Fall 2014

Joint Architecture For Reusable Vehicle-Integrated Software (J.A.R.V.I.S)

Anthony Mark Kane
Purdue University

Graduate School Form 30
(Revised 08/14)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Mark Anthony Kane

Entitled
JOINT ARCHITECTURE FOR REUSABLE VEHICLE-INTEGRATED SOFTWARE (J.A.R.V.I.S)

For the degree of Master of Science in Aeronautics and Astronautics

Is approved by the final examining committee:

Inseok Hwang

Daniel A. DeLaurentis

Dengfeng Sun

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification/Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Inseok Hwang

Approved by Major Professor(s): _____

_____

Approved by: Wayne Chen 11/17/2014

Head of the Department Graduate Program Date

JOINT ARCHITECTURE FOR REUSABLE VEHICLE-INTEGRATED SOFTWARE (J.A.R.V.I.S)

A Thesis

Submitted to the Faculty

of

Purdue University

by

Mark A. Kane

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science in Aeronautics and Astronautics

December 2014

Purdue University

West Lafayette, Indiana

To my wife and children for helping to maintain my sanity.

ACKNOWLEDGEMENTS

I would like to acknowledge Dr. Inseok Hwang for allowing me the opportunity to perform research as a distance student and for the input that he provided along the way. I would like to thank Christopher D'Souza for his patience and time taken to help me through the derivation of the UDU Kalman filter. The task of integrating JARVIS with the ground control station was performed by Brian Killeen, thank you for allowing me to focus my efforts elsewhere.

TABLE OF CONTENTS

Page

Page

Page

LIST OF FIGURES

Figure Page

LIST OF ABBREVIATIONS

CNTRL	Control
DoF	Degrees of Freedom
ECRV	Exponentially Correlated Random Variable
GN&C	Guidance, Navigation, and Control
GUID	Guidance
IDE	Integrated Development Environment
NAV	Navigation

ABSTRACT

Kane, Mark A. M.S.A.A., Purdue University, December 2014. Joint Architecture for Reusable Vehicle-Integrated Software (J.A.R.V.I.S). Major Professor: Inseok Hwang.

An integrated software architecture for development of unmanned research vehicles is developed. It has been created under the premise that all unmanned vehicles require a core set of functionality that is common across platforms and that priority should be to the readability and reusability of the code base. The architecture defines the top-level system interfaces allowing internal algorithms to be manipulated without affecting the rest of the system. A robust aerospace toolbox has been developed that provides a means to rapidly prototype algorithms without the need of recreating commonly used functions or the use of expensive, proprietary software.

CHAPTER 1. INTRODUCTION

1.1 Problem Overview and Motivation

When solving an engineering problem the first step that should be taken is to research how similar problems have been solved before and then to decide how to modify the solution to fit one's needs. Most often the final solution chosen is the one that has the most heritage both because it has worked in the past and it provides a comfort level that can only be gained through experience and use. The unwillingness to take any risk, while respectable, prevents growth. In guidance, navigation and control (GN&C) new techniques go unused for decades due to the long development cycle of vehicles, particularly those used in human transport. The question becomes how to gain heritage with a new algorithm when we are always attempting to minimize risk. Or perhaps more importantly, how does an engineer gain the experience necessary to take an algorithm from paper and implement it for real-time use?

The conventional solution is to expend hours developing system models and then test in a simulated environment. While simulation is invaluable in initial development it should be considered a first-step in the development cycle as two primary failings are inherent to computer modeling.

One is that the engineer does not gain necessary experience required to anticipate problems with the integrated system and often has little knowledge of how their portion integrates into the final vehicle. The second issue is that computer models are simplifications of real-world dynamics and despite great efforts can be incorrect or include simplifications that remove effects that are important to the behavior of the system.

Prior to the large-scale computational power that is now available, full-scale prototypes were employed at a great monetary expense that has reduced their use today. Fortunately, due to the rapid pace of technological development, micro unmanned vehicles, land, air, and sea, offer a low-cost method for rapid development and testing of software and hardware. There are currently a number of commercial auto-pilot systems available that allow for the conversion of R/C platforms into unmanned vehicles. A few examples include the ArduPilot by DIY Drones[1] and the MicroPilot[2]. These platforms offer a path to test new algorithms or sensors without the need of special permissions or infrastructure to operate.

The autopilot solutions available have been developed with the primary goal of making the system as simple as possible for a consumer to plug-and-play and place high priority on the telemetry systems. The accompanying software is most often closed-source and does not allow for manipulation of the code, or code manipulation is limited.

---

[1] Diydrones.com

[2] Micropilot.com

Open source software solutions are tailored to a specific type of vehicle making code modifications laborious. They also do not include a robust library toolset that lends itself to GN&C development.

By learning from the current autonomous solutions, J.A.R.V.I.S provides a simple, adaptable integrated software solution for rapid unmanned vehicle development and research.

1.2 Contributions of This Research

The purpose of this research is not to develop a new algorithm or to design a revolutionary vehicle and is instead intended to facilitate the creation and research of these items. The problem addressed by this research is the lack of a robust integrated architecture for use in developing unmanned vehicles for research purposes. Development often focuses on a particular algorithm, such as a new optimal control technique, and completely overlooks the integrated system or the infrastructure necessary to run the algorithm.

The goal of this research is the development of core-software that can be used to quickly build an unmanned system and also test new GN&C techniques using various types of vehicle and hardware platforms. This goal is achieved through a software architecture that is designed to be interchangeable as well as a library of tools similar to those found in MatLab[3] that are necessary for rapid prototyping without the added cost of proprietary software.

[3] mathworks.com

1.3 Overview

As mentioned previously the intent of this research is to develop a software architecture and library set that will enable rapid algorithm prototyping and unmanned vehicle development. In-depth derivation of equations and library functions is avoided for brevity and can be found in the references provided.

This thesis first examines a popular solution for platform development and outlines the reasoning behind the need for a better set of tools. This is followed by the architecture layout and a description of the tools that are provided by the software developed as part of this research. Chapters 4 and 5 attempt to show the ease in which algorithms can be implemented and vehicles can be developed by utilizing JARVIS. Before continuing to the subject matter it is first necessary to define how the terms guidance, navigation, and control are used in the proceeding sections to avoid confusion.

Navigation refers to the determination of the current vehicle state including attitude, position, velocity, and any number of other variables that may be desired such as air speed. Navigation can be commonly thought of as “Where am I?”

Guidance refers to the determination of the desired path of travel and informs the vehicle how to reach a desired target state. Guidance determines “How do I get there?”

Control refers to the actuation of effectors, such as thrusters or aero surfaces, necessary to track the guidance commands. While the other systems are largely unchanged between vehicle platforms control is specific to the type of actuators available to effect a change in the vehicle state.

Another system that will be discussed is the Pilot system. Piloting can be performed through external commands or autonomously. In both cases the purpose of the pilot is to determine the desired target state; “Where do I want to go?”

CHAPTER 2. ARDUPILOT CASE STUDY

The ArduPilot system is an inexpensive auto-pilot solution that is gaining popularity due to the low cost and because the code is open-source. Additionally, the software is coded in the C++ language allowing for any custom libraries created to be used on any platform capable of compiling and running C++ code.

The ArduPilot has a large open-source community that is actively developing the code and capability of the system that can make it difficult to keep up with the latest revision, for this research the code being inspected is the ArduCopter v2.8.1 retrieved from the online repository that is provided by DIY drones.

The code has been investigated for the purpose of determining the accomplishments and deficiencies inherent to the architecture. This is not meant as a critique of the accomplishment of the DIY Drone team or the open-source community developing the software. It is known that a number of updates have been released after v2.8.1 that address some of the problems that are discussed in the proceeding sections.

The architecture developed for this research has different priorities and aims to build on that which has already been accomplished.

2.1 High Level Overview

The ArduPilot hardware consists of an APM 1.0, APM 2.x, or PX4 system. The hardware is a small form factor and contains a micro-processor, barometer gyroscope, accelerometer, and a number of input and output pins that include those necessary to read and control R/C equipment such as a servo. Sensors can be expanded to include GPS and magnetic compasses.

Software for the ArduPilot is platform dependent and includes the ArduPlane, ArduCopter, ArduRover, and ArduBoat. As each name implies the individual software solutions are intended for different types of vehicles such as conventional aircraft, helicopter and multi-rotors, land-based vehicles, and watercrafts.

For most users the setup of an unmanned vehicle using the ArduPilot system is accomplished through use of the MissionPlanner software that is obtained from the DIY website. Hardware options and configuration variables are set through the GUI and the necessary software is downloaded, compiled, and then uploaded to the vehicle. The user is not required to interact with the base code.

Advanced users are able to modify the code by first downloading the latest software from an online repository. It can then be modified, compiled and uploaded using the Arduino integrated development environment (IDE).

2.2 Architecture

The base software architecture follows typical Arduino formats. The main code that calls the initialization routines as well as performs the operation of the vehicle when powered is contained in a ".pde" file of the same name as the sketch, *ArduCopter.pde* in the case of the code being investigated. Logic is then separated into other .pde file and within the libraries.

2.2.1 ArduCopter Code

There are 35 files that are specific to the ArduCopter code, library files are used in for the other variants. File names as well as a brief description of the logic contained within are given in Table 1.

Table 1: ArduCopter Code

ArduCopter	
File	Purpose
APM_Config.h	Allow the user to set configuration variables Overwrites any previous definition
APM_Config_mavlink_hil.h	Configuration file allowing the telemetry to operate when in "hardware-in-the-loop" mode
ArduCopter.pde	Variable definitions as well as calls all subsystems necessary to operate the vehicle
Attitude.pde	Combination of GN&C routines used in controlling the vehicle attitude
commands.pde	Common function definitions used in commanding the vehicle
commands_logic.pde	Level above commands.pde used in commanding the vehicle

Table 1: Continued

commands_process.pde	Top level commands used in commanding the vehicle
config.h	Default vehicle configuration file
config_channels.h	Configure the vehicle channels
control_modes.pde	Functions used to switch between different control modes
defines.h	Enumeration defines used to assist in making the code readable
events.pde	Functions to handle failsafe or low battery conditions
failsafe.pde	Allows checking of a software lock
flip.pde	Logic to invert the vehicle
GCS.h	Interface definition for ground control protocols
GCS.pde	Interface functions for ground control protocols
GCS_Mavlink.pde	Interface for ground control using mavlink
inertia.pde	Inertial integration of the accelerometer
leds.pde	Functions to change LED behavior
Log.pde	Functions to read/write to memory
Limits.pde	Logic to return-to-home if vehicle goes out-of-bounds
motors.pde	Arms motors Starts the barometer Stores the initial position
navigation.pde	Calculates state errors with respect to desired Filters inertial velocities
Parameters.h	Structure containing parameters used by various subsystems
Parameters.pde	Load and store parameters to memory
planner.pde	Access to the mission planner
radio.pde	Reads the R/C input

Table 1: Continued

sensors.pde	Functions to read the compass, optical-flow, battery, and barometer sensors
setup.pde	Menus for setting up the vehicle via the command line
system.pde	Functions to initialize hardware
test.pde	Functions to test system functionality
UserCode.pde	Empty function allowing for custom user functionality
UserVariables.h	Custom user-defined variables

System variables used throughout the system are defined globally in the ArduCopter.pde file. There is not a consistent naming convention or unit system that is being used; units are metric but may be given in meters, centimeters, degrees times 100 or any number of other combinations. Choice of variable units is most likely a result of electing to perform mathematics using integers. This is done presumably so as to reduce computational time while maintaining accuracy and limiting memory use. In the past compilers for ARM based processers were notorious for producing erroneous results when using floating point variables.

2.2.2 Hardware Operation

The key item to note is that sensor access is asynchronous. Readings are stored in internal sensor variables that are updated when the system detects that the sensor is ready. This interrupts the current process. When a sensor reading is necessary during operations the data is accessed through the stored variable and does not actually call the sensor.

2.2.3 Subsystems

The software being investigated is not grouped into logical GN&C partitions. It also does not appear to follow conventional definitions for GN&C. Throughout the code navigation is most routinely used to reference methods or variables that would typically be considered part of guidance although some routines are used to determine the current state.

2.3 Conclusions

The ArduPilot system has a number of features that are useful in vehicle development as well as a number of architectural choices that make code modification and reading difficult.

Advantages of this system are:

- Inexpensive

The ArduPilot can be purchased for a few hundred dollars. At this price the system is quite affordable to anyone and is less painful to replace in case of a catastrophic failure during vehicle testing.

- Wide range of hardware capability

The hardware that is provided offers an interface to a wide variety of sensors including GPS, accelerometers, gyroscopes, compasses, barometers, and many more. The built in PWM generator is particularly useful since it does not require any additional hardware when controlling R/C servos.

- Small and light-weight

All of the aforementioned capability comes in a small, light-weight, form factor that can be flown on any number of micro aerial vehicles without affecting performance or being too cumbersome to mount.

- Coded in C++

C++ is a cross-platform language. This allows for the development of libraries that can be used not only on Arduino hardware but also on PCs and other devices.

Disadvantages include:

- Over use of global variables

  While global variables provide easy access for subsystems it is the opinion of the author that their use is a dangerous coding practice particularly for autonomous hardware. It is difficult to determine what is using a global variable and were it is being manipulated. Comments within the Arduino code indicate that it is uncertain what some of the variables are being used for.

- Inconsistent Naming Convention and Unit Selection:

  The lack of a consistent naming convention prevents a user from being able to identify the purpose of a variable or what units are being used. The reasoning behind the choice of units in the code commentary is often sparse leaving the user to guess what is being used.

- Logic Grouping

  Functionality is not grouped into logical subsystems and the GN&C definitions do not follow conventional descriptions. Not having a well-defined grouping of logic makes it difficult to determine how and where GN&C is being performed. This makes modification of the code particularly difficult.

- Lack of robust mathematical library

Many GN&C systems, including those that the ArduCopter system is intended, have multiple inputs and outputs (MIMO). These types of systems can be conveniently represented in a state-space, or matrix, format. Some tools have been created that handle 3x3 matrices and 3x1 vectors but are not generic to be expanded to n-size arrays. Linear algebra routines are also missing making implementation of GN&C algorithms time consuming and tedious.

CHAPTER 3. J.A.R.V.I.S

3.1 Architecture

The premise of this architecture is that all autonomous vehicles require basic, common, functionality that is unchanged between vehicle platforms. JARVIS software is designed such that common logic is grouped into subsystems and these subsystems only access each other via input and output busses. Communication between subsystems is shown in Figure 1. Additionally all hardware functionality is abstracted to provide a common interface. The hardware abstraction layer provides access to the sensors and brings external information into the system as well as sending information out of the system.

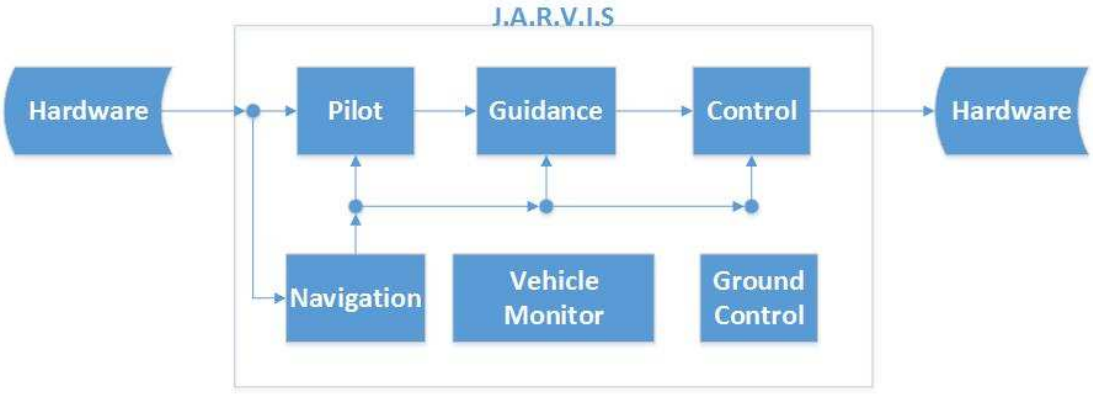

Figure 1: JARVIS Block Diagram

3.1.1 Subsystems

Subsystems contain common logic that is necessary to operate unmanned systems such as guidance, navigation, and control. Additional subsystems included are pilot, vehicle monitor, and ground control. The purpose of each is detailed below. Separation of logic in this way allows for simple replacement of an entire subsystem to occur without affecting the other subsystems.

The systems detailed in the proceeding sections should be thought of as an empty box. The base classes described provide a common interface and allow the user to populate the box to suit their purpose.

3.1.1.1 Pilot

The primary function of the Pilot subsystem is to provide a state command. This command informs the vehicle about where it "wants" to be. Piloting consists of three different modes:

- Manual
  - Reads user input, throttle, rudder, aileron, elevator
  - Control uses the commands directly and maps them to the effectors
- Pilot-Assist
  - Reads user input, throttle, rudder, aileron, elevator
  - User input is modified by Guidance and Control
  - Vehicle is completely stabilized by the autopilot, allows a novice to pilot any vehicle
  - Requires the state estimate from a Navigation subsystem

- Autonomous
    - Independent of user
    - Uses a pre-defined set of waypoints or maneuvers to perform a mission
    - Requires the current state from Navigation

The base functionality and variables required for piloting are included in the pilot class shown in Table 2.

Table 2: Pilot Base Class Description

PILOT Class	
VARIABLES	
Pilot_Mode	Output variable indicating the desired piloting mode
Command	Output vector containing the pilot command
State_Actual	Input vector containing the current state
User_Input	Input vector containing the command from the user, interpreted as a rate command
FUNCTIONS	
Calculate_Command	Generate a state command to be used by Guidance

The base class provides a common interface that allows for custom development for the underlying systems. The pilot base class will be the foundation for future classes that can be created to perform maneuvers such as circling, inverting, and waypoint tracking.

3.1.1.2 Guidance

Guidance is an often overlooked subsystem as, depending on the design, the functionality can be included in control. The approach taken in the JARVIS architecture is to have the pilot system provide a state command to guidance. Guidance is then used to introduce additional information to the command and provide an output that is used by control. For example the pilot command could be interpreted as a desired rate. Guidance can then modify the rate with attitude information to provide attitude control.

By using this separation states can be introduced at different levels and allows the system to be tuned incrementally and assists in removing coupling problems that could otherwise make control tuning comparably more difficult.

The guidance base class is shown in Table 3.

Table 3: Guidance Base Class Description

Guidance Class	
VARIABLES	
Command	Output vector containing the commanded vehicle state
State_Actual	Input vector containing the current state
State_Desired	Input vector containing the desired state from Pilot
Command	Output vector containing the commanded vehicle state
FUNCTIONS	
Calculate_Command	Generate a state command to be used by Control

3.1.1.3 Navigation

Navigation is one of the most notorious, and possibly most important, subsystems for autonomous operations simply due to the fact that without accurate NAV information it is impossible to operate a vehicle regardless of how advanced the supporting infrastructure is.

To assist in understanding what a NAV output is providing the variable should be descriptive and conform to a naming convention. The variable naming convention utilized in JARVIS is outlined in Table 4.

Table 4: NAV Variable Naming Conventions

_B	Indicates that the variable is in the body frame
_NED	Indicates that the variable is in the North-East-Down (NED) frame
_YPR	Rotation sequence is Yaw-Pitch-Roll (3-2-1)
P	Position, [X,Y,Z]’
w	Angular rate
E	Euler angle
Quat	Attitude quaternion
T	Transformation matrix Subscript is read as TO axis system FROM axis system T_BNED is the transformation to the body frame from the NED frame

In general navigation algorithms consist of predicting the state, most often done by propagating dynamic equations using the sensor readings, and correcting the state with an external observation.

For example, in the ArduCopter software the direction cosine matrix (DCM) is propagated using the rate gyro. The gravity vector is then used to estimate the error in Roll and Pitch. A PD controller then o augments the gyro rates in subsequent propagation states to correct the DCM.

The class given in Table 5 will be utilized later in this thesis when implementing an attitude estimation filter.

Table 5: Navigation Base Class

Navigation Class	
VARIABLES	
User Defined	Variables are specific to the NAV implementation and depend on user needs
FUNCTIONS	
Predict_State	Predict what the state is, typically done by propagating dynamic equations
Correct_State	Correct the state estimate using external observations

3.1.1.4 Control

The purpose of the control system is to generate effector commands that will realize the desired state. There are a wide variety of control algorithms that can be used, some of which will be discussed later. The base functionality for control is to take in a desired state and generate an effector command. The base class is shown in Table 6.

Table 6: Control Base Class

Control Class	
VARIABLES	
Command	Commands sent to the effectors
State_Desired	Desired state that control is attempting to achieve
State_Actual	Current state of the vehicle
State_Error	Difference between the desired and actual state
Effector_Map	Matrix that maps control commands to the effectors
FUNCTIONS	
Calculate_Command	Function to generate an effector command

3.2 Libraries

To facilitate the implementation of various GN&C algorithms it is necessary to have a robust library as a base to build from. There is a notable lack of tools available for the open-source autopilot systems and while there are a wide variety of commercial applications that provide these tools they are often cost prohibitive.

MatLab is one example of software that is popular in the field of engineering as it provides a large suite of libraries that provide means to convert units and coordinate systems, solve linear systems, and quickly plot results. It also includes tools for converting models into C++ code that can be run on hardware.

MatLab however does come at a considerable monetary investment and being proprietary does not often allow one to see the underlying algorithms. Auto-coding software adds an additional layer that can contain mistakes or operate in unintended ways. A simple example is if the *sign* returns zero or positive one when the input has a value of zero. Code generated though an auto-coding process can also be difficult to read and debug if necessary.

