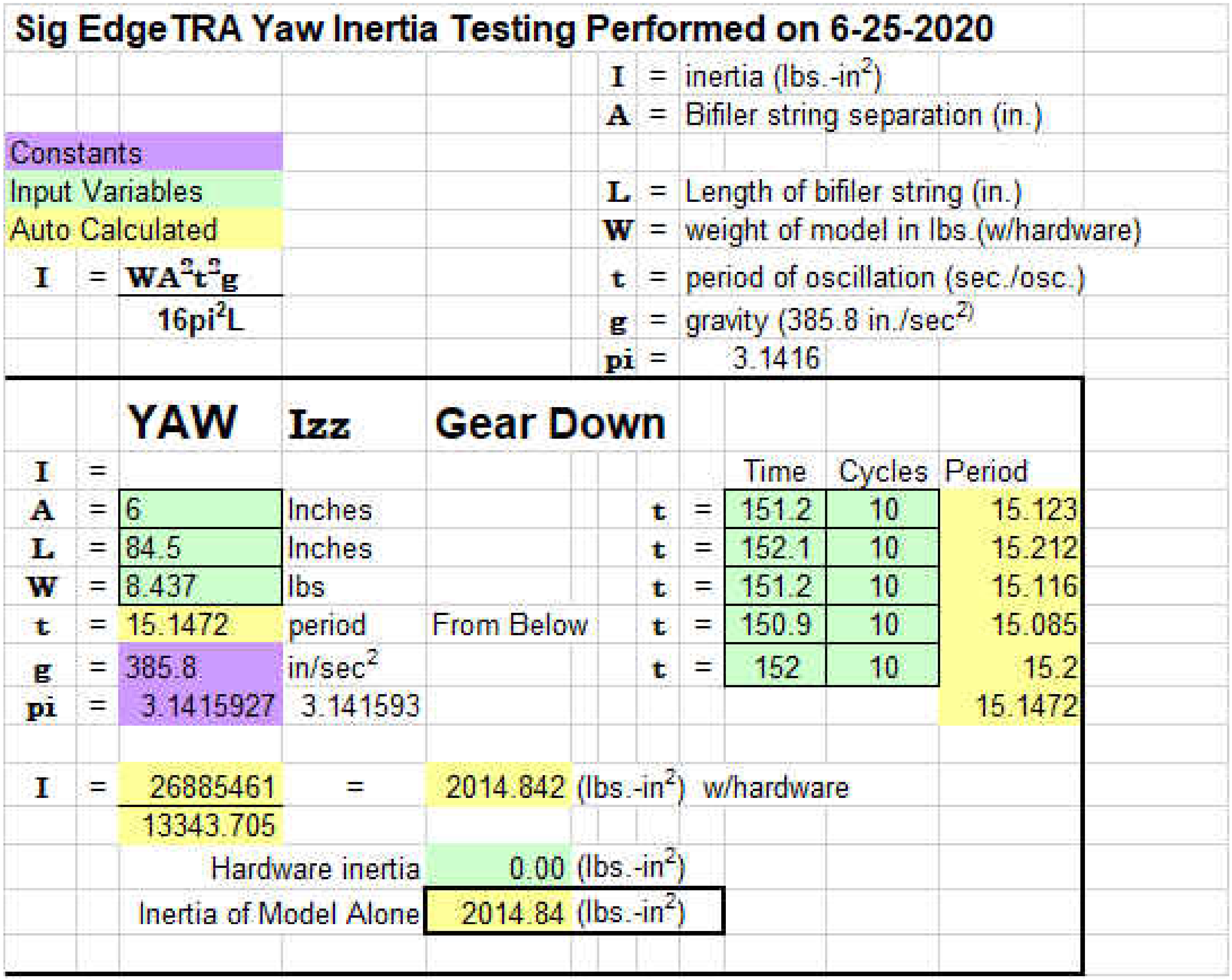
$$I = \frac{WA^2t^2g}{16pi^2L}$$

Pitch Iyy Gear Down

I	=						Time	Cycles	Period
A	=	6.125	Inches		**t**	=	120.5	10	12.053
L	=	63.25	Inches		**t**	=	122.6	10	12.264
W	=	8.437	lbs		**t**	=	121	10	12.1
t	=	12.1448	period	From Below	**t**	=	121.5	10	12.147
g	=	385.8	in/sec^2		**t**	=	121.6	10	12.16
pi	=	3.1415927	3.141593						12.1448

$$I = \frac{18011223}{9988.0397} = 1803.279 \text{ (lbs.-in}^2\text{)}$$ w/hardware

