67th International Astronautical Congress (IAC), Guadalajara, Mexico, 26-30 September 2016.

IAC-16-D1.IP.2x32540

Development of a Fault Protection Architecture Based Upon State Machines

Peter Z. Schulte[a]*, David A. Spencer[b], Neil G. Smith[c], Matthew F. McCabe[d]

[a] *Graduate Research Assistant, Space Systems Design Laboratory, Georgia Institute of Technology, Atlanta, Georgia, United States of America*, pzschulte@gatech.edu
[b] *Associate Professor, Space Flight Projects Laboratory, Purdue University, West Lafayette, Indiana, United States of America*, spencer@purdue.edu
[c] *Research Scientist, Visual Computing Center, King Abdullah University of Science and Technology, Thuwal, Saudi Arabia*, neil.smith@kaust.edu.sa
[d] *Associate Professor, Water Desalination and Reuse Center, King Abdullah University of Science and Technology, Thuwal, Saudi Arabia*, matthew.mccabe@kaust.edu.sa
* Corresponding Author

Abstract

This paper describes an advance in the state-of-the-art of aerospace vehicle fault protection through development of an architecture that utilizes state machines for Fault Detection, Isolation, and Recovery. Through the application of state machine logic, the architecture actively responds to hardware and software faults, allowing autonomous recovery to a safe state. The study leverages a MATLAB/Simulink six degree-of-freedom simulation environment, allowing the evaluation of the fault detection algorithms in flight-like mission scenarios. The modularity of the simulation environment allows the investigator to define the sensor/actuator suite and software modules to test various combinations of algorithms and hardware models.

Within Simulink, a tool called Stateflow is used to implement complex logical relationships by using state charts, also known as state machines, to represent the current state of different spacecraft hardware or software components. The fault protection architecture is developed as a Stateflow block that receives measurements of state variables from spacecraft software and hardware components in Simulink to decide the current state of the system. Based on that state, the fault protection algorithms determine if any faults are present (detection), determine the type of fault and likely location (isolation), and command actions to contain or prevent further faults (recovery). Outputs from the fault protection Stateflow charts issue commands back to the spacecraft software and hardware models, allowing an automated response to spacecraft faults.

This fault protection architecture is designed to be generic, modular, and portable to flight software. The simulation environment allows setting parameters such as physical dimensions and trajectory, is applicable to a multitude of possible mission scenarios and allows alternate configurations, such as multiple cooperative or non-cooperative vehicles. The visual block diagram environment offered by MATLAB/Simulink can be reconfigured to test many combinations of software and hardware components. Finally, the capability to easily convert into flight software code (i.e. autocoding) is available through the MATLAB/Simulink platform.

The study advances the state-of-the-art in fault protection and builds on previous work by bringing together capabilities including Stateflow decision logic, autocoding to flight software, and model-based design into a single generic, modular architecture that is portable to embedded systems. The resulting architecture is intended to be broadly applicable for aerospace missions, advancing flight system capabilities for automated mission operations.

Keywords: fault protection, autonomous systems, state machines, decision logic, fault detection isolation & recovery

Acronyms/Abbreviations

6DOF	Six-Degree-of-Freedom
ASCII	American Standard Code for Information Interchange
ESC	Electronic Speed Control
FDIR	Fault Detection, Isolation, & Recovery
FSW	Flight Software
GN&C	Guidance, Navigation, & Control
HALO	Hydrology Agriculture and Land Observation
HITL	Hardware-in-the-Loop
KAUST	King Abdullah University of Science and Technology
KNN	K-Nearest Neighbors
MATLAB	Mathematics Laboratory
NASA	National Aeronautics & Space Administration
ProxOps	Proximity Operations
TABS	Technology Area Breakdown Structure
UAV	Unmanned Aerial Vehicle
V&V	Verification and Validation

1. Introduction

The capability to recover gracefully from hardware faults or algorithm convergence issues is critical for many aerospace applications, particularly for missions involving proximity operations (ProxOps), where multiple vehicles are operating at close range. Previous ProxOps missions have experienced faults that resulted in a failure to meet mission objectives. For example, NASA's Demonstration of Autonomous Rendezvous Technology experienced a complete mission failure when it collided with its target spacecraft during automated operations due to software errors that led to an inaccurate range estimation [1].

In the development of aerospace systems, verification and validation (V&V) are often focused on demonstrating that software algorithms and systems will work under nominal conditions. The robustness of the system to off-nominal scenarios is often not tested. Even when system robustness is evaluated, it is difficult to evaluate all possible failure modes. As more missions undertake autonomous operations, there is an increased need for real-time detection and correction of failures through Fault Detection, Isolation, and Recovery (FDIR). These capabilities are especially necessary for time-critical operations such as rendezvous and ProxOps.

1.1 Previous Work

Over the last few years, the development of FDIR for space missions has advanced significantly. A typical aerospace FDIR system is "a smart embedded system that is able to react to some know[n] events and to select a decision among a predefined set" [2]. Currently, the state-of-the-art in spacecraft FDIR involves using a set of rules or conditions that are checked against telemetry, with preprogramed responses that are executed when one of these rules is violated. For example, if a parameter persistently exceeds its expected range, a signal is sent to ground operators to warn them. Also, space mission teams usually develop custom FDIR systems from scratch for each new mission based upon the specific needs and requirements of the mission.

The use of model-based fault protection has been explored and implemented in some scenarios, but it has not been widely adopted for various reasons. In recent years, several space mission teams have made use of the Stateflow toolbox within MATLAB/Simulink to develop FDIR algorithms and autocode those algorithms into Flight Software (FSW) including Deep Space 1 [3], and Deep Impact [4]. Recently, NASA's Johnson Space Center has used MATLAB/Simulink, including Stateflow, to develop algorithms for Guidance, Navigation, and Control (GN&C), which are later autocoded into FSW [5]. Stateflow has also been used to evaluate errors in FDIR algorithms during spacecraft system Verification & Validation (V&V) [6]. Another FDIR architecture developed with Stateflow uses model-based design techniques to bring in V&V earlier in the design cycle by providing a link between subsystem design and FDIR design [7].

One space systems engineering team at the Jet Propulsion Laboratory has begun to analyze the FDIR problem in depth using model-based systems engineering approaches. This team has developed an FDIR architecture using the SysML software [8]. This architecture is used for identifying, evaluating, and managing failure modes during the design and V&V phases, though the implementation of FDIR for FSW does not stem directly from the architecture.

1.2 Advances in State-of-the-Art

The study presented here advances the state-of-the-art in FDIR and builds on previous work by bringing together capabilities including Stateflow decision logic, autocoding to FSW, and model-based design into a single generic, modular FDIR architecture that is portable to FSW. Most previous FDIR studies have involved large, high-resource missions with custom-built FDIR, while the proposed FDIR architecture is designed with a focus on small aerospace vehicles and will be applicable to a wide variety of missions. Stateflow logic allows complex decisions to be made in a hierarchical way where conditions and logical states in individual spacecraft software components, FDIR algorithms, and higher level "master" FSW mode logic all influence one another. Also, various initial conditions, environmental scenarios, and physical vehicle properties can be re-defined simply and easily in a MATLAB initialization script. The architecture also allows alternate configurations that enable testing of various scenarios.