To avoid the cost overhead, a library has been developed in C++. MatLab has been used to independently verify functionality. To facilitate conversion from one environment to another function names and call list are as similar to the MatLab environment as possible. For those familiar with MatLab this provides a familiar feel when developing for JARVIS.

The following sections give a brief overview of the current capability of the library and as with any tool it is in continuous development. A full list of available functions is provided in Appendix A.

3.2.1 Use of Classes

To facilitate the reusability and readability of the code the C++ classes will be used heavily. C++ classes contain all the variables and related functions desired for a particular subsystem and can be used in higher-level code due to the feature of inheritance, illustrated in Figure 2. Classes also allow for variables to be protected by making them private ensuring that only a subset of data is visible to external functions so that it is not changed or overwritten accidentally. The Inheritance feature becomes very useful when creating different GN&C algorithms.

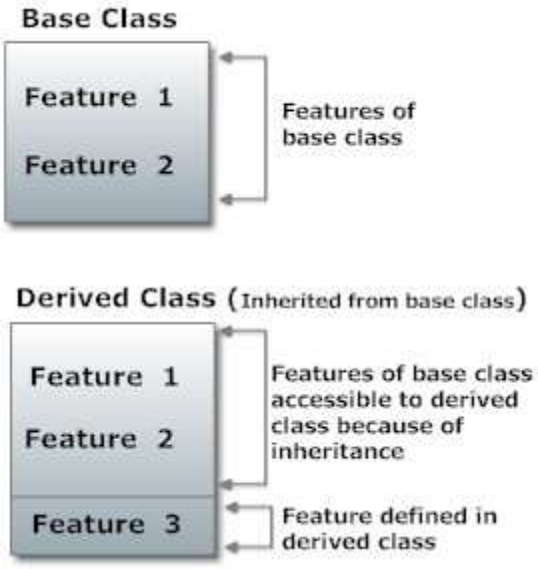

Figure 2: Class Inheritance (www.programiz.com, 2014)

Using templates allows a class to be used for different types of variables and eliminates the need to recreate identical code so that it can work with both *double* and *floating* point values for example.

3.2.2 Math Enhanced

This library has been created to provide resizable arrays as well as linear algebra tools. It contains an array class, matrix class, linear algebra functions and some simple signal handling.

3.2.2.1 Arrays

It is desirable to create an array of objects that can be accessed by index and that if necessary be dynamically resized thus allowing one to loop through data in a parameterized fashion and maintain code cleanliness. For example, if there are four servos on the vehicle and there is a “Servo” class that provides access to basic servo functionality and commanding, the ArduCopter code would need four “Servo” variables. The array class declaration for the same situation is:

Array<Servo> Servos(4, 1)

This allows for each servo to be accessed from the “Servos” variable as would be possible in MatLab with the exception that indices are base-zero to stay consistent with the C++ language were as MatLab is base-one .

The array class also contains the information on its size eliminating the need for additional checks when performing a loop:

```
for(int i; i < Servos.nrows(); i++)
Servos(i).SendCommand;
```

To further enhance the capability the number of servos can be parameterized as “nServos” that can be defined in the configuration file:

```
Array<Servo> Servos(nServos,1);
```

The array class operates in two modes, the default is static sizing and the secondary is dynamic sizing. The static size assumes that the array is a fixed size and although it is possible to resize the variable manually, the data will not be maintained when doing so. The dynamic sizing option allows the array to be resized automatically, in that if one were to attempt to set an element that was outside of the current array size, the array size would be increased while maintaining all previous data. The dynamic size method is useful for off-line loading of data and was implemented for use in the time-series class that is described in section 3.2.2.3.

3.2.2.2 Matrices

The matrix class is intended to be used for numerical computations and inherits all of the capabilities from the array class while providing additional capabilities, such as internal functions that zero the elements or populate the elements so that the variable is an identity matrix. Variable declaration is done in the same manner as the array class:

Matrix<float> A(3,3)

3.2.2.3 Time-Series Class

The time series class is intended for storing time-sequential data primarily for use in post-processing data. This class contains two internal arrays, one that stores the time vector, and one that stores the data. Functions are included within the class that allow for data to be accessed by time or index.

3.2.2.4 Linear Algebra

A number of functions have been written to provide basic routines necessary when constructing GN&C algorithms. These include matrix multiplication, addition and scaling, QR and UD matrix decomposition. Also included is the capability to solve linear systems and invert or orthonormalize a matrix. These functions rely on the matrix class and have been compared against results produced by MatLab.

3.2.3 Aerospace Toolbox

The Aerospace toolbox has been created to provide a number of tools necessary in unmanned vehicle development. These include routines for transforming coordinate systems, converting units, and filtering signals. This library also contains a quaternion class that is expected to be used for variables that represent attitude using a quaternion. Control classes have been implemented and can be used within the control subsystem when generating effector commands.

3.2.3.1 Controllers

The control toolbox is intended to contain a number of control classes that can be used to generate control commands. The classes contain the variables and functions necessary for any implementation of that particular algorithm. Examples that are commonly seen in textbooks are state-space, positional-integral-derivative (PID), linear quadratic regulator (LQR), and model predictive control (MPC).

The only control classes that are currently available are for PID and state space control. Future improvements will be to add more control algorithms.

3.2.3.2 Conversions

One of the most common tools necessary for unmanned development is the ability to transform coordinate systems between each other as well as convert units. Routines have been implemented that provide basic unit conversions as well as common transformations such as converting between the quaternion attitude representation and the Euler-angle representation. A full list is given in Appendix A.

3.2.3.3 Filters

A number of filters have been constructed that allow for input signal filtering as well as estimation.

3.2.3.3.1 Average

The average filter is used to average input values over a moving window. The user defines the number of values (n) in the averaging window and updates the filter by passing in values from the signal to be filtered (x); y is the filtered output.

$$y = \frac{\sum_{1}^{n} x_n}{n} \tag{3.1}$$

3.2.3.3.2 High-Pass

High-pass filtering allows for high frequency content to pass through the system while filtering out low frequencies. At initialization the user specifies the smoothing factor (α) with a value ranging from 0 to 1. The filter is updated by passing in the unfiltered value (x_k) and outputs the filtered value (y_k). The previous input and outputs are stored internally as x_{k-1} and y_{k-1}.

$$y_k = \alpha(y_{k-1} + x_k - x_{k-1}) \tag{3.2}$$

3.2.3.3.3 Low-Pass

Low-pass filtering allows for low frequency content to pass through and removes high frequency content. As with the high pass filter the user specifies a smoothing factor (α) at initialization. The filter is updated by passing in the unfiltered value (x_k) and outputs the filtered value (y_k). The previous input and outputs are stored internally as x_{k-1} and y_{k-1}.

$$y_k = y_{k-1} + \alpha(x_k - y_{k-1}) \tag{3.3}$$

3.2.3.3.4 Kalman Filter

The Kalman filter is at the heart of most modern aerospace navigation systems. Each implementation of a Kalman filter is unique due to the states that are being estimated as well as the sensors available for measurements. Fortunately large portions of the algorithm are reusable allowing for the development of a library class that reduces the time and effort necessary to develop the state filter.

The Kalman filter can be computationally burdensome and, depending on the method chosen, numerically unstable. Both qualities are undesirable and make implementation of Kalman filters challenging. The approach chosen to address this problem has been taken from (Holt, Greg N.; D'Souza, Christopher) and is unique in that it is numerically stable and computationally efficient. What is given below is a brief synopsis of how the filter is formed, for a more thorough understanding on the derivation of the equations please refer to the references provided.

3.2.3.3.4.1 Dynamic System Model

The system is modeled as linear with white Gaussian noise. The discrete system model is given in equation(3.4). The state vector is defined as X , F is the state transition matrix and is calculated at each time step. G is the noise mapping matrix, and w is the white Gaussian noise. In this section the subscript k is used to indicate the current value and k-1 is the previous value.

$$\mathrm{X}_k = F_k \mathrm{X}_{k-1} + G w_k \tag{3.4}$$

3.2.3.3.4.2 Measurement Model

The measurement model is also assumed to be linear with Gaussian noise. Given a state vector that is corrupted by zero mean Gaussian noise (v) and the sensitivity matrix (H) the measurements (y) are modeled as:

$$y_k = H_k X_k + v_k \tag{3.5}$$

3.2.3.3.4.3 Updating Covariance

In Kalman filtering there are two primary steps, one is to perform a time update and is also known as prediction. The second is to correct the state prediction using measurements from external sensors. The following details the formulation used for the covariance matrix and the base equations necessary to perform these steps.

3.2.3.3.4.3.1 UDU Formulation of the Covariance Matrix

In this approach to improve computational stability the covariance matrix (P) is factorized into an upper-triangular matrix (U), that is also orthogonal, and a diagonal matrix (D) that contains the singular values:

$$P = UDU^T \tag{3.6}$$

A singular value decomposition (SVD) function is included in the library so as to allow initialization of the filter using the standard covariance matrix.

3.2.3.3.4.3.2 Parameterization

Computational efficiency is improved by parameterizing the states. This benefit is realized in filters with a large number of ECRVs, such as a sensor bias or mounting error, by taking advantage of the sparseness of the corresponding matrices.

The state vector is partitioned into "real" states (χ) and ECRVs (p) in the following manner:

$$X = \begin{bmatrix} \chi \\ p \end{bmatrix} \tag{3.7}$$

Likewise the U and D matrices are partitioned as shown in equations (3.8) and (3.9). The subscript $(\chi\chi)$ indicates the portion of the matrix containing the first order partial derivatives of the real states with respect to the real states. The subscript (χp) indicates the first order partial derivatives of the real states with respect to the ECRV states. The subscript (pp) indicates the first order partial derivatives of the ECRV states with respect to the ECRV states.

$$U = \begin{bmatrix} U_{xx} & U_{xp} \\ 0 & U_{pp} \end{bmatrix} \tag{3.8}$$

$$D = \begin{bmatrix} D_{xx} & 0 \\ 0 & D_{pp} \end{bmatrix} \tag{3.9}$$

The state transition matrix partitioning is given in equation (3.10). Here M is a diagonal matrix with each non-zero element being the inverse of the ECRV time constant.

$$F = \begin{bmatrix} F_{xx} & F_{xp} \\ 0 & M \end{bmatrix} = \begin{bmatrix} I & 0 \\ 0 & M \end{bmatrix} \begin{bmatrix} F_{xx} & F_{xp} \\ 0 & I \end{bmatrix} = F_1 F_2 \tag{3.10}$$

The weighting matrix is partitioned as shown in equation(3.11).

$$Q = \begin{bmatrix} Q_{xx} & 0 \\ 0 & Q_{pp} \end{bmatrix} = \begin{bmatrix} Q_{xx} & 0 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & Q_{pp} \end{bmatrix} = Q_1 + Q_2 \tag{3.11}$$

3.2.3.3.4.3.3 Performing the Time Update

Using the modifications described in the previous sections, the time update for the covariance matrix takes the form:

$$P_k = U_k D_k U_k^T = F_{2_k} \left[F_{1_k} U_{k-1} D_{k-1} U_{k-1}^T F_{1_k}^T + Q_{1_k} \right] F_{2_k}^T + Q_{2_k} \tag{3.12}$$

From equation (3.12) it is observed that the time-update can be performed using two-steps. Here the k and k-1 subscripts have been dropped for brevity.

1. Compute the inner product. This can be done using a modified Gram-Schmidt process (Thornton and Bierman):

$$UDU^T = F_1 UDU^T F_1^T + Q_1 \tag{3.13}$$

2. Compute the outer product. This can be accomplished with the Agee-Turner rank-one update (Agee):

$$UDU^T = F_2 UDUF_2^T + Q_2 \tag{3.14}$$

3.2.3.3.4.3.4 Measurement Update of Covariance

Measurements are processed one at a time the *a posteriori* covariance $\left(P^+\right)$ given is a function of the Kalman gain (K), the measurement partial (H), and the *a priori* covariance $\left(P^-\right)$. When discussing measurement updates the superscript (+) indicates the *a posteriori value* and (-) indicates the *a priori value.*

$$P^+ = P^- - KHP^- \tag{3.15}$$

The optimal Kalman gain is given in equation(3.16) with R being the measurement co-variance.

$$K = P^- H^T \left(HP^- H^T + R\right)^{-1} \tag{3.16}$$

Using the modifications described in the preceding sections the covariance measurement update then takes the form:

$$P^{+} = U^{+} D^{+} U^{+T} = U^{-} \left[D^{-} - \frac{1}{\alpha} v v^{T} \right] U^{-T} \tag{3.17}$$

With the following definitions:

$$v = D^{-} U^{-T} H^{T} \tag{3.18}$$

$$\alpha = H U^{-} D^{-} U^{-T} H^{T} + R \tag{3.19}$$

$$K = U^{-} \left[\frac{1}{\alpha} v v^{T} \right] \tag{3.20}$$

The Carlson rank-one update is used to solve for U^{+} and D^{+} using the definitions given in equations (3.21) through (3.23).

$$\bar{U} \bar{D} \bar{U}^{T} = D^{-} \frac{1}{\alpha} v v^{T} \tag{3.21}$$

$$U^{+} = U^{-} \bar{U} \tag{3.22}$$

$$D^{+} = \bar{D} \tag{3.23}$$

Because measurements are processed individually H is a row vector that has a size of 1 by $\left(n_x + n_p\right)$ where n_x is the number of real states and n_p is the number of ECRV states. This results in a scalar value for α and eliminates the need for matrix inversions that are computationally burdensome and reduce numerical stability.

3.2.3.3.4.4 Updating the State Estimate

As with the covariance matrix the state update consists of prediction and correction.

3.2.3.3.4.4.1 State Time Update

The time update is often the propagation of dynamic equations, or alternatively the integration of the inertial sensors at each time step. The time update is specific to the filter that is being designed and does not lend itself to being part of a generic Kalman filter class. Instead it is expected that a higher-level class will be created that contains the logic necessary to propagate the state as well as populate the state transition matrices necessary for performing the covariance updates.

3.2.3.3.4.4.2 State Measurement Update

The state measurement is performed with the covariance update. The Kalman filter is used to estimate the error in the state prediction (Δx) that is then used to correct the state estimate.

Measurements are again performed one at a time with y being the measurement and h the expected, or estimated, value.

$$\Delta x^{+} = \Delta x^{-} + K\left[y - h - H\Delta x^{-} \right] \tag{3.24}$$

Substituting equation (3.20) for K:

$$\Delta x^{+} = \Delta x^{-} + U^{-}\left[\frac{1}{\alpha} v v^{T} \right]\left[y - h - H\Delta x^{-} \right] \tag{3.25}$$

For an Extended Kalman Filter the state correction is linear:

$$X^{+} = X^{-} + \Delta x \tag{3.26}$$

Δx must be re-zeroed each time the state is corrected. Using the methodology described any number of measurements can occur before the state vector is corrected.

3.2.3.3.4.4.3 Measurement Rejection

One of the goals of this research is to improve the robustness of navigation by providing a method to reject erroneous sensor measurements. Measurement rejection is accomplished by comparing the residual $(y - h - H\Delta x)$ to a user-defined threshold that is specified at initialization.

$$\begin{cases} \left\| [y - h - H\Delta x]\sqrt{\frac{1}{\alpha}} \right\| > Threshold, & Discard \\ \left\| [y - h - H\Delta x]\sqrt{\frac{1}{\alpha}} \right\| <= Threshold, & Keep \end{cases} \tag{3.27}$$

3.2.3.3.5 Library Implementation

The logic described in the previous sections is implemented in the UDU_EKF class. The class includes the necessary variables for running an EKF filter as well as all the low-level logic to perform the heavy lifting for the covariance time update as well as the state and covariance measurement updates. This class is generic to the filter design and expects that the user will include it as part of an estimation algorithm.

The amount of system memory used and the number of computations required is reduced by taking advantage of the sparseness of the parameterized matrices and storing the diagonal matrices as column vectors.

Table 7: UDU EKF Class Description

UDU_EKF	
VARIABLES	
U	Post-update ($n_x + n_p$)-by-($n_x + n_p$) unit upper triangular matrix
D	Post-update ($n_x + n_p$)-by-1 vector of the diagonal elements of the matrix D
PHI_x	(n_x)-by-($n_x + n_p$) state transition matrix associated with the non-ECRV states
PHI_p	(n_p)-by-1 vector of the diagonal elements of the ECRV state-transition matrix
G_x	(n_x)-by-(n_x) process noise mapping matrix corresponding to non-ECRV states (matrix is unit upper triangular)
Q_x	(n_x)-by-1 vector containing the diagonal terms of the process noise covariance matrix Q_X corresponding to non-ECRV states
Q_p	(n_p)-by-1 vector containing the diagonal terms of the process noise covariance matrix Q_P corresponding to ECRV states
H	1-by-($n_x + n_p$) vector containing the partial derivative of the measurement with respect to the state using the current estimate
Delta_X	($n_x + n_p$)-by-1 vector containing the current estimate of the state error

Table 7: Continued

FUNCTIONS	
Initialize(n_x, n_p)	Initialize the filter with the number of non-ECRV and ECRV states; Expects the user to initialize U and D afterwards.
set_Rejection_Threshold(Threshold)	Set the multiple of the measurement uncertainty used in rejecting bad measurements
set_Underweighting(Threshold, Coefficient)	Set the underweighting threshold and coefficient
Covariance_Time_Update()	Perform the time update of the covariance matrix; Expects that the user has set the state transition variables (PHI_x, PHI_p, G_x, Q_x, Q_p) prior to calling this function
Measurement_Update(Y, h, R)	Perform the measurement update of the covariance and state error Expects that the user has set the sensitivity vector (H) prior to calling this function

3.2.3.4 Estimators

Estimation classes have been developed to provide capability to produce attitude and position estimates using the aforementioned UDU_EKF class. These classes are intended to be used in a navigation system but are otherwise independent of the navigation architecture.

3.2.3.4.1 Multiplicative Extended Kalman Filter

A multiplicative extended Kalman filter (MEKF) has been implemented for estimating attitude, body rates, as well as gyro bias. By using the UDU algorithm the computational burden as well as the memory footprint has been reduced significantly allowing the filter to be used real-time despite using a low-power system.

3.2.3.4.1.1 Attitude Representation

Several different methods can be used to describe the attitude of a vehicle including Euler angles, quaternions, and Rodrigues parameters. For a body moving in three-dimensional space the quaternion formulation is appealing since no singularities are present[4] and the state error can be estimated by only three variables. The quaternion representation that will be used in this thesis will be the scalar (q_0) followed by the vector (q_v) and is how the quaternion class has been implemented in the *Aero-Toolbox* library.

$$q = \begin{bmatrix} q_0 & q_v \end{bmatrix}^T \tag{3.28}$$

$$q_v = \begin{bmatrix} q_1, q_2, q_3 \end{bmatrix}^T \tag{3.29}$$

[4] A singularity occurs when the error is exactly 360 degrees. Errors this large cause additional problems as the derivation assumes a small error and in any situation is very unlikely to occur considering that the filter is being performed at a high rate (>30Hz).

Additionally quaternion must obey the following constraint:

$$q^T q = 1 \tag{3.30}$$

3.2.3.4.1.2 Attitude Kinematics

The quaternion kinematics (q) are a function of the body angular rates (ω). The formulation as given in (Dreier) is given in equation (3.31). The variables p , q , and r are used to represent the angular rate about the body X, Y, and Z axis respectively. $\otimes$ is used to represent quaternion multiplication.

$$\dot{q} = \frac{1}{2}\Omega q = \frac{1}{2}\Xi_q \omega = \frac{1}{2}\begin{bmatrix} 0 \\ \omega \end{bmatrix} \otimes q \tag{3.31}$$

$$\Omega = \begin{bmatrix} 0 & -p & -q & -r \\ p & 0 & r & -q \\ q & -r & 0 & p \\ r & q & -p & 0 \end{bmatrix}, \Xi_q = \begin{bmatrix} -q_1 & -q_2 & -q_3 \\ q_0 & -q_3 & q_2 \\ q_3 & q_0 & -q_1 \\ -q_2 & -q_1 & -q_0 \end{bmatrix} \tag{3.32}$$

3.2.3.4.1.3 Sensor Modeling

Observations are used to correct the state estimate and are collected from a variety of sensors including rate gyroscopes, accelerometers, magnetometers and GPS.