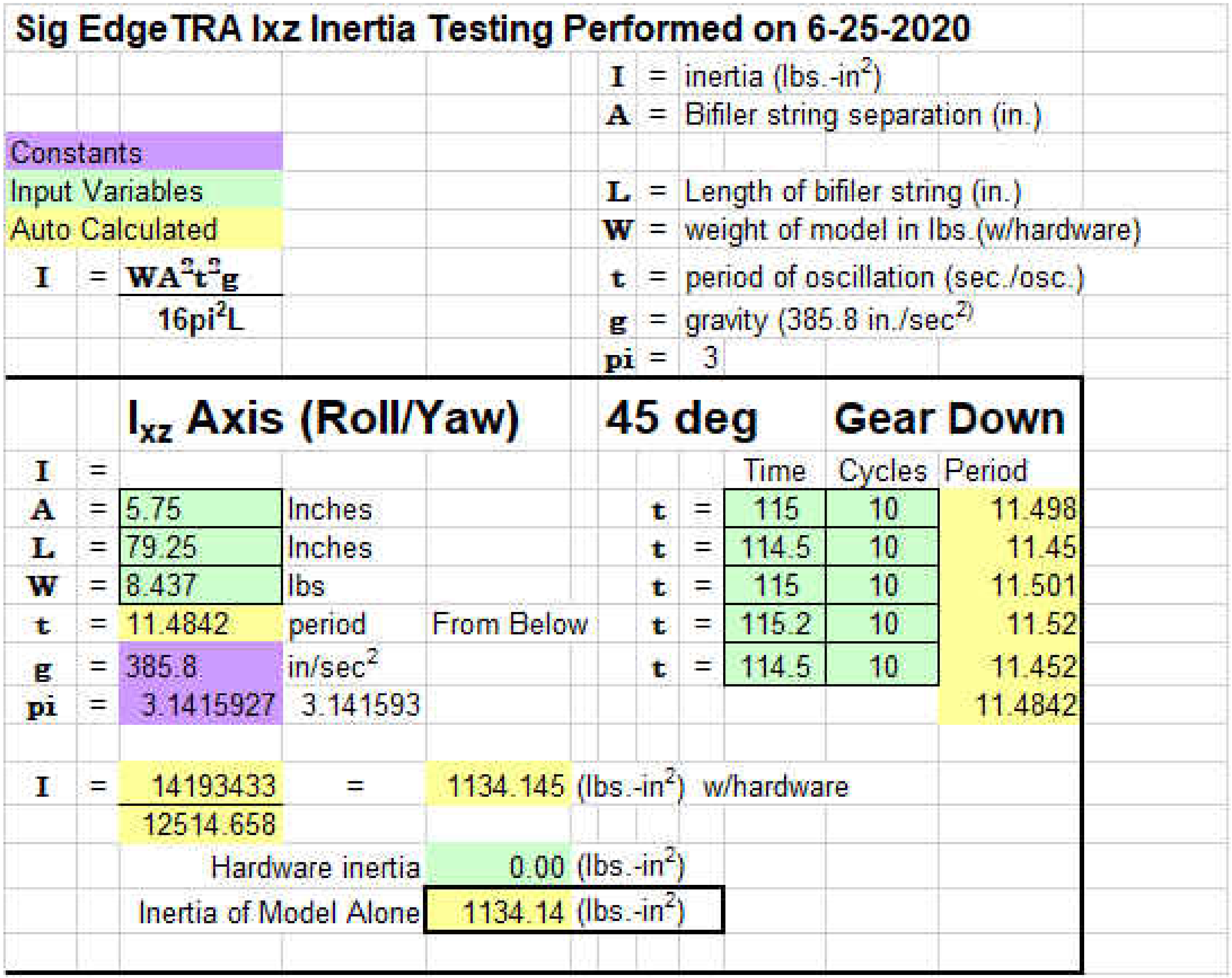
Hardware inertia	0.00	(lbs.-in^2)
Inertia of Model Alone	1803.28	(lbs.-in^2)