Numeric software algorithms such as Kalman Filters may or may not converge, depending on a variety of factors. For autonomous systems, divergent algorithms can lead to mission-critical failures if not detected and corrected. Many FDIR methods have been developed for software failures such as these, and the architecture developed in this work enables these algorithms to be rigorously tested and implemented. Numerous FDIR algorithms developed in academic environments never move from concept design to flight test implementation [2]. The architecture developed in this study allows these algorithms to be further developed in an integrated environment that closely models the behavior of aerospace vehicles in relevant environments. The proposed software environment also features an autocoding capability to convert integrated software modules and FDIR algorithms directly to FSW for further testing in hardware-in-the-loop (HITL) and flight applications. This will greatly facilitate the transition of new FDIR algorithms from concept design to implementation.

The primary area of applicability of the proposed study to the NASA Technology Area Breakdown Structure (TABS) is element 4.5.1 System Health Management under section 4.5 System Level Autonomy within Technology Area 04: Robotics and Autonomous Systems. System health management "monitors, predicts, detects, and diagnoses faults and accommodates or mitigates the effects either on-board or through telemetry processing on the ground" [9]. The proposed FDIR architecture results in on-board real-time system health management software and will address many of the desired technical capabilities of TABS element 4.5.1. For example, the complex logic enabled by Stateflow charts allows the FDIR architecture to include prognostic and diagnostic components as an integral part of the system. The logic is also able to take complicated vehicle states into account to avoid false positives when faults are not present and false negatives when faults are present. It may even be used to anticipate faults and adapt to new situations that do not have pre-programmed responses. Finally, this study advances paradigm-shifting state machine-based approaches for FDIR that can be easily transitioned to FSW and validated using HITL testing.

1.3 Paper Organization

The paper is organized in the following manner. Section 1 introduces background for the topic, Section 2 presents the overall concept of the FDIR Architecture, Section 3 presents a proof-of-concept of the FDIR Architecture, and Section 4 presents future work.

2. FDIR Architecture Concept

The concept discussed here leverages the development of a Six-Degree-of-Freedom (6DOF) simulation environment for the Prox-1 small satellite mission at Georgia Tech. The original purpose of this MATLAB/Simulink platform was for Guidance, Navigation, and Control (GN&C) algorithm integration and testing [10,11], but it can be adapted for the development of a more general FDIR system. Functionality is added to the simulation environment that can be applied generally to aerospace mission scenarios to test a variety of fault detection algorithms and mission architectures. For example, modularity of the simulation environment allows the investigator to replace the current sensor/actuator suite and software modules to test various combinations of state-of-the-art algorithms and hardware models.

2.1 Development Environment

Within MATLAB/Simulink, the Stateflow toolbox can be used to implement complex logical relationships. Stateflow is a simple graphical tool using state charts, also known as state machines, to represent the current state of different vehicle hardware or software components [12]. These charts can be integrated with larger simulations in Simulink using a Stateflow block, with variables being input to the block to influence the current state of the chart and variables being output from the block to influence the behavior of other blocks based on that state. These charts can be very simple, representing only a few possibilities, or they can involve complicated nested sets of states. Stateflow animates the status of the state charts during simulation so that the developer can monitor the simulation in real time for debugging and confirmation that the chart is properly constructed. An example Stateflow chart representing a thruster controller [10] is shown in Figure 1. This chart simply contains three states (Startup, ThrustOff, ThrustOn), transition conditions between the states, and an embedded function written in MATLAB syntax. The chart determines whether the thruster should be on or off based on whether the controller has received a command to fire ("ready"), the amount of time commanded, and the fuel margin (determined by the output of the MATLAB function). The Stateflow chart is integrated within a Simulink simulation as a Stateflow block with inputs and outputs, as shown in Figure 1.

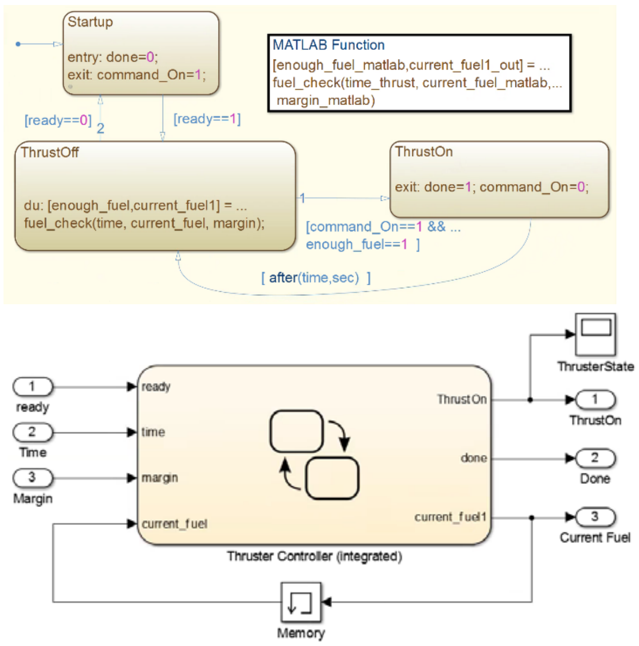


Fig. 1. Sample Stateflow chart representing a thruster controller (top); Demonstration of Stateflow block integration in Simulink (bottom) [10]

The FDIR architecture takes data from the vehicle which is used to determine the likely state of the vehicle. This state can be classified as either "fault" or "no fault" based on how the decision logic is structured. Future versions of the architecture may also be able to isolate a fault from an unknown source and perform preventative actions to recover from the failure before it becomes mission-critical. Outputs from the architecture can either send commands to the vehicle autonomously or notify ground operators to take corrective action.

2.2 FDIR Architecture Requirements

High-level requirements have been identified to guide the development of this FDIR architecture: it should be generic, modular, and portable to FSW. A generic architecture is not limited to any specific mission. The simulation environment should allow setting vehicle parameters such as physical dimensions and trajectory. It should be applicable to a multitude of possible mission scenarios and allow alternate configurations, such as individual vehicles, or multiple cooperative or non-cooperative vehicles. The architecture should also contain generic modules for commonly used components such as sensors and actuators. A modular FDIR architecture allows components to be easily rearranged. The visual block diagram environment offered by MATLAB/Simulink can be altered and reconfigured easily and allows for testing of many combinations of software modules and hardware components.

The FDIR architecture should allow rapid transition from development to flight. A flight-like FDIR architecture should accurately model in-flight conditions of actual vehicles and missions. It should contain environment and hardware models with configurable settings. In addition, the computational requirements of the architecture should match the capability available on flight processors. The architecture should also have the ability to make the kinds of complex decisions normally required for autonomous FSW and should be evaluated by testing its response to realistic stochastic conditions rather than “canned” scenarios. It should be well integrated with other hardware and software components, allowing new components to be quickly evaluated. Finally, the capability to easily convert the architecture into FSW code (i.e. autocoding) is highly desirable.

V&V of the FDIR architecture will assess the capability to meet the following key requirements. First, it should detect and possibly correct in real-time component, subsystem, and system-level software and hardware failures. These failures include sensor/actuator failures, errors, or degradation, improper controller gain settings, non-convergence of GN&C algorithms, and software or hosting hardware (i.e. processor) failures.

Secondly, the architecture should detect and avoid mission-level failure modes, such as vehicle collision or uncontrolled behavior that renders the mission objectives unattainable. Thirdly, it should utilize complex decision logic in Stateflow to select the best course of action when multiple options exist. Finally, it should demonstrate fault protection logic that allows the system to avoid aborts by responding to correctible errors in real-time and still meet mission objectives.