The gyroscope measures body angular rates (ω) and are modeled as a function of the true angular rates, the gyro bias (β_g), and zero-mean Gaussian noise (n_g).

$$\omega = \omega + \beta_g + n_g \tag{3.33}$$

The accelerometer on each axis measures translational acceleration (a) in the body reference frame and is modeled as a function of true acceleration (a) and zero-mean Gaussian noise (a_g).

$$a = a + n_a \tag{3.34}$$

3.2.3.4.1.4 Derivation of the MEKF Error Model

In an extended Kalman filter (EKF) the attitude quaternion constraint given in equation (3.30) can be violated by the linear measurement update. This obstacle is overcome by using the multiplicative error quaternion. This method represents the attitude quaternion as the product of the estimated attitude $(\hat{q})$ and the deviation from the estimate (δq).

$$q = \delta q \otimes \hat{q} \tag{3.35}$$

Following the procedure given in (Crassidis and Junkins) and by using the quaternion representation given in(3.28), the error between the true and estimated attitude quaternion is given as:

$$\delta q = q \otimes \hat{q}^{-1} \tag{3.36}$$

Using the chain rule, the time derivative of the error quaternion (δq) is given in equation (3.37).

$$\delta q = q \otimes \hat{q}^{-1} + q \otimes \hat{q}^{-1} \tag{3.37}$$

With the definitions given in equations (3.38) and (3.39). Here δq_0 represents the error in the scalar portion of the attitude quaternion and δq_v is the error in the vector portion of the attitude quaternion.

$$\delta q = [\delta q_0, \delta q_v]^T \tag{3.38}$$

$$q^{-1} = [q_0, -q_v] \tag{3.39}$$

The expressions for q, q and $\hat{q}^{-1}$ are available leaving only an expression for $\hat{q}^{-1}$ to be determined. The estimated quaternion kinematics model is:

$$\dot{\hat{q}} = \frac{1}{2}\hat{\Omega}\hat{q} = \frac{1}{2}\Xi_q\hat{\omega} = \frac{1}{2}\begin{bmatrix}0\\ \hat{\omega}\end{bmatrix}\otimes\hat{q} \tag{3.40}$$

The error between the quaternion estimate and itself is the zero-quaternion:

$$\hat{q}\otimes\hat{q}^{-1} = \begin{bmatrix}1 & 0 & 0 & 0\end{bmatrix}^T \tag{3.41}$$

Equation (3.41) is constant so the time derivative is equal to zero:

$$\frac{\partial}{\partial t}\hat{q}\otimes\hat{q}^{-1} = \dot{\hat{q}}\otimes\hat{q}^{-1} + \hat{q}\otimes\dot{\hat{q}}^{-1} = 0 \tag{3.42}$$

Substituting (3.40) into(3.42) gives the following:

$$\frac{1}{2}\hat{\Omega}\hat{q}\otimes\hat{q}^{-1} + \hat{q}\otimes\dot{\hat{q}}^{-1} = 0 \tag{3.43}$$

$\dot{\hat{q}}^{-1}$ is can then be found by substituting equations (3.41) and (3.32) into(3.43):

$$\dot{\hat{q}}^{-1} = -\frac{1}{2}\hat{q}^{-1}\otimes\begin{bmatrix}\hat{\omega}\\ 0\end{bmatrix} \tag{3.44}$$

Substituting $\delta\omega \equiv \omega - \hat{\omega}$ along with equation(3.44) and equation (3.37) results in:

$$\delta q = \frac{1}{2}\left\{\begin{bmatrix} 0 \\ \hat{\omega} \end{bmatrix} \otimes \delta q - \delta q \otimes \begin{bmatrix} 0 \\ \hat{\omega} \end{bmatrix}\right\} + \frac{1}{2}\begin{bmatrix} 0 \\ \delta\hat{\omega} \end{bmatrix} \otimes \delta q \tag{3.45}$$

That can be equivalently written as:

$$\delta q = \frac{1}{2}\left\{\Omega\delta q - \Gamma\begin{bmatrix} 0 \\ \hat{\omega} \end{bmatrix}\right\} + \frac{1}{2}\begin{bmatrix} 0 \\ \delta\hat{\omega} \end{bmatrix} \otimes \delta q \tag{3.46}$$

With:

$$\Gamma = \begin{bmatrix} 0 & -p & -q & -r \\ p & 0 & -r & q \\ q & r & 0 & -p \\ r & -q & p & 0 \end{bmatrix} \tag{3.47}$$

Using equations (3.32) and (3.47) leads to:

$$\delta q = -\begin{bmatrix} 0 \\ \hat{\omega}_x \delta q_v \end{bmatrix} + \frac{1}{2}\begin{bmatrix} 0 \\ \delta\hat{\omega} \end{bmatrix} \otimes \delta q \tag{3.48}$$

The skew-symmetric cross-product matrix is defined as:

$$\omega_x = \begin{bmatrix} 0 & -r & q \\ r & 0 & -p \\ -q & p & 0 \end{bmatrix} \tag{3.49}$$

For the next portion of the derivation the assumption is made that the error is small; the true quaternion is close to the estimated quaternion. This allows the scalar q_0 to be approximated as a constant that is equal to one and removed from the system leaving only δq_v to be estimated. Using a first-order approximation, the linearized model is:

$$\delta q_v = -\hat{\omega}_x \delta q_v + \frac{1}{2}\delta\omega \tag{3.50}$$

$$\delta q_0 = 0 \tag{3.51}$$

The estimated angular velocity (ω) is given as a function of the measured angular velocity and the estimated gyro bias $\left(\hat{\beta}_g\right)$:

$$\hat{\omega} = \omega - \hat{\beta}_g \tag{3.52}$$

Using equations (3.33) and(3.52), and treating the gyro bias as a constant, equation (3.50) takes the form given in equation (3.53) with $\Delta\beta_g$ representing the error in the gyro bias.

$$\delta q_v = -\hat{\omega}_x \delta q_v + \frac{1}{2}(\ \beta_g + n_g) \tag{3.53}$$

Then assuming a small angle $\delta q_v \approx \frac{\delta\alpha}{2}$:

$$\dot{\delta\alpha} = -\omega_x \delta\alpha - (\ \beta + n_g) \tag{3.54}$$

$$\delta\alpha = \begin{bmatrix} \delta\phi & \delta\theta & \delta\varphi \end{bmatrix}^T \tag{3.55}$$

3.2.3.4.1.4.1 EKF Error Model

The state estimated by the MEKF is $\delta\alpha$ and the error in the gyro bias $\left(\delta\beta_g\right)$.

$$\Delta\hat{x}(t) = \begin{bmatrix} \delta\alpha^T(t) & \delta\beta_g{}^T(t) \end{bmatrix}^T \tag{3.56}$$

The continuous-time derivative of the state error $\left(\Delta\hat{x}\right)$ is modeled as a function of the state-transition matrix $\left(F\right)$ computed at time (t), the noise mapping matrix $\left(G\right)$ and Gaussian noise $\left(w\right)$.

$$\Delta\dot{\hat{x}}(t) = F(\hat{x}(t), t)\Delta\hat{x} + G(t)w(t) \tag{3.57}$$

The Gaussian noise comes from noise in the gyroscope measurements $\left(n_g\right)$ and the accelerometer measurements $\left(n_a\right)$.

$$w(t) = \begin{bmatrix} n_g^T & n_a^T \end{bmatrix}^T \tag{3.58}$$

The state transition matrix is a function of the skew-symmetric cross product matrix generated using the estimated body rates $(\hat{\omega}_x(t))$ and the gyro time constant (τ_{gyro}).

$$F(\hat{x}(t),t) = \frac{\partial x}{\partial x} = \begin{bmatrix} -\hat{\omega}_x(t) & -I_{3x3} \\ 0_{3x3} & I_{3x3}\dfrac{1}{\tau_{gyro}} \end{bmatrix} \tag{3.59}$$

The noise mapping matrix and weight matrix (Q(t)) are assumed to be constant.

$$G(t) = \begin{bmatrix} -I_{3x3} & 0_{3x3} \\ 0_{3x3} & I_{3x3} \end{bmatrix}, Q(t) = \begin{bmatrix} \sigma_g^2 I_{3x3} & 0_{3x3} \\ 0_{3x3} & \sigma_a^2 I_{3x3} \end{bmatrix} \tag{3.60}$$

3.2.3.4.1.5 Modification for Use with the UDU EKF Algorithm

Equations(3.58), (3.59), and (3.60) are not parameterized and are in the continuous time form. To use the UDU algorithm constructed for the library, the system needs to be converted into a discrete time form. Assuming a small time step (dt) allows for the following approximation to be made:

$$\Delta\hat{x} = \Delta\hat{x} + \Delta\dot{\hat{x}}dt = \Delta\hat{x} + \left[F(\hat{x},t)\Delta\hat{x} + Gw(t)\right]dt \tag{3.61}$$

$$\Delta\hat{x} = \left[\begin{bmatrix} I & 0 \\ 0 & I \end{bmatrix} + \begin{bmatrix} F_{\chi\chi} & F_{\chi p} \\ 0 & F_{pp} \end{bmatrix} dt \right] \Delta\hat{x} + Gw(t)dt \tag{3.62}$$

Using the nomenclature from the UD_EKF class:

$$PHI_{\chi} = \begin{bmatrix} F_{\chi\chi}dt + I & F_{\chi p}dt \end{bmatrix} \tag{3.63}$$

PHI_p, is diagonal and can be discretized directly by taking the exponential of the diagonal terms multiplied by the time delta.

$$PHI_p = e^{F_p} = I_{3x3}e^{\frac{dt}{\tau_{gyro}}} \tag{3.64}$$

Lastly:

$$G_x = I_{3x3}dt \tag{3.65}$$

3.2.3.4.1.6 Attitude Observations

Although the accelerometers measure the translational state, they can be used to observe pitch and roll through the gravity vector. The gravity vector as measured by the accelerometers is represented by a rotation of the NED gravity vector to the body frame using the NED-to-body transformation matrix $\left(T_{NED}^{\mathcal{B}}\right)$.

$$y = T_{NED}^{B}\begin{bmatrix}0 & 0 & 1\end{bmatrix}^{T} + n_a \tag{3.66}$$

$\hat{a}_b$ is the expected acceleration in the body frame and is found using the current estimate of the attitude $\left(\hat{q}_k^-\right)$.

$$\hat{a}_b(\hat{q}_k^{\,-}) = T_{NED}^{B}(\hat{q}_k^{\,-})\begin{bmatrix}0 & 0 & 1\end{bmatrix}^{T} \tag{3.67}$$

The sensitivity matrix for the acceleration measurements is then:

$$H_k(\hat{x}_k^{\,-}) = \frac{\partial y}{\partial \Delta x} = \begin{bmatrix} 0 & -\hat{a}_{b3} & \hat{a}_{b2} & 0 & 0 & 0 \\ \hat{a}_{b3} & 0 & -\hat{a}_{b1} & 0 & 0 & 0 \\ -\hat{a}_{b2} & \hat{a}_{b1} & 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.68}$$

The accelerometer is unable to observe the yaw angle and instead needs to be corrected by use of a magnetometer, or GPS that provides a ground heading. In these cases the sensors measure the yaw angle directly and can be used to calculate North in the body frame ($\hat{N}_b$) to be used with equation(3.68) in the update:

$$\hat{N}_b = \begin{bmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix}(q)\begin{bmatrix}1 & 0 & 0\end{bmatrix}^{T} + n_{sensor} \tag{3.69}$$

3.2.3.4.1.7 Updating the State Estimate

The state estimate is updated using the estimate of the state error obtained through the measurement updates.

The gyro bias update is linear:

$$\hat{\beta}_k^+ = \hat{\beta}_k^- + \hat{\beta}_k^+ \tag{3.70}$$

The quaternion update is multiplicative:

$$\hat{q}_k^+ = \begin{bmatrix} 1 \\ \frac{1}{2}\delta\hat{\alpha}_k^+ \end{bmatrix} \otimes \hat{q}_k^- \tag{3.71}$$

3.2.3.4.1.8 Implementation

Utilizing the UDU_EKF library class and the MEKF derivation given in the previous section an attitude and heading reference system (AHRS) has been constructed:

Table 8: AHRS_MEKF Class

AHRS_MEKF	
VARIABLES	
Gyro_Variance	Variance of the gyro measurements
Gyro_Bias_Variance	Variance of the gyro bias
Gyro_Time_Constant	Time constant of the gyro
Filter	UDU_EKF declaration used for Kalman filtering
FUNCTIONS	
Initialize(Gyro_Variance, Gyro_Bias_Variance, Gyro_Time_Constant)	Initialize the estimator Inputs are the gyro sensor parameters
aClear()	Clear all allocated memory Filter is no longer valid
Clear_Workspace()	Clear workspace memory to save memory Maintains filter states
Initialize_Workspace()	Initialize workspace variables
Propogate (Quat_BI, w_B, dt)	Propagate the attitude quaternion and covariance Inputs are the attitude quaternion, the estimate of body rates, and the time-step
calcLinSys(w_B, dt)	Calculate the discrete-time system matrices used by the UDU filter (PHI_x, PHI_p, Q_x, Q_p, G_x)
Measurement_Update(Measured, Expected, Variance)	Update the state error estimate using a measurement
Update_Estimate(Quat_BI, Gyro_Bias)	Update the estimate of the attitude quaternion and the gyro bias

3.2.3.4.2 Inertial Extended Kalman Filter

Estimation of position and velocity is required to be able to traverse from one location to the next. The primary sensor used to determine position and velocity is GPS. The purpose of the filter to provide reasonable estimates of the state when there is a loss of the GPS signal, or when there is interference that results in erroneous GPS readings.

As with attitude estimation a Kalman filter will be used to provide the translational state estimates. It is possible to develop a filter that contains both the attitude and translational states however both are being kept separate for this thesis. The purpose behind the decoupling of the attitude and positional states is two-fold. One reason that will be discussed later is that the APM hardware does not have enough system RAM to be able to cope with the full state filter so by implementing them separately some of the states can still be filtered depending on what is desired. The second is that including both in the same filter can cause coupling issues that are difficult to tune.

3.2.3.4.2.1 Coordinate Systems

GPS provides coordinates in latitude, longitude, and are in the Earth-Centered-Earth-Fixed (ECEF) reference frame. The ECEF is not as intuitive and instead the North-East-Down (NED) reference frame is used in the filter.

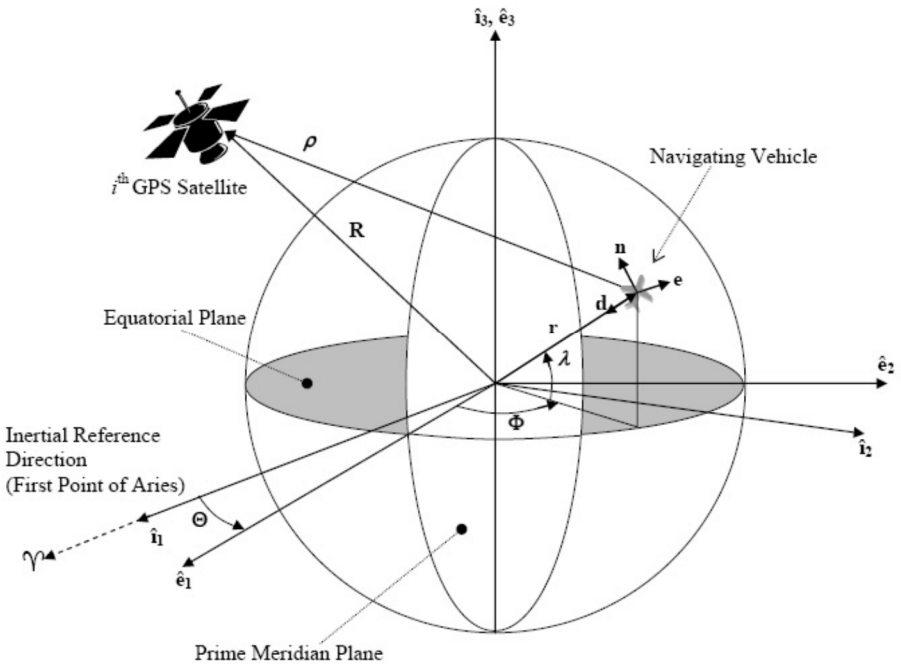

Figure 3: Definitions of Various Coordinate Systems [Centinello III, 2007]

3.2.3.4.2.2 Translational Kinematics

Equations for translational kinematics are taken from (Centinello III). In the equations given below h represents altitude, λ is the latitude given in radians, and Φ is the longitude also in radians. Velocities are in the NED reference frame. The continuous time state derivatives are given in equations (3.72) through (3.77). In the following sections the subscripts N, E, and D are used to indicate that the variables are with respect to the North (N), East (E), or Down (D) axis. ω_e is the rotational rate of the earth. Unless otherwise indicated velocity is represented by (v) and acceleration by (a).

$$\lambda = \frac{v_N}{R_\lambda + h} \tag{3.72}$$

$$\Phi = \frac{v_E}{\left(R_\Phi + h\right)\cos\lambda} \tag{3.73}$$

$$h = -v_D \tag{3.74}$$

$$v_N = -\left[\frac{v_E}{\left(R_\Phi + h\right)cos\lambda} + 2\omega_e\right] v_E \sin\lambda + \frac{v_N v_D}{\left(R_\lambda + h\right)} + a_N \tag{3.75}$$

$$v_E = -\left[\frac{v_E}{\left(R_\Phi + h\right)cos\lambda} + 2\omega_e\right] v_N \sin\lambda + \frac{v_E v_D}{\left(R_\lambda + h\right)} + 2\omega_e v_D \cos\lambda + a_E \tag{3.76}$$

$$v_D = -\frac{{v_E}^2}{R_\Phi + h} - \frac{{v_N}^2}{R_\lambda + h} - 2\omega_e v_E \cos\lambda + g + a_D \tag{3.77}$$

The X $\left(R_\lambda\right)$ and Z $\left(R_\Phi\right)$ location on the Earth's surface are found using the current latitude as well as the Earth's semi-major axis $\left(a\right)$ and eccentricity $\left(e\right)$.

$$R_\lambda = \frac{a\left(1 - e^2\right)}{\left(1 - e^2 \sin^2\lambda\right)^{3/2}} \tag{3.78}$$

$$R_{\Phi} = \frac{a}{\left(1 - e^2 \sin^2 \lambda\right)^{1/2}} \tag{3.79}$$

Gravity (g) is modeled in the NED frame using equation (3.80). The model constants are given in Table 9.

$$g = A\left(1 + B\sin^2 \lambda - C\sin^2 2\lambda\right) - \left(D - E\sin^2 \lambda\right)h + Fh^2 \tag{3.80}$$

Table 9: Gravity Model Constants

Constant	Value
A	32.185 (English) or 9.81 (Metric)
B	5.3024e-3
C	5.8e-6
D	3.0877e-6
E	4.4e-9
F	7.72e-14

3.2.3.4.2.3 Sensor Modeling

The GPS pseudo-range (ρ_i) is defined as the norm of the radius vector, taken in the ECEF reference frame, between the user $\left(r^E\right)$ and the individual satellite $\left(R_i^E\right)$ with a clock bias (β_c).

$$\rho_i = \left\| R_i^E - r^E \right\| + \beta_c + v, i = 1,2,...,n \tag{3.81}$$

Accelerometer measurements (a) are with respect to the body frame and are modeled as a function of true body acceleration (a), the accelerometer bias (β_a) and Gaussian noise (n_a).

$$a = a + \beta_a + \eta_a \tag{3.82}$$

3.2.3.4.2.4 EKF Model

The state (X) and state error (ΔX) vectors are given in equations (3.83) and (3.84). The (^) accent is used to indicate the estimated values.

$$X = \begin{bmatrix} \lambda & \Phi & h & v_N & v_E & v_D & \beta_{a_x} & \beta_{a_y} & \beta_{a_z} \end{bmatrix}^T \tag{3.83}$$

$$\Delta X = \begin{bmatrix} \Delta\lambda & \Delta\Phi & \Delta h & \Delta v_N & \Delta v_E & \Delta v_D & \Delta\beta_{a_x} & \Delta\beta_{a_y} & \Delta\beta_{a_z} \end{bmatrix}^T \tag{3.84}$$

The continuous-time derivative of the state error $\left(\Delta\hat{x}\right)$ is modeled as a function of the state-transition matrix (F) computed at time (t), the noise mapping matrix (G) and Gaussian noise (w).

$$\Delta\hat{x}(t) = F(\hat{x}(t), t)\Delta\hat{x} + G(t)w(t) \tag{3.85}$$

The state transition matrix is defined as:

$$F = \begin{bmatrix} \frac{\partial P}{\partial P} & \frac{\partial P}{\partial V_{NED}} & 0_{3x3} \\ \frac{\partial V_{NED}}{\partial P} & \frac{\partial V_{NED}}{\partial V_{NED}} & \frac{\partial V_{NED}}{\partial \beta_a} \\ 0_{3x3} & 0_{3x3} & \frac{\partial \beta_a}{\partial \beta_a} \end{bmatrix} \tag{3.86}$$