Sig EdgeTRA Yaw Inertia Testing Performed on 6-25-2020

		I	=	inertia (lbs.-in^2)
		A	=	Bifiler string separation (in.)
Constants				
Input Variables		**L**	=	Length of bifiler string (in.)
Auto Calculated		**W**	=	weight of model in lbs.(w/hardware)
I =	$\frac{WA^2t^2g}{16pi^2L}$	**t**	=	period of oscillation (sec./osc.)
		g	=	gravity (385.8 in./sec$^{2)}$
		pi	=	3.1416

		YAW	**Izz**	**Gear Down**					
I	=						Time	Cycles	Period
A	=	6	Inches		**t**	=	151.2	10	15.123
L	=	84.5	Inches		**t**	=	152.1	10	15.212
W	=	8.437	lbs		**t**	=	151.2	10	15.116
t	=	15.1472	period	From Below	**t**	=	150.9	10	15.085
g	=	385.8	in/sec^2		**t**	=	152	10	15.2
pi	=	3.1415927	3.141593						15.1472
I	=	26885461	=	2014.842	(lbs.-in^2)	w/hardware			
		13343.705							
			Hardware inertia	0.00	(lbs.-in^2)				
			Inertia of Model Alone	2014.84	(lbs.-in^2)				



Sig Edge TRA Ixz Inertia Testing Performed on 6-25-2020

Constants		**I**	=	inertia (lbs.-in^2)
Input Variables		**A**	=	Bifiler string separation (in.)
Auto Calculated		**L**	=	Length of bifiler string (in.)
		W	=	weight of model in lbs.(w/hardware)
I	= $\frac{WA^2t^2g}{16pi^2L}$	**t**	=	period of oscillation (sec./osc.)
		g	=	gravity (385.8 in./sec^2)
		pi	=	3

I_{xz} Axis (Roll/Yaw)   45 deg   Gear Down

I	=						Time	Cycles	Period
A	=	5.75	Inches		**t**	=	115	10	11.498
L	=	79.25	Inches		**t**	=	114.5	10	11.45
W	=	8.437	lbs		**t**	=	115	10	11.501
t	=	11.4842	period	From Below	**t**	=	115.2	10	11.52
g	=	385.8	in/sec^2		**t**	=	114.5	10	11.452
pi	=	3.1415927	3.141593						11.4842

I	=	$\frac{14193433}{12514.658}$	=	1134.145	(lbs.-in^2) w/hardware
		Hardware inertia		0.00	(lbs.-in^2)
		Inertia of Model Alone		1134.14	(lbs.-in^2)

VITA

Brian Edward Duvall was born in Charlottesville, Virginia, on January 7, 1991. During his high school years, he became interested in dynamics and decided to continue his education at Old Dominion University (ODU) to learn more. At ODU, Brian completed the curriculum for a B.S. degree in mechanical engineering. During the B.S. mechanical engineering program, he became interested in model aircraft, which led him to pursue an M.S. degree in aerospace engineering. Following his M.S. degree in August of 2016, Brian continued at ODU to obtain his Ph.D. in aerospace engineering to further his UAV technology knowledge. While pursuing his M.S. and Ph.D. degrees, he was able to help the Old Dominion University Society of Automotive Engineering Aero East team compete in a design-build and fly competition. This allowed him to further his knowledge of aircraft design and apply what he learned in the classroom to real-life applications. Brian received his Ph.D. degree in aerospace engineering in December 2020.