3. Proof-of-Concept: UAV Nervous System

FalconViz is a start-up company based out of the King Abdullah University of Science and Technology (KAUST) in Saudi Arabia. It was founded in 2015 by two research faculty and a PhD student at KAUST: Dr. Neil Smith, Dr. Mohamad Shalaby, and Luca Passone. FalconViz designs and flies custom unmanned aerial vehicles (UAVs) for a variety of applications such as aerial surveying & mapping, inspection & monitoring, and surveillance. The company also collaborates with other research groups at KAUST such as the Hydrology, Agriculture and Land Observation (HALO) group led by Dr. Matthew McCabe. The HALO group uses modelling, remote sensing, and in-situ measurements to better understand elements such as water usage, crop health, and regional climate conditions. One effort of the HALO group involves the use of UAVs to capture thermal and hyperspectral imagery of desert agricultural plots.

At KAUST during Summer 2016, a FalconViz hexacopter (six propellers) shown in Figure 2 was used as a proof-of-concept testbed for the FDIR software architecture described here. Two specific failures were addressed as a starting point: unbalanced propellers (leading to excess vibration) and overheating components. Detecting these failures provides a more reliable vehicle for performing aerial surveys and other tasks with FalconViz UAVs.

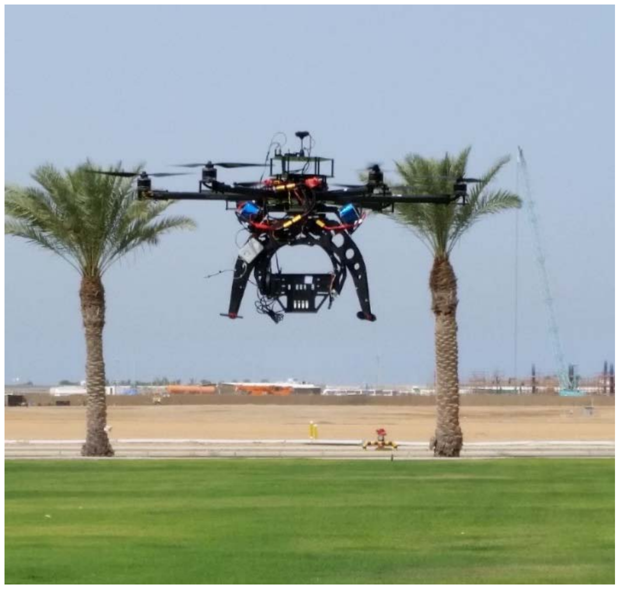


Fig. 2. FalconViz hexacopter in flight

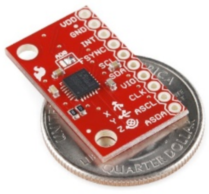
3.1 Vibration Detection Hardware

Unbalanced propellers cause excess vibration and can lead to screws coming loose and potential crashes.

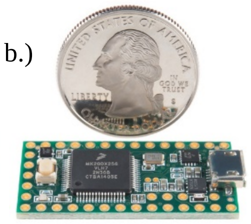
Vibration detection is accomplished by evaluating accelerometer data measured from the arms of the UAV that house the propellers. A machine-learning algorithm determines the health of the system from the data. If the propellers are unbalanced, then there will be much more vibration in the system. Once it is trained and validated on the ground, the machine-learning model then identifies the health of the system from live data onboard the UAV. These outputs are sent into the state-based FDIR architecture in Stateflow.

A SparkFun Triple Axis Accelerometer and Gyro Breakout – MPU-6050 [13], shown in Figure 3a, is installed on one arm of the hexacopter. Data is collected via a microcontroller programmed with Arduino protocols called the Teensy 3.2 [14], shown in Figure 3b. The Teensy is then connected to a MeegoPad T02 compute stick [15], shown in Figure 3c, via USB.

a.)



b.)



c.)

Fig. 3. (a) SparkFun Triple Axis Accelerometer & Gyro Breakout – MPU-6050 [13]; (b) Teensy 3.2 [14]; (c) MeegoPad compute stick [15]

The shrink-wrapped MPU-6050 breakout board is mounted just below the propeller motor, as shown in Figure 4. The Teensy is installed on a SparkFun Teensy Arduino Shield Adapter [16] and connected to the MPU-6050 and other components via jumper cables and custom harnesses, as shown in Figure 5. A Simulink model run in Windows on the MeegoPad records data from the accelerometer and feeds it through a MATLAB supervised machine-learning classification algorithm called K-nearest neighbors (KNN) [17] to determine if the propeller is "balanced" or not. The propeller is unbalanced by adding a few pieces of electrical tape on one side, as shown in Figure 6.

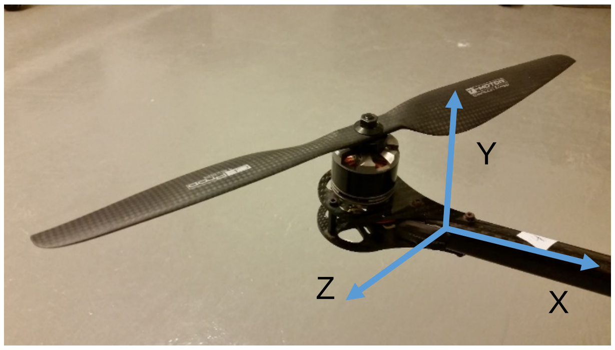


Fig. 4. Shrink-wrapped accelerometer breakout board (MPU-6050) installed on hexacopter arm, with sensor coordinate axes indicated.

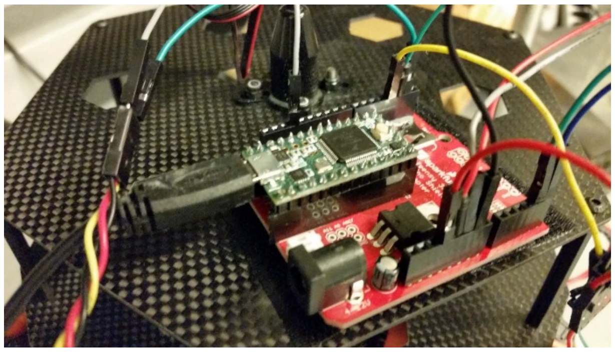


Fig. 5: Teensy installed on Arduino Shield Adapter [16] with USB and jumper cable connections

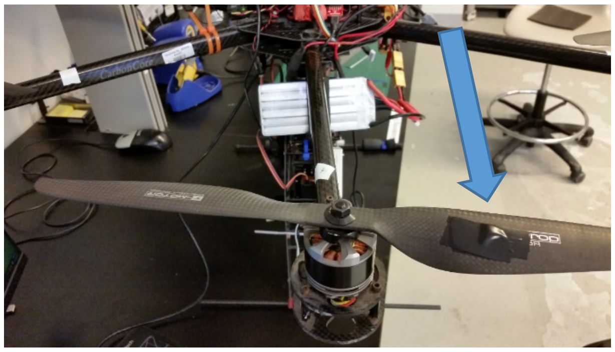


Fig. 6. Hexacopter propeller unbalanced by adding electrical tape.

3.2 Vibration Detection Software

Flight test data is captured for both an unbalanced propeller (with tape) and a balanced propeller (without tape) and is used to train the KNN classification model in MATLAB on the ground. The raw data used to train the KNN model is shown in Figure 7. Data recording begins when the Simulink model is started on the lab bench. The copter then must be carried outside before flight, so the data is cropped in post-processing to start at the beginning of the flight.