With the first-order partial derivatives given as:

$$\frac{\partial P}{\partial P} = \begin{bmatrix} -\frac{v_N}{(R_\lambda + h)^2}\frac{\partial R_\lambda}{\partial \lambda} & 0 & -\frac{v_N}{(R_\lambda + h)^2} \\ -\frac{v_E \sec\lambda}{(R_\Phi + h)^2}\frac{\partial R_\Phi}{\partial \lambda} + \frac{v_E \sec\lambda \tan\lambda}{R_\Phi + h} & 0 & -\frac{v_E \sec\lambda}{(R_\Phi + h)^2} \\ 0 & 0 & 0 \end{bmatrix} \tag{3.87}$$

$$\frac{\partial R_\Phi}{\partial \lambda} = \frac{ae^2 \sin\lambda\cos\lambda}{(1 - e^2 \sin^2\lambda)^{3/2}} \tag{3.88}$$

$$\frac{\partial R_\lambda}{\partial \lambda} = \frac{3a(1 - e^2)\sin\lambda\cos\lambda}{(1 - e^2 \sin^2\lambda)^{5/2}} \tag{3.89}$$

$$\frac{\partial P}{\partial V_{NED}} = \begin{bmatrix} \frac{1}{R_\lambda + h} & 0 & 0 \\ 0 & \frac{\sec\lambda}{R_\Phi + h} & 0 \\ 0 & 0 & -1 \end{bmatrix} \tag{3.90}$$

$$\frac{\partial V_{NED}}{\partial P} = \begin{bmatrix} Y_{11} & 0 & Y_{13} \\ Y_{21} & 0 & Y_{23} \\ Y_{31} & 0 & Y_{33} \end{bmatrix} \tag{3.91}$$

$$Y_{11} = -\frac{{v_E}^2 \sec^2\lambda}{R_\Phi + h} + \frac{{v_E}^2 \tan\lambda}{\left(R_\Phi + h\right)^2}\frac{\partial R_\Phi}{\partial\lambda} - 2\omega_e v_E \cos\lambda - \frac{v_N v_D}{\left(R_\lambda + h\right)^2}\frac{\partial R_\lambda}{\partial\lambda} \tag{3.92}$$

$$Y_{13} = \frac{{v_E}^2 \tan\lambda}{\left(R_\Phi + h\right)^2} - \frac{v_N v_D}{\left(R_\lambda + h\right)^2} \tag{3.93}$$

$$Y_{21} = -\frac{v_E \mathrm{v}_N \sec^2\lambda}{R_\Phi + h} - \frac{v_E \mathrm{v}_N \tan\lambda}{\left(R_\Phi + h\right)^2}\frac{\partial R_\Phi}{\partial\lambda} + 2\omega_e v_N \cos\lambda - \frac{v_E v_D}{\left(R_\Phi + h\right)^2}\frac{\partial R_\Phi}{\partial\lambda} - 2\omega_e v_D \sin\lambda \tag{3.94}$$

$$Y_{23} = -v_E\left[\frac{v_N \tan\lambda + v_D}{\left(R_\Phi + h\right)^2}\right] \tag{3.95}$$

$$Y_{31} = \frac{{v_E}^2}{\left(R_\Phi + h\right)^2}\frac{\partial R_\Phi}{\partial\lambda} + \frac{{v_N}^2}{\left(R_\lambda + h\right)^2}\frac{\partial R_\lambda}{\partial\lambda} + 2\omega_e v_E \sin\lambda + \frac{\partial g}{\partial\lambda} \tag{3.96}$$

$$Y_{33} = \frac{v_E^{\,2}}{\left(R_\Phi + h\right)^2} + \frac{v_N^{\,2}}{\left(R_\lambda + h\right)^2} + \frac{\partial g}{\partial h} \tag{3.97}$$

$$\frac{\partial g}{\partial \lambda} = A\left(2B\sin\lambda\cos\lambda - 4C\sin 2\lambda\cos 2\lambda\right) + \left(2E\sin\lambda\cos\lambda\right)h \tag{3.98}$$

$$\frac{\partial g}{\partial h} = -D + E\sin^2\lambda + 2F \tag{3.99}$$

$$\frac{\partial V_{NED}}{\partial V_{NED}} = \begin{bmatrix} \frac{v_D}{R_\lambda + h} & -\frac{2v_E\tan\lambda}{R_\Phi + h} + 2\omega_e\sin\lambda & \frac{v_N}{R_\lambda + h} \\ \frac{v_E\tan\lambda}{R_\Phi + h} + 2\omega_e\sin\lambda & \frac{v_D + v_N\tan\lambda}{R_\Phi + h} & \frac{v_E}{R_\Phi + h} + 2\omega_e\cos\lambda \\ -\frac{2v_N}{R_\lambda + h} & -\frac{2v_E}{R_\Phi + h} - 2\omega_e\cos\lambda & 0 \end{bmatrix} \tag{3.100}$$

$$\frac{\partial V_{NED}}{\partial \beta_a} = -I_{3x3} \tag{3.101}$$

$$\frac{\partial \beta_a}{\partial \beta_a} = -\frac{1}{\tau_a} I_{3x3} \tag{3.102}$$

The noise mapping matrix and weight matrix (Q) are constant:

$$G = \begin{bmatrix} 0_{3x3} & 0_{3x3} \\ 0_{3x3} & -I_{3x3} \\ 0_{3x3} & I_{3x3} \end{bmatrix}, Q = \begin{bmatrix} 0_{3x3} & 0_{3x3} & 0_{3x3} \\ 0_{3x3} & \sigma_a I_{3x3} & 0_{3x3} \\ 0_{3x3} & 0_{3x3} & \sigma_{\beta_a} I_{3x3} \end{bmatrix} \tag{3.103}$$

3.2.3.4.2.5 Modification for Use with the UDU EKF Algorithm

As with the MEKF derivation equations in the preceding section need to be discretized and parameterized for use with the available UDU algorithm. Assuming a small time step (dt) allows for the following approximation to be made:

$$\Delta\hat{x} = \hat{x} + \hat{x}dt = \Delta\hat{x} + \left[F(\hat{x},t)\Delta\hat{x} + Gw(t)\right]dt \tag{3.104}$$

$$\Delta\hat{x} = \left[\begin{bmatrix} I & 0 \\ 0 & I \end{bmatrix} + \begin{bmatrix} F_{\chi\chi} & F_{\chi p} \\ 0 & F_{pp} \end{bmatrix} dt\right]\Delta\hat{x} + Gw(t)dt \tag{3.105}$$

Using the nomenclature from the UDU_EKF class:

$$PHI_x = \begin{bmatrix} F_{xx}dt + I & F_{xp}dt \end{bmatrix} \tag{3.106}$$

$$PHI_X = \begin{bmatrix} \dfrac{\partial P}{\partial P}dt + I_{3x3} & \dfrac{\partial P}{\partial V_{NED}}dt & 0_{3x3} \\ \dfrac{\partial V_{NED}}{\partial P}dt & \dfrac{\partial V_{NED}}{\partial V_{NED}}dt + I_{3x3} & \dfrac{\partial V_{NED}}{\partial \beta_a}dt \end{bmatrix} \tag{3.107}$$

PHI_p, is diagonal and can be discretized directly by taking the exponential of the diagonal terms multiplied by the time delta. τ_a represents the accelerometer time constant.

$$PHI_p = I_{3x3} e^{\frac{dt}{\tau_a}} \tag{3.108}$$

Lastly:

$$G_x = \begin{bmatrix} 0_{3x3} & 0_{3x3} \\ 0_{3x3} & -I_{3x3} dt \end{bmatrix} \tag{3.109}$$

3.2.3.4.2.6 Inertial Observations

GPS is the primary sensor that is used to correct the translational state and provides direct measurements of latitude, longitude, and altitude. Additionally it also provides a ground speed $(|V|)$ and the ground course (ψ) that can be used to correct the velocities. Using the state defined in (3.83) the sensitivity matrix for updating latitude:

$$H_{Latitude} = \frac{\partial X}{\partial \lambda} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.110}$$

Longitude:

$$H_{Longitude} = \frac{\partial X}{\partial \Phi} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.111}$$

Altitude:

$$H_{Altitude} = \frac{\partial X}{\partial h} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.112}$$

The sensitivity matrix for updating NED velocities using the velocity magnitude is:

$$H_{Ground\_Speed} = \frac{\partial X}{\partial |V|} = \begin{bmatrix} 0 & 0 & 0 & \frac{V_N}{|V_{NED}|} & \frac{V_E}{|V_{NED}|} & \frac{V_D}{|V_{NED}|} & 0 & 0 & 0 \end{bmatrix} \quad (3.113)$$

Updating NED velocities using the ground course:

$$H_{Ground\_Course} = \frac{\partial X}{\partial \psi} = \begin{bmatrix} 0 & 0 & 0 & \frac{1}{V_N\left(1+\frac{V_E^2}{V_N^2}\right)} & \frac{-V_E}{V_N^2\left(1+\frac{V_E^2}{V_N^2}\right)} & 0 & 0 & 0 & 0 \end{bmatrix} \quad (3.114)$$

3.2.3.4.2.7 Updating the State Estimate

The state estimate is updated using the estimate of the state error generated by the measurement updates. After updating the state the estimate of the error is reset to zero:

$$\hat{X}^{+} = \hat{X}^{-} + \Delta\hat{x} \quad (3.115)$$

3.2.3.4.2.8 Implementation

Utilizing the UDU_EKF library class and the EKF derivation given in the previous section a translational filter has been constructed:

Table 10: INRTL_Geod_EKF Class

INRTL_EKF	
VARIABLES	
Accel_Variance	Variance of the accelerometer measurements
Accel_Bias_Variance	Variance of the accelerometer bias
Accel_Time_Constant	Time constant of the accelerometer
Filter	UDU_EKF declaration used for Kalman filtering
FUNCTIONS	
Initialize(Accel_Variance, Accel_Bias_Variance, Accel_Time_Constant)	Initialize the estimator Inputs are the accelerometer sensor parameters
Clear()	Clear all allocated memory Filter is no longer valid
Clear_Workspace()	Clear workspace memory to save memory Maintains filter states
Initialize_Workspace()	Initialize workspace variables
Propogate (Latitude, Longitude, Altitude, V_NED, Accel_Bias, a_B, Quat_BNED, dt)	Propagate the state and covariance. Inputs are the current state, a time step, and the attitude quaternion.
calcLinSys(Latitude, Altitude, V_NED, dt)	Calculate the discrete-time system matrices used by the UDU filter (PHI_x, PHI_p, Q_x, Q_p, G_x)
Geod_Update(Lat_Measured, Lat_Epected, Lat_Variance, Lon_Measured, Lon_Expected, Lon_Variance)	Update the latitude and longitude error estimates
Altitude_Update(Alt_Measured, Alt_Expected, Variance)	Update the altitude error estimate using a measurement.
Vmag_Update(Vmag_Measured, V_NED, Variance)	Update the V_NED error estimate using a measurement of the velocity magnitude.
Update_Estimate(Latitude, Longitude, Altitude, V_NED)	Update the estimate of the translational state and the accelerometer bias

3.2.3.5 Hardware and Sensor Classes

Hardware is what allows software to interact with the external world and can come in a seemingly infinite number of configurations. Even when hardware is intended for the same purpose, reading acceleration for example, it often is accessed quite differently between manufacturers and requires custom drivers.

To facilitate interaction between software and hardware, the hardware interface must be abstracted. This is accomplished in JARVIS by creating a base class for each sensor. These classes are then used to create sensor-specific classes that are then incorporated into a master Hardware class. This operates under the assumption that between different sensors of the same classification the only model-specific code relates to initialization and reading of the sensor. This allows for all hardware functionality to be accessed through the Hardware class and prevents the need for any software changes when using different hardware platforms.

3.2.3.5.1 Hardware

The hardware class is made up of the sensor classes and can contain any additional functionality necessary to initialize the platform.

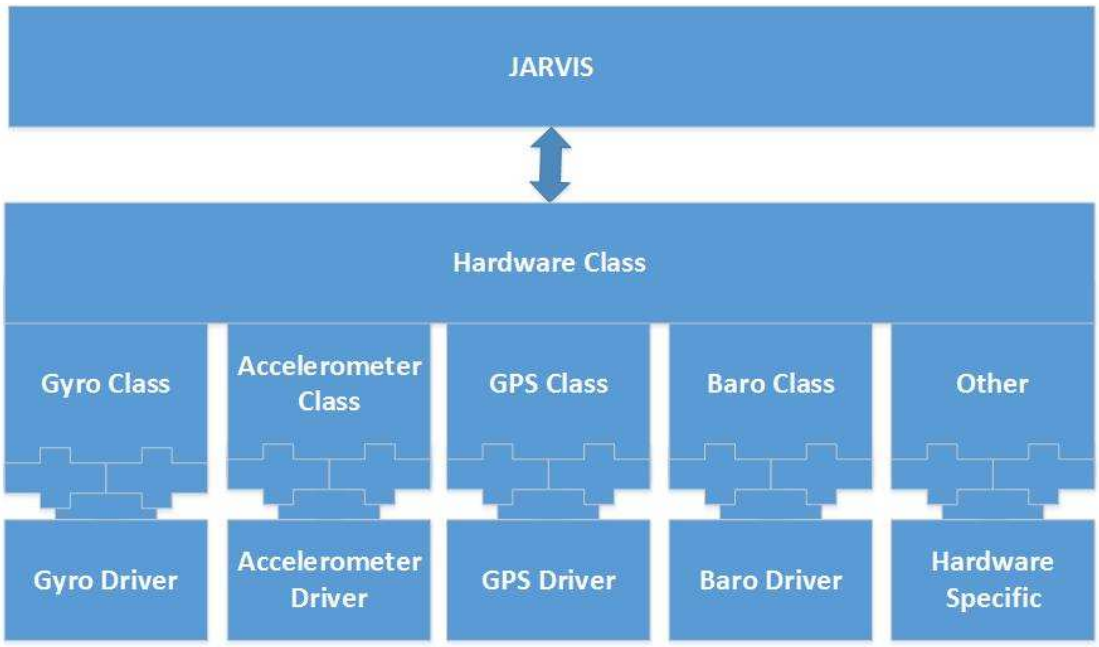

Figure 4: Hardware Abstraction Diagram

3.2.3.5.2 Accelerometer

Table 11: Accelerometer Base Class

Accelerometer_Class	
VARIABLES	
Enabled	Flag to indicate if the sensor is enabled
Healthy	Flag to indicate the sensor health
Initialized	Flag to indicate if the sensor has been initialized
Reading	Array containing the sensor reading
FUNCTIONS	
getReading()	Return the last sensor reading
getCalibration()	Return the calibration values
setCalibration(Calibration)	Set the calibrated values
setOrientation(Psi, Theta, Phi)	Set the orientation of the sensor with respect to the body axis
Calibrate()	Called after the sensor is initialized Calibrates the output

3.2.3.5.3 Gyroscope

Table 12: Gyro Base Class

Gyro_Class	
VARIABLES	
Enabled	Flag to indicate if the sensor is enabled
Healthy	Flag to indicate the sensor health
Initialized	Flag to indicate if the sensor has been initialized
Reading	Array containing the sensor reading
FUNCTIONS	
getReading()	Return the last sensor reading
getCalibration()	Return the calibration values
setCalibration(Calibration)	Set the calibrated values
setOrientation(Psi, Theta, Phi)	Set the orientation of the sensor with respect to the body axis
Calibrate()	Called after the sensor is initialized Calibrates the output

3.2.3.5.4 Compass

Table 13: Compass Base Class

Compass_Class	
VARIABLES	
Enabled	Flag to indicate if the sensor is enabled
Healthy	Flag to indicate the sensor health
Initialized	Flag to indicate if the sensor has been initialized
Reading	Array containing the sensor reading
FUNCTIONS	
getReading()	Return the last sensor reading
getCalibration()	Return the calibration values
setCalibration(Calibration)	Set the calibrated values
setOrientation(Psi, Theta, Phi)	Set the orientation of the sensor with respect to the body axis.
Calibrate()	Called after the sensor is initialized Calibrates the output

3.2.3.5.5 Baro

Table 14: Baro Base Class

Baro_Class	
VARIABLES	
Enabled	Flag to indicate if the sensor is enabled
Healthy	Flag to indicate the sensor health
Initialized	Flag to indicate if the sensor has been initialized
Ground_Pressure	Initial pressure reading at the ground, altitude is calculated as above-ground-level (AGL)
Ground_Temperature	Initial temperature reading at the ground
Pressure	Pressure reading
Temperature	Temperature reading
Altitude	Calculated altitude
FUNCTIONS	
getAltitude()	Calculate and return the altitude
getPressure()	Return the pressure reading
getTemperature()	Return the temperature reading
Calibrate()	Called after the sensor is initialized Calibrates the output

3.2.3.5.6 GPS

Currently the GPS library is nearly unchanged from the one provided with the ArduCopter software. A base class has been created although is currently unused.

Table 15: GPS Base Class

GPS_Class	
VARIABLES	
Enabled	Flag to indicate if the sensor is enabled
Healthy	Flag to indicate the sensor health
Initialized	Flag to indicate if the sensor has been initialized
New_Data	Flag to indicate that new data is available
GPS_Lock	Flag to indicate that the GPS has a lock
Latitude	Latitude reading
Longitude	Longitude reading
Altitude	Altitude reading
Ground_Speed	Ground speed reading
Ground_Course	Ground course reading
FUNCTIONS	
N/A	N/A

3.2.4 Code Validation

With the complexity of an integrated system and the number of required library functions used it is important to be able to validate the outputs of the functions independently as well as the integrated system.

3.2.4.1 Library Functions and Classes

Each function or class was first prototyped in the MatLab environment. Once prototyped the new item was called and the output compared to any existing MatLab function. For items without a MatLab counterpart outputs were compared to any number of other sources including hand calculations and online calculators.

The prototype was then converted into the C++ language and added to the library. A test program is then written that calls the library function and prints the output to the screen or a file. This output is then compared to the output of the original prototype.

Once the code had been verified off-line it was considered ready to be used on the flight hardware. Functionality on hardware was validated by creation of a unit test program, or by use of debug print statements.

3.2.4.2 Integrated System

Even with the library functions performing as intended the integrated system can present unknown irregularities. Testing of the integrated system is performed using the MatLab implementation of JARVIS, by compiling the JARVIS code on an x86 PC, or by running it on the hardware. A *makefile* has been created that allows JARVIS to be compiled easily on a Unix-based system or on Windows using Cygwin[5].

[5] A large collection of GNU and Open Source tools which provide functionality similar to a Linux distribution on Windows (cygwin.com)

A simulation has been constructed for the off-line (x86/MatLab) code that allows the software to be driven through either logged sensor data, or by using a 6-DoF simulation with the sensors being modeled. The ability to reprocess logged data is powerful as it allows the user to log a minimal set of data while being able to fully reconstruct the internal signals when analyzing vehicle performance.

CHAPTER 4. GUIDANCE

The Guidance method currently being employed is a linear guidance algorithm. Here the pilot command is interpreted as a rate command that is augmented by guidance. Guidance generates a delta command from a gain matrix and a state (X) error.

Note that with the system architecture chosen Guidance should not augment the rate command with rate errors, instead only with positional errors. It is intended that control will interpret the guidance command as a rate command and compute the rate error internally.

The linear Guidance routine is implemented as follows:

$$X_{Error} = X_{Desired} - X_{Actual} \tag{4.1}$$

$$X_{Command} = KX_{Error} \tag{4.2}$$

$$X_{Command} = X_{Pilot\_Command} + X_{Command} \tag{4.3}$$

The Guidance gain matrix can be set to zero allowing for direct tuning of the rate controller. Afterwards the states can be introduced incrementally by adjusting elements in the Guidance gain matrix.

CHAPTER 5. NAVIGATION

Navigation is divided into subsystems that follow the architecture outlined in section 3.1.1.3. Each subsystem is responsible for producing estimates of desired states such as the attitude and position. Dividing navigation in this manner allows for individual tuning of the estimators as well as the expansion or contraction of the states being estimated with little additional work. Currently an attitude and translational navigation system have been created. Future iterations will include an atmosphere-relative navigation system.

5.1 Attitude

Attitude navigation is built upon the AHRS_MEKF class. The measurement updates occur at 2Hz and follow the logic flow shown in Figure 5.

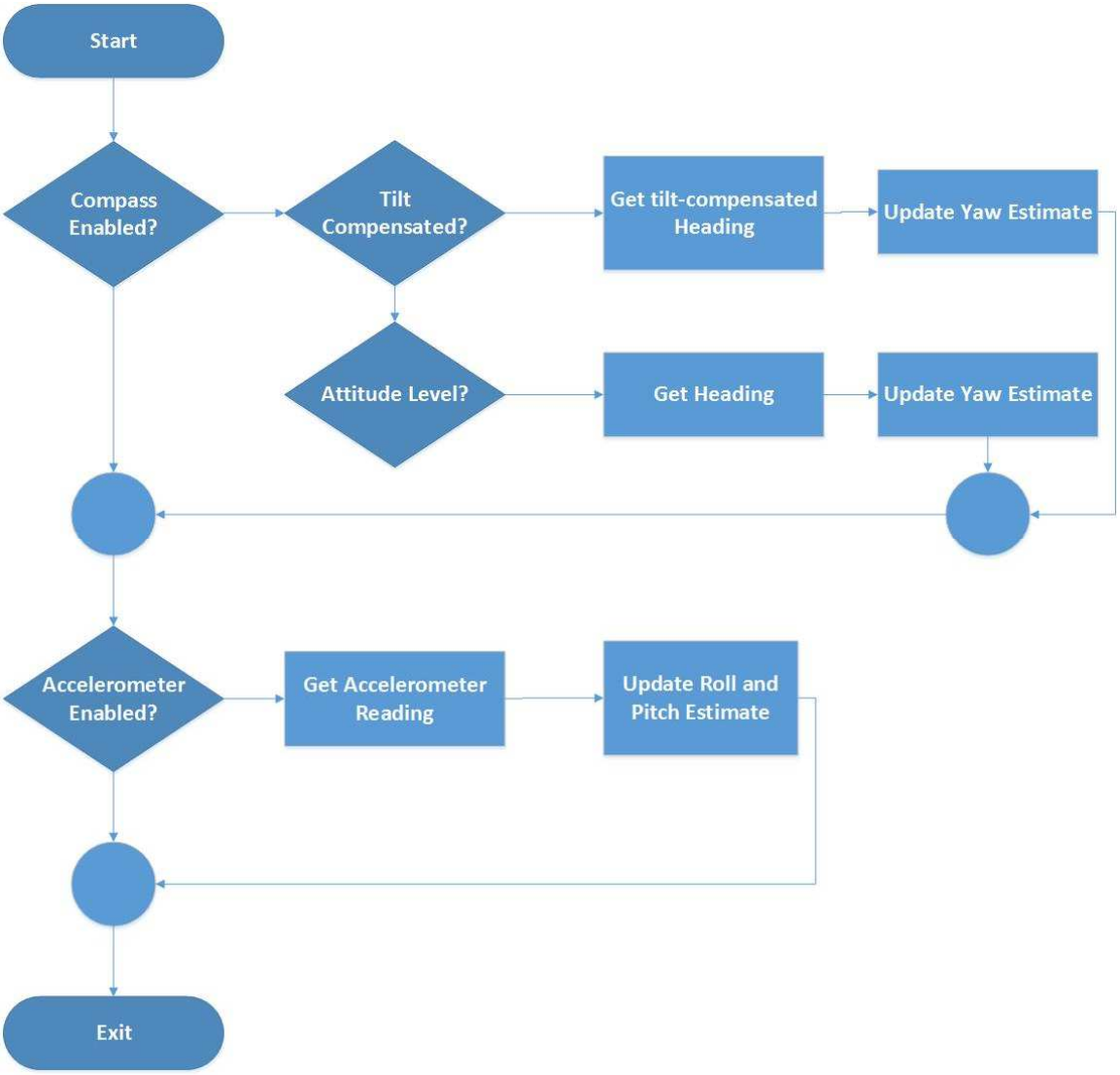

Figure 5: Navigation AHRS: Measurement Update Logic Flow

State prediction is called at 50Hz and follows the logic flow shown in Figure 6.

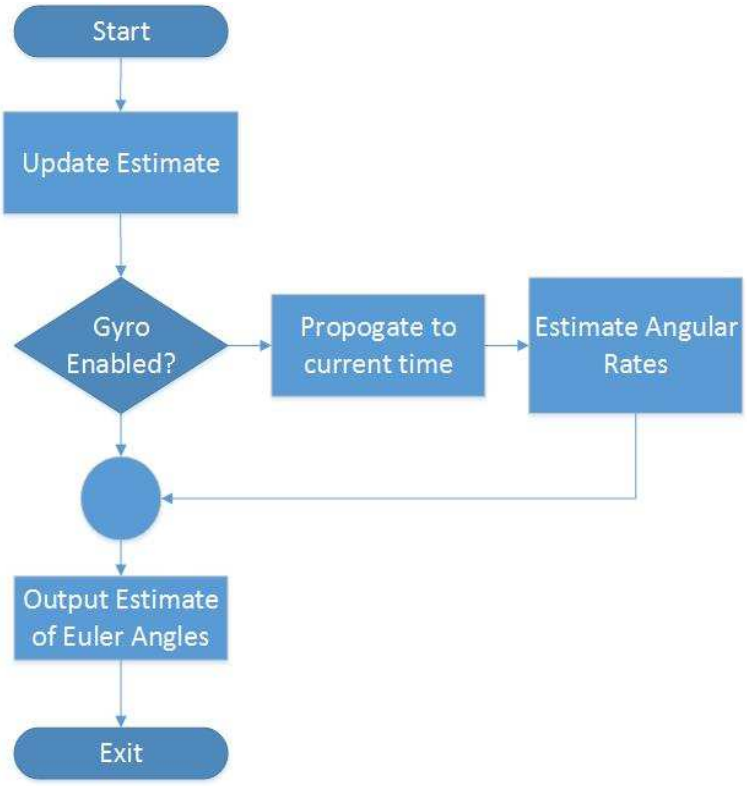

Figure 6: Navigation AHRS: Predict State Logic Flow

5.1.1 Simulated Results

The first set of results was generated in MatLab using modeled sensor data for a stationary rigid body. The AHRS class performs as intended and provides reasonably accurate estimates of the attitude and rates that are within the 3-sigma boundaries.

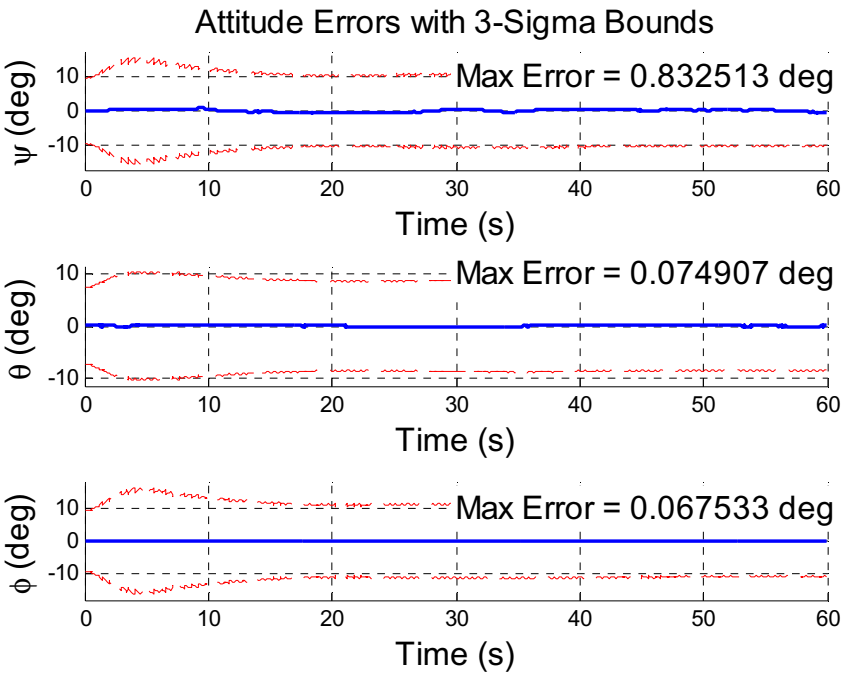

Figure 7: Attitude Error with 3-Sigma Bounds

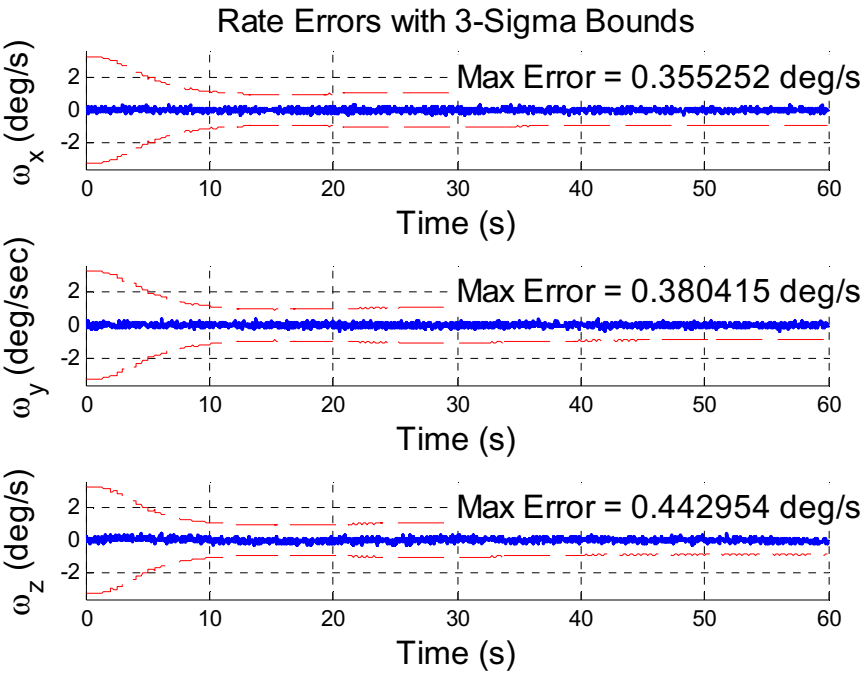

Figure 8: Rate Error with 3-Sigma Bounds

5.1.2 Flight-Results

The figures below were generated from logged flight data. Gyroscope, accelerometer, and compass sensors were used in the filter. Unfortunately a rate table or similar device was not available preventing truth data to be collected. Instead the hardware was rotated by hand 360 degrees about each axis individually starting with the Z-body axis.

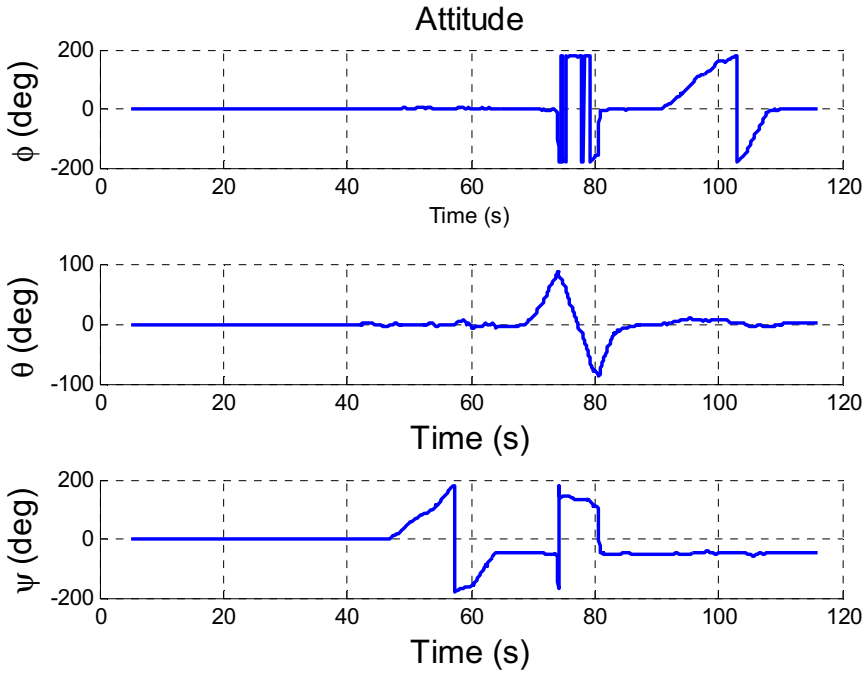

Figure 9: Flight-Test: Attitude Estimate

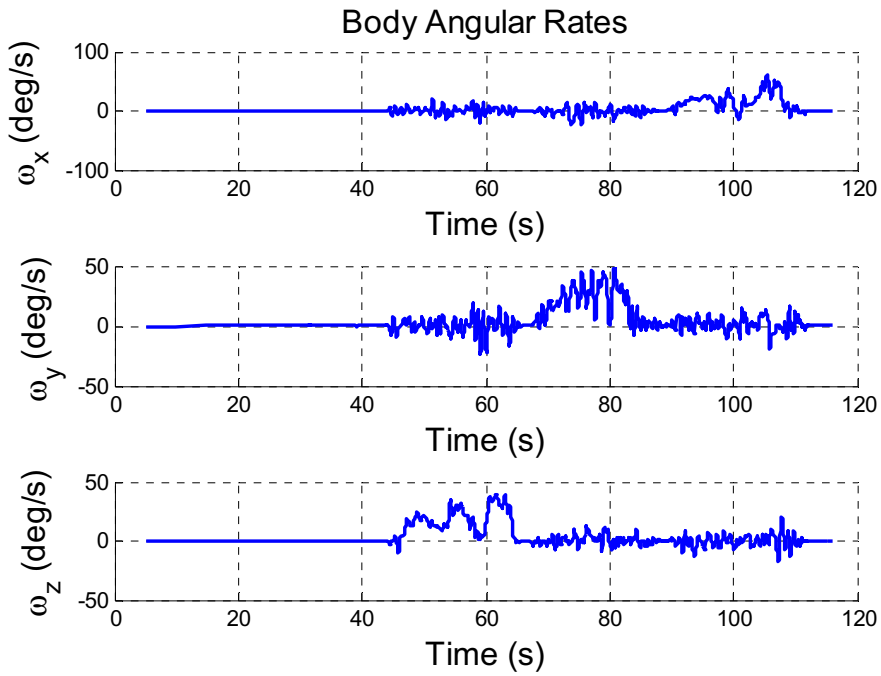

Figure 10: Flight-Test: Rate Estimate

The attitude estimate appears to be accurate and behaves as desired. This result, paired with the simulated results, provides evidence that the system is working as intended and is well-behaved. During this test the ArduPilot was able to output a new state estimate and control command at a rate of approximately 30Hz, this is impressive considering the software is operating on a 16MHz processor with only 8 kilobytes of system RAM.

5.2 Inertial Estimation

The navigation subsystem responsible for producing estimates of position and velocity is built using the INRTL_Geod_EKF class. State prediction is performed at 50Hz and measurement updates at 1Hz. Logic flow is shown in Figure 11 and Figure 12.

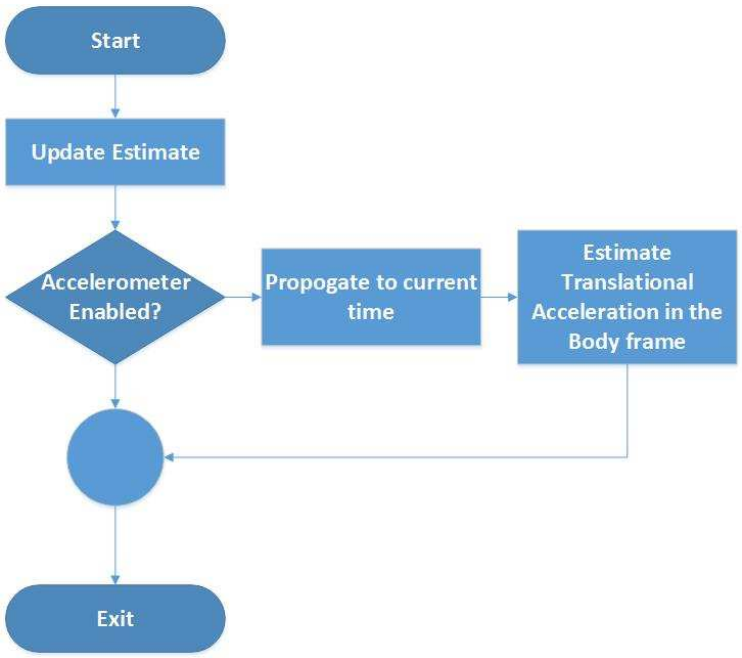

Figure 11: Navigation INRTL: Predict State Logic Flow

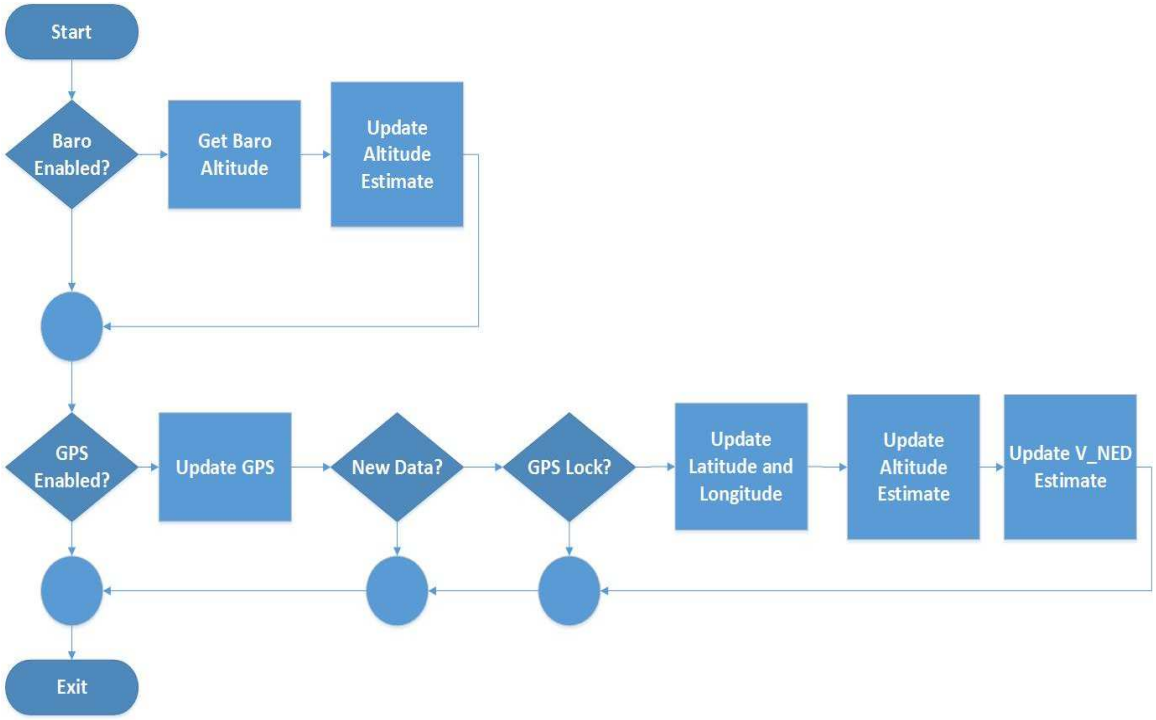

Figure 12: Navigation INRTL: Measurement Update Logic Flow

5.2.1.1 Simulated Results

JARVIS is tested in a MatLab simulation environment. The initial state includes velocities in the North and East direction, the only external force acting on the body is that due to gravity. In this test the filter is able to converge within the 3-Sigma bounds for position, velocity, and acceleration within sixty seconds.

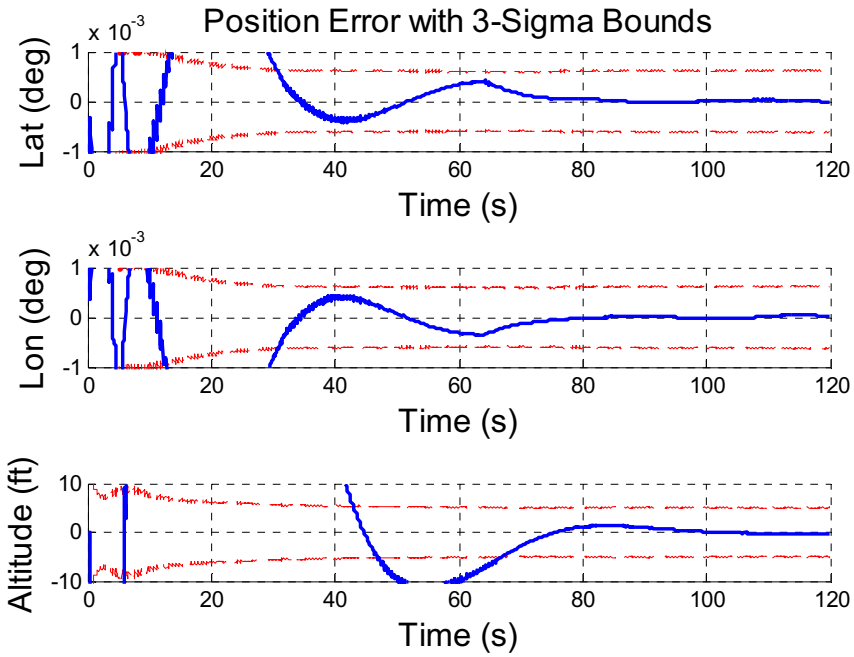

Figure 13: Geodetic Position Error with 3-sigma Bounds

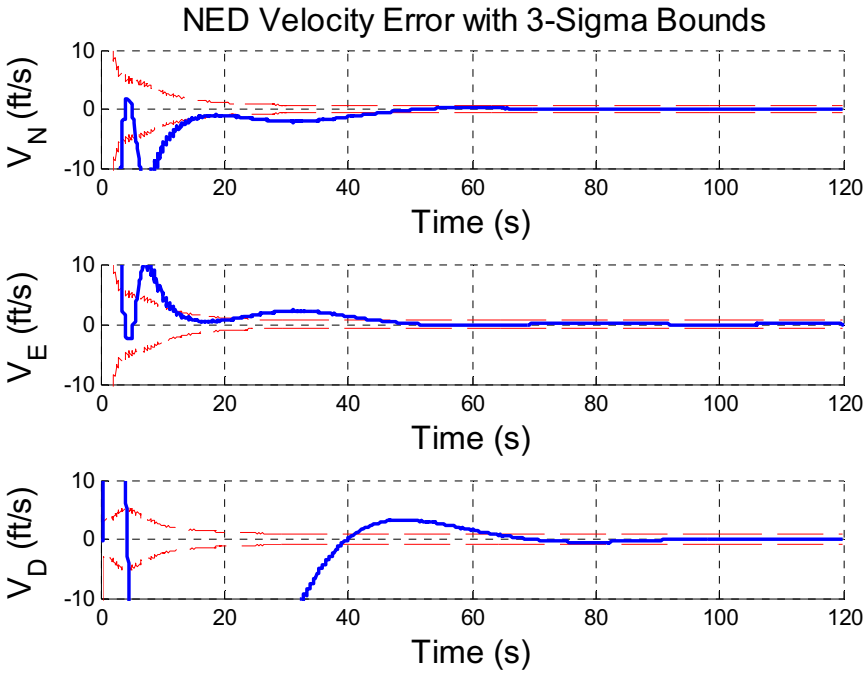

Figure 14: NED Velocity Error with 3-sigma Bounds

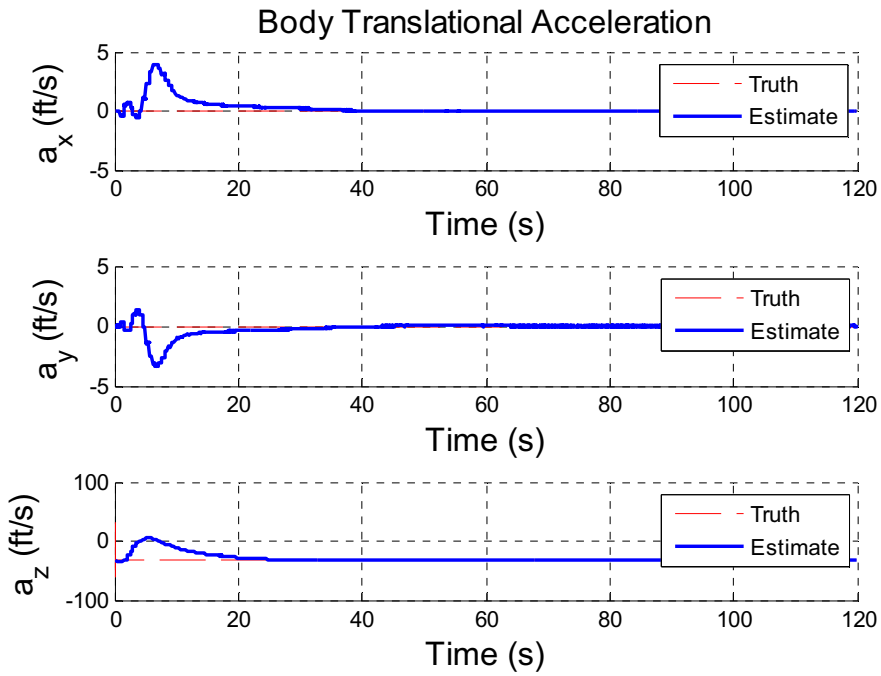

Figure 15: Estimated Translational Acceleration

5.2.1.2 Flight-Results

Flight results are currently unavailable. The C++ code has been unit tested and verified for functionality, however the ArduPilot system is unable to operate with both the AHRS and INRTL navigation subsystems. This primary issue is that the on-board processer does not contain enough RAM and the ArduPilot hardware locks when performing the measurement updates. Efforts were made to remove any variables not absolutely necessary for operation as well as clearing any workspace variables between measurement update and state estimation algorithm calls.