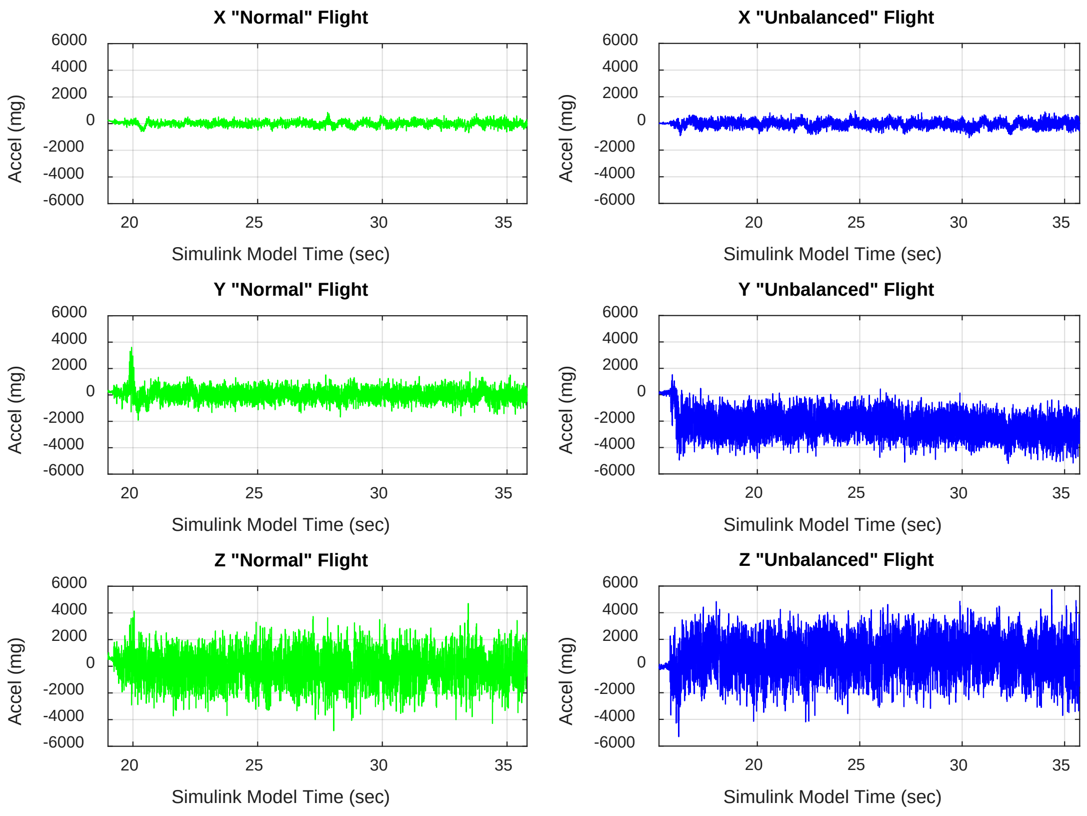


Fig. 7. Training data for KNN classification model

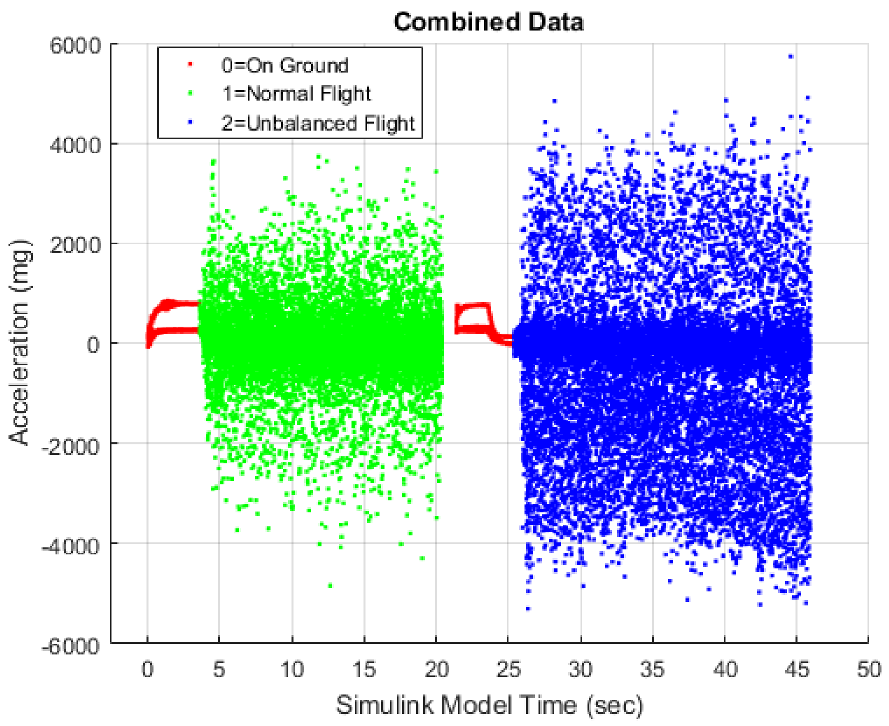


Fig. 8. Combined and labelled data for KNN classification model training; note that at this point there is no detection being done by the algorithm because the labels are set manually by the user to seed the KNN classification process