5.3 Conclusion

While it is disappointing that this entire state cannot be filtered using the classes created on the ArduPilot hardware it does show the capability of JARVIS in adding and removing functionality.

The filter is performing as expected in the simulated results and once a more powerful platform is selected the inertial navigation subsystem can easily be inserted back into the system.

CHAPTER 6. VEHICLE TESTING

6.1 Gimbaled Tri-Ducted Fan

6.1.1 Motivation

Historically the idea of building a vertical take-off and landing (VTOL) aircraft came about due to the desire to create a single stage to orbit (SSTO). The VTOL was the first concept contrived which was deemed able to demonstrate the benefits of an SSTO. With the recent advances in technology and UAV capabilities there is renewed interest in developing a highly maneuverable UAV. The success of the Osprey program, along with a fondness for futuristic aircraft found in science fiction films, it was conceived to build a tri-tilt turbine VTOL aircraft. It is the desire to make such a vehicle reality that drove the development of JARVIS.

6.1.2 Vehicle Concept

The concept of this vehicle is to have three micro-turbine engines that are gimbaled about a single axis so as to provide thrust vectoring illustrated in Figure 16. The configuration is similar to the Osprey tilt-rotor aircraft that is deployed by the United States military.

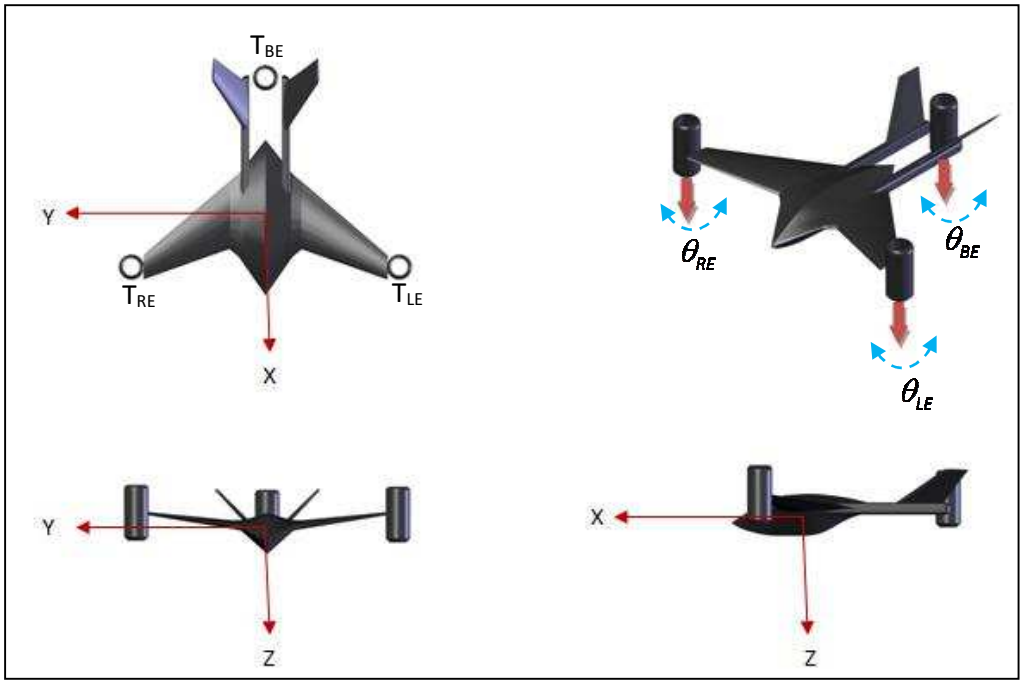

Figure 16: Tri-Duct Concept and Control Effectors

6.1.3 Hover Prototype

A test bed was designed and fabricated for the purpose of testing vertical flight and hover. Due to the minimal thrust capability of the power plants used for the test bed it was deemed necessary to position them in the form of an equilateral triangle. This configuration allows for each ducted-fan to make an equal contribution towards lifting the vehicle.

The frame is of aluminum construction and lift is provided by three electric 56mm ducted fans that are gimbaled. Thrust direction is controlled by three digital servos. The ArduPilot 1.0 board provides the sensor array and on-board processing used to generate the navigated state and control commands.

The ducted-fans are located 11 inches from the center and separated by 120 degrees and the platform stands 2 ¾ inches from the ground.

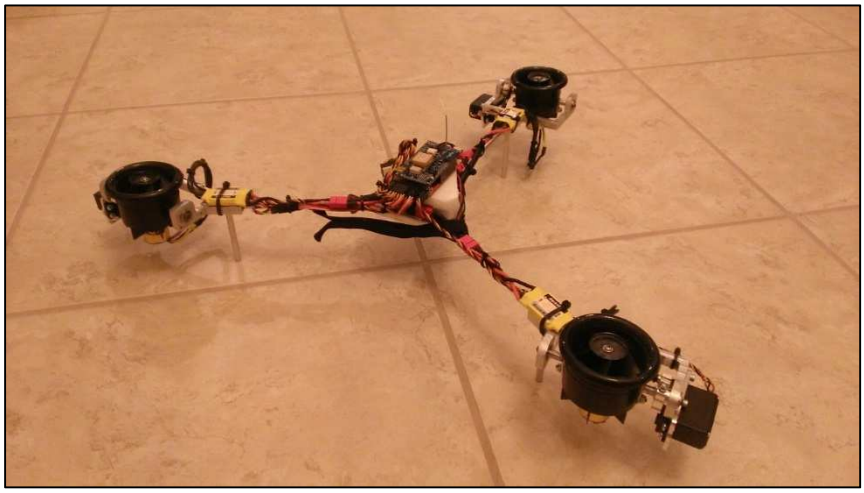

Figure 17: Tri-Duct Hover Prototype

6.1.4 Control Configuration

The goal of the controller in this iteration is to provide attitude stabilization and allow a person to pilot the vehicle externally with commands sent through an R/C transmitter. The state vector (X) to be controlled consists of roll (ϕ), pitch (θ), and yaw (φ):

$$X = [\phi \quad \theta \quad \varphi]^T \tag{6.1}$$

Control output (U) is the engine throttle and moments about the body axis (M_x, M_y, M_z):

$$U = \begin{bmatrix} Throttle & M_x & M_y & M_z \end{bmatrix}^T \tag{6.2}$$

PID control was the method selected to generate commands although any form of controller could be selected. Control commands are then realized through the engine thrust and gimbal angles as given in Figure 16. The control output-to-effector map is:

$$T_U^{Effector} = \begin{bmatrix} 1.0 & -0.5 & 0.0 & 0.0 \\ 1.0 & 0.0 & 1.0 & 0.0 \\ 1.0 & 0.5 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.5 \\ 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & -0.5 \end{bmatrix} \tag{6.3}$$

The output to the effectors is then:

$$U_{Effector} = \begin{bmatrix} T_{RE} & T_{BE} & T_{LE} & \theta_{RE} & \theta_{BE} & \theta_{LE} \end{bmatrix}^T = T_U^{Effector} U \tag{6.4}$$

6.1.5 Simulated Results

A simple model for the vehicles was developed to work within the MatLab rigid body simulation that was used previously to demonstrate JARVIS navigation performance. Future work will be to increase the fidelity of the simulation and tune the models to match the physical dynamics of the system. The simulation is only intended to demonstrate proper functionality of JARVIS and provide an initial control configuration as the intent of this research is not simulation development or control tuning.

The model produces body forces and moments using the effector commands output from JARVIS that and are then fed into the rigid body dynamics. Currently the motor and servo models contain zero error due to effector delay or hardware mounting. A two percent Gaussian dispersion was applied to each thruster individually so that the output thrust magnitude is not ideal. The simulated software and flight software operate identically as that was one of the goals when developing JARVIS.

As the current hardware is unable to filter the translational state the initial goal is to provide attitude stabilization to the system. This would then allow a pilot to manually direct the position. A simulation run was performed that suppresses control for the initial 15 seconds to allow navigation to converge after which a twenty degree step input is applied in the roll channel.

Channel response is shown in Figure 18, Figure 19, and Figure 20. In these simulations position is ignored.

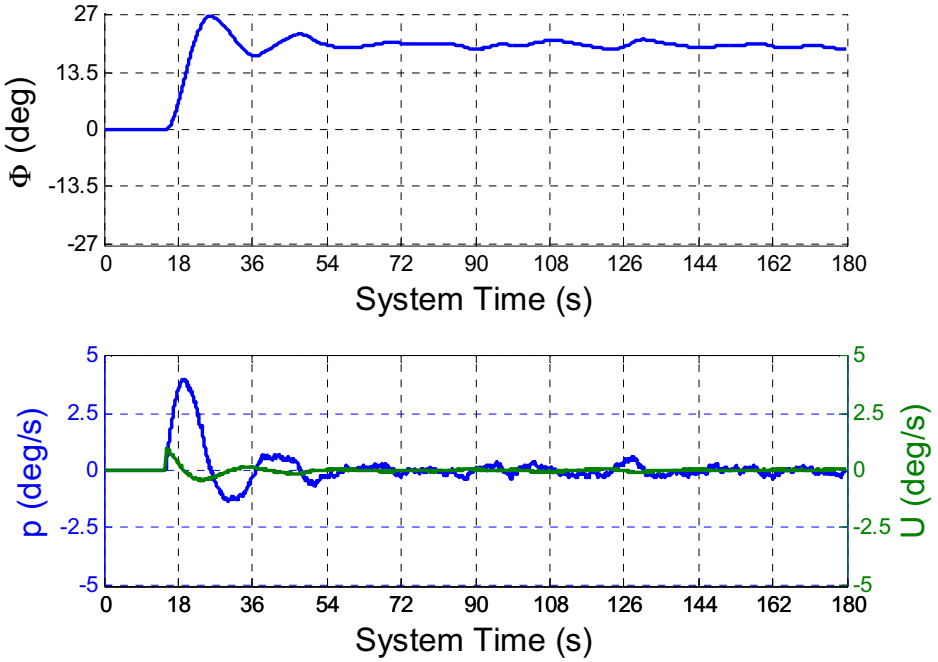

Figure 18: Ducted-Fan Simulation: Roll Step Response

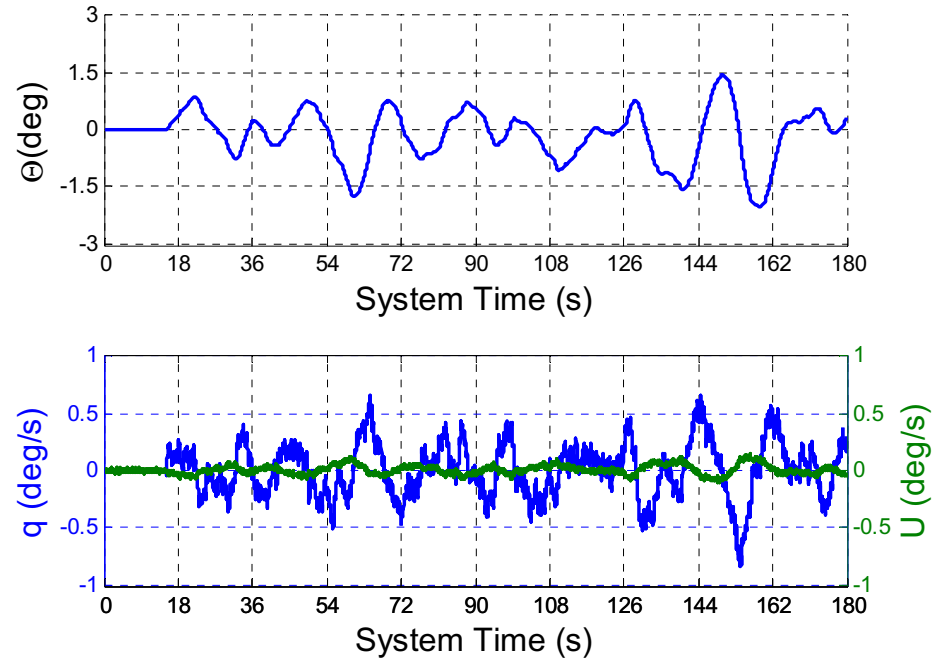

Figure 19: Ducted-Fan Simulation: Pitch Channel

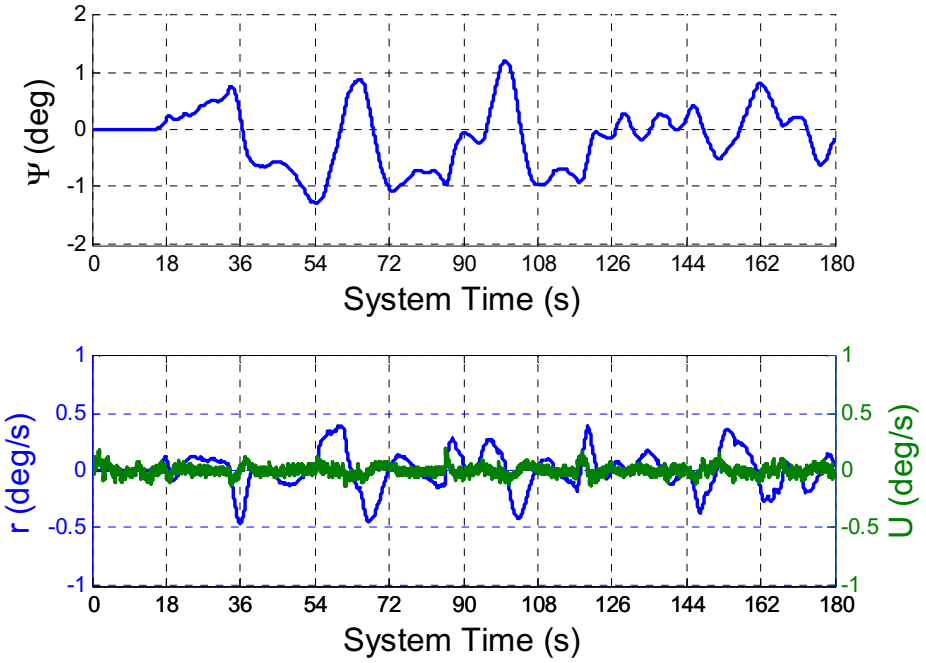

Figure 20: Ducted-Fan Simulation: Yaw Channel

The state controller is able to maneuver the vehicle to the desired roll angle while keeping both pitch and yaw errors within two degrees. There is a small overshoot seen in the roll channel so as a verification check the roll rate gain was increased so as to over-damp the system. As expected the overshoot was eliminated as shown in Figure 21.

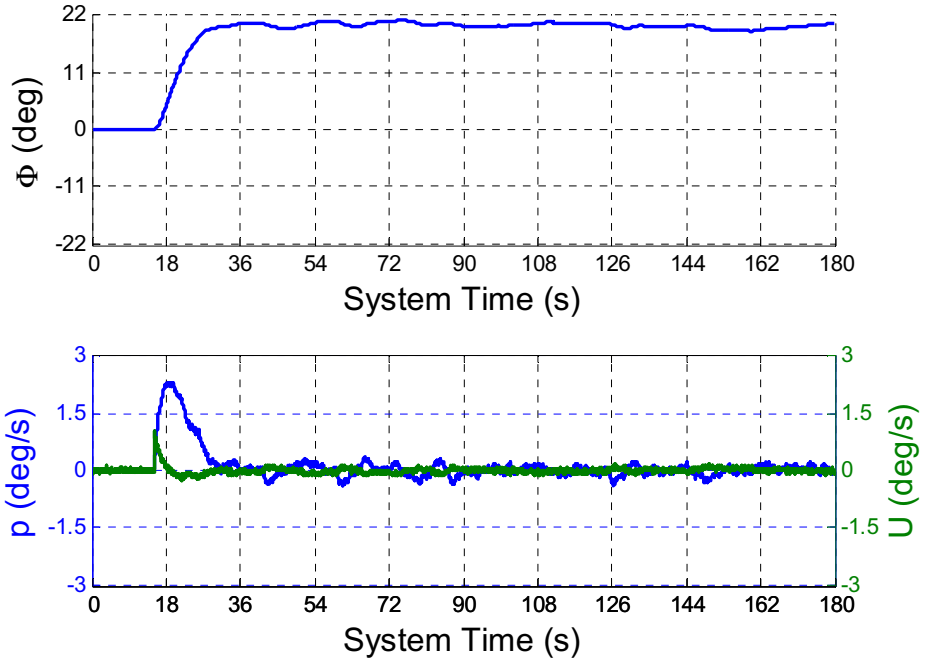

Figure 21: Ducted-Fan Simulation: Roll Step Response with Larger Rate Gain

6.1.6 Flight Test

JARVIS was uploaded to the ArduPilot using the configuration found during the simulation runs. During flight the vehicle is able to move in six degrees of freedom and initial attempts to pilot the vehicle with attitude control proved to be marginally successful.

A secondary approach was taken that restricted all motion except about the roll axis to allow for testing of roll control. For this test after navigation convergence an external disturbance was applied about the roll axis. The roll channel for this test is shown in Figure 22.

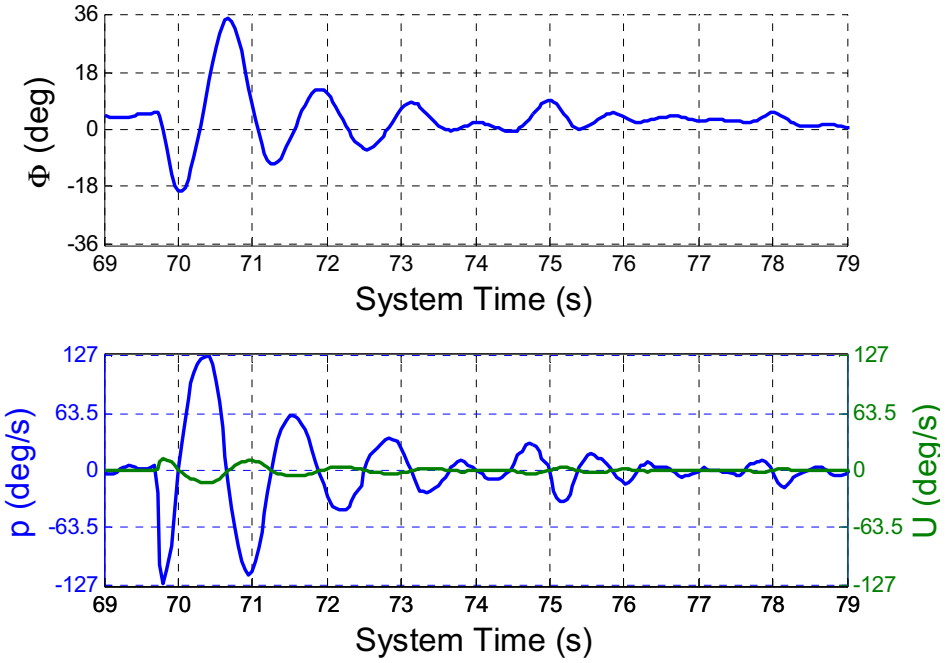

Figure 22: Tri-Ducted Fan Flight with Roll Disturbance

JARVIS is able to stabilize the roll channel using the gains found during simulation. The dynamic response is much quicker than what was shown in simulation. While this is not unexpected it does go to show how simulated dynamics may not match dynamics seen in flight. Further tuning is necessary before the platform performs as intended.

6.2 Helicopter

6.2.1 Motivation

Radio-controlled helicopters have been around for quite some time and have improved significantly within the past decade. Electric models are simple to use and with the latest battery technology the larger variants can handle payloads of approximately 10 pounds.

3D helicopters are able to translate in all directions, fly upside down, and can hover. This allows for robust testing GN&C algorithms and sensors in both single and multi-vehicle configurations. Unfortunately these vehicles are difficult to pilot and require hours of training and practice to become proficient in their operation. Using JARVIS the goal is to remove this barrier and provide a stable platform that is simple to pilot. For the purposes of this research moving to a helicopter will demonstrate the robustness of JARVIS in its ability to operate any number of vehicles with updates to the configuration file being the only modifications necessary.

6.2.2 Frames and Notation

The helicopter is considered to be a rigid body that is able to freely move in 6-DoF space. The inertial-fixed frame that will be used is a right-hand North-East-Down (NED) frame where the earth is assumed to be flat.

The body-fixed frame is centered at the body center of mass with X-positive towards the nose, Y-positive to the right, and Z-positive down, shown in Figure 23.

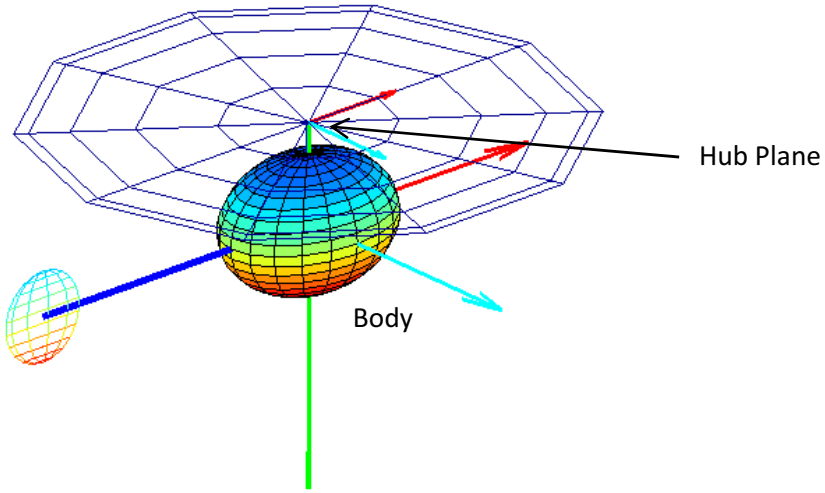

Figure 23: Body and Hub Frame Definition

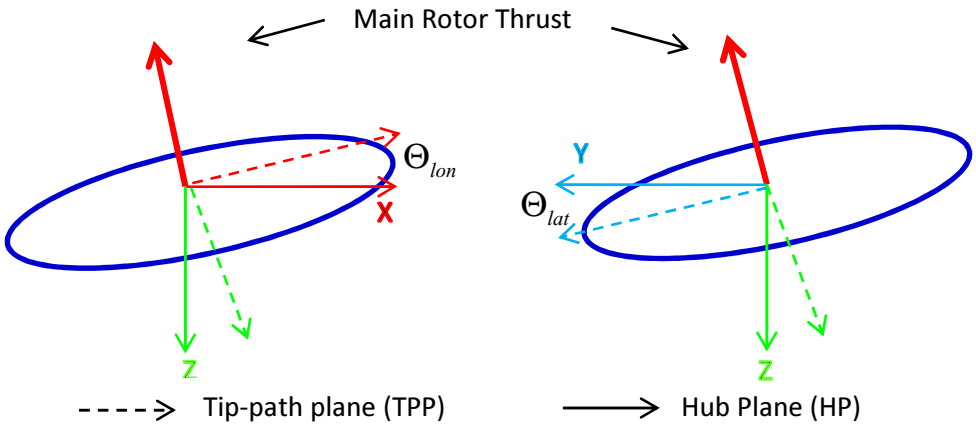

Figure 24: Hub Plane, Tip-path Plane, and Main Rotor Thrust Vector

The hub plane originates at the center of the main rotor hub and is aligned with the body axis. The Tip-path plane is the plane that traces the path of the tip of the main rotor blade as it rotates and is centered at the hub. The main rotor thrust vector is normal to the tip-path plane with the origin at the center of the hub.

6.2.3 Mechanical Overview

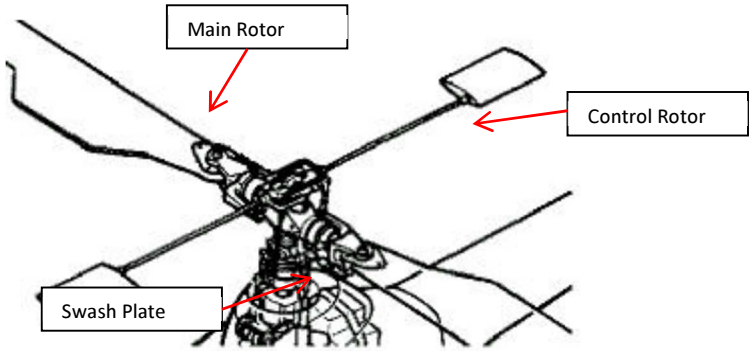

Figure 25: Helicopter Mechanical Overview (Munzinger)

The primary components for a helicopter hub assembly are the swash plate, the main rotor itself, and the control rotor. Control inputs enter the main rotor system through the swash plate. Linkages connected to servos are moved up and down changing the “pitch,” “roll,” and z-axis location of the swash plate. The position of the swash plate dictates what the pitch angle of the blade is as it travels around the hub thus changing the main rotor force vector magnitude and direction providing maneuverability. The control rotor is used to damp vehicle dynamics and is not controlled directly. The control rotor is becoming obsolete and is being replaced with direct-flight-control (DFC) systems that use 3-axis gyroscopes to electronically damp the dynamics.

6.2.4 LQR Control

A traditional helicopter has five control inputs that can be commanded by the pilot, these are:

- Lateral δlat_{MR} : Increases main rotor blade pitch on right or left side of helicopter causing the helicopter to rotate about the x-body axis (roll)
- Longitudinal δlon_{MR} : Increases main rotor blade pitch in front or back causing the helicopter to rotate about the y-body axis (pitch)
- Collective δcol_{MR} : Increases or decreases main rotor blade pitch uniformly causing the helicopter to move up or down along the z-body axis

- Rudder δcol_{TR} : Increases or decreases tail rotor blade pitch uniformly causing the helicopter to rotate about the z-body axis
- Main rotor speed Ω_{MR} : Typically this is held to a constant value

For R/C helicopters the main rotor speed is held constant by a governor and the yaw rate is damped out by an onboard gyroscope. This leaves three control inputs for the controller to use to maintain hover:

$$U = \begin{bmatrix} \delta col_{MR} & \delta lat_{MR} & \delta lon_{MR} \end{bmatrix} \tag{6.5}$$

The state to be controlled is roll (ϕ) , pitch (θ) , roll rate (p) and pitch rate (q):

$$X = \begin{bmatrix} \phi & \theta & p & q \end{bmatrix} \tag{6.6}$$

6.2.5 Helicopter Equations of Motion

The 6-DoF equations of motion are given in (Dreier). The positional derivative is a function of body velocity (V_b) and the transformation from the earth frame to the body frame $\left(T_E^B\right)$:

$$P_e = \left[T_E^B \right]^T V_b \tag{6.7}$$

The angular acceleration of the inertial angles is a function of the current attitude and the body rates (ω):

$$\alpha_E = \begin{bmatrix} 1 & 0 & -\sin(\theta) \\ 1 & \cos(\phi) & \sin(\phi)\cos(\theta) \\ 0 & -\sin(\phi) & \cos(\phi)\cos(\theta) \end{bmatrix} \omega \tag{6.8}$$

The angular acceleration in the body frame is a function of the moments acting on the body (M) and the body inertia matrix (I_n):

$$\omega = I_n^{-1}\left[M - \omega \times (I_n \omega)\right] \tag{6.9}$$

The translational acceleration in the body frame is a function of the forces acting on the body (F) with M_b being a square matrix with the vehicle mass on the diagonal:

$$V = M_b^{-1}\left[F - \omega \times (mV)\right] \tag{6.10}$$

The only external force and moments that will be modeled are generated by gravity, the main rotor, and the tail rotor. As the helicopter is in hover it is assumed that there is no wind velocity and the forces induced by the fuselage are negligible.

The main rotor generates a thrust and introduces the following force and moment contributions to the kinematic equations:

$$F_{MR} = T_{TPP}^{HP}\begin{bmatrix}0 & 0 & -T_{MR}\end{bmatrix}^T \tag{6.11}$$

T_{TPP}^{HP} is the rotation matrix from the tip-path plane to the hub plane and is a function of lateral and longitudinal control. The moment generated by the main rotor is:

$$M_{MR} = T_{TPP}^{HP}\begin{bmatrix}0 & 0 & Q_{MR}\end{bmatrix}^T + \begin{bmatrix}x_{hub} & y_{hub} & z_{hub}\end{bmatrix}^T \times M_{MR} \tag{6.12}$$

Q_{MR} is the torque generated by the main rotor drag, x_{hub}, y_{hub}, and z_{hub} are the moment arms measured from the body center of mass to the rotor hub in the x, y and z axis directions. Necessary equations to compute rotor thrust are given by (Munzinger):

$$T_{MR} = (w_{blade} - v_i)\frac{\rho\Omega_{MR}R^2 aBc}{4} \tag{6.13}$$

Main rotor thrust given by equation (6.13) is a function of the velocity at the blade (w_{blade}), atmospheric density (ρ), main rotor rotational speed (Ω_{MR}), the number of blades (B), the airfoil lift-curve slope (a), the blade radius (R), the mean blade chord length (c) and the velocity generated by the blade (induced velocity v_i).

Induced velocity is found by:

$$v_i^2 = \sqrt{\left(\frac{\hat{v}^2}{2}\right)^2 + \left(\frac{T}{2\rho\pi R^2}\right)^2} - \frac{\hat{v}^2}{2} \tag{6.14}$$

With:

$$w_{blade} = w_r + \frac{2}{3}\Omega_{MR} R\left(\theta_{coll} - \frac{3}{4}\theta_{twist}\right) \tag{6.15}$$

$$w_r = w + \left(flap_{lat} + i_s\right)u - flap_{lon}v \tag{6.16}$$

$$\hat{v} = u^2 + v^2 + w^2(w_r - 2v_i) \tag{6.17}$$

Main rotor torque $\left(Q_{MR}\right)$ is the ratio of main rotor power $\left(P_{MR}\right)$ over the rotational speed $\left(\Omega_{MR}\right)$:

$$Q_{MR} = \frac{P_{MR}}{\Omega_{MR}} \tag{6.18}$$

$$P_{MR} = P_{pr} + P_i + P_{pa} + P_c \tag{6.19}$$

Because the helicopter is in hover, climb power (P_c) and fuselage parasite drag power (P_{pa}) can be neglected. This leaves only the blade profile power $\left(P_{pr}\right)$ and induced power $\left(P_i\right)$:

$$P_{pr} = \frac{\rho}{2}\frac{c_{D,0}bcR}{4}\Omega_{MR}((\Omega_{MR}R)^2 + 4.6(u^2 + v^2)) \tag{6.20}$$

Induced power is given by:

$$P_i = T_{MR}v_i \tag{6.21}$$

The equations shown above are the most relevant to what is discussed in the proceeding sections and have been implemented in MATLAB to provide a means for non-linear 6-DoF simulation of a helicopter. More in-depth modeling information can be found in (Munzinger).

6.2.6 Linearization

For continuous-time LQR the system needs to be in a linearized form:

$$X = AX + BU \tag{6.22}$$

Where A is the state transition matrix and U is the control matrix.

$$A = \frac{\partial X}{\partial X}, B = \frac{\partial X}{\partial U} \tag{6.23}$$

The equilibrium state (X_e) for the system described in hover is:

$$X_e = \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix}^T \tag{6.24}$$

The control positions at equilibrium (U_e) are:

$$U_e = \begin{bmatrix} \Theta_{col\_e} & \Theta_{lat\_e} & \Theta_{lon\_e} \end{bmatrix} \tag{6.25}$$

Control is dependent on the helicopter parameters such as mass, rotor radius, airfoil, and chord length. For a helicopter in hover the main rotor thrust must cancel the force of gravity. Control is also highly coupled, for example when changing the lateral control to cancel a rolling moment less thrust will be in the vertical direction and the collective must be increased to prevent the altitude from changing.

To simplify the trimming process a function that calculates the forces and moments acting on a designated helicopter at a given state using the non-linear equations has been created. This function is then used to generate a performance index that is minimized using available tools in MATLAB. This provides a means to compute the equilibrium control quickly for any set of conditions and helicopter specifications.

6.2.6.1 Main Rotor Thrust

When attempting to linearize thrust, the problem arises that the main rotor thrust is a function of the induced velocity and induced velocity is a function of thrust. An implicit relationship is born that is fourth order in the general case and has no simple algebraic solution (Dreier). To circumvent this problem linearized main rotor thrust $\left(T_{MRlin}\right)$ will be assumed to take the form:

$$T_{MRlin} = m_{linTMR}\Theta_{col} + b_{linTMR} \tag{6.26}$$

m_{linTMR} is the rate of change in thrust due to the collective (Θ_{col}) and is determined using small perturbation theory; here δ is the amount of perturbation:

$$m_{linTMR} = \frac{T_{MR}(\Theta_{col\_e} + \delta) - T_{MR}(\Theta_{col\_e} - \delta)}{2\delta} \tag{6.27}$$

6.2.6.2 Main Rotor Drag

Main rotor drag is also difficult to find an analytic solution for because it is a function of thrust. As with thrust the linearization is performed using small perturbation theory.

$$Q_{MRlin} = m_{linQMR}\Theta_{col} + b_{linQMR} \tag{6.28}$$

$$m_{linQMR} = \frac{Q_{MR}(\Theta_{col\_e} + \delta) - Q_{MR}(\Theta_{col\_e} - \delta)}{2\delta} \tag{6.29}$$

6.2.7 Flight Test

The platform used for testing is the T-Rex 700E with a fly-bar (control rotor). The ArduPilot 1.0 hardware is mounted in the front of the helicopter and all of the radio signals with the exception of the throttle-cutoff are being passed through the flight computer. These signals are then able to be manipulated by JARVIS and are output to the tail gyro that is included with the system. In this configuration the autopilot is able to control all effectors available while allowing the pilot to kill the power as a failsafe.

A windows-based laptop is used in conjunction with the MissionPlanner software to provide real-time telemetry. Sensor data, NAV states, and pilot input are recorded on the ArduPilot flash memory for reconstruction post-flight.

Figure 26: Helicopter: Ground Station and Vehicle

For testing the autopilot system the helicopter was flown manually by an experienced pilot. Once brought to altitude and in a stable hover the pilot activated the pilot-assist mode, if instability occurred control was changed back to manual and the vehicle was recovered.

Figure 27: Helicopter: In Flight

The flight control was first configured with a gain matrix produced using the LQR method described in the preceding sections, gains were then tuned by hand to achieve the results shown in Figure 28 and Figure 29. The time-window shown in the figures is during the period that the auto-pilot is active.

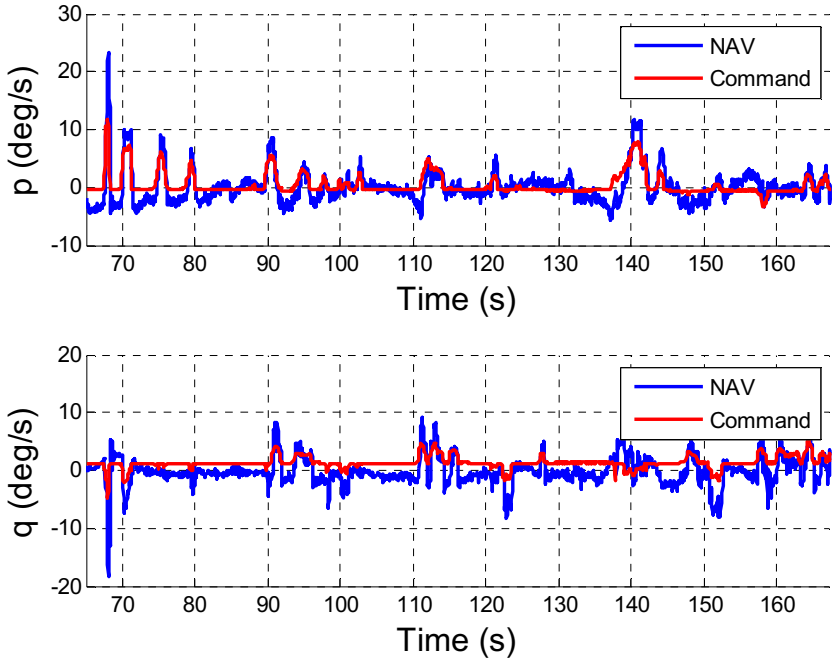

Figure 28: Helicopter Flight: pq Performance

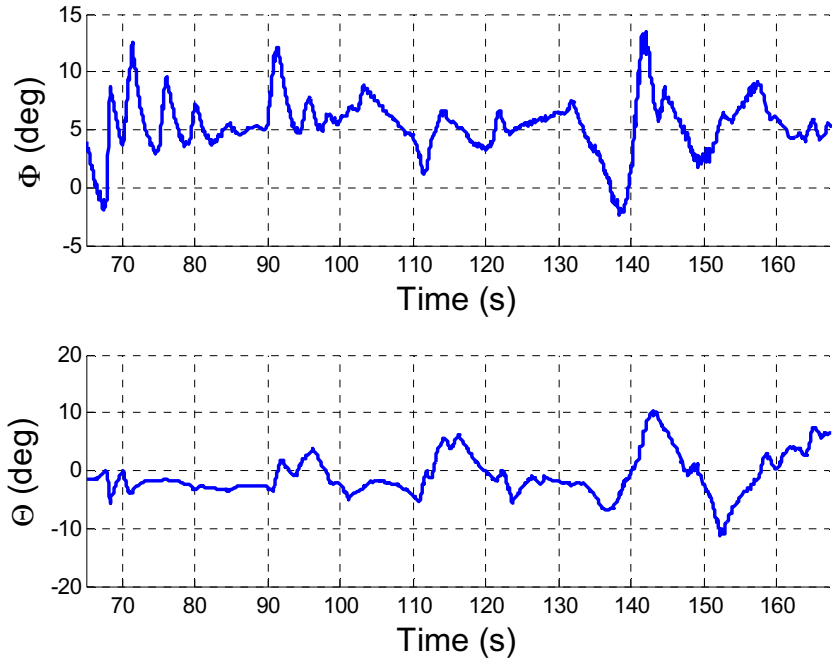

Figure 29: Helicopter Flight: □-□ Performance

Control was able to stabilize the rates and provide marginal attitude stabilization. Attitude control suffered from a steady state error that is due to an offset in the accelerometer readings. During these flight tests the sensors were automatically calibrated after system power-up. The accelerometer calibration process assumes that the vehicle is level and that gravity is positive down. Unfortunately the ground that the helicopter was resting on was not level and resulted in approximately a five degree offset in the roll channel and negative two degree offset in the pitch channel. With these errors, and without positional control, the vehicle was unable to maintain a hover state.

CHAPTER 7. FINAL CONCLUSIONS

7.1 Research Goals Revisited

The problem addressed by this research is the lack of a robust integrated architecture and library that can be used in developing unmanned vehicles, particularly for research purposes in a low-cost environment. The result of this research is a software architecture that is simple to follow and library code base that is nearly platform independent, in that it can operate on any hardware capable of compiling C++ code. The code created has been done so with readability and reusability in mind. JARVIS currently operates in the MatLab environment, on ArduPilot hardware, and on x86 PC's.

A library was created that provides necessary functionality in developing aerospace systems. This functionality includes coordinate transformations, unit conversions, matrix capability as well as linear algebra tools. Using the base architecture and library it was then demonstrated that GN&C algorithms could be easily taken from text and implemented in code for real-time use.

By abstracting the GN&C subsystems and pursuing an object-oriented approach the code is fully interchangeable. Entire subsystems can be replaced without affecting the rest of the system. This allows for rapid testing of new algorithms as well as comparison between algorithms. The added benefit to this method is that new code can be reused in later projects and provides the user more tools for development.

JARVIS is common across vehicle platforms only requiring modification to the configuration file to enable operation on a new platform. This eliminates duplication of code. When the same autopilot hardware is used the configuration file between vehicles is nearly identical. Two configuration file examples are provided in Appendix B.

7.2 Lessons Learned

"Models are not perfect." Inherently all engineers know, or should know, that this statement is true. Unfortunately hands-on experience is not always available and because GN&C development is most often done in simulation and on paper it is easy to be lulled into a false sense of security that because something works in simulation it should work on a physical vehicle. The modeling used to simulate the vehicles in this thesis was known to be very simplified and was not expected to produce the correct response. However, it was enlightening to experience the difference between the simulated result and the real-world result. The creation of JARVIS was to enable one to gain this experience so the work was very rewarding.

Hardware provides a number of limitations that may not be immediately considered. When developing a GN&C algorithm for real-time use it is not uncommon to spend a considerable amount of time optimizing the algorithm for speed. The limitation encountered during development of the navigation subsystem used here was the RAM available on the system. While the ArduPilot provided enough memory for the program and was able to cope with the processing requirements there was not enough system RAM for the program to operate. This resulted in the system locking up and it was not immediately apparent what was causing the issue.

Code development and validation is time consuming. While not a revolutionary thought this underscores the importance of writing the code to be readable and reusable. When the code is designed in parts and can be built upon itself, by use of classes and inheritance, and time spent on future development can be greatly reduced.

7.3 Future Work

In its current state JARVIS provides a powerful tool that can be used to create an unmanned vehicle. The current barrier to running more advanced algorithms is the ArduPilot platform and requires a move to more powerful autopilot hardware. The move to a different platform should be straight forward and only involve writing the low-level drivers to access sensors and output R/C signals.

This research was concerned with the development of JARVIS so as to facilitate the creation of unmanned vehicles. As such time spent actually developing and testing a fully-functional unmanned vehicle has been limited. Work on the platforms used in this research will continue with the final goal being that the vehicles are fully autonomous.

LIST OF REFERENCES

LIST OF REFERENCES

Agee, William S. *Triangular Decomposition of a Positive Definite Matrix Plus a Symmetric Dyad with Applications to Kalman Filtering*. Springfield: National Technical Information Service, 1972.

Centinello III, Frank J. "Analysis of the NED and ECEF Covariance Propagation for the Navigational Extended Kalman Filter." *International Astronautical Congress*. 2007.

Crassidis, John L. and John L. Junkins. *Optimal Estimation of Dynamic Systems*. Chapman & Hall/CRC, 2004.

Dreier, Mark E. *Introduction to Helicopter and Tiltrotor Flight Simulation*. AIAA, 2007.

Holt, Greg N.; D'Souza, Christopher. "Orion Absolute Navigation System Progress and Challenges." *AIAA GNC Conference*. Minneapolis, 2012. 17.

Munzinger, Christian. "Development of a Real-Time Flight Simulator for an Experimental Model Helicopter." 1998.

Thornton, C. L. and G. J. Bierman. *Gram-Schmidt Algorithms for Covariance Propagation*. Pasadena, 1975.