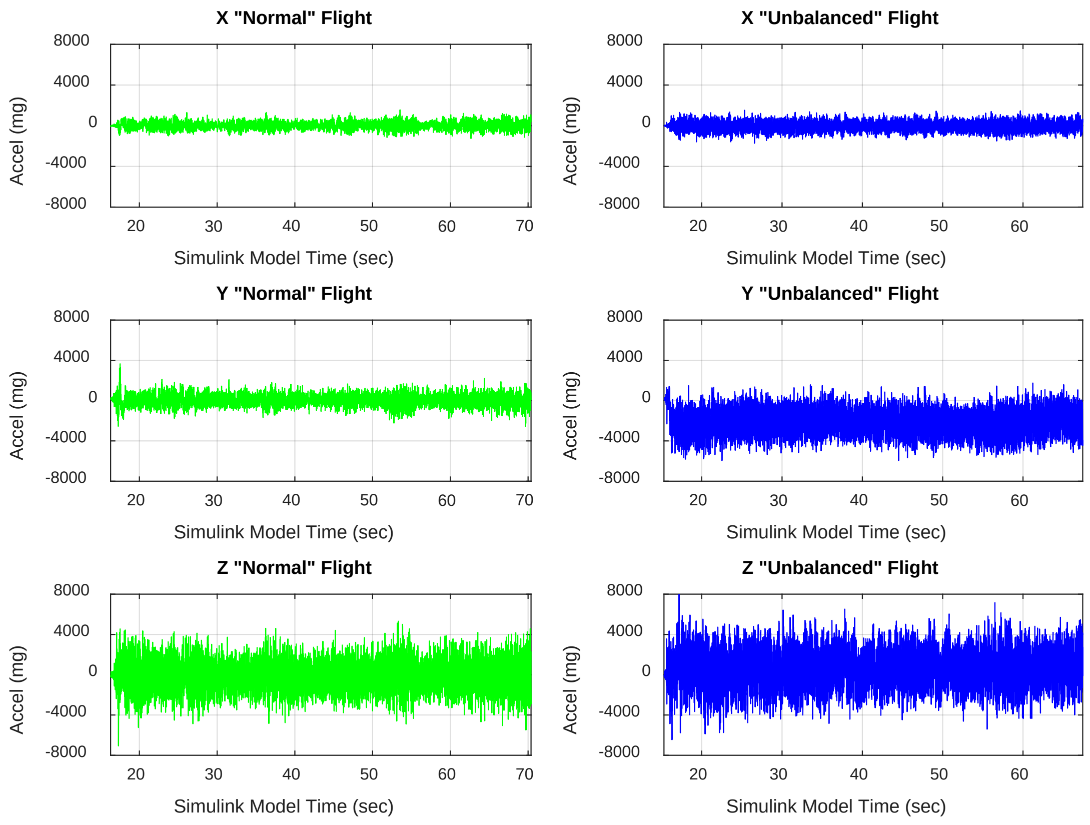


Fig. 9. Validation Data for KNN Classification Model

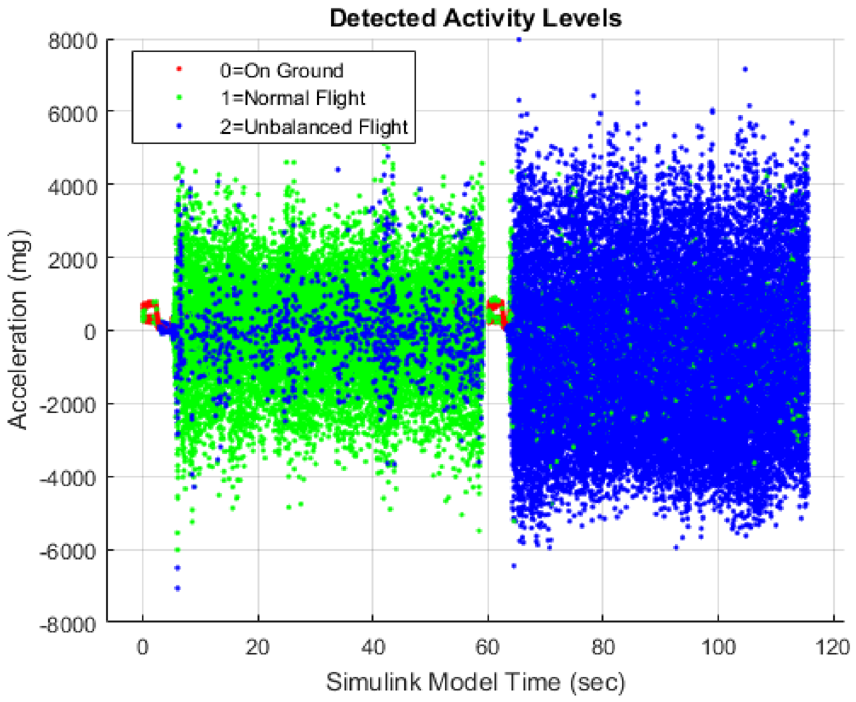


Fig. 10. Detected activity levels from KNN classification model validation data

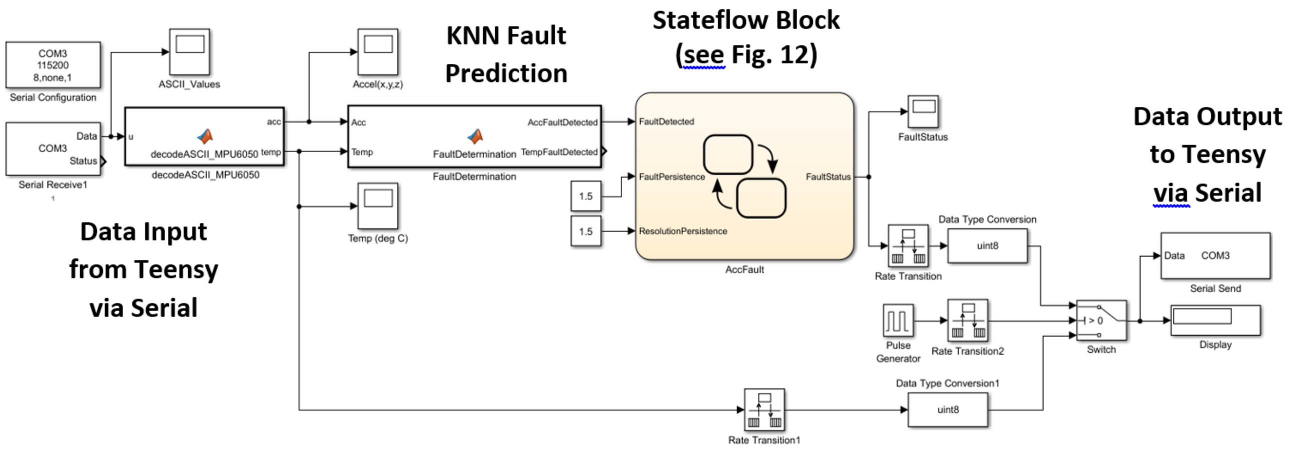


Fig. 11. Simulink diagram for UAV nervous system

Note that the timescale is based on the Simulink model's internal time, which is not synchronized with real time. Actual flights lasted around two minutes, while in Simulink model time they lasted 40 sec. The timescale difference occurs because Simulink is not set up for "real time" operation. This does not impact system performance because input data are immediately processed in Simulink as they are received based on number of samples, not time duration. Figure 7 shows the unbalanced flight has a higher magnitude in y and z and y shifts to the negative region. However, this information is not provided to the KNN training algorithm. Instead, the raw data from the two flights is combined and manually labelled by the user as shown in Figure 8. Combined raw data for each axis (x,y,z) and assigned labels are fed into the KNN training algorithm in MATLAB. After training, the static KNN model is stored for use in flight and is not adapted during flight.

The detection accuracy of the static model is verified with independent flight test validation data. This data is shown in Figure 9, and is captured in the same method as the training data, except that the validation flights were twice as long (about 4 minutes each). The KNN classification detection algorithm then uses the trained KNN model to select the labels for each data point. The validation results, shown in Figure 10, have a detection accuracy of 86.8%.

Once model training and validation is complete, the system is ready for in-flight detection. Data collected by the Teensy real-time is fed into the Simulink model shown in Figure 11 via a serial connection over USB. The Teensy and serial connections run at 115,200 baud (bits per second). The data is converted from ASCII characters to numerical values by a custom MATLAB function and is saved to memory. It is then fed into the KNN fault detection algorithm, which uses the trained static model to determine if the propeller is balanced or not. The detection is run on 100 samples at a time, and if 50 or more of these samples are classified as "unbalanced" by KNN, then the vibration FaultDetected flag is set to 1; otherwise the flag is set to 0, indicating the propeller is "balanced".

This FaultDetected flag is fed into the Stateflow diagram shown in Figure 12. The Stateflow toolbox within MATLAB/Simulink [18] allows for tracking of nested flowchart states, with transitions indicated by blue arrows with Boolean conditions. If a condition registers as true, the transition will be activated to move from one state (or substate) to another. Default transitions specify the initial conditions of the diagram and are indicated by an arrow beginning at a dot and ending at the initial state or substate. The Stateflow diagram for vibration fault detection in Figure 12 begins with an initial state of "Normal" at the bottom right and an initial substate of "Standby". If FaultDetected is set to 1, the substate within "Normal" transitions to "PotentialFault." If the condition FaultDetected==1 persists for a length of time specified by FaultPersistence, then the state transitions from "Normal" to "Fault". However, if FaultDetected does not remain at 1 for long enough, then the state will remain "Normal" and the substate will return to "Standby". Very similar logic applies for transitioning from "Fault" back to "Normal": the condition FaultDetected==0 must persist for a length of time specified by ResolutionPersistence. The FaultStatus flag is the output signal from the current state of the Stateflow chart, with 0 indicating "Normal" and 1 indicating "Fault". Note that these time durations are tuned to account for the difference between real time and Simulink model time.

Simulink sends the FaultStatus signal back to the Teensy and then on to the FrSky X8R telemetry receiver [19] shown in Figure 13a. The pilot can view the value of FaultStatus on their handheld Taranis X9D radio controller [20], shown in Figure 13b, to indicate whether the propeller is balanced or not (0 or 1). If the variable is set to 1, the controller is programmed to begin beeping. When the variable is set to 0, the controller stops beeping.

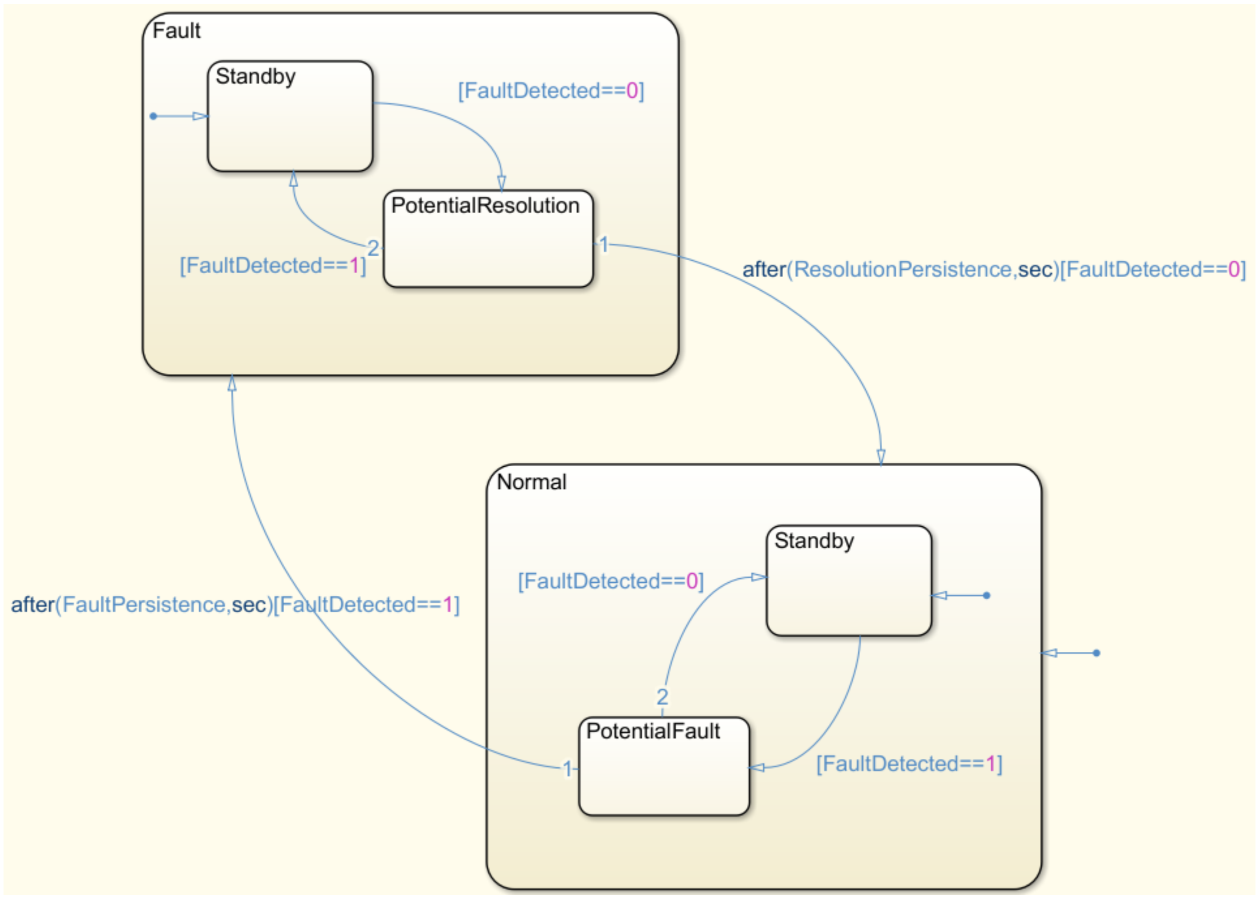


Fig. 12. Stateflow diagram for vibration fault detection

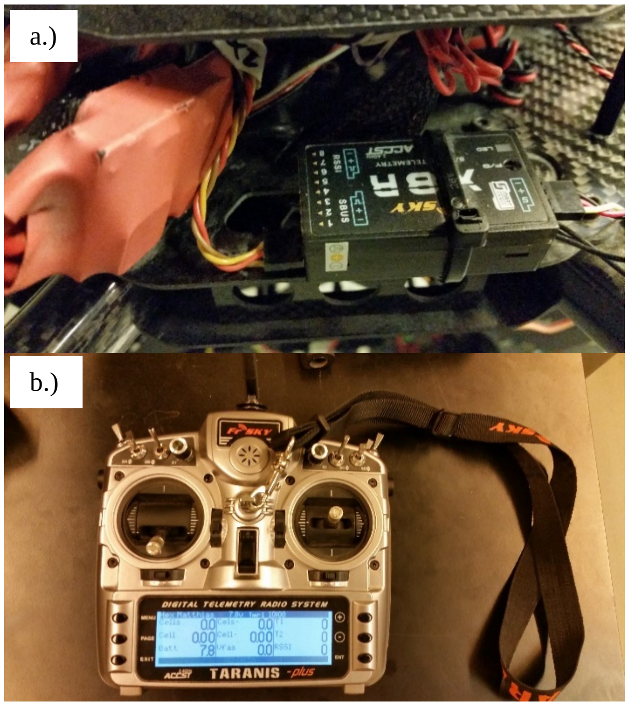


Fig. 13. (a) FrSky X8R telemetry receiver [19] installed on the hexacopter (b) Taranis X9D Plus radio [20]

3.3 Vibration Detection Flight Test Results

The UAV Nervous System has been flight-tested and successfully indicates the state of vibrations during flight. Figure 14 shows data recorded during a final test flight. The top plot shows acceleration from the MPU-6050 and the bottom plot shows the FaultStatus signal output by the Stateflow diagram. The flight begins with the copter on the ground in segment A, and tape is placed on the propeller to unbalance it. The copter takes off and flies with an unbalanced propeller in segment B. The nervous system quickly detects the imbalance and outputs a FaultStatus of 1 at around 10 sec, shortly after segment B begins. During segment C, the copter lands, and the tape is removed to restore the propeller balance. Segment D shows balanced flight, and at around 25 sec, the nervous system detects that balance has been restored and sets FaultStatus to 0. The copter lands again in segment E and tape is added again. During segment E near 35 sec, a FaultStatus of 1 occurs, and since there is no persistent “normal flight” data entering the system, FaultStatus does not return to 0. Unbalanced flight resumes during segment F, and FaultStatus remains at 1. The tape is not well adhered to the propeller and it comes loose and flies off at 40 sec. The copter transitions to balanced flight in segment G,