APPENDICES

Appendix A Library Function List

Math Enhanced	
Function	Purpose
saturate	Saturate variable X when above or below input values
sign	Return the sign of variable X 0 is assumed to be positive
round	Round a value to the nearest integer value

Array Class	
Function	Purpose
Array<T>	Construct an array with variable type T
Array<T>(m,n)	Construct an array with variable type T and size m by n
Initialize	Initialize an array
Clear	Clear allocated memory for an array variable
Resize	Resize an array variable By default old values are lost, if the array is made to be dynamic the values are kept
ExpandSize	Increase the size of an array
ContractSize	Reduce the size of an array
makeDynamic	Allows for dynamic reallocation of the array size. Memory is kept when resizing dimensions
makeStatic	Fixes the size of the matrix If resized memory is lost
isInitialized	Return the initialization status of the variable
nRows	Return the number of rows in the array
nCols	Return the number of columns in the array
(m,n)	Access element (m,n) in the array
getData(m,n)	Access element (m,n) in the array
(Index)	Treat the array as a column vector and access element at (Index)
getRow(R,m)	Copy row m to vector R
getCol(C,n)	Copy row n to vector C
assignRow(m, B, mb)	Copy row mb from matrix B to row m
assignCol(n, B, nb)	Copy column nb from matrix B to column n
=	Assign matrix A = B
setData(m,n, Val)	Set the value of (m,n)
copy	Copy matrix B to current matrix

transpose	Transpose the current matrix
==	Compare two matrices, return true if A==B
!=	Compare two matricies, return true if A != B

Matrix Class	
Function	Purpose
Matrix<T>	Construct a matrix variable of type T
Matrix<T>(m,n)	Construct a matrix variable of type T and size m-by-n Initial data is set to 0
Clear	De-allocate memory used for the matrix
Resize	Resize a matrix Data is not maintained
Matrix_Initialized	Returns initialization status of matrix
nrows	Returns the number of rows
ncols	Returns the number of columns
(m,n)	Access to data located at (m,n)
getData(m,n)	Return data located at (m,n)
setData(m,n, Val)	Set the value of (m,n)
(index)	Access data located at index treating the matrix as a single column
trace	Return the trace of the matrix
getRow(R,m)	Copy row m to vector R
getCol(C,n)	Copy row n to vector C
assignRow(m,B,mb)	Copy row mb from matrix B to row m
assignCol(n,B,nb)	Copy column nb from matrix B to column n
=	Assign matrix A = B
Diag	Create a diagonal matrix using values from an input vector
Diag	Create a diagonal matrix using a string containing the diagonal values
copy	Copy matrix B to current matrix
transpose	Transpose the current matrix
zeros	Set all values in a matrix to 0
ones	Set all values in a matrix to 1
nans	Set all values in a matrix to NaN
eye	Set a matrix to identity
==	Compare two matrices, return true if A==B
!=	Compare two matricies, return true if A != B
prints	Prints the matrix to the terminal

Linear Algebra	
Function	Purpose
mmult	Multiply two matrices
mscale	Store a scaled matrix
madd	Add or subtract matrices
mQR	Perform QR decomposition on matrix A
mLU	Perform LU decomposition on matrix A
LUSolve	Use LU decomposition to solve Ax=b when b is a vector
LUSolve	Solve Ax=b when b is a vector and LU decomposition has already been performed
mLUSolve	Solve AX=B when B is a matrix
mdet3x3	Return the determinant of a 3-by-3 matrix
mdet	Return the determinant of a n-by-n matrix
minv3x3	Invert a 3-by-3 matrix
minv	Invert a n-by-n matrix
dot	Return the dot product of two vectors
cross3	Compute the cross product of two vectors
vnorm	Return the magnitude of a vector
vunit	Unitize a vector
orthagonalize	Orthogonalize a matrix
UDU	Decompose a such that A = U*D*U', D is stored as a vector

Conversions	
Function	Purpose
quat2DCM	Convert from attitude quaternion to attitude DCM
quat2axis_angle	Convert from attitude quaternion to Euler axis-angle representation
dcm2quat	Convert from DCM to attitude quaternion
dcm2angle	Convert from DCM to Euler angles following a specified rotation sequence
angle2dcm	Convert from Euler angles to DCM using a specified rotation sequence
angle2quat	Convert from Euler angles to attitude quaternion
geodetic2ecef	Convert position from geodetic frame to ECEF frame
geodetic2ned	Convert position from geodetic position to topodetic NED frame
rad2deg	Convert angle from radians to degrees
deg2rad	Convert angle from degrees to radians
ft2m	Convert from feet to meters
m2ft	Convert from meters to feet
C2F	Convert from Celsius to Fahrenheit
C2K	Convert from Celsius to Kelvin
F2C	Convert from Fahrenheit to Celsius
F2K	Convert from Fahrenheit to Kelvin
Psf2Pa	Convert from pounds per feet squared to Pascal
Pa2Psf	Convert from Pascal to pounds per feet squared
juliandate	Convert Gregorian date to Julian date
juliandate2GMST	Convert Julian date to Greenwich mean sidereal time
juliandate2GAST	Convert Julian date to Greenwich Apparent Sidereal Time

Estimators	
Function	Purpose
AHRS_MEKF	Generic MEKF attitude filter used to estimate attitude and angular rates
INRTL_EKF	EKF used to estimate inertial position

Filters	
Function	Purpose
UDU_EKF	Generic Kalman filter using the UDU formulation, Rank1_Update, and Modified Grahm-Schmidt Orthogonalization.

Constants	
Constant	Purpose
pi	Provide common definition of pi: 3.14159265
Gravitational_Accel	Provide a common definition of gravity: 32.1740486 ft/s

TimeSeries Class	
Function	Purpose
TimeSeries<T>	Declare TimeSeries variable TS
addNewData(Time, Data)	Add new data point at a specified time
getNumPoints()	Return the number of data points available
getTime(Index)	Return the time using an input index
getIndexFromTime(Time)	Return the index that the input time occurs
getDataFromTime(Time)	Return the data at a particular time

Attitude	
Function	Purpose
DCMKin	Calculate the time derivative of the direction cosine matrix
quatmult	Multiply two attitude quaternions
quatkin	Calculate the time derivative of the attitude quaternion

Appendix B Example Vehicle Configuration Files

B.1 Helicopter Configuration file:

```
/*=============================================================================
==============
HW Configuratio
=============================================================================
=============*
// Hardware Type,
#define Selected_Hardware   Hardware_APM1

// Memory Storage
#define Selected_DataFlash   DataFlash_APM1

// Sensor Selection
#define Selected_GPS         GPS

// Hardware Options
#define CompassEnable                false                  // Flag to enable sensor
#define CompassTiltCompensation      false                  // Flag tilt-compensate
compass
#define AccelerometerEnable          true                   // Flag to enable sensor
#define GyroEnable           true                  // Flag to enable sensor
#define BaroEnable           false                 // Flag to enable sensor
#define GPSEnable            true                  // Flag to enable sensor

// Mounting APM1, RH 0:0:0 X - Pins Forward Z-Down (Sensors Up)
#define HW_Yaw                       PI                     // Yaw angle of hardware
mounting
#define HW_Pitch             0.0                   // Pitch angle of hardware mounting
#define HW_Roll                      -PI/2                  // Roll angle of hardware
mounting

#define Compass_Yaw                  PI/2                   // Yaw angle of compass
mounting
#define Compass_Pitch                0.0                    // Pitch angle of compass
mounting
#define Compass_Roll         -PI/2                 // Yaw angle of compass mounting

// RC Inputs
```

```
#define RC_In_Reversing             "[1, 1, 1, 1, 0, 1, 0, 0]"          // Reverse signal from
input pins
#define RC_In_nDirs          "[2, 2, 1, 2, 2, 2, 1, 2]"          // Number of directions
input travels
#define RC_In_Joy_Pins              "[5, 3, 1, 0]"              // Joystick inputs
#define Pin_APMode          6                       // Pin  for switching piloting mode

// RC Outputs
#define RC_Out_Reversing    "[0, 0, 0, 1, 0, 1, 0, 1]" // Reverse RC direction
#define RC_Out_nDirs                "[2, 2, 2, 2, 2, 2, 1, 2]"          // Number of
directions that RC can travel
// RC maximum PW
#define RC_Out_End_Max              "[2100, 2100, 2100, 2100, 2100, 2100, 2100, 2100]"

// RC minimum PW
#define RC_Out_End_Min              "[900, 900, 900,        900, 900, 900, 900, 900]"
// RC trim point
#define RC_Out_Sub_Trim             "[1500, 1500, 1500, 1500, 1500, 1500, 1500, 1500]"


#define nEffectors          4                       // Number of control
effectors/servos
#define Effector_Out_Pins           "[0, 5, 1, 3]"              // Output pins for the
effectors'

/*=================================================================
==============
COM Parameters
=================================================================
============*/
#define Selected_GCS                GCS_MAVLink_MP
#define GCS_Port            Serial3                 // Wireless Radio
#define GCS_Baud            57600

/*=================================================================
==============
PILOT Parameters
=================================================================
============*/
#define Selected_PILOT              Pilot_RC_Attitude_Assist

/*=================================================================
==============
```

```
Guidance Parameters
========================================================================
============*/
#define Selected_GUID               Guidance_Linear             // Select guidance
type

// Guidance type-specific variables:

#define GUID_nStates        3                   // Number states [PHI THETA PSI]'
#define GUID_nCommands      4                   // Number of states to command

// Map desired state
#define GUID_State_Desired  State_Desired(0)    = 0.0;\
                            State_Desired(1)    = 0.0;\
                            State_Desired(2)    = 0.0;\

// Map actual state
#define GUID_State_Actual   State_Actual(0)     = NAV->E_YPR(2);\
                            State_Actual(1)     = NAV->E_YPR(1);\
                            State_Actual(2)     = NAV->E_YPR(0);

// Guidance gain matrix: State_Commanded = Pilot_Command +
K_guid*(Position_Desired - Positon_Actual)
// K_guid*(Position_Desired - Positon_Actual) modifies the pilot command (rates) to
include positional information
#define GUID_Linear_K               "[ 0.0, 0.0, 0.0;\
                                0.05, 0.0, 0.0;\
                                0.0, 0.05, 0.0;\
                                0.0, 0.0, 0.0]"

/*====================================================================
==============
Control Parameters
========================================================================
============*/
#define Selected_CNTRL              Control_StateSpace          // Select controller
type

// Croller type-specific variables
#define CNTRL_nControls             4                   // Number of control
variables
#define CNTRL_nStates               3                   // Number of input states [p,
q, r]
```

```
// Maximum rate of change of control input (Udot)
#define CNTRL_Udot_Saturation       "[10.0, 5.0, 5.0, 10.0]"


// Map from control command to actuator output: Effector_Out = Effector_Map*U

#define CNTRL_Effector_Map          "[  1.0, -0.5,  0.5, 0.0;\
                                     1.0,  0.5, 0.5, 0.0;\
                                    1.0,  0.0, -1.0, 0.0;\
                                    0.0,  0.0,  0.0, 1.0 ]"

// Map desired state
#define CNTRL_State_Desired         State_Desired(0)       = GUID->Command(1);\
                                    State_Desired(1)       = GUID->Command(2);\
                                    State_Desired(2)       = GUID->Command(3);

// Map actual state
#define CNTRL_State_Actual  State_Actual(0)        = NAV->w_B(0);\
                                    State_Actual(1)        = NAV->w_B(1);\
                                    State_Actual(2)        = NAV->w_B(2);

// Control gain matrix: U = U_e + K_control*(State_Commanded - State_Actual)
#define CNTRL_Linear_K              "[  0.00000,   0.00000,  0.00000;\
                                     0.1000,   0.00000,  0.00000;\
                                     0.00000,   0.15000,  0.00000;\
                                     0.00000,   0.00000,  0.00000 ]"

/*=====================================================================
==============
NAV Parameters
=======================================================================
============*/
//-----------------------------------Attitude-------------------------------------------------------*/
#define         Selected_NAV_Attitude               Navigation_Attitude_MEKF
// NAV attitude type-specific variables
#define NAV_Attitude_MEKF_dt            0.02                    // Expected dt
#define NAV_Attitude_MEKF_Sigma_a       0.1                     // Accelerometer
variance
#define NAV_Attitude_MEKF_Sigma_g       (1.0*PI/180.0)                  // Gyro
variance
#define NAV_Attitude_MEKF_Sigma_gb      ((1.0/60.0)*PI/180.0)  // Gyro bias variance
#define NAV_Attitude_MEKF_Sigma_c       20.0*PI/180.0           // Compass variance
```

```
#define NAV_Attitude_MEKF_Gyro_Tau      343.78                      // Gyro time
constant
#define NAV_Attitude_MEKF_Residual_Threshold          3       // multiple of the
                                                              //measurement uncertainty
                                                              used //in rejecting bad
                                                              measurements
#define NAV_Attitude_MEKF_Underweighting_Threshold  200

#define NAV_Attitude_MEKF_Underweighting_Coeff              0

//-----------------------------------Inertial-----------------------------------------------------------*/
#define         Selected_NAV_INRTL                  Navigation_Inertial_EKF

// NAV inertial type-specific variables
#define NAV_Inertial_EKF_Sigma_a            NAV_Attitude_MEKF_Sigma_a
#define NAV_Inertial_EKF_Sigma_ab               0.001
#define NAV_Inertial_EKF_Accel_Tau              3600.0
#define NAV_Inertial_EKF_Sigma_baro             1.0
#define NAV_Inertial_EKF_Sigma_GPS_Alt          0.5
#define NAV_Inertial_EKF_Sigma_GPS_Vmag         0.01
#define NAV_Inertial_EKF_Residual_Threshold     3

#define NAV_Inertial_EKF_Underweighting_Threshold      200

#define NAV_Inertial_EKF_Underweighting_Coeff          0

/*=================================================================
==============
LOGGING
=================================================================
============*/
#define Selected_LOG                        Log_APM
#define LogEnable                           true
```

B.2 Gimbaled Tri-Ducted Fan Configuration file:

```
/*=================================================================
==============
HW Configuration
=================================================================
=============*
// Hardware Type,
#define Selected_Hardware   Hardware_APM1

// Memory Storage
#define Selected_DataFlash   DataFlash_APM1

// Sensor Selection
#define Selected_GPS         GPS

// Hardware Options
#define CompassEnable                true                    // Flag to enable sensor
#define CompassTiltCompensation   false                    // Flag tilt-compensate
compass
#define AccelerometerEnable          true                  // Flag to enable sensor
#define GyroEnable           true                  // Flag to enable sensor
#define BaroEnable           false                 // Flag to enable sensor
#define GPSEnable            true                  // Flag to enable sensor

// Mounting APM1, RH 0:0:0 X - Pins Forward Z-Down (Sensors Up)
#define HW_Yaw                       0.0                     // Yaw angle of hardware
mounting
#define HW_Pitch             0.0                   // Pitch angle of hardware mounting
#define HW_Roll                      0.0                     // Roll angle of hardware
mounting

#define Compass_Yaw                  PI/2                    // Yaw angle of compass
mounting
#define Compass_Pitch                0.0                     // Pitch angle of compass
mounting
#define Compass_Roll         0.0                   // Yaw angle of compass mounting

// RC Inputs
#define RC_In_Reversing              "[0, 1, 1, 0, 0, 0, 1, 0]"          // Reverse signal from
input pins
#define RC_In_nDirs          "[2, 2, 1, 2, 2, 2, 1, 2]"          // Number of directions
input travels
```

```
#define RC_In_Joy_Pins              "[2, 3, 1, 0]"          // Joystick inputs
#define Pin_APMode          6                       // Pin  for switching piloting mode

// RC Outputs
#define RC_Out_Reversing    "[0, 1, 0, 0, 0, 0, 0, 0]" // Reverse RC direction
#define RC_Out_nDirs                "[2, 2, 1, 2, 2, 2, 1, 2]"          // Number of
directions that RC can travel
// RC maximum PW
#define RC_Out_End_Max              "[2100, 1735, 2100, 2100, 1689, 2100, 2100, 1689]"

// RC minimum PW
#define RC_Out_End_Min              "[900, 1310, 900,       900, 1265, 900, 900, 1265]"
// RC trim point
#define RC_Out_Sub_Trim             "[1500, 1537, 1500, 1500, 1520, 1500, 1500, 1436]"


#define nEffectors          6                       // Number of control
effectors/servos
#define Effector_Out_Pins           "[6, 3, 0, 7, 4, 1]"                // Output pins for the
effectors'

/*=================================================================
==============
COM Parameters
=================================================================
============*/
#define Selected_GCS                GCS_MAVLink_MP
#define GCS_Port            Serial3                 // Wireless Radio
#define GCS_Baud            57600
#define GCS_nParameters             9                       // # of COM-configurable
params

/*=================================================================
==============
PILOT Parameters
=================================================================
============*/
#define Selected_PILOT              Pilot_RC_Attitude_Assist

/*=================================================================
==============
Guidance Parameters
```

```
================================================================================*/
#define Selected_GUID                Guidance_Linear              // Select guidance type

// Guidance type-specific variables:

#define GUID_nStates          3                    // Number states [PHI THETA PSI]'
#define GUID_nCommands        4                    // Number of states to command

// Map desired state
#define GUID_State_Desired    State_Desired(0)     = 0.0;\
                              State_Desired(1)     = 0.0;\
                              State_Desired(2)     = 0.0;\

// Map actual state
#define GUID_State_Actual     State_Actual(0)      = NAV->E_YPR(2);\
                              State_Actual(1)      = NAV->E_YPR(1);\
                              State_Actual(2)      = NAV->E_YPR(0);

// Guidance gain matrix: State_Commanded = Pilot_Command + K_guid*(Position_Desired - Positon_Actual)
// K_guid*(Position_Desired - Positon_Actual) modifies the pilot command (rates) to include positional information
#define GUID_Linear_K                "[ 0.0, 0.0, 0.0;\
                                    0.0, 0.0, 0.0;\
                                    0.0, 0.0, 0.0;\
                                    0.0, 0.0, 0.0]"

/*================================================================================
==============
Control Parameters
================================================================================
============*/
#define Selected_CNTRL               Control_PID          // Select controller type

// Controller-type specific variables
#define CNTRL_nControls              4                    // Number of control variables#define CNTRL_nStates                4                    // Number of input states

// Maximum rate of change of control input (Udot)
#define CNTRL_Udot_Saturation        "[10.0, 5.0, 5.0, 10.0]"
```

```
// Map from control command to actuator output: Effector_Out = Effector_Map*U

#define CNTRL_Effector_Map          "[ 1.0, -0.5,  0.0,         0.00;\
                                    1.0,  0.0,  -1.0, 0.00;\
                                    1.0,  0.5,  0.0,          0.0;\
                                    0.0,  0.0,  0.0,          0.5;\
                                    0.0,  0.0,  0.0,          0.0;\
                                    0.0,  0.0,  0.0,          -0.5 ]"

// Set Equillibrium Control
#define CNTRL_U_e             U_e(0) = 0.5;\
                              U_e(1) = 0.0;\
                              U_e(2) = 0.0;\
                              U_e(3) = 0.0;\

// Map desired state
#define CNTRL_State_Desired         State_Desired = GUID->Command;

// Map actual state
#define CNTRL_State_Actual  State_Actual(0)       = 0.5;\
                            State_Actual(1)       = NAV->E_YPR(2);\
                            State_Actual(2)       = NAV->E_YPR(1);\
                            State_Actual(3)       = NAV->w_B(2);\

// Control gains
#define CNTRL_Kp            "[0.500;        0.005;  0.005;  0.000] "
#define CNTRL_Ki            "[0.000;        0.000;  0.000;  0.0000] "
#define CNTRL_Kd            "[0.000;        0.005;  0.005;  0.5000] "

/*=====================================================================
==============
NAV Parameters
=====================================================================
============*/
//----------------------------------Attitude-----------------------------------------------------*/
#define         Selected_NAV_Attitude               Navigation_Attitude_MEKF
// NAV attitude type-specific variables
#define NAV_Attitude_MEKF_dt            0.02                    // Expected dt
#define NAV_Attitude_MEKF_Sigma_a       0.1                     // Accelerometer
variance
#define NAV_Attitude_MEKF_Sigma_g       (1.0*PI/180.0)                  // Gyro
variance
```

```
#define NAV_Attitude_MEKF_Sigma_gb      ((1.0/60.0)*PI/180.0)  // Gyro bias variance
#define NAV_Attitude_MEKF_Sigma_c       20.0*PI/180.0          // Compass variance
#define NAV_Attitude_MEKF_Gyro_Tau      343.78                 // Gyro time
constant
#define NAV_Attitude_MEKF_Residual_Threshold          3        // multiple of the
                                                               //measurement uncertainty
                                                               used //in rejecting bad
                                                               measurements
#define NAV_Attitude_MEKF_Underweighting_Threshold    200

#define NAV_Attitude_MEKF_Underweighting_Coeff        0

//-----------------------------------Inertial-----------------------------------------------------*/
#define         Selected_NAV_INRTL              Navigation_Inertial_EKF

// NAV inertial type-specific variables
#define NAV_Inertial_EKF_Sigma_a          NAV_Attitude_MEKF_Sigma_a
#define NAV_Inertial_EKF_Sigma_ab         0.001
#define NAV_Inertial_EKF_Accel_Tau        3600.0
#define NAV_Inertial_EKF_Sigma_baro       1.0
#define NAV_Inertial_EKF_Sigma_GPS_Alt    0.5
#define NAV_Inertial_EKF_Sigma_GPS_Vmag   0.01
#define NAV_Inertial_EKF_Residual_Threshold   3

#define NAV_Inertial_EKF_Underweighting_Threshold   200

#define NAV_Inertial_EKF_Underweighting_Coeff       0

/*=================================================================
==============
LOGGING
=================================================================
============*/
#define Selected_LOG                    Log_APM
#define LogEnable                       true
```