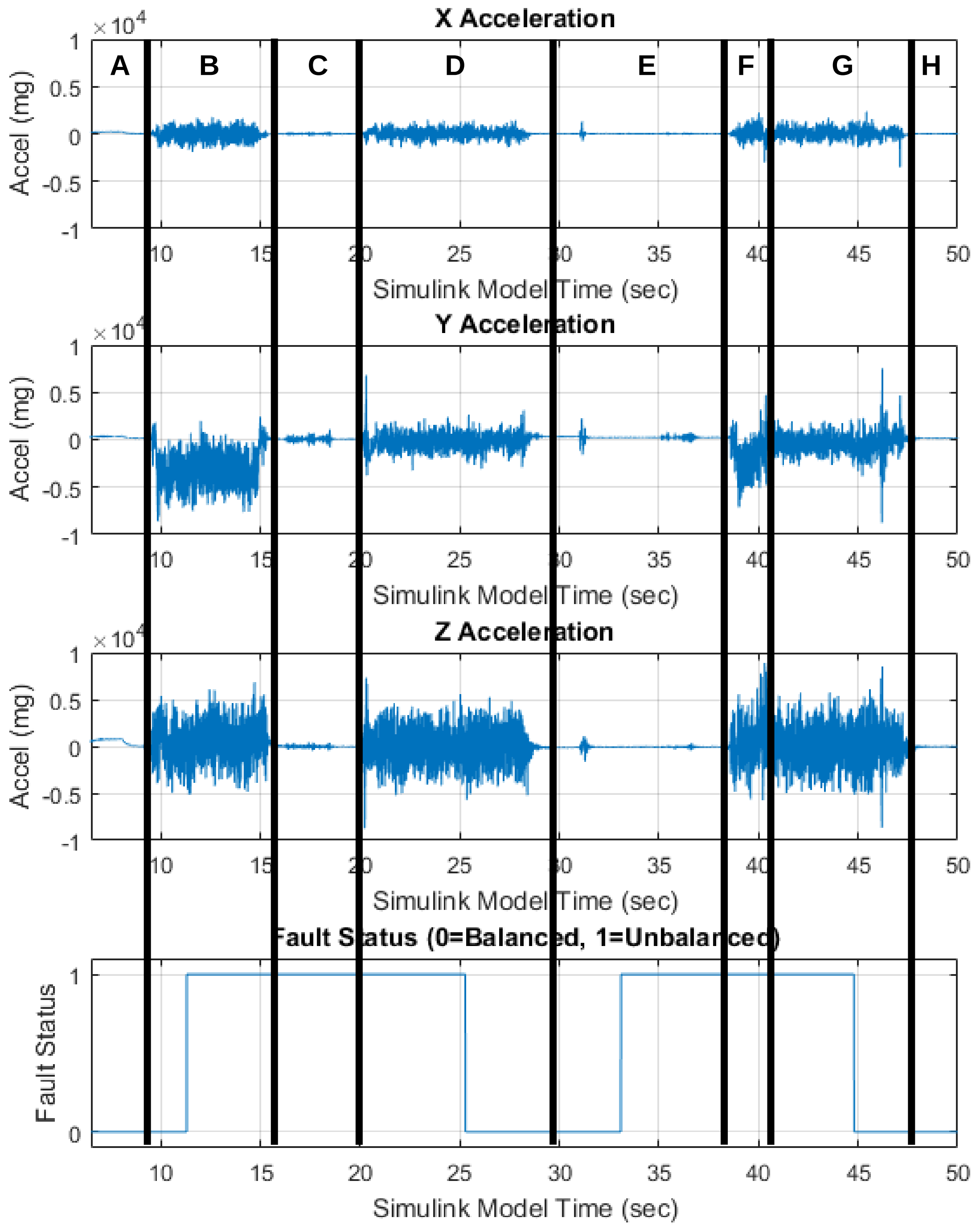


Fig. 14. Flight test data demonstrating successful vibration fault detection

which the nervous system detects around 45 sec, returning FaultStatus to 0. The delays in FaultStatus transitions are expected, as the system is tuned to avoid constant flipping between 0 and 1.

3.4 Overheating Detection

In addition to the vibration sensor, a One Wire Digital Temperature Sensor DS18B20 [21] is used to monitor heating of the motors and electronic speed controls (ESCs). It is important to detect when ESCs overheat because they shut down and can lead to complete hardware failure. Figure 15a shows the DS18B20 sensor by itself and Figure 15b shows it installed on an ESC. The temperature reading in deg C is collected by the Teensy, then downlinked to the Taranis radio via the telemetry receiver. This value is displayed on the radio for the pilot, and the radio is programmed to give a verbal warning ("too high") when the temperature exceeds a predetermined threshold. This threshold is set by the pilot on the radio itself. An example plot of saved temperature values from the DS18B20 is shown in Figure 16.

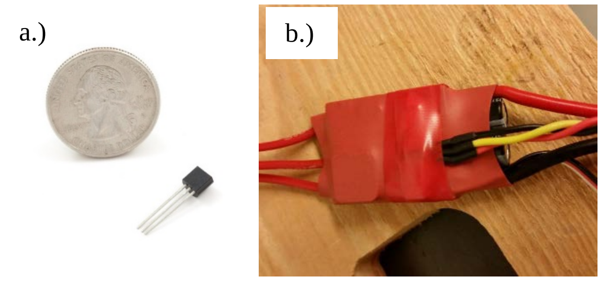


Fig 15. (a) One Wire Digital Temperature Sensor DS18B20 [21] (b) Temperature sensor installed on an electronic speed control (ESC)

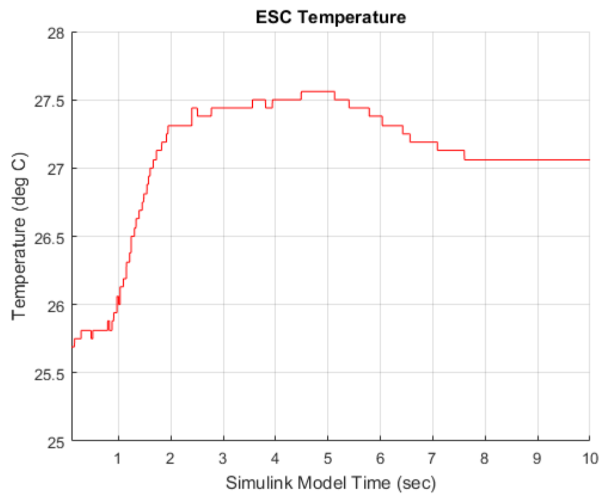


Fig. 16. Example plot of DS18B20 temperature data

4. Future Work

The basic UAV nervous system architecture has been developed and tested with a single acceleration sensor and a single temperature sensor. A next step to expand the nervous system would be to expand the number of sensors. One accelerometer could be placed on each arm of the copter, and vibration fault detection can be performed on all arms independently. Then, if an imbalance is detected, the nervous system can indicate which propeller needs to be balanced. Similarly, temperature sensors can be placed on each ESC and motor and the nervous system can record all temperatures as a function of time and indicate to the pilot which components tend to overheat first.

Further sensors can also be added to expand the set of detectable failures. For example, many times when there is a problem with a UAV the first indication to the operators is an unusual sound. Thus, microphones collecting audio data near each propeller may be able to provide additional warning of faults. Data from microphones could be processed through a machine learning algorithm similarly to the acceleration data. Also, current/voltage sensors can be placed on the ESCs to detect electrical issues before they become mission critical. Examples of other issues/failures on the UAVs that could be addressed by future work include monitoring the magnetic compass (which can malfunction in flight, leading to the regularly reported problem of "fly-aways") and ensuring healthy navigation filters (i.e. GPS position/velocity and attitude determination for roll/pitch/yaw angles and rates). Monitoring navigation variables would require communication with the flight controller, and redundant navigation systems could also aid in detecting navigation errors.

Another necessary update to the nervous system is smoothing out the startup process. Although quite convenient for prototyping and rapid development and testing, running Simulink in Windows onboard the copter is not the most elegant solution. It requires manually starting up Windows and initiating the Simulink model in the lab while connected to a monitor, then carrying the copter outside to begin flying. An intermediate step is to set up an HD video downlink to interact with Simulink in the field, but ultimately it would be desirable to remove the Windows/Simulink component from the system and perform all fault detection and data recording directly on the Teensy (with an SD card shield attached). The Teensy can be fully customized by programming in C, and Simulink has the capability to generate C code via autocoding. If a simple KNN classification detection algorithm is implemented in C, it can be integrated with autocode from Simulink and sensor interface code directly on the Teensy. This would streamline the process for using the nervous system and make it much easier to seamlessly integrate it with a copter for any mission.

In addition to upgrading the UAV Nervous System, the FDIR architecture needs to be generalized and modularized in order to meet the requirements defined in Section 2.2. The Simulink model, especially the Stateflow chart, can easily be reconfigured for a general system that is applicable to UAV, satellite, and many other aerospace vehicle applications.

Another proposed method for V&V of the FDIR architecture for space environments is through a HITL test platform designed to simulate a small satellite during ProxOps. This test platform will contain typical small satellite sensor and actuator hardware as well as flight processors and can be attached to the Simulink simulation using the Simulink Real Time toolbox from MATLAB. Simulink Real Time allows a simulation to be run on a desktop computer while accepting inputs from sensor hardware and sending outputs to actuator hardware, with both inputs and outputs passing through the flight processor. The FDIR algorithms can also be autocoded from MATLAB/Simulink into C and integrated with FSW code to run directly on the flight processor during testing. This platform will be used verify that the FDIR architecture is properly designed for space applications and validate that it works as intended through testing in various scenarios with flight-like hardware and software.

Acknowledgements

This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-1148903. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. FalconViz and King Abdullah University of Science and Technology (KAUST) provided funding support and technical guidance for development of the UAV Nervous System.

References

[1] N. Dennehy, J.R. Carpenter, NESC Review of Demonstration of Autonomous Rendezvous Technology (DART) Mission Mishap Investigation Board Review (MIB), NASA Engineering and Safety Center Report, RP-06-119, Dec. 2006, http://www.nasa.gov/pdf/167813main_RP-06-119_05-020-E_DART_Report_Final_Dec_27.pdf, (accessed 10/9/15).

[2] A. Zolghadri, The Challenge of Advanced Model-Based FDIR Techniques for Aerospace Systems: The 2011 Situation, Progress in Flight Dynamics, Guidance, Navigation, Control, Fault Detection, and Avionics. 6 (Dec. 2013) 231-248.

[3] N.F. Rouquette, T. Neilson, G. Chen, The 13th Technology of Deep Space One, IEEE Aerospace Conference, Aspen, CO, Mar. 1999.

[4] P.J. Pingree, et. al., Validation of Mission Critical Software Design and Implementation Using Model Checking, IEEE Digital Avionics Systems Conference, Oct. 2002.

[5] M.C. Jackson, J.R. Henry, Orion GN&C Model Based Development: Experience and Lessons Learned, AIAA-2012-5036, AIAA Guidance, Navigation, and Control Conference, Minneapolis, Minnesota, August 2012.

[6] M. Aguilar, Fault Management Using Model Based System Engineering (MBSE) Tools and Techniques, NASA Spacecraft Fault Management Workshop, Sept. 2011, http://www.nasa.gov/pdf/637605main_day_1-michael_aguilar.pdf. (accessed 10/8/15).

[7] A.M. Homar, AOCS Fault Detection, Isolation and Recovery: A Model-Based Dynamic Verification and Validation Approach, Master's Thesis, Department of Computer Science, Electrical and Space Engineering, Lulea University of Technology, Sept. 2014.

[8] J. Day, A. Murray, P. Meakin, Toward a Model-Based Approach to Flight System Fault Protection, IEEE International Conference for Aerospace, Big Sky, MT, Mar. 2012.

[9] "2015 NASA Technology Roadmaps - TA 4: Robotics and Autonomous Systems," National Aeronautics and Space Administration, July 2015. [http://www.nasa.gov/sites/default/files/atoms/files/2015_nasa_technology_roadmaps_ta_4_robotics_and_autonomous_systems_final.pdf. Accessed 10/9/15.]

[10] P.Z. Schulte, D.A. Spencer, Development of an Integrated Spacecraft Guidance, Navigation, & Control Subsystem for Automated Proximity Operations, Acta Astronautica, 118 (Jan-Feb 2016), 168-186, doi:10.1016/j.actaastro.2015.10.010.

[11] D.A. Spencer, S.B. Chait, P.Z. Schulte, K.J. Okseniuk, M. Veto, Automated Trajectory Control for On-Orbit Inspection in the Prox-1 Mission, Journal of Spacecraft and Rockets, accepted May 2016.

[12] M.D. Ingham, R.D. Rasmussen, M.B. Bennett, and A.C. Moncada, Engineering Complex Embedded Systems with State Analysis and the Mission Data System, Journal of Aerospace Computing, Information, and Communication, 2 (Dec. 2005).

[13] SparkFun Triple Axis Accelerometer and Gyro Breakout - MPU-6050, SparkFun Electronics, https://www.sparkfun.com/products/11028, (accessed 7/6/16).

[14] Teensy 3.2, SparkFun Electronics, https://www.sparkfun.com/products/13736, (accessed 7/6/16).

[15] MeegoPad T02 Second Generation Intel Windows TV Stick, http://www.x86pad.com/t02.html, (accessed 7/6/16).

[16] Teensy Arduino Shield Adapter, SparkFun Electronics, https://www.sparkfun.com/products/13288, (accessed 7/6/16).

[17] Predict k-nearest neighbor classification – MATLAB, The Mathworks, Inc., http://www.mathworks.com/help/stats/classificationknn.predict.html, (accessed 7/6/16).

[18] Stateflow Documentation, The Mathworks, Inc., http://www.mathworks.com/help/stateflow/index.html, (accessed 7/6/16).

[19] X8R-products, FrSky Electronic Co., Ltd., http://www.frsky-rc.com/product/pro.php?pro_id=105, (accessed 7/6/16).

[20] Taranis X9D Plus, FrSky Electronic Co., Ltd., http://www.frsky-rc.com/product/pro.php?pro_id=137, (accessed 7/6/16).

[21] One Wire Digital Temperature Sensor - DS18B20, SparkFun Electronics, https://www.sparkfun.com/products/245, (accessed 7/6/16).